Arm® Architecture Reference Manual

Armv8, for Armv8-A architecture profile

Beta

arm

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
ARM DDI 0487F.a (ID021920)

Arm Architecture Reference Manual
Armv8, for Armv8-A architecture profile

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.

Release Information

The following releases of this document have been made.

Release history

Date	Issue	Confidentiality	Change
30 April 2013	A.a-1	Confidential-Beta Draft	Beta draft of first issue, limited circulation
12 June 2013	A.a-2	Confidential-Beta Draft	Second beta draft of first issue, limited circulation
04 September 2013	A.a	Non-Confidential Beta	Beta release
24 December 2013	A.b	Non-Confidential Beta	Second beta release
18 July 2014	A.c	Non-Confidential Beta	Third beta release
09 October 2014	A.d	Non-Confidential Beta	Fourth beta release
17 December 2014	A.e	Non-Confidential Beta	Fifth beta release
25 March 2015	A.f	Non-Confidential Beta	Sixth beta release
10 July 2015	A.g	Non-Confidential Beta	Seventh beta release
30 September 2015	A.h	Non-Confidential Beta	Eighth beta release
28 January 2016	A.i	Non-Confidential Beta	Ninth beta release
03 June 2016	A.j	Non-Confidential EAC	EAC release
30 September 2016	A.k	Non-Confidential Armv8.0 EAC	Updated EAC release
31 March 2017	B.a	Non-Confidential Armv8.1 EAC, v8.2 Beta	Initial release incorporating Armv8.1 and Armv8.2
26 September 2017	B.b	Non-Confidential Armv8.2 EAC	Initial Armv8.2 EAC release, incorporating SPE
20 December 2017	C.a	Non-Confidential Armv8.3 EAC	Initial Armv8.3 EAC release
31 October 2018	D.a	Non-Confidential Armv8.4 EAC	Initial Armv8.4 EAC release
29 April 2019	D.b	Non-Confidential Armv8.4 EAC	Updated Armv8.4 EAC release incorporating accessibility changes
05 July 2019	E.a	Non-Confidential Armv8.5 EAC	Initial Armv8.5 EAC release
21 February 2020	F.a	Non-Confidential Armv8.6 Beta	Initial Armv8.6 Beta release

Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. **No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.**

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED “AS IS”. ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms. This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. You must follow the Arm's trademark usage guidelines
http://www.arm.com/company/policies/trademarks.

Copyright © 2013-2020 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.
110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

In this document, where the term Arm is used to refer to the company it means "Arm or any of its subsidiaries as appropriate".

Note

- The term Arm can refer to versions of the Arm architecture, for example Armv8 refers to version 8 of the Arm architecture. The context makes it clear when the term is used in this way.
- This document describes only the Armv8-A architecture profile. For the behaviors required by the previous version of this architecture profile, ARMv7-A, see the *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*.

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Product Status

The information in this document is for a Beta product, that is a product under development.

Web Address

`http://www.arm.com`

Limitations of this issue

This issue of the Armv8 Architecture Reference Manual contains many improvements and corrections. Validation of this document has identified the following issues that Arm will address in future issues:

- *PE state on reset to AArch64 state* on page D1-2336 and *PE state on reset into AArch32 state* on page G1-5761 require further update. Since the reset information is present in the register descriptions, this does not affect the quality status of the release.
- Appendix K13 *Arm Pseudocode Definition* requires further review and update. Since this appendix is informative, rather than being part of the architecture specification, this does not affect the quality status of this release.
- For a list of the known issues in the System register and instruction XML content, please refer to the Release Notes on https://developer.arm.com/architectures/cpu-architecture/a-profile/exploration-tools

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Contents
Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile

Preface
About this manual xviii
Using this manual xx
Conventions xxvi
Additional reading xxviii
Feedback xxx

Part A **Armv8 Architecture Introduction and Overview**

Chapter A1 **Introduction to the Armv8 Architecture**
A1.1 About the Arm architecture A1-34
A1.2 Architecture profiles A1-36
A1.3 Armv8 architectural concepts A1-37
A1.4 Supported data types A1-40
A1.5 Advanced SIMD and floating-point support A1-52
A1.6 The Arm memory model A1-59

Chapter A2 **Armv8-A Architecture Extensions**
A2.1 Armv8 architecture extensions A2-62
A2.2 Architectural features within Armv8.0 architecture A2-65
A2.3 The Armv8 Cryptographic Extension A2-68
A2.4 The Armv8.1 architecture extension A2-69
A2.5 The Armv8.2 architecture extension A2-72
A2.6 The Armv8.3 architecture extension A2-81
A2.7 The Armv8.4 architecture extension A2-85
A2.8 The Armv8.5 architecture extension A2-90

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A2.9 The Armv8.6 architecture extension A2-94
A2.10 The Reliability, Availability, and Serviceability Extension (RAS Extension) A2-97
A2.11 The Statistical Profiling Extension (SPE) A2-98
A2.12 The Scalable Vector Extension (SVE) A2-99
A2.13 The Activity Monitors Extension A2-100
A2.14 The Memory Partitioning and Monitoring (MPAM) Extension A2-101

Part B The AArch64 Application Level Architecture

Chapter B1 The AArch64 Application Level Programmers' Model
B1.1 About the Application level programmers' model B1-106
B1.2 Registers in AArch64 Execution state B1-107
B1.3 Software control features and EL0 ..... B1-112

Chapter B2 The AArch64 Application Level Memory Model
B2.1 About the Arm memory model B2-116
B2.2 Atomicity in the Arm architecture B2-118
B2.3 Definition of the Armv8 memory model B2-123
B2.4 Caches and memory hierarchy B2-143
B2.5 Alignment support B2-148
B2.6 Endian support B2-150
B2.7 Memory types and attributes B2-153
B2.8 Mismatched memory attributes B2-164
B2.9 Synchronization and semaphores B2-167

Part C The AArch64 Instruction Set

Chapter C1 The A64 Instruction Set
C1.1 About the A64 instruction set C1-182
C1.2 Structure of the A64 assembler language C1-183
C1.3 Address generation C1-189
C1.4 Instruction aliases C1-192

Chapter C2 About the A64 Instruction Descriptions
C2.1 Understanding the A64 instruction descriptions C2-194
C2.2 General information about the A64 instruction descriptions C2-197

Chapter C3 A64 Instruction Set Overview
C3.1 Branches, Exception generating, and System instructions C3-202
C3.2 Loads and stores C3-209
C3.3 Data processing - immediate C3-226
C3.4 Data processing - register C3-231
C3.5 Data processing - SIMD and floating-point C3-239

Chapter C4 A64 Instruction Set Encoding
C4.1 A64 instruction set encoding C4-268

Chapter C5 The A64 System Instruction Class
C5.1 The System instruction class encoding space C5-376
C5.2 Special-purpose registers C5-389
C5.3 A64 System instructions for cache maintenance C5-475
C5.4 A64 System instructions for address translation C5-536
C5.5 A64 System instructions for TLB maintenance C5-561
C5.6 A64 System instructions for prediction restriction C5-760

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter C6 **A64 Base Instruction Descriptions**
C6.1 About the A64 base instructions C6-772
C6.2 Alphabetical list of A64 base instructions C6-775

Chapter C7 **A64 Advanced SIMD and Floating-point Instruction Descriptions**
C7.1 About the A64 SIMD and floating-point instructions C7-1398
C7.2 Alphabetical list of A64 Advanced SIMD and floating-point instructions C7-1400

Part D The AArch64 System Level Architecture

Chapter D1 **The AArch64 System Level Programmers' Model**
D1.1 Exception levels D1-2314
D1.2 Exception terminology D1-2315
D1.3 Execution state D1-2317
D1.4 Security state D1-2318
D1.5 Virtualization D1-2320
D1.6 Registers for instruction processing and exception handling D1-2323
D1.7 Process state, PSTATE D1-2330
D1.8 Program counter and stack pointer alignment D1-2333
D1.9 Reset D1-2335
D1.10 Exception entry D1-2339
D1.11 Exception return D1-2350
D1.12 Synchronous exception types, routing and priorities D1-2354
D1.13 Asynchronous exception types, routing, masking and priorities D1-2363
D1.14 Configurable instruction enables and disables, and trap controls D1-2373
D1.15 System calls D1-2396
D1.16 Mechanisms for entering a low-power state D1-2397
D1.17 Self-hosted debug D1-2403
D1.18 Event monitors D1-2405
D1.19 Interprocessing D1-2406
D1.20 The effect of implementation choices on the programmers' model D1-2419

Chapter D2 **AArch64 Self-hosted Debug**
D2.1 About self-hosted debug D2-2424
D2.2 The debug exception enable controls D2-2428
D2.3 Routing debug exceptions D2-2429
D2.4 Enabling debug exceptions from the current Exception level D2-2431
D2.5 The effect of powerdown on debug exceptions D2-2433
D2.6 Summary of the routing and enabling of debug exceptions D2-2434
D2.7 Pseudocode description of debug exceptions D2-2436
D2.8 Breakpoint Instruction exceptions D2-2437
D2.9 Breakpoint exceptions D2-2439
D2.10 Watchpoint exceptions D2-2457
D2.11 Vector Catch exceptions D2-2471
D2.12 Software Step exceptions D2-2472
D2.13 Synchronization and debug exceptions D2-2485

Chapter D3 **AArch64 Self-hosted Trace**
D3.1 About self-hosted trace D3-2488
D3.2 Prohibited regions in self-hosted trace D3-2489
D3.3 Self-hosted trace timestamps D3-2491
D3.4 Synchronization in self-hosted trace D3-2493

Chapter D4 **The AArch64 System Level Memory Model**
D4.1 About the memory system architecture D4-2496
D4.2 Address space D4-2497
D4.3 Mixed-endian support D4-2498

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D4.4 Cache support D4-2499
D4.5 External aborts D4-2524
D4.6 Memory barrier instructions D4-2526
D4.7 Pseudocode description of general memory System instructions D4-2527

Chapter D5 The AArch64 Virtual Memory System Architecture
D5.1 About the Virtual Memory System Architecture (VMSA) D5-2532
D5.2 The VMSAv8-64 address translation system D5-2540
D5.3 VMSAv8-64 translation table format descriptors D5-2593
D5.4 Memory access control D5-2606
D5.5 Memory region attributes D5-2628
D5.6 Virtualization Host Extensions D5-2638
D5.7 Nested virtualization D5-2644
D5.8 VMSAv8-64 memory aborts D5-2652
D5.9 Translation Lookaside Buffers (TLBs) D5-2662
D5.10 TLB maintenance requirements and the TLB maintenance instructions D5-2668
D5.11 Caches in a VMSAv8-64 implementation D5-2685

Chapter D6 Armv8.5 Memory Tagging Extension
D6.1 Introduction D6-2690
D6.2 Allocation Tags D6-2691
D6.3 Tag checking D6-2692
D6.4 Tagged and Untagged Addresses D6-2693
D6.5 PE access to Allocation Tags D6-2694
D6.6 Enabling the Memory Tagging Extension D6-2695
D6.7 PE handling of Tag Check Fault D6-2696
D6.8 PE generation of Tag Checked and Tag Unchecked accesses D6-2697

Chapter D7 The Performance Monitors Extension
D7.1 About the Performance Monitors D7-2700
D7.2 Accuracy of the Performance Monitors D7-2703
D7.3 Behavior on overflow D7-2705
D7.4 Attributability D7-2707
D7.5 Prohibiting event counting D7-2709
D7.6 Multithreaded implementations D7-2710
D7.7 Event filtering D7-2711
D7.8 Performance Monitors and Debug state D7-2713
D7.9 Counter enables D7-2714
D7.10 Counter access D7-2715
D7.11 PMU events and event numbers D7-2716
D7.12 Performance Monitors Extension registers D7-2772

Chapter D8 The Activity Monitors Extension
D8.1 About the Activity Monitors Extension D8-2774
D8.2 Properties and behavior of the activity monitors D8-2775
D8.3 AMU events and event numbers D8-2777

Chapter D9 The Statistical Profiling Extension
D9.1 About the Statistical Profiling Extension D9-2780
D9.2 Defining the sample population D9-2782
D9.3 Controlling when an operation is sampled D9-2784
D9.4 Enabling profiling D9-2787
D9.5 Filtering sample records D9-2789
D9.6 The profiling data D9-2790
D9.7 The Profiling Buffer D9-2799
D9.8 Profiling Buffer management D9-2803
D9.9 Synchronization and Statistical Profiling D9-2807

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter D10 **Statistical Profiling Extension Sample Record Specification**
D10.1 About the Statistical Profiling Extension Sample Records D10-2810
D10.2 Alphabetical list of Statistical Profiling Extension packets D10-2813

Chapter D11 **The Generic Timer in AArch64 state**
D11.1 About the Generic Timer D11-2838
D11.2 The AArch64 view of the Generic Timer D11-2842

Chapter D12 **AArch64 System Register Encoding**
D12.1 The System register encoding space D12-2850
D12.2 op0==0b10, Moves to and from debug and trace System registers D12-2851
D12.3 op0==0b11, Moves to and from non-debug System registers, Special-purpose registers D12-2853

Chapter D13 **AArch64 System Register Descriptions**
D13.1 About the AArch64 System registers D13-2868
D13.2 General system control registers D13-2876
D13.3 Debug registers D13-3560
D13.4 Performance Monitors registers D13-3663
D13.5 Activity Monitors registers D13-3724
D13.6 Statistical Profiling Extension registers D13-3755
D13.7 RAS registers D13-3801
D13.8 Generic Timer registers D13-3839

Part E **The AArch32 Application Level Architecture**

Chapter E1 **The AArch32 Application Level Programmers' Model**
E1.1 About the Application level programmers' model E1-3948
E1.2 The Application level programmers' model in AArch32 state E1-3949
E1.3 Advanced SIMD and floating-point instructions E1-3960
E1.4 About the AArch32 System register interface E1-3971
E1.5 Exceptions E1-3972

Chapter E2 **The AArch32 Application Level Memory Model**
E2.1 About the Arm memory model E2-3974
E2.2 Atomicity in the Arm architecture E2-3976
E2.3 Definition of the Armv8 memory model E2-3980
E2.4 Ordering of translation table walks E2-3998
E2.5 Caches and memory hierarchy E2-3999
E2.6 Alignment support E2-4004
E2.7 Endian support E2-4006
E2.8 Memory types and attributes E2-4010
E2.9 Mismatched memory attributes E2-4020
E2.10 Synchronization and semaphores E2-4023

Part F **The AArch32 Instruction Sets**

Chapter F1 **The AArch32 Instruction Sets Overview**
F1.1 Support for instructions in different versions of the Arm architecture F1-4036
F1.2 Unified Assembler Language F1-4037
F1.3 Branch instructions F1-4039
F1.4 Data-processing instructions F1-4040
F1.5 PSTATE and banked register access instructions F1-4048
F1.6 Load/store instructions F1-4049
F1.7 Load/store multiple instructions F1-4052
F1.8 Miscellaneous instructions F1-4053

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F1.9 Exception-generating and exception-handling instructions F1-4055
F1.10 System register access instructions F1-4057
F1.11 Advanced SIMD and floating-point load/store instructions F1-4058
F1.12 Advanced SIMD and floating-point register transfer instructions F1-4060
F1.13 Advanced SIMD data-processing instructions F1-4061
F1.14 Floating-point data-processing instructions F1-4072

Chapter F2 About the T32 and A32 Instruction Descriptions
F2.1 Format of instruction descriptions F2-4076
F2.2 Standard assembler syntax fields F2-4080
F2.3 Conditional execution F2-4081
F2.4 Shifts applied to a register F2-4083
F2.5 Memory accesses F2-4085
F2.6 Encoding of lists of general-purpose registers and the PC F2-4086
F2.7 General information about the T32 and A32 instruction descriptions F2-4087
F2.8 Additional pseudocode support for instruction descriptions F2-4100
F2.9 Additional information about Advanced SIMD and floating-point instructions .. F2-4101

Chapter F3 T32 Instruction Set Encoding
F3.1 T32 instruction set encoding F3-4108
F3.2 About the T32 Advanced SIMD and floating-point instructions and their encoding F3-4178

Chapter F4 A32 Instruction Set Encoding
F4.1 A32 instruction set encoding F4-4180
F4.2 About the A32 Advanced SIMD and floating-point instructions and their encoding F4-4240

Chapter F5 T32 and A32 Base Instruction Set Instruction Descriptions
F5.1 Alphabetical list of T32 and A32 base instruction set instructions F5-4242
F5.2 Encoding and use of banked register transfer instructions F5-4951

Chapter F6 T32 and A32 Advanced SIMD and Floating-point Instruction Descriptions
F6.1 Alphabetical list of Advanced SIMD and floating-point instructions F6-4956

Part G The AArch32 System Level Architecture

Chapter G1 The AArch32 System Level Programmers' Model
G1.1 About the AArch32 System level programmers' model G1-5672
G1.2 Exception levels G1-5673
G1.3 Exception terminology G1-5674
G1.4 Execution state G1-5676
G1.5 Instruction Set state G1-5678
G1.6 Security state G1-5679
G1.7 Security state, Exception levels, and AArch32 execution privilege G1-5682
G1.8 Virtualization G1-5684
G1.9 AArch32 state PE modes, and general-purpose and Special-purpose registers G1-5686
G1.10 Process state, PSTATE G1-5695
G1.11 Instruction set states G1-5701
G1.12 Handling exceptions that are taken to an Exception level using AArch32 ..... G1-5703
G1.13 Routing of aborts taken to AArch32 state G1-5722
G1.14 Exception return to an Exception level using AArch32 ..... G1-5725
G1.15 Asynchronous exception behavior for exceptions taken from AArch32 state . G1-5730
G1.16 AArch32 state exception descriptions G1-5738
G1.17 Reset into AArch32 state G1-5760
G1.18 Mechanisms for entering a low-power state G1-5764
G1.19 The AArch32 System register interface G1-5769
G1.20 Advanced SIMD and floating-point support G1-5772
G1.21 Configurable instruction enables and disables, and trap controls G1-5778

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter G2 AArch32 Self-hosted Debug
G2.1 About self-hosted debug ... G2-5814
G2.2 The debug exception enable controls ... G2-5818
G2.3 Routing debug exceptions ... G2-5819
G2.4 Enabling debug exceptions from the current Privilege level and Security state G2-5821
G2.5 The effect of powerdown on debug exceptions ... G2-5823
G2.6 Summary of permitted routing and enabling of debug exceptions ... G2-5824
G2.7 Pseudocode description of debug exceptions ... G2-5826
G2.8 Breakpoint Instruction exceptions ... G2-5827
G2.9 Breakpoint exceptions ... G2-5830
G2.10 Watchpoint exceptions ... G2-5855
G2.11 Vector Catch exceptions ... G2-5869
G2.12 Synchronization and debug exceptions ... G2-5876

Chapter G3 AArch32 Self-hosted Trace
G3.1 About self-hosted trace ... G3-5880
G3.2 Prohibited regions in self-hosted trace ... G3-5881
G3.3 Self-hosted trace timestamps ... G3-5882
G3.4 Synchronization in self-hosted trace ... G3-5883

Chapter G4 The AArch32 System Level Memory Model
G4.1 About the memory system architecture ... G4-5886
G4.2 Address space ... G4-5887
G4.3 Mixed-endian support ... G4-5888
G4.4 AArch32 cache and branch predictor support ... G4-5889
G4.5 System register support for IMPLEMENTATION DEFINED memory features G4-5913
G4.6 External aborts ... G4-5914
G4.7 Memory barrier instructions ... G4-5916
G4.8 Pseudocode description of general memory System instructions ... G4-5917

Chapter G5 The AArch32 Virtual Memory System Architecture
G5.1 About VMSAv8-32 ... G5-5922
G5.2 The effects of disabling address translation stages on VMSAv8-32 behavior G5-5930
G5.3 Translation tables ... G5-5934
G5.4 The VMSAv8-32 Short-descriptor translation table format ... G5-5939
G5.5 The VMSAv8-32 Long-descriptor translation table format ... G5-5948
G5.6 Memory access control ... G5-5968
G5.7 Memory region attributes ... G5-5979
G5.8 Translation Lookaside Buffers (TLBs) ... G5-5991
G5.9 TLB maintenance requirements ... G5-5995
G5.10 Caches in VMSAv8-32 ... G5-6009
G5.11 VMSAv8-32 memory aborts ... G5-6012
G5.12 Exception reporting in a VMSAv8-32 implementation ... G5-6024
G5.13 Address translation instructions ... G5-6043
G5.14 Pseudocode description of VMSAv8-32 memory system operations ... G5-6050
G5.15 About the System registers for VMSAv8-32 ... G5-6052
G5.16 Functional grouping of VMSAv8-32 System registers ... G5-6058

Chapter G6 The Generic Timer in AArch32 state
G6.1 About the Generic Timer in AArch32 state ... G6-6060
G6.2 The AArch32 view of the Generic Timer ... G6-6064

Chapter G7 AArch32 System Register Encoding
G7.1 The AArch32 System register encoding space ... G7-6074
G7.2 VMSAv8-32 organization of registers in the (coproc==0b1110) encoding space G7-6075
G7.3 VMSAv8-32 organization of registers in the (coproc==0b1111) encoding space G7-6078

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter G8 **AArch32 System Register Descriptions**
G8.1 About the AArch32 System registers G8-6096
G8.2 General system control registers G8-6111
G8.3 Debug registers G8-6586
G8.4 Performance Monitors registers G8-6694
G8.5 Activity Monitors registers G8-6759
G8.6 RAS registers G8-6792
G8.7 Generic Timer registers G8-6840

Part H **External Debug**

Chapter H1 **About External Debug**
H1.1 Introduction to external debug H1-6922
H1.2 External debug H1-6923
H1.3 Required debug authentication H1-6924

Chapter H2 **Debug State**
H2.1 About Debug state H2-6926
H2.2 Halting the PE on debug events H2-6927
H2.3 Entering Debug state H2-6933
H2.4 Behavior in Debug state H2-6936
H2.5 Exiting Debug state H2-6963

Chapter H3 **Halting Debug Events**
H3.1 Introduction to Halting debug events H3-6966
H3.2 Halting Step debug events H3-6968
H3.3 Halt Instruction debug event H3-6978
H3.4 Exception Catch debug event H3-6979
H3.5 External Debug Request debug event H3-6983
H3.6 OS Unlock Catch debug event H3-6984
H3.7 Reset Catch debug events H3-6985
H3.8 Software Access debug event H3-6986
H3.9 Synchronization and Halting debug events H3-6987

Chapter H4 **The Debug Communication Channel and Instruction Transfer Register**
H4.1 Introduction H4-6990
H4.2 DCC and ITR registers H4-6991
H4.3 DCC and ITR access modes H4-6994
H4.4 Flow control of the DCC and ITR registers H4-6998
H4.5 Synchronization of DCC and ITR accesses H4-7002
H4.6 Interrupt-driven use of the DCC H4-7008
H4.7 Pseudocode description of the operation of the DCC and ITR registers H4-7009

Chapter H5 **The Embedded Cross-Trigger Interface**
H5.1 About the Embedded Cross-Trigger (ECT) H5-7012
H5.2 Basic operation on the ECT H5-7014
H5.3 Cross-triggers on a PE in an Armv8 implementation H5-7018
H5.4 Description and allocation of CTI triggers H5-7019
H5.5 CTI registers programmers' model H5-7023
H5.6 Examples H5-7024

Chapter H6 **Debug Reset and Powerdown Support**
H6.1 About Debug over powerdown H6-7028
H6.2 Power domains and debug H6-7029
H6.3 Core power domain power states H6-7030
H6.4 Powerup request mechanism H6-7032
H6.5 Emulating low-power states H6-7033

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H6.6 Debug OS Save and Restore sequences H6-7035
H6.7 Reset and debug H6-7041

Chapter H7 The PC Sample-based Profiling Extension
H7.1 About the PC Sample-based Profiling Extension H7-7044

Chapter H8 About the External Debug Registers
H8.1 Relationship between external debug and System registers H8-7048
H8.2 Endianness and supported access sizes H8-7049
H8.3 Synchronization of changes to the external debug registers H8-7050
H8.4 Memory-mapped accesses to the external debug interface H8-7054
H8.5 External debug interface register access permissions H8-7057
H8.6 External debug interface registers H8-7061
H8.7 Cross-trigger interface registers H8-7068
H8.8 External debug register resets H8-7070

Chapter H9 External Debug Register Descriptions
H9.1 About the debug registers H9-7074
H9.2 External debug registers H9-7075
H9.3 Cross-Trigger Interface registers H9-7177

Part I Memory-mapped Components of the Armv8 Architecture

Chapter I1 Requirements for Memory-mapped Components
I1.1 Supported access sizes I1-7224
I1.2 Synchronization of memory-mapped registers I1-7226
I1.3 Access requirements for reserved and unallocated registers I1-7228

Chapter I2 System Level Implementation of the Generic Timer
I2.1 About the Generic Timer specification I2-7230
I2.2 Memory-mapped counter module I2-7232
I2.3 Memory-mapped timer components I2-7236

Chapter I3 Recommended External Interface to the Performance Monitors
I3.1 About the external interface to the Performance Monitors registers I3-7242

Chapter I4 Recommended External Interface to the Activity Monitors
I4.1 About the external interface to the Activity Monitors Extension registers I4-7248

Chapter I5 External System Control Register Descriptions
I5.1 About the external system control register descriptions I5-7250
I5.2 External Performance Monitors registers summary I5-7252
I5.3 Performance Monitors external register descriptions I5-7255
I5.4 External Activity Monitors Extension registers summary I5-7328
I5.5 Activity Monitors external register descriptions I5-7330
I5.6 Generic Timer memory-mapped registers overview I5-7365
I5.7 Generic Timer memory-mapped register descriptions I5-7366
I5.8 RAS Registers I5-7410

Part J Architectural Pseudocode

Chapter J1 Armv8 Pseudocode
J1.1 Pseudocode for AArch64 operation J1-7508
J1.2 Pseudocode for AArch32 operation J1-7627
J1.3 Shared pseudocode J1-7705

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Part K Appendixes

Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors
K1.1 AArch32 CONSTRAINED UNPREDICTABLE behaviors K1-7830
K1.2 AArch64 CONSTRAINED UNPREDICTABLE behaviors K1-7855

Appendix K2 Recommended External Debug Interface
K2.1 About the recommended external debug interface K2-7874
K2.2 PMUEVENT bus K2-7878
K2.3 Recommended authentication interface K2-7879
K2.4 Management registers and CoreSight compliance K2-7881

Appendix K3 Recommendations for Performance Monitors Event Numbers for IMPLEMENTATION DEFINED Events
K3.1 Arm recommendations for IMPLEMENTATION DEFINED event numbers K3-7894
K3.2 Summary of events for exceptions taken to an Exception level using AArch64 K3-7909

Appendix K4 Recommendations for Reporting Memory Attributes on an Interconnect
K4.1 Arm recommendations for reporting memory attributes on an interconnect ... K4-7912

Appendix K5 Additional Information for Implementations of the Generic Timer
K5.1 Providing a complete set of features in a system level implementation K5-7914
K5.2 Gray-count scheme for timer distribution scheme K5-7916

Appendix K6 Legacy Instruction Syntax for AArch32 Instruction Sets
K6.1 Legacy Instruction Syntax K6-7918

Appendix K7 Address Translation Examples
K7.1 AArch64 Address translation examples K7-7926
K7.2 AArch32 Address translation examples K7-7938

Appendix K8 Example OS Save and Restore Sequences
K8.1 Save Debug registers K8-7948
K8.2 Restore Debug registers K8-7950

Appendix K9 Recommended Upload and Download Processes for External Debug
K9.1 Using memory access mode in AArch64 state K9-7954

Appendix K10 Software Usage Examples
K10.1 Use of the Advanced SIMD complex number instructions K10-7958
K10.2 Use of the Armv8.2 extensions to the Cryptographic Extension K10-7960

Appendix K11 Barrier Litmus Tests
K11.1 Introduction K11-7968
K11.2 Load-Acquire, Store-Release and barriers K11-7971
K11.3 Load-Acquire Exclusive, Store-Release Exclusive and barriers K11-7975
K11.4 Using a mailbox to send an interrupt K11-7980
K11.5 Cache and TLB maintenance instructions and barriers K11-7981
K11.6 Armv7 compatible approaches for ordering, using DMB and DSB barriers .. K11-7993

Appendix K12 Random Number Generation
K12.1 Properties of the generated random number K12-8008

Appendix K13 Arm Pseudocode Definition
K13.1 About the Arm pseudocode K13-8010

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K13.2 Pseudocode for instruction descriptions K13-8011
K13.3 Data types K13-8013
K13.4 Operators K13-8018
K13.5 Statements and control structures K13-8024
K13.6 Built-in functions K13-8029
K13.7 Miscellaneous helper procedures and functions K13-8032
K13.8 Arm pseudocode definition index K13-8034

Appendix K14 Registers Index

K14.1 Introduction and register disambiguation K14-8038
K14.2 Alphabetical index of AArch64 registers and System instructions K14-8043
K14.3 Functional index of AArch64 registers and System instructions K14-8058
K14.4 Alphabetical index of AArch32 registers and System instructions K14-8074
K14.5 Functional index of AArch32 registers and System instructions K14-8084
K14.6 Alphabetical index of memory-mapped registers K14-8096
K14.7 Functional index of memory-mapped registers K14-8103

Glossary

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Preface

This preface introduces the *Arm Architecture Reference Manual, Armv8, for Armv8-A architecture profile*. It contains the following sections:

- *About this manual* on page xviii.
- *Using this manual* on page xx.
- *Conventions* on page xxvi.
- *Additional reading* on page xxviii.
- *Feedback* on page xxx.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

About this manual

This manual describes the Arm® architecture v8, Armv8. The architecture describes the operation of an Armv8-A *Processing element (PE)*, and this manual includes descriptions of:

- The two Execution states, AArch64 and AArch32.
- The instruction sets:
  - In AArch32 state, the A32 and T32 instruction sets, that are compatible with earlier versions of the Arm architecture.
  - In AArch64 state, the A64 instruction set.
- The states that determine how a PE operates, including the current Exception level and Security state, and in AArch32 state the PE mode.
- The Exception model.
- The interprocessing model, that supports transitioning between AArch64 state and AArch32 state.
- The memory model, that defines memory ordering and memory management. This manual covers a single architecture profile, Armv8-A, that defines a *Virtual Memory System Architecture* (VMSA).
- The programmers' model, and its interfaces to System registers that control most PE and memory system features, and provide status information.
- The Advanced SIMD and floating-point instructions, that provide high-performance:
  - Single-precision, half-precision, and double-precision floating-point operations.
  - Conversions between double-precision, single-precision, and half-precision floating-point values.
  - Integer, single-precision floating-point, half-precision floating-point, and in A64, double-precision vector operations in all instruction sets.
  - Single-precision, half-precision, and double-precision floating-point vector operations in the A64 instruction set.
- The security model, that provides two security states to support secure applications.
- The virtualization model, that support the virtualization of Non-secure operation.
- The Debug architecture, that provides software access to debug features.

This manual gives the assembler syntax for the instructions it describes, meaning that it describes instructions in textual form. However, this manual is not a tutorial for Arm assembler language, nor does it describe Arm assembler language, except at a very basic level. To make effective use of Arm assembler language, read the documentation supplied with the assembler being used.

This manual is organized into parts:

Part A Provides an introduction to the Armv8-A architecture, and an overview of the AArch64 and AArch32 Execution states.

Part B Describes the application level view of the AArch64 Execution state, meaning the view from EL0. It describes the application level view of the programmers' model and the memory model.

Part C Describes the A64 instruction set, that is available in the AArch64 Execution state. The descriptions for each instruction also include the precise effects of each instruction when executed at EL0, described as *unprivileged* execution, including any restrictions on its use, and how the effects of the instruction differ at higher Exception levels. This information is of primary importance to authors and users of compilers, assemblers, and other programs that generate Arm machine code.

Part D Describes the system level view of the AArch64 Execution state. It includes details of the System registers, most of which are not accessible from EL0, and the system level view of the programmers' model and the memory model. This part includes the description of self-hosted debug.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Part E Describes the application level view of the AArch32 Execution state, meaning the view from the EL0. It describes the application level view of the programmers' model and the memory model.

Note

In AArch32 state, execution at EL0 is execution in User mode.

Part F Describes the T32 and A32 instruction sets, that are available in the AArch32 Execution state. These instruction sets are backwards-compatible with earlier versions of the Arm architecture. This part describes the precise effects of each instruction when executed in User mode, described as *unprivileged* execution or execution at EL0, including any restrictions on its use, and how the effects of the instruction differ at higher Exception levels. This information is of primary importance to authors and users of compilers, assemblers, and other programs that generate Arm machine code.

Note

User mode is the only mode where software execution is unprivileged.

Part G Describes the system level view of the AArch32 Execution state, that is generally compatible with earlier versions of the Arm architecture. This part includes details of the System registers, most of which are not accessible from EL0, and the instruction interface to those registers. It also describes the system level view of the programmers' model and the memory model.

Part H Describes the Debug architecture for external debug. This provides configuration, breakpoint and watchpoint support, and a *Debug Communications Channel* (DCC) to a debug host.

Part I Describes additional features of the architecture that are not closely coupled to a *processing element* (PE), and therefore are accessed through memory-mapped interfaces. Some of these features are OPTIONAL.

Part J Provides pseudocode that describes various features of the Armv8 architecture.

Part K, Appendixes

Provide additional information. Some appendixes give information that is not part of the Armv8 architectural requirements. The cover page of each appendix indicates its status.

Glossary Defines terms used in this document that have a specialized meaning.

Note

Terms that are generally well understood in the microelectronics industry are not included in the Glossary.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Using this manual

The information in this manual is organized into parts, as described in this section.

Part A, Introduction and Architecture Overview

Part A gives an overview of the Armv8-A architecture profile, including its relationship to the other Arm PE architectures. It introduces the terminology used to describe the architecture, and gives an overview of the Executions states, AArch64 and AArch32. It contains the following chapter:

Chapter A1 ***Introduction to the Armv8 Architecture***

Read this for an introduction to the Armv8 architecture.

Part B, The AArch64 Application Level Architecture

Part B describes the AArch64 state application level view of the architecture. It contains the following chapters:

Chapter B1 ***The AArch64 Application Level Programmers' Model***

Read this for an application level description of the programmers' model for software executing in AArch64 state. It describes execution at EL0 when EL0 is using AArch64 state.

Chapter B2 ***The AArch64 Application Level Memory Model***

Read this for an application level description of the memory model for software executing in AArch64 state. It describes the memory model for execution in EL0 when EL0 is using AArch64 state. It includes information about Arm memory types, attributes, and memory access controls.

Part C, The A64 Instruction Set

Part C describes the A64 instruction set, that is used in AArch64 state. It contains the following chapters:

Chapter C1 ***The A64 Instruction Set***

Read this for a description of the A64 instruction set and common instruction operation details.

Chapter C2 ***About the A64 Instruction Descriptions***

Read this to understand the format of the A64 instruction descriptions.

Chapter C3 ***A64 Instruction Set Overview***

Read this for an overview of the individual A64 instructions, that are divided into five functional groups.

Chapter C4 ***A64 Instruction Set Encoding***

Read this for a description of the A64 instruction set encoding.

Chapter C5 ***The A64 System Instruction Class***

Read this for a description of the AArch64 System instructions and register descriptions, and the System instruction class encoding space.

Chapter C6 ***A64 Base Instruction Descriptions***

Read this for information on key aspects of the A64 base instructions and for descriptions of the individual instructions, which are listed in alphabetical order.

Chapter C7 ***A64 Advanced SIMD and Floating-point Instruction Descriptions***

Read this for information on key aspects of the A64 Advanced SIMD and floating-point instructions and for descriptions of the individual instructions, which are listed in alphabetical order.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Part D, The AArch64 System Level Architecture

Part D describes the AArch64 state system level view of the architecture. It contains the following chapters:

Chapter D1 ***The AArch64 System Level Programmers' Model***

Read this for a description of the AArch64 state system level view of the programmers' model.

Chapter D2 ***AArch64 Self-hosted Debug***

Read this for an introduction to, and a description of, self-hosted debug in AArch64 state.

Chapter D3 ***AArch64 Self-hosted Trace***

Read this for an introduction to, and a description of, self-hosted trace in AArch64 state.

Chapter D4 ***The AArch64 System Level Memory Model***

Read this for a description of the AArch64 state system level view of the general features of the memory system.

Chapter D5 ***The AArch64 Virtual Memory System Architecture***

Read this for a system level view of the AArch64 Virtual Memory System Architecture (VMSA), the memory system architecture of an Armv8 implementation that is executing in AArch64 state.

Chapter D7 ***The Performance Monitors Extension***

Read this for a description of an implementation of the Arm Performance Monitors, that are an optional non-invasive debug component.

Chapter D8 ***The Activity Monitors Extension***

Read this for a description of an implementation of the Arm Activity Monitors, an optional non-invasive component.

Chapter D9 ***The Statistical Profiling Extension***

Read this for a description of an implementation of the Statistical Profiling Extension, that is an optional AArch64 state non-invasive debug component.

Chapter D10 ***Statistical Profiling Extension Sample Record Specification***

Read this for a description the sample records generated by the Statistical Profiling Extension.

Chapter D11 ***The Generic Timer in AArch64 state***

Read this for a description of the AArch64 view of an implementation of the Arm Generic Timer.

Chapter D12 ***AArch64 System Register Encoding***

Read this for a description of the description of the encoding of the AArch64 System registers, and the other uses of the AArch64 System registers encoding space.

Chapter D13 ***AArch64 System Register Descriptions***

Read this for an introduction to, and description of, each of the AArch64 System registers.

Part E, The AArch32 Application Level Architecture

Part E describes the AArch32 state application level view of the architecture. It contains the following chapters:

Chapter E1 ***The AArch32 Application Level Programmers' Model***

Read this for an application level description of the programmers' model for software executing in AArch32 state. It describes execution at EL0 when EL0 is using AArch32 state.

Chapter E2 ***The AArch32 Application Level Memory Model***

Read this for an application level description of the memory model for software executing in AArch32 state. It describes the memory model for execution in EL0 when EL0 is using AArch32 state. It includes information about Arm memory types, attributes, and memory access controls.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Part F, The AArch32 Instruction Sets

Part F describes the T32 and A32 instruction sets, that are used in AArch32 state. It contains the following chapters:

Chapter F1 ***The AArch32 Instruction Sets Overview***

Read this for an overview of the T32 and A32 instruction sets.

Chapter F2 ***About the T32 and A32 Instruction Descriptions***

Read this to understand the format of the T32 and A32 instruction descriptions.

Chapter F3 ***T32 Instruction Set Encoding***

Read this for a description of the T32 instruction set encoding. This includes the T32 encoding of the Advanced SIMD and floating-point instructions.

Chapter F4 ***A32 Instruction Set Encoding***

Read this for a description of the A32 instruction set encoding. This includes the A32 encoding of the Advanced SIMD and floating-point instructions.

Chapter F5 ***T32 and A32 Base Instruction Set Instruction Descriptions***

Read this for a description of each of the T32 and A32 base instructions.

Chapter F6 ***T32 and A32 Advanced SIMD and Floating-point Instruction Descriptions***

Read this for a description of each of the T32 and A32 Advanced SIMD and floating-point instructions.

Part G, The AArch32 System Level Architecture

Part G describes the AArch32 state system level view of the architecture. It contains the following chapters:

Chapter G1 ***The AArch32 System Level Programmers' Model***

Read this for a description of the AArch32 state system level view of the programmers' model for execution in an Exception level that is using AArch32.

Chapter G2 ***AArch32 Self-hosted Debug***

Read this for an introduction to, and a description of, self-hosted debug in AArch64 state.

Chapter G3 ***AArch32 Self-hosted Trace***

Read this for an introduction to, and a description of, self-hosted trace in AArch64 state.

Chapter G4 ***The AArch32 System Level Memory Model***

Read this for a system level view of the general features of the memory system.

Chapter G5 ***The AArch32 Virtual Memory System Architecture***

Read this for a description of the AArch32 Virtual Memory System Architecture (VMSA).

Chapter G6 ***The Generic Timer in AArch32 state***

Read this for a description of the AArch32 view of an implementation of the Arm Generic Timer.

Chapter G7 ***AArch32 System Register Encoding***

Read this for a description of the description of the encoding of the AArch32 System registers, including the System instructions that are part of the AArch32 System registers encoding space.

Chapter G8 ***AArch32 System Register Descriptions***

Read this for a description of each of the AArch32 System registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Part H, External Debug

Part H describes the architecture for external debug. It contains the following chapters:

Chapter H1 ***About External Debug***

Read this for an introduction to external debug, and a definition of the scope of this part of the manual.

Chapter H2 ***Debug State***

Read this for a description of debug state, which the PE might enter as the result of a Halting debug event.

Chapter H3 ***Halting Debug Events***

Read this for a description of the external debug events referred to as Halting debug events.

Chapter H4 ***The Debug Communication Channel and Instruction Transfer Register***

Read this for a description of the communication between a debugger and the PE debug logic using the Debug Communications Channel and the Instruction Transfer register.

Chapter H5 ***The Embedded Cross-Trigger Interface***

Read this for a description of the embedded cross-trigger interface.

Chapter H6 ***Debug Reset and Powerdown Support***

Read this for a description of reset and powerdown support in the Debug architecture.

Chapter H7 ***The PC Sample-based Profiling Extension***

Read this for a description of the PC Sample-based Profiling Extension that is an OPTIONAL extension to an Armv8 implementation.

Chapter H8 ***About the External Debug Registers***

Read this for some additional information about the external debug registers.

Chapter H9 ***External Debug Register Descriptions***

Read this for a description of each external debug register.

Part I, Memory-mapped Components of the Armv8 Architecture

Part I describes the memory-mapped components in the architecture. It contains the following chapters:

Chapter I1 ***Requirements for Memory-mapped Components***

Read this for descriptions of some general requirements for memory-mapped components within a system that complies with the Armv8 Architecture.

Chapter I2 ***System Level Implementation of the Generic Timer***

Read this for a definition of a system level implementation of the Generic Timer.

Chapter I3 ***Recommended External Interface to the Performance Monitors***

Read this for a description of the recommended memory-mapped and external debug interfaces to the Performance Monitors.

Chapter I4 ***Recommended External Interface to the Activity Monitors***

Read this for a description of the recommended memory-mapped interface to the Activity Monitors.

Chapter I5 ***External System Control Register Descriptions***

Read this for a description of each memory-mapped system control register.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Part J, Architectural Pseudocode

Part J contains pseudocode that describes various features of the Arm architecture. It contains the following chapter:

Chapter J1 ***Armv8 Pseudocode***

Read this for the pseudocode definitions that describe various features of the Armv8 architecture, for operation in AArch64 state and in AArch32 state.

Part K, Appendixes

This manual contains the following appendixes:

Appendix K1 ***Architectural Constraints on UNPREDICTABLE Behaviors***

Read this for a description of the architecturally-required constraints on UNPREDICTABLE behaviors in the Armv8 architecture, including AArch32 behaviors that were UNPREDICTABLE in previous versions of the architecture.

Appendix K2 ***Recommended External Debug Interface***

Read this for a description of the recommended external debug interface.

Note

This description is not part of the Arm architecture specification. It is included here as supplementary information, for the convenience of developers and users who might require this information.

Appendix K3 ***Recommendations for Performance Monitors Event Numbers for IMPLEMENTATION DEFINED Events***

Read this for a description of Arm recommendations for the use of the IMPLEMENTATION DEFINED event numbers.

Note

This description is not part of the Arm architecture specification. It is included here as supplementary information, for the convenience of developers and users who might require this information.

Appendix K4 ***Recommendations for Reporting Memory Attributes on an Interconnect***

Read this for the Arm recommendations about how the architectural memory attributes are reported on an interconnect.

Appendix K5 ***Additional Information for Implementations of the Generic Timer***

Read this for additional information about implementations of the Arm Generic Timer. This information does not form part of the architectural definition of the Generic Timer.

Appendix K6 ***Legacy Instruction Syntax for AArch32 Instruction Sets***

Read this for information about the pre-UAL syntax of the AArch32 instruction sets, which can still be valid for the A32 instruction set.

Appendix K7 ***Address Translation Examples***

Read this for examples of translation table lookups using the translation regimes described in Chapter D5 *The AArch64 Virtual Memory System Architecture* and Chapter G5 *The AArch32 Virtual Memory System Architecture*.

Appendix K8 ***Example OS Save and Restore Sequences***

Read this for software examples that perform the OS Save and Restore sequences for an Armv8 debug implementation.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

Chapter H6 *Debug Reset and Powerdown Support* describes the OS Save and Restore mechanism.

Appendix K9 *Recommended Upload and Download Processes for External Debug*

Read this for information about implementing and using the Arm architecture.

Note

This description is not part of the Arm architecture specification. It is included here as supplementary information, for the convenience of developers and users who might require this information.

Appendix K10 *Software Usage Examples*

Read this for software examples that help understanding of some aspects of the Arm architecture.

Note

This description is not part of the Arm architecture specification. It is included here as supplementary information, for the convenience of developers and users who might require this information.

Appendix K11 *Barrier Litmus Tests*

Read this for examples of the use of barrier instructions provided by the Armv8 architecture.

Note

This description is not part of the Arm architecture specification. It is included here as supplementary information, for the convenience of developers and users who might require this information.

Appendix K13 *Arm Pseudocode Definition*

Read this for definitions of the AArch32 pseudocode.

Appendix K14 *Registers Index*

Read this for an alphabetic and functional index of AArch32 and AArch64 registers, and memory-mapped registers.

Glossary

Defines terms used in this document that have a specialized meaning.

Note

Terms that are generally well understood in the microelectronics industry are not included in the Glossary.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Conventions

The following sections describe conventions that this book can use:

- *Typographic conventions*.
- *Signals* on page xxvii.
- *Numbers* on page xxvii.
- *Pseudocode descriptions* on page xxvii.
- *Assembler syntax descriptions* on page xxvii.

Typographic conventions

The typographical conventions are:

italic Introduces special terminology, and denotes citations.

bold Denotes signal names, and is used for terms in descriptive lists, where appropriate.

`monospace` Used for assembler syntax descriptions, pseudocode, and source code examples.

Also used in the main text for instruction mnemonics and for references to other items appearing in assembler syntax descriptions, pseudocode, and source code examples.

SMALL CAPITALS

Used in body text for a few terms that have specific technical meanings, and are defined in the *Glossary*.

Colored text Indicates a link. This can be:

- A URL, for example `http://infocenter.arm.com`.
- A cross-reference, that includes the page number of the referenced information if it is not on the current page, for example, *Assembler syntax descriptions* on page xxvii.
- A link, to a chapter or appendix, or to a glossary entry, or to the section of the document that defines the colored term, for example *Simple sequential execution* or SCTLR.

{ and } Braces, { and }, have two distinct uses:

Optional items

In syntax descriptions braces enclose optional items. In the following example they indicate that the `<shift>` parameter is optional:

```
ADD <Wd|WSP>, <Wn|WSP>, #<imm>{, <shift>}
```

Similarly they can be used in generalized field descriptions, for example TCR_ELx.{I}PS refers to a field in the TCR_ELx registers that is called either IPS or PS.

Sets of items

Braces can be used to enclose sets. For example, HCR_EL2.{E2H, TGE} refers to a set of two register fields, HCR_EL2.E2H and HCR_EL2.TGE

Notes Notes are formatted as:

Note

This is a Note.

In this Manual, Notes are used only to provide additional information, usually to help understanding of the text. While a Note may repeat architectural information given elsewhere in the Manual, a Note never provides any part of the definition of the architecture.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Signals

In general this specification does not define hardware signals, but it does include some signal examples and recommendations. The signal conventions are:

Signal level The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means:

- HIGH for active-HIGH signals.
- LOW for active-LOW signals.

Lower-case n At the start or end of a signal name denotes an active-LOW signal.

Numbers

Numbers are normally written in decimal. Binary numbers are preceded by `0b`, and hexadecimal numbers by `0x`. In both cases, the prefix and the associated value are written in a `monospace` font, for example `0xFFFF0000`. To improve readability, long numbers can be written with an underscore separator between every four characters, for example `0xFFFF_0000_0000_0000`. Ignore any underscores when interpreting the value of a number.

Pseudocode descriptions

This manual uses a form of pseudocode to provide precise descriptions of the specified functionality. This pseudocode is written in `monospace` font, and is described in Appendix K13 *Arm Pseudocode Definition*.

Assembler syntax descriptions

This manual contains numerous syntax descriptions for assembler instructions and for components of assembler instructions. These are shown in a `monospace` font, and use the conventions described in *Structure of the A64 assembler language* on page C1-183, Appendix K13 *Arm Pseudocode Definition*, and *Pseudocode operators and keywords on page K12-5648*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Additional reading

This section lists relevant publications from Arm and third parties.

See the Infocenter, https://developer.arm.com, for access to Arm documentation.

Arm publications

- *ARM® AMBA® 4 ATB Protocol Specification, ATBv1.0 and ATBv1.1*, (ARM IHI 0032B).
- *ARM® Architecture Reference Manual, ARMv7-A and ARMv7-R edition* (ARM DDI 0406).
- *ARM® Architecture Reference Manual Supplement, ARMv8, for the ARMv8-R AArch32 architecture profile* (ARM DDI 0568).
- *ARM® Debug Interface Architecture Specification, ADIv6.0* (ARM IHI 0074).
- *ARM® Debug Interface Architecture Specification, ADIv5.0 to ADIv5.2* (ARM IHI 0031).
- *ARM® Embedded Trace Macrocell Architecture Specification, ETMv4* (ARM IHI 0064).
- *ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0 and version 4.0* (ARM IHI 0069).
- *ARM® CoreSight™ SoC Technical Reference Manual* (ARM DDI 0480).
- *ARM® CoreSight™ Architecture Specification* (ARM IHI 0029).
- *ARM® Procedure Call Standard for the ARM 64-bit Architecture* (ARM IHI 0055).
- *ARM® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile* (ARM DDI 0587).
- *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A* (ARM DDI 0584).
- *Arm® Architecture Reference Manual Supplement, Memory System Resource Partitioning and Monitoring* (MPAM), for Armv8-A (ARM DDI 0598).

Other publications

The following publications are referred to in this manual, or provide more information:

- *Announcing the Advanced Encryption Standard (AES)*, Federal Information Processing Standards Publication 197, November 2001.
- IEEE Std 754-2008, *IEEE Standard for Floating-point Arithmetic*, August 2008.
- IEEE Std 754-1985, *IEEE Standard for Floating-point Arithmetic*, March 1985.
- *Secure Hash Standard (SHA)*, Federal Information Processing Standards Publication 180-2, August 2002.
- *The Galois/Counter Mode of Operation*, McGraw, D. and Viega, J., Submission to NIST Modes of Operation Process, January 2004.
- *Memory Consistency Models for Shared Memory-Multiprocessors*, Gharachorloo, Kourosh, 1995, Stanford University Technical Report CSL-TR-95-685.
- *Standard Manufacturer's Identification Code, JEP106*, JEDEC Solid State Technology Association.
- *SM3 Cryptographic Hash Algorithm*, China Internet Network Information Center (CNNIC).
- *SM4 Block Cipher Algorithm*, China Internet Network Information Center (CNNIC).
- *The QARMA Block Cipher Family*, Roberto Avanzi, Qualcomm Product Security Initiative.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Available from https://eprint.iacr.org/2016/444.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Feedback

Arm welcomes feedback on its documentation.

Feedback on this manual

If you have comments on the content of this manual, send e-mail to `errata@arm.com`. Give:

- The title.
- The number, ARM DDI 0487F.a.
- The page numbers to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests PDFs only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the appearance or behavior of any document when viewed with any other PDF reader.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Part A

Armv8 Architecture Introduction and Overview

Chapter A1
Introduction to the Armv8 Architecture

This chapter introduces the Arm architecture. It contains the following sections:

- *About the Arm architecture* on page A1-34.
- *Architecture profiles* on page A1-36.
- *Armv8 architectural concepts* on page A1-37.
- *Supported data types* on page A1-40.
- *Advanced SIMD and floating-point support* on page A1-52.
- *The Arm memory model* on page A1-59.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A1.1 About the Arm architecture

The Arm architecture described in this Architecture Reference Manual defines the behavior of an abstract machine, referred to as a *processing element*, often abbreviated to *PE*. Implementations compliant with the Arm architecture must conform to the described behavior of the processing element. It is not intended to describe how to build an implementation of the PE, nor to limit the scope of such implementations beyond the defined behaviors.

Except where the architecture specifies differently, the programmer-visible behavior of an implementation that is compliant with the Arm architecture must be the same as a simple sequential execution of the program on the processing element. This programmer-visible behavior does not include the execution time of the program.

The Arm Architecture Reference Manual also describes rules for software to use the processing element.

The Arm architecture includes definitions of:

- An associated debug architecture, see:
  - Chapter D2 *AArch64 Self-hosted Debug*.
  - Chapter G2 *AArch32 Self-hosted Debug*.
  - Part H of this manual, *External Debug* on page 6919.
- Associated trace architectures that define PE Trace Units that implementers can implement with the associated processor hardware. For more information, see:
  - The *Embedded Trace Macrocell Architecture Specification.*
  - Chapter D3 *AArch64 Self-hosted Trace*.
  - Chapter G3 *AArch32 Self-hosted Trace*.

Note

A PE Trace Unit may be named a trace macrocell in other documentation.

The Arm architecture is a *Reduced Instruction Set Computer* (RISC) architecture with the following RISC architecture features:

- A large uniform register file.
- A *load/store* architecture, where data-processing operations only operate on register contents, not directly on memory contents.
- Simple addressing modes, with all load/store addresses determined from register contents and instruction fields only.

The architecture defines the interaction of the PE with memory, including caches, and includes a memory translation system. It also describes how multiple PEs interact with each other and with other observers in a system.

This document defines the Armv8-A architecture *profile*. See *Architecture profiles* on page A1-36 for more information.

The Arm architecture supports implementations across a wide range of performance points. Implementation size, performance, and very low power consumption are key attributes of the Arm architecture.

An important feature of the Armv8 architecture is backwards compatibility, combined with the freedom for optimal implementation in a wide range of standard and more specialized use cases. The Armv8 architecture supports:

- A 64-bit Execution state, AArch64.
- A 32-bit Execution state, AArch32, that is compatible with previous versions of the Arm architecture.

Note

The AArch32 Execution state is compatible with the Armv7-A architecture profile, and enhances that profile to support some features included in the AArch64 Execution state.

Features that are optional are explicitly defined as such in this Manual.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

The presence of an ID register field for a feature does not imply that the feature is optional.

Both Execution states support SIMD and floating-point instructions:

- AArch32 state provides:
  - SIMD instructions in the base instruction sets that operate on the 32-bit general-purpose registers.
  - Advanced SIMD instructions that operate on registers in the *SIMD and floating-point register* (SIMD&FP register) file.
  - Floating-point instructions that operate on registers in the SIMD&FP register file.
- AArch64 state provides:
  - Advanced SIMD instructions that operate on registers in the SIMD&FP register file.
  - Floating-point instructions that operate on registers in the SIMD&FP register file.

Note

See *Conventions* on page xxvi for information about conventions used in this manual, including the use of SMALL CAPITALS for particular terms that have Arm-specific meanings that are defined in the *Glossary*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A1.2 Architecture profiles

The Arm architecture has evolved significantly since its introduction, and Arm continues to develop it. Eight major versions of the architecture have been defined to date, denoted by the version numbers 1 to 8. Of these, the first three versions are now obsolete.

The generic names AArch64 and AArch32 describe the 64-bit and 32-bit Execution states:

AArch64 Is the 64-bit Execution state, meaning addresses are held in 64-bit registers, and instructions in the base instruction set can use 64-bit registers for their processing. AArch64 state supports the A64 instruction set.

AArch32 Is the 32-bit Execution state, meaning addresses are held in 32-bit registers, and instructions in the base instruction sets use 32-bit registers for their processing. AArch32 state supports the T32 and A32 instruction sets.

Note

The *Base instruction set* comprises the supported instructions other than the Advanced SIMD and floating-point instructions.

See sections *Execution state* on page A1-37 and *The Armv8 instruction sets* on page A1-38 for more information.

Arm defines three architecture profiles:

A Application profile, described in this manual:

- Supports a *Virtual Memory System Architecture* (VMSA) based on a *Memory Management Unit* (MMU).

  Note

  An Armv8-A implementation can be called an AArchv8-A implementation.

- Supports the A64, A32, and T32 instruction sets.

R Real-time profile:

- Supports a *Protected Memory System Architecture* (PMSA) based on a *Memory Protection Unit* (MPU).
- Supports the A32 and T32 instruction sets.

M Microcontroller profile:

- Implements a programmers' model designed for low-latency interrupt processing, with hardware stacking of registers and support for writing interrupt handlers in high-level languages.
- Implements a variant of the R-profile PMSA.
- Supports a variant of the T32 instruction set.

Note

This Architecture Reference Manual describes only the Armv8-A profile.

For information about the R and M architecture profiles, and earlier Arm architecture versions see:

- The *ARM® Architecture Reference Manual Supplement, ARMv8, for the ARMv8-R AArch32 architecture profile.*
- The *ARM® Architecture Reference Manual, ARMv7-A and ARMv7-R edition*.
- The *Arm®v8-M Architecture Reference Manual*.
- The *ARM®v7-M Architecture Reference Manual*.
- The *ARM®v6-M Architecture Reference Manual*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A1.3 Armv8 architectural concepts

Armv8 introduces major changes to the Arm architecture, while maintaining a high level of consistency with previous versions of the architecture. The Armv8 Architecture Reference Manual includes significant changes in the terminology used to describe the architecture, and this section introduces both the Armv8 architectural concepts and the associated terminology.

The following subsections describe key Armv8 architectural concepts. Each section introduces the corresponding terms that are used to describe the architecture:

- *Execution state*.
- *The Armv8 instruction sets* on page A1-38.
- *System registers* on page A1-38.
- *Armv8 Debug* on page A1-39.

A1.3.1 Execution state

The Execution state defines the PE execution environment, including:

- The supported register widths.
- The supported instruction sets.
- Significant aspects of:
  - The exception model.
  - The *Virtual Memory System Architecture* (VMSA).
  - The programmers' model.

The Execution states are:

AArch64 The 64-bit Execution state. This Execution state:

- Provides 31 64-bit general-purpose registers, of which X30 is used as the procedure link register.
- Provides a 64-bit *program counter* (PC), *stack pointer*s (SPs), and *exception link registers* (ELRs).
- Provides 32 128-bit registers for SIMD vector and scalar floating-point support.
- Provides a single instruction set, A64. For more information, see *The Armv8 instruction sets* on page A1-38.
- Defines the Armv8 Exception model, with up to four Exception levels, EL0 - EL3, that provide an *execution privilege* hierarchy, see *Exception levels* on page D1-2314.
- Provides support for 64-bit *virtual addressing*. For more information, including the limits on address ranges, see Chapter D5 *The AArch64 Virtual Memory System Architecture*.
- Defines a number of *Process state* (PSTATE) elements that hold PE state. The A64 instruction set includes instructions that operate directly on various PSTATE elements.
- Names each System register using a suffix that indicates the lowest Exception level at which the register can be accessed.

AArch32 The 32-bit Execution state. This Execution state:

- Provides 13 32-bit general-purpose registers, and a 32-bit PC, SP, and *link register* (LR). The LR is used as both an ELR and a procedure link register.

  Some of these registers have multiple *banked* instances for use in different PE *modes*.
- Provides a single ELR, for exception returns from Hyp mode.
- Provides 32 64-bit registers for Advanced SIMD vector and scalar floating-point support.
- Provides two instruction sets, A32 and T32. For more information, see *The Armv8 instruction sets* on page A1-38.
- Supports the Armv7-A exception model, based on *PE modes*, and maps this onto the Armv8 Exception model, that is based on the Exception levels.
- Provides support for 32-bit virtual addressing.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Defines a number of *Process state* (PSTATE) elements that hold PE state. The A32 and T32 instruction sets include instructions that operate directly on various PSTATE elements, and instructions that access PSTATE by using the *Application Program Status Register* (APSR) or the *Current Program Status Register* (CPSR).

Later subsections give more information about the different properties of the Execution states.

Transferring control between the AArch64 and AArch32 Execution states is known as *interprocessing*. The PE can move between Execution states only on a change of Exception level, and subject to the rules given in *Interprocessing* on page D1-2406. This means different software layers, such as an application, an operating system kernel, and a hypervisor, executing at different Exception levels, can execute in different Execution states.

A1.3.2 The Armv8 instruction sets

In Armv8 the possible instruction sets depend on the Execution state:

AArch64 AArch64 state supports only a single instruction set, called A64. This is a fixed-length instruction set that uses 32-bit instruction encodings.

For information on the A64 instruction set, see Chapter C3 *A64 Instruction Set Overview*.

AArch32 AArch32 state supports the following instruction sets:

A32 This is a fixed-length instruction set that uses 32-bit instruction encodings.

T32 This is a variable-length instruction set that uses both 16-bit and 32-bit instruction encodings.

In previous documentation, these instruction sets were called the ARM and Thumb instruction sets. Armv8 extends each of these instruction sets. In AArch32 state, the Instruction set state determines the instruction set that the PE executes.

For information on the A32 and T32 instruction sets, see Chapter F1 *The AArch32 Instruction Sets Overview*.

The Armv8 instruction sets support SIMD and scalar floating-point instructions. See *Advanced SIMD and floating-point support* on page A1-52.

A1.3.3 System registers

System registers provide control and status information of architected features.

The System registers use a standard naming format: <register_name>.<bit_field_name> to identify specific registers as well as control and status bits within a register.

Bits can also be described by their numerical position in the form <register_name>[x:y] or the generic form bits[x:y].

In addition, in AArch64 state, most register names include the lowest Exception level that can access the register as a suffix to the register name:

- <register_name>_ELx, where x is 0, 1, 2, or 3.

For information about Exception levels, see *Exception levels* on page D1-2314.

The System registers comprise:

- The following registers that are described in this manual:
  - General system control registers.
  - Debug registers.
  - Generic Timer registers.
  - Optionally, Performance Monitor registers.
  - Optionally, the Activity Monitors registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Optionally, one or more of the following groups of registers that are defined in other Arm architecture specifications:
  - Trace System registers, as defined in the *Embedded Trace Macrocell Architecture Specification, ETMv4*.
  - Scalable Vector Extension System registers, as defined in the Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A.
  - *Generic Interrupt Controller* (GIC) System registers, see *The Arm Generic Interrupt Controller System registers*.
- RAS Extension System registers, as defined in the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*. The RAS Extension is a mandatory extension to the Armv8.2 architecture, and an optional extension to the Armv8.0 and the Armv8.1 architectures.

For information about the AArch64 System registers, see Chapter D13 *AArch64 System Register Descriptions*.

For information about the AArch32 System registers, see Chapter G8 *AArch32 System Register Descriptions*.

The Arm Generic Interrupt Controller System registers

From version 3 of the Arm Generic Interrupt Controller architecture, GICv3, the GIC architecture specification defines a System register interface to some of its functionality. The System register summaries in this manual include these registers, see:

- *About the GIC System registers* on page D12-2865, for more information about the AArch64 GIC System registers.
- *About the GIC System registers* on page G7-6092, for more information about the AArch32 GIC System registers.

These sections give only short overviews of the GIC System registers. For more information, including descriptions of the registers, see the *ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0 and version 4.0* (ARM IHI 0069).

Note

The programmers' model for earlier versions of the GIC architecture is wholly memory-mapped.

A1.3.4 Armv8 Debug

Armv8 supports the following:

Self-hosted debug

In this model, the PE generates *debug exceptions*. Debug exceptions are part of the Armv8 Exception model.

External debug

In this model, *debug events* cause the PE to enter *Debug state*. In Debug state, the PE is controlled by an external debugger.

All Armv8 implementations support both models. The model chosen by a particular user depends on the debug requirements during different stages of the design and development life cycle of the product. For example, external debug might be used during debugging of the hardware implementation and OS bring-up, and self-hosted debug might be used during application development.

For more information about self-hosted debug:

- In AArch64 state, see Chapter D2 *AArch64 Self-hosted Debug*.
- In AArch32 state, see Chapter G2 *AArch32 Self-hosted Debug*.

For more information about external debug, see Part H *External Debug* on page 6919.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A1.4 Supported data types

The Armv8 architecture supports the following integer data types:

Byte 8 bits.

Halfword 16 bits.

Word 32 bits.

Doubleword 64 bits.

Quadword 128 bits.

The architecture also supports the following floating-point data types:

- Half-precision, see *Half-precision floating-point formats* on page A1-44 for details.
- Single-precision, see *Single-precision floating-point format* on page A1-46 for details.
- Double-precision, see *Double-precision floating-point format* on page A1-47 for details.
- BFloat16, see *BFloat16 floating-point format* on page A1-48 for details.

It also supports:

- Fixed-point interpretation of words and doublewords. See *Fixed-point format* on page A1-50.
- Vectors, where a register holds multiple elements, each of the same data type. See *Vector formats* on page A1-41 for details.

The Armv8 architecture provides two register files:

- A general-purpose register file.
- A SIMD&FP register file.

In each of these, the possible register widths depend on the Execution state.

In AArch64 state:

- A general-purpose register file contains 64-bit registers:
  - Many instructions can access these registers as 64-bit registers or as 32-bit registers, using only the bottom 32 bits.
- A SIMD&FP register file contains 128-bit registers:
  - The quadword integer data types only apply to the SIMD&FP register file.
  - The floating-point data types only apply to the SIMD&FP register file.
  - While the AArch64 vector registers support 128-bit vectors, the effective vector length can be 64-bits or 128-bits depending on the A64 instruction encoding used, see *Instruction Mnemonics* on page C1-185.

For more information on the register files in AArch64 state, see *Registers in AArch64 Execution state* on page B1-107.

In AArch32 state:

- A general-purpose register file contains 32-bit registers:
  - Two 32-bit registers can support a doubleword.
  - Vector formatting is supported, see Figure A1-4 on page A1-44.
- A SIMD&FP register file contains 64-bit registers:
  - AArch32 state does not support quadword integer or floating-point data types.

  Note

  Two consecutive 64-bit registers can be used as a 128-bit register.

For more information on the register files in AArch32 state, see *The general-purpose registers, and the PC, in AArch32 state* on page E1-3951.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A1.4.1 Vector formats

In an implementation that includes the SIMD instructions that operate on the SIMD&FP register file, a register can hold one or more packed elements, all of the same size and type. The combination of a register and a data type describes a vector of elements. The vector is considered to be an array of elements of the data type specified in the instruction. The number of elements in the vector is implied by the size of the data elements and the size of the register.

Vector indices are in the range 0 to (number of elements – 1). An index of 0 refers to the least significant end of the vector.

Vector formats in AArch64 state

In AArch64 state, the SIMD&FP registers can be referred to as V*n*, where *n* is a value from 0 to 31.

The SIMD&FP registers support three data formats for loads, stores, and data-processing operations:

- A single, scalar, element in the least significant bits of the register.
- A 64-bit vector of byte, halfword, or word elements.
- A 128-bit vector of byte, halfword, word, or doubleword elements.

The element sizes are defined in Table A1-1 with the vector format described as:

- For a 128-bit vector: Vn{.2D, .4S, .8H, .16B}.
- For a 64-bit vector: Vn{.1D, .2S, .4H, .8B}.

Table A1-1 SIMD elements in AArch64 state

Mnemonic	Size
B	8 bits
H	16 bits
S	32 bits
D	64 bits

Figure A1-1 on page A1-42 shows the SIMD vectors in AArch64 state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

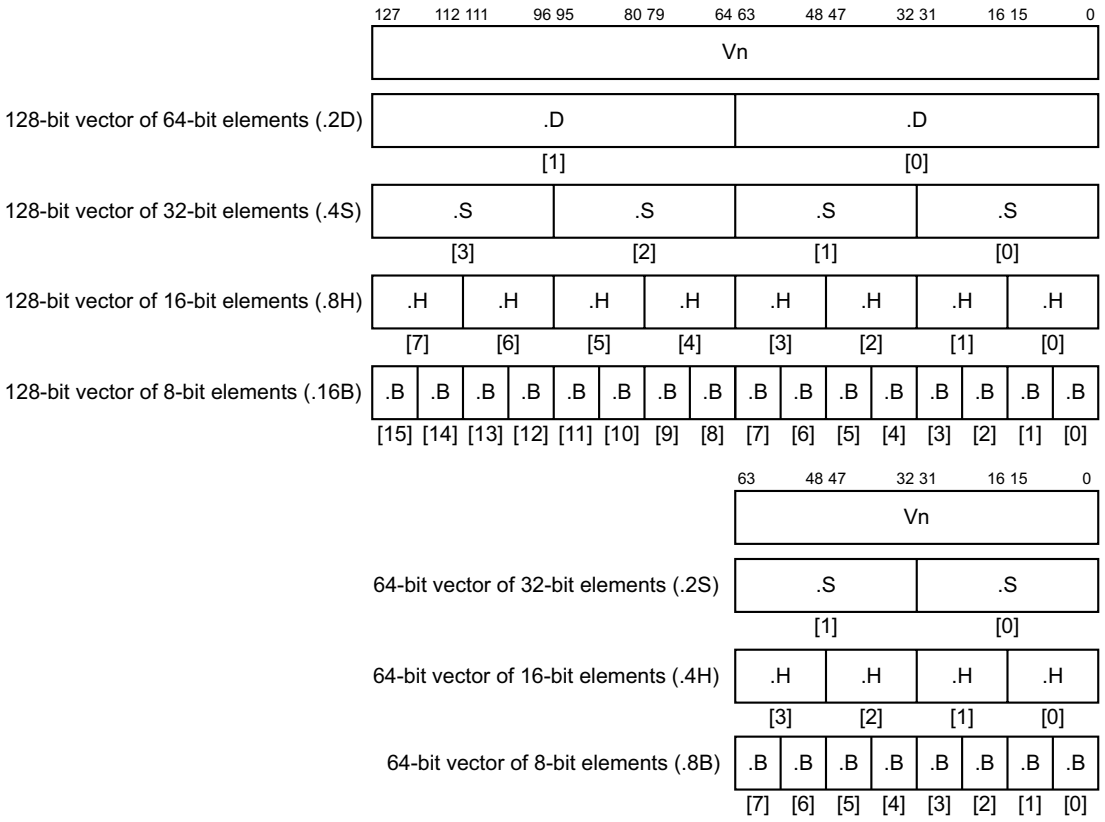


Figure A1-1 SIMD vectors in AArch64 state

Vector formats in AArch32 state

Table A1-2 shows the available formats. Each instruction description specifies the data types that the instruction supports.

Table A1-2 Advanced SIMD data types in AArch32 state

Data type specifier	Meaning
`.<size>`	Any element of `<size>` bits
`.F<size>`	Floating-point number of `<size>` bits
`.I<size>`	Signed or unsigned integer of `<size>` bits
`.P<size>`	Polynomial over {0, 1} of degree less than `<size>`
`.S<size>`	Signed integer of `<size>` bits
`.U<size>`	Unsigned integer of `<size>` bits

Polynomial arithmetic over {0, 1} on page A1-50 describes the polynomial data type.

The `.F16` data type is the half-precision data type selected by the FPSCR.AHP bit, see *Half-precision floating-point formats* on page A1-44.

The `.F32` data type is the Arm standard single-precision floating-point data type, see *Single-precision floating-point format* on page A1-46.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The instruction definitions use a data type specifier to define the data types appropriate to the operation. Figure A1-2 shows the hierarchy of the Advanced SIMD data types.

<table>
<tr><td rowspan="4">.8</td><td rowspan="2">.I8</td><td>.S8</td></tr>
<tr><td>.U8</td></tr>
<tr><td colspan="2">.P8</td></tr>
<tr><td colspan="2">-</td></tr>
<tr><td rowspan="4">.16</td><td rowspan="2">.I16</td><td>.S16</td></tr>
<tr><td>.U16</td></tr>
<tr><td colspan="2">.P16 †</td></tr>
<tr><td colspan="2">.F16</td></tr>
<tr><td rowspan="4">.32</td><td rowspan="2">.I32</td><td>.S32</td></tr>
<tr><td>.U32</td></tr>
<tr><td colspan="2">-</td></tr>
<tr><td colspan="2">.F32</td></tr>
<tr><td rowspan="4">.64</td><td rowspan="2">.I64</td><td>.S64</td></tr>
<tr><td>.U64</td></tr>
<tr><td colspan="2">.P64 ‡</td></tr>
<tr><td colspan="2">-</td></tr>
</table>

† Output format only. See `VMULL` instruction description.

‡ Available only if the Cyptographic Extension is implemented. See `VMULL` instruction description.

Figure A1-2 Advanced SIMD data type hierarchy in AArch32 state

For example, a multiply instruction must distinguish between integer and floating-point data types.

An integer multiply instruction that generates a double-width (long) result must specify the input data types as signed or unsigned. However, some integer multiply instructions use modulo arithmetic, and therefore do not have to distinguish between signed and unsigned inputs.

Figure A1-3 on page A1-44 shows the Advanced SIMD vectors in AArch32 state.

Note

In AArch32 state, a pair of even and following odd numbered doubleword registers can be concatenated and treated as a single quadword register.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

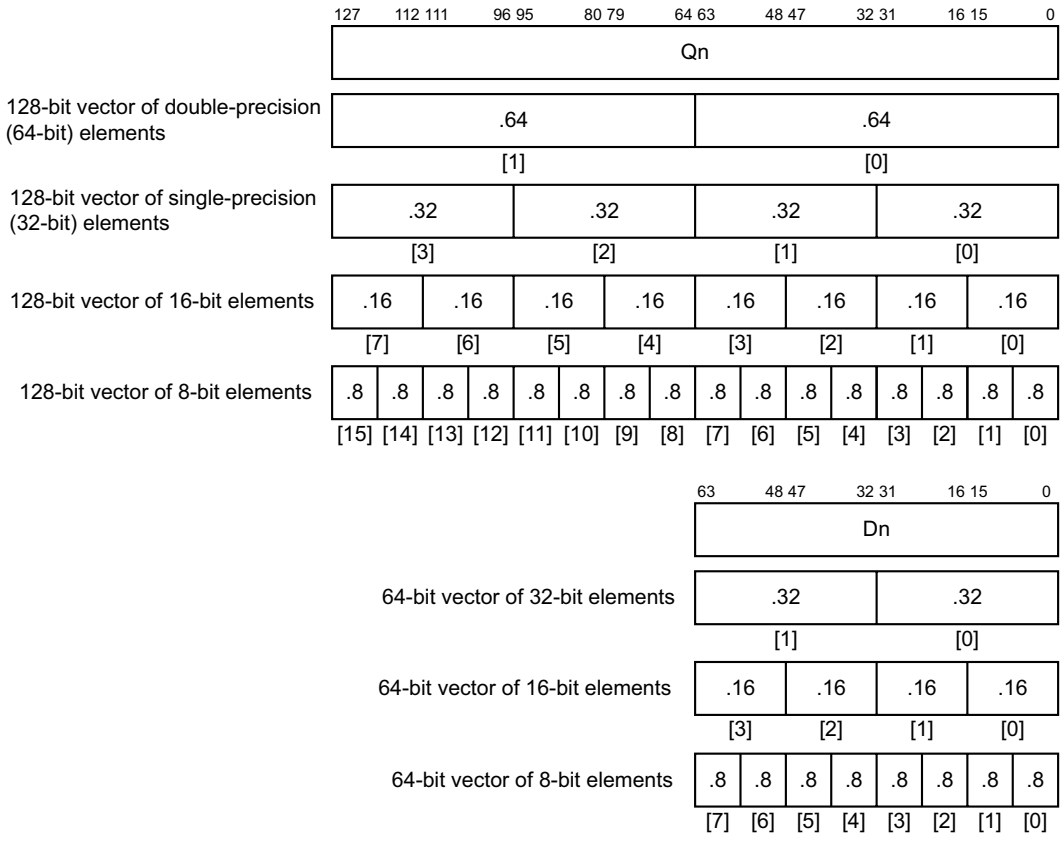


Figure A1-3 Advanced SIMD vectors in AArch32 state

The AArch32 general-purpose registers support vectors formats for use by the SIMD instructions in the Base instruction set. Figure A1-4 shows these formats, that means that a general-purpose register can be treated as either 2 halfwords or 4 bytes.

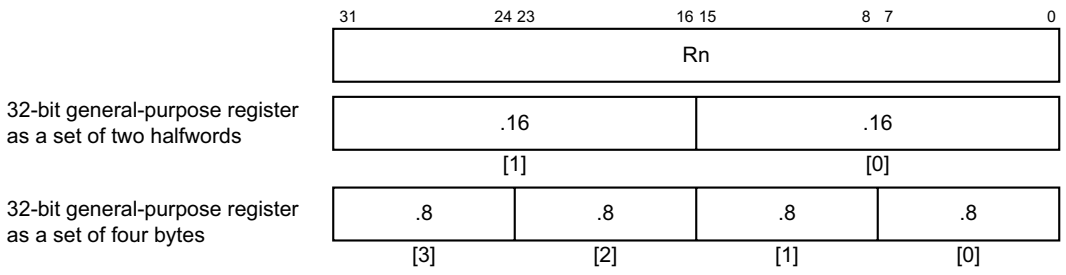


Figure A1-4 Vector formatting in AArch32 state

A1.4.2 Half-precision floating-point formats

Armv8 supports two half-precision floating-point formats:

- IEEE half-precision, as described in the IEEE 754-2008 standard.
- Arm *alternative half-precision* format.

Note

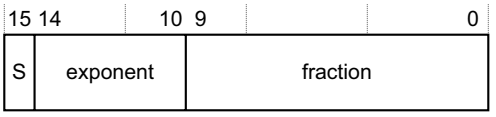
BFloat16 is not a half-precision floating-point format, see *BFloat16 floating-point format* on page A1-48.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Both formats can be used for conversions to and from other floating-point formats. FPCR.AHP controls the format in AArch64 state and FPSCR.AHP controls the format in AArch32 state. ARMv8.2-FP16 adds half-precision data-processing instructions, which always use the IEEE format. These instructions ignore the value of the relevant AHP field, and behave as if it has an *Effective value* of 0.

The description of IEEE half-precision includes Arm-specific details that are left open by the standard, and is only an introduction to the formats and to the values they can contain. For more information, especially on the handling of infinities, NaNs, and signed zeros, see the IEEE 754 standard.

For both half-precision floating-point formats, the layout of the 16-bit format is the same. The format is:



The interpretation of the format depends on the value of the exponent field, bits[14:10] and on which half-precision format is being used.

0 < exponent < `0x1F`

The value is a normalized number and is equal to:

$(-1)^S \times 2^{(exponent-15)} \times (1.fraction)$

The minimum positive normalized number is 2^{-14}, or approximately $6.104 \cdot 10^{-5}$.

The maximum positive normalized number is $(2 - 2^{-10}) \times 2^{15}$, or 65504.

Larger normalized numbers can be expressed using the alternative format when the exponent == `0x1F`.

exponent == 0

The value is either a zero or a denormalized number, depending on the fraction bits:

fraction == 0

The value is a zero. There are two distinct zeros:

+0 when S==0

–0 when S==1.

fraction != 0

The value is a denormalized number and is equal to:

$(-1)^S \times 2^{-14} \times (0.fraction)$

The minimum positive denormalized number is 2^{-24}, or approximately 5.960×10^{-8}.

Half-precision denormalized numbers are not flushed to zero by default. When ARMv8.2-FP16 is implemented, the FPCR.FZ16 bit controls whether Flush-to-Zero mode is enabled for half-precision data-processing instructions. For details, see *Flush-to-zero* on page A1-55.

exponent == `0x1F`

The value depends on which half-precision format is being used:

IEEE half-precision

The value is either an infinity or a Not a Number (NaN), depending on the fraction bits:

fraction == 0

The value is an infinity. There are two distinct infinities:

+infinity When S==0. This represents all positive numbers that are too big to be represented accurately as a normalized number.

-infinity When S==1. This represents all negative numbers with an absolute value that is too big to be represented accurately as a normalized number.

fraction != 0

The value is a NaN, and is either a *quiet NaN* or a *signaling NaN*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The two types of NaN are distinguished by their most significant fraction bit, bit[9]:

bit[9] == 0 The NaN is a signaling NaN. The sign bit can take any value, and the remaining fraction bits can take any value except all zeros.

bit[9] == 1 The NaN is a quiet NaN. The sign bit and remaining fraction bits can take any value.

Alternative half-precision

The value is a normalized number and is equal to:

$-1^{S} \times 2^{16} \times (1.\text{fraction})$

The maximum positive normalized number is $(2-2^{-10}) \times 2^{16}$ or 131008.

A1.4.3 Single-precision floating-point format

The single-precision floating-point format is as defined by the IEEE 754 standard.

This description includes Arm-specific details that are left open by the standard. It is only intended as an introduction to the formats and to the values they can contain. For full details, especially of the handling of infinities, NaNs, and signed zeros, see the IEEE 754 standard.

A single-precision value is a 32-bit word with the format:

31	30 … 23	22 … 0
S	exponent	fraction

The interpretation of the format depends on the value of the exponent field, bits[30:23]:

0 < exponent < `0xFF`

The value is a *normalized number* and is equal to:

$(-1)^{S} \times 2^{(\text{exponent} - 127)} \times (1.\text{fraction})$

The minimum positive normalized number is 2^{-126}, or approximately 1.175×10^{-38}.

The maximum positive normalized number is $(2 - 2^{-23}) \times 2^{127}$, or approximately 3.403×10^{38}.

exponent == 0

The value is either a zero or a *denormalized number*, depending on the fraction bits:

fraction == 0

The value is a zero. There are two distinct zeros:

+0 When S==0.

–0 When S==1.

These usually behave identically. In particular, the result is *equal* if +0 and –0 are compared as floating-point numbers. However, they yield different results in some circumstances. For example, the sign of the infinity produced as the result of dividing by zero depends on the sign of the zero. The two zeros can be distinguished from each other by performing an integer comparison of the two words.

fraction != 0

The value is a denormalized number and is equal to:

$(-1)^{S} \times 2^{-126} \times (0.\text{fraction})$

The minimum positive denormalized number is 2^{-149}, or approximately 1.401×10^{-45}.

Denormalized numbers are always flushed to zero in Advanced SIMD processing in AArch32 state. They are optionally flushed to zero in floating-point processing and in Advanced SIMD processing in AArch64 state. For details, see *Flush-to-zero* on page A1-55.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

exponent == `0xFF`

The value is either an *infinity* or a *Not a Number* (NaN), depending on the fraction bits:

fraction == 0

The value is an infinity. There are two distinct infinities:

+infinity When S==0. This represents all positive numbers that are too big to be represented accurately as a normalized number.

-infinity When S==1. This represents all negative numbers with an absolute value that is too big to be represented accurately as a normalized number.

fraction != 0

The value is a NaN, and is either a *quiet NaN* or a *signaling NaN*.

The two types of NaN are distinguished by their most significant fraction bit, bit[22]:

bit[22] == 0

The NaN is a signaling NaN. The sign bit can take any value, and the remaining fraction bits can take any value except all zeros.

bit[22] == 1

The NaN is a quiet NaN. The sign bit and remaining fraction bits can take any value.

For details of the *default NaN*, see *NaN handling and the Default NaN* on page A1-56.

Note

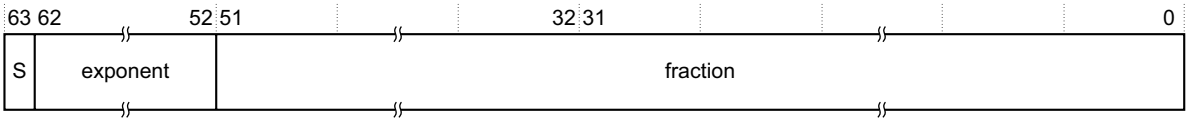
NaNs with different sign or fraction bits are distinct NaNs, but this does not mean software can use floating-point comparison instructions to distinguish them. This is because the IEEE 754 standard specifies that a NaN compares as *unordered* with everything, including itself.

A1.4.4 Double-precision floating-point format

The double-precision floating-point format is as defined by the IEEE 754 standard. Double-precision floating-point is supported by both SIMD and floating-point instructions in AArch64 state, and only by floating-point instructions in AArch32 state.

This description includes implementation-specific details that are left open by the standard. It is only intended as an introduction to the formats and to the values they can contain. For full details, especially of the handling of infinities, NaNs, and signed zeros, see the IEEE 754 standard.

A double-precision value is a 64-bit doubleword, with the format:



Double-precision values represent numbers, infinities, and NaNs in a similar way to single-precision values, with the interpretation of the format depending on the value of the exponent:

0 < exponent < `0x7FF`

The value is a normalized number and is equal to:

$(-1)^S \times 2^{(exponent-1023)} \times (1.fraction)$

The minimum positive normalized number is 2^{-1022}, or approximately 2.225×10^{-308}.

The maximum positive normalized number is $(2 - 2^{-52}) \times 2^{1023}$, or approximately 1.798×10^{308}.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

exponent == 0

The value is either a zero or a denormalized number, depending on the fraction bits:

fraction == 0

The value is a zero. There are two distinct zeros that behave in the same way as the two single-precision zeros:

+0 when S==0

–0 when S==1.

fraction != 0

The value is a denormalized number and is equal to:

$(-1)^S \times 2^{-1022} \times (0.\text{fraction})$

The minimum positive denormalized number is 2^{-1074}, or approximately 4.941×10^{-324}.

Optionally, denormalized numbers are flushed to zero in floating-point calculations. For details, see *Flush-to-zero* on page A1-55.

exponent == `0x7FF`

The value is either an infinity or a NaN, depending on the fraction bits:

fraction == 0

The value is an infinity. As for single-precision, there are two infinities:

+infinity When S==0.

-infinity When S==1.

fraction != 0

The value is a NaN, and is either a *quiet NaN* or a *signaling NaN*.

The two types of NaN are distinguished by their most significant fraction bit, bit[51] of the doubleword:

bit[51] == 0

The NaN is a signaling NaN. The sign bit can take any value, and the remaining fraction bits can take any value except all zeros.

bit[51] == 1

The NaN is a quiet NaN. The sign bit and the remaining fraction bits can take any value.

For details of the *default NaN*, see *NaN handling and the Default NaN* on page A1-56.

Note

NaNs with different sign or fraction bits are distinct NaNs, but this does not mean software can use floating-point comparison instructions to distinguish them. This is because the IEEE 754 standard specifies that a NaN compares as *unordered* with everything, including itself.

A1.4.5 BFloat16 floating-point format

BFloat16, or BF16, is a 16-bit floating-point storage format. The BF16 format inherits many of its properties and behaviors from the single-precision format defined by the IEEE754 standard, as described in *Single-precision floating-point format* on page A1-46.

For the BFloat16 floating-point format, the layout is:

15	14 … 7	6 … 0
S	exponent	fraction

0 < exponent < `0xFF`

The value is a normalized number and is equal to:

$(-1)^S \times 2^{(\text{exponent}-127)} \times (1.\text{fraction})$

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The minimum positive normalized number is 2^{-126}, or approximately $1.175 \cdot 10^{-38}$.

The maximum positive normalized number is $(2 - 2^{-7}) \times 2^{127}$, or approximately $3.390 \cdot 10^{38}$.

exponent == 0

The value is either a zero or a denormalized number, depending on the fraction bits:

fraction == 0

The value is a zero. There are two distinct zeros:

+0 when S==0

–0 when S==1.

These usually behave identically. However, they yield different results in some circumstances. For example, the sign of the result produced as the result of multiplying by zero depends on the sign of the zero. The two zeros can be distinguished from each other by performing an integer bitwise comparison of the two halfwords.

fraction != 0

The value is a denormalized number and is equal to:

$(-1)^S \times 2^{-126} \times (0.\text{fraction})$

The minimum positive denormalized number is 2^{-133}, or approximately 9.184×10^{-41}.

If *Flush-to-zero* on page A1-55 is enabled, for the conversion instructions that generate a BF16 result, a result will be flushed to zero if it satisfies the condition $0 < \texttt{Abs(result)} < 2^{-126}$.

Denormalized numbers are unconditionally flushed to zero by the BF16 arithmetic instructions, and by Advanced SIMD floating-point instructions in AArch32 state. They might be flushed to zero by other floating-point instructions, see *Flush-to-zero* on page A1-55.

For the conversion instructions that generate a BF16 result, the Flush-to-zero mode is enabled by the FPCR.FZ bit in AArch64 state and the FPSCR.FZ bit in AArch32 state.

exponent == `0xFF`

The value is either an infinity or a Not a Number (NaN), depending on the fraction bits:

fraction == 0

The value is an infinity. There are two distinct infinities:

+infinity When S==0. This represents all positive numbers that are too big to be represented accurately as a normalized number.

-infinity When S==1. This represents all negative numbers with an absolute value that is too big to be represented accurately as a normalized number.

fraction != 0

The value is a NaN, and is either a *quiet NaN* or a *signaling NaN*.

The two types of NaN are distinguished by their most significant fraction bit, bit[6]:

bit[6] == 0 The NaN is a signaling NaN. The sign bit can take any value, and the remaining fraction bits can take any value except all zeros.

bit[6] == 1 The NaN is a quiet NaN. The sign bit and remaining fraction bits can take any value.

In the arithmetic instructions that accept BF16 inputs, there is no distinction between quiet and signalling input NaNs, since these instructions cannot signal a floating-point exception, and any type of input NaN generates the same Default NaN result.

If Default NaN mode is enabled, for the conversion instructions that generate a BF16 result, a BF16 Default NaN result is encoded as S=0, exponent =`0xFF`, fraction= `0x40` (bit[6] == 1).

BF16 values are 16-bit half-words that software can convert to single-precision format, by appending 16 zero bits, so that single-precision arithmetic instructions can be used. A single-precision value can be converted to BF16 format if required, either by:

- Truncating, by removing the least significant 16 bits.
- Using the BFloat16 conversion instructions, see *BFloat16 floating-point instructions* on page C3-246.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A1.4.6 Fixed-point format

Fixed-point formats are used only for conversions between floating-point and fixed-point values. They apply to general-purpose registers.

Fixed-point values can be signed or unsigned, and can be 16-bit or 32-bit. Conversion instructions take an argument that specifies the number of fraction bits in the fixed-point number. That is, it specifies the position of the binary point.

A1.4.7 Conversion between floating-point and fixed-point values

Armv8 supports the conversion of a scalar floating-point to or from a signed or unsigned fixed-point value in a general-purpose register.

The instruction argument #fbits indicates that the general-purpose register holds a fixed-point number with fbits bits after the binary point, where fbits is in the range 1 to 64 for a 64-bit general-purpose register, or 1 to 32 for a 32-bit general-purpose register.

More specifically:

- For a 64-bit register X_d:
  - The integer part is X_d[63:#fbits].
  - The fractional part is X_d[(#fbits-1):0].
- For a 32-bit register W_d or R_d:
  - The integer part is W_d[31:#fbits] or R_d[31:#fbits].
  - The fractional part is W_d[(#fbits-1):0] or R_d[(#fbits-1):0].

These instructions can cause the following floating-point exceptions:

Invalid Operation When the floating-point input is NaN or Infinity or when a numerical value cannot be represented within the destination register.

Inexact When the numeric result differs from the input value.

Input Denormal When Flush-to-zero mode is enabled and the denormal input is replaced by a zero.

Note

An out of range fixed-point result is saturated to the destination size.

For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

A1.4.8 Polynomial arithmetic over {0, 1}

Some SIMD instructions that operate on SIMD&FP registers can operate on polynomials over {0, 1}, see *Supported data types* on page A1-40. The polynomial data type represents a polynomial in x of the form $b_{n-1}x^{n-1} + \ldots + b_1x + b_0$ where b_k is bit[k] of the value.

The coefficients 0 and 1 are manipulated using the rules of Boolean arithmetic:

- $0 + 0 = 1 + 1 = 0$
- $0 + 1 = 1 + 0 = 1$
- $0 \times 0 = 0 \times 1 = 1 \times 0 = 0$
- $1 \times 1 = 1$.

That is:

- Adding two polynomials over {0, 1} is the same as a bitwise exclusive OR.
- Multiplying two polynomials over {0, 1} is the same as integer multiplication except that partial products are exclusive-ORed instead of being added.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A64, A32, and T32 provide instructions for performing polynomial multiplication of 8-bit values.

- For AArch32, see *VMUL (integer and polynomial)* on page F6-5355 and *VMULL (integer and polynomial)* on page F6-5361.
- For AArch64, see *PMUL* on page C7-1886 and *PMULL, PMULL2* on page C7-1888.

The Cryptographic Extension adds the ability to perform long polynomial multiplies of 64-bit values. See *PMULL, PMULL2* on page C7-1888.

Pseudocode description of polynomial multiplication

In pseudocode, polynomial addition is described by the EOR operation on bitstrings.

Polynomial multiplication is described by the `PolynomialMult()` function defined in Chapter J1 *Armv8 Pseudocode*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A1.5 Advanced SIMD and floating-point support

Note

In AArch32 state, the SIMD instructions that operate on SIMD&FP registers are always described as the Advanced SIMD instructions, to distinguish them from the SIMD instructions in the base instruction sets, that operate on the 32-bit general-purpose registers. The A64 instruction set does not provide any SIMD instructions that operate on the general-purpose registers, and therefore some AArch64 state descriptions use SIMD as a synonym for Advanced SIMD. Unless the context clearly indicates otherwise, this section describes the support for SIMD instructions that operate on SIMD&FP registers.

Armv8 can support the following levels of support for Advanced SIMD and floating-point instructions:

- Full SIMD and floating-point support without exception trapping.
- Full SIMD and floating-point support with exception trapping.
- No floating-point or SIMD support. This option is licensed only for implementations targeting specialized markets.

  Note

  All systems that support standard operating systems with rich application environments provide hardware support for Advanced SIMD and floating-point. It is a requirement of the ARM Procedure Call Standard for AArch64, see *Procedure Call Standard for the Arm 64-bit Architecture*.

Armv8 supports single-precision (32-bit) and double-precision (64-bit) floating-point data types and arithmetic as defined by the IEEE 754 floating-point standard. It also supports the half-precision (16-bit) floating-point data type for data storage, by supporting conversions between single-precision and half-precision data types and double-precision and half-precision data types. When ARMv8.2-FP16 is implemented, it also supports the half-precision floating-point data type for data-processing operations.

The SIMD instructions provide packed *Single Instruction Multiple Data (SIMD)* and single-element scalar operations, and support:

- Single-precision and double-precision arithmetic in AArch64 state.
- Single-precision arithmetic only in AArch32 state.
- When ARMv8.2-FP16 is implemented, half-precision arithmetic is supported in AArch64 and AArch32 states.

Floating-point support in AArch64 state SIMD is IEEE 754-2008 compliant with:

- Configurable rounding modes.
- Configurable Default NaN behavior.
- Configurable Flush-to-zero behavior.

Floating-point computation using AArch32 Advanced SIMD instructions remains unchanged from Armv7. A32 and T32 Advanced SIMD floating-point always uses Arm standard floating-point arithmetic and performs IEEE 754 floating-point arithmetic with the following restrictions:

- Denormalized numbers are flushed to zero, see *Flush-to-zero* on page A1-55.
- Only default NaNs are supported, see *NaN handling and the Default NaN* on page A1-56.
- The Round to Nearest rounding mode is used.
- Untrapped floating-point exception handling is used for all floating-point exceptions.

If floating-point exception trapping is supported, floating-point exceptions, such as Overflow or Divide by Zero, can be handled without trapping. This applies to both SIMD and floating-point operations. When handled in this way, a floating-point exception causes a cumulative status register bit to be set to 1 and a default result to be produced by the operation. For more information about floating-point exceptions, see *Floating-point exceptions and exception traps* on page D1-2360.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In AArch64 state, the following registers control floating-point operation and return floating-point status information:

- The Floating-Point Control Register, FPCR, controls:
  - The half-precision format where applicable, FPCR.AHP bit.
  - Default NaN behavior, FPCR.DN bit.
  - Flush-to-zero behavior, FPCR.{FZ, FZ16} bits. If ARMv8.2-FP16 is not implemented, FPCR.FZ16 is RES0.
  - Rounding mode support, FPCR.Rmode field.
  - Len and Stride fields associated with execution in AArch32 state, and only supported for a context save and restore from AArch64 state. These fields are obsolete in Armv8 and can be implemented as RAZ/WI. If they are implemented as RW and are programmed to a nonzero value, they make some AArch32 floating-point instructions UNDEFINED.
  - Floating-point exception trap controls, the FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} bits, see *Floating-point exceptions and exception traps* on page D1-2360.
- The Floating-Point Status Register, FPSR, provides:
  - Cumulative floating-point exceptions flags, FPSR.{IDC, IXC, UFC, OFC, DZC, IOC and QC}.
  - The AArch32 floating-point comparison flags {N,Z,C,V}. These bits are RES0 if AArch32 floating-point is not implemented.

    Note

    In AArch64 state, the process state flags, PSTATE.{N,Z,C,V} are used for all data-processing compares and any associated conditional execution.

    If ARMv8.5-CondM is implemented, the instructions `AXFLAG` and `XAFLAG` convert between the Arm condition flag format and an alternative format shown in *Relationship between ARM format and alternative format PSTATE condition flags* on page C6-774.

AArch32 state provides a single Floating-Point Status and Control Register, FPSCR, combining the FPCR and FPSR fields.

For system level information about the SIMD and floating-point support, see *Advanced SIMD and floating-point support* on page G1-5772.

A1.5.1 Instruction support

The Advanced SIMD and floating-point instructions support:

- Load and store for single elements and vectors of multiple elements.

  Note

  Single elements are also referred to as scalar elements.

- Data processing on single and multiple elements for both integer and floating-point data types.
- When ARMv8.3-CompNum is implemented, complex number arithmetic.
- Floating-point conversion between different levels of precision.
- Conversion between floating-point, fixed-point integer, and integer data types.
- Floating-point rounding.

For more information on the SIMD and floating-point instructions in AArch64 state, see Chapter C3 *A64 Instruction Set Overview*.

For more information on the Advanced SIMD and floating-point instructions in AArch32 state, see Chapter F1 *The AArch32 Instruction Sets Overview*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A1.5.2 Floating-point standards, and terminology

The Arm includes support for all the required features of ANSI/IEEE Std 754-2008, *IEEE Standard for Binary Floating-Point Arithmetic*, referred to as IEEE 754-2008. However, some terms in this manual are based on the 1985 version of this standard, referred to as IEEE 754-1985:

- Arm floating-point terminology generally uses the IEEE 754-1985 terms. This section summarizes how IEEE 754-2008 changes these terms.
- References to IEEE 754 that do not include the issue year apply to either issue of the standard.

Table A1-3 shows how the terminology in this manual differs from that used in IEEE 754-2008.

Table A1-3 Floating-point terminology

This manual	IEEE 754-2008
Normalized [a]	Normal
Denormal, or denormalized	Subnormal
Round towards Minus Infinity (RM)	roundTowardsNegative
Round towards Plus Infinity (RP)	roundTowardsPositive
Round towards Zero (RZ)	roundTowardZero
Round to Nearest (RN)	roundTiesToEven
Round to Nearest with Ties to Away	roundTiesToAway
Rounding mode	Rounding-direction attribute

a. *Normalized number* is used in preference to *normal number*, because of the other specific uses of *normal* in this manual.

A1.5.3 Arm standard floating-point input and output values

Armv8 provides full IEEE 754 floating-point arithmetic support. In AArch32 state, floating-point operations performed using Advanced SIMD instructions are limited to *Arm standard floating-point operation*, regardless of the selected rounding mode in the FPSCR. Unlike AArch32, AArch64 SIMD floating point arithmetic is performed using the rounding mode selected by the FPCR.

Arm standard floating-point arithmetic supports the following input formats defined by the IEEE 754 floating-point standard:

- Zeros.
- Normalized numbers.
- Denormalized numbers are flushed to 0 before floating-point operations, see *Flush-to-zero* on page A1-55.
- NaNs.
- Infinities.

Arm standard floating-point arithmetic supports the Round to Nearest (roundTiesToEven) rounding mode defined by the IEEE 754 standard.

Arm standard floating-point arithmetic supports the following output result formats defined by the IEEE 754 standard:

- Zeros.
- Normalized numbers.
- Results that are less than the minimum normalized number are flushed to zero, see *Flush-to-zero* on page A1-55.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- NaNs produced in floating-point operations are always the default NaN, see *NaN handling and the Default NaN* on page A1-56.
- Infinities.

A1.5.4 Flush-to-zero

The performance of floating-point processing can be reduced when doing calculations involving denormalized numbers and Underflow exceptions. In many algorithms, this performance can be recovered, without significantly affecting the accuracy of the final result, by replacing the denormalized operands and intermediate results with zeros. To permit this optimization, Arm floating-point implementations allow a Flush-to-zero mode to be used for different floating-point formats as follows:

For AArch64:

- If FPCR.FZ==1, then Flush-to-Zero mode is used for all Single-Precision and Double-Precision inputs and outputs of all instructions. When ARMv8.2-BF16 is implemented, Flush-to-Zero mode is used for all Single-Precision and Double-Precision inputs and outputs of all instructions, and conversion instructions that generate a BF16 result.
- If FPCR.FZ16==1, then Flush-to-Zero mode is used for all Half-Precision inputs and outputs of floating-point instructions, other than:
  - Conversions between Half-Precision and Single-Precision numbers.
  - Conversions between Half-Precision and Double-Precision numbers.

For AArch32:

- If FPSCR.FZ==0, then Flush-to-Zero mode is used for all Single-Precision and Double-Precision inputs and outputs of all Advanced SIMD floating-point instructions.
- If FPSCR.FZ==1, then Flush-to-Zero mode is used for all Single-Precision and Double-Precision inputs and outputs of all instructions. When ARMv8.2-AA32BF16 is implemented, Flush-to-Zero mode is used for all Single-Precision and Double-Precision inputs and outputs of all instructions, and conversion instructions that generate a BF16 result.
- If FPSCR.FZ16==1, then Flush-to-Zero mode is used for all Half-Precision inputs and outputs of floating-point instructions, other than:
  - Conversions between Half-Precision and Single-Precision numbers.
  - Conversions between Half-Precision and Double-Precision numbers.

If Flush-To-Zero mode is used on an Single-precision or Double-precision input:

- All inputs to floating-point operations that are denormalized numbers in their represented precision are treated as though they were zero with the same sign as the input, and an Input Denormal floating-point exception is generated.

  Note

  The Input Denormal floating-point exception occurs only in Flush-to-zero mode.

- In AArch32 state, the FPSCR contains a cumulative exception bit FPSCR.IDC and optional trap enable bit FPSCR.IDE corresponding to the Input Denormal floating-point exception.
- In AArch64 state, the FPSR contains a cumulative exception bit FPSR.IDC and optional trap enable bit FPCR.IDE corresponding to the Input Denormal floating-point exception.
- The occurrence of all floating-point exceptions except Input Denormal is determined using the input values that are treated as zero by this mechanism.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If Flush-To-Zero mode is used on a Half-precision input:

- All inputs to floating-point operations that are denormalized numbers in their represented precision are treated as though they were zero with the same sign as the input.

  Note

  When ARMv8.2-FP16 is implemented, when in Flush-to-zero mode, a half-precision floating-point number that is flushed to zero does not generate an Input Denormal floating-point exception. This is because this situation is much less exceptional than for double-precision or single-precision denormalized numbers.

- The occurrence of all floating-point exceptions is determined using the input values that are treated as zero by this mechanism.

If Flush-To-Zero mode is used on any output of an instruction:

- The output is returned as zero, with the same sign bit as the result, if the result before rounding of the operation specified by the instruction satisfies the condition:

  0 < Abs(result) < MinNorm, where:

  — MinNorm is 2^-14 for half-precision.

  — MinNorm is 2^-126 for single-precision.

  — MinNorm is 2^-1022 for double-precision.

  If this occurs, then:

  — An Underflow Exception is generated, but in all implementations, the Underflow Exception is not trapped even if the AArch32 FPSCR.UFE==1 or the AArch64 FPCR.UFE==1.

  — An Inexact Exception is not generated.

Note

Flush-to-zero mode is incompatible with the IEEE 754 standard, and must not be used when IEEE 754 compatibility is a requirement. Flush-to-zero mode must be used with care. Although it can improve performance on some algorithms, there are significant limitations on its use. These are application dependent:

- On many algorithms, it has no noticeable effect, because the algorithm does not normally use denormalized numbers.
- On other algorithms, it can cause exceptions to occur or seriously reduce the accuracy of the results of the algorithm.

A1.5.5 NaN handling and the Default NaN

The IEEE 754 standard specifies that:

- An operation that causes an Invalid Operation floating-point exception generates a quiet NaN as its result if that exception is untrapped.
- An operation involving a quiet NaN operand, but not a signaling NaN operand, returns an input NaN as its result.

The floating-point processing behavior when Default NaN mode is disabled adheres to this, with the following additions:

- If an untrapped Invalid Operation floating-point exception occurs, the quiet NaN result is derived from:

  — The first signaling NaN operand, if the exception occurs because at least one of the operands is a signaling NaN.

  — Otherwise, the default NaN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If an untrapped Invalid Operation floating-point exception does not occur, but at least one of the operands is a quiet NaN, the result is derived from the first quiet NaN operand.

Depending on the operation, the exact value of a derived quiet NaN result may differ in both sign and number of fraction bits from its source. For a quiet NaN result derived from signaling NaN operand, the most-significant fraction bit is set to 1.

Note

- In these descriptions, *first operand* relates to the left-to-right ordering of the arguments to the pseudocode function that describes the operation.
- The IEEE 754 standard specifies that the sign bit of a NaN has no significance.

The SIMD and floating-point processing behavior when Default NaN mode is enabled is that the Default NaN is the result of all floating-point operations that either:

- Cause untrapped Invalid Operation floating-point exceptions.
- Have one or more quiet NaN inputs, but no signaling NaN inputs.

Table A1-4 shows the format of the default NaN for Arm floating-point operations.

Default NaN mode is selected for the floating-point processing by setting the FPCR.DN bit to 1.

Other aspects of the functionality of the Invalid Operation floating-point exception are not affected by Default NaN mode. These are that:

- If untrapped, it causes the FPSR.IOC bit to be set to 1.
- If trapped, it causes a user trap handler to be invoked.

Table A1-4 Default NaN encoding

	Half-precision, IEEE Format	Single-precision	Double-precision
Sign bit	0	0	0
Exponent	`0x1F`	`0xFF`	`0x7FF`
Fraction	Bit[9] == 1, bits[8:0] == 0	Bit[22] == 1, bits[21:0] == 0	Bit[51] == 1, bits[50:0] == 0

A1.5.6 Round to Odd mode

Round to Odd mode is not defined by IEEE 754, and differs between the `FCVTXN` instruction and the BFloat16 instructions.

Round to Odd mode in FCVTXN

The `FCVTXN` instructions use Round to Odd rounding mode to ensure that if the result of the conversion is inexact the least significant bit of the mantissa is forced to 1.

This rounding mode enables a floating-point value to be converted to a lower precision format via an intermediate precision format while avoiding double rounding errors. For example, a 64-bit floating-point value can be converted to a correctly rounded 16-bit floating-point value by first using this instruction to produce a 32-bit value and then using another instruction with the required rounding mode to convert the 32-bit value to the final 16-bit floating-point value.

Round to Odd mode in BFloat16

When ARMv8.2-BF16 or ARMv8.2-AA32BF16 are implemented, then BFloat16 instructions are implemented. The BFloat16 instructions use Round to Odd mode, sometimes known as sticky rounding, to avoid double rounding errors when converting a floating-point value to a lower precision format in two steps, if the intermediate format has at least two more bits of precision than the result format.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When used, the effect of Round to Odd mode depends on the rounded value:

- When the rounded value is inexact, the least significant bit of the fraction is set to 1.
- When the rounded value has a magnitude that is too large to represent in the single-precision format, then an appropriately signed single-precision Infinity is generated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A1.6 The Arm memory model

The Arm memory model supports:

- Generating an exception on an unaligned memory access.
- Restricting access by applications to specified areas of memory.
- Translating *virtual addresses* (VAs) provided by executing instructions to *physical addresses* (PAs).
- Altering the interpretation of multi-byte data between big-endian and little-endian.
- Controlling the order of accesses to memory.
- Controlling caches and address translation structures.
- Synchronizing access to shared memory by multiple PEs.
- Barriers that control and prevent speculative access to memory.

VA support depends on the Execution state, as follows:

AArch64 state

Supports 64-bit virtual addressing, with the Translation Control Register determining the supported VA range. Execution at EL1 and EL0 supports two independent VA ranges, each with its own translation controls.

AArch32 state

Supports 32-bit virtual addressing, with the Translation Control Register determining the supported VA range. For execution at EL1 and EL0, system software can split the VA range into two subranges, each with its own translation controls.

The supported PA space is IMPLEMENTATION DEFINED, and can be discovered by system software.

Regardless of the Execution state, the *Virtual Memory System Architecture* (VMSA) can translate VAs to blocks or pages of memory anywhere within the supported PA space.

For more information, see:

For execution in AArch64 state

- Chapter B2 *The AArch64 Application Level Memory Model*.
- Chapter D4 *The AArch64 System Level Memory Model*.
- Chapter D5 *The AArch64 Virtual Memory System Architecture*.

For execution in AArch32 state

- Chapter E2 *The AArch32 Application Level Memory Model*.
- Chapter G4 *The AArch32 System Level Memory Model*.
- Chapter G5 *The AArch32 Virtual Memory System Architecture*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter A2
Armv8-A Architecture Extensions

This chapter introduces the Arm architecture versions and extensions. It contains the following sections:

- *Armv8 architecture extensions* on page A2-62.
- *Architectural features within Armv8.0 architecture* on page A2-65.
- *The Armv8 Cryptographic Extension* on page A2-68.
- *The Armv8.1 architecture extension* on page A2-69.
- *The Armv8.2 architecture extension* on page A2-72.
- *The Armv8.3 architecture extension* on page A2-81.
- *The Armv8.4 architecture extension* on page A2-85.
- *The Armv8.5 architecture extension* on page A2-90.
- *The Armv8.6 architecture extension* on page A2-94.
- *The Reliability, Availability, and Serviceability Extension (RAS Extension)* on page A2-97.
- *The Statistical Profiling Extension (SPE)* on page A2-98.
- *The Scalable Vector Extension (SVE)* on page A2-99.
- *The Activity Monitors Extension* on page A2-100.
- *The Memory Partitioning and Monitoring (MPAM) Extension* on page A2-101.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A2.1 Armv8 architecture extensions

The original Armv8-A architecture is called Armv8.0. The following sections of this manual describe or summarize permitted extensions to Armv8.0:

- *The Armv8 Cryptographic Extension* on page A2-68.
- *The Reliability, Availability, and Serviceability Extension (RAS Extension)* on page A2-97.
- *Event monitors* on page D1-2405.
- *The IVIPT Extension* on page D5-2687.
- Chapter H7 *The PC Sample-based Profiling Extension*.

In addition to describing Armv8.0, this manual describes the following architectural extensions:

Features added to Armv8.0 in later releases

Architectural features and architectural requirements have been added to the original Armv8-A architecture. For more information, see:

- *Additional functionality added to Armv8.0 in later releases* on page A2-65.
- *Architectural requirements within Armv8.0 architecture* on page A2-67.

For more information, see *Architectural features within Armv8.0 architecture* on page A2-65.

The Armv8.1 architectural extension

The Armv8.1 architecture extension adds both:

- Architectural features. Some of these are mandatory, others are optional. Some features must be implemented together.
- Architectural requirements. These are mandatory.

An implementation is Armv8.1 compliant when all of the following apply:

- It includes all of the Armv8.1 architectural features that are mandatory. This includes all architectural features of an optional architecture component or extension that are defined as mandatory, if the Armv8.1 compliant implementation includes the optional architecture component or extension. See *Architectural features added by Armv8.1* on page A2-69 for all of the Armv8.1 architectural features.
- It includes all of the Armv8.1 architectural requirements. *Additional requirements of Armv8.1* on page A2-71 lists these requirements.

For more information, see *The Armv8.1 architecture extension* on page A2-69.

The Armv8.2 architectural extension

The Armv8.2 architecture extension is an extension to Armv8.1. It adds both:

- Architectural features. Some of these are mandatory, others are optional. Some features must be implemented together.
- Architectural requirements. These are mandatory.

An implementation is Armv8.2 compliant if all of the following apply:

- It is Armv8.1 compliant.
- It includes all of the Armv8.2 architectural features that are mandatory. This includes all architectural features of an optional architecture component or extension that are defined as mandatory, if the Armv8.2 compliant implementation includes the optional architecture component or extension. The features are listed at:
  - *Architectural features added by Armv8.2* on page A2-72, which lists the original Armv8.2 architectural features.
  - *Features added to the Armv8.2 extension in later releases* on page A2-78, which lists additional Armv8.2 architectural features.
- It includes all of the Armv8.2 architectural requirements. *Additional requirements of Armv8.2* on page A2-78 lists these requirements.

For more information, see *The Armv8.2 architecture extension* on page A2-72.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The Armv8.3 architectural extension

The Armv8.3 architecture extension is an extension to Armv8.2. It adds both:

- Architectural features. Some of these are mandatory, others are optional. Some features must be implemented together.
- Architectural requirements. These are mandatory.

An implementation is Armv8.3 compliant if all of the following apply:

- It is Armv8.2 compliant.
- It includes all of the Armv8.3 architectural features that are mandatory. This includes all architectural features of an optional architecture component or extension that are defined as mandatory, if the Armv8.3 compliant implementation includes the optional architecture component or extension. The features are listed at:
  - *Architectural features added by Armv8.3* on page A2-81, which lists the original Armv8.3 architectural features.
  - *Features added to the Armv8.3 extension in later releases* on page A2-83, which lists additional Armv8.3 architectural features.
- It includes all of the Armv8.3 architectural requirements. *Additional requirements of Armv8.3* on page A2-83 lists these requirements.

For more information, see *The Armv8.3 architecture extension* on page A2-81.

The Armv8.4 architectural extension

The Armv8.4 architecture extension is an extension to Armv8.3. It adds architectural features. Some of these are mandatory, others are optional. Some features must be implemented together.

An implementation is Armv8.4 compliant if all of the following apply:

- It is Armv8.3 compliant.
- It includes all of the Armv8.4 architectural features that are mandatory. This includes all architectural features of an optional architecture component or extension that are defined as mandatory, if the Armv8.4 compliant implementation includes the optional architecture component or extension. See *Architectural features added by Armv8.4* on page A2-85 for all of the Armv8.4 architectural features.

For more information, see *The Armv8.4 architecture extension* on page A2-85.

The Armv8.5 architectural extension

The Armv8.5 architecture extension is an extension to Armv8.4. It adds architectural features. Some of these are mandatory, others are optional. Some features must be implemented together.

An implementation is Armv8.5 compliant if all of the following apply:

- It is Armv8.4 compliant.
- It includes all of the Armv8.5 architectural features that are mandatory. This includes all architectural features of an optional architecture component or extension that are defined as mandatory, if the Armv8.5 compliant implementation includes the optional architecture component or extension. See *Architectural features added by Armv8.5* on page A2-90 for all of the Armv8.5 architectural features.
- It includes all of the Armv8.5 architectural requirements. *Additional requirements of Armv8.5* on page A2-92 lists these requirements.

For more information, see *The Armv8.5 architecture extension* on page A2-90.

The Armv8.6 architectural extension

The Armv8.6 architecture extension is an extension to Armv8.5. It adds architectural features. Some of these are mandatory, others are optional. Some features must be implemented together.

An implementation is Armv8.6 compliant if all of the following apply:

- It is Armv8.5 compliant.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- It includes all of the Armv8.6 architectural features that are mandatory. This includes all architectural features of an optional architecture component or extension that are defined as mandatory, if the Armv8.6 compliant implementation includes the optional architecture component or extension. See *Architectural features added by Armv8.6* on page A2-94 for all of the Armv8.6 architectural features.
- It includes all of the Armv8.6 architectural requirements. *Additional requirements of Armv8.6* on page A2-95 lists these requirements.

For more information, see *The Armv8.6 architecture extension* on page A2-94.

The Statistical Profiling Extension (SPE)

SPE is an optional extension to Armv8.2. That is, SPE requires the implementation of Armv8.2.

For more information, see *The Statistical Profiling Extension (SPE)* on page A2-98.

The Scalable Vector Extension (SVE)

SVE is an optional extension to Armv8.2. That is, SVE requires the implementation of Armv8.2.

For more information, see *The Scalable Vector Extension (SVE)* on page A2-99.

The Activity Monitors Extension (AMU)

AMU is an optional extension to Armv8.4. That is, AMU requires the implementation of Armv8.4.

For more information, see *The Activity Monitors Extension* on page A2-100.

The Memory Partitioning and Monitoring Extension (MPAM)

MPAM is an optional extension to Armv8.2. That is, MPAM requires the implementation of Armv8.2.

For more information, see *The Memory Partitioning and Monitoring (MPAM) Extension* on page A2-101.

See also *Permitted implementation of subsets of Armv8.x and Armv8.(x+1) architectural features*.

A2.1.1 Permitted implementation of subsets of Armv8.x and Armv8.(x+1) architectural features

An Armv8.*x* compliant implementation can include any arbitrary subset of the architectural features of Armv8.(*x*+1), subject only to those constraints that require that certain features be implemented together.

Unless this manual permits otherwise, an Armv8.*x* compliant implementation does not include any features of Armv8.(*x*+2) or later.

Note

The addition of Armv8.(*x*+1) features to an Armv8.*x* compliant implementation is only permitted if the implementer has a license to Armv8.(*x*+1) in addition to the license to Armv8.*x*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A2.2 Architectural features within Armv8.0 architecture

This includes architectural features and architectural requirements that have been added to the Armv8.0 architecture since the initial release, that were not part of the original Armv8-A architecture, see:

- *Additional functionality added to Armv8.0 in later releases*.
- *Architectural requirements within Armv8.0 architecture* on page A2-67.

A2.2.1 Additional functionality added to Armv8.0 in later releases

An implementation of Armv8.0 can include any or all of the features that this section describes.

The Armv8.0 architecture extension adds the following architectural features, which are identified by the architectural feature name and a short description of the feature:

ARMv8.0-SB, Armv8.0 Speculation Barrier

ARMv8.0-SB introduces a barrier to control speculation.

This instruction is supported in both AArch64 and AArch32 states.

This feature is OPTIONAL in Armv8.0 implementations and mandatory in Armv8.5 implementations.

The following fields identify the presence of ARMv8.0-SB:

- ID_AA64ISAR1_EL1.SB
- ID_ISAR6_EL1.SB
- ID_ISAR6.SB

For more information, see:

- *Speculation Barrier (SB)* on page B2-136.
- *Barriers and CLREX instructions* on page C3-205.
- *Speculation Barrier (SB)* on page E2-3993.
- *Miscellaneous instructions* on page F1-4053.

ARMv8.0-SSBS, Armv8.0 Speculative Store Bypass Safe

ARMv8.0-SSBS allows software to indicate whether hardware is permitted to load or store speculatively in a manner that could give rise to a cache timing side channel, which in turn could be used to derive an address from values loaded to a register from memory. To do this, the software sets the PSTATE.SSBS.

This feature is supported in both AArch64 and AArch32 states.

This feature is OPTIONAL in Armv8.0 implementations and mandatory in Armv8.5 implementations.

The following fields identify the presence of ARMv8.0-SSBS:

- ID_AA64PFR1_EL1.SSBS
- ID_PFR2_EL1.SSBS
- ID_PFR2.SSBS

For more information, see:

- *Speculative Store Bypass Safe (SSBS)* on page B2-133.
- *Speculative Store Bypass Safe (SSBS)* on page E2-3990.

ARMv8.0-CSV2, Armv8.0 Cache Speculation Variant 2

Armv8.0-CSV2 adds a mechanism to identify if hardware cannot disclose information about whether branch targets trained in one hardware described context can affect speculative execution in a different hardware described context. In AArch64, the feature also adds the SCXTNUM_EL0, SCXTNUM_EL1, SCXTNUM_EL2 and the SCXTNUM_EL3 registers, which provide a number that can be used to separate out different context numbers within their respective Exception levels for the purpose of protecting against side-channels using branch prediction and similar resources.

This feature is supported in both AArch64 and AArch32 states. The SCXTNUM_ELx registers are only supported in AArch64 state.

This feature is OPTIONAL in Armv8.0 implementations and mandatory in Armv8.5 implementations.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following fields identify the presence of ARMv8.0-CSV2:

- ID_AA64PFR0_EL1.CSV2
- ID_PFR0_EL1.CSV2
- ID_PFR0.CSV2

For more information, see:

- *Restrictions on the effects of speculation* on page B2-133.
- *Restrictions on the effects of speculation* on page E2-3989.

ARMv8.0-CSV3, Armv8.0 Cache Speculation Variant 3

Armv8.0-CSV3 adds a mechanism to identify if hardware cannot disclose information about whether data loaded under speculation with a permission or domain fault can be used to form an address, generate condition codes, or generate SVE predicate values, to be used by instructions newer than the load in the speculative sequence.

This feature is supported in both AArch64 and AArch32 states.

This feature is OPTIONAL in Armv8.0 implementations and mandatory in Armv8.5 implementations.

The following fields identify the presence of ARMv8.0-CSV3:

- ID_AA64PFR0_EL1.CSV3
- ID_PFR2_EL1.CSV3
- ID_PFR2.CSV3

ARMv8.0-PredInv, Armv8.0 Prediction Invalidation

ARMv8.0-PredInv adds the CFP RCTX, CPP RCTX, DVP RCTX, CFPRCTX, CPPRCTX, and DVPRCTX System instructions. These instructions prevent predictions based on information gathered from earlier execution within a particular execution context from affecting the later speculative execution within that context, to the extent that the speculative execution is observable through side channels.

This feature is supported in both AArch64 and AArch32 states.

This feature is OPTIONAL in Armv8.0 implementations and mandatory in Armv8.5 implementations.

The following fields identify the presence of ARMv8.0-PredInv:

- ID_AA64ISAR1_EL1.SPECRES
- ID_ISAR6_EL1.SPECRES
- ID_ISAR6.SPECRES

For more information, see:

- *Prediction restriction instructions* on page C5-382.
- *Execution and data prediction restriction System instructions* on page D4-2522.
- *Execution and data prediction restriction System instructions* on page G4-5910.

ARMv8.0-CP15SDISABLE2, Armv8.0 CP15SDISABLE2

ARMv8.0-CP15SDISABLE2 provides an implementation-defined mechanism, the **CP15SDISABLE2** signal, which when asserted HIGH prevents writes to a set of Secure CP15 registers. This signal is analogous to the existing **CP15SDISABLE** signal.

This feature is only supported when EL3 is executing in AArch32 state.

This feature is OPTIONAL in Armv8.0 implementations.

For more information, see:

- *The CP15SDISABLE and CP15SDISABLE2 input signals* on page G5-6056.

ARMv8.0-DoubleLock, Armv8.0 Double Lock

The ARMv8.0-DoubleLock feature is the mnemonic used for the OS Double Lock.

If ARMv8.3-DoPD is not implemented, this feature is OPTIONAL if ARMv8.2-Debug is implemented and mandatory if ARMv8.2-Debug is not implemented.

If ARMv8.3-DoPD is implemented, this feature is not implemented.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The ID_AA64DFR0_EL1.DoubleLock field identifies that the OS Double Lock has been implemented.

ARMv8.0-DGH, Armv8.0 Data Gathering Hint

ARMv8.0-DGH adds the Data Gathering Hint instruction to the hint space.

This instruction is added to the A64 instruction set only.

This feature is OPTIONAL in Armv8.0 implementations.

The ID_AA64ISAR1_EL1.DGH field identifies the presence of ARMv8.0-DGH.

For more information, see: *Hint instructions* on page C3-205.

ARMv8.0-ETS, Armv8.0 Enhanced Translation Synchronization

ARMv8.0-ETS adds support for enhanced memory access ordering requirements for translation table walks.

This feature is supported in AArch64 and AArch32 states.

This feature is OPTIONAL in Armv8.0 implementations.

The following fields identify the presence of ARMv8.0-ETS:

- ID_AA64MMFR1_EL1.ETS.
- ID_MMFR5_EL1.ETS.
- ID_MMFR5.ETS.

For more information, see:

- *Ordering of memory accesses from translation table walks* on page D5-2563.
- *Ordering of translation table walks* on page E2-3998.

A2.2.2 Architectural requirements within Armv8.0 architecture

The Armv8.0 architecture includes some mandatory changes that have been added to the architecture at a later date that are not associated with a feature. These are:

Prefetch speculation protection

When substituting a direct branch with another direct branch, or a `NOP` with a direct branch, by the modified PE, at around the time that the executing PE is executing the software being modified, prefetch speculation protection prevents the old instructions from accidentally being fetched to the executing PE. For further information on implementation of these requirements, see:

- *Ordering of instruction fetches* on page B2-132.
- *Ordering of instruction fetches* on page E2-3989.

An implementation of the Armv8.0 architecture must comply with all of the additional requirements. When combined with the mandatory architectural features that have been added to the Armv8.0 architecture, such an implementation is also called an implementation of the Armv8.0 architecture.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A2.3 The Armv8 Cryptographic Extension

The Armv8 Cryptographic Extension provides instructions for the acceleration of encryption and decryption, and includes the following features:

- ARMv8.0-AES, which includes `AESD` and `AESE` instructions.
- ARMv8.0-SHA, which includes the `SHA1*` and `SHA256*` instructions.

The presence of the Cryptographic Extension in an implementation is subject to export license controls. The Cryptographic Extension is an extension of the SIMD support and operates on the vector register file.

The Cryptographic Extension also provides multiply instructions that operate on long polynomials.

The Cryptographic Extension provides this functionality in AArch64 state and AArch32 state, and an implementation that supports both AArch64 state and AArch32 state provides the same Cryptographic Extension functionality in both states.

For more information, see *The Cryptographic Extension* on page C3-261 or *The Cryptographic Extension in AArch32 state* on page F1-4070.

A2.3.1 Armv8.2 extensions to the Cryptographic Extension

From Armv8.2, an implementation of the Armv8.0 Cryptographic Extension can include either or both of:

- The AES functionality, including support for multiplication of 64-bit polynomials. The ID_AA64ISAR0_EL1.AES field indicates whether this functionality is supported.
- The SHA1 and SHA2-256 functionality. The ID_AA64ISAR0_EL1.{SHA2, SHA1} fields indicate whether this functionality is supported.

In addition, Armv8.2 adds two optional extensions to the Armv8 Cryptographic Extension, that provide cryptographic functionality in AArch64 state only. These two optional features are:

ARMv8.2-SHA, SHA2-512 and SHA3 functionality

In the A64 instruction set only, ARMv8.2-SHA adds Advanced SIMD instructions that support:

- SHA2-512 (SHA512).
- SHA3.

Implementation of ARMv8.2-SHA requires implementation of the Armv8.0 Cryptographic Extension SHA-1 and SHA256 functionality.

The ID_AA64ISAR0_EL1.{SHA2, SHA3} fields identify the presence of Armv8.2-SHA.

For more information see *Armv8.2-SHA, SHA2-512 and SHA3* on page C3-262.

ARMv8.2-SM, SM3 and SM4 functionality

In the A64 instruction set only, ARMv8.2-SM adds Advanced SIMD instructions that support the Chinese cryptography algorithms SM3 and SM4.

Implementation of ARMv8.2-SM is independent of the implementation of any SHA functionality.

The ID_AA64ISAR0_EL1.{SM3, SM4} fields identify the presence of ARMv8.2-SM.

Note

This means ARMv8.2-SM can be implemented without any other Cryptographic Extension features.

For more information see *Armv8.2-SM, SM3 and SM4* on page C3-264.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A2.4 The Armv8.1 architecture extension

The Armv8.1 architecture extension adds both architectural features and architectural requirements, see:

- *Architectural features added by Armv8.1*.
- *Additional requirements of Armv8.1* on page A2-71.

A2.4.1 Architectural features added by Armv8.1

An implementation of the Armv8.1 extension must include all of the features that this section describes as mandatory. Such an implementation, when combined with the additional requirements of Armv8.1, is also called an implementation of the Armv8.1 architecture.

The Armv8.1 architecture extension adds the following architectural features, which are identified by the architectural feature name and a short description of the feature:

ARMv8.1-LSE, Armv8.1 Large System Extensions

ARMv8.1-LSE introduces a set of atomic instructions:

- Compare and Swap instructions, `CAS` and `CASP`.
- Atomic memory operation instructions, `LD<OP>` and `ST<OP>`, where `<OP>` is one of `ADD`, `CLR`, `EOR`, `SET`, `SMAX`, `SMIN`, `UMAX`, and `UMIN`.
- Swap instruction, `SWP`.

These instructions are only added to the A64 instruction set.

This feature is mandatory in Armv8.1 implementations.

Implementations of ARMv8.1-VHE require the implementation of ARMv8.1-LSE.

The ID_AA64ISAR0_EL1.Atomic field identifies the presence of ARMv8.1-LSE.

For more information, see:

- *Compare and Swap* on page C3-221.
- *Atomic memory operations* on page C3-222.
- *Swap* on page C3-224.

ARMv8.1-RDMA, Armv8.1 Advanced SIMD instructions

ARMv8.1-RDMA introduces Rounding Double Multiply Add/Subtract Advanced SIMD instructions. For more information, see:

For the A64 instruction set

- *SQRDMLAH (by element)* on page C7-2047.
- *SQRDMLAH (vector)* on page C7-2050.
- *SQRDMLSH (by element)* on page C7-2052.
- *SQRDMLSH (vector)* on page C7-2055.

For the T32 and A32 instruction sets

- *VQRDMLAH* on page F6-5437.
- *VQRDMLSH* on page F6-5441.

This feature is mandatory in Armv8.1 implementations.

The following fields identify the presence of Armv8.1-RDMA:

- ID_AA64ISAR0_EL1.RDM.
- ID_ISAR5_EL1.RDM.
- ID_ISAR5.RDM.

ARMv8.1-LOR, Limited ordering regions

Limited ordering regions allow large systems to perform special load-acquire and store-release instructions that provide order between the memory accesses to a region of the PA map as observed by a limited set of observers.

This feature is supported in AArch64 state only.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This feature is mandatory in Armv8.1 implementations.

The ID_AA64MMFR1_EL1.LO field identifies the support for ARMv8.1-LOR.

For more information, see:

- *Limited ordering regions* on page B2-141.

ARMv8.1-HPD, Hierarchical permission disables

ARMv8.1-HPD introduces the facility to disable the hierarchical attributes, APTable, PXNTable, and UXNTable, in the translation tables. This disable has no effect on the NSTable bit.

This feature is mandatory in Armv8.1 implementations.

This feature is added only to the VMSAv8-64 translation regimes. Armv8.2 extends this to the AArch32 translation regimes, see ARMv8.2-AA32HPD.

The ID_AA64MMFR1_EL1.HPDS field identifies the support for ARMv8.1-HPD.

ARMv8.1-TTHM, Hardware management of the Access flag and dirty state

In Armv8.0, all updates to the translation tables are performed by software. From Armv8.1, for the VMSAv8-64 translation regimes only, hardware can perform updates to the translation tables in two contexts:

- Hardware management of the Access flag.
- Hardware management of dirty state, with updates to a dirty state in the translation tables.

The dirty state is introduced in Armv8.1.

Hardware management of dirty state can only be enabled when hardware management of the Access flag is also enabled.

This feature is optional in Armv8.1 implementations. It is IMPLEMENTATION DEFINED whether this is implemented.

The ID_AA64MMFR1_EL1.HAFDBS field identifies the support for ARMv8.1-TTHM.

For more information, see:

- *The dirty state* on page D5-2618.
- *Hardware management of the Access flag and dirty state* on page D5-2619.

ARMv8.1-PAN, Privileged access never

ARMv8.1-PAN adds a bit to PSTATE. When the value of this PAN state bit is 1, any privileged data access from EL1, or EL2 when HCR_EL2.E2H is 1, to a virtual memory address that is accessible at EL0, generates a Permission fault.

This feature is mandatory in Armv8.1 implementations.

This feature is supported in AArch64 and AArch32 states.

The following fields identify the support for ARMv8.1-PAN:

- ID_AA64MMFR1_EL1.PAN.
- ID_MMFR3_EL1.PAN.
- ID_MMFR3.PAN.

For more information, see:

- *About PSTATE.PAN* on page D5-2607.
- *About the PAN bit* on page G5-5971.

ARMv8.1-VMID16, 16-bit VMID

In an Armv8.1 implementation, when EL2 is using AArch64, the VMID size is an IMPLEMENTATION DEFINED choice of 8 bits or 16 bits.

This feature is optional in Armv8.1 implementations. It is IMPLEMENTATION DEFINED whether this is implemented.

When implemented, this feature is supported only when EL2 is using AArch64.

The ID_AA64MMFR1_EL1.VMIDBits field identifies the supported VMID size.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For more information, see:

- *VMID size* on page D5-2664.

ARMv8.1-VHE, Virtualization Host Extensions

Armv8.1 introduces the *Virtualization Host Extensions* (VHE) that provide enhanced support for Type 2 hypervisors in Non-secure state.

This feature is mandatory in Armv8.1 implementations.

An implementation that includes ARMv8.1-VHE requires ARMv8.1-LSE to be implemented.

The ID_AA64MMFR1_EL1.VH field identifies the support for ARMv8.1-VHE.

The following fields indicate the presence of the Virtualization Host Extensions for debug, including the changes for the PC Sample-based Profiling Extension and the Performance Monitors Extension:

- ID_AA64DFR0_EL1.DebugVer.
- ID_DFR0_EL1.{CopSDbg, CopDbg}.

For more information, see:

- *Virtualization Host Extensions* on page D5-2638.

ARMv8.1-PMU, Armv8.1 PMU Extension

Armv8.1 makes the following enhancements to the Performance Monitors Extension:

- The event number space is extended to 16 bits to allow additional IMPLEMENTATION DEFINED event types, and the reserved space for future additions to the architecturally-defined event types is extended.
- The HPMD bit is added to MDCR_EL2. This bit disables event counting at EL2.
- The STALL_FRONTEND and STALL_BACKEND events are required to be implemented. For more information, see *Required events* on page D7-2769.

The Performance Monitors Extension is an optional feature, but if it is implemented, an Arm8.1 implementation must include ARMv8.1-PMU.

The following fields identify the support for ARMv8.1-PMU:

- ID_AA64DFR0_EL1.PMUVer.
- ID_DFR0_EL1.PerfMon.
- ID_DFR0.PerfMon.

A2.4.2 Additional requirements of Armv8.1

The Armv8.1 architecture includes some mandatory changes that are not associated with a feature. These are:

Changes to CRC32 instructions

All implementations of the Armv8.1 architecture are required to implement the `CRC32*` instructions. These are optional in Armv8.0.

The following fields identify the support for the `CRC32*` instructions:

- ID_AA64ISAR0_EL1.CRC32.
- ID_ISAR5_EL1.CRC32.
- ID_ISAR5.CRC32.

An implementation of the Armv8.1 extension must comply with all of the additional requirements. Such an implementation, when combined with the mandatory architectural features of Armv8.1, is also called an implementation of the Armv8.1 architecture.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A2.5 The Armv8.2 architecture extension

The Armv8.2 architecture extension adds both architectural features and architectural requirements, see:

- *Architectural features added by Armv8.2*.
- *Additional requirements of Armv8.2* on page A2-78.
- *Features added to the Armv8.2 extension in later releases* on page A2-78.

The Armv8.2 architecture extension also adds functionality to the Cryptographic Extension, see *Armv8.2 extensions to the Cryptographic Extension* on page A2-68.

A2.5.1 Architectural features added by Armv8.2

An implementation of the Armv8.2 extension must include all of the features that this section describes as mandatory. Such an implementation, when combined with the additional requirements of Armv8.2, is also called an implementation of the Armv8.2 architecture.

The Armv8.2 architecture extension adds the following architectural features, which are identified by the architectural feature name and a short description of the feature:

ARMv8.2-A64ISA, Armv8.2 changes to the A64 ISA

ARMv8.2-A64ISA adds the `BFC` instruction to the A64 instruction set as an alias of `BFM`. It also requires that the `BFC` instruction and the A64 pseudo-instruction `REV64` are implemented by assemblers.

Note

- In Armv8.0 and Armv8.1, the A64 pseudo-instruction `REV64` is optional.
- Because this feature relates to support for an instruction alias and for a pseudo-instruction, there are no corresponding feature ID register fields.

This change to the instruction set and assembler requirements is mandatory in an Armv8.2 implementation.

For more information, see:

- *BFC* on page C6-822.
- *REV64* on page C6-1180.

ARMv8.2-ATS1E1, `AT S1E1R` and `AT S1E1W` instruction variants, taking account of PSTATE.PAN

ARMv8.2-ATS1E1 adds variants of the AArch64 `AT S1E1R` and `AT S1E1W` instructions and the AArch32 `ATS1CPR` and `ATS1CPW` instructions. These instructions factor in the PSTATE.PAN bit when determining whether or not the location will generate a permission fault for a privileged access, as is reported in the PAR. For more information, see:

For the AArch64 System instructions

- *AT S1E1RP, Address Translate Stage 1 EL1 Read PAN* on page C5-551.
- *AT S1E1WP, Address Translate Stage 1 EL1 Write PAN* on page C5-555.

For the AArch32 System instructions

- *ATS1CPRP, Address Translate Stage 1 Current state PL1 Read PAN* on page G8-6134.
- *ATS1CPWP, Address Translate Stage 1 Current state PL1 Write PAN* on page G8-6136.

This feature is mandatory in Armv8.2 implementations.

These instructions are added to the A64 and A32/T32 instruction sets.

The following fields identify the presence of ARMv8.2-ATS1E1:

- ID_AA64MMFR1_EL1.PAN.
- ID_MMFR3_EL1.PAN.
- ID_MMFR3.PAN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For more information, see:

- *Address translation instructions* on page D5-2589.
- *ATS1C**, Address translation stage 1, current security state* on page G5-6044.
- *Encoding and availability of the address translation instructions* on page G5-6045.

ARMv8.2-FP16, Half-precision floating-point data processing

ARMv8.2-FP16 supports:

- Half-precision data-processing instructions for Advanced SIMD and floating-point in both AArch64 and AArch32 states.
- The FPCR.FZ16 and FPSCR.FZ16 bits, that enable a Flush-to-zero mode for half-precision data-processing instructions.

This feature is optional in Armv8.2 implementations, unless The Scalable Vector Extension (SVE) is implemented, in which case ARMv8.2-FP16 is mandatory. When this feature is implemented it is implemented in both Advanced SIMD and floating-point, and in AArch64 and AArch32 states.

The following fields identify the presence of ARMv8.2-FP16:

- ID_AA64PFR0_EL1.{FP, AdvSIMD}.
- MVFR1_EL1.{FPHP, SIMDHP}.
- MVFR1.{FPHP, SIMDHP}.

For more information, see:

- *Half-precision floating-point formats* on page A1-44.
- *Flush-to-zero* on page A1-55.
- *Modified immediate constants in A64 instructions* on page C2-198.

ARMv8.2-DotProd, SIMD Dot Product

ARMv8.2-DotProd provides instructions to perform the dot product of two 32-bit vectors, accumulating the result in a third 32-bit vector. This can be performed using signed or unsigned arithmetic.

This feature is optional in Armv8.2 implementations, and mandatory in Armv8.4 implementations.

These instructions are added to the A64 and A32/T32 instruction sets.

The following fields identify the presence of ARMv8.2-DotProd:

- ID_AA64ISAR0_EL1.DP.
- ID_ISAR6_EL1.DP.
- ID_ISAR6.DP.

For more information, see:

- *SIMD dot product* on page C3-259.
- *Advanced SIMD dot product instructions* on page F1-4067.

ARMv8.2-FHM, Floating-point multiplication variant

ARMv8.2-FHM adds floating-point multiplication instructions.

These instructions are added to the A64 and A32/T32 instruction sets.

This feature is optional in Armv8.2 implementations, and can only be implemented when ARMv8.2-FP16 is implemented. This feature is mandatory in Armv8.4 implementations when ARMv8.2-FP16 is implemented. This feature is not implemented in Armv8.4 implementations when ARMv8.2-FP16 is not implemented.

The following fields identify the presence of ARMv8.2-FHM:

- ID_AA64ISAR0_EL1.FHM.
- ID_ISAR6_EL1.FHM.
- ID_ISAR6.FHM.

For more information, see:

- *SIMD arithmetic* on page C3-247.
- *SIMD by element arithmetic* on page C3-253.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *Advanced SIMD multiply instructions* on page F1-4066.

ARMv8.2-LSMAOC, Load/Store Multiple atomicity and ordering controls

ARMv8.2-LSMAOC adds controls that disable legacy behavior of AArch32 Load Multiple and Store Multiple instructions, and provide a trap of one aspect of this legacy behavior.

Implementation of ARMv8.2-LSMAOC is optional. When implemented it provides:

- LSMAOE fields in the SCTLR_EL1, SCTLR_EL2, HSCTLR, and SCTLR registers. These fields can have the following effects on the behavior of AArch32 Load Multiple and Store Multiple instructions:
  - An interrupt can be taken between two memory accesses made by a single Load Multiple or Store Multiple instruction.
  - The memory accesses made by a single Load Multiple or Store Multiple instruction to Device memory with the non-Reordering attribute can be reordered.
- nTLSMD fields in the SCTLR_EL1, SCTLR_EL2, HSCTLR, and SCTLR registers. These fields can cause an access to Device-nGRE, Device-nGnRE, or Device-nGnRnE memory by an AArch32 Load Multiple and Store Multiple instruction to generate an Alignment fault.

Note

Armv8.2 deprecates software dependence on the legacy behavior of AArch32 Load Multiple and Store Multiple instructions, and these fields disable this behavior.

The following fields identify the support for ARMv8.2-LSMAOC:

- ID_AA64MMFR2_EL1.LSM
- ID_MMFR4_EL1.LSM
- ID_MMFR4.LSM.

For more information, see the register field descriptions and:

- *Generation of Alignment faults by Load/store multiple accesses to Device memory* on page E2-4005.
- *Multi-register loads and stores that access Device memory* on page E2-4018.
- *Taking an interrupt or other exception during a multiple-register load or store* on page G1-5737.

ARMv8.2-UAO, PSTATE override of Unprivileged Load/Store

Armv8.2 adds a bit to PSTATE. When the value of PSTATE.UAO is 1, and when executed at EL1 or at EL2 with HCR_EL2.{E2H, TGE} == {1, 1}, the memory accesses made by the Load/Store unprivileged instructions behave as if they were made by the Load/Store register instructions. See *Load/Store unprivileged* on page C3-213 and *Load/Store register* on page C3-209.

This feature is mandatory in Armv8.2 implementations.

This feature is supported in AArch64 state only.

The ID_AA64MMFR2_EL1.UAO field identifies the support for ARMv8.2-UAO.

For more information, see *About PSTATE.UAO* on page D5-2608.

ARMv8.2-DCPoP, Data cache clean to Point of Persistence

ARMv8.2-DCPoP introduces a mechanism to identify and manage persistent memory locations in a shared memory hierarchy, including adding the `DC CVAP` instruction.

This feature is mandatory in Armv8.2 implementations.

This feature is supported in AArch64 state only.

The ID_AA64ISAR1_EL1.DPB field identifies the support for ARMv8.2-DCPoP.

For more information about ARMv8.2-DCPoP, see *Memory hierarchy* on page B2-143.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ARMv8.2-VPIPT, VMID-aware PIPT instruction cache

ARMv8.2-VPIPT supports a instruction cache type, described as the *VMID-aware PIPT* (VPIPT) instruction cache.

Note

Armv8.2 adds VPIPT to the set of supported cache types, meaning an Armv8.2 implementation is permitted to implement VPIPT caches, but is not required to do so.

This feature is supported in AArch64 and AArch32 states.

The CTR_EL0.L1Ip and CTR.L1Ip fields identify the support for ARMv8.2-VPIPT.

For more information, see:

- *VPIPT (VMID-aware PIPT) instruction caches* on page D5-2686.
- *VPIPT (VMID-aware PIPT) instruction caches* on page G5-6010.

ARMv8.2-AA32HPD, AArch32 Hierarchical permission disables

ARMv8.1-HPD introduced the ability to disable the hierarchical attributes, APTable, PXNTable, and UXNTable, in the VMSAv8-64 translation regimes. ARMv8.2-AA32HPD extends this functionality to the VMSAv8-32 translation regimes when those regimes are using the Long descriptor translation table format.

This feature is optional in Armv8.2 implementations. It is IMPLEMENTATION DEFINED whether this is implemented.

The ID_MMFR4_EL1.HPDS and ID_MMFR4.HPDS fields identify the support for ARMv8.2-AA32HPD.

For more information, see *Attribute fields in VMSAv8-32 Long-descriptor translation table format descriptors* on page G5-5952.

ARMv8.2-TTPBHA, Translation table page-based hardware attributes

Armv8.2 provides a mechanism to allow operating systems or hypervisors to make up to four bits of translation table final-level descriptors available for IMPLEMENTATION DEFINED hardware use.

This functionality is available for all translation regimes in AArch64 state and for stages of translation in AArch32 state that use the Long descriptor translation table format.

ARMv8.2-TTPBHA is optional in Armv8.2 implementations, but implementation of ARMv8.2-TTPBHA requires implementation of both:

- ARMv8.1-HPD.
- ARMv8.2-AA32HPD, if any Exception level higher than EL0 can use AArch32.

Note

For stage 1 translations, page-based hardware attributes can only be used for a stage of translation for which the Hierarchical permission disables field has a value of 1.

The following fields identify the support for ARMv8.2-TTPBHA:

- ID_AA64MMFR1_EL1.HPDS
- ID_MMFR4_EL1.HPDS
- ID_MMFR4.HPDS.

For more information, see:

- *Memory attribute fields in the VMSAv8-64 translation table format descriptors* on page D5-2599.
- *Attribute fields in VMSAv8-32 Long-descriptor translation table format descriptors* on page G5-5952.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ARMv8.2-LPA, Large PA and IPA support

ARMv8.2-LPA:

- Allows a larger *intermediate physical address* (IPA) and PA space of up to 52 bits when using the 64KB translation granule.
- Allows a level 1 block size where the block covers a 4TB address range for the 64KB translation granule if the implementation support 52 bits of PA.

This is an optional feature in Armv8.2 implementations. It is IMPLEMENTATION DEFINED whether it is implemented.

This feature is supported in AArch64 state only.

The ID_AA64MMFR0_EL1.PARange field identifies the support for ARMv8.2-LPA.

For more information about ARMv8.2-LPA, see:

- *VMSA address types and address spaces* on page D5-2533.
- *Address size configuration* on page D5-2547.
- *Extending addressing above 48 bits* on page D5-2552.
- *VMSAv8-64 translation table level 0, level 1, and level 2 descriptor formats* on page D5-2593.
- *Armv8 translation table level 3 descriptor formats* on page D5-2597.

ARMv8.2-LVA, Large VA support

ARMv8.2-LVA supports a larger VA space for each translation table base register of up to 52 bits when using the 64KB translation granule.

This feature is supported in AArch64 state only.

This is an optional feature in Armv8.2 implementations. It is IMPLEMENTATION DEFINED whether it is implemented.

If ARMv8.2-LVA is implemented, then any implemented trace macrocell must be at least ETMv4.2.

The ID_AA64MMFR2_EL1.VARange field identifies the support for ARMv8.2-LVA.

For more information about ARMv8.2-LVA, see:

- *VMSA address types and address spaces* on page D5-2533.
- *Address size configuration* on page D5-2547.
- *Extending addressing above 48 bits* on page D5-2552.
- *VMSAv8-64 translation table level 0, level 1, and level 2 descriptor formats* on page D5-2593.
- *Armv8 translation table level 3 descriptor formats* on page D5-2597.

ARMv8.2-TTCNP, Translation table Common not private translations

ARMv8.2-TTCNP permits multiple PEs in the same Inner Shareable domain to use the same translation tables for a given stage of address translation.

This feature is mandatory in Armv8.2 implementations.

This facility is available for all VMSAv8-64 translation regimes and for VMSAv8-32 translation stages that use the Long descriptor translation table format.

The following fields identify the support for ARMv8.2-TTCNP:

- ID_AA64MMFR2_EL1.CnP.
- ID_MMFR4_EL1.CnP.
- ID_MMFR4.CnP.

For more information, see:

- *Common not private translations* on page D5-2663.
- *Common not private translations in VMSAv8-32* on page G5-5999.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ARMv8.2-TTS2UXN, Translation table stage 2 Unprivileged Execute-never

ARMv8.2-TTS2UXN extends the stage 2 translation table access permissions to provide control of whether memory is executable at EL0 independent of whether it is executable at EL1.

This feature is mandatory in Armv8.2 implementations.

This facility is available for stage 2 translation stages in VMSAv8-64 and VMSAv8-32.

The following fields identify the support for ARMv8.2-TTS2UXN:

- ID_AA64MMFR1_EL1.XNX.
- ID_MMFR4_EL1.XNX.
- ID_MMFR4.XNX.

For more information, see:

- *Access permissions for instruction execution* on page D5-2612.
- *Access permissions for instruction execution* on page G5-5972.

ARMv8.2-Debug, Armv8.2 Debug

ARMv8.2-Debug covers a selection of mandatory changes, including:

- If the Core power domain is powered up and `DoubleLockStatus()` == TRUE, EDPRSR.{DLK,SPD,PU} is only permitted to read {UNKNOWN, 0, 0}.
- The definition of Exception Catch debug events is extended to include reset entry.
- All CONSTRAINED UNPREDICTABLE cases that generate Exception Catch debug events are removed.
- Controls are added to EDECCR to control Exception Catch debug event generation on exception return.
- All IMPLEMENTATION DEFINED control of external debug accesses to OSLAR_EL1 is removed.
- `ExternalSecureNoninvasiveDebugEnabled()` cannot override software controls of counting attributable events in Secure state.

If ARMv8.2-Debug is implemented, ARMv8.0-DoubleLock is optional.

The fields that identify the support for ARMv8.2-Debug are:

- ID_AA64DFR0_EL1.DebugVer and DBGDIDR.Version.
- ID_DFR0_EL1.{CopSDbg, CopDbg} and ID_DFR0.{CopSDbg, CopDbg}.
- EDDEVARCH.ARCHID.

For more information, see:

- *Exception Catch debug events from Armv8.2* on page H3-6981.
- *EDPRSR.{DLK, SPD, PU} and the Core power domain* on page H6-7035.
- *Interaction with EL3* on page D7-2701.
- *External access disabled* on page H8-7057.

ARMv8.2-PCSample, PC Sample-based Profiling

In Armv8.2, the control and implementation of the optional PC Sample-based Profiling extension is moved from ED*SR Debug registers to PM*SR registers in the Performance Monitors address space. See Chapter H7 *The PC Sample-based Profiling Extension*.

The PC Sample-based Profiling Extension is an OPTIONAL feature. If it is implemented, an Arm8.2 implementation must also include ARMv8.2-PCSample.

If Secure EL2 and PC Sample-based Profiling are both implemented, ARMv8.2-PCSample is mandatory.

The following fields identify the support for ARMv8.2-PCSample:

- EDDEVID.PCSample.
- DBGDEVID.PCSample.
- EDDEVID1.PCSROffset.
- DBGDEVID1.PCSROffset.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- PMDEVID.PCSample.

ARMv8.2-IESB, Implicit error synchronization event

ARMv8.2-IESB adds an implicit error synchronization event at exception entry and return, controlled by the added SCTLR_ELx.IESB fields. An IESB field is added to the ESR_ELx syndrome registers.

The implicit error synchronization events affect the same synchronizable asynchronous events that are synchronized by the `ESB` instruction, see *The Reliability, Availability, and Serviceability Extension (RAS Extension)* on page A2-97.

This feature is optional in Armv8.2 implementations.

This feature is supported in AArch64 state only.

The ID_AA64MMFR2_EL1.IESB field identifies the support for ARMv8.2-IESB.

For more information, see the *ARM® Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for the ARMv8-A architecture profile*.

Extensions to the Arm Cryptographic Extensions

See the description of the ARMv8.2-SHA and ARMv8.2-SM features in *Armv8.2 extensions to the Cryptographic Extension* on page A2-68.

A2.5.2 Additional requirements of Armv8.2

The Armv8.2 architecture includes some mandatory changes that are not associated with a feature. These are:

Changes to ACTLR2 and HCTLR2 registers

In AArch32 state, the ACTLR2 and HACTLR2 registers become mandatory.

Implementation of RAS Extension

The RAS Extension must be implemented, see *The Reliability, Availability, and Serviceability Extension (RAS Extension)* on page A2-97.

An implementation of the Armv8.2 extension must comply with all of the additional requirements. Such an implementation, when combined with the mandatory architectural features of Armv8.2, is also called an implementation of the Armv8.2 architecture.

If PMUv3 is implemented, the feature ARMv8.4-PMU is optional in Armv8.2 implementations.

A2.5.3 Features added to the Armv8.2 extension in later releases

ARMv8.2-EVT, Armv8.2 Enhanced Virtualization Traps

ARMv8.2-EVT introduces additional traps for EL1 and EL0 Cache controls. These traps are independent of existing controls.

This feature is supported in AArch64 and AArch32 states.

This feature is OPTIONAL in Armv8.2 implementations and is mandatory in Armv8.5.

ID_AA64MMFR2_EL1.EVT identifies the support for the AArch64 traps controls.

ID_MMFR4_EL1.EVT and ID_MMFR4.EVT identify support for the AArch32 traps.

For more information, see:

- HCR_EL2.{TTLBIS, TTLBOS, TICAB, TOCU, TID4}.
- HCR2.{TTLBIS, TICAB, TOCU, TID4}.

ARMv8.2-DCCVADP, Armv8.2 Cache clean to Point of Deep Persistence

ARMv8.2-DCCVADP allows two levels of cache clean to the Point of Persistence by:

- Redefining Point of Persistence, which changes the scope of `DC CVAP`.
- Defining a Point of Deep Persistence.
- Adding the DC CVADP System instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This feature is supported in AArch64 state only.

This feature is OPTIONAL in Armv8.2 implementations, and is mandatory in Armv8.5 implementations.

The ID_AA64ISAR1_EL1.DPB identifies the support for ARMv8.2-DCCVADP.

For further information, see *Terminology for Clean, Invalidate, and Clean and Invalidate instructions* on page D4-2507.

ARMv8.2-BF16, Armv8.2 AArch64 BFloat16 Extension

ARMv8.2-BF16 supports the BFloat16, or BF16, 16-bit floating-point storage format in AArch64 state. This format supports:

- The BFloat16 floating-point data type.
- Arithmetic instructions to accelerate dot products and matrix multiplications of BF16 values.
- Instructions to convert single-precision floating-point values to BF16 format.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in Armv8.2 implementations and mandatory in Armv8.6 implementations.

The ID_AA64ISAR1_EL1.BF16 field identifies the presence of ARMv8.2-BF16.

When both Advanced SIMD and SVE are implemented, the ID_AA64ISAR1_EL1.BF16 and the ID_AA64ZFR0_EL1.BF16 fields must return the same value.

For further information, see:

- *BFloat16 floating-point format* on page A1-48.
- *BFloat16 floating-point instructions* on page C3-246.
- *SIMD BFloat16* on page C3-260.
- *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*.

ARMv8.2-AA32BF16, Armv8.2 AArch32 BFloat16 Extension

ARMv8.2-AA32BF16 supports the BFloat16, or BF16, 16-bit floating-point storage format in AArch32 state. This format supports:

- The BFloat16 floating-point data type.
- Arithmetic instructions to accelerate dot products and matrix multiplications of BF16 values.
- Instructions to convert single-precision floating-point values to BF16 format.

This feature is supported in AArch32 state only.

This feature is OPTIONAL in Armv8.2 implementations.

The ID_ISAR6_EL1.BF16 and ID_ISAR6.BF16 fields identify the presence of ARMv8.2-AA32BF16.

For further information, see:

- *BFloat16 floating-point format* on page A1-48.
- *Advanced SIMD BFloat16 instructions* on page F1-4068.
- *Floating-point data-processing* on page F3-4131.

ARMv8.2-I8MM, Armv8.2 Int8 Matrix Multiplication

ARMv8.2-I8MM introduces integer matrix multiply-accumulate instructions and mixed sign dot product instructions.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in Armv8.2 implementations and mandatory in Armv8.6 implementations.

The ID_AA64ISAR1_EL1.I8MM field identifies the presence of ARMv8.2-I8MM.

When both Advanced SIMD and SVE are implemented, the ID_AA64ISAR1_EL1.I8MM and the ID_AA64ZFR0_EL1.I8MM fields must return the same value.

For further information, see:

- *SIMD dot product* on page C3-259.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *SIMD matrix multiplication* on page C3-261.
- *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*.

ARMv8.2-AA32I8MM, Armv8.2 Int8 Matrix Multiplication

ARMv8.2-AA32I8MM introduces integer matrix multiply-accumulate instructions and mixed sign dot product instructions.

This feature is supported in AArch32 state only.

This feature is OPTIONAL in Armv8.2 implementations.

The ID_ISAR6_EL1.I8MM and ID_ISAR6.I8MM fields identify the presence of ARMv8.2-AA32I8MM.

For further information, see:

- *Advanced SIMD dot product instructions* on page F1-4067.
- *Advanced SIMD matrix multiply instructions* on page F1-4068.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A2.6 The Armv8.3 architecture extension

The Armv8.3 architecture extension adds both architectural features and additional requirements, see:

- *Architectural features added by Armv8.3*.
- *Additional requirements of Armv8.3* on page A2-83.
- *Features added to the Armv8.3 extension in later releases* on page A2-83.

A2.6.1 Architectural features added by Armv8.3

An implementation of the Armv8.3 extension must include all of the features that this section describes as mandatory. Such an implementation is also called an implementation of the Armv8.3 architecture.

The Armv8.3 architecture extension adds the following architectural features, which are identified by the architectural feature name and a short description of the feature:

ARMv8.3-CompNum, SIMD complex number support

ARMv8.3-CompNum introduces instructions for floating-point multiplication and addition of complex numbers.

These instructions are added to the A64 and A32/T32 instruction sets.

This feature is mandatory in Armv8.3 implementations.

The half-precision versions of these instructions are implemented only if ARMv8.2-FP16 is implemented. Otherwise they are UNDEFINED.

The fields that identify the presence of ARMv8.3-CompNum are:

- ID_AA64ISAR1_EL1.FCMA.
- ID_ISAR5_EL1.VCMA.
- ID_ISAR5.VCMA.

For more information, see:

- *SIMD complex number arithmetic* on page C3-260.
- *Advanced SIMD complex number arithmetic instructions* on page F1-4067.

ARMv8.3-JSconv, Javascript conversion instructions

ARMv8.3-JSConv introduces instructions that perform a conversion from a double-precision floating point value to a signed 32-bit integer, with rounding to zero. For more information, see:

For the A64 instruction set

- *FJCVTZS* on page C7-1621.

For the A32/T32 instruction set

- *VJCVT* on page F6-5203.

These instructions are added to the A64 and A32/T32 instruction sets.

The feature is mandatory in Armv8.3 implementations.

The fields that identify the presence of ARMv8.3-JSconv are:

- ID_AA64ISAR1_EL1.JSCVT
- ID_ISAR6_EL1.JSCVT
- ID_ISAR6.JSCVT.

For more information, see:

- *Floating-point conversion* on page C3-241.
- *About the A64 SIMD and floating-point instructions* on page C7-1398.
- *Advanced SIMD and floating-point instructions* on page E1-3960.
- *Floating-point data-processing instructions* on page F1-4072.

ARMv8.3-RCPC, Weaker release consistency

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ARMv8.3-RCPC introduces three instructions to support the weaker *Release Consistency processor consistent* (RCpc) model that enables the reordering of a Store-Release followed by a Load-Acquire to a different address:

- *LDAPR* on page C6-944.
- *LDAPRB* on page C6-946.
- *LDAPRH* on page C6-948.

These instructions are added to the A64 instruction set.

The feature is mandatory in Armv8.3 implementations.

The ID_AA64ISAR1_EL1.LRCPC field identifies the presence of ARMv8.3-RCPC.

For more information, see:

- *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.
- *Load-Acquire/Store-Release* on page C3-214.

ARMv8.3-NV, Nested Virtualization

ARMv8.3-NV provides support for a Guest Hypervisor to run in Non-secure EL1 and ensures that the Guest Hypervisor is unaware that it is running at that Exception level. A Guest Hypervisor is supported regardless of the value of HCR_EL2.E2H.

This feature is supported in AArch64 state only.

The feature is optional in Armv8.3 implementations. This feature must be implemented if ARMv8.4-NV is implemented.

The ID_AA64MMFR2_EL1.NV field identifies the support for ARMv8.3-NV.

For more information, see *Nested virtualization* on page D5-2644.

ARMv8.3-CCIDX, Cache extended number of sets

ARMv8.3-CCIDX introduces the following registers to allow caches to be described with greater numbers of sets and greater associativity:

- A 64-bit format of CCSIDR_EL1.
- CCSIDR2_EL1.
- CCSIDR2.

This feature is supported in AArch64 and AArch32 states.

This feature is optional in Armv8.3 implementations.

The following fields identify the support for ARMv8.3-CCIDX:

- ID_AA64MMFR2_EL1.CCIDX
- ID_MMFR4_EL1.CCIDX.
- ID_MMFR4.CCIDX.

For more information, see:

- *Possible formats of the Cache Size Identification Register, CCSIDR_EL1* on page D4-2501.
- *Possible formats of the Cache Size Identification Registers, CCSIDR and CCSIDR2* on page G4-5891.

ARMv8.3-PAuth, Pointer Authentication

ARMv8.3-PAuth adds functionality that supports address authentication of the contents of a register before that register is used as the target of an indirect branch, or as a load.

This feature is supported only in AArch64 state.

This feature is mandatory in Armv8.3 implementations.

The fields ID_AA64ISAR1_EL1.{GPI, GPA, API, APA} identify the presence for ARMv8.3-PAuth.

For more information, see *Pointer authentication in AArch64 state* on page D5-2536.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A2.6.2 Additional requirements of Armv8.3

If PMUv3 is implemented, the feature ARMv8.4-PMU is optional in Armv8.3 implementations.

A2.6.3 Features added to the Armv8.3 extension in later releases

ARMv8.3-SPE, Armv8.3 Statistical Profiling Extensions

The ARMv8.3-SPE adds an Alignment Flag in the Events packet and filtering on this event using PMSEVFR_EL1, together with support for the profiling of Scalable Vector Extension operations.

This feature is only supported in AArch64.

This feature is optional in Armv8.3 implementations. When ARMv8.3-SPE is implemented the Statistical Profiling Extension must be implemented.

The fields in ID_AA64DFR0_EL1.PMSVer identify the support for ARMv8.3-SPE.

For more information see Chapter D9 *The Statistical Profiling Extension* and Chapter D10 *Statistical Profiling Extension Sample Record Specification*.

ARMv8.3-DoPD, Armv8.3 Debug over Powerdown

The ARMv8.3-DoPD provides a debug programmers' model where all debug and PMU registers are in the Core power domain, all CTI registers are in the Debug power domain. Power control is provided by a CoreSight Granular Power requestor (GPR) component.

When the OPTIONAL powerup mechanism is implemented and this feature is implemented, the debugger makes power control requests for the Core power domain using a CoreSight Class 0x9 ROM Table block, instead of using EDRCR.COREPURQ. EDRCR.COREPURQ is not implemented. Refer to the *ARM® CoreSight Architecture Specification* for more information.

This feature is optional in Armv8.3 implementations.

When ARMv8.3-DoPD is implemented:

- ARMv8.0-DoubleLock is not implemented.
- ARMv8.2-Debug must be implemented.
- If PC Sample-based profiling is implemented, ARMv8.2-PCSample must be implemented.
- The optional Software Lock is not implemented by the architecturally-defined debug components in the PE Core power domain.
- If an ETMv4 PE Trace Unit is implemented, the ETM must implement:
  - ETMv4.2 or later.
  - The Unified Power Domain Model.

The fields that identify the presence of ARMv8.3-DoPD are:

- EDDEVID.DebugPower.
- CTIDEVARCH.REVISION.

For more information see Chapter H6 *Debug Reset and Powerdown Support*.

ARMv8.3-PAuth2, Armv8.3 Pointer Authentication Enhancements

ARMv8.3-PAuth2 adds enhanced pointer authentication functionality that changes the mechanism by which a PAC is added to the pointer.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in Armv8.3 implementations, and mandatory in Armv8.6 implementations.

The ID_AA64ISAR1_EL1.APA and ID_AA64ISAR1_EL1.API fields identify the support for ARMv8.3-PAuth2.

For more information, see *Pointer authentication in AArch64 state* on page D5-2536.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ARMv8.3-FPAC, Armv8.3 Faulting on AUT* instructions

ARMv8.3-FPAC introduces faulting on an `AUT*` instruction and, optionally, on the combined instructions that perform pointer authentication. ARMv8.3-FPAC is added as a further extension to ARMv8.3-PAuth2.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in Armv8.3 implementations, and can only be implemented if ARMv8.3-PAuth2 is implemented.

The ID_AA64ISAR1_EL1.APA and ID_AA64ISAR1_EL1.API fields identify the support for ARMv8.3-FPAC.

For more information, see *Faulting on pointer authentication* on page D5-2539.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A2.7 The Armv8.4 architecture extension

The Armv8.4 architecture extension adds architectural features, see *Architectural features added by Armv8.4*. It also adds features to earlier architecture extensions, see *Features added to earlier extensions* on page A2-89.

A2.7.1 Architectural features added by Armv8.4

An implementation of the Armv8.4 extension must include all of the features that this section describes as mandatory. Such an implementation is also called an implementation of the Armv8.4 architecture.

The Armv8.4 architecture extension adds the following architectural features, which are identified by the architectural feature name and a short description of the feature:

ARMv8.4-DIT, Data Independent Timing instructions

ARMv8.4-DIT provides independent timing for data processing instructions with the addition of the PSTATE.DIT and CPSR.DIT fields.

This feature is supported in AArch64 and AArch32 states.

This feature is mandatory in Armv8.4 implementations.

The following fields identify the support for ARMv8.4-DIT:

- ID_AA64PFR0_EL1.DIT.
- ID_PFR0_EL1.DIT.
- ID_PFR0.DIT.

For more information, see:

- *About PSTATE.DIT* on page B1-113.
- *About the DIT bit* on page E1-3959.

ARMv8.4-CondM, Condition flag Manipulation

ARMv8.4-CondM provides instructions which manipulate the PSTATE.{N,Z,C,V} flags.

These instructions are added to the A64 instruction set only.

This feature is mandatory in Armv8.4 implementations.

The ID_AA64ISAR0_EL1.TS field identifies the presence of ARMv8.4-CondM.

For more information, see *Flag manipulation instructions* on page C3-233.

ARMv8.4-RCPC, Armv8.4 enhancements to weaker release consistency

ARMv8.4-RCPC provides versions of `LDAPR` and `STLR` with a 9-bit unscaled signed immediate offset.

These instructions are added to the A64 instruction set only.

This feature is mandatory in Armv8.4 implementations.

The ID_AA64ISAR1_EL1.LRCPC field identifies the presence of ARMv8.4-RCPC.

For more information, see:

- *Changes to single-copy atomicity in Armv8.4* on page B2-119.
- *Non-exclusive Load-Acquire and Store-Release instructions* on page C3-215.
- *A64 instructions that are changed in Debug state* on page H2-6937.

ARMv8.4-LSE, Large System Extensions

ARMv8.4-LSE introduces changes to single-copy atomicity requirements for loads and stores, and changes to alignment requirements for loads and stores.

This feature is supported in AArch64 state only.

This feature is mandatory in Armv8.4 implementations.

The ID_AA64MMFR2_EL1.AT field identifies the support for ARMv8.4-LSE.

For more information, see:

- *Requirements for single-copy atomicity* on page B2-118.
- *Alignment of data accesses* on page B2-148.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ARMv8.4-TLBI, TLB maintenance and TLB range instructions

ARMv8.4-TLBI provides TLBI maintenance instructions that extend to the Outer Shareable domain and TLBI invalidation instructions that apply to a range of input addresses.

This feature is supported in AArch64 state only.

This feature is mandatory in Armv8.4 implementations.

The field ID_AA64ISAR0_EL1.TLB identifies the presence of ARMv8.4-TLBI.

For more information, see:

- *TLB maintenance instruction syntax* on page D5-2671.
- *TLB range maintenance instructions* on page D5-2679.

ARMv8.4-TTL, Translation Table Level

ARMv8.4-TTL provides the TTL field to indicate the level of translation table walk holding the leaf entry for the address that is being invalidated. This field is provided in all TLB maintenance instructions that take a VA or an IPA argument.

This feature is supported in AArch64 state only.

This feature is mandatory in Armv8.4 implementations.

The field ID_AA64MMFR2_EL1.TTL identifies the presence of ARMv8.4-TTL.

For more information, see:

- *TLB maintenance instruction syntax* on page D5-2671.
- *TLB range maintenance instructions* on page D5-2679.

ARMv8.4-S2FWB, Stage 2 forced Write-Back

ARMv8.4-S2FWB reduces the requirement of additional cache maintenance instructions in systems where the data Cacheability attributes used by the Guest operating system are different from those expected by the Hypervisor.

This feature is supported in AArch64 state.

This feature is mandatory in Armv8.4 implementations.

The ID_AA64MMFR2_EL1.FWB field identifies the support for ARMv8.4-S2FWB.

For more information, see:

- *Memory region attributes* on page D5-2628.
- *The stage 2 memory region attributes, EL1&0 translation regime* on page D5-2630.

ARMv8.4-TTST, Small Translation tables

ARMv8.4-TTST relaxes the lower limit on the size of translation tables, by increasing the maximum permitted value of the T1SZ and T0SZ fields in TCR_EL1, TCR_EL2, TCR_EL3, VTCR_EL2 and VSTCR_EL2.

This feature is supported in AArch64 state only.

This feature is mandatory in Armv8.4 implementations or if ARMv8.4-SecEL2 is implemented.

This feature is optional if ARMv8.4-SecEL2 is not implemented.

The ID_AA64MMFR2_EL1.ST field identifies the support for ARMv8.4-TTST.

For more information, see:

- *Input address size* on page D5-2549.
- *Overview of the VMSAv8-64 address translation stages* on page D5-2564.

ARMv8.4-TTRem, Change in size of translation table mappings

ARMv8.4-TTRem provides support to identify the requirements of hardware to have break-before-make sequences when changing between block size for a translation.

This feature is supported in AArch64 state only.

This feature is mandatory in Armv8.4 implementations.

The ID_AA64MMFR2_EL1.BBM field identifies the support for ARMv8.4-TTRem.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For more information, see:

- *Memory attribute fields in the VMSAv8-64 translation table format descriptors* on page D5-2599.
- *Support levels for changing block size* on page D5-2670.

ARMv8.4-SecEL2, Secure EL2

ARMv8.4-SecEL2 permits EL2 to be implemented in Secure state. When Secure EL2 is enabled, a translation regime is introduced that follows the same format as the other Secure translation regimes.

This feature is not supported if EL2 is using AArch32.

This feature is mandatory in Armv8.4 implementations which implement both EL2 and Secure state.

The ID_AA64PFR0_EL1.SEL2 field identifies the support for ARMv8.4-SecEL2.

For more information, see:

- *Virtualization* on page D1-2320.
- *The VMSAv8-64 address translation system* on page D5-2540.

ARMv8.4-NV, Enhanced support for Nested Virtualization

Armv8.4 supports nested virtualization by redirecting register accesses that would be trapped to EL1 and EL2 to access memory instead. The address of the memory access depends on information held in introduced register, VNCR_EL2.

This feature is supported in AArch64 state only.

This feature is optional in Armv8.4 implementations.

The ID_AA64MMFR2_EL1.NV field identifies the support for ARMv8.4-NV.

For more information, see *Enhanced support for nested virtualization* on page D5-2646.

ARMv8.4-IDST, ID Space Trap handling

ARMv8.4-IDST causes all AArch64 read accesses to the feature ID space when exceptions are generated, to be reported in ESR_ELx using the EC code `0x18`.

This feature is supported in AArch64 state only.

This feature is mandatory in Armv8.4 implementations.

The ID_AA64MMFR2_EL1.IDS field identifies the support for ARMv8.4-IDST.

ARMv8.4-CNTSC, Generic Counter Scaling

ARMv8.4-CNTSC adds a scaling register to the memory-mapped counter module that allows the frequency of the counter that is generated to be scaled from the basic frequency reported in the counter ID mechanisms.

This feature is supported in AArch64 and AArch32 states.

This feature is optional in Armv8.4 implementations.

The CNTID.CNTSC field identifies the support for ARMv8.4-CNTSC.

For more information, see:

- *CNTCR, Counter Control Register* on page I5-7369.

ARMv8.4-Debug, Armv8.4 Debug relaxations and extensions

ARMv8.4-Debug covers a selection of mandatory changes, including:

- The fields MDCR_EL3.{EPMAD, EDAD} control Non-secure access to the debug and PMU registers. The bus master is responsible for other debug authentication.
- The feature ARMv8.0-DoubleLock has been introduced. See *Features added to earlier extensions* on page A2-89.
- The Software Lock is obsolete.
- Non-invasive Debug controls are relaxed.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Secure and Non-secure views of the debug registers are enabled.

The fields that identify the support for ARMv8.4-Debug are:

- ID_AA64DFR0_EL1.DebugVer
- DBGDIDR.Version
- ID_DFR0_EL1.{CopSDbg, CopDbg}
- ID_DFR0.{CopSDbg, CopDbg}
- EDDEVARCH.ARCHID

For more information, see:

- *Definition and constraints of a debugger in the context of external debug* on page H1-6922
- *External debug interface register access permissions* on page H8-7057

ARMv8.4-Trace, Armv8.4 Self-hosted Trace Extensions

ARMv8.4-Trace adds controls of trace in a self-hosted system through System registers.

The feature provides:

- Control of Exception levels and Security states where trace generation is prohibited.
- Control of whether an offset is used for the timestamp recorded with trace information.
- A context synchronization instruction `TSB CSYNC` which can be used to prevent reordering of trace operation accesses with respect to other accesses of the same System registers.

If an ETM Architecture PE Trace Unit is implemented, this feature is mandatory, and the ETM PE Trace Unit must implement System register access to its control registers. If a different PE Trace Unit is implemented, this feature is optional.

The reset state of the PE has prohibited regions controlled by the feature and not the external authentication signals. An external trace controller must override the internal controls before enabling trace, including trace from reset. This is a change from previous trace architectures and is not backwards-compatible.

The fields that identify the support for ARMv8.4-Trace are:

- ID_AA64DFR0_EL1.TraceFilt,
- ID_DFR0_EL1.TraceFilt
- ID_DFR0.TraceFilt
- EDDFR.TraceVer
- ID_AA64DFR0_EL1.TraceVer

For more information, see:

- Chapter D3 *AArch64 Self-hosted Trace*.
- Chapter G3 *AArch32 Self-hosted Trace*.

ARMv8.4-PMU, Armv8.4 PMU Extensions

ARMv8.4-PMU introduces the PMMIR_EL1 and PMMIR registers.

This feature is supported in AArch64 and AArch32 states.

The Performance Monitors Extension is an optional feature, but if it is implemented, an Armv8.4 implementation must include ARMv8.4-PMU.

The fields that identify the support for ARMv8.4-PMU are:

- ID_AA64DFR0_EL1.PMUVer.
- ID_DFR0_EL1.Perfmon.
- ID_DFR0.Perfmon.
- EDDFR.PMUVer,

For more information, see *PMU events and event numbers* on page D7-2716.

ARMv8.4-RAS, Armv8.4 RAS Extension

ARMv8.4-RAS implements RAS System Architecture v1.1 and adds support for:

- ARMv8.4-DFE.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Simplifications to ERR<n>STATUS.
- Additional ERR<n>MISC<m> registers.
- The optional RAS Common Fault Injection Model Extension.

This feature is supported in AArch64 and AArch32 states.

This feature is mandatory in Armv8.4 implementations.

The following fields identify the support or partial support for ARMv8.4-RAS:

- ID_AA64PFR0_EL1.RAS.
- ID_AA64PFR1_EL1.RAS_frac.
- ID_PFR0_EL1.RAS.
- ID_PFR2_EL1.RAS_frac.
- ID_PFR0.RAS.
- ID_PFR2.RAS_frac.

For more information, see:

- *The Reliability, Availability, and Serviceability Extension (RAS Extension)* on page A2-97.
- *ARM® Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for the ARMv8-A architecture profile*.

ARMv8.4-DFE, Armv8.4 Double Fault Extension

ARMv8.4-DFE provides two controls:

- SCR_EL3.EASE.
- SCR_EL3.NMEA.

This feature is supported in AArch64 state only.

This feature is mandatory in Armv8.4 implementations if EL3 is implemented and EL3 uses AArch64. Otherwise, it is not implemented.

This feature is implemented if ID_AA64PFR0_EL1.RAS >= `0b0010` and the implementation includes EL3 using AArch64.

For more information, see:

- *The Reliability, Availability, and Serviceability Extension (RAS Extension)* on page A2-97.
- *ARM® Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for the ARMv8-A architecture profile*.

A2.7.2 Features added to earlier extensions

The existing functionality of OS Double Lock is added as a feature mnemonic in Armv8.0, see *ARMv8.0-DoubleLock* on page A2-66.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A2.8 The Armv8.5 architecture extension

The Armv8.5 architecture extension adds architectural features and additional requirements, see:

- *Architectural features added by Armv8.5*
- *Additional requirements of Armv8.5* on page A2-92

Features are also added to earlier architecture extensions, see *Features added to earlier extensions* on page A2-93.

A2.8.1 Architectural features added by Armv8.5

An implementation of the Armv8.5 extension must include all of the features that this section describes as mandatory. Such an implementation is also called an implementation of the Armv8.5 architecture.

The Armv8.5 architecture extension adds the following architectural features, which are identified by the architectural feature name and a short description of the feature:

ARMv8.5-CondM, Armv8.5 Condition flag manipulation

ARMv8.5-CondM provides instructions that convert between the PSTATE condition flag format used by the FCMP instruction and an alternative format described in *Relationship between ARM format and alternative format PSTATE condition flags* on page C6-774.

These instructions are added to the A64 instruction set only.

This feature is mandatory in Armv8.5 implementations.

The ID_AA64ISAR0_EL1.TS field identifies the presence of ARMv8.5-CondM.

For more information, see:

- *Flag manipulation instructions* on page C3-233.
- *Relationship between ARM format and alternative format PSTATE condition flags* on page C6-774.

ARMv8.5-FRINT, Armv8.5 Floating-point to integer

ARMv8.5-FRINT provides instructions that round a floating-point number to an integral valued floating-point number that fits in a 32-bit or 64-bit integer number range.

These instructions are added to the A64 instruction set only.

This feature requires SIMD&FP, and is mandatory in Armv8.5 implementations when SIMD&FP is implemented.

The ID_AA64ISAR1_EL1.FRINTTS identifies the presence of ARMv8.5-FRINT.

For more information, see *Floating-point round to integral value* on page C3-242.

ARMv8.5-CSEH, Armv8.5 Context synchronization and exception handling

ARMv8.5-CSEH provides a mechanism to control whether exception entry and exception return are context synchronization events. Fields in the SCTLR_ELx registers enable and disable context synchronization at exception entry and return at an Exception level.

This feature is supported in AArch64 state only.

This feature is optional in Armv8.5 implementations.

The ID_AA64MMFR0_EL1.ExS identifies the presence of ARMv8.5-CSEH.

For more information, see:

- *SCTLR_EL1, System Control Register (EL1)* on page D13-3393, SCTLR_EL2 and SCTLR_EL3.
- *Context synchronization event* on page Glossary-8112

ARMv8.5-GTG, Armv8.5 Guest translation granule size

ARMv8.5-GTG allows a hypervisor to support different granule sizes for stage 2 and stage 1 translation. and allows a nested hypervisor to determine what stage 2 granule sizes are available.

This feature is supported in AArch64 state only.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This feature is mandatory in Armv8.5 implementations.

The ID_AA64MMFR0_EL1.{TGran16_2, TGran64_2, TGran4_2} fields identify whether each of the granule sizes is supported for stage 2 translation. The ID_AA64MMFR0_EL1.{TGran16, TGran64, TGran4} fields identify whether each of the granule sizes is supported for stage 1 translations.

For more information, see *Memory translation granule size* on page D5-2554.

ARMv8.5-BTI, Armv8.5 Branch target identification

ARMv8.5-BTI allows memory pages to be guarded against the execution of instructions that are not the intended target of a branch. To do this, it introduces:

- The GP field, which denotes the blocks and pages in stage 1 translation tables that are guarded pages,
- The PSTATE.BTYPE field, which is used to determine whether an access to a guarded memory region will generate a Branch Target exception.
- The `BTI` instruction, which is used to guard against the execution of instructions that are not the intended target of a branch.

This feature is supported in AArch64 state only.

This feature is mandatory in Armv8.5 implementations.

The ID_AA64PFR1_EL1.BT field identifies the presence of ARMv8.5-BTI.

For more information, see:

- *Exception entry* on page D1-2339.
- *Synchronous exception types, routing and priorities* on page D1-2354.
- *VMSAv8-64 translation table level 0, level 1, and level 2 descriptor formats* on page D5-2593.
- *About PSTATE.BTYPE* on page D5-2608.
- *Effect of entering Debug state on PSTATE* on page H2-6935.

ARMv8.5-E0PD, Armv8.5 Preventing EL0 access to halves of address maps

ARMv8.5-E0PD prevents access at EL0 to half of the addresses in the memory map.

This feature is supported in AArch64 state only. When EL1 is using AArch64 state, this feature affects access to EL0, in either Execution state.

This feature is mandatory in Armv8.5 implementations.

The ID_AA64MMFR2_EL1.E0PD field identifies support for ARMv8.5-E0PD.

For more information, see:

- *Preventing EL0 access to halves of the address map* on page D5-2610.
- TCR_EL1.{E0PD0, E0PD1}.
- TCR_EL2.{E0PD0, E0PD1}.

ARMv8.5-RNG, Armv8.5 Random number generator

ARMv8.5-RNG introduces the RNDR and RNDRRS registers. Reads to these registers return a 64-bit random number. A read to RNDRRS will cause a reseeding of the random number before the generation of the random number that is returned.

This feature is supported in AArch64 state only.

This feature is optional in Armv8.5 implementations.

The ID_AA64ISAR0_EL1.RNDR field identifies support for ARMv8.5-RNG.

- *Effect of random number generation instructions on Condition flags* on page C6-774.
- Appendix K12 *Random Number Generation*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ARMv8.5-MemTag, Armv8.5 Memory Tagging Extension

ARMv8.5-MemTag provides architectural support for runtime, always-on detection of various classes of memory error to aid with software debugging to eliminate vulnerabilities arising from memory-unsafe languages.

This feature is supported in AArch64 state only.

This feature is optional in Armv8.5 implementations.

The field that identifies the support for ARMv8.5-MemTag is ID_AA64PFR1_EL1.MTE.

For more information, see:

- Chapter D6 *Armv8.5 Memory Tagging Extension*.
- Chapter B2 *The AArch64 Application Level Memory Model*.
- *PMU events and event numbers* on page D7-2716.
- Chapter D9 *The Statistical Profiling Extension*.
- Chapter H2 *Debug State*.

ARMv8.5-PMU, Armv8.5 PMU Extensions

ARMv8.5-PMU extends event counters to 64-bit event counters, and adds mechanisms to disable the cycle counter in Secure state and in EL2.

ARMv8.5-PMU relaxes the behavior of PMCR.{IMP, IDCODE}, and deprecates use of these fields.

This feature is supported in AArch64 and AArch32 states.

The Performance Monitors Extension is an optional feature, but if it is implemented, an Armv8.5 implementation must include ARMv8.5-PMU.

The fields that identify the support for ARMv8.5-PMU are:

- ID_AA64DFR0_EL1.PMUVer.
- ID_DFR0_EL1.PerfMon.
- ID_DFR0.PerfMon.
- EDDFR.PMUVer,

For more information, see:

- *Behavior on overflow* on page D7-2705
- *Prohibiting event counting* on page D7-2709.
- *PMU events and event numbers* on page D7-2716.

A2.8.2 Additional requirements of Armv8.5

The Armv8.5 architecture includes some mandatory changes that are not associated with a feature. These are:

Restrictions on effects of speculation

Further restrictions are placed on execution for:

- Execution prediction instructions that predict addresses or register values
- Data loaded under speculation with a permission or domain fault.
- Any System register read under speculation to a register that is not architecturally accessible from the current Exception level.

For more information, see:

- *Restrictions on the effects of speculation* on page B2-133.
- *Restrictions on the effects of speculation* on page E2-3989.

Changes to CTIDEVARCH, CTIDEVAFF0, and CTIDEVAFF1

CTIDEVARCH, CTIDEVAFF0, and CTIDEVAFF1 must be implemented.

Changes to the input channel gate function

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the Cross Trigger Matrix (CTM) is implemented, the input channel gate function must be implemented.

Deprecation of EDPRCR.CWRR

EDPRCR.CWRR is deprecated.

Mandatory changes are also made to earlier architectural extensions, see *Architectural requirements added to earlier extensions*.

A2.8.3 Features added to earlier extensions

The features that have been added to earlier architectural extensions are:

- *ARMv8.0-SB* on page A2-65.
- *ARMv8.0-SSBS* on page A2-65.
- *ARMv8.0-CSV2* on page A2-65.
- *ARMv8.0-CSV3* on page A2-66.
- *ARMv8.0-PredInv* on page A2-66.
- *ARMv8.0-CP15SDISABLE2* on page A2-66.
- *ARMv8.2-EVT* on page A2-78.
- *ARMv8.2-DCCVADP* on page A2-78.
- *ARMv8.3-SPE* on page A2-83.
- *ARMv8.3-DoPD* on page A2-83.

A2.8.4 Architectural requirements added to earlier extensions

The additional architectural requirement that has been added to earlier extensions is *Prefetch speculation protection* on page A2-67.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A2.9 The Armv8.6 architecture extension

The Armv8.6 architecture extension adds architectural features and additional requirements, see:

- *Architectural features added by Armv8.6*.
- *Additional requirements of Armv8.6* on page A2-95.

Features are also added to earlier architecture extensions, see *Features added to earlier extensions* on page A2-95.

A2.9.1 Architectural features added by Armv8.6

An implementation of the Armv8.6 extension must include all of the features that this section describes as mandatory. Such an implementation is also called an implementation of the Armv8.6 architecture.

The Armv8.6 architecture extension adds the following architectural features, which are identified by the architectural feature name and a short description of the feature:

ARMv8.6-ECV, ARMv8.6 Enhanced Counter Virtualization

ARMv8.6-ECV enhances the Generic Timer architecture.

When executing in AArch64 state or AArch32 state, ARMv8.6-ECV provides:

- Self-synchronizing views of the virtual and physical timers in AArch64 and AArch32 state.
- The ability to scale the generation of the event stream.

When EL2 is using AArch64 state, ARMv8.6-ECV provides:

- An optional offset between the EL1 or EL0 view of physical time, and the EL2 or EL3 view of physical time.
- Traps configurable in CNTHCTL_EL2 that trap EL0 and EL1 accesses to the virtual counter or timer registers, and accesses to the physical timer registers when they are accessed using an EL02 descriptor.

The optional offset to views of physical time, and the configurable traps in CNTHCTL_EL2, both apply to EL1 and EL0 whether EL1 and EL0 are in AArch64 state or AArch32 state.

This feature is mandatory in Armv8.6 implementations.

The ID_AA64MMFR0_EL1.ECV field identifies support for ARMv8.6-ECV. The ID_PFR1_EL1.GenTimer and ID_PFR1.GenTimer fields identify support for self-synchronized counter views in AArch32 state.

For more information, see:

- *Self-hosted trace timestamps* on page D3-2491.
- *The profiling data* on page D9-2790.
- *The AArch64 view of the Generic Timer* on page D11-2842.
- *The AArch32 view of the Generic Timer* on page G6-6064.

ARMv8.6-FGT, ARMv8.6 Fine Grain Traps

ARMv8.6-FGT introduces additional traps to EL2 of EL1 and EL0 access to individual or small groups of System registers and instructions. The traps are independent of existing controls.

This feature is supported in AArch64, and when EL1 is using AArch64, EL0 accesses using AArch32 are also trapped.

This feature is mandatory in Armv8.6 implementations.

The ID_AA64MMFR0_EL1.FGT field identifies support for ARMv8.6-FGT.

For more information, see:

- *Traps to EL3 of EL2 accesses to fine-grained trap registers* on page D1-2393.
- *Traps to EL2 of EL0 and EL1 accesses to the Debug Communications Channel registers* on page D1-2389.
- *Traps to EL3 of EL2, EL1, and EL0 accesses to Debug Communication Channel registers* on page D1-2392.
- *Fine-grained traps to EL2 of EL0 and EL1 accesses to System registers* on page D1-2387.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *Fine-grained traps to EL2 of EL0 and EL1 accesses to the debug, trace, and PMU registers* on page D1-2387.
- *Fine-grained Traps to EL2 of EL0 and EL1 accesses to instructions* on page D1-2387.
- *Fine-grained traps to EL2 of EL0 and EL1 read accesses to Activity Monitors registers* on page D1-2382.

ARMv8.6-TWED, ARMv8.6 Delayed Trapping of WFE

ARMv8.6-TWED introduces support for configurable delayed trapping of the `WFE` instruction.

This feature is supported in AArch64 and AArch32 states.

This feature is optional in Armv8.6 implementations.

ID_AA64MMFR1_EL1.TWED field identifies the presence of ARMv8.6-TWED.

For more information, see *The Wait For Event instruction* on page D1-2398.

ARMv8.6-AMU, ARMv8.6 AMU Extensions

ARMv8.6-AMU introduces support for virtualization of Activity Monitors event counters, and introduces controls to disable access to auxiliary event counters below the highest Exception level.

This feature is supported in AArch32 state and AArch64 state, if the hypervisor is using AArch64.

This feature is optional in Armv8.6 implementations if the optional AMUv1 is implemented.

The fields ID_AA64PFR0_EL1.AMU, ID_PFR0_EL1.AMU and ID_PFR0.AMU identify the presence of ARMv8.6-AMU.

For more information, see Chapter D8 *The Activity Monitors Extension*.

ARMv8.6-MTPMU, ARMv8.6 Multi-threaded PMU Extensions

ARMv8.6-MTPMU introduces controls to disable PMEVTYPER<n>_EL0.MT.

This feature requires at least one of EL2 and EL3. If neither is implemented, this feature is not implemented.

If EL2 or EL3 are implemented, the feature is optional if PMUv3 is implemented.

Multithreaded Armv8.6 implementations with PMUv3 implemented, must implement ARMv8.6-MTPMU to enable any multithreaded event counting.

This feature is supported in AArch64 state and AArch32 state.

The fields ID_AA64DFR0_EL1.MTPMU and ID_DFR1.MTPMU identify the presence of ARMv8.6-MTPMU.

For more information, see:

- *Multithreaded implementations* on page D7-2710.
- MDCR_EL3.MTPME, SDCR.MTPME, MDCR_EL2.MTPME, and HDCR.MTPME.
- *Common event numbers* on page D7-2722.

A2.9.2 Additional requirements of Armv8.6

The Armv8.6 architecture includes some mandatory changes that are not associated with a feature. These are:

Changes to the frequency of the physical counter

The frequency of CNTFRQ_EL0 is standardized to a frequency of 1GHz. This means that the system counter must be implemented at 64 bits. For more information, see:

- *The system counter* on page D11-2840.
- *The system counter* on page G6-6062.

A2.9.3 Features added to earlier extensions

The features that have been added to earlier architectural extensions are:

- *ARMv8.0-DGH* on page A2-67.
- *ARMv8.0-ETS* on page A2-67.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *ARMv8.2-BF16* on page A2-79.
- *ARMv8.2-AA32BF16* on page A2-79.
- *ARMv8.2-I8MM* on page A2-79.
- *ARMv8.2-AA32I8MM* on page A2-80.
- *ARMv8.3-PAuth2* on page A2-83.
- *ARMv8.3-FPAC* on page A2-84.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A2.10 The Reliability, Availability, and Serviceability Extension (RAS Extension)

The RAS Extension is a mandatory extension to the Armv8.2 architecture, and an optional extension to the Armv8.0 and the Armv8.1 architectures.

The RAS Extension improves the dependability of a system by providing:

- Reliability, that is, the continuity of correct service.
- Availability, that is, the readiness for correct service.
- Serviceability, that is, the ability to undergo modifications and repairs.

ID_AA64PFR0_EL1.RAS in AArch64 state, and ID_PFR0.RAS in AArch32 state, indicate whether the RAS Extension is implemented.

The RAS Extension introduces a barrier instruction, the Error Synchronization Barrier (`ESB`), to the A32, T32, and A64 instruction sets.

System registers introduced by the RAS Extension are described in:

- For AArch64, *RAS registers* on page D13-3801.
- For AArch32, *RAS registers* on page G8-6792.

In addition, the RAS Extension introduces a number of memory-mapped registers. These are described in the *ARM® Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for the Armv8-A architecture profile*.

Armv8.2 introduces the following architectural features to the RAS Extension:

- ARMv8.2-IESB.

Armv8.4 introduces the following architectural features to the RAS Extension:

- ARMv8.4-RAS.
- ARMv8.4-DFE.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A2.11 The Statistical Profiling Extension (SPE)

The Statistical Profiling Extension (SPE) is an optional extension introduced by the Armv8.2 architecture. Implementation of the Statistical Profiling Extension requires implementation of at least Armv8.1 of the Armv8-A architecture profile. The Statistical Profiling Extension is only supported in AArch64 state.

The Statistical Profiling Extension provides a non-invasive method of sampling software and hardware using randomized sampling of either architectural instructions, as defined by the instruction set architecture, or by microarchitectural operations.

ID_AA64DFR0_EL1.PMSVer indicates whether the Statistical Profiling Extension is implemented.

For more information see Chapter D9 *The Statistical Profiling Extension*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A2.12 The Scalable Vector Extension (SVE)

The Scalable Vector Extension is an optional extension introduced by the Armv8.2 architecture. SVE is supported in AArch64 state only.

The Scalable Vector Extension provides vector instructions that, primarily, support wider vectors than the Arm Advanced SIMD instruction set. The *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A* describes the SVE.

ID_AA64PFR0_EL1.SVE indicates whether the Scalable Vector Extension is implemented.

The Scalable Vector Extension affects some AArch64 System registers, and those register changes are included in this issue of this Manual, where they are identified as SVE features. SVE also introduces AArch64 System registers, however these do not appear in this manual. For more information about the System registers introduced by SVE, please see the *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*.

The SVE and Advanced SIMD events are documented in Chapter D7 *The Performance Monitors Extension*.

The Scalable Vector Extension introduces the following System registers:

- ID_AA64ZFR0_EL1.
- ZCR_EL1, and an EL2 alias of this register, ZCR_EL12.
- ZCR_EL2.
- ZCR_EL3.

The Scalable Vector Extension modifies the following existing System registers:

- CPACR_EL1.
- CPTR_EL2.
- CPTR_EL3.
- ESR_ELx.
- ID_AA64PFR0_EL1.
- TCR_EL1.
- TCR_EL2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A2.13 The Activity Monitors Extension

The Activity Monitors Extension is an optional extension introduced by the Armv8.4 architecture. AMU is supported in AArch64 and AArch32 states.

The Activity Monitors Extension implements version 1 of the Activity Monitors architecture, AMUv1, which provides a function similar to a subset of the existing Performance Monitors Extension functionality, intended for system management use rather than debugging and profiling.

The Activity Monitors Extension implements a System register interface to the Activity Monitors registers, and also supports an optional external memory-mapped interface.

The fields that identify the presence of the Activity Monitors Extension are:

- ID_AA64PFR0_EL1.AMU.
- ID_PFR0_EL1.AMU.
- ID_PFR0.AMU.
- EDPFR.AMU.

For more information, see Chapter D8 *The Activity Monitors Extension*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A2.14 The Memory Partitioning and Monitoring (MPAM) Extension

The Memory Partitioning and Monitoring Extension is an optional extension introduced by the Armv8.4 architecture and requires implementation of at least Armv8.2 of the Armv8-A architecture profile. MPAM is supported in AArch64 state only.

The MPAM Extension provides a framework for memory-system component controls that partition one or more of the performance resources of the component.

The fields that identify the presence of the MPAM Extension are:

- ID_AA64PFR0_EL1.MPAM.
- EDPFR.MPAM.

For more information, see *ARM® Architecture Reference Manual Supplement, Memory System Resource Partitioning and Monitoring (MPAM), for ARMv8-A*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Part B

The AArch64 Application Level Architecture

Chapter B1
The AArch64 Application Level Programmers' Model

- *About the Application level programmers' model* on page B1-106.
- *Registers in AArch64 Execution state* on page B1-107.
- *Software control features and EL0* on page B1-112.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

B1.1 About the Application level programmers' model

This chapter contains the programmers' model information required for application development.

The information in this chapter is distinct from the system information required to service and support application execution under an operating system, or higher level of system software. However, some knowledge of the system information is needed to put the Application level programmers' model into context.

Depending on the implementation choices, the architecture supports multiple levels of execution privilege, indicated by different *Exception levels* that number upwards from EL0 to EL3. EL0 corresponds to the lowest privilege level and is often described as unprivileged. The Application level programmers' model is the programmers' model for software executing at EL0. For more information see *Exception levels* on page D1-2314.

System software determines the Exception level, and therefore the level of privilege, at which software runs. When an operating system supports execution at both EL1 and EL0, an application usually runs unprivileged at EL0. This:

- Permits the operating system to allocate system resources to an application in a unique or shared manner.
- Provides a degree of protection from other processes, and so helps protect the operating system from malfunctioning software.

This chapter indicates where some system level understanding is necessary, and where relevant it gives a reference to the system level description.

Execution at any Exception level above EL0 is often referred to as privileged execution.

For more information on the system level view of the architecture refer to Chapter D1 *The AArch64 System Level Programmers' Model*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

B1.2 Registers in AArch64 Execution state

This section describes the registers and process state visible at EL0 when executing in the AArch64 state. It includes the following:

- *Registers in AArch64 state*
- *Process state, PSTATE* on page B1-108
- *System registers* on page B1-110

B1.2.1 Registers in AArch64 state

In the AArch64 application level view, an Arm processing element has:

R0-R30 31 general-purpose registers, R0 to R30. Each register can be accessed as:

- A 64-bit general-purpose register named X0 to X30.
- A 32-bit general-purpose register named W0 to W30.

See the register name mapping in Figure B1-1.

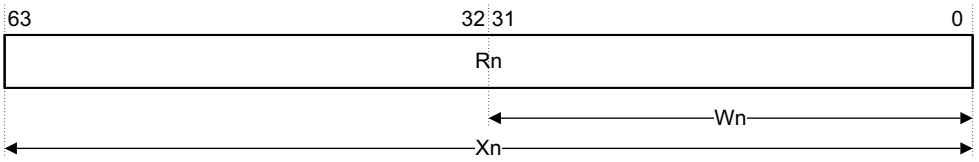


Figure B1-1 General-purpose register naming

The X30 general-purpose register is used as the procedure call link register.

Note

In instruction encodings, the value `0b11111` (31) is used to indicate the ZR (zero register). This indicates that the argument takes the value zero, but does not indicate that the ZR is implemented as a physical register.

SP A 64-bit dedicated Stack Pointer register. The least significant 32 bits of the stack-pointer can be accessed via the register name WSP.

The use of SP as an operand in an instruction, indicates the use of the current stack pointer.

Note

Stack pointer alignment to a 16-byte boundary is configurable at EL1. For more information see the *Procedure Call Standard for the Arm 64-bit Architecture*.

PC A 64-bit Program Counter holding the address of the current instruction.

Software cannot write directly to the PC. It can only be updated on a branch, exception entry or exception return.

Note

Attempting to execute an A64 instruction that is not word-aligned generates a PC alignment fault, see *PC alignment checking* on page D1-2333.

V0-V31 32 SIMD&FP registers, V0 to V31. Each register can be accessed as:

- A 128-bit register named Q0 to Q31.
- A 64-bit register named D0 to D31.
- A 32-bit register named S0 to S31.
- A 16-bit register named H0 to H31.
- An 8-bit register named B0 to B31.
- A 128-bit vector of elements.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- A 64-bit vector of elements.

Where the number of bits described by a register name does not occupy an entire SIMD&FP register, it refers to the least significant bits. See Figure B1-2.

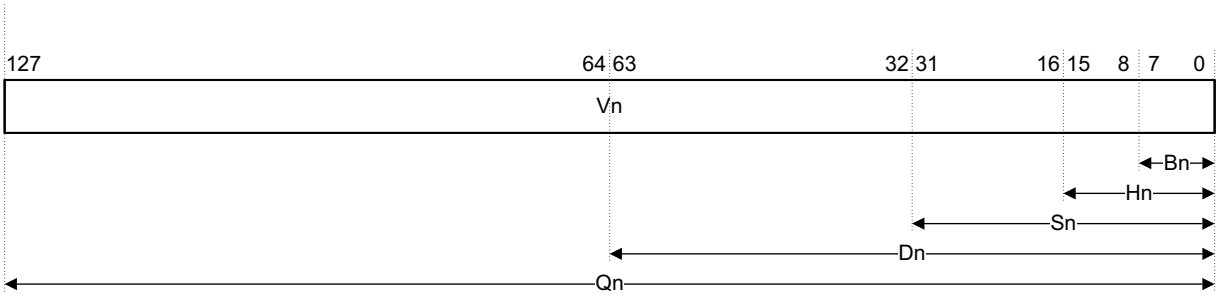


Figure B1-2 SIMD and floating-point register naming

For more information about data types and vector formats, see *Supported data types* on page A1-40.

FPCR, FPSR Two SIMD and floating-point control and status registers, FPCR and FPSR.

See *Registers for instruction processing and exception handling* on page D1-2323 for more information on the registers.

Pseudocode description of registers in AArch64 state

In the pseudocode functions that access registers:

- The assignment form is used for register writes.
- The non-assignment for register reads.

The uses of the `X[]` function are:

- Reading or writing X0-X30, using `n` to index the required register.
- Reading the zero register ZR, accessed as `X[31]`.

Note

The pseudocode use of X[31] to represent the zero register does not indicate that hardware must implement this register.

The AArch64 `SP[]` function is used to read or write the current SP.

The AArch64 `PC[]` function is used to read the PC.

The AArch64 `V[]` function is used to read or write the Advanced SIMD and floating-point registers V0-V31, using a parameter `n` to index the required register.

The AArch64 `Vpart[]` function is used to read or write a part of one of V0-V31, using a parameter `n` to index the required register, and a parameter `part` to indicate the required part of the register, see the function description for more information.

The `SP[]`, `PC[]`, `V[]`, and `Vpart[]` functions are defined in Chapter J1 *Armv8 Pseudocode*.

B1.2.2 Process state, PSTATE

Process state or PSTATE is an abstraction of process state information. All of the instruction sets provide instructions that operate on elements of PSTATE.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following PSTATE information is accessible at EL0:

The Condition flags

Flag-setting instructions set these. They are:

N Negative Condition flag. If the result of the instruction is regarded as a two's complement signed integer, the PE sets this to:

- 1 if the result is negative.
- 0 if the result is positive or zero.

Z Zero Condition flag. Set to:

- 1 if the result of the instruction is zero.
- 0 otherwise.

A result of zero often indicates an equal result from a comparison.

C Carry Condition flag. Set to:

- 1 if the instruction results in a carry condition, for example an unsigned overflow that is the result of an addition.
- 0 otherwise.

V Overflow Condition flag. Set to:

- 1 if the instruction results in an overflow condition, for example a signed overflow that is the result of an addition.
- 0 otherwise.

Conditional instructions test the N, Z, C and V Condition flags, combining them with the Condition code for the instruction to determine whether the instruction must be executed. In this way, execution of the instruction is conditional on the result of a previous operation. For more information about conditional execution, see *Condition flags and related instructions* on page C6-773.

The exception masking bits

D Debug exception mask bit. When EL0 is enabled to modify the mask bits, this bit is visible and can be modified. However, this bit is architecturally ignored at EL0.

A SError interrupt mask bit.

I IRQ interrupt mask bit.

F FIQ interrupt mask bit.

For each bit, the values are:

0 Exception not masked.

1 Exception masked.

Access at EL0 using AArch64 state depends on SCTLR_EL1.UMA. See *Traps to EL1 of EL0 accesses to the PSTATE.{D, A, I, F} interrupt masks* on page D1-2377.

See *Process state, PSTATE* on page D1-2330 for the system level view of PSTATE.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing PSTATE fields at EL0

At EL0 using AArch64 state, PSTATE fields can be accessed using Special-purpose registers that can be directly read using the MRS instruction and directly written using the MSR (register) instructions. Table B1-1 shows the Special-purpose registers that access the PSTATE fields that hold AArch64 state when the PE is at EL0 using AArch64. All other PSTATE fields do not have direct read and write access at EL0.

Table B1-1 Accessing PSTATE fields at EL0 using MRS and MSR (register)

Special-purpose register	PSTATE fields
NZCV	N, Z, C, V
DAIF	D, A, I, F

Software can also use the MSR (immediate) instruction to directly write to PSTATE.{D, A, I, F}. Table B1-2 shows the MSR (immediate) operands that can directly write to PSTATE.{D, A, I, F} when the PE is at EL0 using AArch64 state.

Table B1-2 Accessing PSTATE.{D, A, I, F} at EL0 using MSR (immediate)

Operand	PSTATE fields	Notes
DAIFSet	D, A, I, F	Directly sets any of the PSTATE.{D,A, I, F} bits to 1
DAIFClr	D, A, I, F	Directly clears any of the PSTATE.{D, A, I, F} bits to 0

However, access to the PSTATE.{D, A, I, F} fields at EL0 using AArch64 state depends on SCTLR_EL1.UMA. *Traps to EL1 of EL0 accesses to the PSTATE.{D, A, I, F} interrupt masks* on page D1-2377.

Writes to the PSTATE fields have side-effects on various aspects of the PE operation. All of these side-effects, are guaranteed:

- Not to be visible to earlier instructions in the execution stream.
- To be visible to later instructions in the execution stream.

B1.2.3 System registers

System registers provide support for execution control, status and general system configuration. The majority of the System registers are not accessible at EL0.

However, some System registers can be configured to allow access from software executing at EL0. Any access from EL0 to a System register with the access right disabled causes the instruction to behave as UNDEFINED. The registers that can be accessed from EL0 are:

Cache ID registers The CTR_EL0 and DCZID_EL0 registers provide implementation parameters for EL0 cache management support.

Debug registers A debug communications channel is supported by the MDCCSR_EL0, DBGDTR_EL0, DBGDTRRX_EL0 and DBGDTRTX_EL0 registers.

Performance Monitors registers

The Performance Monitors Extension provides counters and configuration registers. Software executing at EL1 or a higher Exception level can configure some of these registers to be accessible at EL0.

For more details, see Chapter D7 *The Performance Monitors Extension*.

Activity Monitors registers

The Activity Monitors Extension provides counters and configuration registers. Software executing at EL1 or a higher Exception level can configure these registers to be accessible at EL0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For more details, see Chapter D8 *The Activity Monitors Extension*.

Thread ID registers

The TPIDR_EL0 and TPIDRRO_EL0 registers are two thread ID registers with different access rights.

Timer registers

In Armv8 the following operations are performed:

- Read access to the system counter clock frequency using CNTFRQ_EL0.
- Physical and virtual timer count registers, CNTPCT_EL0 and CNTVCT_EL0.
- Physical up-count comparison, down-count value and timer control registers, CNTP_CVAL_EL0, CNTP_TVAL_EL0, and CNTP_CTL_EL0.
- Virtual up-count comparison, down-count value and timer control registers, CNTV_CVAL_EL0, CNTV_TVAL_EL0, and CNTV_CTL_EL0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

B1.3 Software control features and EL0

The following sections describe the EL0 view of the Armv8 software control features:

- *Exception handling*
- *Wait for Interrupt and Wait for Event*
- *The YIELD instruction*
- *Application level cache management* on page B1-113
- *Instructions relating to Debug* on page B1-113

B1.3.1 Exception handling

In the Arm architecture, an *exception* causes a change of program flow. Execution of an exception handler starts, at an Exception level higher than EL0, from a defined vector that relates to the exception taken.

Exceptions include:

- Interrupts.
- Memory system aborts.
- Exceptions generated by attempting to execute an instruction that is UNDEFINED.
- System calls.
- Secure monitor or Hypervisor traps.
- Debug exceptions.

Most details of exception handling are not visible to application level software, and are described in Chapter D1 *The AArch64 System Level Programmers' Model*.

The `SVC` instruction causes a Supervisor Call exception. This provides a mechanism for unprivileged software to make a system call to an operating system.

The `BRK` instruction generates a Breakpoint Instruction exception. This provides a mechanism for debugging software using debugger executing on the same PE, see *Breakpoint Instruction exceptions* on page D2-2437.

Note

The `BRK` instruction is supported only in the A64 instruction set. The equivalent instruction in the T32 and A32 instruction sets is `BKPT`.

B1.3.2 Wait for Interrupt and Wait for Event

Issuing a `WFI` instruction indicates that no further execution is required until a WFI wake-up event occurs, see *Wait For Interrupt* on page D1-2400. This permits entry to a low-power state.

Issuing a `WFE` instruction indicates that no further execution is required until a WFE wake-up event occurs, see *Wait for Event mechanism and Send event* on page D1-2397. This permits entry to a low-power state.

B1.3.3 The YIELD instruction

The `YIELD` instruction provides a hint that the task performed by a thread is of low importance so that it could yield, see *YIELD* on page C6-1395. This mechanism can be used to improve overall performance in a *Symmetric Multithreading* (SMT) or *Symmetric Multiprocessing* (SMP) system.

Examples of when the `YIELD` instruction might be used include a thread that is sitting in a spin-lock, or where the arbitration priority of the snoop bit in an SMP system is modified. The `YIELD` instruction permits binary compatibility between SMT and SMP systems.

The `YIELD` instruction is a `NOP` hint instruction.

The `YIELD` instruction has no effect in a single-threaded system, but developers of such systems can use the instruction to flag its intended use for future migration to a multiprocessor or multithreading system. Operating systems can use `YIELD` in places where a yield hint is wanted, knowing that it will be treated as a NOP if there is no implementation benefit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

B1.3.4 Application level cache management

A small number of cache management instructions can be enabled at EL0 from higher levels of privilege using the SCTLR_EL1 System register. Any access from EL0 to an operation with the access right disabled causes the instruction to behave as UNDEFINED.

About the available operations, see *Application level access to functionality related to caches* on page B2-144.

B1.3.5 Instructions relating to Debug

Exception handling on page B1-112 refers to the `BRK` instruction, which generates a Breakpoint Instruction exception. In addition, in both AArch64 state and AArch32 state, the `HLT` instruction causes the PE to halt execution and enter Debug state. This provides a mechanism for debugging software using a debugger that is external to the PE, see Chapter H1 *About External Debug*.

Note

In AArch32 state, previous versions of the architecture defined the `DBG` instruction, that could provide a hint to the debug system. In Armv8, this instruction executes as a `NOP`. Arm deprecates the use of the `DBG` instruction.

B1.3.6 About PSTATE.DIT

When the value of PSTATE.DIT is 1:

- The instructions listed in DIT are required to have;
  - Timing which is independent of the values of the data supplied in any of its registers, and the values of the NZCV flags.
  - Responses to asynchronous exceptions which do not vary based on the values supplied in any of their registers, or the values of the NZCV flags.
- All loads and stores must have their timing insensitive to the value of the data being loaded or stored.

Note

- Arm recommends that the ARMv8.3-PAuth instructions do not have their timing dependent on the key value used in the pointer authentication, regardless of the PSTATE.DIT bit.
- When the value of PSTATE.DIT is 0, the architecture makes no statement about the timing properties of any instructions. However, it is likely that these instructions have timing that is invariant of the data in many situations.

A corresponding DIT bit is added to PSTATE in AArch64 state, and to CPSR in AArch32 state.

On an exception that is taken from AArch64 state to AArch64 state, PSTATE.DIT is copied to SPSR_ELx.DIT.

On an exception that is taken from AArch32 state to AArch64 state, CPSR.DIT is copied to SPSR_ELx.DIT.

On an exception return from AArch64 state:

- SPSR_ELx.DIT is copied to PSTATE.DIT, when the target Exception level is in AArch64 state.
- SPSR_ELx.DIT is copied to CPSR.DIT, when the target Exception level is in AArch32 state.

PSTATE.DIT can be written and read at all exception levels.

Note

- PSTATE.DIT is unchanged on entry into Debug state.
- PSTATE.DIT is not guaranteed to have any effect in Debug state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter B2
The AArch64 Application Level Memory Model

This chapter gives an application level view of the memory model. It contains the following sections:

- *About the Arm memory model* on page B2-116.
- *Atomicity in the Arm architecture* on page B2-118.
- *Definition of the Armv8 memory model* on page B2-123.
- *Caches and memory hierarchy* on page B2-143.
- *Alignment support* on page B2-148.
- *Endian support* on page B2-150.
- *Memory types and attributes* on page B2-153.
- *Mismatched memory attributes* on page B2-164.
- *Synchronization and semaphores* on page B2-167.

Note

In this chapter, System register names usually link to the description of the register in Chapter D13 *AArch64 System Register Descriptions*, for example SCTLR_EL1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

B2.1 About the Arm memory model

The Arm architecture is a weakly ordered memory architecture that permits the observation and completion of memory accesses in a different order from the program order. The following sections of this chapter provide the complete definition of the Armv8 memory model, this introduction is not intended to contradict the definition found in those sections. In general, the basic principles of the Armv8 memory model are:

- To provide a memory model that has similar weaknesses to those found in the memory models used by high-level programming languages such as C or Java. For example, by permitting independent memory accesses to be reordered as seen by other observers.
- To avoid the requirement for multi-copy atomicity in the majority of memory types.
- The provision of instructions and memory barriers to compensate for the lack of multi-copy atomicity in the cases where it would be needed.
- The use of address, data, and control dependencies in the creation of order so as to avoid having excessive numbers of barriers or other explicit instructions in common situations where some order is required by the programmer or the compiler.
- If ARMv8.5-MemTag is implemented and enabled the definitions of the memory model which apply to data accesses and data apply to Allocation Tag accesses and Allocation tags, unless otherwise specified in Chapter D6 *Armv8.5 Memory Tagging Extension*.

This section contains:

- *Address space*.
- *Memory type overview*.

B2.1.1 Address space

Address calculations are performed using 64-bit registers. However, supervisory software can configure the top eight address bits for use as a tag, as described in *Address tagging in AArch64 state* on page D5-2534. If this is done, address bits[63:56]:

- Are not considered when determining whether the address is valid.
- Are never propagated to the program counter.

Supervisory software determines the valid address range. Attempting to access an address that is not valid generates an MMU fault.

Simple sequential execution of instructions might overflow the valid address range. For more information, see *Virtual address space overflow* on page D4-2497.

Memory accesses use the `Mem[]` function. This function makes an access of the required type. If supervisory software configures the top eight address bits for use as a tag, the top eight address bits are ignored.

The `AccType{}` enumeration defines the different access types.

Note

- Chapter D4 *The AArch64 System Level Memory Model* and Chapter D5 *The AArch64 Virtual Memory System Architecture* include descriptions of memory system features that are transparent to the application, including memory access, address translation, memory maintenance instructions, and alignment checking and the associated fault handling. These chapters also include pseudocode descriptions of these operations.
- For information on the pseudocode that relates to memory accesses, see *Basic memory access* on page D4-2527, *Unaligned memory access* on page D4-2528, and *Aligned memory access* on page D4-2528.

B2.1.2 Memory type overview

Armv8 provides the following mutually-exclusive memory types:

Normal This is generally used for bulk memory operations, both read/write and read-only operations.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Device The Arm architecture forbids *Speculative* reads of any type of Device memory. This means Device memory types are suitable attributes for read-sensitive Locations.

Locations of the memory map that are assigned to peripherals are usually assigned the Device memory attribute.

Device memory has additional attributes that have the following effects:

- They prevent aggregation of reads and writes, maintaining the number and size of the specified memory accesses. See *Gathering* on page B2-159.
- They preserve the access order and synchronization requirements for accesses to a single peripheral. See *Reordering* on page B2-160
- They indicate whether a write can be acknowledged other than at the end point. See *Early Write Acknowledgement* on page B2-161.

For more information on Normal memory and Device memory, see *Memory types and attributes* on page B2-153.

Note

Earlier versions of the Arm architecture defined a single Device memory type and a Strongly-ordered memory type. A *Note* in *Device memory* on page B2-157 describes how these memory types map onto the Armv8 memory types.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

B2.2 Atomicity in the Arm architecture

Atomicity is a feature of memory accesses, described as *atomic* accesses. The Arm architecture description refers to two types of atomicity, *single-copy atomicity* and *multi-copy atomicity*. In the Armv8 architecture, the atomicity requirements for memory accesses depend on the memory type, and whether the access is explicit or implicit. For more information, see:

- *Requirements for single-copy atomicity*.
- *Properties of single-copy atomic accesses* on page B2-120.
- *Multi-copy atomicity* on page B2-120.
- *Requirements for multi-copy atomicity* on page B2-120.
- *Concurrent modification and execution of instructions* on page B2-120.

For more information about the memory types, see *Memory type overview* on page B2-116.

B2.2.1 Requirements for single-copy atomicity

For explicit memory accesses generated from an Exception level the following rules apply:

- A read that is generated by a load instruction that loads a single general-purpose register and is aligned to the size of the read in the instruction is single-copy atomic.
- A write that is generated by a store instruction that stores a single general-purpose register and is aligned to the size of the write in the instruction is single-copy atomic.
- Reads that are generated by a Load Pair instruction that loads two general-purpose registers and are aligned to the size of the load to each register are treated as two single-copy atomic reads, one for each register being loaded.
- Writes that are generated by a Store pair instruction that stores two general-purpose registers and are aligned to the size of the store of each register are treated as two single-copy atomic writes, one for each register being stored.
- Load-Exclusive Pair instructions of two 32-bit quantities and Store-Exclusive Pair instructions of 32-bit quantities are single-copy atomic.
- When the Store-Exclusive of a Load-Exclusive/Store-Exclusive pair instruction using two 64-bit quantities succeeds, it causes a single-copy atomic update of the entire memory location being updated.

  Note

  To atomically load two 64-bit quantities, perform a Load-Exclusive pair/Store-Exclusive pair sequence of reading and writing the same value for which the Store-Exclusive pair succeeds, and use the read values from the Load-Exclusive pair.

- Where translation table walks generate a read of a translation table entry, this read is single-copy atomic.
- For the atomicity of instruction fetches, see *Concurrent modification and execution of instructions* on page B2-120.
- Reads to SIMD and floating-point registers of a single 64-bit or smaller quantity that is aligned to the size of the quantity being loaded are treated as single-copy atomic reads.
- Writes from SIMD and floating-point registers of a single 64-bit or smaller quantity that is aligned to the size of the quantity being stored are treated as single-copy atomic writes.
- Element or Structure Reads to SIMD and floating-point registers of 64-bit or smaller elements, where each element is aligned to the size of the element being loaded, have each element treated as a single-copy atomic read.
- Element or Structure Writes from SIMD and floating-point registers of 64-bit or smaller elements, where each element is aligned to the size of the element being stored, have each element treated as a single-copy atomic store.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Reads to SIMD and floating-point registers of a 128-bit value that is 64-bit aligned in memory are treated as a pair of single-copy atomic 64-bit reads.
- Writes from SIMD and floating-point registers of a 128-bit value that is 64-bit aligned in memory are treated as a pair of single-copy atomic 64-bit writes.
- For unaligned memory accesses, the single-copy atomicity is described in *Alignment of data accesses* on page B2-148.
- The reads and writes of the two words or two double-words accessed by CASP instructions are single-copy atomic at the size of the two words or double-words.

All other memory accesses are regarded as streams of accesses to bytes, and no atomicity between accesses to different bytes is ensured by the architecture.

All accesses to any byte are single-copy atomic.

Note

In AArch64 state, no memory accesses from a DC ZVA have single-copy atomicity of any quantity greater than individual bytes.

If, according to these rules, an instruction is executed as a sequence of accesses, exceptions, including interrupts, can be taken during that sequence, regardless of the memory type being accessed. If any of these exceptions are returned from using their preferred return address, the instruction that generated the sequence of accesses is re-executed, and so any access performed before the exception was taken is repeated. See also *Taking an interrupt or other exception during a multi-access load or store* on page D1-2372.

Note

The exception behavior for these multiple access instructions means that they are not suitable for use for writes to memory for the purpose of software synchronization.

Changes to single-copy atomicity in Armv8.4

Instructions that are introduced in ARMv8.4-RCPC are single-copy atomic when the following conditions are true:

- All bytes being accessed are within the same 16-byte quantity aligned to 16 bytes.
- Accesses are to Inner Write-Back, Outer Write-Back Normal cacheable memory.

Otherwise it is IMPLEMENTATION DEFINED whether they are single-copy atomic.

If ARMv8.4-LSE is implemented, all loads and stores are single-copy atomic when the following conditions are true:

- Accesses are unaligned to their data size but are aligned within a 16-byte quantity that is aligned to 16 bytes.
- Accesses are to Inner Write-Back, Outer Write-Back Normal cacheable memory.

Otherwise it is IMPLEMENTATION DEFINED whether loads and stores are single-copy atomic.

If ARMv8.4-LSE is implemented, `LDP`, `LDNP`, and `STP` instructions that load or store two 64-bit registers are single-copy atomic when the following conditions are true:

- The overall memory access is aligned to 16 bytes.
- Accesses are to Inner Write-Back, Outer Write-Back Normal cacheable memory.

If ARMv8.4-LSE is implemented, `LDP`, `LDNP`, and `STP` instructions that access fewer than 16 bytes are single-copy atomic when the following conditions are true:

- All bytes being accessed are within a 16-byte quantity aligned to 16 bytes.
- Accesses are to Inner Write-Back, Outer Write-Back Normal cacheable memory.

Otherwise it is IMPLEMENTATION DEFINED whether `LDP`, `LDNP`, or `STP` instructions that access fewer than 16 bytes are single-copy atomic.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

B2.2.2 Properties of single-copy atomic accesses

A memory access instruction that is single-copy atomic has the following properties:

1. For a pair of overlapping single-copy atomic store instructions, all of the overlapping writes generated by one of the stores are Coherence-after the corresponding overlapping writes generated by the other store.

2. For a single-copy atomic load instruction L_1 that overlaps a single-copy atomic store instruction S_2, if one of the overlapping reads generated by L_1 Reads-from one of the overlapping writes generated by S_2, then none of the overlapping writes generated by S_2 are Coherence-after the corresponding overlapping reads generated by L_1.

For more information, see *Definition of the Armv8 memory model* on page B2-123.

B2.2.3 Multi-copy atomicity

In a multiprocessing system, writes to a memory location are *multi-copy atomic* if the following conditions are both true:

- All writes to the same location are *serialized*, meaning they are observed in the same order by all observers, although some observers might not observe all of the writes.
- A read of a location does not return the value of a write until all observers observe that write.

Note

Writes that are not coherent are not multi-copy atomic.

B2.2.4 Requirements for multi-copy atomicity

For Normal memory, writes are not required to be multi-copy atomic.

For Device memory, writes are not required to be multi-copy atomic.

The Armv8 memory model is Other-multi-copy atomic. For more information, see *Ordering constraints* on page B2-128.

B2.2.5 Concurrent modification and execution of instructions

The Armv8 architecture limits the set of instructions that can be executed by one thread of execution as they are being modified by another thread of execution without requiring explicit synchronization.

Concurrent modification and execution of instructions can lead to the resulting instruction performing any behavior that can be achieved by executing any sequence of instructions that can be executed from the same Exception level, except where each of the instruction before modification and the instruction after modification is one of a `B`, `BL`, `BRK`, `HVC`, `ISB`, `NOP`, `SMC`, or `SVC` instruction.

For the `B`, `BL`, `BRK`, `HVC`, `ISB`, `NOP`, `SMC`, and `SVC` instructions the architecture guarantees that, after modification of the instruction, behavior is consistent with execution of either:

- The instruction originally fetched.
- A fetch of the modified instruction.

For all other instructions, to avoid UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior, instruction modifications must be explicitly synchronized before they are executed. The required synchronization is as follows:

1. No PE must be executing an instruction when another PE is modifying that instruction.

2. To ensure that the modified instructions are observable, a PE that is writing the instructions must issue the following sequence of instructions and operations:

```
; Coherency example for data and instruction accesses within the same Inner Shareable domain.
; enter this code with <Wt> containing a new 32-bit instruction,
; to be held in Cacheable space at a location pointed to by Xn.
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
STR Wt, [Xn]
DC CVAU, Xn         ; Clean data cache by VA to point of unification (PoU)
DSB ISH             ; Ensure visibility of the data cleaned from cache
IC IVAU, Xn         ; Invalidate instruction cache by VA to PoU
DSB ISH
```

Note

- The `DC CVAU` operation is not required if the area of memory is either Non-cacheable or Write-Through Cacheable.
- If the contents of physical memory differ between the mappings, changing the mapping of VAs to PAs can cause the instructions to be concurrently modified by one PE and executed by another PE. If the modifications affect instructions other than those listed as being acceptable for modification, synchronization must be used to avoid UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior.

3. In a multiprocessor system, the `IC IVAU` is broadcast to all PEs within the Inner Shareable domain of the PE running this sequence. However, when the modified instructions are observable, each PE that is executing the modified instructions must issue the following instruction to ensure execution of the modified instructions:

```
ISB                             ; Synchronize fetched instruction stream
```

For more information about the required synchronization operation, see *Synchronization and coherency issues between data and instruction accesses* on page B2-146.

For information about memory accesses caused by instruction fetches, see *Ordering relations* on page B2-126.

B2.2.6 Possible implementation restrictions on using atomic instructions

In some implementations, and for some memory types, the properties of atomicity can be met only by functionality outside the PE. Some system implementations might not support atomic instructions for all regions of the memory. In particular, this can apply to:

- Any type of memory in the system that does not support hardware cache coherency.
- Device, Non-cacheable memory, or memory that is treated as Non-cacheable, in an implementation that does support hardware cache coherency.

In such implementations, it is defined by the system:

- Whether the atomic instructions are atomic in regard to other agents that access memory.
- If the atomic instructions are atomic in regard to other agents that access memory, which address ranges or memory types this applies to.

An implementation can choose which memory type is treated as Non-cacheable.

The memory types for which it is architecturally guaranteed that the atomic instructions will be atomic are:

- Inner Shareable, Inner Write-Back, Outer Write-Back Normal memory with Read allocation hints and Write allocation hints and not transient.
- Outer Shareable, Inner Write-Back, Outer Write-Back Normal memory with Read allocation hints and Write allocation hints and not transient.

If the atomic instructions are not atomic in regard to other agents that access memory, then performing an atomic instruction to such a location can have one or more of the following effects:

- The instruction generates a synchronous External abort.
- The instruction generates a System Error interrupt.
- The instruction generates an IMPLEMENTATION DEFINED MMU fault reported using the Data Abort Fault status code of ESR_ELx.DFSC = `110101`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the EL1&0 translation regime, if the atomic instruction is not supported because of the memory type that is defined in the first stage of translation, or the second stage of translation is not enabled, then this exception is a first stage abort and is taken to EL1. Otherwise, the exception is a second stage abort and is taken to EL2.

- The instruction is treated as a `NOP`.
- The instructions are performed, but there is no guarantee that the memory accesses were performed atomically in regard to other agents that access memory. In this case, the instruction might also generate a System Error interrupt.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

B2.3 Definition of the Armv8 memory model

This section describes observation and ordering in the Armv8 memory model. It contains the following subsections:

- *Locations*.
- *Ordering and observability* on page B2-124.
- *Ordering constraints* on page B2-128.
- *Completion and endpoint ordering* on page B2-130.
- *Ordering of instruction fetches* on page B2-132.
- *Restrictions on the effects of speculation* on page B2-133.
- *Memory barriers* on page B2-135.
- *Limited ordering regions* on page B2-141.

For more information about endpoint ordering of memory accesses, see *Reordering* on page B2-160.

In the Armv8 memory model, the Shareability memory attribute indicates the degree to which hardware must ensure memory coherency between a set of observers, see *Memory types and attributes* on page B2-153.

The Armv8 architecture defines additional memory attributes and associated behaviors, which are defined in the system level section of this manual. See:

- Chapter D4 *The AArch64 System Level Memory Model*.
- Chapter D5 *The AArch64 Virtual Memory System Architecture*.

See also *Mismatched memory attributes* on page B2-164.

B2.3.1 Locations

The Armv8 memory model provides a set of definitions that are used to constrain the permitted sequences of accesses to memory. The Armv8 memory model defines:

- The ordering of observation of memory accesses between different observers.
- The ordering of arrival of memory accesses arriving at an endpoint.
- The mechanisms to control the ordering of observation of memory accesses and the arrival of memory accesses at an endpoint.

Locations, Memory effects, and Observers

The Armv8 memory model provides the following definition of a *Location* in memory:

Location

A *Location* refers to a single byte in memory.

As part of its execution an instruction might generate a Memory effect. Observers in the system might observe the Memory effects of that instruction on a Location. The Armv8 memory model provides the following definitions of a Memory effect and an Observer:

Memory effect

The *Memory effects* of an instruction are the read, write, or barrier effects of that instruction. For an instruction that accesses memory:

- A read effect is generated for each Location that is read by the instruction.
- A write effect is generated for each Location that is written by the instruction.

An instruction can generate both read and write effects.

The Memory effects of an instruction I_1 are said to appear in program order before the Memory effects of instruction I_2 if and only if I_1 occurs before I_2 in program order.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the purposes of describing the Armv8 memory model, all read and write effects access only Normal memory locations in a Common Shareability Domain, and excludes the situations described in *Mismatched memory attributes* on page B2-164. Where this section refers to a read, write, or memory barrier without any qualification, then it is referring to the corresponding Memory effect.

Observer

An *Observer* refers to either a processing element, or some other memory accessing agent that can generate reads from or writes to memory.

Common Shareability Domain

A *Common Shareability Domain* for a program is the smallest Shareability domain that contains all of the active Observers of the Memory effects generated by a program.

B2.3.2 Ordering and observability

The Armv8 memory model permits reordering of memory accesses. This section defines the constraints placed on the reordering of memory accesses using the following:

- Register value dependencies to establish order between instructions on a PE.
- Ordering constraints to establish order between accesses to a Location.

Register value dependencies

The Armv8 memory model defines the following dependencies between instructions:

Register dependency

A *Register dependency* from a first data value V_1 to a second data value V_2 exists within a PE if and only if either:

- The register, excluding the AArch64 zero register (XZR or WZR), that is used to hold V_1 is used in the calculation of V_2.
- There is a Register dependency from V_1 to a third data value V_3 and there is a register dependency from V_3 to V_2.

Register data dependency

A *Register data dependency* from a first data value V_1 to a second data value V_2 exists within a PE if and only if either:

- The register, excluding the AArch64 zero register (XZR or WZR) and the AArch32 PC, that is used to hold V_1 and is used in the calculation of V_2, and the calculation between V_1 and the V_2 does not consist of either:
  - A conditional branch whose condition is determined by V_1.
  - A conditional selection, move, or computation whose condition is determined by V_1, where the input data values for the selection, move, or computation do not have a data dependency on V_1.
- There is a Register data dependency from V_1 to a third data value V_3, and there is a Register data dependency from V_3 to V_2.

Address dependency

An *Address dependency* from a read R_1 to a subsequent read R_2 exists if and only if there is a Register data dependency from the data value that is returned by R_1 to the address used by R_2.

An *Address dependency* from a read R_1 to a subsequent write W_2 exists if and only if there is a Register dependency from the data value that is returned by R_1 to the address used by W_2.

Data dependency

A *Data dependency* from a read R_1 to a subsequent write W_2 exists if and only if there is a Register dependency from the data value returned by R_1 to the data value written by W_2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Control dependency

A *Control dependency* from a read R_1 to a subsequent instruction I_2 exists if and only if either:

- There is a Register dependency from the data value returned by R_1 to the data value used in the evaluation of a conditional branch, and I_2 is only executed as a result of one of the possible outcomes of that conditional branch.
- There is a Register dependency from the data value returned by R_1 to the data value used in the determination of a synchronous exception on an instruction I_3, and I_2 appears in program order after I_3.

Ordering and observability at a Location

Memory effects on a Location are related by the following relations:

Reads-from

A *Reads-from* relation that couples reads and writes to the same Location such that each read is paired with a single write in the program. A read R_2 of a Location Reads-from a write W_1 to the same Location if and only if R_2 takes its data from W_1.

Note

The Reads-from relation represents a read being satisfied by a write and then returning the written data.

Local read successor

A read R_2 of a Location is the *Local read successor* of a write W_1 from the same Observer to the same Location if and only if W_1 appears in program order before R_2 and there is not a write W_3 from the same Observer to the same Location appearing in program order between W_1 and R_2.

Local write successor

A write W_2 of a Location is a *Local write successor* of a write W_1 from the same Observer to the same Location if and only if W_1 appears in program order before W_2.

Coherence order

A *Coherence order* relation for each Location in the program that provides a total order on all writes from all coherent Observers to that Location, starting with a notional write of the initial value.

Note

The Coherence order of a Location represents the order in which writes to the Location arrive at memory.

Coherence-after

A write W_2 to a Location is *Coherence-after* another write W_1 to the same Location if and only if W_2 is sequenced after W_1 in the Coherence order of the Location.

A write W_2 to a Location is Coherence-after a read R_1 of the same location if and only if R_1 Reads-from a write W_3 to the same Location and W_2 is Coherence-after W_3.

Overlapping accesses

Two Memory effect overlap if and only if they access the same Location. Two instructions overlap if and only if one or more of their generated Memory effects overlap.

Observed-by

A read or a write RW_1 from an Observer is *Observed-by* a write W_2 from a different Observer if and only if W_2 is coherence-after RW_1.

A write W_1 from an Observer is Observed-by a read R_2 from a different Observer if and only if R_2 Reads-from W_1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

The Observed-by relation only relates accesses generated by different Observers.

`DMB FULL`

A `DMB FULL` is a `DMB` with neither the `LD` or the `ST` qualifier.

Where this section refers to `DMB` without any qualification, then it is referring to all types of `DMB`. Unless a specific shareability domain is defined, a `DMB` applies to the Common Shareability Domain.

All properties that apply to `DMB` also apply to the corresponding `DSB`.

Ordering relations

In addition to the ordering relations for a single Location, the Armv8 memory model also provides ordering relations to describe the ordering of Memory effects to multiple Locations. These are as follows:

Dependency-ordered-before

A dependency creates externally-visible order between a read and another Memory effect generated by the same Observer. A read R_1 is *Dependency-ordered-before* a read or write RW_2 from the same Observer if and only if R_1 appears in program order before RW_2 and any of the following cases apply:

- There is an Address dependency or a Data dependency from R_1 to RW_2.
- RW_2 is a write W_2 and there is a Control dependency from R_1 to W_2.
- RW_2 is a read R_2 generated by an instruction appearing in program order after an instruction I_3 that generates a Context synchronization event, and there is a Control dependency from R_1 to I_3.
- RW_2 is a write W_2 appearing in program order after a read or a write RW_3 and there is an Address dependency from R_1 to RW_3.
- RW_2 is a read R_2 that is a Local read successor of W_3 and there is an Address dependency or a Data dependency from R_1 to W_3.

Atomic-ordered-before

Load-Exclusive and Store-Exclusive instructions provide some ordering guarantees, even in the absence of dependencies. A read or a write RW_1 is *Atomic-ordered-before* a read or a write RW_2 from the same Observer if and only if RW_1 appears in program order before RW_2 and either of the following cases apply:

- RW_1 is a read R_1 and RW_2 is a write W_2 such that R_1 and W_2 are generated by an atomic instruction or a successful Load-Exclusive/Store-Exclusive instruction pair to the same Location.
- RW_1 is a write W_1 generated by an atomic instruction or a successful Store-Exclusive instruction and RW_2 is a read R_2 generated by an instruction with Acquire or AcquirePC semantics such that R_2 is a Local read successor of W_1.

For more information, see *Synchronization and semaphores* on page B2-167.

Barrier-ordered-before

Barrier instructions order prior Memory effects before subsequent Memory effects generated by the same Observer. A read or a write RW1 is *Barrier-ordered-before* a read or a write RW_2 from the same Observer if and only if RW1 appears in program order before RW_2 and any of the following cases apply:

- RW_1 appears in program order before a `DMB FULL` that appears in program order before RW_2.
- At least one of RW_1 and RW_2 is generated by an atomic instruction with both Acquire and Release semantics.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- RW_1 is a write W_1 generated by an instruction with Release semantics and RW_2 is a read R_2 generated by an instruction with Acquire semantics.
- RW_1 is a read R_1 and either:
  - — R_1 appears in program order before a `DMB LD` that appears in program order before RW_2.
  - — R_1 is generated by an instruction with Acquire or AcquirePC semantics.
- RW_2 is a write W_2 and either:
  - — RW_1 is a write W_1 appearing in program order before a `DMB ST` that appears in program order before W_2.
  - — W_2 is generated by an instruction with Release semantics.

Tag-determined-by

If ARMv8.5-MemTag is implemented, a Checked access read or write, RW_1, is *Tag-determined-by* a Tag read R_2 if and only if all of the following applies:

- RW_1 and R_2 are generated by the same instruction.
- R_2 reads the Allocation Tag at a tag physical address and compares it with the physical address Tag passed as part of the access RW_1.

If the result of the comparison can cause a precise exception and the result is negative then RW_1 does not architecturally occur.

Tag-ordered-before

If ARMv8.5-MemTag is implemented, a read, R_1, is *Tag-ordered-before* a read or a write RW_2 generated by the same instruction if and only if R_1 is a Tag read and RW_2 is a Checked data access Tag-determined-by R_1.

Locally-ordered-before

Dependencies, Local write successor, Load/Store-Exclusive, atomic and barrier instructions can be composed within an Observer to create externally-visible order. A read or write RW_1 is *Locally-ordered-before* a read or write RW_2 from the same Observer if and only if any of the following apply:

- RW_1 is a write W_1 and RW_2 is a write W_2 such that RW_2 is, or is generated by the same instruction as, a Local write successor of RW_1.
- RW_1 is Dependency-ordered-before RW_2.
- RW_1 is Atomic-ordered-before RW_2.
- RW_1 is Barrier-ordered-before RW_2.
- If ARMv8.5-MemTag is implemented, RW_1 is Tag-ordered-before RW_2.
- RW_1 is Locally-ordered-before a read or a write that is Locally-ordered-before RW_2.

Ordered-before

An arbitrary pair of Memory effects is ordered if it can be linked by a chain of ordered accesses consistent with external observation. A read or a write RW_1 is *Ordered-before* a read or a write RW_2 if and only if any of the following cases apply:

- RW_1 is Observed-by a read or write RW_3 that is equal to or generated by the same instruction as RW_2.
- RW_1 is Locally-ordered-before RW_2.
- RW_1 is Ordered-before a read or write that is Ordered-before RW_2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

B2.3.3 Ordering constraints

The Armv8 memory model is described as being Other-multi-copy atomic. The definition of Other-multi-copy atomic is as follows:

Other-multi-copy atomic

In an *Other-multi-copy atomic* system, it is required that a write from an Observer, if observed by a different Observer, is then observed by all other Observers that access the Location coherently. It is, however, permitted for an Observer to observe its own writes prior to making them visible to other observers in the system.

The Other-multi-copy atomic property of the Armv8 memory model is enforced by placing constraints on the possible executions of a program. Those executions that meet the constraints given by the ordering model are said to be *architecturally well-formed*. An implementation that is executing a program is only permitted to exhibit behavior consistent with an architecturally well-formed execution:

Architecturally well-formed

An *architecturally well-formed* execution must satisfy both the Internal visibility requirement and one of the following three alternative representations of the external ordering requirements:

- External visibility requirement.
- External completion requirement.
- External global completion requirement.

Internal visibility requirement

For a read or a write RW_1 that appears in program order before a read or a write RW_2 to the same Location, the internal visibility requirement requires that exactly one of the following statements is true:

- RW_2 is a write W_2 that is Coherence-after RW_1.
- RW_1 is a write W_1 and RW_2 is a read R_2 such that either:
  - R_2 Reads-from W_1.
  - R_2 Reads-from another write that is Coherence-after W_1.
- RW_1 and RW_2 are both reads R_1 and R_2 such that R_1 Reads-from a write W_3 and either:
  - R_2 Reads-from W_3.
  - R_2 Reads-from another write that is Coherence-after W_3.

Note

If a Memory effect M_1 from an Observer appears in program order before a Memory effect M_2 from the same Observer, then M_1 will be seen to occur before M_2 by that Observer.

External visibility requirement

For a read or a write RW_1 from an Observer that is Ordered-before a read or a write RW_2 from a different Observer, the external visibility constraint requires that RW_2 is not Observed-by RW_1. This means that an Architecturally well-formed execution must not exhibit a cycle in the Ordered-before relation.

Note

If a Memory effect M_1 from an Observer is Ordered-before another Memory effect M_2, from a different Observer, then M_1 will be seen to occur before M_2 by all Observers in the system.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Alternative equivalent formulations of external visibility requirement

Single-copy-atomic-ordered-before

A read R_1 is *Single-copy-atomic-ordered-before* another read R_2 if and only if all of the following statements are true:

- R_1 and R_2 are reads from the same instruction.
- R_1 does not Local read successor a write.
- R_2 Local read successor a write.

Total time order

The *Total time order* (TTO) is a total order that corresponds to the order in which memory effects complete within the system. The following effects constitute a single entry in the TTO:

- Writes from the same instruction.
- Reads from the same instruction which read from external writes.
- Writes from the same instruction which read from the same internal write.

All other reads constitute distinct entries in the TTO.

Completes-before

A read or a write RW_1 *Completes-before* a read or a write RW_2 if and only if RW_1 appears in the TTO before RW_2.

Deriving Reads-from the TTO

The TTO can be used to resolve the Reads-from relation for every memory access in the system as follows:

- For a read RW_1 of a memory location by an observer, then:
  - — If there is a write W_2 to the same Location from the same Observer and all of the following are true:
    - — W_2 appears in program order before R_1.
    - — R_1 Completes-before W_2.
    - — There are no writes to the Location appearing in program order between W_2 and R_1 then R_1 reads from W_2.
- Otherwise, if there is a write W_2 to the same Location and all of the following are true:
  - — W_2 Completes-before R_1.
  - — There are no writes to the Location appearing in program order or in the TTO between W_2 and R_1 then R_1 Reads-from W_2.
- If no such write exists, then R_1 Reads-from the initial value of the memory location.

Deriving Coherence order from the TTO

The TTO can be used to resolve the Coherence order relation for every memory access in the system as follows:

The Coherence order of writes to a memory location is the order in which those writes appear in the TTO. The final value of each memory location is therefore determined by the final write to each Location in the TTO. If no such write exists for a given location, the final value is the initial value of that Location.

External completion requirement

A read or a write RW_1 Globally-completes-before a read or a write RW_2 if and only if any of the following statements are true:

- RW_1 is Locally-ordered-before RW_2.
- RW_1 is a read R_1 and either:
  - — RW_2 is a read R_2 and R_1 is Single-copy-atomic-ordered-before R_2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- — R_1 is Single-copy-atomic-ordered-before a read R_3, such that:
  - — R_3 is Locally-ordered-before RW_2.
  - — R_3 Local read successor a write W4 that is not Locally-ordered-before RW_2.

Total coherence order

The total coherence order (TCO) is a total order that corresponds to the order in which memory effects globally-complete within the system. The following effects constitute a single entry in the TCO:

- Writes from the same instruction.
- Reads from the same instruction which read from external writes.
- Writes from the same instruction which read from the same internal write.

All other reads constitute distinct entries in the TCO.

Globally-completes-before

A read or a write RW_1 *globally-completes-before* a read or a write RW_2 if and only if RW_1 appears in the TCO before RW_2.

Deriving Reads-from and Coherence order from the TCO

The TCO can be used to resolve the Reads-from and Coherence order relations for every memory access in the system as follows:

- A read R_1 of a memory location by an Observer Reads-from its closest preceding write in the TCO to the same Location. If no such write exists, then R_1 Reads-from the initial value of the memory location.
- The Coherence order of writes to a memory location is the order in which those writes appear in the TCO. The final value of each memory location is therefore determined by the final write to each Location in the TCO. If no such write exists for a given Location, the final value is the initial value of that Location.

External global completion requirement

For a read or a write RW_1 that is Locally-ordered-before a read or a write RW_2, the external global completion requirement requires that RW_1 Globally-completes-before RW_2 if and only if any of the following statements are true:

- RW_1 is Locally-ordered-before RW_2 and either:
  - — RW_1 is a write.
  - — RW_1 is a read R_1 and either:
    - — R_1 does not Local read successor a write.
    - — R_1 Local read successor a write that is Locally-ordered-before RW_2.
- RW_1 is a read R_1 and either:
  - — RW_2 is a read R_2 and R_1 is Single-copy-atomic-ordered-before R_2.
  - — R_1 is Single-copy-atomic-ordered-before a read R_3, such that:
    - — R_3 is Locally-ordered-before RW_2.
    - — R_3 Local read successor a write W_4 that is not Locally-ordered-before RW_2.

B2.3.4 Completion and endpoint ordering

Interaction between Observers in a system is not restricted to communication via shared variables in coherent memory. For example, an Observer could configure an interrupt controller to raise an interrupt on another Observer as a form of message passing. These interactions typically involve an additional agent, which defines the instruction sequence that is required to establish communication links between different Observers. When these forms of interaction are used in conjunction with shared variables, a `DSB` instruction can be used to enforce ordering between them.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For all memory, the completion rules are defined as:

- A read R_1 to a Location is complete for a shareability domain when all of the following are true:
  - Any write to the same Location by an Observer within the shareability domain will be Coherence-after R_1.
  - Any translation table walks associated with R_1 are complete for that shareability domain.
- A write W_1 to a Location is complete for a shareability domain when all of the following are true:
  - Any write to the same Location by an Observer within the shareability domain will be Coherence-after W_1.
  - Any read to the same Location by an Observer within the shareability domain will either Reads-from W_1 or Reads-from a write that is Coherence-after W_1.
  - Any translation table walks associated with the write are complete for that shareability domain.
- A translation table walk is complete for a shareability domain when the memory accesses, including the updates to translation table entries, associated with the translation table walk are complete for that shareability domain, and the TLB is updated.
- A cache maintenance instruction is complete for a shareability domain when the memory effects of the instruction are complete for that shareability domain, and any translation table walks that arise from the instruction are complete for that shareability domain.
- A TLB invalidate instruction is complete when all memory accesses using the TLB entries that have been invalidated are complete.

The completion of any cache or TLB maintenance instruction includes its completion on all PEs that are affected by both the instruction and the `DSB` operation that is required to guarantee visibility of the maintenance instruction.

Note

These completion rules mean that, for example, a cache maintenance instruction that operates by VA to the PoC completes only after memory at the PoC has been updated.

Additionally, for Device-nGnRnE memory, a read or write of a Location in a Memory-mapped peripheral that exhibits side-effects is complete only when the read or write both:

- Can begin to affect the state of the Memory-mapped peripheral.
- Can trigger all associated side-effects, whether they affect other peripheral devices, PEs, or memory.

Note

This requirement for Device-nGnRnE memory is consistent with the memory access having reached the peripheral endpoint.

Peripherals

This section defines a Memory-mapped peripheral and the total order of reads and writes to a peripheral which is defined as the Peripheral coherence order:

Memory-mapped peripheral

A *Memory-mapped peripheral* occupies a memory region of IMPLEMENTATION DEFINED size and can be accessed using load and store instructions. Memory effects to a Memory-mapped peripheral can have side-effects, such as causing the peripheral to perform an action. Values that are read from addresses within a Memory-mapped peripheral might not correspond to the last data value written to those addresses. As such, Memory effects to a Memory-mapped peripheral might not appear in the Reads-from or Coherence order relations.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Peripheral coherence order

The *Peripheral coherence order* of a Memory-mapped peripheral is a total order on all reads and writes to that peripheral.

Note

The Peripheral coherence order for a Memory-mapped peripheral signifies the order in which accesses arrive at the endpoint.

For a read or a write RW_1 and a read or a write RW_2 to the same peripheral, then RW_1 will appear in the Peripheral coherence order for the peripheral before RW_2 if either of the following cases apply:

- RW_1 and RW_2 are accesses using Non-cacheable or Device attributes and RW_1 is Ordered-before RW_2.
- RW_1 and RW_2 are accesses using Device-nGnRE or Device-nGnRnE attributes and RW_1 appears in program order before RW_2.

Out-of-band-ordered-before

A read or a write RW_1 is *Out-of-band-ordered-before* a read or a write RW_2 if and only if either of the following cases apply:

- RW_1 appears in program order before a `DSB` instruction that begins an IMPLEMENTATION DEFINED instruction sequence indirectly leading to the generation of RW_2.
- RW_1 is Ordered-before a read or a write RW_3 and RW_3 is Out-of-band-ordered-before RW_2.

If a Memory effect M_1 is Out-of-band-ordered-before a read or a write M_2, then M_1 is seen to occur before M_2 by all Observers.

B2.3.5 Ordering of instruction fetches

For two memory locations A and B, if A has been written to and been made coherent with the instruction fetches of the shareability domain, before an update to B by an observer in the same shareability domain, then the instruction stream of each observer in the shareability domain will not see the updated value of B without also seeing the updated value of A.

A write has been made coherent with an instruction fetch of a shareability domain when:

CTR_EL0.{DIC, IDC} == {0, 0}

The location written to has been cleaned to the *Point of unification (PoU)* from the data cache, and that clean is complete for the shareability domain. Subsequently the location has been invalidated to the *Point of unification (PoU)* from the instruction cache, and that invalidation is complete for the shareability domain.

CTR_EL0.{DIC, IDC} == {1, 0}

The location written to has been cleaned to the *Point of unification (PoU)* from the data cache, and that clean is complete for the shareability domain.

CTR_EL0.{DIC, IDC} == {0, 1}

The write is complete for the shareability domain. Subsequently the location has been invalidated to the *Point of unification (PoU)* from the instruction cache, and that invalidation is complete for the shareability domain.

CTR_EL0.{DIC, IDC} == {1, 1}

The write is complete for the shareability domain.

Note

Microarchitecturally, this means that these situations cannot both be true in an implementation:

- After delays in fetching from memory, the instruction queue can have entries written into it out of order.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- For an implementation:
  - When CTR_EL0.DIC == 0, if there is an outstanding entry in the instruction queue, then later entries in the instruction queue are not impacted by the `IC IVAU` instructions of a different core.
  - When CTR_EL0.DIC == 1, if there is a write to the location that is held in the queue when there is an outstanding entry in the instruction queue for an older entry, then the instruction queue does not have entries invalidated from it.

B2.3.6 Restrictions on the effects of speculation

This section covers restrictions on speculation effects, including:

- *Restrictions on the effects of speculation*.
- *Speculative Store Bypass Safe (SSBS)*.
- *Restrictions on the effects of speculation from Armv8.5* on page B2-134.

Restrictions on the effects of speculation

The Arm architecture places certain restrictions on the effects of speculation. These are:

- Each load from a location using a particular VA after an exception return that is a Context synchronization event will not speculatively read an entry from earlier in the coherence order for the location being loaded from than the entry generated by the latest store to that location using the same VA before the exception exit.
- Each load from a location using a particular VA after an exception entry that is a Context synchronization event will not speculatively read an entry from earlier in the coherence order for the location being loaded from than the entry generated by the latest store to that location using the same VA before the exception entry.
- Any load from a location using a particular VA before an exception entry that is a Context synchronization event will not speculatively read data from a store to the same location using the same VA after the exception entry.
- Any load from a location using a particular VA before an exception return that is a Context synchronization event will not speculatively read data from a store to the same location using the same VA after the exception exit.
- When data is loaded under speculation with a translation fault, it cannot be used to form an address, generate condition codes, or generate SVE predicate values to be used by instructions newer than the load in the speculative sequence.
- When data is loaded under speculation from a location without a translation for the translation regime being speculated in, the data cannot be used to form an address, generate condition codes, or generate SVE predicate values to be used by instructions newer than the load in the speculative sequence.
- Changes to System registers must not occur speculatively in a way that can affect a speculative memory access that can cause a change to the micro-architectural state.
- Changes to Special-purpose registers can occur speculatively.

Speculative Store Bypass Safe (SSBS)

When ARMv8.0-SSBS is implemented, PSTATE.SSBS is a control that can be set by software to indicate whether hardware is permitted to load or store speculatively, in a manner that could be exploited to produce a cache timing side channel using an address derived from a register value that has been loaded from memory using a load instruction that speculatively read an entry for the location being loaded from, where the entry that is speculatively read is from earlier in the coherence order than the entry generated by the latest store to that location using the same virtual address as the load instruction.

When the value of PSTATE.SSBS is 0, hardware is not permitted to load or store speculatively in this way.

When the value of PSTATE.SSBS is 1, hardware is permitted to load or store speculatively in this way.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

- If speculation is permitted, then cache timing side channels can lead to addresses being derived using reads of address values that have been speculatively loaded from memory to a register.
- Software written for architectures from Armv8.0 to Armv8.4 will set SPSR_ELx.SSBS to 0. This means that PSTATE.SSBS will not set, so hardware will not be permitted to use speculative loads with outstanding memory disambiguation issues for any subsequent speculative memory accesses if there is any possibility of those subsequent memory accesses creating a cache timing side channel.

Restrictions on the effects of speculation from Armv8.5

From Armv8.5, there are some further restrictions on the effects of speculation in addition to those in Armv8.0:

- Data loaded under speculation with a permission or domain fault cannot be used to form an address, to generate condition codes, or to generate SVE predicate values to be used by instructions newer than the load in the speculative sequence.
- Any System register read under speculation to a register that is not architecturally accessible from the current Exception level cannot be used to form an address, to generate condition codes, or to generate SVE predicate values to be used by instructions newer than the load in the speculative sequence.

  Note

  As the effects of speculation are not architecturally visible, this restriction requires that the effect of any speculation cannot give rise to side channels that will leak the values of memory locations, System registers, or Special-purpose registers to a level of privilege that would otherwise not be able to determine those values.

- For all execution prediction resources that predict address or register values, speculative execution at one hardware defined context should be separated in a hard-to-determine manner from the predictions trained in a different hardware defined context. In the case of this definition, the hardware defined context is determined by:
  - The Exception level.
  - The Security state.
  - When executing at EL1 and when EL2 is enabled in the current Security state, the VMID.
  - When executing at EL0 and using the EL1&0 translation regime, the ASID and, when EL2 is enabled in the current Security state, the VMID.
  - When executing at EL0 and using the EL2&0 translation regime, the ASID.
  - When in AArch64 state, the current SCXTNUM_ELx value.

  Note

  - The definition of “hard-to-determine manner” is left open to implementations. Examples could include the complete separation of prediction resources, or the isolation of the predictions using a cryptographic or pseudo-random mechanism to separate each context.
  - The architecture does not require that prediction resources that simply predict the direction of a branch are separated in this way.

- Changes to System registers must not occur speculatively in a way that can affect a speculative memory access that can cause a change to the micro-architectural state.
- Changes to Special-purpose registers can occur speculatively.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

B2.3.7 Memory barriers

Memory barrier is the general term applied to an instruction, or sequence of instructions, that forces synchronization events by a PE with respect to retiring Load/Store instructions. The memory barriers defined by the Armv8 architecture provide a range of functionality, including:

- Ordering of Load/Store instructions.
- Completion of Load/Store instructions.
- Context synchronization.

The following subsections describe the Armv8 memory barrier instructions:

- *Instruction Synchronization Barrier (ISB)*
- *Data Memory Barrier (DMB)*.
- *Data Synchronization Barrier (DSB)* on page B2-138.
- *Speculation Barrier (SB)* on page B2-136.
- *Consumption of Speculative Data Barrier (CSDB)* on page B2-136.
- *Speculative Store Bypass Barrier (SSBB)* on page B2-137.
- *Profiling Synchronization Barrier (PSB CSYNC)* on page B2-137.
- *Physical Speculative Store Bypass Barrier (PSSBB)* on page B2-137.
- *Trace Synchronization Barrier (TSB CSYNC)* on page B2-138
- *Shareability and access limitations on the data barrier operations* on page B2-139.
- *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.
- *LoadLOAcquire, StoreLORelease* on page B2-141.

Note

Depending on the required synchronization, a program might use memory barriers on their own, or it might use them in conjunction with cache maintenance and memory management instructions that in general are only available when software execution is at EL1 or higher.

`DMB` and `DSB` instructions affect reads and writes to the memory system generated by Load/Store instructions and data or unified cache maintenance instructions being executed by the PE. Instruction fetches or accesses caused by a hardware translation table access are not explicit accesses.

Instruction Synchronization Barrier (ISB)

An `ISB` instruction ensures that all instructions that come after the `ISB` instruction in program order are fetched from the cache or memory after the `ISB` instruction has completed. Using an `ISB` ensures that the effects of context-changing operations executed before the `ISB` are visible to the instructions fetched after the `ISB` instruction. Examples of context-changing operations that require the insertion of an `ISB` instruction to ensure the effects of the operation are visible to instructions fetched after the `ISB` instruction are:

- Completed cache and TLB maintenance instructions.
- Changes to System registers.

Any context-changing operations appearing in program order after the `ISB` instruction only take effect after the `ISB` has been executed.

The pseudocode function for the operation of an `ISB` is `InstructionSynchronizationBarrier()`.

See also *Memory barriers* on page D4-2529.

Data Memory Barrier (DMB)

The `DMB` instruction is a memory barrier instruction that ensures the relative order of memory accesses before the barrier with memory accesses after the barrier. The `DMB` instruction does not ensure the completion of any of the memory accesses for which it ensures relative order.

The full definition of the `DMB` instruction is covered formally in the *Definition of the Armv8 memory model* on page B2-123 and this introduction to the `DMB` instruction is not intended to contradict that section.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The basic principle of a DMB instruction is to introduce order between memory accesses that are specified to be affected by the DMB options supplied as arguments to the DMB instruction. The DMB instruction ensures that all affected memory accesses by the PE executing the DMB instruction that appear in program order before the DMB instruction and those which originate from a different PE, to the extent required by the DMB options, which have been Observed-by the PE before the DMB instruction is executed, are Observed-by each PE, to the extent required by the DMB options, before any affected memory accesses that appear in program order after the DMB instruction are Observed-by that PE.

The use of a DMB instruction creates order between the Memory effects of instructions as described in the definition of Barrier-ordered-before.

The DMB instruction only affects memory accesses and the operation of data cache and unified cache maintenance instructions, see *A64 Cache maintenance instructions* on page D4-2510. It has no effect on the ordering of any other instructions executing on the PE. A DMB instruction intended to ensure the completion of cache maintenance instructions must have an access type of both loads and stores.

The pseudocode function for the operation of a DMB instruction is `DataMemoryBarrier()`.

Speculation Barrier (SB)

An SB instruction is a memory barrier that prevents speculative execution of instructions until after the barrier has completed when those instructions could be observed through side-channels.

Until the barrier completes, the speculative execution of any instruction appearing later in the program order than the barrier:

- Cannot be performed to the extent that such speculation can be observed through side-channels as a result of control flow speculation or data value speculation.
- Can be performed when predicting that a instruction that could generate an exception does not generate an exception.

Speculative execution of an SB instruction:

- Cannot be as a result of control flow speculation.
- Cannot be as a result of data value speculation.
- Can be as a result of predicting that an instruction that could generate an exception does not generate an exception.

An SB instruction can complete when:

- It is known that it is not speculative.
- All the predicted data values generated by instructions appearing in program order before the SB instruction have their predicted values confirmed.

Note

The SB instruction has no effect on the use of prediction resources to predict the instruction stream that is being fetched, so long as the prediction of the instruction stream is not informed by data taken from the register outputs of the speculative execution of instructions appearing in program order after the SB instruction.

Consumption of Speculative Data Barrier (CSDB)

The CSDB instruction is a memory barrier instruction that controls speculative execution and data value prediction. This includes:

- Data value predictions of any instructions.
- PSTATE.{N,Z,C,V} predictions of any instructions other than conditional branch instructions appearing in program order before the CSDB that have not been architecturally resolved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Predictions of SVE predication state for any SVE instructions.

For purposes of the definition of CSDB, `PSTATE.{N,Z,C,V}` is not considered a data value. This definition permits:

- Control flow speculation before and after the `CSDB` instruction.
- Speculative execution of conditional data processing instructions after the CSDB instruction, unless they use the results of data value or `PSTATE.{N,Z,C,V}` predictions of instructions appearing in program order before the `CSDB` instruction that have not been architecturally resolved.

Speculative Store Bypass Barrier (SSBB)

The `SSBB` instruction is a memory barrier that prevents speculative loads from bypassing earlier stores to the same virtual address under certain conditions.

The semantics of the Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the `SSBB` instruction, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store uses the same virtual address as the load.
  - The store appears in program order before the `SSBB` instruction.
- When a load to a location appears in program order before the `SSBB` instruction, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store uses the same virtual address as the load.
  - The store appears in program order after the `SSBB` instruction.

Profiling Synchronization Barrier (PSB CSYNC)

The `PSB CSYNC` instruction is a memory barrier that ensures that all existing profiling data for the current PE has been formatted, and profiling buffer addresses have been translated such that all writes to the profiling buffer have been initiated. A following `DSB` instruction completes when the writes to the profiling buffer have completed.

If the Statistical Profiling Extension is not implemented, this instruction executes as a `NOP`.

Physical Speculative Store Bypass Barrier (PSSBB)

The `PSSBB` instruction is a memory barrier that prevents speculative loads from bypassing earlier stores to the same physical address under certain conditions.

The semantics of the Physical Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the `PSSBB` instruction, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order before the `PSSBB` instruction.
- When a load to a location appears in program order before the `PSSBB` instruction, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order after the `PSSBB` instruction.

Note

The effect of this barrier applies to accesses to the same location even if they are accessed with different virtual addresses and from different Exception levels.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Trace Synchronization Barrier (TSB CSYNC)

The `TSB CSYNC` instruction is a memory barrier instruction that preserves the relative order of memory accesses to System registers due to trace operations and other memory accesses to the same registers.

A trace operation is an operation of the PE Trace Unit generating trace for an instruction when ARMv8.4-Trace is implemented and enabled.

A `TSB CSYNC` instruction is not required to execute in program order with respect to other instructions. This includes being reordered with respect to other trace instructions. One or more context synchronization events are required to ensure that `TSB CSYNC` instruction is executed in the necessary order.

If trace is generated between a context synchronization event and a `TSB CSYNC` operation, these trace operations may be reordered with respect to the `TSB CSYNC` operation, and therefore may not be synchronized.

The following situations are synchronized using a `TSB CSYNC` operation:

- A direct write B to a System register is ordered after an indirect read or indirect write of the same register by a trace operation of a traced instruction A, if all of the following are true:
  - A is executed in program order before a context synchronization event C.
  - C is in program order before a `TSB CSYNC` operation T.
  - B is executed in program order after T.
- A direct read B of a System register is ordered after an indirect write to the same register by a trace operation of a traced instruction A if all the following are true:
  - A is executed in program order before a context synchronization event C1.
  - C1 is in program order before `TSB CSYNC` operation T.
  - T is executed in program order before a second context synchronization event C2.
  - B is executed in program order after C2.

A `TSB CSYNC` operation is not needed to ensure a direct write B to a System register is ordered before an indirect read or indirect write of the same register by a trace operation of a traced instruction A, if all the following are true:

- A is executed in program order after a context synchronization event C.
- B is executed in program order before C.

The pseudocode function for the operation of a `TSB CSYNC` instruction is `TraceSynchronizationBarrier()`.

Data Synchronization Barrier (DSB)

A `DSB` instruction is a memory barrier that ensures that memory accesses that occur before the `DSB` instruction have completed before the completion of the `DSB` instruction. In doing this, it acts as a stronger barrier than a DMB and all ordering that is created by a DMB with specific options is also generated by a DSB with the same options.

Execution of a `DSB` instruction:

- At EL2 ensures that any memory accesses caused by Speculative translation table walks from the EL1&0 translation regime have been observed.
- At EL3 ensures that any memory accesses caused by speculative translation table walks from the EL2, EL1&0 or EL2&0 translation regimes have been observed.

For more information, see *Use of out-of-context translation regimes* on page D5-2554.

A `DSB` instruction executed by a PE, PEe, completes when all of the following apply:

- All explicit memory accesses of the required access types appearing in program order before the DSB are complete for the set of observers in the required shareability domain.
- If the required access types of the `DSB` is reads and writes, then all cache maintenance instructions, all TLB maintenance instructions, and all `PSB CYNC` instructions issued by PEe before the `DSB` are complete for the required shareability domain.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In addition, no instruction that appears in program order after the DSB instruction can alter any state of the system or perform any part of its functionality until the DSB completes other than:

- Being fetched from memory and decoded.
- Reading the general-purpose, SIMD and floating-point, Special-purpose, or System registers that are directly or indirectly read without causing side-effects.

If ARMv8.5-MemTag is implemented and enabled on completion of a DSB instruction, all updates to TFSR_ELx.TFx or TFSRE0_EL1.TFx due to Tag Check fails caused by accesses for which the DSB operates will be complete. For more information on ARMv8.5-MemTag see Chapter D6 *Armv8.5 Memory Tagging Extension*.

The pseudocode function for the operation of a DSB is DataSynchronizationBarrier().

See also *Memory barriers* on page D4-2529.

Shareability and access limitations on the data barrier operations

The DMB and DSB instructions take an argument that specifies:

- The shareability domain over which the instruction must operate. This is one of:
  - Full system.
  - Outer Shareable.
  - Inner Shareable.
  - Non-shareable.
- The accesses for which the instruction operates. This is one of:
  - Read and write accesses, both before and after the barrier instruction.
  - Write accesses only, before and after the barrier instruction.
  - Read accesses before the barrier instruction, and read and write accesses after the barrier instruction.

    Note

    This form of a DMB or DSB instruction can be described as a Load-Load/Store barrier.

For more information on whether an access is before or after a barrier instruction, see *Data Memory Barrier (DMB)* on page B2-135 or *Data Synchronization Barrier (DSB)* on page B2-138.

Table B2-1 shows how these options are encoded in the <option> field of the instruction:

Table B2-1 Encoding of the DMB and DSB <option> parameter

Accesses		Shareability domain			
Before the barrier	**After the barrier**	**Full system**	**Outer Shareable**	**Inner Shareable**	**Non-shareable**
Reads and writes	Reads and writes	SY	OSH	ISH	NSH
Writes	Writes	ST	OSHST	ISHST	NSHST
Reads	Reads and writes	LD	OSHLD	ISHLD	NSHLD

See the instruction descriptions for more information:

- *DMB* on page C6-911.
- *DSB* on page C6-914.

Note

ISB also supports an optional limitation argument that can only contain one value that corresponds to full system operation, see *ISB* on page C6-936.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Load-Acquire, Load-AcquirePC, and Store-Release

Armv8 provides a set of instructions with Acquire semantics for loads, and Release semantics for stores. These instructions support the *Release Consistency sequentially consistent* (RCsc) model. In addition, ARMv8.3-RCPC provides Load-AcquirePC instructions. The combination of Load-AcquirePC and Store-Release can be use to support the weaker *Release Consistency processor consistent* (RCpc) model.

The full definitions of the Load-Acquire and Load-AcquirePC instructions are covered formally in the *Definition of the Armv8 memory model* on page B2-123. This introduction to the Load-Acquire and Load-AcquirePC instructions is not intended to contradict that section.

The basic principle of both Load-Acquire and Load-AcquirePC instructions is to introduce order between:

- The memory access generated by the Load-Acquire or Load-AcquirePC instruction.
- The memory accesses appearing in program order after the Load-Acquire or Load-AcquirePC instruction, such that the memory access generated by the Load-Acquire or Load-AcquirePC instruction is Observed-by each PE to the extent that the PE is required to observe the access coherently, before any of the memory accesses appearing in program order after the Load-Acquire or Load-AcquirePC instruction are Observed-by that PE to the extent that the PE is required to observe the accesses coherently.

The use of a Load-Acquire or Load-AcquirePC instruction creates order between the Memory effects of instructions as described in the definition of Barrier-ordered-before.

The full definition of the Store-Release instruction is covered formally in the *Definition of the Armv8 memory model* on page B2-123 and this introduction to the Store-Release instruction is not intended to contradict that section.

The basic principle of a Store-Release instruction is to introduce order between the following:

- A set of memory accesses, RWx, that are generated by the PE executing the Store-Release instruction and that appear in program order before the Store-Release instruction, together with those that originate from a different PE to the extent that the PE is required to observe them coherently, Observed-by the PE before executing the Store-release.
- The memory access generated by the Store-Release (Wrel), such that all of the memory accesses, RWx, are Observed-by each PE to the extent that the PE is required to observe those accesses coherently, before Wrel is Observed-by that PE to the extent that the PE is required to observe that access coherently.

The use of a Store-Release instruction creates order between the Memory effects of instructions as described in the definition of Barrier-ordered-before.

Where a Load-Acquire appears in program order after a Store-Release, the memory access generated by the Store-Release instruction is Observed-by each PE to the extent that PE is required to observe the access coherently, before the memory access generated by the Load-Acquire instruction is Observed-by that PE, to the extent that the PE is required to observe the access coherently. In addition, the use of a Load-Acquire, Load-AcquirePC or a Store-Release instruction on accesses to a Memory-mapped peripheral introduces order between the Memory effects of the instructions that access that peripheral, as described in the definition of Peripheral coherence order.

Load-Acquire, Load-AcquirePC and Store-Release, other than Load-Acquire Exclusive Pair and Store-Release-Exclusive Pair, access only a single data element. This access is single-copy atomic. The address of the data object must be aligned to the size of the data element being accessed, otherwise the access generates an Alignment fault.

Load-Acquire Exclusive Pair and Store-Release Exclusive Pair access two data elements. The address supplied to the instructions must be aligned to twice the size of the element being loaded, otherwise the access generates an Alignment fault.

A Store-Release Exclusive instruction only has the release semantics if the store is successful.

Note

- Each Load-Acquire Exclusive and Store-Release Exclusive instruction is essentially a variant of the equivalent Load-Exclusive or Store-Exclusive instruction. All usage restrictions and single-copy atomicity properties:
  - That apply to the Load-Exclusive instructions also apply to the Load-Acquire Exclusive instructions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

  — That apply to the Store-Exclusive instructions also apply to the Store-Release Exclusive instructions.

- The Load-Acquire, Load-AcquirePC, and Store-Release instructions can remove the requirement to use the explicit `DMB` instruction.

LoadLOAcquire, StoreLORelease

For each PE, the Non-secure physical memory map is divided into a set of LORegions using a table that is held within the PE. Any PA in the Non-secure memory map can be a member of one LORegion. If a PA is assigned to more than one LORegion, then an implementation might treat it as if it has been assigned to fewer LORegions than that have been specified. A PA in the Secure physical memory map cannot be a member of any LORegion. For more information, see *Limited ordering regions*.

Armv8.1 provides a set of instructions with Acquire semantics for loads, and Release semantics for stores that apply in relation to the defined LORegions. The new variants of the Load-Acquire and Store-Release instructions are LoadLOAcquire and StoreLORelease. See *LoadLOAcquire/StoreLORelease* on page C3-216.

For all memory types, these instructions have the following ordering requirements:

- LoadLOAcquire has the same semantics as Load-Acquire except that the memory accesses affected lie within the same LORegion as the address of the memory access generated by the LoadLOAcquire instruction. See *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.
- StoreLORelease has the same semantics as Store-Release except that the memory accesses affected lie within the same LORegion as the address of the memory access generated by the StoreLORelease instruction. See *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

In addition, for accesses to Memory-mapped peripherals:

- LoadLOAcquire has the same semantics as Load-Acquire except that the affected Memory effects of instructions that access the peripheral lie within the same LORegion as the address of the memory access generated by the LoadLOAcquire instruction. See *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.
- StoreLORelease has the same semantics as Store-Release except that the affected Memory effects of instructions that access the peripheral lie within the same LORegion as the address of the memory access generated by the StoreLORelease instruction. See *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

Note

The LoadLOAcquire/StoreLORelease instructions can remove the requirement to use the explicit `DMB` instruction.

B2.3.8 Limited ordering regions

Armv8.1 introduces *limited ordering regions* (LORegions), which allow large systems to perform special load-acquire and store-release instructions that provide order between the memory accesses to a region of the PA map as observed by a set of observers.

This feature is supported in AArch64 state only.

Specification of the LORegions

The LORegions are defined in the Non-secure physical memory map using a set of LORegion descriptors. The number of LORegion descriptors is IMPLEMENTATION DEFINED, and can be discovered by reading the LORID_EL1 register.

Each LORegion descriptor consists of:

- A tuple of the following values:
  — A Start Address.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

  - An End Address.
  - An LORegion Number.
- Valid bit which indicates whether that LORegion descriptor is valid.

A memory location lies within the LORegion identified by the LORegion Number if the PA lies between the Start Address and the End Address, inclusive. The Start Address must be defined to be aligned to 64KB and the End Address must be defined as the top byte of a 64KB block of memory.

The LORegion descriptors are programmed using the LORSA_EL1, LOREA_EL1, LORN_EL1, and LORC_EL1 registers in the System register space. These registers are only supported in the Non-secure memory map.

If a LoadLOAcquire or a StoreLORelease does not match with any LORegion, then:

- The LoadLOAcquire will behave as a Load-Acquire, and will be ordered in the same way with respect to all accesses, independent of their LORegions.
- The StoreLORelease will behave as a Store-Release, and will be ordered in the same way with respect to all accesses, independent of their LORegions.

Note

If no LORegions are implemented, then the LoadLOAcquire and StoreLORelease will therefore behave as a Load-Acquire and Store-Release.

A new access type `AccType_LIMITEDORDERED` has been added for these limited ordering instructions to be identified.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

B2.4 Caches and memory hierarchy

The implementation of a memory system depends heavily on the microarchitecture and therefore many details of the memory system are IMPLEMENTATION DEFINED. Armv8 defines the application level interface to the memory system, including a hierarchical memory system with multiple levels of cache. This section describes an application level view of this system. It contains the subsections:

- *Introduction to caches*.
- *Memory hierarchy*.
- *Application level access to functionality related to caches* on page B2-144
- *Implication of caches for the application programmer* on page B2-145.
- *Preloading caches* on page B2-147.

B2.4.1 Introduction to caches

A cache is a block of high-speed memory that contains a number of entries, each consisting of:

- Main memory address information, commonly known as a *tag*.
- The associated data.

Caches increase the average speed of a memory access. Caching takes account of two principles of locality:

Spatial locality

An access to one Location is likely to be followed by accesses to adjacent Locations. Examples of this principle are:

- Sequential instruction execution.
- Accessing a data structure.

Temporal locality

An access to an area of memory is likely to be repeated in a short time period. An example of this principle is the execution of a software loop.

To minimize the quantity of control information stored, the spatial locality property groups several locations together under the same tag. This logical block is commonly known as a *cache line*. When data is loaded into a cache, access times for subsequent loads and stores are reduced, resulting in overall performance benefits. An access to information already in a cache is known as a *cache hit*, and other accesses are called *cache misses*.

Normally, caches are self-managing, with the updates occurring automatically. Whenever the PE accesses a cacheable memory location, the cache is checked. If the access is a cache hit, the access occurs in the cache. Otherwise, the access is made to memory. Typically, when making this access, a cache location is allocated and the cache line loaded from memory. Armv8 permits different cache topologies and access policies, provided they comply with the memory coherency model described in this manual.

Caches introduce a number of potential problems, mainly because:

- Memory accesses can occur at times other than when the programmer would expect them.
- A data item can be held in multiple physical locations.

B2.4.2 Memory hierarchy

Typically memory close to a PE has very low latency, but is limited in size and expensive to implement. Further from the PE it is common to implement larger blocks of memory but these have increased latency. To optimize overall performance, an Armv8 memory system can include multiple levels of cache in a hierarchical memory system that exploits this trade-off between size and latency. Figure B2-1 on page B2-144 shows an example of such a system in an Armv8-A system that supports virtual addressing.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

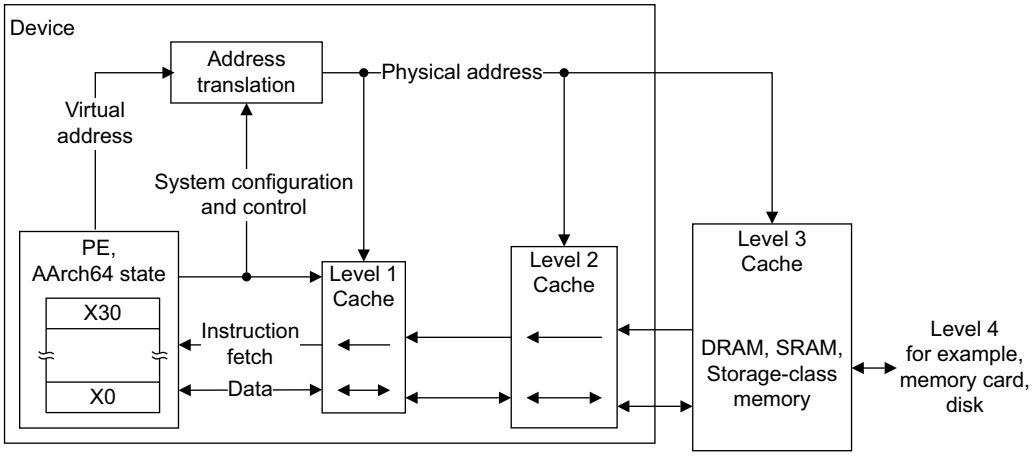


Figure B2-1 Multiple levels of cache in a memory hierarchy

Note

In this manual, in a hierarchical memory system, Level 1 refers to the level closest to the processing element, as shown in Figure B2-1.

Instructions and data can be held in separate caches or in a unified cache. A cache hierarchy can have one or more levels of separate instruction and data caches, with one or more unified caches that are located at the levels closest to the main memory. Memory coherency for cache topologies can be defined using the conceptual points Point of Unification (PoU), Point of Coherency (PoC), Point of Persistence (PoP), and Point of Deep Persistence (PoDP).

For more information, including the definitions of PoU, PoC, PoP, and PoDP, see *About cache maintenance in AArch64 state* on page D4-2506.

If ARMv8.5-MemTag is implemented and enabled the behavior of cache maintenance instructions is modified for more information see *Allocation Tags* on page D6-2691.

The cacheability and shareability memory attributes

Cacheability and shareability are two attributes that describe the memory hierarchy in a multiprocessing system:

Cacheability This attribute defines whether memory locations are allowed to be allocated into a cache or not. Cacheability is defined independently for Inner and Outer Cacheability locations.

Shareability This attribute defines whether memory locations are shareable between different agents in a system. Marking a memory location as shareable for a particular domain requires hardware to ensure that the location is coherent for all agents in that domain. Shareability is defined independently for Inner and Outer Shareability domains.

For more information about Cacheability and Shareability, see *Memory types and attributes* on page B2-153.

B2.4.3 Application level access to functionality related to caches

As indicated in *About the Application level programmers' model* on page B1-106, the application level corresponds to execution at EL0. The architecture defines a set of cache maintenance instructions that software can use to manage cache coherency. Software executing at a higher Exception level can enable use of some of this functionality from EL0, as follows:

When the value of SCTLR_EL1.UCI is 1

Software executing at EL0 can access:

- The data cache maintenance instructions, `DC CVAU`, `DC CVAC`, `DC CVAP`, `DC CVADP`, and `DC CIVAC`. See *The data cache maintenance instruction (DC)* on page D4-2511.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The instruction cache maintenance instruction `IC IVAU`. See *The instruction cache maintenance instruction (IC)* on page D4-2511.

Attempted execution of these instructions might generate a Permission fault as described in *Permission fault* on page D5-2653.

When the value of SCTLR_EL1.UCT is 1

Software executing at EL0 can access the cache type register. See CTR_EL0.

When the value of SCTLR_EL1.DZE is 1

Software executing at EL0 can access the data cache zero instruction `DC ZVA`. See *Data cache zero instruction* on page D4-2520.

The SCTLR_EL1.{UCI, UCT, DZE} control fields are only accessible by software executing at EL1 or higher.

When HCR_EL2.{E2H, TGE} == 1 the controls {UCI, UCT and DZE} are found in SCTLR_EL2.

This functionality is UNDEFINED at EL0 when the value of the corresponding SCTLR_EL1 control field is 0, see:

- *Traps to EL1 of EL0 execution of cache maintenance instructions* on page D1-2377.
- *Traps to EL1 of EL0 accesses to the CTR_EL0* on page D1-2377.
- *Traps to EL1 of EL0 execution of DC ZVA instructions* on page D1-2377.

B2.4.4 Implication of caches for the application programmer

In normal operation, the caches are largely invisible to the application programmer. However they can become visible when there is a breakdown in the coherency of the caches. Such a breakdown can occur:

- When memory locations are updated by other agents in the system that do not use hardware management of coherency.
- When memory updates made from the application software must be made visible to other agents in the system, without the use of hardware management of coherency.

For example:

- In the absence of hardware management of coherency of DMA accesses, in a system with a DMA controller that reads memory locations that are held in the data cache of a PE, a breakdown of coherency occurs when the PE has written new data in the data cache, but the DMA controller reads the old data held in memory.
- In a Harvard cache implementation, where there are separate instruction and data caches, a breakdown of coherency occurs when new instruction data has been written into the data cache, but the instruction cache still contains the old instruction data.

Data coherency issues

Software can ensure the data coherency of caches in the following ways:

- By not using the caches in situations where coherency issues can arise. This can be achieved by:
  - Using Non-cacheable or, in some cases, Write-Through Cacheable memory.
  - Not enabling caches in the system.
- By using cache maintenance instructions to manage the coherency issues in software. See *Application level access to functionality related to caches* on page B2-144.
- By using hardware coherency mechanisms to ensure the coherency of data accesses to memory for cacheable locations by observers within the different shareability domains, see *Non-shareable Normal memory* on page B2-155 and *Shareable, Inner Shareable, and Outer Shareable Normal memory* on page B2-154.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

The performance of these hardware coherency mechanisms is highly implementation-specific. In some implementations, the mechanism suppresses the ability to cache shareable locations. In other implementations, cache coherency hardware can hold data in caches while managing coherency between observers within the shareability domains.

Note

Not all these mechanisms are directly available to software operating at EL0 and might involve interaction with software operating at a higher Exception level.

Synchronization and coherency issues between data and instruction accesses

How far ahead of the current point of execution instructions are fetched from is IMPLEMENTATION DEFINED. Such prefetching can be either a fixed or a dynamically varying number of instructions, and can follow any or all possible future execution paths. For all types of memory:

- The PE might have fetched the instructions from memory at any time since the last *Context synchronization event* on that PE.
- Any instructions fetched in this way might be executed multiple times, if this is required by the execution of the program, without being refetched from memory. In the absence of a *Context synchronization event*, there is no limit on the number of times such an instruction might be executed without being refetched from memory.

The Arm architecture requires the hardware to ensure coherency between instruction caches and memory, even for locations of shared memory. A write has been made coherent with an instruction fetch of a shareability domain when:

CTR_EL0.{DIC, IDC} == {0, 0}

The location written to has been cleaned to the *Point of unification (PoU)* from the data cache, and that clean is complete for the shareability domain. Subsequently the location has been invalidated to the *Point of unification (PoU)* from the instruction cache, and that invalidation is complete for the shareability domain.

CTR_EL0.{DIC, IDC} == {1, 0}

The location written to has been cleaned to the *Point of unification (PoU)* from the data cache, and that clean is complete for the shareability domain.

CTR_EL0.{DIC, IDC} == {0, 1}

The write is complete for the shareability domain. Subsequently the location has been invalidated to the *Point of unification (PoU)* from the instruction cache, and that invalidation is complete for the shareability domain.

CTR_EL0.{DIC, IDC} == {1, 1}

The write is complete for the shareability domain.

If software requires coherency between instruction execution and memory, it must manage this coherency using *Context synchronization event*s and cache maintenance instructions. The following code sequence can be used to allow a PE to execute code that the same PE has written.

```
; Coherency example for data and instruction accesses within the same Inner Shareable domain.
; Enter this code with <Wt> containing a new 32-bit instruction,
; to be held in Cacheable space at a location pointed to by Xn.
        STR Wt, [Xn]
        DC CVAU, Xn         ; Clean data cache by VA to point of unification (PoU)
        DSB ISH             ; Ensure visibility of the data cleaned from cache
        IC IVAU, Xn         ; Invalidate instruction cache by VA to PoU
        DSB ISH             ; Ensure completion of the invalidations
        ISB                 ; Synchronize the fetched instruction stream
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

- If this sequence is not executed between writing data to a location and executing the instruction at that location, the lack of coherency between instruction caches and memory means that the instructions that are executed might be the old instruction or the updated instruction, and which is used can arbitrarily vary during execution. It must not be assumed by software, before the synchronization sequence is executed, that when the updated instruction has been seen, the old instruction will not be seen again.
- For Non-cacheable or Write-Through accesses, the clean data cache by VA instruction is not required. However, the invalidate instruction cache instruction is required because the Armv8-A AArch64 architecture allows Non-cacheable accesses to be held in an instruction cache. See *Non-cacheable accesses and instruction caches* on page D4-2505.
- This code can be used when the thread of execution modifying the code is the same thread of execution that is executing the code. The Armv8 architecture limits the set of instructions that can be executed by one thread of execution as they are being modified by another thread of execution without requiring explicit synchronization. See *Concurrent modification and execution of instructions* on page B2-120.
- The system software controls whether these cache maintenance instructions are available to the application level by setting SCTLR_EL1.UCI.

B2.4.5 Preloading caches

The Arm architecture provides memory system hints `PRFM`, `LDNP`, and `STNP` that software can use to communicate the expected use of memory locations to the hardware. The memory system can respond by taking actions that are expected to speed up the memory accesses if they occur. The effect of these memory system hints is IMPLEMENTATION DEFINED. Typically, implementations use this information to bring the data or instruction locations into caches.

The Preload instructions are hints, and so implementations can treat them as `NOP`s without affecting the functional behavior of the device. The instructions cannot generate synchronous Data Abort exceptions, but the resulting memory system operations might, under exceptional circumstances, generate an asynchronous External abort, which is taken using an SError interrupt exception. For more information, see *ISS encoding for an exception from a Data Abort* on page D13-2992.

`PrefetchHint{}` defines the prefetch hint types.

The `Hint_Prefetch()` function signals to the memory system that memory accesses of the type `hint` to or from the specified `address` are likely to occur in the near future. The memory system might take some action to speed up the memory accesses when they do occur, such as preloading the specified address into one or more caches as indicated by the innermost cache level `target` and non-temporal hint `stream`.

For more information on `PRFM` and Load/Store instructions that provide hints to the memory system, see *Prefetch memory* on page C3-220 and *Load/Store SIMD and Floating-point Non-temporal pair* on page C3-218.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

B2.5 Alignment support

This section describes alignment support. It contains the following subsections:

- *Instruction alignment*.
- *Alignment of data accesses*.

B2.5.1 Instruction alignment

A64 instructions must be word-aligned.

Attempting to fetch an instruction from a misaligned location results in a PC alignment fault. See *PC alignment checking* on page D1-2333.

B2.5.2 Alignment of data accesses

An unaligned access to any type of Device memory causes an Alignment fault.

Unaligned accesses to Normal memory

The behavior of unaligned accesses to Normal memory is dependent on all of the following:

- The instruction causing the memory access.
- The memory attributes of the accessed memory.
- The value of SCTLR_ELx{A, nAA}.
- Whether or not ARMv8.4-LSE is implemented.

Load or Store of Single or Multiple registers

For all instructions that load or store single or multiple registers, but not Load-Exclusive, Store-Exclusive, Load-Acquire/Store-Release and Atomic instructions, if the address that is accessed is not aligned to the size of the data element being accessed, then:

When the value of SCTLR_ELx.A applicable to the current Exception level is 1, an Alignment fault is generated.

When the value of SCTLR_ELx.A applicable to the current Exception level is 0:

- An unaligned access is performed.
- If ARMv8.4-LSE is not implemented, the access is not guaranteed to be single-copy atomic except at the byte access level.
- If ARMv8.4-LSE is implemented:
  - — If all the bytes of the memory access lie within a 16-byte quantity aligned to 16 bytes and are to Normal Inner Write-Back, Outer Write-Back Cacheable memory, the memory access is single-copy atomic. For a Load-Pair or Store-Pair, including load non-temporal pair, instructions the entire memory access will be single-copy atomic.
  - — If all the bytes of the memory accessed do not lie within a 16-byte quantity aligned to 16 bytes or the access is not to Normal Inner Write-Back, Outer Write-Back Cacheable memory the access is not guaranteed to be single-copy atomic except at the byte access level.

For these instructions, the definition of an unaligned access is based on the size of the accessed elements, not the overall size of the memory access. This affects SIMD element and structure loads and stores, and also Load/store pair instructions.

Load-Exclusive/ Store-Exclusive and Atomic instructions

For Load-Exclusive/Store-Exclusive, and Atomic instructions including those with acquire or acquire-release semantics:

When the value of SCTLR_ELx.A applicable to the current Exception level is 1, an Alignment fault is generated.

When the value of SCTLR_ELx.A applicable to the current Exception level is 0:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If ARMv8.4-LSE is not implemented, these instructions generate an Alignment fault if the address being accessed is not aligned to the size of the data structure being accessed.

If ARMv8.4-LSE is implemented, then:

- If all the bytes of the memory access lie within a 16-byte quantity aligned to 16 bytes and are to Normal Inner Write-Back, Outer Write-Back Cacheable memory, an unaligned access is performed.
- If all the bytes of the memory access do not lie within a 16-byte quantity aligned to 16-bytes, or the memory access is not to Normal Inner Write-Back, Outer Write-Back Cacheable memory, then it is an IMPLEMENTATION DEFINED choice of either of the following:
  - An unaligned access is performed meeting all of the semantics of the instruction.
  - An Alignment fault is generated.

Where memory access is performed, then it is single-copy atomic.

For these instructions, the definition of an unaligned access is based on the overall access size.

Non-atomic Load-Acquire/Store-Release instructions

For Load-Acquire/Store-Release instructions which do not have exclusive or atomic behaviors:

When the value of SCTLR_ELx.A applicable to the current Exception level is 1, an Alignment fault is generated.

When the value of SCTLR_ELx.A applicable to the current Exception level is 0:

If ARMv8.4-LSE is not implemented, then these instructions generate an Alignment fault if the address being accessed is not aligned to the size of the data structure being accessed.

If ARMv8.4-LSE is implemented, then:

- If the memory access is not to Normal Inner Write-Back or Outer Write-Back Cacheable memory, then it is an IMPLEMENTATION DEFINED choice of either of the following:
  - An unaligned access is performed meeting all of the semantics of the instruction.
  - An Alignment fault is generated.
- If all of the bytes of the memory access do not lie within a 16-byte quantity aligned to 16 bytes then the following applies:
  - If SCTLR_ELx.nAA applicable to the current Exception level is 0 an Alignment fault is generated.
  - If SCTLR_ELx.nAA applicable to the current Exception level is 1 then an unaligned access is performed which is not guaranteed to be single-copy atomic except at the byte access level.

  In this case, the architecture does no define the order of the different transactions of the access defined by the single instructions relative to each other.

Note

- Unaligned accesses typically take additional cycles to complete compared to a naturally-aligned access.
- An operation that is not single-copy atomic above the byte level can abort on any memory access that it makes and can abort on more than one access. This means that an unaligned access that occurs across a page boundary can generate an abort on either side of the page boundary.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

B2.6 Endian support

General description of endianness in the Arm architecture describes the relationship between endianness and memory addressing in the Arm architecture.

The following subsections then describe the endianness schemes supported by the architecture:

- *Instruction endianness* on page B2-151.
- *Data endianness* on page B2-151.
- *Endianness of memory-mapped peripherals* on page B2-152.

B2.6.1 General description of endianness in the Arm architecture

This section only describes memory addressing and the effects of endianness for data elements up to quadwords of 128 bits. However, this description can be extended to apply to larger data elements.

For an address A, Figure B2-2 shows, for big-endian and little-endian memory systems, the relationship between:

- The quadword at address A.
- The doubleword at address A and A+8.
- The words at addresses A, A+4, A+8, and A+12.
- The halfwords at addresses A, A+2, A+4, A+6, A+8, A+10, A+12, and A+14.
- The bytes at addresses A, A+1, A+2, A+3, A+4, A+5, A+6, A+7, A+8, A+9, A+10, A+11, A+12, A+13, A+14, and A+15.

The terms in Figure B2-2 have the following definitions:

B_A Byte at address A.

HW_A Halfword at address A.

MSByte Most significant byte.

LSByte Least significant byte.

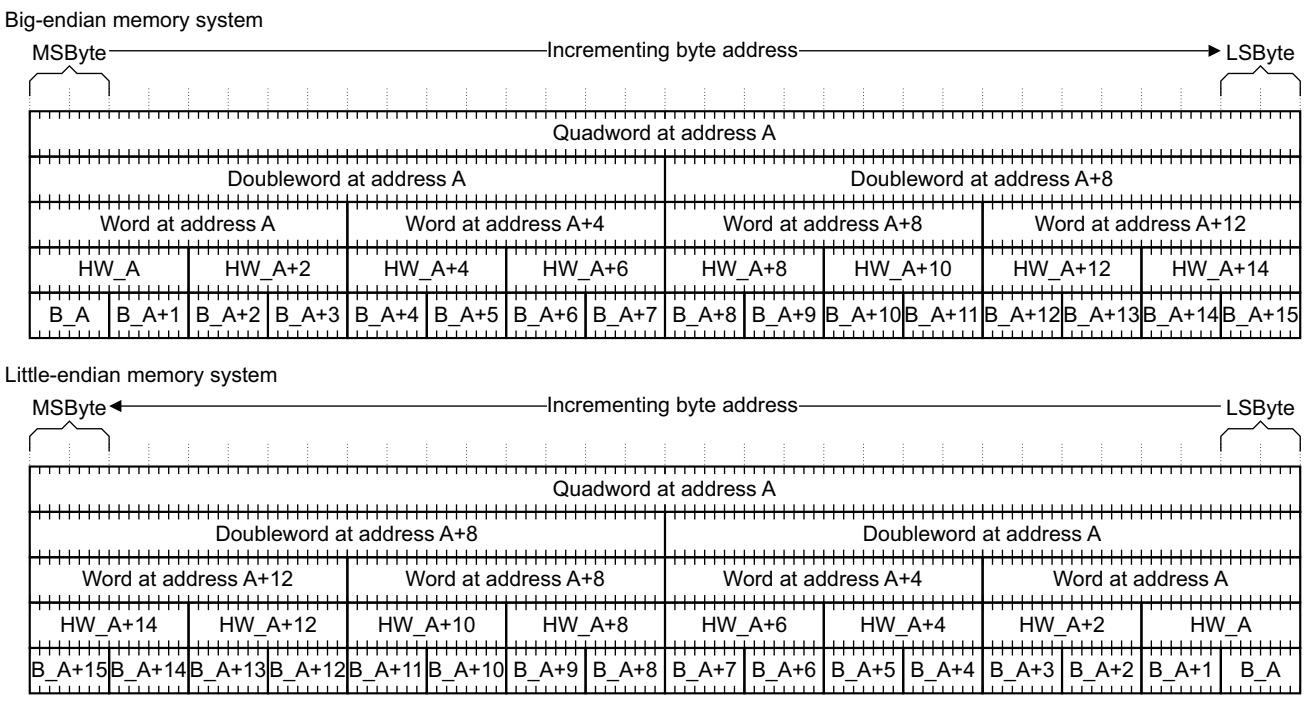


Figure B2-2 Endianness relationships

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The big-endian and little-endian mapping schemes determine the order in which the bytes of a quadword, doubleword, word, or halfword are interpreted. For example, a load of a word from address `0x1000` always results in an access to the bytes at memory locations `0x1000`, `0x1001`, `0x1002`, and `0x1003`. The endianness mapping scheme determines the significance of these 4 bytes.

B2.6.2 Instruction endianness

In Armv8-A, A64 instructions have a fixed length of 32 bits and are always little-endian.

B2.6.3 Data endianness

SCTLR_EL1.E0E, configurable at EL1 or higher, determines the data endianness for execution at EL0.

The data size used for endianness conversions:

- Is the size of the data value that is loaded or stored for SIMD and floating-point register and general-purpose register loads and stores.
- Is the size of the data element that is loaded or stored for SIMD element and data structure loads and stores. For more information, see *Endianness in SIMD operations*.

Note

This means the Armv8 architecture introduces a requirement for 128-bit endian conversions.

Instructions to reverse bytes in a general-purpose register or a SIMD and floating-point register

An application or device driver might have to interface to memory-mapped peripheral registers or shared memory structures that are not the same endianness as the internal data structures. Similarly, the endianness of the operating system might not match that of the peripheral registers or shared memory. In these cases, the PE requires an efficient method to transform explicitly the endianness of the data.

Table B2-2 shows the instructions that provide this functionality:

Table B2-2 Byte reversal instructions

Function	Instructions	Notes
Reverse bytes in 32-bit word or words[a]	REV32	For use with general-purpose registers
Reverse bytes in whole register	REV	For use with general-purpose registers
Reverse bytes in 16-bit halfwords	REV16	For use with general-purpose registers
Reverse elements in doublewords, vector	REV64	For use with SIMD and floating-point registers
Reverse elements in words, vector	REV32	For use with SIMD and floating-point registers
Reverse elements in halfwords, vector	REV16	For use with SIMD and floating-point registers

a. Can operate on multiple words.

Endianness in SIMD operations

SIMD element Load/Store instructions transfer vectors of elements between memory and the SIMD and floating-point register file. An instruction specifies both the length of the transfer and the size of the data elements being transferred. This information is used to load and store data correctly in both big-endian and little-endian systems.

For example:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
LD1 {V0.4H}, [X1]
```

This loads a 64-bit register with four 16-bit values. The four elements appear in the register in array order, with the lowest indexed element fetched from the lowest address. The order of bytes in the elements depends on the endianness configuration, as shown in Figure B2-3. Therefore, the order of the elements in the registers is the same regardless of the endianness configuration.

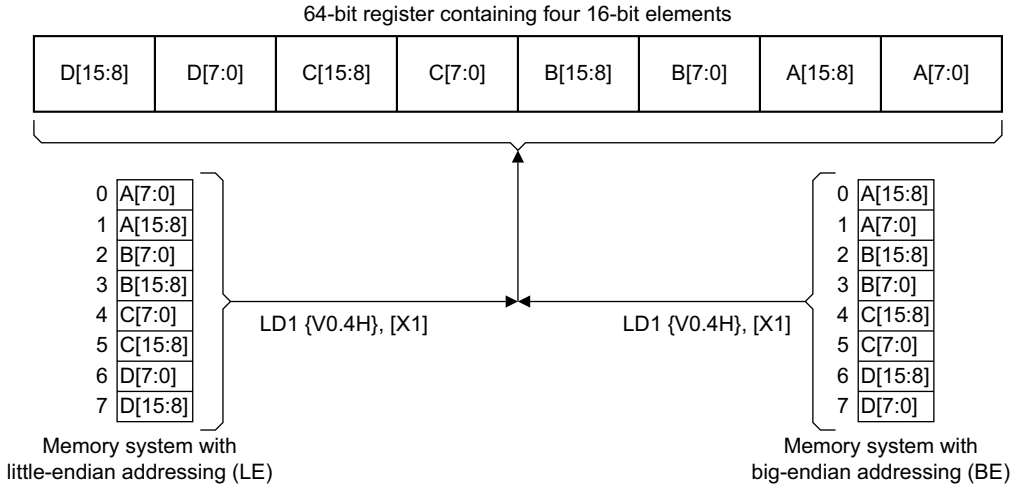


Figure B2-3 SIMD byte order example

The `BigEndian()` pseudocode function determines the current endianness of the data.

The `BigEndianReverse()` pseudocode function reverses the endianness of a bitstring.

The `BigEndian()` and `BigEndianReverse()` functions are defined in Chapter J1 *Armv8 Pseudocode*.

B2.6.4 Endianness of memory-mapped peripherals

All memory-mapped peripherals defined in the Arm architecture must be little-endian.

Peripherals to which this requirement applies include:

- Memory-mapped register interfaces to a debugger, or to a cross-trigger interface, see Chapter H8 *About the External Debug Registers*.
- The memory-mapped register interface to the system level implementation of the Generic Timer, see Chapter I2 *System Level Implementation of the Generic Timer*.
- A memory-mapped register interface to the Performance Monitors, see Chapter I3 *Recommended External Interface to the Performance Monitors*.
- A memory-mapped register interface to the Activity Monitors, see Chapter I4 *Recommended External Interface to the Activity Monitors*.
- Memory-mapped register interfaces to an Arm Generic Interface Controller, see the *ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0 and version 4.0.*
- The memory-mapped register interface to an Arm trace component. See, for example, the *ARM® Embedded Trace Macrocell Architecture Specification, ETMv4.*

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

B2.7 Memory types and attributes

In Armv8 the ordering of accesses for addresses in memory, referred to as the memory order model, is defined by the memory attributes. The following sections describe this model:

- *Normal memory*.
- *Device memory* on page B2-157.
- *Memory access restrictions* on page B2-162.

B2.7.1 Normal memory

The Normal memory type attribute applies to most memory in a system. It indicates that the hardware is permitted by the architecture to perform *Speculative* data read accesses to these locations, regardless of the access permissions for these locations.

The Normal memory type has the following properties:

- A write to a memory location with the Normal attribute completes in finite time.
- Writes to a memory location with the Normal memory type that is either Non-cacheable or Write-through cacheable for both the Inner and Outer cacheability must reach the endpoint for that location in the memory system in finite time. Two writes to the same location, where at least one is using the Normal memory type, might be merged before they reach the endpoint unless there is an ordered-before relationship between the two writes.
- Unaligned memory accesses can access Normal memory if the system is configured to generate such accesses.
- There is no requirement for the memory system beyond the PE to be able to identify the elements accessed by multi-register Load/Store instructions. See *Multi-register loads and stores that access Normal memory* on page B2-157.

Note

- The Normal memory attribute is appropriate for locations of memory that are idempotent, meaning that they exhibit all of the following properties:
  - Read accesses can be repeated with no side-effects.
  - Repeated read accesses return the last value written to the resource being read.
  - Read accesses can fetch additional memory locations with no side-effects.
  - Write accesses can be repeated with no side-effects if the contents of the location accessed are unchanged between the repeated writes or as the result of an exception, as described in this section.
  - Unaligned accesses can be supported.
  - Accesses can be merged before accessing the target memory system.
- Normal memory allows speculative reads and may be affected by intermediate buffering and forwarding of data. If non-idempotent memory locations are mapped as Normal memory, the following may occur:
  - Memory accesses return UNKNOWN values.
  - UNPREDICTABLE effects on memory-mapped peripherals.
- An instruction that generates a sequence of accesses as described in *Atomicity in the Arm architecture* on page B2-118 might be abandoned as a result of an exception being taken during the sequence of accesses. On return from the exception the instruction is restarted, and therefore, one or more of the memory locations might be accessed multiple times. This can result in repeated write accesses to a location that has been changed between the write accesses.

For accesses to Normal memory, a `DMB` instruction is required to ensure the required ordering.

The following sections describe the other attributes for Normal memory:

- *Shareable Normal memory* on page B2-154.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *Non-shareable Normal memory* on page B2-155.
- *Cacheability attributes for Normal memory* on page B2-155.

See also:

- *Multi-register loads and stores that access Normal memory* on page B2-157.
- *Atomicity in the Arm architecture* on page B2-118.
- *Memory barriers* on page B2-135.
- *Concurrent modification and execution of instructions* on page B2-120.

Shareable Normal memory

A Normal memory location has a Shareability attribute that is one of:

- Inner Shareable, meaning it applies across the Inner Shareable shareability domain.
- Outer Shareable, meaning it applies across both the Inner Shareable and the Outer Shareable shareability domains.
- Non-shareable.

The shareability attributes define the data coherency requirements of the location, that hardware must enforce. They do not affect the coherency requirements of instruction fetches, see *Synchronization and coherency issues between data and instruction accesses* on page B2-146.

Note

- System designers can use the shareability attribute to specify the locations in Normal memory for which coherency must be maintained. However, software developers must not assume that specifying a memory location as Non-shareable permits software to make assumptions about the incoherency of the location between different PEs in a shared memory system. Such assumptions are not portable between different multiprocessing implementations that might use the shareability attribute. Any multiprocessing implementation might implement caches that are shared, inherently, between different processing elements.
- This architecture assumes that all PEs that use the same operating system or hypervisor are in the same Inner Shareable shareability domain.

Shareable, Inner Shareable, and Outer Shareable Normal memory

The Arm architecture abstracts the system as a series of Inner and Outer Shareability domains.

Each Inner Shareability domain contains a set of observers that are data coherent for each member of that set for data accesses with the Inner Shareable attribute made by any member of that set.

Each Outer Shareability domain contains a set of observers that are data coherent for each member of that set for data accesses with the Outer Shareable attribute made by any member of that set.

The following properties also hold:

- Each observer is only a member of a single Inner Shareability domain.
- Each observer is only a member of a single Outer Shareability domain.
- All observers in an Inner Shareability domain are always members of the same Outer Shareability domain. This means that an Inner Shareability domain is a subset of an Outer Shareability domain, although it is not required to be a proper subset.

Note

- Because all data accesses to Non-cacheable locations are data coherent to all observers, Non-cacheable locations are always treated as Outer Shareable.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The Inner Shareable domain is expected to be the set of PEs controlled by a single hypervisor or operating system.

The details of the use of the shareability attributes are system-specific. Example B2-1 shows how they might be used.

Example B2-1 Use of shareability attributes

In an implementation, a particular subsystem with two clusters of PEs has the requirement that:

- In each cluster, the data caches or unified caches of the PEs in the cluster are transparent for all data accesses to memory locations with the Inner Shareable attribute.
- However, between the two clusters, the caches:
  - Are not required to be coherent for data accesses that have only the Inner Shareable attribute.
  - Are coherent for data accesses that have the Outer Shareable attribute.

In this system, each cluster is in a different shareability domain for the Inner Shareable attribute, but all components of the subsystem are in the same shareability domain for the Outer Shareable attribute.

A system might implement two such subsystems. If the data caches or unified caches of one subsystem are not transparent to the accesses from the other subsystem, this system has two Outer Shareable shareability domains.

Having two levels of shareability means system designers can reduce the performance and power overhead for shared memory locations that do not need to be part of the Outer Shareable shareability domain.

For shareable Normal memory, the Load-Exclusive and Store-Exclusive synchronization primitives take account of the possibility of accesses by more than one observer in the same Shareability domain.

Non-shareable Normal memory

For Normal memory locations, the Non-shareable attribute identifies Normal memory that is likely to be accessed only by a single PE.

A location in Normal memory with the Non-shareable attribute does not require the hardware to make data accesses by different observers coherent, unless the memory is Non-cacheable. For a Non-shareable location, if other observers share the memory system, software must use cache maintenance instructions, if the presence of caches might lead to coherency issues when communicating between the observers. This cache maintenance requirement is in addition to the barrier operations that are required to ensure memory ordering.

For Non-shareable Normal memory, it is IMPLEMENTATION DEFINED whether the Load-Exclusive and Store-Exclusive synchronization primitives take account of the possibility of accesses by more than one observer.

Cacheability attributes for Normal memory

In addition to being Outer Shareable, Inner Shareable or Non-shareable, each region of Normal memory is assigned a Cacheability attribute that is one of:

- Write-Through Cacheable.
- Write-Back Cacheable.
- Non-cacheable.

Also, for Write-Through Cacheable and Write-Back Cacheable Normal memory regions:

- A region might be assigned cache allocation hints for read and write accesses.
- It is IMPLEMENTATION DEFINED whether the cache allocation hints can have an additional attribute of Transient or Non-transient.

For more information see *Cacheability, cache allocation hints, and cache transient hints* on page D4-2502.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A memory location can be marked as having different cacheability attributes, for example when using aliases in a VA to PA mapping:

- If the attributes differ only in the cache allocation hint, this does not affect the behavior of accesses to that location.
- For other cases, see *Mismatched memory attributes* on page B2-164.

The cacheability attributes provide a mechanism of coherency control with observers that lie outside the shareability domain of a region of memory. In some cases, the use of Write-Through Cacheable or Non-cacheable regions of memory might provide a better mechanism for controlling coherency than the use of hardware coherency mechanisms or the use of cache maintenance routines. To this end, the architecture requires the following properties for Non-cacheable or Write-Through Cacheable memory:

- A completed write to a memory location that is Non-cacheable or Write-Through Cacheable for a level of cache made by an observer accessing the memory system inside the level of cache is visible to all observers accessing the memory system outside the level of cache without the need of explicit cache maintenance.
- A completed write to a memory location that is Non-cacheable for a level of cache made by an observer accessing the memory system outside the level of cache is visible to all observers accessing the memory system inside the level of cache without the need of explicit cache maintenance.
- For accesses to Normal memory that is Non-cacheable, a `DMB` instruction introduces a Barrier-ordered-before relation on all accesses to a single peripheral or block of memory that is of IMPLEMENTATION DEFINED size. For more information, see *Ordering and observability* on page B2-124.

Note

Implementations can use the cache allocation hints to indicate a probable performance benefit of caching. For example, a programmer might know that a piece of memory is not going to be accessed again and would be better treated as Non-cacheable. The distinction between memory regions with attributes that differ only in the cache allocation hints exists only as a hint for performance.

For Normal memory, the Arm architecture provides cacheability attributes that are defined independently for each of two conceptual levels of cache, the *inner* and the *outer* cache. The relationship between these conceptual levels of cache and the implemented physical levels of cache is IMPLEMENTATION DEFINED, and can differ from the boundaries between the Inner and Outer Shareability domains. However:

- Inner refers to the innermost caches, meaning the caches that are closest to the PE, and always includes the lowest level of cache.
- No cache that is controlled by the Inner cacheability attributes can lie outside a cache that is controlled by the Outer cacheability attributes.
- An implementation might not have any outer cache.

Example B2-2, Example B2-3 on page B2-157, and Example B2-4 on page B2-157 describe the possible ways of implementing a system with three levels of cache, *level 1* (L1) to *level 3* (L3).

Note

- L1 cache is the level closest to the PE, see *Memory hierarchy* on page B2-143.
- When managing coherency, system designs must consider both the inner and outer cacheability attributes, as well as the shareability attributes. This is because hardware might have to manage the coherency of caches at one conceptual level, even when another conceptual level has the Non-cacheable attribute.

Example B2-2 Implementation with two inner and one outer cache levels

Implement the three levels of cache in the system, L1 to L3, with:

- The Inner cacheability attribute applied to L1 and L2 cache.
- The Outer cacheability attribute applied to L3 cache.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Example B2-3 Implementation with three inner and no outer cache levels

Implement the three levels of cache in the system, L1 to L3, with the Inner cacheability attribute applied to L1, L2, and L3 cache. Do not use the Outer cacheability attribute.

Example B2-4 Implementation with one inner and two outer cache levels

Implement the three levels of cache in the system, L1 to L3, with:

- The Inner cacheability attribute applied to L1 cache.
- The Outer cacheability attribute applied to L2 and L3 cache.

Multi-register loads and stores that access Normal memory

For all instructions that load or store more than one general-purpose register from an Exception level there is no requirement for the memory system beyond the PE to be able to identify the size of the elements accessed by these load or store instructions.

For all instructions that load or store more than one general-purpose register from an Exception level the order in which the registers are accessed is not defined by the architecture.

For all instructions that load or store one or more SIMD&FP registers from an Exception level, there is no requirement for the memory system beyond the PE to be able to identify the size of the element accessed by these load or store instructions.

B2.7.2 Device memory

The Device memory type attributes define memory locations where an access to the location can cause side-effects, or where the value returned for a load can vary depending on the number of loads performed. Typically, the Device memory attributes are used for memory-mapped peripherals and similar locations.

The attributes for Armv8 Device memory are:

Gathering Identified as G or nG, see *Gathering* on page B2-159.

Reordering Identified as R or nR, see *Reordering* on page B2-160.

Early Write Acknowledgement

Identified as E or nE, see *Early Write Acknowledgement* on page B2-161.

The Armv8 Device memory types are:

Device-nGnRnE Device non-Gathering, non-Reordering, No Early write acknowledgement.

Equivalent to the Strongly-ordered memory type in earlier versions of the architecture.

Device-nGnRE Device non-Gathering, non-Reordering, Early Write Acknowledgement.

Equivalent to the Device memory type in earlier versions of the architecture.

Device-nGRE Device non-Gathering, Reordering, Early Write Acknowledgement.

Armv8 adds this memory type to the translation table formats found in earlier versions of the architecture. The use of barriers is required to order accesses to Device-nGRE memory.

Device-GRE Device Gathering, Reordering, Early Write Acknowledgement.

Armv8 adds this memory type to the translation table formats found in earlier versions of the architecture. Device-GRE memory has the fewest constraints. It behaves similar to Normal memory, with the restriction that *Speculative* accesses to Device-GRE memory is forbidden.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Collectively these are referred to as *any Device memory type*. Going down the list, the memory types are described as getting *weaker*; conversely the going up the list the memory types are described as getting *stronger*.

Note

- As the list of types shows, these additional attributes are hierarchical. For example, a memory location that permits Gathering must also permit Reordering and Early Write Acknowledgement.
- The architecture does not require an implementation to distinguish between each of these memory types and Arm recognizes that not all implementations will do so. The subsection that describes each of the attributes, describes the implementation rules for the attribute.

All of these memory types have the following properties:

- Speculative data accesses are not permitted to any memory location with any Device memory attribute. This means that each memory access to any Device memory type must be one that would be generated by a simple sequential execution of the program.

  The following exceptions to this apply:

  — Reads generated by the SIMD and floating-point instructions can access bytes that are not explicitly accessed by the instruction if the bytes accessed are in a 16-byte window, aligned to 16-bytes, that contains at least one byte that is explicitly accessed by the instruction.

  — For Device memory with the Gathering attribute, reads generated by the `LDNP` instructions are permitted to access bytes that are not explicitly accessed by the instruction, provided that the bytes accessed are in a 128-byte window, aligned to 128-bytes, that contains at least one byte that is explicitly accessed by the instruction.

  — Where a load or store instruction performs a sequence of memory accesses, as opposed to one single-copy atomic access as defined in the rules for single-copy atomicity, these accesses might occur multiple times as a result of executing the load or store instruction. See *Properties of single-copy atomic accesses* on page B2-120.

  Note

  — An instruction that generates a sequence of accesses as described in *Atomicity in the Arm architecture* on page B2-118 might be abandoned as a result of an exception being taken during the sequence of accesses. On return from the exception, the instruction is restarted, and therefore, one or more of the memory locations might be accessed multiple times. This can result in repeated accesses to a location where the program only defines a single access. For this reason, Arm strongly recommends that no accesses to Device memory are performed from a single instruction that spans the boundary of a translation granule or which in some other way could lead to some of the accesses being aborted.

  — Write speculation that is visible to other observers is prohibited for all memory types.

- A write to a memory location with any Device memory type completes in finite time.
- If a value that would be returned from a read of a memory location with the Device memory type changes without an explicit write by an observer, this change must also be globally observed for all observers in the system in finite time. Such a change might occur in a peripheral location that holds status information.
- Data accesses to memory locations are coherent for all observers in the system, and correspondingly are treated as being Outer Shareable.
- A memory location with any Device memory attribute cannot be allocated into a cache.
- Writes to a memory location with any Device memory attribute must reach the endpoint for that address in the memory system in finite time. Two writes of Device memory type to the same location might be merged before they reach the endpoint, unless both writes have the non-Gathering attribute or there is an ordered-before relationship between the two writes.
- For accesses to any Device memory type, a `DMB` instruction introduces a Barrier-ordered-before relation on all accesses to a single peripheral or block of memory that is of implementation defined size. For more information, see *Ordering and observability* on page B2-124.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- All accesses to memory with any Device memory attribute must be aligned. Any unaligned access generates an Alignment fault at the first stage of translation that defined the location as being Device.

  Note

  In the EL1&0 translation regime in systems where HCR_EL2.TGE==1 and HCR_EL2.DC==0, any Alignment fault that results from the fact that all locations are treated as Device is a fault at the first stage of translation. This causes ESR_EL2.ISS[24] to be 0.

- Hardware does not prevent speculative instruction fetches from a memory location with any of the Device memory attributes unless the memory location is also marked as Execute-never for all Exception levels.

  Note

  This means that to prevent speculative instruction fetches from memory locations with Device memory attributes, any location that is assigned any Device memory type must also be marked as Execute-never for all Exception levels. Failure to mark a memory location with any Device memory attribute as Execute-never for all Exception levels is a programming error.

See also *Memory access restrictions* on page B2-162.

The memory types for translation table walks cannot be defined as any Device memory type within the TCR_ELx. For the EL1&0 translation regime, the memory accesses made during a stage 1 translation table walk are subject to a stage 2 translation, and as a result of this second stage of translation, the accesses from the first stage translation table walk might be made to memory locations with any Device memory type. These accesses might be made speculatively. When the value of the HCR_EL2.PTW bit is 1, a stage 2 permission fault is generated if a first stage translation table walk is made to any Device memory type.

Note

In general, making a translation table walk to any Device memory type is the result of a programming error.

For instruction fetches, if branches cause the program counter to point to an area of memory with the Device attribute which is not marked as Execute-never for the current Exception level, an implementation can either:

- Treat the instruction fetch as if it were to a memory location with the Normal Non-cacheable attribute.
- Take a Permission fault.

Gathering

In the Device memory attribute:

G Indicates that the location has the Gathering attribute.

nG Indicates that the location does not have the Gathering attribute, meaning it is non-Gathering.

The Gathering attribute determines whether it is permissible for either:

- Multiple memory accesses of the same type, read or write, to the same memory location to be merged into a single transaction.
- Multiple memory accesses of the same type, read or write, to different memory locations to be merged into a single memory transaction on an interconnect.

Note

This also applies to writebacks from the cache, whether caused by a *Natural eviction* or as a result of a cache maintenance instruction.

For memory types with the Gathering attribute, either of these behaviors is permitted, provided that the ordering and coherency rules of the memory location are followed.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For memory types with the non-Gathering attribute, neither of these behaviors is permitted. As a result:

- The number of memory accesses that are made corresponds to the number that would be generated by a simple sequential execution of the program.
- All accesses occur at their programmed size, except that there is no requirement for the memory system beyond the PE to be able to identify the elements accessed by multi-register Load/Store instructions. See *Multi-register loads and stores that access Device memory* on page B2-162.

Gathering between memory accesses separated by a memory barrier that affects those memory accesses is not permitted.

Gathering between two memory accesses generated by a Load-Acquire/Store-Release is not permitted.

A read from a memory location with the non-Gathering attribute cannot come from a cache or a buffer, but must come from the endpoint for that address in the memory system. Typically this is a peripheral or physical memory.

Note

- A read from a memory location with the Gathering attribute can come from intermediate buffering of a previous write, provided that:
  - The accesses are not separated by a `DMB` or `DSB` barrier that affects both of the accesses.
  - The accesses are not separated by other ordering constructions that require that the accesses are in order. Such a construction might be a combination of Load-Acquire and Store-Release.
  - The accesses are not generated by a Store-Release instruction.
- The Arm architecture only defines programmer visible behavior. Therefore, gathering can be performed if a programmer cannot tell whether gathering has occurred.

An implementation is permitted to perform an access with the Gathering attribute in a manner consistent with the requirements specified by the non-Gathering attribute.

An implementation is not permitted to perform an access with the non-Gathering attribute in a manner consistent with the relaxations allowed by the Gathering attribute.

Reordering

In the Device memory attribute:

R Indicates that the location has the Reordering attribute. Accesses to the location can be reordered within the same rules that apply to accesses to Normal Non-cacheable memory. All memory types with the Reordering attribute have the same ordering rules as accesses to Normal Non-cacheable memory, see *Ordering relations* on page B2-126.

nR Indicates that the location does not have the Reordering attribute, meaning it is non-Reordering.

Note

Some interconnect fabrics, such as PCIe, perform very limited reordering, which is not important for the software usage. It is outside the scope of the Arm architecture to prohibit the use of a non-Reordering memory type with these interconnects.

For all memory types with the non-Reordering attribute, the order of memory accesses arriving at a single peripheral of IMPLEMENTATION DEFINED size, as defined by the peripheral, must be the same order that occurs in a simple sequential execution of the program. That is, the accesses appear in program order. This ordering applies to all accesses using any of the memory types with the non-Reordering attribute. As a result, if there is a mixture of Device-nGnRE and Device-nGnRnE accesses to the same peripheral, these occur in program order. If the memory accesses are not to a peripheral, then this attribute imposes no restrictions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

- The IMPLEMENTATION DEFINED size of the single peripheral is the same as applies for the ordering guarantee provided by the `DMB` instruction.
- The Arm architecture only defines programmer visible behavior. Therefore, reordering can be performed if a programmer cannot tell whether reordering has occurred.
- The non-Reordering property is only required by the architecture to apply the order of arrival of accesses to a single memory-mapped peripheral of an IMPLEMENTATION DEFINED size, and is not required to have an impact on the order of observation of memory accesses to SDRAM. For this reason, there is no effect of the non-Reordering attribute on the ordering relations between accesses to different locations described in *Ordering and observability* on page B2-124 as part of the formal definition of the memory model.
- If the same memory location is mapped with different aliases, with a different Reordering attribute value or any different Device memory attribute values, these are a type of mismatched attribute. For information about the effects of accessing memory with mismatched attributes, see *Mismatched memory attributes* on page B2-164.

An implementation:

- Is permitted to perform an access with the Reordering attribute in a manner consistent with the requirements specified by the non-Reordering attribute.
- Is not permitted to perform an access with the non-Reordering attribute in a manner consistent with the relaxations allowed by the Reordering attribute.

The non-Reordering attribute does not require any additional ordering, other than that which applies to Normal memory, between:

- Accesses to one physical address with the non-Reordering attribute and accesses to a different physical address with the Reordering attribute.
- Access to one physical address with the non-Reordering attribute and access to a different physical address to Normal memory.
- Accesses with the non-Reordering attribute and accesses to different peripherals of IMPLEMENTATION DEFINED size.

The non-Reordering attribute has no effect on the ordering of cache maintenance instructions, even if the memory location specified in the instruction has the non-Reordering attribute.

Early Write Acknowledgement

In the Device memory attribute:

E Indicates that the location has the Early Write Acknowledgement attribute.

nE Indicates that the location has the No Early Write Acknowledgement attribute.

For memory system endpoints where the system architecture in which the PE is operating requires that acknowledgement of a write comes from the endpoint, assigning the No Early Write Acknowledgement attribute to a Device memory location guarantees that:

- Only the endpoint of the write access returns a write acknowledgement of the access.
- No earlier point in the memory system returns a write acknowledgement.

This means that a `DSB` barrier instruction, executed by the PE that performed the write to the No Early Write Acknowledgement Location, completes only after the write has reached its endpoint in the memory system.

Peripherals are an example of system endpoints that require that the acknowledgement of a write comes from the endpoint.

Note

- The Early Write Acknowledgement attribute only affects where the endpoint acknowledgement is returned from, and does not affect the ordering of arrival at the endpoint between accesses, which is determined by either the Device Reordering attribute, or the use of barriers to create order.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The areas of the physical memory map for which write acknowledgement from the endpoint is required is outside the scope of the Arm Architecture definition and must be defined as part of the system architecture in which the PE is operating. In particular, regions of memory handled as PCIe configuration writes are expected to support write acknowledgement from the endpoint.
- Arm recognizes that not all areas of a physical memory map will be capable of supporting write acknowledgement from the endpoint. In particular, Arm expects that regions of memory handled as posted writes under PCIe will not support write acknowledgement from the endpoint.
- For maximum software compatibility, Arm strongly recommends that all peripherals for which standard software drivers expect that the use of a `DSB` instruction will determine that a write has reached its endpoint are placed in areas of the physical memory map that support write acknowledgement from the endpoint.

Multi-register loads and stores that access Device memory

For all instructions that load or store more than one general-purpose register from an Exception level there is no requirement for the memory system beyond the PE to be able to identify the size of the elements accessed by these load or store instructions.

For all instructions that load or store more than one general-purpose register from an Exception level the order in which the registers are accessed is not defined by the architecture. This applies even to accesses to any type of Device memory.

For all instructions that load or store one or more SIMD and floating-point registers from an Exception level, there is no requirement for the memory system beyond the PE to be able to identify the size of the element accessed by these load or store instructions, even for access to any type of Device memory.

B2.7.3 Memory access restrictions

The following restrictions apply to memory accesses:

- For accesses to any two bytes, *p* and *q*, that are generated by the same instruction:
  - The bytes *p* and *q* must have the same memory type and shareability attributes, otherwise the results are CONSTRAINED UNPREDICTABLE. For example, an `LD1`, `ST1`, or an unaligned load or store that spans the boundary between Normal memory and Device memory is CONSTRAINED UNPREDICTABLE.
  - Except for possible differences in the cache allocation hints, Arm deprecates having different cacheability attributes for bytes *p* and *q*.

  For the permitted CONSTRAINED UNPREDICTABLE behavior, see *Crossing a page boundary with different memory types or Shareability attributes* on page K1-7861.
- If the accesses of an instruction that causes multiple accesses to any type of Device memory cross an address boundary that corresponds to the smallest implemented translation granule, then behavior is CONSTRAINED UNPREDICTABLE, and *Crossing a peripheral boundary with a Device access* on page K1-7861 describes the permitted behaviors. For this reason, it is important that an access to a volatile memory device is not made using a single instruction that crosses an address boundary of the size of the smallest implemented translation granule.

  Note

  - The boundary referred to is between two Device memory regions that are both of the size of the smallest implemented translation granule and aligned to the size of the smallest implemented translation granule.
  - This restriction means it is important that an access to a volatile memory device is not made using a single instruction that crosses an address boundary of the size of the smallest implemented translation granule.
  - Arm expects this restriction to constrain the placing of volatile memory devices in the system memory map, rather than expecting a compiler to be aware of the alignment of memory accesses.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

B2.8 Mismatched memory attributes

Memory attributes are controlled by privileged software. For more information, see Chapter D5 *The AArch64 Virtual Memory System Architecture*.

Physical memory locations are accessed with *mismatched attributes* if all accesses to the location do not use a common definition of all of the following attributes of that location:

- Memory type, Device or Normal.
- Shareability.
- Cacheability, for the same level of the inner or outer cache, but excluding any cache allocation hints.

Collectively these are referred to as memory attributes.

If ARMv8.5-MemTag is implemented, accesses to a location which use a common definition of the memory attributes but the Tagged attribute of that location differs do not cause a mismatched access to occur.

Note

In this document, the terms *location* and *memory location* refer to any byte within the current coherency granule and are used interchangeably.

When a memory Location is accessed with mismatched attributes, the only software visible effects are one or more of the following:

- Uniprocessor semantics for reads and writes to that memory Location might be lost. This means:
  - A read of the memory Location by one agent might not return the value most recently written to that memory Location by the same agent.
  - Multiple writes to the memory Location by one agent with different memory attributes might not be ordered in program order.
- There might be a loss of coherency when multiple agents attempt to access a memory Location.
- There might be a loss of properties derived from the memory type, as described in later bullets in this section.
- If all Load-Exclusive/Store-Exclusive instructions executed across all threads to access a given memory Location do not use consistent memory attributes, the Exclusives monitor state becomes UNKNOWN.
- Bytes written without the Write-Back cacheable attribute within the same Write-Back granule as bytes written with the Write-Back cacheable attribute might have their values reverted to the old values as a result of cache Write-Back.

The loss of properties associated with mismatched memory type attributes refers only to the following properties of Device memory that are additional to the properties of Normal memory:

- Prohibition of *Speculative* read accesses.
- Prohibition on Gathering.
- Prohibition on reordering.

For the following situations, when a physical memory Location is accessed with mismatched attributes, a more restrictive set of behaviors applies. The description of each situation also describes the behaviors that apply:

1. Any agent that reads that memory Location using the same common definition of the Memory type, Shareability and Cacheability attributes is guaranteed to access it coherently, to the extent required by that common definition of the memory attributes, only if all the following conditions are met:
   - All writes are performed to an alias of the memory Location that uses the same definition of the Memory type, Shareability and Cacheability attributes.
   - Either:
     - In the EL1&0 translation regime, HCR_EL2.MIOCNCE has a value of 0.
     - All aliases with write permission have the Inner Cacheability attribute the same as the Outer Cacheability attribute.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Either:
  - All writes are performed to an alias of the memory Location that has Inner Cacheability and Outer Cacheability attributes both as Non-cacheable.
  - All aliases to a memory Location use a definition of the Shareability attributes that encompasses all the agents with permission to access the Location.

2. The possible software-visible effects caused by mismatched attributes for a memory Location are defined more precisely if all of the mismatched attributes define the memory Location as one of:
   - Any Device memory type.
   - Inner Non-cacheable, Outer Non-cacheable Normal memory.

   In these cases, the only permitted software-visible effects of the mismatched attributes are one or more of the following:
   - Possible loss of properties derived from the memory type when multiple agents attempt to access the memory Location.
   - Possible reordering of memory transactions to the same memory Location with different memory attributes, potentially leading to a loss of coherency or uniprocessor semantics. Any possible loss of coherency or uniprocessor semantics can be avoided by inserting `DMB` barrier instructions between accesses to the same memory Location that might use different attributes.

Where there is a loss of the uniprocessor semantics, ordering, or coherency, the following approaches can be used:

1. If the mismatched attributes for a memory location all assign the same shareability attribute to a Location that has a cacheable attribute, any loss of uniprocessor semantics, ordering, or coherency within a shareability domain can be avoided by use of software cache management. To do so, software must use the techniques that are required for the software management of the ordering or coherency of cacheable Locations between agents in different shareability domains. This means:
   - Before writing to a cacheable Location not using the Write-Back attribute, software must invalidate, or clean, a Location from the caches if any agent might have written to the Location with the Write-Back attribute. This avoids the possibility of overwriting the Location with stale data.
   - After writing to a cacheable Location with the Write-Back attribute, software must clean the Location from the caches, to make the write visible to external memory.
   - Before reading the Location with a cacheable attribute, software must invalidate, or clean and invalidate, the Location from the caches, to ensure that any value held in the caches reflects the last value made visible in external memory.
   - Executing a `DMB` barrier instruction, with scope that applies to the common shareability of the accesses, between any accesses to the same cacheable Location that use different attributes.

   In all cases:
   - Location refers to any byte within the current coherency granule.
   - A clean and invalidate instruction can be used instead of a clean instruction, or instead of an invalidate instruction.
   - In the sequences outlined in this section, all cache maintenance instructions and memory transactions must be completed, or ordered by the use of barrier operations, if they are not naturally ordered by the use of a common address, see *Ordering and completion of data and instruction cache instructions* on page D4-2517.

   Note

   With software management of coherency, race conditions can cause loss of data. A race condition occurs when different agents write simultaneously to bytes that are in the same Location, and the invalidate, write, clean sequence of one agent overlaps with the equivalent sequence of another agent. A race condition also occurs if the first operation of either sequence is a clean, rather than an invalidate.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

2. If the mismatched attributes for a Location mean that multiple cacheable accesses to the Location might be made with different shareability attributes, then uniprocessor semantics, ordering, and coherency are guaranteed only if:
   - Each PE that accesses the Location with a cacheable attribute performs a clean and invalidate of the Location before and after accessing that Location.
   - A `DMB` barrier with scope that covers the full shareability of the accesses is placed between any accesses to the same memory Location that use different attributes.

   Note

   The Note in rule 1 of this list, about possible race conditions, also applies to this rule.

In addition, if multiple agents attempt to use Load-Exclusive or Store-Exclusive instructions to access a Location, and the accesses from the different agents have different memory attributes associated with the Location, the Exclusives monitor state becomes UNKNOWN.

Arm strongly recommends that software does not use mismatched attributes for aliases of the same Location. An implementation might not optimize the performance of a system that uses mismatched aliases.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

B2.9 Synchronization and semaphores

Armv8 provides non-blocking synchronization of shared memory, using *synchronization primitives*. The information in this section about memory accesses by synchronization primitives applies to accesses to both Normal memory and to any type of Device memory.

Note

Use of the Armv8 synchronization primitives scales for multiprocessing system designs.

Table B2-3 shows the synchronization primitives and the associated CLREX instruction.

Table B2-3 Synchronization primitives and associated instruction, A64 instruction set

Transaction size	Additional semantics	Load-Exclusive[a]	Store-Exclusive[a]	Other[a]
Byte	-	LDXRB	STXRB	-
	Load-Acquire/Store-Release	LDAXRB	STLXRB	-
Halfword	-	LDXRH	STXRH	-
	Load-Acquire/Store-Release	LDAXRH	STLXRH	-
Register[b]	-	LDXR	STXR	-
	Load-Acquire/Store-Release	LDAXR	STLXR	-
Pair[b]	-	LDXP	STXP	-
	Load-Acquire/Store-Release	LDAXP	STLXP	-
None	Clear-Exclusive	-	-	CLREX

a. Instruction in the A64 instruction set.
b. A register instruction operates on a doubleword if accessing an X register, or on a word if accessing a W register
A pair instruction operates on two doublewords if access X registers, or on two words if accessing W registers.

Except for the row showing the CLREX instruction, the two instructions in a single row are a Load-Exclusive/Store-Exclusive instruction pair. The model for the use of a Load-Exclusive/Store-Exclusive instruction pair accessing a non-aborting memory address *x* is:

- The Load-Exclusive instruction reads a value from memory address *x*.
- The corresponding Store-Exclusive instruction succeeds in writing back to memory address *x* only if no other observer, process, or thread has performed a more recent store to address *x*. The Store-Exclusive instruction returns a status bit that indicates whether the memory write succeeded.

A Load-Exclusive instruction marks a small block of memory for exclusive access. The size of the marked block is IMPLEMENTATION DEFINED, see *Marking and the size of the marked memory block* on page B2-173. A Store-Exclusive instruction to any address in the marked block clears the marking.

Note

In this section, the term PE includes any observer that can generate a Load-Exclusive or a Store-Exclusive instruction.

The following sections give more information:

- *Exclusive access instructions and Non-shareable memory locations* on page B2-168.
- *Exclusive access instructions and Shareable memory locations* on page B2-169.
- *Marking and the size of the marked memory block* on page B2-173.
- *Context switch support* on page B2-173.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *Load-Exclusive and Store-Exclusive instruction usage restrictions* on page B2-174.
- *Use of WFE and SEV instructions by spin-locks* on page B2-177.

B2.9.1 Exclusive access instructions and Non-shareable memory locations

For memory locations for which the shareability attribute is Non-shareable, the exclusive access instructions rely on a *local Exclusives monitor*, or *local monitor*, that marks any address from which the PE executes a Load-Exclusive instruction. Any non-aborted attempt by the same PE to use a Store-Exclusive instruction to modify any address is guaranteed to clear the marking.

A Load-Exclusive instruction performs a load from memory, and:

- The executing PE marks the physical memory address for exclusive access.
- The local monitor of the executing PE transitions to the Exclusive Access state.

A Store-Exclusive instruction performs a conditional store to memory that depends on the state of the local monitor:

If the local monitor is in the Exclusive Access state

- If the address of the Store-Exclusive instruction is the same as the address that has been marked in the monitor by an earlier Load-Exclusive instruction, then the store occurs. Otherwise, it is IMPLEMENTATION DEFINED whether the store occurs.
- A status value is returned to a register:
  - If the store took place, the status value is 0.
  - Otherwise, the status value is 1.
- The local monitor of the executing PE transitions to the Open Access state.

When an Exclusives monitor is in the Exclusive Access state, the monitor is *set*.

If the local monitor is in the Open Access state

- No store takes place.
- A status value of 1 is returned to a register.
- The local monitor remains in the Open Access state.

When an Exclusives monitor is in the Open Access state, the monitor is *clear*.

The Store-Exclusive instruction defines the register to which the status value is returned.

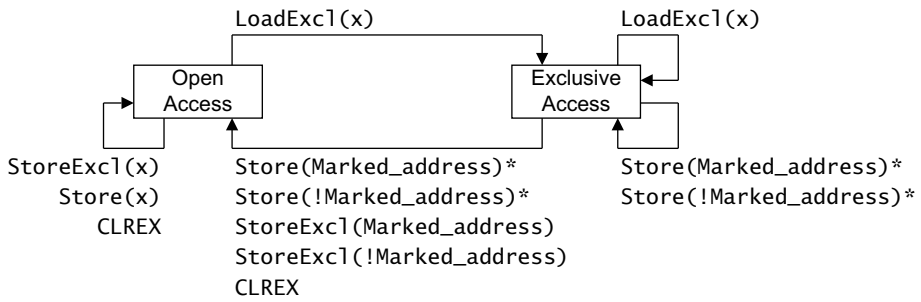
When a PE writes using any instruction other than a Store-Exclusive instruction:

- If the write is to a PA that is not marked as Exclusive Access by its local monitor and that local monitor is in the Exclusive Access state, it is IMPLEMENTATION DEFINED whether the write affects the state of the local monitor.
- If the write is to a PA that is marked as Exclusive Access by its local monitor, it is IMPLEMENTATION DEFINED whether the write affects the state of the local monitor.

It is IMPLEMENTATION DEFINED whether a store to a marked PA causes a mark in the local monitor to be cleared if that store is by an observer other than the one that caused the PA to be marked.

Figure B2-4 on page B2-169 shows the state machine for the local monitor and the effect of each of the operations shown in the figure.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta



Operations marked * are possible alternative IMPLEMENTATION DEFINED options.
In the diagram: `LoadExcl` represents any Load-Exclusive instruction
`StoreExcl` represents any Store-Exclusive instruction
`Store` represents any other store instruction.

Any `LoadExcl` operation updates the marked address to the most significant bits of the address x used for the operation.

Figure B2-4 Local monitor state machine diagram

For more information about marking, see *Marking and the size of the marked memory block* on page B2-173.

Note

For the local monitor state machine, as shown in Figure B2-4:

- The IMPLEMENTATION DEFINED options for the local monitor are consistent with the local monitor being constructed so that it does not hold any PA, but instead treats any access as matching the address of the previous Load-Exclusive instruction.
- A local monitor implementation can be unaware of Load-Exclusive and Store-Exclusive instructions from other PEs.
- The architecture does not require a load instruction by another PE, that is not a Load-Exclusive instruction, to have any effect on the local monitor.
- It is IMPLEMENTATION DEFINED whether the transition from Exclusive Access to Open Access state occurs when the `Store` or `StoreExcl` is from another observer.

Changes to the local monitor state resulting from speculative execution

The architecture permits a local monitor to transition to the Open Access state as a result of speculation, or from some other cause. This is in addition to the transitions to Open Access state caused by the architectural execution of an operation shown in Figure B2-4.

An implementation must ensure that:

- The local monitor cannot be seen to transition to the Exclusive Access state except as a result of the architectural execution of one of the operations shown in Figure B2-4.
- Any transition of the local monitor to the Open Access state not caused by the architectural execution of an operation shown in Figure B2-4 must not indefinitely delay forward progress of execution.

B2.9.2 Exclusive access instructions and Shareable memory locations

In the context of this section, a shareable memory location is a memory location that has, or is treated as if it has, a Shareability attribute of Inner Shareable or Outer Shareable.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For shareable memory locations, exclusive access instructions rely on:

- A *local monitor* for each PE in the system, that marks any address from which the PE executes a Load-Exclusive. The local monitor operates as described in *Exclusive access instructions and Non-shareable memory locations* on page B2-168, except that for shareable memory any Store-Exclusive is then subject to checking by the global monitor if it is described in that section as doing at least one of the following:
  - Updating memory.
  - Returning a status value of 0.

  The local monitor can ignore accesses from other PEs in the system.

- A *global monitor* that marks a PA as exclusive access for a particular PE. This marking is used later to determine whether a Store-Exclusive to that address that has not been failed by the local monitor can occur. Any successful write to the marked block by any other observer in the shareability domain of the memory location is guaranteed to clear the marking. For each PE in the system, the global monitor:
  - Can hold at least one marked block.
  - Maintains a state machine for each marked block it can hold.

  Note

  For each PE, the architecture only requires global monitor support for a single marked address. Any situation that might benefit from the use of multiple marked addresses on a single PE is UNPREDICTABLE or CONSTRAINED UNPREDICTABLE, see *Load-Exclusive and Store-Exclusive instruction usage restrictions* on page B2-174.

Note

The global monitor can either reside within the PE, or exist as a secondary monitor at the memory interfaces. The IMPLEMENTATION DEFINED aspects of the monitors mean that the global monitor and local monitor can be combined into a single unit, provided that the unit performs the global monitor and local monitor functions defined in this manual.

For shareable memory locations, in some implementations and for some memory types, the properties of the global monitor require functionality outside the PE. Some system implementations might not implement this functionality for all locations of memory. In particular, this can apply to:

- Any type of memory in the system implementation that does not support hardware cache coherency.
- Non-cacheable memory, or memory treated as Non-cacheable, in an implementation that does support hardware cache coherency.

In such a system, it is defined by the system:

- Whether the global monitor is implemented.
- If the global monitor is implemented, which address ranges or memory types it monitors.

Note

If ARMv8.5-MemTag is implemented and enabled it is IMPLEMENTATION DEFINED whether a global monitor monitors access the Tag PA space. For more information see Chapter D6 *Armv8.5 Memory Tagging Extension*.

Note

To support the use of the Load-Exclusive/Store-Exclusive mechanism when address translation is disabled, a system might define at least one location of memory, of at least the size of the translation granule, in the system memory map to support the global monitor for all Arm PEs within a common Inner Shareable domain. However, this is not an architectural requirement. Therefore, architecturally-compliant software that requires mutual exclusion must not rely on using the Load-Exclusive/Store-Exclusive mechanism, and must instead use a software algorithm such as Lamport's Bakery algorithm to achieve mutual exclusion.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Because implementations can choose which memory types are treated as Non-cacheable, the only memory types for which it is architecturally guaranteed that a global Exclusives monitor is implemented are:

- Inner Shareable, Inner Write-Back, Outer Write-Back Normal memory with Read allocation hints and Write allocation hints and not transient.
- Outer Shareable, Inner Write-Back, Outer Write-Back Normal memory with Read allocation hints and Write allocation hints and not transient.

If the global monitor is not implemented for an address range or memory type, then performing a Load-Exclusive or a Store-Exclusive instruction to such a location has one or more of the following effects:

- The instruction generates an External abort.
- The instruction generates an IMPLEMENTATION DEFINED MMU fault. This is reported using the Data Abort Fault status code of ESR_ELx.DFSC = `110101`.

  If the IMPLEMENTATION DEFINED MMU fault is generated for the EL1&0 translation regime then:

  — If the fault is generated because of the memory type defined in the first stage of translation, or if the second stage of translation is disabled, then this is a first stage fault and the exception is taken to EL1.

  — Otherwise, the fault is a second stage fault and the exception is taken to EL2.

  The priority of this fault is IMPLEMENTATION DEFINED.
- The instruction is treated as a `NOP`.
- The Load-Exclusive instruction is treated as if it were accessing a Non-shareable location, but the state of the local monitor becomes UNKNOWN.
- The Store-Exclusive instruction is treated as if it were accessing a Non-shareable location, but the state of the local monitor becomes UNKNOWN. In this case, if the store exclusive instruction is a store exclusive pair of 64-bit quantities, then the two quantities being stored might not be stored atomically.
- The value held in the result register of the Store-Exclusive instruction becomes UNKNOWN.

In addition, for write transactions generated by non-PE observers that do not implement exclusive accesses or other atomic access mechanisms, the effect that writes have on the global and local monitors used by Arm PEs is IMPLEMENTATION DEFINED. The writes might not clear the global monitors of other PEs for:

- Some address ranges.
- Some memory types.

Operation of the global Exclusives monitor

A Load-Exclusive instruction from shareable memory performs a load from memory, and causes the PA of the access to be marked as exclusive access for the requesting PE. This access can also cause the exclusive access mark to be removed from any other PA that has been marked by the requesting PE.

Note

The global monitor only supports a single outstanding exclusive access to shareable memory per PE.

A Load-Exclusive instruction by one PE has no effect on the global monitor state for any other PE.

A Store-Exclusive instruction performs a conditional store to memory:

- The store is guaranteed to succeed only if the PA accessed is marked as exclusive access for the requesting PE and both the local monitor and the global monitor state machines for the requesting PE are in the Exclusive Access state. In this case:

  — A status value of 0 is returned to a register to acknowledge the successful store.

  — The final state of the global monitor state machine for the requesting PE is IMPLEMENTATION DEFINED.

  — If the address accessed is marked for exclusive access in the global monitor state machine for any other PE, then that state machine transitions to Open Access state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If no address is marked as exclusive access for the requesting PE, the store does not succeed:
  - A status value of 1 is returned to a register to indicate that the store failed.
  - The global monitor is not affected and remains in Open Access state for the requesting PE.
- If a different PA is marked as exclusive access for the requesting PE, it is IMPLEMENTATION DEFINED whether the store succeeds or not:
  - If the store succeeds a status value of 0 is returned to a register, otherwise a value of 1 is returned.
  - If the global monitor state machine for the PE was in the Exclusive Access state before the Store-Exclusive instruction it is IMPLEMENTATION DEFINED whether that state machine transitions to the Open Access state.

The Store-Exclusive instruction defines the register to which the status value is returned.

In a shared memory system, the global monitor implements a separate state machine for each PE in the system. The state machine for accesses to shareable memory by PE(n) can respond to all the shareable memory accesses visible to it. This means that it responds to:

- Accesses generated by PE(n).
- Accesses generated by the other observers in the shareability domain of the memory location. These accesses are identified as (!n).

In a shared memory system, the global monitor implements a separate state machine for each observer that can generate a Load-Exclusive or a Store-Exclusive instruction in the system.

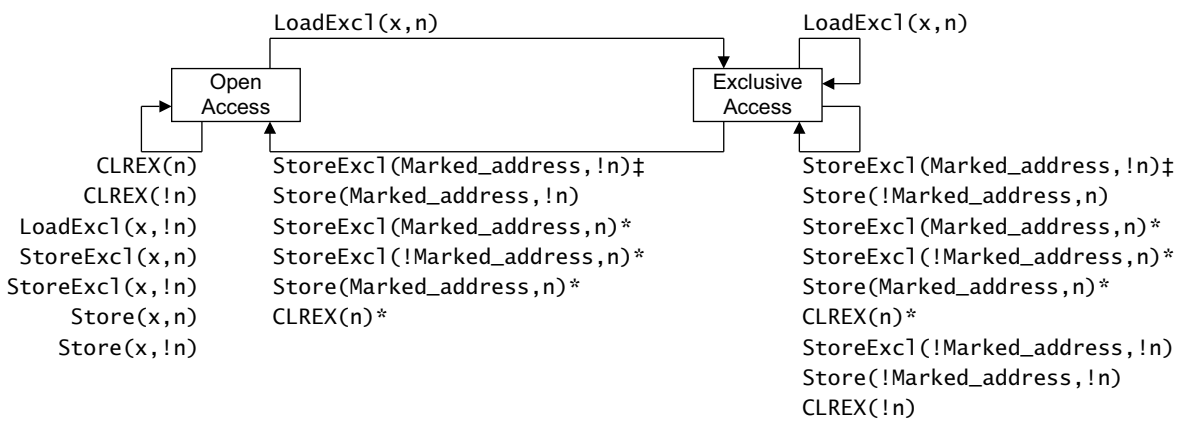
A global monitor:

- In the Exclusive Access state is *set*.
- In the Open Access state is *clear*.

Clear global monitor event

Whenever the global monitor state for a PE changes from Exclusive access to Open access, an event is generated and held in the Event register for that PE. This register is used by the Wait for Event mechanism, see *Mechanisms for entering a low-power state* on page D1-2397.

Figure B2-5 shows the state machine for PE(n) in a global monitor.



‡`StoreExcl`(Marked_address,!n) clears the monitor only if the `StoreExcl` updates memory
Operations marked * are possible alternative IMPLEMENTATION DEFINED options.
In the diagram: `LoadExcl` represents any Load-Exclusive instruction
`StoreExcl` represents any Store-Exclusive instruction
`Store` represents any other store instruction.

Any `LoadExcl` operation updates the marked address to the most significant bits of the address x used for the operation.

Figure B2-5 Global monitor state machine diagram for PE(n) in a multiprocessor system

For more information about marking, see *Marking and the size of the marked memory block* on page B2-173.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

For the global monitor state machine, as shown in Figure B2-5 on page B2-172:

- The architecture does not require a load instruction by another PE, that is not a Load-Exclusive instruction, to have any effect on the global monitor.
- Whether a Store-Exclusive instruction successfully updates memory or not depends on whether the address accessed matches the marked shareable memory address for the PE issuing the Store-Exclusive instruction, and whether the local and global monitors are in the exclusive state. For this reason, Figure B2-5 on page B2-172 only shows how the operations by (!n) cause state transitions of the state machine for PE(n).
- A Load-Exclusive instruction can only update the marked shareable memory address for the PE issuing the Load-Exclusive instruction.
- When the global monitor is in the Exclusive Access state, it is IMPLEMENTATION DEFINED whether a `CLREX` instruction causes the global monitor to transition from Exclusive Access to Open Access state.
- It is IMPLEMENTATION DEFINED:
  - Whether a modification to a Non-shareable memory location can cause a global monitor to transition from Exclusive Access to Open Access state.
  - Whether a Load-Exclusive instruction to a Non-shareable memory location can cause a global monitor to transition from Open Access to Exclusive Access state.

B2.9.3 Marking and the size of the marked memory block

When a Load-Exclusive instruction is executed, the resulting marked block ignores the least significant bits of the 64-bit memory address.

When a Load-Exclusive instruction is executed, a marked block of size 2^a bytes is created by ignoring the least significant bits of the memory address. A marked address is any address within this marked block. The size of the marked memory block is called the *Exclusives reservation granule*. The Exclusives reservation granule is IMPLEMENTATION DEFINED in the range 4-512 words.

Note

This definition means that the Exclusives reservation granule is:

- 4 words in an implementation where a is 4.
- 512 words in an implementation where a is 11.

For example, in an implementation where a is 4, a successful `LDXRB` of address `0x341B4` defines a marked block using bits[47:4] of the address. This means that the four words of memory from `0x341B0` to `0x341BF` are marked for exclusive access.

In some implementations the CTR identifies the Exclusives reservation granule, see CTR_EL0. Otherwise, software must assume that the maximum Exclusives reservation granule, 512 words, is implemented.

B2.9.4 Context switch support

An exception return clears the local monitor. As a result, performing a `CLREX` instruction as part of a context switch is not required in most situations.

Note

Context switching is not an application level operation. However, this information is included here to complete the description of the exclusive operations.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

B2.9.5 Load-Exclusive and Store-Exclusive instruction usage restrictions

The Load-Exclusive and Store-Exclusive instructions are intended to work together as a pair, for example a `LDXP`/`STXP` pair or a `LDXR`/`STXR` pair. To support different implementations of these functions, software must follow the notes and restrictions given here.

The following notes describe the use of a `LoadExcl`/`StoreExcl` instruction pair, to indicate the use of any of the Load-Exclusive/Store-Exclusive instruction pairs shown in Table B2-3 on page B2-167. In this context, a `LoadExcl`/`StoreExcl` pair comprises two instructions in the same thread of execution:

- The exclusives support a single outstanding exclusive access for each PE thread that is executed. The architecture makes use of this by not requiring an address or size check as part of the `IsExclusiveLocal()` function. If the target VA of a `StoreExcl` is different from the VA of the preceding `LoadExcl` instruction in the same thread of execution, behavior can be CONSTRAINED UNPREDICTABLE with the following behavior:
  - The `StoreExcl` either passes or fails, the status value returned by the `StoreExcl` is UNKNOWN, and the states of the local and global monitors for that PE are UNKNOWN.

    Note

    This means the `StoreExcl` might pass for some instances of a `LoadExcl`/`StoreExcl` pair with mismatched addresses, and fail for other instances of a `LoadExcl`/`StoreExcl` pair with mismatched addresses.

  - The data at the address accessed by the `LoadExcl`, and at the address accessed by the `StoreExcl`, is UNKNOWN.

  This means software can rely on a `LoadExcl`/`StoreExcl` pair to eventually succeed only if the `LoadExcl` and the `StoreExcl` are executed with the same VA.

- An implementation of the Load-Exclusive and Store-Exclusive instructions can require that, in any thread of execution, the transaction size of a `StoreExcl` instruction is the same as the transaction size of the preceding `LoadExcl` instruction executed in that thread. If the transaction size of a `StoreExcl` instruction is different from the preceding `LoadExcl` instruction in the same thread of execution, behavior can be CONSTRAINED UNPREDICTABLE with the following behavior:
  - The `StoreExcl` either passes or fails, and the status value returned by the `StoreExcl` is UNKNOWN.

    Note

    This means the `StoreExcl` might pass for some instances of a `LoadExcl`/`StoreExcl` pair with mismatched transaction sizes, and fail for other instances of a `LoadExcl`/`StoreExcl` pair with mismatched transaction sizes.

  - The block of data of the size of the larger of the transaction sizes used by the `LoadExcl`/`StoreExcl` pair at the address accessed by the `LoadExcl`/`StoreExcl` pair, is UNKNOWN.

  This means software can rely on a `LoadExcl`/`StoreExcl` pair to eventually succeed only if the `LoadExcl` and the `StoreExcl` have the same transaction size.

- An implementation of the `LoadExcl` and `StoreExcl` instructions can require that, in any thread of execution, the `StoreExcl` instruction accesses the same number of registers as the preceding `LoadExcl` instruction executed in that thread. If the `StoreExcl` instruction accesses a different number of registers than the preceding `LoadExcl` instruction in the same thread of execution, behavior is CONSTRAINED UNPREDICTABLE. As a result, software can rely on an `LoadExcl`/`StoreExcl` pair to eventually succeed only if they access the same number of registers. For more information, see *CONSTRAINED UNPREDICTABLE behavior when Load-Exclusive/Store-Exclusive access a different number of registers* on page B2-177.

- An implementation of the Load-Exclusive and Store-Exclusive instructions can require that, in any thread of execution, the Tag Checked property of a memory access due to a `StoreExcl` instruction is the same as the Tag Checked property of a memory access by the preceding `LoadExcl` instruction executed in that thread. If the Tag Checked property of memory accesses due to a `LoadExcl`/`StoreExcl` pair in the same thread of execution differ, behavior can be CONSTRAINED UNPREDICTABLE with the following behavior:
  - The `StoreExcl` either passes or fails, and the status value returned by the `StoreExcl` is UNKNOWN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

This means the `StoreExcl` might pass for some instances of such a `LoadExcl`/`StoreExcl` pair, and fail for other instances of such a `LoadExcl`/`StoreExcl` pair.

— The data at the address accessed by the LoadExcl/StoreExcl pair is UNKNOWN.

This means software can rely on a `LoadExcl`/`StoreExcl` pair to eventually succeed only if the memory is accessed with the same Tag Checked property.

- `LoadExcl`/`StoreExcl` loops are guaranteed to make forward progress only if, for any `LoadExcl`/`StoreExcl` loop within a single thread of execution, the software meets all of the following conditions:

  1 Between the Load-Exclusive and the Store-Exclusive, there are no explicit memory accesses, preloads, direct or indirect System register writes, address translation instructions, cache or TLB maintenance instructions, exception generating instructions, exception returns, or indirect branches.

  2 Between the Store-Exclusive returning a failing result and the retry of the corresponding Load-Exclusive:

  - There are no stores or `PRFM` instructions to any address within the Exclusives reservation granule accessed by the Store-Exclusive.
  - There are no loads or preloads to any address within the Exclusives reservation granule accessed by the Store-Exclusive that use a different VA alias to that address.
  - There are no direct or indirect System register writes, address translation instructions, cache or TLB maintenance instructions, exception generating instructions, exception returns, or indirect branches.
  - All loads and stores are to a block of contiguous virtual memory of not more than 512 bytes in size.

  The Exclusives monitor can be cleared at any time without an application-related cause, provided that such clearing is not systematically repeated so as to prevent the forward progress in finite time of at least one of the threads that is accessing the Exclusives monitor. However, it is permissible for the `LoadExcl`/`StoreExcl` loop not to make forward progress if a different thread is repeatedly doing any of the following in a tight loop:

  — Performing stores to a PA covered by the Exclusives monitor.

  — Prefetching with intent to write to a PA covered by the Exclusives monitor.

  — Executing data cache clean, data cache invalidate, or data cache clean and invalidate instructions to a PA covered by the Exclusives monitor.

  — Executing instruction cache invalidate all instructions.

  — Executing instruction cache invalidate by VA instructions to a PA covered by the Exclusives monitor.

  — Executing TLB maintenance to a PA covered by the Exclusives monitor.

- Implementations can benefit from keeping the `LoadExcl` and `StoreExcl` operations close together in a single thread of execution. This minimizes the likelihood of the Exclusives monitor state being cleared between the `LoadExcl` instruction and the `StoreExcl` instruction. Therefore, for best performance, Arm strongly recommends a limit of 128 bytes between `LoadExcl` and `StoreExcl` instructions in a single thread of execution.

- The architecture sets an upper limit of 2048 bytes on the Exclusives reservation granule that can be marked as exclusive. For performance reasons, Arm recommends that objects that are accessed by exclusive accesses are separated by the size of the Exclusives reservation granule. This is a performance guideline rather than a functional requirement.

- After taking a Data Abort exception, the state of the Exclusives monitors is UNKNOWN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- For the memory location accessed by a `LoadExcl`/`StoreExcl` pair, if the memory attributes for a `StoreExcl` instruction are different from the memory attributes for the preceding `LoadExcl` instruction in the same thread of execution, behavior is CONSTRAINED UNPREDICTABLE. Where this occurs because the translation of the accessed address changes between the `LoadExcl` instruction and the `StoreExcl` instruction, the CONSTRAINED UNPREDICTABLE behavior is as follows:
  - The `StoreExcl` either passes or fails, and the status value returned by the `StoreExcl` is UNKNOWN.

    Note

    This means the `StoreExcl` might pass for some instances of a `LoadExcl`/`StoreExcl` pair with changed memory attributes, and fail for other instances of a `LoadExcl`/`StoreExcl` pair with changed memory attributes.
  - The data at the address accessed by the `StoreExcl` is UNKNOWN.

  Note

  Another bullet point in this list covers the case where the memory attributes of a `LoadExcl`/`StoreExcl` pair differ as a result of using different VAs with different attributes that point to the same PA.

- The effect of a data or unified cache invalidate, clean, or clean and invalidate instruction on a local or global Exclusives monitor that is in the Exclusive Access state is CONSTRAINED UNPREDICTABLE, and the instruction might clear the monitor, or it might leave it in the Exclusive Access state. For address-based maintenance instructions, this also applies to the monitors of other PEs in the same shareability domain as the PE executing the cache maintenance instruction, as determined by the shareability domain of the address being maintained.

  Note

  Arm strongly recommends that implementations ensure that the use of such maintenance instructions by a PE in the Non-secure state cannot cause a denial of service on a PE in the Secure state.

- If the mapping of the VA to PA is changed between the `LoadExcl` instruction and the `STREX` instruction, and the change is performed using a break-before-make sequence as described in *Using break-before-make when updating translation table entries* on page D5-2669, if the `StoreExcl` is performed after another write to the same PA as the `StoreExcl`, and that other write was performed after the old translation was properly invalidated and that invalidation was properly synchronized, then the `StoreExcl` will not pass its monitor check.

  Note

  - The TLB invalidation will clear either the local or global monitor.
  - The PA will be checked between the `LoadExcl` and `StoreExcl`.

- The Exclusive Access state for an address accessed by a PE can be lost as a result of a `PFRM PST*` instruction to the same PA executed by another PE. This means that a very high rate of repeated `PFRM PST*` accesses to a memory location might impede the forward progress of another PE.

- If ARMv8.5-MemTag is implemented and enabled a store exclusive instruction which if Unchecked would not perform the store and return a status value of 1, it is CONSTRAINED UNPREDICTABLE whether:
  - The instruction is a Checked access,
  - The instruction is an Unchecked access.

  For more information see Chapter D6 *Armv8.5 Memory Tagging Extension*.

Note

In the event of repeatedly-contending `LoadExcl`/`StoreExcl` instruction sequences from multiple PEs, an implementation must ensure that forward progress is made by at least one PE.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CONSTRAINED UNPREDICTABLE behavior when Load-Exclusive/Store-Exclusive access a different number of registers

As stated in this section, an implementation can require that the instructions of a Load-Exclusive/Store-Exclusive pair access the same number of registers. In such an implementation, this means behavior is CONSTRAINED UNPREDICTABLE if, in a single thread of execution, either:

- An `LDXP` instruction of two 32-bit quantities is followed by an `STXR` instruction of one 64-bit quantity at the same address.
- An `LDXR` instruction of one 64-bit quantity is followed by an `STXP` instruction of two 32-bit quantities at the same address.

In these cases, the CONSTRAINED UNPREDICTABLE behavior must be one of:

- The `STXP` or `STXR` instruction generates an external Data Abort.
- The `STXP` or `STXR` instruction generates an IMPLEMENTATION DEFINED MMU fault reported using the Data Abort Fault status code of ESR_ELx.DFSC = `0b110101`.
- The `STXP` or `STXR` instruction always fails, returning a status of 1.
- The `STXP` or `STXR` instruction always passes, returning a status of 0.
- This `STXP` or `STXR` instruction has the same pass or fail behavior that it would have had if the instruction had used the same size and number of registers as the preceding `LDXR` or `LDXP` instruction.

B2.9.6 Use of WFE and SEV instructions by spin-locks

Armv8 provides Wait For Event, Send Event, and Send Event Local instructions, `WFE`, `SEV`, and `SEVL`, that can assist with reducing power consumption and bus contention caused by PEs repeatedly attempting to obtain a spin-lock. These instructions can be used at the application level, but a complete understanding of what they do depends on a system level understanding of exceptions. They are described in *Wait for Event mechanism and Send event* on page D1-2397. However, in Armv8, when the global monitor for a PE changes from Exclusive Access state to Open Access state, an event is generated.

Note

This is equivalent to issuing an `SEVL` instruction on the PE for which the monitor state has changed. It removes the need for spinlock code to include an `SEV` instruction after clearing a spinlock.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Part C

The AArch64 Instruction Set

Chapter C1
The A64 Instruction Set

This chapter describes the A64 instruction set. It contains the following sections:

- *About the A64 instruction set* on page C1-182.
- *Structure of the A64 assembler language* on page C1-183.
- *Address generation* on page C1-189.
- *Instruction aliases* on page C1-192.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C1.1 About the A64 instruction set

The A64 instruction set is the instruction set supported in the AArch64 Execution state.

All A64 instructions have a width of 32 bits. The A64 encoding structure breaks down into the following functional groups:

- A miscellaneous group of branch instructions, exception generating instructions, and System instructions.
- Data-processing instructions associated with general-purpose registers. These instructions are supported by two *functional groups*, depending on whether the operands:
  - Are all held in registers.
  - Include an operand with a constant immediate value.
- Load and store instructions associated with the general-purpose register file and the SIMD and floating-point register file.
- SIMD and scalar floating-point data-processing instructions that operate on the SIMD and floating-point registers.

The encoding hierarchy within a functional group breaks down as follows:

- A functional group consists of a set of related instruction classes. *A64 instruction set encoding* on page C4-268 provides an overview of the instruction encodings in the form of a list of instruction classes within their functional groups.
- An instruction class consists of a set of related instruction forms. Instruction forms are documented in one of two alphabetic lists:
  - The load, store, and data-processing instructions associated with the general-purpose registers, together with those in the other instruction classes. See Chapter C6 *A64 Base Instruction Descriptions*.
  - The load, store, and data-processing instructions associated with the SIMD and floating-point support. See Chapter C7 *A64 Advanced SIMD and Floating-point Instruction Descriptions*.
- An instruction form might support a single instruction syntax. Where an instruction supports more than one syntax, each syntax is an *instruction variant*. Instruction variants can occur because of differences in:
  - The size or format of the operands.
  - The register file used for the operands.
  - The addressing mode used for load/load/store memory operands.

  Instruction variants might also arise as the result of other factors.

  Instruction variants are described in the instruction description for the individual instructions.

A64 instructions have a regular bit encoding structure:

- 5-bit register operand fields at fixed positions within the instruction. For general-purpose register operands, the values 0-30 select one of 31 registers. The value 31 is used as a special case that can:
  - Indicate use of the current stack pointer, when identifying a load/store base register or in a limited set of data-processing instructions. See *The stack pointer registers* on page D1-2323.
  - Indicate the value zero when used as a source register operand.
  - Indicate discarding the result when used as a destination register operand.

  For SIMD and floating-point register access, the value used selects one of 32 registers.
- Immediate bits that provide constant data-processing values or address offsets are placed in contiguous bitfields. Some computed values in instruction variants use one or more immediate bitfields together with the secondary encoding bitfields.

All encodings that are not fully defined are described as unallocated. An attempt to execute an unallocated instruction is UNDEFINED, unless the behavior is otherwise defined in this Manual.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C1.2 Structure of the A64 assembler language

The following sections describe the A64 assembler syntax:

- *General requirements*.
- *Common syntax terms*.
- *Instruction Mnemonics* on page C1-185.
- *Condition code* on page C1-185.
- *Register names* on page C1-186.

C1.2.1 General requirements

The letter `W` denotes a general-purpose register holding a 32-bit word, and `X` denotes a general-purpose register holding a 64-bit doubleword.

An A64 assembler recognizes both uppercase and lowercase variants of the instruction mnemonics and register names, but not mixed case variants. An A64 disassembler can output either uppercase or lowercase mnemonics and register names. Program and data labels are case-sensitive.

The A64 assembly language does not require the # character to introduce constant immediate operands, but an assembler must allow immediate values introduced with or without the # character. Arm recommends that an A64 disassembler outputs a # before an immediate operand.

In Example C1-1 on page C1-185, the sequence // is used as a comment leader and A64 assemblers are encouraged to accept this syntax.

C1.2.2 Common syntax terms

The following syntax terms are used frequently throughout the A64 instruction set description.

`UPPER` Text in upper-case letters is fixed. Text in lower-case letters is variable. This means that register name `Xn` indicates that the `X` is required, followed by a variable register number, for example `X29`.

`< >` Any text enclosed by angle braces, < >, is a value that the user supplies. Subsequent text might supply additional information.

`{ }` Any item enclosed by curly brackets, { }, is optional. A description of the item and how its presence or absence affects the instruction is normally supplied by subsequent text. In some cases curly braces are actual symbols in the syntax, for example when they surround a register list. These cases are called out in the surrounding text.

`[ ]` Any items enclosed by square brackets, [ ], constitute a list of alternative characters. A single one of the characters can be used in that position and the subsequent text describes the meaning of the alternatives. In some case the square brackets are part of the syntax itself, such as addressing modes or vector elements. These cases are called out in the surrounding text.

`a|b` Alternative words are separated by a vertical bar, |, and can be surrounded by parentheses to delimit them. For example, `U(ADD|SUB)W` represents `UADDW` or `USUBW`.

`±` This indicates an optional + or - sign. If neither is used then + is assumed.

`uimm`*n* An *n*-bit unsigned, positive, immediate value.

`simm`*n* An *n*-bit two's complement, signed immediate value, where *n* includes the sign bit.

`SP` See *Register names* on page C1-186.

`Wn` See *Register names* on page C1-186.

`WSP` See *Register names* on page C1-186.

`WZR` See *Register names* on page C1-186.

`Xn` See *Register names* on page C1-186.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`XZR` See *Register names* on page C1-186

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C1.2.3 Instruction Mnemonics

The A64 assembly language overloads instruction mnemonics and distinguishes between the different forms of an instruction based on the operand types. For example, the following `ADD` instructions all have different opcodes. However, the programmer must only remember one mnemonic, as the assembler automatically chooses the correct opcode based on the operands. The disassembler follows the same procedure in reverse.

Example C1-1 ADD instructions with different opcodes

```
ADD W0, W1, W2              // add 32-bit register
ADD X0, X1, X2              // add 64-bit register
ADD X0, X1, W2, SXTW        // add 64-bit extended register
ADD X0, X1, #42             // add 64-bit immediate
```

C1.2.4 Condition code

The A64 ISA has some instructions that set Condition flags or test Condition codes or both. For information about instructions that set the Condition flags or use the condition mnemonics, see *Condition flags and related instructions* on page C6-773.

Table C1-1 shows the available Condition codes.

Table C1-1 Condition codes

cond	Mnemonic	Meaning (integer)	Meaning (floating-point)[a]	Condition flags
0000	EQ	Equal	Equal	Z == 1
0001	NE	Not equal	Not equal or unordered	Z == 0
0010	CS or HS	Carry set	Greater than, equal, or unordered	C == 1
0011	CC or LO	Carry clear	Less than	C == 0
0100	MI	Minus, negative	Less than	N == 1
0101	PL	Plus, positive or zero	Greater than, equal, or unordered	N == 0
0110	VS	Overflow	Unordered	V == 1
0111	VC	No overflow	Ordered	V == 0
1000	HI	Unsigned higher	Greater than, or unordered	C ==1 && Z == 0
1001	LS	Unsigned lower or same	Less than or equal	!(C ==1 && Z ==0)
1010	GE	Signed greater than or equal	Greater than or equal	N == V
1011	LT	Signed less than	Less than, or unordered	N! = V
1100	GT	Signed greater than	Greater than	Z == 0 && N == V
1101	LE	Signed less than or equal	Less than, equal, or unordered	!(Z == 0 && N == V)
1110	AL	Always	Always	Any
1111	NV[b]	Always	Always	Any

a. Unordered means at least one NaN operand.

b. The Condition code `NV` exists only to provide a valid disassembly of the `0b1111` encoding, otherwise its behavior is identical to `AL`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C1.2.5 Register names

This section describes the AArch64 registers. It contains the following subsections:

- *General-purpose register file and zero register and stack pointer*.
- *SIMD and floating-point register file* on page C1-187.
- *SIMD and floating-point scalar register names* on page C1-187.
- *SIMD vector register names* on page C1-187.
- *SIMD vector element names* on page C1-188.

General-purpose register file and zero register and stack pointer

The 31 general-purpose registers in the general-purpose register file are named R0-R30 and encoded in the instruction register fields with values 0-30. In a general-purpose register field the value 31 represents either the current stack pointer or the zero register, depending on the instruction and the operand position.

When the registers are used in a specific instruction variant, they must be qualified to indicate the operand data size, 32 bits or 64 bits, and the data size of the instruction.

When the data size is 32 bits, the lower 32 bits of the register are used and the upper 32 bits are ignored on a read and cleared to zero on a write.

Table C1-2 shows the qualified names for registers, where *n* is a register number 0-30.

Table C1-2 Naming of general-purpose registers, the zero register, and the stack pointer

Name	Size	Encoding	Description
Wn	32 bits	0-30	General-purpose register 0-30
Xn	64 bits	0-30	General-purpose register 0-30
WZR	32 bits	31	Zero register
XZR	64 bits	31	Zero register
WSP	32 bits	31	Current stack pointer
SP	64 bits	31	Current stack pointer

This list gives more information about the instruction arguments shown in Table C1-2:

- The names Xn and Wn both refer to the same general-purpose register, Rn.
- There is no register named W31 or X31.
- The name SP represents the stack pointer for 64-bit operands where an encoding of the value 31 in the corresponding register field is interpreted as a read or write of the current stack pointer. When instructions do not interpret this operand encoding as the stack pointer, use of the name SP is an error.
- The name WSP represents the current stack pointer in a 32-bit context.
- The name XZR represents the zero register for 64-bit operands where an encoding of the value 31 in the corresponding register field is interpreted as returning zero when read or discarding the result when written. When instructions do not interpret this operand encoding as the zero register, use of the name XZR is an error.
- The name WZR represents the zero register in a 32-bit context.
- The architecture does not define a specific name for general-purpose register R30 to reflect its role as the link register on procedure calls. However, an A64 assembler must always use W30 and X30 for this purpose, and additional software names might be defined as part of the Procedure Call Standard, see *Procedure Call Standard for the Arm 64-bit Architecture*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

SIMD and floating-point register file

The 32 registers in the SIMD and floating-point register file, V0-V31, hold floating-point operands for the scalar floating-point instructions, and both scalar and vector operands for the SIMD instructions. When they are used in a specific instruction form, the names must be further qualified to indicate the data shape, that is the data element size and the number of elements or lanes within the register. A similar requirement is placed on the general-purpose registers. See *General-purpose register file and zero register and stack pointer* on page C1-186.

Note

The data type is described by the instruction mnemonics that operate on the data. The data type is not described by the register name. The data type is the interpretation of bits within each register or vector element, whether these are integers, floating-point values, polynomials, or cryptographic hashes.

SIMD and floating-point scalar register names

SIMD and floating-point instructions that operate on scalar data only access the lower bits of a SIMD and floating-point register. The unused high bits are ignored on a read and cleared to 0 on a write.

Table C1-3 shows the qualified names for accessing scalar SIMD and floating-point registers. The letter *n* denotes a register number between 0 and 31.

Table C1-3 SIMD and floating-point scalar register names

Size	Name
8 bits	`Bn`
16 bits	`Hn`
32 bits	`Sn`
64 bits	`Dn`
128 bits	`Qn`

SIMD vector register names

If a register holds multiple data elements on which arithmetic is performed in a parallel, SIMD, manner, then a qualifier describes the vector shape. The vector shape is the element size and the number of elements or lanes. If the element size in bits multiplied by the number of lanes does not equal 128, then the upper 64 bits of the register are ignored on a read and cleared to zero on a write.

Table C1-4 shows the SIMD vector register names. The letter *n* denotes a register number between 0 and 31.

Table C1-4 SIMD vector register names

Shape	Name
8 bits × 8 lanes	`Vn.8B`
8 bits × 16 lanes	`Vn.16B`
16 bits × 4 lanes	`Vn.4H`
16 bits × 8 lanes	`Vn.8H`
32 bits × 2 lanes	`Vn.2S`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C1-4 SIMD vector register names (continued)

Shape	Name
32 bits × 4 lanes	`Vn.4S`
64 bits × 1 lane	`Vn.1D`
64 bits × 2 lanes	`Vn.2D`

SIMD vector element names

Appending a constant, zero-based element index to the register name inside square brackets indicates that a single element from a SIMD and floating-point register is used as a scalar operand. The number of lanes is not represented, as it is not encoded in the instruction and can only be inferred from the index value.

Table C1-5 shows the vector register names and the element index. The letter *i* denotes the element index.

Table C1-5 Vector register names with element index

Size	Name
8 bits	`Vn.B[i]`
16 bits	`Vn.H[i]`
32 bits	`Vn.S[i]`
64 bits	`Vn.D[i]`

An assembler must accept a fully qualified SIMD register name if the number of lanes is greater than the index value. See *SIMD vector register names* on page C1-187. For example, an assembler must accept all of the following forms as the name for the 32-bit element in bits [63:32] of the SIMD and floating-point register V9:

```
V9.S[1]     //standard disassembly
V9.2S[1]    //optional number of lanes
V9.4S[1]    //optional number of lanes
```

Note

The SIMD and floating-point register element name `Vn.S[0]` is not equivalent to the scalar SIMD and floating-point register name `Sn`. Although they represent the same bits in the register, they select different instruction encoding forms, either the vector element or the scalar form.

SIMD vector register list

Where an instruction operates on multiple SIMD and floating-point registers, for example vector Load/Store structure and table lookup operations, the registers are specified as a list enclosed by curly braces. This list consists of either a sequence of registers separated by commas, or a register range separated by a hyphen. The registers must be numbered in increasing order, modulo 32, in increments of one. The hyphenated form is preferred for disassembly if there are more than two registers in the list and the register number are increasing. The following examples are equivalent representations of a set of four registers `V4` to `V7`, each holding four lanes of 32-bit elements:

```
{ V4.4S - V7.4S }              //standard disassembly
{ V4.4S, V5.4S, V6.4S, V7.4S } //alternative representation
```

SIMD vector element list

Registers in a list can also have a vector element form. For example, the `LD4` instruction can load one element into each of four registers, and in this case the index is appended to the list as follows:

```
{ V4.S - V7.S }[3]                //standard disassembly
{ V4.4S, V5.4S, V6.4S, V7.4S }[3]  //alternative with optional number of lanes
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C1.3 Address generation

The A64 instruction set supports 64-bit virtual addresses (VAs). The valid VA range is determined by the following factors:

- The size of the implemented virtual address space.
- *Memory Management Unit* (MMU) configuration settings.

Limits on the VA size mean that the most significant bits of the virtual address do not hold valid address bits. These unused bits can hold:

- A tag, see *Address tagging in AArch64 state* on page D5-2534.
- If ARMv8.3-PAuth is implemented, a Pointer authentication code (PAC), see *Pointer authentication in AArch64 state* on page D5-2536.

For more information on memory management and address translation, see Chapter D5 *The AArch64 Virtual Memory System Architecture*.

C1.3.1 Register indexed addressing

The A64 instruction set allows a 64-bit index register to be added to the 64-bit base register, with optional scaling of the index by the access size. Additionally it allows for sign-extension or zero-extension of a 32-bit value within an index register, followed by optional scaling.

C1.3.2 PC-relative addressing

The A64 instruction set has support for position-independent code and data addressing:

- PC-relative literal loads have an offset range of ± 1MB.
- Process state flag and compare based conditional branches have a range of ± 1MB. Test bit conditional branches have a restricted range of ± 32KB.
- Unconditional branches, including branch and link, have a range of ± 128MB.

PC-relative Load/Store operations, and address generation with a range of ± 4GB can be performed using two instructions.

C1.3.3 Load/Store addressing modes

Load/Store addressing modes in the A64 instruction set require a 64-bit base address from a general-purpose register X0-X30 or the current stack pointer, `SP`, with an optional immediate or register offset. Table C1-6 shows the assembler syntax for the complete set of Load/Store addressing modes.

Table C1-6 A64 Load/Store addressing modes

Addressing Mode	Offset		
	Immediate	Register	Extended Register
Base register only (no offset)	`[base{, #0}]`	-	-
Base plus offset	`[base{, #imm}]`	`[base, Xm{, LSL #imm}]`	`[base, Wm, (S|U)XTW {#imm}]`
Pre-indexed	`[base, #imm]!`	-	-
Post-indexed	`[base], #imm`	`[base], Xm`[a]	-
Literal (PC-relative)	`label`	-	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

a. The post-indexed by register offset mode can be used with the SIMD Load/Store structure instructions described in *Load/Store Vector* on page C3-218. Otherwise the post-indexed by register offset mode is not available.

Some types of Load/Store instruction support only a subset of the Load/Store addressing modes listed in Table C1-6 on page C1-189. Details of the supported modes are as follows:

- Base plus offset addressing means that the address is the value in the 64-bit base register plus an offset.
- Pre-indexed addressing means that the address is the sum of the value in the 64-bit base register and an offset, and the address is then written back to the base register.
- Post-indexed addressing means that the address is the value in the 64-bit base register, and the sum of the address and the offset is then written back to the base register.
- Literal addressing means that the address is the value of the 64-bit program counter for this instruction plus a 19-bit signed word offset. This means that it is a 4 byte aligned address within ±1MB of the address of this instruction with no offset. Literal addressing can only be used for loads of at least 32 bits and for prefetch instructions. The PC cannot be referenced using any other addressing modes. The syntax for labels is specific to individual toolchains.
- An immediate offset can be unsigned or signed, and scaled or unscaled, depending on the type of Load/Store instruction. When the immediate offset is scaled it is encoded as a multiple of the transfer size, although the assembly language always uses a byte offset, and the assembler or disassembler performs the necessary conversion. The usable byte offsets therefore depend on the type of Load/Store instruction and the transfer size.

  Table C1-7 shows the offset and the type of Load/Store instruction.

Table C1-7 Immediate offsets and the type of Load/Store instruction

Offset bits	Sign	Scaling	Write-Back	Load/Store type
0	-	-	-	Exclusive/acquire/release
7	Signed	Scaled	Optional	Register pair
9	Signed	Unscaled	Optional	Single register
12	Unsigned	Scaled	No	Single register

- A register offset means that the offset is the 64 bits from a general-purpose register, Xm, optionally scaled by the transfer size, in bytes, if `LSL #imm` is present and where `imm` must be equal to $\log_2$(transfer_size).
- An extended register offset means that offset is the bottom 32 bits from a general-purpose register Wm, sign-extended or zero-extended to 64 bits, and then scaled by the transfer size if so indicated by `#imm`, where imm must be equal to $\log_2$(transfer_size). An assembler must accept Wm or Xm as an extended register offset, but Wm is preferred for disassembly.
- Generating an address lower than the value in the base register requires a negative signed immediate offset or a register offset holding a negative value.
- When stack alignment checking is enabled by system software and the base register is the SP, the current stack pointer must be initially quadword aligned, that is aligned to 16 bytes. Misalignment generates a Stack Alignment fault. The offset does not have to be a multiple of 16 bytes unless the specific Load/Store instruction requires this. SP cannot be used as a register offset.

Address calculation

General-purpose arithmetic instructions can calculate the result of most addressing modes and write the address to a general-purpose register or, in most cases, to the current stack pointer.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C1-8 shows the arithmetic instructions that can compute addressing modes.

Table C1-8 Arithmetic instructions to compute addressing modes

Addressing Form	Offset		
	Immediate	Register	Extended Register
Base register (no offset)	`MOV Xd|SP, base`	-	-

Base plus offset	`ADD Xd|SP, base, #imm`
	or
	`SUB Xd|SP, base, #imm`
		`ADD <Xd|SP>, base, Xm{,LSL#imm}`	`ADD <Xd|SP>, base, Wm,(S|U)XT(W|H|B|) {#imm}`

Pre-indexed	-	-	-
Post-indexed	-	-	-
Literal (PC-relative)	`ADR Xd, label`	-	-

Note

- To calculate a base plus immediate offset the `ADD` instructions defined in *Arithmetic (immediate)* on page C3-226 accept an unsigned 12-bit immediate offset, with an optional left shift by 12. This means that a single `ADD` instruction cannot support the full range of byte offsets available to a single register Load/Store with a scaled 12-bit immediate offset. For example, a quadword `LDR` effectively has a 16-bit byte offset. To calculate an address with a byte offset that requires more than 12 bits it is necessary to use two `ADD` instructions. The following example shows this:

```
ADD  Xd, base, #(imm & 0xFFF)
ADD  Xd, Xd, #(imm>>12), LSL #12
```

- To calculate a base plus extended register offset, the `ADD` instructions defined in *Arithmetic (extended register)* on page C3-232 provide a superset of the addressing mode that also supports sign-extension or zero-extension of a byte or halfword value with any shift amount between 0 and 4, for example:

```
ADD   Xd, base, Wm, SXTW #3     // Xd = base + (SignExtend(Wm) LSL 3)
ADD   Xd, base, Wm, UXTH #4     // Xd = base + (ZeroExtend(Wm<15:0>) LSL 4)
```

- If the same extended register offset is used by more than one Load/Store instruction, then, depending on the implementation, it might be more efficient to calculate the extended and scaled intermediate result just once, and then reuse it as a simple register offset. The extend and scale calculation can be performed using the `SBFIZ` and `UBFIZ` bitfield instructions defined in *Bitfield move* on page C3-228, for example:

```
SBFIZ Xd, Xm, #3, #32     //Xd = "Wm, SXTW #3"
UBFIZ Xd, Xm, #4, #16     //Xd = "Wm, UXTH #4"
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C1.4 Instruction aliases

Some instructions have an associated *architecture alias* that is used for disassembly of the encoding when the associated conditions are met. Architecture alias instructions are included in the alphabetic lists of instruction types and clearly presented as an alias form in descriptions for the individual instructions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter C2
About the A64 Instruction Descriptions

This chapter describes the *instruction descriptions* contained in Chapter C6 *A64 Base Instruction Descriptions* and Chapter C7 *A64 Advanced SIMD and Floating-point Instruction Descriptions*.

It contains the following sections:

- *Understanding the A64 instruction descriptions* on page C2-194.
- *General information about the A64 instruction descriptions* on page C2-197.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C2.1 Understanding the A64 instruction descriptions

Each instruction description in Chapter C6 and Chapter C7 has the following content:

1. A title.
2. An introduction to the instruction.
3. The instruction encoding or encodings.
4. Any alias conditions.
5. A list of the assembler symbols for the instruction.
6. Pseudocode describing how the instruction operates.
7. Notes, if applicable.

The following sections describe each of these.

C2.1.1 The title

The title of an instruction description includes the base mnemonic for the instruction.

If different forms of an instruction use the same base mnemonic, each form has its own description. In this case, the title is the mnemonic followed by a short description of the instruction form in parentheses. This is most often used when an operand is an immediate value in one instruction form, but is a register in another form.

For example, in Chapter C6 there are the following titles for different forms of the `ADD` instruction:

- *ADD (extended register)* on page C6-780.
- *ADD (immediate)* on page C6-783.
- *ADD (shifted register)* on page C6-785.

C2.1.2 An introduction to the instruction

This briefly describes the function of the instruction. The introduction is not a complete description of the instruction, and it is not definitive. If there is any conflict between it and the more detailed information that follows it, the more detailed information takes priority.

C2.1.3 The instruction encoding or encodings

This shows the instruction encoding diagram, or if the instruction has more than one encoding, shows all of the encoding diagrams. Each diagram has a subheading.

For example, for load and store instructions, the subheadings might be:

- Post-index.
- Pre-index.
- Unsigned offset.

Each diagram numbers the bits from 31 to 0. The diagram for an instruction at address *A* shows, from left to right, the bytes at addresses *A*+3, *A*+2, *A*+1, and *A*.

There might be variants of an encoding, if the *assembler syntax prototype* differs depending on the value in one or more of the encoding fields. In this case, each variant has a subheading that describes the variant and shows the distinguishing field value or values in parentheses. For example, in Chapter C6 there are the following subheadings for variants of the `ADC` instruction encoding:

- 32-bit variant (sf = 0).
- 64-bit variant (sf = 1).

The assembler syntax prototype for an encoding or variant of an encoding shows how to form a complete assembler source code instruction that assembles to the encoding. Unless otherwise stated, the prototype is also the preferred syntax for a disassembler to disassemble the encoding to. Disassemblers are permitted to omit optional symbols that represent the default value of a field or set of fields, to produce more readable disassembled code, provided that the output re-assembles to the same encoding.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Each encoding diagram, and its associated assembler syntax prototypes, is followed by encoding-specific pseudocode that translates the fields of that encoding into inputs for the encoding-independent pseudocode that describes the operation of the instruction. See *Pseudocode describing how the instruction operates* on page C2-196.

C2.1.4 Any alias conditions, if applicable

This is an optional part of an instruction description. If included, it describes the set of conditions for which an alternative mnemonic and its associated assembler syntax prototypes are preferred for disassembly by a disassembler. It includes a link to the alias instruction description that defines the alternative syntax. The alias syntax and the original syntax can be used interchangeably in the assembler source code.

Arm recommends that if a disassembler outputs the alias syntax, it consistently outputs the alias syntax.

C2.1.5 A list of the assembler symbols for the instruction

The *Assembler symbols* subsection of the instruction description contains a list of the symbols that the assembler syntax prototype or prototypes use, if any.

In assembler syntax prototypes, the following conventions are used:

`< >` Angle brackets. Any symbol enclosed by these is a name or a value that the user supplies. For each symbol, there is a description of what the symbol represents. The description usually also specifies which encoding field or fields encodes the symbol.

`{ }` Brace brackets. Any symbols enclosed by these are optional. For each optional symbol, there is a description of what the symbol represents and how its presence or absence is encoded.

In some assembler syntax prototypes, some brace brackets are mandatory, for example if they surround a register list. When the use of brace brackets is mandatory, they are separated from other syntax items by one or more spaces.

`#` This usually precedes a numeric constant. All uses of # are optional in A64 assembler source code. Arm recommends that disassemblers output the # where the assembler syntax prototype includes it.

`+/-` This indicates an optional + or - sign. If neither is coded, + is assumed.

Single spaces are used for clarity, to separate syntax items. Where a space is mandatory, the assembler syntax prototype shows two or more consecutive spaces.

Any characters not shown in this conventions list must be coded exactly as shown in the assembler syntax prototype. Apart from brace brackets, the characters shown are used as part of a meta-language to define the architectural assembler syntax for an instruction encoding or alias, but have no architecturally defined significance in the input to an assembler or in the output from a disassembler.

The following symbol conventions are used:

`<Xn>` The 64-bit name of a general-purpose register (X0-X30) or the zero register (XZR).

`<Wn>` The 32-bit name of a general-purpose register (W0-W30) or the zero register (WZR).

`<Xn|SP>` The 64-bit name of a general-purpose register (X0-X30) or the current stack pointer (SP).

`<Wn|WSP>` The 32-bit name of a general-purpose register (W0-W30) or the current stack pointer (WSP).

`<Bn>, <Hn>, <Sn>, <Dn>, <Qn>`
The 8, 16, 32, 64 or 128-bit name of a SIMD and floating-point register in a scalar context as described in section *Register names* on page C1-186.

`<Vn>` The name of a SIMD and floating-point register name in a vector context as described in *Register names* on page C1-186.

If the description of a symbol specifies that the symbol is a register, the description might also specify that the range of permitted registers is extended or restricted. It also specifies any differences from the default rules for such fields.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

Register names on page C1-186 provides the A64 register names.

C2.1.6 Pseudocode describing how the instruction operates

The *Operation* subsection of the instruction description contains this pseudocode.

It is encoding-independent pseudocode that provides a precise description of what the instruction does.

Note

For a description of Arm pseudocode, see Appendix K13 *Arm Pseudocode Definition*. This appendix also describes the execution model for an instruction.

C2.1.7 Notes, if applicable

If applicable, other notes about the instruction appear under additional subheadings.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C2.2 General information about the A64 instruction descriptions

This section provides general information about the A64 instruction descriptions. Some of this information also applies to System register descriptions, for example the terms defined in *Fixed values in AArch64 instruction and System register descriptions* apply to the AArch64 descriptions throughout this manual. The following subsections provide this information:

- *Execution of instructions in debug state*.
- *Fixed values in AArch64 instruction and System register descriptions*.
- *Modified immediate constants in A64 instructions* on page C2-198.

C2.2.1 Execution of instructions in debug state

In general, except for the instructions described in *Debug state* on page C3-204, the A64 instruction descriptions do not indicate any differences in the behavior of the instruction if it is executed in Debug state. For this information, see *Executing instructions in Debug state* on page H2-6937.

Note

For many instructions, execution is unchanged in Debug state. *Executing instructions in Debug state* on page H2-6937 identifies these instructions,

C2.2.2 Fixed values in AArch64 instruction and System register descriptions

This section summarizes the terms used to describe fixed values in AArch64 register and instruction descriptions. The Glossary gives full descriptions of these terms, and each entry in this section includes a link to the corresponding Glossary entry.

Note

In register descriptions, the meaning of some bits depends on the PE state. This affects the definitions of RES0 and RES1, as shown in the Glossary.

The following terms are used to describe bits or fields with fixed values:

RAZ Read-As-Zero. See *Read-As-Zero (RAZ)*.

In diagrams, a RAZ bit can be shown as 0.

(0), RES0 Reserved, *Should-Be-Zero (SBZ)* or *RES0*.

In instruction encoding diagrams, and sometimes in other descriptions, `(0)` indicates an SBZ bit. If the bit is set to 1, behavior is CONSTRAINED UNPREDICTABLE, and must be one of the following:

- The instruction is UNDEFINED.
- The instruction is treated as a `NOP`.
- The instruction executes as if the value of the bit was 0.
- Any destination registers of the instruction become UNKNOWN.

This notation can be expanded for fields, so a three-bit field can be shown as either `(0)(0)(0)` or as `(000)`.

In register diagrams, but not in the A64 encoding and instruction descriptions, bits or fields can be shown as RES0. See the Glossary definition of RES0 for more information.

Note

Some of the System instruction descriptions in this chapter are based on the *field description* of the input value for the instruction. These are register descriptions and therefore can include RES0 fields,

The `(0)` and RES0 descriptions can be applied to bits or bitfields that are read-only, or are write-only. The Glossary definitions cover these cases.

RAO Read-As-One. See *Read-As-One (RAO)*.

In diagrams, a RAO bit can be shown as 1.

(1), RES1 Reserved, *Should-Be-One (SBO)* or *RES1*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In instruction encoding diagrams, and sometimes in other descriptions, `(1)` indicates a SBO bit. If the bit is set to 0, behavior is CONSTRAINED UNPREDICTABLE, and must be one of the following:

- The instruction is UNDEFINED.
- The instruction is treated as a `NOP`.
- The instruction executes as if the value of the bit was 1.
- Any destination registers of the instruction become UNKNOWN.

This notation can be expanded for fields, so a three-bit field can be shown as either `(1)(1)(1)` or as `(111)`.

In register diagrams, but not in the A64 encoding and instruction descriptions, bits or fields can be shown as RES1. See the Glossary definition of RES1 for more information.

Note

Some of the System instruction descriptions in this chapter are based on the *field description* of the input value for the instruction. These are register descriptions and therefore can include RES1 fields,

The `(1)` and RES1 descriptions can be applied to bits or bitfields that are read-only, or are write-only. The Glossary definitions cover these cases.

C2.2.3 Modified immediate constants in A64 instructions

It contains the following subsections:

- *Modified immediate constants in A64 floating-point instructions*.

Modified immediate constants in A64 floating-point instructions

Table C2-1 shows the immediate constants available in `FMOV` (scalar, immediate) and `FMOV` (vector, immediate) floating-point instructions.

Table C2-1 A64 Floating-point modified immediate constants

Data type	immediate	Constant [a]
`F16`	abcdefgh	`aBbbcdef gh000000`
`F32`	abcdefgh	`aBbbbbbc defgh000 00000000 00000000`
`F64`	abcdefgh	`aBbbbbbb bbcdefgh 00000000 00000000 00000000 00000000 00000000 00000000`

a. In this column, `B` = NOT(`b`). The bit pattern represents the floating-point number $(-1)^{S} \times 2^{exp} \times$ mantissa, where S = `UInt(a)`, exp = `UInt(NOT(b):c:d)-3` and mantissa = `(16+UInt(e:f:g:h))/16`.

The immediate value shown in the table is either:

- The value of the `imm8` field for an `FMOV` (scalar, immediate) instruction, see *FMOV (scalar, immediate)* on page C7-1691.
- The value obtained by concatenating the `a:b:c:d:e:f:g:h` fields field for an `FMOV` (vector, immediate) instruction, see *FMOV (vector, immediate)* on page C7-1684.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C2-2 shows the floating-point constant values encoded in the `b:c:d:e:f:g:h` fields of the `FMOV` (vector, immediate) instruction.

Table C2-2 Floating-point constant values

efgh	bcd							
	000	001	010	011	100	101	110	111
0000	2.0	4.0	8.0	16.0	0.125	0.25	0.5	1.0
0001	2.125	4.25	8.5	17.0	0.1328125	0.265625	0.53125	1.0625
0010	2.25	4.5	9.0	18.0	0.140625	0.28125	0.5625	1.125
0011	2.375	4.75	9.5	19.0	0.1484375	0.296875	0.59375	1.1875
0100	2.5	5.0	10.0	20.0	0.15625	0.3125	0.625	1.25
0101	2.625	5.25	10.5	21.0	0.1640625	0.328125	0.65625	1.3125
0110	2.75	5.5	11.0	22.0	0.171875	0.34375	0.6875	1.375
0111	2.875	5.75	11.5	23.0	0.1796875	0.359375	0.71875	1.4375
1000	3.0	6.0	12.0	24.0	0.1875	0.375	0.75	1.5
1001	3.125	6.25	12.5	25.0	0.1953125	0.390625	0.78125	1.5625
1010	3.25	6.5	13.0	26.0	0.203125	0.40625	0.8125	1.625
1011	3.375	6.75	13.5	27.0	0.2109375	0.421875	0.84375	1.6875
1100	3.5	7.0	14.0	28.0	0.21875	0.4375	0.875	1.75
1101	3.625	7.25	14.5	29.0	0.2265625	0.453125	0.90625	1.8125
1110	3.75	7.5	15.0	30.0	0.234375	0.46875	0.9375	1.875
1111	3.875	7.75	15.5	31.0	0.2421875	0.484375	0.96875	1.9375

Operation of modified immediate constants, floating-point instructions

For an A64 floating-point instruction that uses a modified immediate constant, the operation described by the `VFPExpandImm()` pseudocode function returns the value of the immediate constant.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter C3
A64 Instruction Set Overview

This chapter provides an overview of the A64 instruction set. It contains the following sections:

- *Branches, Exception generating, and System instructions* on page C3-202.
- *Loads and stores* on page C3-209.
- *Data processing - immediate* on page C3-226.
- *Data processing - register* on page C3-231.
- *Data processing - SIMD and floating-point* on page C3-239.

For a structured breakdown of instruction groups by encoding, see Chapter C4 *A64 Instruction Set Encoding*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C3.1 Branches, Exception generating, and System instructions

This section describes the branch, exception generating, and System instructions. It contains the following subsections:

- *Conditional branch*.
- *Unconditional branch (immediate)*.
- *Unconditional branch (register)* on page C3-203.
- *Exception generation and return* on page C3-203.
- *System register instructions* on page C3-204.
- *System instructions* on page C3-204.
- *Hint instructions* on page C3-205.
- *Barriers and CLREX instructions* on page C3-205.
- *Pointer authentication instructions* on page C3-206.

For information about the encoding structure of the instructions in this instruction group, see *Branches, Exception Generating and System instructions* on page C4-273.

Note

Software must:

- Use only `BLR` or `BL` to perform a nested subroutine call when that subroutine is expected to return to the immediately following instruction, that is, the instruction with the address of the `BLR` or `BL` instruction incremented by four.
- Use only `RET` to perform a subroutine return, when that subroutine is expected to have been entered by a `BL` or `BLR` instruction.
- Use only `B`, `BR`, or the instructions listed in Table C3-1 to perform a control transfer that is not a subroutine call or subroutine return described in this *Note*.

C3.1.1 Conditional branch

Conditional branches change the flow of execution depending on the current state of the Condition flags or the value in a general-purpose register. See Table C1-1 on page C1-185 for a list of the Condition codes that can be used for `cond`.

Table C3-1 shows the Conditional branch instructions.

Table C3-1 Conditional branch instructions

Mnemonic	Instruction	Branch offset range from the PC	See
`B.cond`	Branch conditionally	±1MB	*B.cond* on page C6-820
`CBNZ`	Compare and branch if nonzero	±1MB	*CBNZ* on page C6-853
`CBZ`	Compare and branch if zero	±1MB	*CBZ* on page C6-854
`TBNZ`	Test bit and branch if nonzero	±32KB	*TBNZ* on page C6-1365
`TBZ`	Test bit and branch if zero	±32KB	*TBZ* on page C6-1366

C3.1.2 Unconditional branch (immediate)

Unconditional branch (immediate) instructions change the flow of execution unconditionally by adding an immediate offset with a range of ±128MB to the value of the program counter that fetched the instruction. The `BL` instruction also writes the address of the sequentially following instruction to general-purpose register, X30.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-2 shows the Unconditional branch instructions with an immediate branch offset.

Table C3-2 Unconditional branch instructions (immediate)

Mnemonic	Instruction	Immediate branch offset range from the PC	See
B	Branch unconditionally	±128MB	*B* on page C6-821
BL	Branch with link	±128MB	*BL* on page C6-834

C3.1.3 Unconditional branch (register)

Unconditional branch (register) instructions change the flow of execution unconditionally by setting the program counter to the value in a general-purpose register. The BLR instruction also writes the address of the sequentially following instruction to general-purpose register X30. The RET instruction behaves identically to BR, but provides an additional hint to the PE that this is a return from a subroutine. Table C3-3 shows Unconditional branch instructions that jump directly to an address held in a general-purpose register.

Table C3-3 Unconditional branch instructions (register)

Mnemonic	Instruction	See
BLR	Branch with link to register	*BLR* on page C6-835
BR	Branch to register	*BR* on page C6-838
RET	Return from subroutine	*RET* on page C6-1172

C3.1.4 Exception generation and return

This section describes the following exceptions:

- *Exception generating*.
- *Exception return* on page C3-204.
- *Debug state* on page C3-204.

Exception generating

Table C3-4 shows the Exception generating instructions.

Table C3-4 Exception generating instructions

Mnemonic	Instruction	See
BRK	Breakpoint Instruction	*BRK* on page C6-841
HLT	Halt Instruction	*HLT* on page C6-932
HVC	Generate exception targeting Exception level 2	*HVC* on page C6-933
SMC	Generate exception targeting Exception level 3	*SMC* on page C6-1205
SVC	Generate exception targeting Exception level 1	*SVC* on page C6-1350

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Exception return

Table C3-5 shows the Exception return instructions.

Table C3-5 Exception return instructions

Mnemonic	Instruction	See
ERET	Exception return using current ELR and SPSR	*ERET* on page C6-923

Debug state

Table C3-6 shows the Debug state instructions.

Table C3-6 Debug state instructions

Mnemonic	Instruction	See
DCPS1	Debug switch to Exception level 1	*DCPS1* on page C6-907
DCPS2	Debug switch to Exception level 2	*DCPS2* on page C6-908
DCPS3	Debug switch to Exception level 3	*DCPS3* on page C6-909
DRPS	Debug restore PE state	*DRPS* on page C6-913

C3.1.5 System register instructions

For detailed information about the System register instructions, see Chapter C5 *The A64 System Instruction Class*. Table C3-7 shows the System register instructions.

Table C3-7 System register instructions

Mnemonic	Instruction	See
MRS	Move System register to general-purpose register	*MRS* on page C6-1128
MSR	Move general-purpose register to System register	*MSR (register)* on page C6-1132
	Move immediate to PE state field	*MSR (immediate)* on page C6-1129

C3.1.6 System instructions

For detailed information about the System instructions, see Chapter C5 *The A64 System Instruction Class*.

Table C3-8 shows the System instructions.

Table C3-8 System instructions

Mnemonic	Instruction	See
SYS	System instruction	*SYS* on page C6-1362
SYSL	System instruction with result	*SYSL* on page C6-1364
IC	Instruction cache maintenance	*IC* on page C6-934 and Table C5-1 on page C5-381

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-8 System instructions (continued)

Mnemonic	Instruction	See
DC	Data cache maintenance	*DC* on page C6-905 and Table C5-1 on page C5-381
AT	Address translation	*AT* on page C6-811 and Table C5-3 on page C5-383
TLBI	TLB Invalidate	*TLBI* on page C6-1367 and Table C5-4 on page C5-384

C3.1.7 Hint instructions

Table C3-9 shows the Hint instructions.

Table C3-9 Hint instructions

Mnemonic	Instruction	See
NOP	No operation	*NOP* on page C6-1146
YIELD	Yield hint	*YIELD* on page C6-1395
WFE	Wait for event	*WFE* on page C6-1391
WFI	Wait for interrupt	*WFI* on page C6-1392
SEV	Send event	*SEV* on page C6-1201
SEVL	Send event local	*SEVL* on page C6-1202
HINT	Unallocated hint	*HINT* on page C6-929
DGH	Data Gathering Hint	*DGH* on page C6-910

C3.1.8 Barriers and CLREX instructions

Table C3-10 shows the barrier and CLREX instructions.

Table C3-10 Barriers and CLREX instructions

Mnemonic	Instruction	See
CLREX	Clear Exclusives monitor	*CLREX* on page C6-869
DMB	Data memory barrier	*DMB* on page C6-911
DSB	Data synchronization barrier	*DSB* on page C6-914
ISB	Instruction synchronization barrier	*ISB* on page C6-936

For more information about DSB, DMB, and ISB, see *Memory barriers* on page B2-135.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-11 shows the speculation and synchronization barriers. If these instructions are not implemented, then these instructions execute as a NOP.

Table C3-11 Speculation and synchronization barriers

Mnemonic	Instruction	See
CSDB	Consumption of Speculative Data Barrier	*CSDB* on page C6-892
ESB	Error synchronization barrier	*ESB* on page C6-925
PSB CSYNC	Profiling synchronization barrier	*PSB CSYNC* on page C6-1169
PSSBB	Physical Speculative Store Bypass Barrier	*PSSBB* on page C6-1170
SB	Speculation Barrier	*SB* on page C6-1188
SSBB	Speculative Store Bypass Barrier	*SSBB* on page C6-1211
TSB CSYNC	Trace Synchronization Barrier	*TSB CSYNC* on page C6-1370

For more information about:

- CSDB, PSSBB, SB, SSBB, TSB CSYNC, see *Memory barriers* on page B2-135.
- ESB, see the *ARM® Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for the ARMv8-A architecture profile*.
- PSB CSYNC, see Chapter D9 *The Statistical Profiling Extension*.

C3.1.9 Pointer authentication instructions

ARMv8.3-PAuth adds support for pointer authentication, see *Pointer authentication in AArch64 state* on page D5-2536. This functionality includes the A64 instructions described in this section. These instructions fall into two groups, see:

- *Basic pointer authentication instructions*.
- *Combined instructions that include pointer authentication* on page C3-208.

Basic pointer authentication instructions

Each of these instructions only performs an operation that supports pointer authentication.

Table C3-12 shows the instructions that add a *Pointer Authentication Code* (PAC) to the address in a register:

Table C3-12 Instructions that add a PAC

Mnemonic	Instruction	See
PACIASP	Add PAC to instruction address using APIAKey_EL1 and SP	*PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA* on page C6-1156
PACIAZ	Add PAC to instruction address using APIAKey_EL1 and zero	
PACIA1716	Add PAC to instruction address X17 using APIAKey_EL1 and X16	
PACIBSP	Add PAC to instruction address using APIBKey_EL1 and SP	*PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB* on page C6-1159
PACIBZ	Add PAC to instruction address using APIBKey_EL1 and zero	
PACIB1716	Add PAC to instruction address X17 using APIBKey_EL1 and X16	
PACIA	Add PAC to instruction address using APIAKey_EL1, registers	*PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA* on page C6-1156

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-12 Instructions that add a PAC (continued)

Mnemonic	Instruction	See
PACDA	Add PAC to data address using APDAKey_EL1, registers	*PACDA, PACDZA* on page C6-1153
PACIB	Add PAC to instruction address using APIBKey_EL1, registers	*PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB* on page C6-1159
PACDB	Add PAC to data address using APDBKey_EL1, registers	*PACDB, PACDZB* on page C6-1154
PACIZA	Add PAC to instruction address using APIAKey_EL1, register and zero	*PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA* on page C6-1156
PACDZA	Add PAC to data address using APDAKey_EL1, register and zero	*PACDA, PACDZA* on page C6-1153
PACIZB	Add PAC to instruction address using APIBKey_EL1, register and zero	*PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB* on page C6-1159
PACDZB	Add PAC to data address using APDBKey_EL1, register and zero	*PACDB, PACDZB* on page C6-1154
PACGA	Add generic PAC using APGAKey_EL1, registers	*PACGA* on page C6-1155

Table C3-13 shows the instructions that authenticate a PAC in a register:

Table C3-13 Instructions that authenticate a PAC

Mnemonic	Instruction	See
AUTIASP	Authenticate PAC for instruction address using APIAKey_EL1 and SP	*AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA* on page C6-815
AUTIAZ	Authenticate PAC for instruction address using APIAKey_EL1 and zero	
AUTIA1716	Authenticate PAC for instruction address X17 using APIAKey_EL1 and X16	
AUTIBSP	Authenticate PAC for instruction address using APIBKey_EL1 and SP	*AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB* on page C6-817
AUTIBZ	Authenticate PAC for instruction address using APIBKey_EL1 and zero	
AUTIB1716	Authenticate PAC for instruction address X17 using APIBKey_EL1 and X16	
AUTIA	Authenticate PAC for instruction address using APIAKey_EL1, registers	*AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA* on page C6-815
AUTDA	Authenticate PAC for data address using APDAKey_EL1, registers	*AUTDA, AUTDZA* on page C6-813
AUTIB	Authenticate PAC for instruction address using APIBKey_EL1, registers	*AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB* on page C6-817
AUTDB	Authenticate PAC for data address using APDBKey_EL1, registers	*AUTDB, AUTDZB* on page C6-814
AUTIZA	Authenticate PAC for instruction address using APIAKey_EL1, register and zero	*AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA* on page C6-815
AUTDZA	Authenticate PAC for data address using APDAKey_EL1, register and zero	*AUTDA, AUTDZA* on page C6-813
AUTIZB	Authenticate PAC for instruction address using APIBKey_EL1, register and zero	*AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB* on page C6-817
AUTDZB	Authenticate PAC for data address using APDBKey_EL1, register and zero	*AUTDB, AUTDZB* on page C6-814

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-14 shows the instructions that strip a PAC from a register, without performing any authentication:

Table C3-14 Instructions that strip a PAC

Mnemonic	Instruction	See
XPACLRI	Strip instruction address PAC from LR	*XPACD, XPACI, XPACLRI* on page C6-1394
XPACI	Strip instruction address PAC, register	
XPACD	Strip data address PAC, register	

Combined instructions that include pointer authentication

Each of these instructions combines a pointer authentication with another operation that uses the authenticated pointer. Table C3-15 shows these instructions:

Table C3-15 Combined pointer authentication instructions

Mnemonic	Instruction	See
RETAA	Authenticate PAC for LR using APIAKey_EL1 and SP, and return	*RETAA, RETAB* on page C6-1173
RETAB	Authenticate PAC for LR using APIBKey_EL1 and SP, and return	
BRAA	Authenticate PAC using APIAKey_EL1 (registers), and branch	*BRAA, BRAAZ, BRAB, BRABZ* on page C6-839
BRAB	Authenticate PAC using APIBKey_EL1 (registers), and branch	
BLRAA	Authenticate PAC using APIAKey_EL1 (registers), and branch with link	*BLRAA, BLRAAZ, BLRAB, BLRABZ* on page C6-836
BLRAB	Authenticate PAC using APIBKey_EL1 (registers), and branch with link	
BRAAZ	Authenticate PAC using APIAKey_EL1 (register and zero), and branch	*BRAA, BRAAZ, BRAB, BRABZ* on page C6-839
BRABZ	Authenticate PAC using APIBKey_EL1 (register and zero), and branch	
BLRAAZ	Authenticate PAC using APIAKey_EL1 (register and zero), and branch with link	*BLRAA, BLRAAZ, BLRAB, BLRABZ* on page C6-836
BLRABZ	Authenticate PAC using APIBKey_EL1 (register and zero), and branch with link	
ERETAA	Authenticate PAC for ELR using APIAKey_EL1 and SP, and exception return	*ERETAA, ERETAB* on page C6-924
ERETAB	Authenticate PAC for ELR using APIBKey_EL1 and SP, and exception return	
LDRAA	Authenticate PAC for data address using APDAKey_EL1 (register and zero) and Load	*LDRAA, LDRAB* on page C6-1007
LDRAB	Authenticate PAC for data address using APDBKey_EL1 (register and zero) and Load	

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C3.2 Loads and stores

This section describes the Load/Store instructions. It contains the following subsections:

- *Load/Store register*.
- *Load/Store register (unscaled offset)* on page C3-210.
- *Load/Store Pair* on page C3-211.
- *Load/Store Non-temporal Pair* on page C3-212.
- *Load/Store unprivileged* on page C3-213.
- *Load-Exclusive/Store-Exclusive* on page C3-213.
- *Load-Acquire/Store-Release* on page C3-214.
- *LoadLOAcquire/StoreLORelease* on page C3-216.
- *Load/Store scalar SIMD and floating-point* on page C3-216.
- *Load/Store Vector* on page C3-218.
- *Prefetch memory* on page C3-220.
- *Compare and Swap* on page C3-221.
- *Atomic memory operations* on page C3-222.
- *Swap* on page C3-224.
- *Memory Tagging instructions* on page C3-225.

The requirements for the alignment of data memory accesses are strict, for more information see *Alignment of data accesses* on page B2-148.

The additional control bits SCTLR_ELx.SA and SCTLR_EL1.SA0 control whether the stack pointer must be quadword aligned when used as a base register. See *SP alignment checking* on page D1-2333. Using a misaligned stack pointer generates an SP alignment fault exception.

For information about the encoding structure of the instructions in this instruction group, see *Loads and Stores* on page C4-281.

Note

In some cases, Load/Store instructions can lead to CONSTRAINED UNPREDICTABLE behavior. See *AArch64 CONSTRAINED UNPREDICTABLE behaviors* on page K1-7855.

C3.2.1 Load/Store register

The Load/Store register instructions support the following addressing modes:

- Base plus a scaled 12-bit unsigned immediate offset or base plus an unscaled 9-bit signed immediate offset.
- Base plus a 64-bit register offset, optionally scaled.
- Base plus a 32-bit extended register offset, optionally scaled.
- Pre-indexed by an unscaled 9-bit signed immediate offset.
- Post-indexed by an unscaled 9-bit signed immediate offset.
- PC-relative literal for loads of 32 bits or more.

See also *Load/Store addressing modes* on page C1-189.

If a Load instruction specifies writeback and the register being loaded is also the base register, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs the load using the specified addressing mode and the base register becomes UNKNOWN. In addition, if an exception occurs during the execution of such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If a Store instruction performs a writeback and the register that is stored is also the base register, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs the store to the designated register using the specified addressing mode, but the value stored is UNKNOWN.

Table C3-16 shows the Load/Store Register instructions.

Table C3-16 Load/Store register instructions

Mnemonic	Instruction	See
LDR	Load register (register offset)	*LDR (register)* on page C6-1005
	Load register (immediate offset)	*LDR (immediate)* on page C6-1000
	Load register (PC-relative literal)	*LDR (literal)* on page C6-1003
LDRB	Load byte (register offset)	*LDRB (register)* on page C6-1012
	Load byte (immediate offset)	*LDRB (immediate)* on page C6-1009
LDRSB	Load signed byte (register offset)	*LDRSB (register)* on page C6-1022
	Load signed byte (immediate offset)	*LDRSB (immediate)* on page C6-1019
LDRH	Load halfword (register offset)	*LDRH (register)* on page C6-1017
	Load halfword (immediate offset)	*LDRH (immediate)* on page C6-1014
LDRSH	Load signed halfword (register offset)	*LDRSH (register)* on page C6-1027
	Load signed halfword (immediate offset)	*LDRSH (immediate)* on page C6-1024
LDRSW	Load signed word (register offset)	*LDRSW (register)* on page C6-1033
	Load signed word (immediate offset)	*LDRSW (immediate)* on page C6-1029
	Load signed word (PC-relative literal)	*LDRSW (literal)* on page C6-1032
STR	Store register (register offset)	*STR (register)* on page C6-1267
	Store register (immediate offset)	*STR (immediate)* on page C6-1264
STRB	Store byte (register offset)	*STRB (register)* on page C6-1272
	Store byte (immediate offset)	*STRB (immediate)* on page C6-1269
STRH	Store halfword (register offset)	*STRH (register)* on page C6-1277
	Store halfword (immediate offset)	*STRH (immediate)* on page C6-1274

C3.2.2 Load/Store register (unscaled offset)

The Load/Store register instructions with an unscaled offset support only one addressing mode:

- Base plus an unscaled 9-bit signed immediate offset.

See *Load/Store addressing modes* on page C1-189.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The Load/Store register (unscaled offset) instructions are required to disambiguate this instruction class from the Load/Store register instruction forms that support an addressing mode of base plus a scaled, unsigned 12-bit immediate offset, because that can represent some offset values in the same range.

The ambiguous immediate offsets are byte offsets that are both:

- In the range 0-255, inclusive.
- Naturally aligned to the access size.

Other byte offsets in the range -256 to 255 inclusive are unambiguous. An assembler program translating a Load/Store instruction, for example `LDR`, is required to encode an unambiguous offset using the unscaled 9-bit offset form, and to encode an ambiguous offset using the scaled 12-bit offset form. A programmer might force the generation of the unscaled 9-bit form by using one of the mnemonics in Table C3-17. Arm recommends that a disassembler outputs all unscaled 9-bit offset forms using one of these mnemonics, but unambiguous offsets can be output using a Load/Store single register mnemonic, for example, `LDR`.

Table C3-17 shows the Load/Store register instructions with an unscaled offset.

Table C3-17 Load/Store register (unscaled offset) instructions

Mnemonic	Instruction	See
`LDUR`	Load register (unscaled offset)	*LDUR* on page C6-1082
`LDURB`	Load byte (unscaled offset)	*LDURB* on page C6-1084
`LDURSB`	Load signed byte (unscaled offset)	*LDURSB* on page C6-1086
`LDURH`	Load halfword (unscaled offset)	*LDURH* on page C6-1085
`LDURSH`	Load signed halfword (unscaled offset)	*LDURSH* on page C6-1088
`LDURSW`	Load signed word (unscaled offset)	*LDURSW* on page C6-1090
`STUR`	Store register (unscaled offset)	*STUR* on page C6-1315
`STURB`	Store byte (unscaled offset)	*STURB* on page C6-1317
`STURH`	Store halfword (unscaled offset)	*STURH* on page C6-1318

C3.2.3 Load/Store Pair

The Load/Store Pair instructions support the following addressing modes:

- Base plus a scaled 7-bit signed immediate offset.
- Pre-indexed by a scaled 7-bit signed immediate offset.
- Post-indexed by a scaled 7-bit signed immediate offset.

See also *Load/Store addressing modes* on page C1-189.

If a Load Pair instruction specifies the same register for the two registers that are being loaded, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs all the loads using the specified addressing mode and the register that is loaded takes an UNKNOWN value.

If a Load Pair instruction specifies writeback and one of the registers being loaded is also the base register, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The instruction performs all of the loads using the specified addressing mode, and the base register becomes UNKNOWN. In addition, if an exception occurs during the instruction, the base address might be corrupted so that the instruction cannot be repeated.

If a Store Pair instruction performs a writeback and one of the registers being stored is also the base register, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs all the stores of the registers indicated by the specified addressing mode, but the value stored for the base register is UNKNOWN.

Table C3-18 shows the Load/Store Pair instructions.

Table C3-18 Load/Store Pair instructions

Mnemonic	Instruction	See
LDP	Load Pair	*LDP* on page C6-994
LDPSW	Load Pair signed words	*LDPSW* on page C6-997
STP	Store Pair	*STP* on page C6-1261

C3.2.4 Load/Store Non-temporal Pair

The Load/Store Non-temporal Pair instructions support only one addressing mode:

- Base plus a scaled 7-bit signed immediate offset.

See *Load/Store addressing modes* on page C1-189.

The Load/Store Non-temporal Pair instructions provide a hint to the memory system that an access is non-temporal or streaming, and unlikely to be repeated in the near future. This means that data caching is not required. However, depending on the memory type, the instructions might permit memory reads to be preloaded and memory writes to be gathered to accelerate bulk memory transfers.

In addition, there is an exception to the usual memory ordering rules. If an address dependency exists between two memory reads, and a Load Non-temporal Pair instruction generated the second read, then in the absence of any other barrier mechanism to achieve order, the memory accesses can be observed in any order by the other observers within the shareability domain of the memory addresses being accessed.

If a Load Non-Temporal Pair instruction specifies the same register for the two registers that are being loaded, then behavior is CONSTRAINED UNPREDICTABLE and one of the following must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs all the loads using the specified addressing mode and the register that is loaded takes an UNKNOWN value.

Table C3-19 shows the Load/Store Non-temporal Pair instructions.

Table C3-19 Load/Store Non-temporal Pair instructions

Mnemonic	Instruction	See
LDNP	Load Non-temporal Pair	*LDNP* on page C6-992
STNP	Store Non-temporal Pair	*STNP* on page C6-1259

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C3.2.5 Load/Store unprivileged

The Load/Store unprivileged instructions support only one addressing mode:

- Base plus an unscaled 9-bit signed immediate offset.

See *Load/Store addressing modes* on page C1-189.

The accesses permissions that apply to accesses made at EL0 apply to the memory accesses made by a Load/Store unprivileged instruction that is executed either:

- At EL1 when the *Effective value* of PSTATE.UAO is 0.
- At EL2 when both the *Effective value* of HCR_EL2.{E2H, TGE} is {1, 1} and the *Effective value* of PSTATE.UAO is 0.

Otherwise, memory accesses made by a Load/Store unprivileged instruction are subject to the access permissions that apply to the Exception level at which the instruction is executed. These are the permissions that apply to the corresponding Load/Store register instruction, see *Load/Store register* on page C3-209.

Note

This means that when the value of PSTATE.UAO is 1 the access permissions for a Load/Store unprivileged instruction are always the same as those for the corresponding Load/Store register instruction.

Table C3-20 shows the Load/Store unprivileged instructions.

Table C3-20 Load-Store unprivileged instructions

Mnemonic	Instruction	See
LDTR	Load unprivileged register	*LDTR* on page C6-1056
LDTRB	Load unprivileged byte	*LDTRB* on page C6-1058
LDTRSB	Load unprivileged signed byte	*LDTRSB* on page C6-1062
LDTRH	Load unprivileged halfword	*LDTRH* on page C6-1060
LDTRSH	Load unprivileged signed halfword	*LDTRSH* on page C6-1064
LDTRSW	Load unprivileged signed word	*LDTRSW* on page C6-1066
STTR	Store unprivileged register	*STTR* on page C6-1297
STTRB	Store unprivileged byte	*STTRB* on page C6-1299
STTRH	Store unprivileged halfword	*STTRH* on page C6-1301

C3.2.6 Load-Exclusive/Store-Exclusive

The Load-Exclusive/Store-Exclusive instructions support only one addressing mode:

- Base register with no offset.

See *Load/Store addressing modes* on page C1-189.

The Load-Exclusive instructions mark the physical address being accessed as an exclusive access. This exclusive access mark is checked by the Store-Exclusive instruction, permitting the construction of atomic read-modify-write operations on shared memory variables, semaphores, mutexes, and spinlocks. See *Synchronization and semaphores* on page B2-167.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The Load-Exclusive/Store-Exclusive instructions other than Load-Exclusive pair and Store-Exclusive pair require natural alignment, and an unaligned address generates an Alignment fault. Memory accesses generated by Load-Exclusive pair or Store-Exclusive pair instructions must be aligned to the size of the pair, otherwise the access generates an Alignment fault. When a Store-Exclusive pair succeeds, it causes a single-copy atomic update of the entire memory location.

Armv8.4 introduces changes to the alignment requirements of Load-Exclusive/Store-Exclusive instructions, see *Load-Exclusive/ Store-Exclusive and Atomic instructions* on page B2-148.

Table C3-21 shows the Load-Exclusive/Store-Exclusive instructions.

Table C3-21 Load-Exclusive/Store-Exclusive instructions

Mnemonic	Instruction	See
LDXR	Load Exclusive register	*LDXR* on page C6-1093
LDXRB	Load Exclusive byte	*LDXRB* on page C6-1095
LDXRH	Load Exclusive halfword	*LDXRH* on page C6-1096
LDXP	Load Exclusive pair	*LDXP* on page C6-1091
STXR	Store Exclusive register	*STXR* on page C6-1322
STXRB	Store Exclusive byte	*STXRB* on page C6-1324
STXRH	Store Exclusive halfword	*STXRH* on page C6-1326
STXP	Store Exclusive pair	*STXP* on page C6-1319

C3.2.7 Load-Acquire/Store-Release

The Load-Acquire, Load-AcquirePC, and Store-Release instructions support only one addressing mode:

- Base register with no offset.

See *Load/Store addressing modes* on page C1-189.

The Load-Acquire, Load-AcquirePC, and Store-Release instructions can remove the requirement to use the explicit DMB memory barrier instruction. For more information about the ordering of Load-Acquire, Load-AcquirePC, and Store-Release, see *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

The Load-Acquire, Load-AcquirePC, and Store-Release instructions other than Load-Acquire pair and Store-Release pair require natural alignment, and an unaligned address generates an Alignment fault. Memory accesses generated by Load-Acquire pair or Store-Release pair instructions must be aligned to the size of the pair, otherwise the access generates an Alignment fault.

A Store-Release Exclusive instruction only has the Release semantics if the store is successful.

Armv8.1 adds more instructions with load-acquire and store-release mechanisms, see *LoadLOAcquire/StoreLORelease* on page C3-216.

Armv8.4 introduces changes to the alignment requirements of Load-Acquire/Store-Release instructions, see *ARMv8.4-RCPC* on page A2-85.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-22 shows the Non-exclusive Load-Acquire/Store-Release instructions.

Table C3-22 Non-exclusive Load-Acquire and Store-Release instructions

Mnemonic	Instruction	See
LDAPR	Load-Acquire RCpc Register	*LDAPR* on page C6-944
LDAPRB	Load-Acquire RCpc Register Byte	*LDAPRB* on page C6-946
LDAPRH	Load-Acquire RCpc Register Halfword	*LDAPRH* on page C6-948
LDAPUR	Load-Acquire RCpc Register (unscaled)	*LDAPUR* on page C6-950
LDAPURB	Load-Acquire RCpc Register Byte (unscaled)	*LDAPURB* on page C6-952
LDAPURH	Load-Acquire RCpc Register Halfword (unscaled)	*LDAPURH* on page C6-954
LDAPURSB	Load-Acquire RCpc Register Signed Byte (unscaled) 32-bit	*LDAPURSB* on page C6-956
LDAPURSB	Load-Acquire RCpc Register Signed Byte (unscaled) 64-bit	*LDAPURSB* on page C6-956
LDAPURSH	Load-Acquire RCpc Register Signed Halfword (unscaled) 32-bit	*LDAPURSH* on page C6-958
LDAPURSH	Load-Acquire RCpc Register Signed Halfword (unscaled) 64-bit	*LDAPURSH* on page C6-958
LDAPURSW	Load-Acquire RCpc Register Signed Word (unscaled)	*LDAPURSW* on page C6-960
LDAR	Load-Acquire Register	*LDAR* on page C6-962
LDARB	Load-Acquire Byte	*LDARB* on page C6-964
LDARH	Load-Acquire Halfword	*LDARH* on page C6-965
STLR	Store-Release Register	*STLR* on page C6-1241
STLRB	Store-Release Byte	*STLRB* on page C6-1242
STLRH	Store-Release Halfword	*STLRH* on page C6-1243
STLUR	Store-Release Register (unscaled)	*STLUR* on page C6-1244
STLURB	Store-Release Register Byte (unscaled)	*STLURB* on page C6-1246
STLURH	Store-Release Register Halfword (unscaled)	*STLURH* on page C6-1248

Table C3-23 shows the Exclusive Load-Acquire/Store-Release instructions.

Table C3-23 Exclusive Load-Acquire and Store-Release instructions

Mnemonic	Instruction	See
LDAXR	Load-Acquire Exclusive register	*LDAXR* on page C6-968
LDAXRB	Load-Acquire Exclusive byte	*LDAXRB* on page C6-970
LDAXRH	Load-Acquire Exclusive halfword	*LDAXRH* on page C6-971
LDAXP	Load-Acquire Exclusive pair	*LDAXP* on page C6-966
STLXR	Store-Release Exclusive register	*STLXR* on page C6-1253

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-23 Exclusive Load-Acquire and Store-Release instructions (continued)

Mnemonic	Instruction	See
STLXRB	Store-Release Exclusive byte	*STLXRB* on page C6-1255
STLXRH	Store-Release Exclusive halfword	*STLXRH* on page C6-1257
STLXP	Store-Release Exclusive pair	*STLXP* on page C6-1250

C3.2.8 LoadLOAcquire/StoreLORelease

The LoadLOAcquire/StoreLORelease instructions support only one addressing mode:

- Base register with no offset.

See *Load/Store addressing modes* on page C1-189.

The LoadLOAcquire/StoreLORelease instructions can remove the requirement to use the explicit DMB memory barrier instruction. For more information about the ordering of LoadLOAcquire/StoreLORelease, see *LoadLOAcquire, StoreLORelease* on page B2-141.

The LoadLOAcquire/StoreLORelease instructions require natural alignment, and an unaligned address generates an Alignment fault.

Table C3-24 shows the LoadLOAcquire/StoreLORelease instructions.

Table C3-24 LoadLOAcquire and StoreLORelease instructions

Mnemonic	Instruction	See
LDLARB	LoadLOAcquire byte	*LDLARB* on page C6-988
LDLARH	LoadLOAcquire halfword	*LDLARH* on page C6-989
LDLAR	LoadLOAcquire register	*LDLAR* on page C6-990
STLLRB	StoreLORelease byte	*STLLRB* on page C6-1237
STLLRH	StoreLORelease halfword	*STLLRH* on page C6-1238
STLLR	StoreLORelease register	*STLLR* on page C6-1239

C3.2.9 Load/Store scalar SIMD and floating-point

The Load/Store scalar SIMD and floating-point instructions operate on scalar values in the SIMD and floating-point register file as described in *SIMD and floating-point scalar register names* on page C1-187. The memory addressing modes available, described in *Load/Store addressing modes* on page C1-189, are identical to the general-purpose register Load/Store instructions, and like those instructions permit arbitrary address alignment unless strict alignment checking is enabled. However, unlike the Load/Store instructions that transfer general-purpose registers, Load/Store scalar SIMD and floating-point instructions make no guarantee of atomicity, even when the address is naturally aligned to the size of the data.

Load/Store scalar SIMD and floating-point register

The Load/Store scalar SIMD and floating-point register instructions support the following addressing modes:

- Base plus a scaled 12-bit unsigned immediate offset or base plus unscaled 9-bit signed immediate offset.
- Base plus 64-bit register offset, optionally scaled.
- Base plus 32-bit extended register offset, optionally scaled.
- Pre-indexed by an unscaled 9-bit signed immediate offset.
- Post-indexed by an unscaled 9-bit signed immediate offset.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- PC-relative literal for loads of 32 bits or more.

For more information on the addressing modes, see *Load/Store addressing modes* on page C1-189.

Note

The unscaled 9-bit signed immediate offset address mode requires its own instruction form, see *Load/Store scalar SIMD and floating-point register (unscaled offset)*.

Table C3-25 shows the Load/Store instructions for a single SIMD and floating-point register.

Table C3-25 Load/Store single SIMD and floating-point register instructions

Mnemonic	Instruction	See
LDR	Load scalar SIMD&FP register (register offset)	*LDR (register, SIMD&FP)* on page C7-1843
	Load scalar SIMD&FP register (immediate offset)	*LDR (immediate, SIMD&FP)* on page C7-1837
	Load scalar SIMD&FP register (PC-relative literal)	*LDR (literal, SIMD&FP)* on page C7-1841
STR	Store scalar SIMD&FP register (register offset)	*STR (register, SIMD&FP)* on page C7-2156
	Store scalar SIMD&FP register (immediate offset)	*STR (immediate, SIMD&FP)* on page C7-2152

Load/Store scalar SIMD and floating-point register (unscaled offset)

The Load /Store scalar SIMD and floating-point register instructions support only one addressing mode:

- Base plus an unscaled 9-bit signed immediate offset.

See also *Load/Store addressing modes* on page C1-189.

The Load/Store scalar SIMD and floating-point register (unscaled offset) instructions are required to disambiguate this instruction class from the Load/Store single SIMD and floating-point instruction forms that support an addressing mode of base plus a scaled, unsigned 12-bit immediate offset. This is similar to the Load/Store register (unscaled offset) instructions, that disambiguate this instruction class from the Load/Store register instruction, see *Load/Store register (unscaled offset)* on page C3-210.

Table C3-26 shows the Load/Store SIMD and floating-point register instructions with an unscaled offset.

Table C3-26 Load/Store SIMD and floating-point register instructions

Mnemonic	Instruction	See
LDUR	Load scalar SIMD&FP register (unscaled offset)	*LDUR (SIMD&FP)* on page C7-1846
STUR	Store scalar SIMD&FP register (unscaled offset)	*STUR (SIMD&FP)* on page C7-2159

Load/Store SIMD and Floating-point register pair

The Load/Store SIMD and floating-point register pair instructions support the following addressing modes:

- Base plus a scaled 7-bit signed immediate offset.
- Pre-indexed by a scaled 7-bit signed immediate offset.
- Post-indexed by a scaled 7-bit signed immediate offset.

See also *Load/Store addressing modes* on page C1-189.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If a Load pair instruction specifies the same register for the two registers that are being loaded, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs all of the loads using the specified addressing mode and the register being loaded takes an UNKNOWN value.

Table C3-27 shows the Load/Store SIMD and floating-point register pair instructions.

Table C3-27 Load/Store SIMD and floating-point register pair instructions

Mnemonic	Instruction	See
LDP	Load pair of scalar SIMD&FP registers	*LDP (SIMD&FP)* on page C7-1833
STP	Store pair of scalar SIMD&FP registers	*STP (SIMD&FP)* on page C7-2149

Load/Store SIMD and Floating-point Non-temporal pair

The Load/Store SIMD and Floating-point Non-temporal pair instructions support only one addressing mode:

- Base plus a scaled 7-bit signed immediate offset.

See also *Load/Store addressing modes* on page C1-189.

The Load/Store Non-temporal pair instructions provide a hint to the memory system that an access is non-temporal or streaming, and unlikely to be repeated in the near future. This means that data caching is not required. However, depending on the memory type, the instructions might permit memory reads to be preloaded and memory writes to be gathered to accelerate bulk memory transfers.

In addition, there is an exception to the usual memory ordering rules. If an address dependency exists between two memory reads, and a Load non-temporal pair instruction generated the second read, then in the absence of any other barrier mechanism to achieve order, those memory accesses can be observed in any order by the other observers within the shareability domain of the memory addresses being accessed.

If a Load Non-temporal pair instruction specifies the same register for the two registers that are being loaded, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs all the loads using the specified addressing mode and the register that is loaded takes an UNKNOWN value.

Table C3-28 shows the Load/Store SIMD and floating-point Non-temporal pair instructions.

Table C3-28 Load/Store SIMD and floating-point Non-temporal pair instructions

Mnemonic	Instruction	See
LDNP	Load pair of scalar SIMD&FP registers	*LDNP (SIMD&FP)* on page C7-1831
STNP	Store pair of scalar SIMD&FP registers	*STNP (SIMD&FP)* on page C7-2147

C3.2.10 Load/Store Vector

The Vector Load/Store structure instructions support the following addressing modes:

- Base register only.
- Post-indexed by a 64-bit register.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Post-indexed by an immediate, equal to the number of bytes transferred.

Load/Store vector instructions, like other Load/Store instructions, allow any address alignment, unless strict alignment checking is enabled. If strict alignment checking is enabled, then alignment checking to the size of the element is performed. However, unlike the Load/Store instructions that transfer general-purpose registers, the Load/Store vector instructions do not guarantee atomicity, even when the address is naturally aligned to the size of the element.

Load/Store structures

Table C3-29 shows the Load/Store structure instructions. A post-increment immediate offset, if present, must be 8, 16, 24, 32, 48, or 64, depending on the number of elements transferred.

Table C3-29 Load/Store multiple structures instructions

Mnemonic	Instruction	See
LD1	Load single 1-element structure to one lane of one register	*LD1 (single structure)* on page C7-1794
	Load multiple 1-element structures to one register or to two, three, or four consecutive registers	*LD1 (multiple structures)* on page C7-1790
LD2	Load single 2-element structure to one lane of two consecutive registers	*LD2 (single structure)* on page C7-1804
	Load multiple 2-element structures to two consecutive registers	*LD2 (multiple structures)* on page C7-1801
LD3	Load single 3-element structure to one lane of three consecutive registers	*LD3 (single structure)* on page C7-1814
	Load multiple 3-element structures to three consecutive registers	*LD3 (multiple structures)* on page C7-1811
LD4	Load single 4-element structure to one lane of four consecutive registers	*LD4 (single structure)* on page C7-1824
	Load multiple 4-element structures to four consecutive registers	*LD4 (multiple structures)* on page C7-1821
ST1	Store single 1-element structure from one lane of one register	*ST1 (single structure)* on page C7-2122
	Store multiple 1-element structures from one register, or from two, three, or four consecutive registers	*ST1 (multiple structures)* on page C7-2118
ST2	Store single 2-element structure from one lane of two consecutive registers	*ST2 (single structure)* on page C7-2129
	Store multiple 2-element structures from two consecutive registers	*ST2 (multiple structures)* on page C7-2126
ST3	Store single 3-element structure from one lane of three consecutive registers	*ST3 (single structure)* on page C7-2136
	Store multiple 3-element structures from three consecutive registers	*ST3 (multiple structures)* on page C7-2133
ST4	Store single 4-element structure from one lane of four consecutive registers	*ST4 (single structure)* on page C7-2143
ST4	Store multiple 4-element structures from four consecutive registers	*ST4 (multiple structures)* on page C7-2140

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Load single structure and replicate

Table C3-30 shows the Load single structure and replicate instructions. A post-increment immediate offset, if present, must be 1, 2, 3, 4, 6, 8, 12, 16, 24, or 32, depending on the number of elements transferred.

Table C3-30 Load single structure and replicate instructions

Mnemonic	Instruction	See
LD1R	Load single 1-element structure and replicate to all lanes of one register	*LD1R* on page C7-1798
LD2R	Load single 2-element structure and replicate to all lanes of two registers	*LD2R* on page C7-1808
LD3R	Load single 3-element structure and replicate to all lanes of three registers	*LD3R* on page C7-1818
LD4R	Load single 4-element structure and replicate to all lanes of four registers	*LD4R* on page C7-1828

C3.2.11 Prefetch memory

The Prefetch memory instructions support the following addressing modes:

- Base plus a scaled 12-bit unsigned immediate offset or base plus an unscaled 9-bit signed immediate offset.
- Base plus a 64-bit register offset. This can be optionally scaled by 8-bits, for example LSL#3.
- Base plus a 32-bit extended register offset. This can be optionally scaled by 8-bits.
- PC-relative literal.

The prefetch memory instructions signal to the memory system that memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory access when they do occur, such as preloading the specified address into one or more caches. Because these signals are only hints, it is valid for the PE to treat any or all prefetch instructions as a NOP.

Because they are hints to the memory system, the operation of a PRFM instruction cannot cause a synchronous exception. However, a memory operation performed as a result of one of these memory system hints might in exceptional cases trigger an asynchronous event, and thereby influence the execution of the PE. An example of an asynchronous event that might be triggered is an SError interrupt.

A PRFM instruction can only have an effect on software visible structures, such as caches and translation lookaside buffers associated with memory locations that can be accessed by reads, writes, or execution as defined in the translation regime of the current Exception level.

A PRFM instruction is guaranteed not to access Device memory.

A PRFM instruction using a PLI hint must not result in any access that could not be performed by the PE speculatively fetching an instruction. Therefore, if all associated MMUs are disabled, a PLI hint cannot access any memory location that cannot be accessed by instruction fetches.

The PRFM instructions require an additional <prfop> operand to be specified, which must be one of the following:

```
PLDL1KEEP, PLDL1STRM, PLDL2KEEP, PLDL2STRM, PLDL3KEEP, PLDL3STRM

PSTL1KEEP, PSTL1STRM, PSTL2KEEP, PSTL2STRM, PSTL3KEEP, PSTL3STRM

PLIL1KEEP, PLIL1STRM, PLIL2KEEP, PLIL2STRM, PLIL3KEEP, PLIL3STRM
```

<prfop> is defined as <type><target><policy>.

Here:

<type> Is one of:
- PLD Prefetch for load.
- PST Prefetch for store.
- PLI Preload instructions.

<target> Is one of:
- L1 Level 1 cache.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

	L2	Level 2 cache.
	L3	Level 3 cache.
<policy>	Is one of:	
	KEEP	Retained or temporal prefetch, allocated in the cache normally.
	STRM	Streaming or non-temporal prefetch, for data that is used only once.

PRFUM explicitly uses the unscaled 9-bit signed immediate offset addressing mode, as described in *Load/Store register (unscaled offset)* on page C3-210.

Table C3-31 shows the Prefetch memory instructions.

Table C3-31 Prefetch memory instructions

Mnemonic	Instruction	See
PRFM	Prefetch memory (register offset)	*PRFM (register)* on page C6-1165
	Prefetch memory (immediate offset)	*PRFM (immediate)* on page C6-1161
	Prefetch memory (PC-relative offset)	*PRFM (literal)* on page C6-1163
PRFUM	Prefetch memory (unscaled offset)	*PRFUM* on page C6-1167

C3.2.12 Compare and Swap

The Compare and Swap instructions support only one addressing mode:

- Base register only.

See also *Load/Store addressing modes* on page C1-189.

For the purpose of permission checking, and for watchpoints, all of the Compare and Swap instructions are treated as performing both a load and a store.

The CAS instructions require natural alignment.

The CASP instructions require alignment to the total size of the memory being accessed.

All Compare and Swap instructions generate an Alignment fault if the address being accessed is not aligned to the size of the data structure being accessed.

The instructions are provided with ordering options, which map to the acquire and release definitions used in the Armv8-A architecture. If a compare and swap instruction does not perform a store, then the instruction does not have release semantics, regardless of the instruction ordering options.

The atomic instructions with release semantics have the same rules as Store-Release instructions regarding multi-copy atomicity.

For the CAS and CASP instructions, the architecture permits that a data read clears any Exclusives monitors associated with that location, even if the compare subsequently fails. If these instructions generate a synchronous Data Abort, the registers which are compared and loaded are restored to the values held in the registers before the instruction was executed.

Table C3-32 Compare and swap instructions

Mnemonic	Instruction	See
CAS	Compare and swap	*CAS, CASA, CASAL, CASL* on page C6-851
CASB	Compare and swap byte	*CASB, CASAB, CASALB, CASLB* on page C6-844
CASH	Compare and swap halfword	*CASH, CASAH, CASALH, CASLH* on page C6-846
CASP	Compare and swap pair	*CASP, CASPA, CASPAL, CASPL* on page C6-848

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C3.2.13 Atomic memory operations

The atomic memory operation instructions support only one addressing mode:

- Base register only.

See also *Load/Store addressing modes* on page C1-189.

For the purpose of permission checking, and for watchpoints, all of the Compare and Swap instructions are treated as performing both a load and a store.

The `LD<OP>` and `ST<OP>` instructions require natural alignment, and an unaligned address generates an Alignment fault.

The instructions are provided with ordering options, which map to the acquire and release definitions used in the Armv8-A architecture. The atomic instructions with release semantics have the same rules as Store-Release instructions regarding multi-copy atomicity. These operations map to the acquire and release definitions, and are counted as Load-Acquire and Store-Release operations respectively.

For the `LD<OP>` instructions, where the source and destination registers are the same, if the instruction generates a synchronous Data Abort, then the source register is restored to the value it held before the instruction was executed.

The `ST<OP>` instructions, and `LD<OP>` instructions where the destination register is WZR or XZR, are not regarded as doing a read for the purpose of a `DMB LD` barrier.

Armv8.4 introduces changes to the alignment requirements of Atomic instructions, see *Load-Exclusive/Store-Exclusive and Atomic instructions* on page B2-148.

Table C3-33 Atomic memory operation instructions

Mnemonic	Instruction	See
`LDADD`	Atomic add	*LDADD, LDADDA, LDADDAL, LDADDL* on page C6-941
`LDADDB`	Atomic add on byte	*LDADDB, LDADDAB, LDADDALB, LDADDLB* on page C6-937
`LDADDH`	Atomic add on halfword	*LDADDH, LDADDAH, LDADDALH, LDADDLH* on page C6-939
`LDCLR`	Atomic bit clear	*LDCLR, LDCLRA, LDCLRAL, LDCLRL* on page C6-976
`LDCLRB`	Atomic bit clear on byte	*LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB* on page C6-972
`LDCLRH`	Atomic bit clear on halfword	*LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH* on page C6-974
`LDEOR`	Atomic exclusive OR	*LDEOR, LDEORA, LDEORAL, LDEORL* on page C6-983
`LDEORB`	Atomic exclusive OR on byte	*LDEORB, LDEORAB, LDEORALB, LDEORLB* on page C6-979
`LDEORH`	Atomic exclusive OR on halfword	*LDEORH, LDEORAH, LDEORALH, LDEORLH* on page C6-981
`LDSET`	Atomic bit set	*LDSET, LDSETA, LDSETAL, LDSETL* on page C6-1039
`LDSETB`	Atomic bit set on byte	*LDSETB, LDSETAB, LDSETALB, LDSETLB* on page C6-1035
`LDSETH`	Atomic bit set on halfword	*LDSETH, LDSETAH, LDSETALH, LDSETLH* on page C6-1037

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-33 Atomic memory operation instructions (continued)

Mnemonic	Instruction	See
LDMAX	Atomic signed maximum	*LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL* on page C6-1046
LDMAXB	Atomic signed maximum on byte	*LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB* on page C6-1042
LDMAXH	Atomic signed maximum on halfword	*LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH* on page C6-1044
LDMIN	Atomic signed minimum	*LDSMIN, LDSMINA, LDSMINAL, LDSMINL* on page C6-1053
LDMINB	Atomic signed minimum on byte	*LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB* on page C6-1049
LDMINH	Atomic signed minimum on halfword	*LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH* on page C6-1051
LDUMAX	Atomic unsigned maximum	*LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL* on page C6-1072
LDUMAXB	Atomic unsigned maximum on byte	*LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB* on page C6-1068
LDUMAXH	Atomic unsigned maximum on halfword	*LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH* on page C6-1070
LDUMIN	Atomic unsigned minimum	*LDUMIN, LDUMINA, LDUMINAL, LDUMINL* on page C6-1079
LDUMINB	Atomic unsigned minimum on byte	*LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB* on page C6-1075
LDUMINH	Atomic unsigned minimum on halfword	*LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH* on page C6-1077
STADD	Atomic add, without return	*STADD, STADDL* on page C6-1218
STADDB	Atomic add on byte, without return	*STADDB, STADDLB* on page C6-1214
STADDH	Atomic add on halfword, without return	*STADDH, STADDLH* on page C6-1216
STCLR	Atomic bit clear, without return	*STCLR, STCLRL* on page C6-1224
STCLRB	Atomic bit clear on byte, without return	*STCLRB, STCLRLB* on page C6-1220
STCLRH	Atomic bit clear on halfword, without return	*STCLRH, STCLRLH* on page C6-1222
STEOR	Atomic exclusive OR, without return	*STEOR, STEORL* on page C6-1230
STEORB	Atomic exclusive OR on byte, without return	*STEORB, STEORLB* on page C6-1226
STEORH	Atomic exclusive OR on halfword, without return	*STEORH, STEORLH* on page C6-1228
STSET	Atomic bit set, without return	*STSET, STSETL* on page C6-1283
STSETB	Atomic bit set on byte, without return	*STSETB, STSETLB* on page C6-1279
STSETH	Atomic bit set on halfword, without return	*STSETH, STSETLH* on page C6-1281
STMAX	Atomic signed maximum, without return	*STSMAX, STSMAXL* on page C6-1289

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-33 Atomic memory operation instructions (continued)

Mnemonic	Instruction	See
STMAXB	Atomic signed maximum on byte, without return	*STSMAXB, STSMAXLB* on page C6-1285
STMAXH	Atomic signed maximum on halfword, without return	*STSMAXH, STSMAXLH* on page C6-1287
STMIN	Atomic signed minimum, without return	*STSMIN, STSMINL* on page C6-1295
STMINB	Atomic signed minimum on byte, without return	*STSMINB, STSMINLB* on page C6-1291
STMINH	Atomic signed minimum on halfword, without return	*STSMINH, STSMINLH* on page C6-1293
STUMAX	Atomic unsigned maximum, without return	*STUMAX, STUMAXL* on page C6-1307
STUMAXB	Atomic unsigned maximum on byte, without return	*STUMAXB, STUMAXLB* on page C6-1303
STUMAXH	Atomic unsigned maximum on halfword, without return	*STUMAXH, STUMAXLH* on page C6-1305
STUMIN	Atomic unsigned minimum, without return	*STUMIN, STUMINL* on page C6-1313
STUMINB	Atomic unsigned minimum on byte, without return	*STUMINB, STUMINLB* on page C6-1309
STUMINH	Atomic unsigned minimum on halfword, without return	*STUMINH, STUMINLH* on page C6-1311

C3.2.14 Swap

The swap instructions support only one addressing mode:

- Base register only.

See also *Load/Store addressing modes* on page C1-189.

For the purpose of permission checking, and for watchpoints, all of the Compare and Swap instructions are treated as performing both a load and a store.

The SWP instructions require natural alignment, and an unaligned address generates an Alignment fault.

The instructions are provided with ordering options, which map to the acquire and release definitions used in the Armv8-A architecture. The atomic instructions with release semantics have the same rules as Store-Release instructions regarding multi-copy atomicity.

For the SWP instructions, where the source and destination registers are the same, if the instruction generates a synchronous Data Abort, then the source register is restored to the value it held before the instruction was executed.

Table C3-34 Swap instructions

Mnemonic	Instruction	See
SWP	Swap	*SWP, SWPA, SWPAL, SWPL* on page C6-1355
SWPB	Swap byte	*SWPB, SWPAB, SWPALB, SWPLB* on page C6-1351
SWPH	Swap halfword	*SWPH, SWPAH, SWPALH, SWPLH* on page C6-1353

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C3.2.15 Memory Tagging instructions

If the ARMv8.5-MemTag is implemented the following instructions are implemented.

Table C3-35 Tag generation instructions

Mnemonic	Instruction	See
ADDG	Add immediate value to Logical Address Tag	*ADDG* on page C6-787
GMI	Tag Mask Insert	*GMI* on page C6-928
IRG	Random Logical Address Tag generation	*IRG* on page C6-935
SUBG	Subtract immediate value to Logical Address Tag	*SUBG* on page C6-1340

Table C3-36 Pointer Arithmetic

Mnemonic	Instruction	See
SUBP(S)	Subtract address and set flags	*SUBPS* on page C6-1342

Table C3-37 Tag setting instructions

Mnemonic	Instruction	See
STG	Store Allocation Tag to granule	*STG* on page C6-1232,
STZG	Store Allocation Tag to granule Zeroing	*STZG* on page C6-1330
ST2G	Store Allocation Tag to two granules	*ST2G* on page C6-1212,
STZ2G	Store Allocation Tag to two granules Zeroing	*STZ2G* on page C6-1328
STGP	Store Allocation Tag to memory	*STGP* on page C6-1235

Table C3-38 Tag getting instructions

Mnemonic	Instruction	See
LDG	Load Allocation Tag	*LDG* on page C6-986

Table C3-39 Bulk Allocation Tag access

Mnemonic	Instruction	See
LDGM	Load an IMPLEMENTATION DEFINED number of Allocation Tags	*LDGM* on page C6-987
STGM	Store an IMPLEMENTATION DEFINED number of Allocation Tags	*STGM* on page C6-1234

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C3.3 Data processing - immediate

This section describes the instruction groups for data processing with immediate operands. It contains the following subsections:

- *Arithmetic (immediate)*.
- *Logical (immediate)*.
- *Move (wide immediate)* on page C3-227.
- *Move (immediate)* on page C3-227.
- *PC-relative address calculation* on page C3-228.
- *Bitfield move* on page C3-228.
- *Bitfield insert and extract* on page C3-229
- *Extract register* on page C3-229.
- *Shift (immediate)* on page C3-229.
- *Sign-extend and Zero-extend* on page C3-230.

For information about the encoding structure of the instructions in this instruction group, see *Data Processing -- Immediate* on page C4-268.

C3.3.1 Arithmetic (immediate)

The Arithmetic (immediate) instructions accept a 12-bit unsigned immediate value, optionally shifted left by 12 bits.

The Arithmetic (immediate) instructions that do not set Condition flags can read from and write to the current stack pointer. The flag setting instructions can read from the stack pointer, but they cannot write to it.

Table C3-40 shows the Arithmetic instructions with an immediate offset.

Table C3-40 Arithmetic instructions with an immediate

Mnemonic	Instruction	See
ADD	Add	*ADD (immediate)* on page C6-783
ADDS	Add and set flags	*ADDS (immediate)* on page C6-791
SUB	Subtract	*SUB (immediate)* on page C6-1336
SUBS	Subtract and set flags	*SUBS (immediate)* on page C6-1346
CMP	Compare	*CMP (immediate)* on page C6-880
CMN	Compare negative	*CMN (immediate)* on page C6-874

C3.3.2 Logical (immediate)

The Logical (immediate) instructions accept a bitmask immediate value that is a 32-bit pattern or a 64-bit pattern viewed as a vector of identical elements of size e = 2, 4, 8, 16, 32 or, 64 bits. Each element contains the same sub-pattern, that is a single run of 1 to (e - 1) nonzero bits from bit 0 followed by zero bits, then rotated by 0 to (e - 1) bits. This mechanism can generate 5334 unique 64-bit patterns as 2667 pairs of pattern and their bitwise inverse.

Note

Values that consist of only zeros or only ones cannot be described in this way.

The Logical (immediate) instructions that do not set the Condition flags can write to the current stack pointer, for example to align the stack pointer in a function prologue.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

Apart from ANDS, and its TST alias, Logical (immediate) instructions do not set the Condition flags. However, the final results of a bitwise operation can be tested by a CBZ, CBNZ, TBZ, or TBNZ conditional branch.

Table C3-41 shows the Logical immediate instructions.

Table C3-41 Logical immediate instructions

Mnemonic	Instruction	See
AND	Bitwise AND	*AND (immediate)* on page C6-797
ANDS	Bitwise AND and set flags	*ANDS (immediate)* on page C6-801
EOR	Bitwise exclusive OR	*EOR (immediate)* on page C6-919
ORR	Bitwise inclusive OR	*ORR (immediate)* on page C6-1149
TST	Test bits	*TST (immediate)* on page C6-1371

C3.3.3 Move (wide immediate)

The Move (wide immediate) instructions insert a 16-bit immediate, or inverted immediate, into a 16-bit aligned position in the destination register. The value of the other bits in the destination register depends on the variant used. The optional shift amount can be any multiple of 16 that is smaller than the register size.

Table C3-42 shows the Move (wide immediate) instructions.

Table C3-42 Move (wide immediate) instructions

Mnemonic	Instruction	See
MOVZ	Move wide with zero	*MOVZ* on page C6-1126
MOVN	Move wide with NOT	*MOVN* on page C6-1124
MOVK	Move wide with keep	*MOVK* on page C6-1122

C3.3.4 Move (immediate)

The Move (immediate) instructions are aliases for a single MOVZ, MOVN, or ORR (immediate with zero register), instruction to load an immediate value into the destination register. An assembler must permit a signed or unsigned immediate, as long as its binary representation can be generated using one of these instructions, and an assembler error results if the immediate cannot be generated in this way. On disassembly, it is unspecified whether the immediate is output as a signed or an unsigned value.

If there is a choice between the MOVZ, MOVN, and ORR instruction to encode the immediate, then an assembler must prefer MOVZ to MOVN, and MOVZ or MOVN to ORR, to ensure reversability. A disassembler must output ORR (immediate with zero register) MOVZ, and MOVN, as a MOV mnemonic except that the underlying instruction must be used when:

- ORR has an immediate that can be generated by a MOVZ or MOVN instruction.
- A MOVN instruction has an immediate that can be encoded by MOVZ.
- MOVZ #0 or MOVN #0 have a shift amount other than LSL #0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-43 shows the Move (immediate) instructions.

Table C3-43 Move (immediate) instructions

Mnemonic	Instruction	See
MOV	Move (inverted wide immediate)	*MOV (inverted wide immediate)* on page C6-1114
	Move (wide immediate)	*MOV (wide immediate)* on page C6-1116
	Move (bitmask immediate)	*MOV (bitmask immediate)* on page C6-1118

C3.3.5 PC-relative address calculation

The ADR instruction adds a signed, 21-bit immediate to the value of the program counter that fetched this instruction, and then writes the result to a general-purpose register. This permits the calculation of any byte address within ±1MB of the current PC.

The ADRP instruction shifts a signed, 21-bit immediate left by 12 bits, adds it to the value of the program counter with the bottom 12 bits cleared to zero, and then writes the result to a general-purpose register. This permits the calculation of the address at a 4KB aligned memory region. In conjunction with an ADD (immediate) instruction, or a Load/Store instruction with a 12-bit immediate offset, this allows for the calculation of, or access to, any address within ±4GB of the current PC.

Note

The term *page* used in the ADRP description is short-hand for the 4KB memory region, and is not related to the virtual memory translation granule size.

Table C3-44 shows the instructions used for PC-relative address calculations are as follows:

Table C3-44 PC-relative address calculation instructions

Mnemonic	Instruction	See
ADRP	Compute address of 4KB page at a PC-relative offset	*ADRP* on page C6-796
ADR	Compute address of label at a PC-relative offset.	*ADR* on page C6-795

C3.3.6 Bitfield move

The Bitfield move instructions copy a field of constant width from bit 0 in the source register to a constant bit position in the destination register, or from a constant bit position in the source register to bit 0 in the destination register. The remaining bits in the destination register are set as follows:

- For BFM, the remaining bits are unchanged.
- For UBFM the lower bits, if any, and upper bits, if any, are set to zero.
- For SBFM, the lower bits, if any, are set to zero, and the upper bits, if any, are set to a copy of the most-significant bit in the copied field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-45 shows the Bitfield move instructions.

Table C3-45 Bitfield move instructions

Mnemonic	Instruction	See
BFM	Bitfield move	*BFM* on page C6-826
SBFM	Signed bitfield move	*SBFM* on page C6-1195
UBFM	Unsigned bitfield move (32-bit)	*UBFM* on page C6-1376

C3.3.7 Bitfield insert and extract

The Bitfield insert and extract instructions are implemented as aliases of the Bitfield move instructions. Table C3-46 shows the Bitfield insert and extract aliases.

Table C3-46 Bitfield insert and extract instructions

Mnemonic	Instruction	See
BFC	Bitfield insert clear	*BFC* on page C6-822
BFI	Bitfield insert	*BFI* on page C6-824
BFXIL	Bitfield extract and insert low	*BFXIL* on page C6-828
SBFIZ	Signed bitfield insert in zero	*SBFIZ* on page C6-1193
SBFX	Signed bitfield extract	*SBFX* on page C6-1197
UBFIZ	Unsigned bitfield insert in zero	*UBFIZ* on page C6-1374
UBFX	Unsigned bitfield extract	*UBFX* on page C6-1378

C3.3.8 Extract register

Depending on the register width of the operands, the Extract register instruction copies a 32-bit or 64-bit field from a constant bit position within a double-width value formed by the concatenation of a pair of source registers to a destination register.

Table C3-47 shows the Extract (immediate) instructions.

Table C3-47 Extract register instructions

Mnemonic	Instruction	See
EXTR	Extract register from pair	*EXTR* on page C6-926

C3.3.9 Shift (immediate)

Shifts and rotates by a constant amount are implemented as aliases of the Bitfield move or Extract register instructions. The shift or rotate amount must be in the range 0 to one less than the register width of the instruction, inclusive.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-48 shows the aliases that can be used as immediate shift and rotate instructions.

Table C3-48 Aliases for immediate shift and rotate instructions

Mnemonic	Instruction	See
ASR	Arithmetic shift right	*ASR (immediate)* on page C6-807
LSL	Logical shift left	*LSL (immediate)* on page C6-1099
LSR	Logical shift right	*LSR (immediate)* on page C6-1105
ROR	Rotate right	*ROR (immediate)* on page C6-1182

C3.3.10 Sign-extend and Zero-extend

The Sign-extend and Zero-extend instructions are implemented as aliases of the Bitfield move instructions.

Table C3-49 shows the aliases that can be used as zero-extend and sign-extend instructions.

Table C3-49 Zero-extend and sign-extend instructions

Mnemonic	Instruction	See
SXTB	Sign-extend byte	*SXTB* on page C6-1357
SXTH	Sign-extend halfword	*SXTH* on page C6-1359
SXTW	Sign-extend word	*SXTW* on page C6-1361
UXTB	Unsigned extend byte	*UXTB* on page C6-1389
UXTH	Unsigned extend halfword	*UXTH* on page C6-1390

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C3.4 Data processing - register

This section describes the instruction groups for data processing with all register operands. It contains the following subsections:

- *Arithmetic (shifted register)*.
- *Arithmetic (extended register)* on page C3-232.
- *Arithmetic with carry* on page C3-233.
- *Flag manipulation instructions* on page C3-233.
- *Logical (shifted register)* on page C3-233.
- *Move (register)* on page C3-234.
- *Shift (register)* on page C3-234.
- *Multiply and divide* on page C3-235.
- *CRC32* on page C3-236.
- *Bit operation* on page C3-237.
- *Conditional select* on page C3-237.
- *Conditional comparison* on page C3-238.

For information about the encoding structure of the instructions in this instruction group, see *Data Processing -- Register* on page C4-313.

C3.4.1 Arithmetic (shifted register)

The Arithmetic (shifted register) instructions apply an optional shift operator to the second source register value before performing the arithmetic operation. The register width of the instruction controls whether the new bits are fed into the intermediate result on a right shift or rotate at bit[63] or bit[31].

The shift operators `LSL`, `ASR`, and `LSR` accept an immediate shift amount in the range 0 to one less than the register width of the instruction, inclusive.

Omitting the shift operator implies `LSL #0`, which means that there is no shift. A disassembler must not output `LSL #0`. However, a disassembler must output all other shifts by zero.

The current stack pointer, SP or WSP, cannot be used with this class of instructions. See *Arithmetic (extended register)* on page C3-232 for arithmetic instructions that can operate on the current stack pointer.

Table C3-50 shows the Arithmetic (shifted register) instructions.

Table C3-50 Arithmetic (shifted register) instructions

Mnemonic	Instruction	See
`ADD`	Add	*ADD (shifted register)* on page C6-785
`ADDS`	Add and set flags	*ADDS (shifted register)* on page C6-793
`SUB`	Subtract	*SUB (shifted register)* on page C6-1338
`SUBS`	Subtract and set flags	*SUBS (shifted register)* on page C6-1348
`CMN`	Compare negative	*CMN (shifted register)* on page C6-876
`CMP`	Compare	*CMP (shifted register)* on page C6-882
`NEG`	Negate	*NEG (shifted register)* on page C6-1138
`NEGS`	Negate and set flags	*NEGS* on page C6-1140

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C3.4.2 Arithmetic (extended register)

The extended register instructions provide an optional sign-extension or zero-extension of a portion of the second source register value, followed by an optional left shift by a constant amount of 1-4, inclusive.

The extended shift is described by the mandatory extend operator `SXTB`, `SXTH`, `SXTW`, `UXTB`, `UXTH`, or `UXTW`. This is followed by an optional left shift amount. If the shift amount is not specified, the default shift amount is zero. A disassembler must not output a shift amount of zero.

For 64-bit instruction forms, the additional operators `UXTX` and `SXTX` use all 64 bits of the second source register with an optional shift. In that case, Arm recommends `UXTX` as the operator. If and only if at least one register is SP, Arm recommends use of the `LSL` operator name, rather than `UXTX`, and when the shift amount is also zero then both the operator and the shift amount can be omitted. `UXTW` and `SXTW` both use all 32 bits of the second source register with an optional shift. In that case Arm recommends `UXTW` as the operator. If and only if at least one register is WSP, Arm recommends use of the `LSL` operator name, rather than `UXTW`, and when the shift amount is also zero then both the operator and the shift amount can be omitted.

For 32-bit instruction forms, the operators `UXTW` and `SXTW` both use all 32 bits of the second source register with an optional shift. In that case, Arm recommends `UXTW` as the operator. If and only if at least one register is WSP, Arm recommends use of the `LSL` operator name, rather than `UXTW`, and when the shift amount is also zero then both the operator and the shift amount can be omitted.

The non-flag setting variants of the extended register instruction permit the use of the current stack pointer as either the destination register and the first source register. The flag setting variants only permit the stack pointer to be used as the first source register.

In the 64-bit form of these instructions, the final register operand is written as `Wm` for all except the `UXTX`/`LSL` and `SXTX` extend operators. For example:

```
CMP X4, W5, SXTW
ADD X1, X2, W3, UXTB #2
SUB SP, SP, X1              // SUB SP, SP, X1, UXTX #0
```

Table C3-51 shows the Arithmetic (extended register) instructions.

Table C3-51 Arithmetic (extended register) instructions

Mnemonic	Instruction	See
`ADD`	Add	*ADD (extended register)* on page C6-780
`ADDS`	Add and set flags	*ADDS (extended register)* on page C6-788
`SUB`	Subtract	*SUB (extended register)* on page C6-1333
`SUBS`	Subtract and set flags	*SUBS (extended register)* on page C6-1343
`CMN`	Compare negative	*CMN (extended register)* on page C6-872
`CMP`	Compare	*CMP (extended register)* on page C6-878

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C3.4.3 Arithmetic with carry

The Arithmetic with carry instructions accept two source registers, with the carry flag as an additional input to the calculation. They do not support shifting of the second source register.

Table C3-52 shows the Arithmetic with carry instructions

Table C3-52 Arithmetic with carry instructions

Mnemonic	Instruction	See
ADC	Add with carry	*ADC* on page C6-776
ADCS	Add with carry and set flags	*ADCS* on page C6-778
SBC	Subtract with carry	*SBC* on page C6-1189
SBCS	Subtract with carry and set flags	*SBCS* on page C6-1191
NGC	Negate with carry	*NGC* on page C6-1142
NGCS	Negate with carry and set flags	*NGCS* on page C6-1144

C3.4.4 Flag manipulation instructions

The Flag manipulation instructions set the value of the NZCV condition flags directly.

The instructions SETF8 and SETF16 accept one source register and set the NZV condition flags based on the value of the input register. The instruction RMIF accepts one source register and two immediate values, rotating the first source register using the first immediate value and setting the NZCV condition flags masked by the second immediate value.

The instructions XAFLAG and AXFLAG convert PSTATE condition flags between the FCMP instruction format and an alternative format. See Table C6-1 on page C6-774 for more information.

Table C3-53 shows the Flag manipulation instructions.

Table C3-53 Flag manipulation instructions

Mnemonic	Instruction	See
AXFLAG	Convert from FCMP comparison format to the alternative format	*AXFLAG* on page C6-819
CFINV	Invert value of the PSTATE.C bit	*CFINV* on page C6-863
RMIF	Rotate, mask insert flags	*RMIF* on page C6-1181
SETF8	Evaluation of 8-bit flags	*SETF8, SETF16* on page C6-1200
SETF16	Evaluation of 16-bit flags	*SETF8, SETF16* on page C6-1200
XAFLAG	Convert from alternative format to FCMP comparison format	*XAFLAG* on page C6-1393

C3.4.5 Logical (shifted register)

The Logical (shifted register) instructions apply an optional shift operator to the second source register value before performing the main operation. The register width of the instruction controls whether the new bits are fed into the intermediate result on a right shift or rotate at bit[63] or bit[31].

The shift operators LSL, ASR, LSR, and ROR accept a constant immediate shift amount in the range 0 to one less than the register width of the instruction, inclusive.

Omitting the shift operator and amount implies LSL #0, which means that there is no shift. A disassembler must not output LSL #0. However, a disassembler must output all other shifts by zero.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

Apart from ANDS, TST, and BICS the logical instructions do not set the Condition flags, but the final result of a bit operation can usually directly control a CBZ, CBNZ, TBZ, or TBNZ conditional branch.

Table C3-54 shows the Logical (shifted register) instructions.

Table C3-54 Logical (shifted register) instructions

Mnemonic	Instruction	See
AND	Bitwise AND	*AND (shifted register)* on page C6-799
ANDS	Bitwise AND and set flags	*ANDS (shifted register)* on page C6-803
BIC	Bitwise bit clear	*BIC (shifted register)* on page C6-830
BICS	Bitwise bit clear and set flags	*BICS (shifted register)* on page C6-832
EON	Bitwise exclusive OR NOT	*EON (shifted register)* on page C6-917
EOR	Bitwise exclusive OR	*EOR (shifted register)* on page C6-921
ORR	Bitwise inclusive OR	*ORR (shifted register)* on page C6-1151
MVN	Bitwise NOT	*MVN* on page C6-1136
ORN	Bitwise inclusive OR NOT	*ORN (shifted register)* on page C6-1147
TST	Test bits	*TST (shifted register)* on page C6-1372

C3.4.6 Move (register)

The Move (register) instructions are aliases for other data processing instructions. They copy a value from a general-purpose register to another general-purpose register or the current stack pointer, or from the current stack pointer to a general-purpose register.

Table C3-55 MOV register instructions

Mnemonic	Instruction	See
MOV	Move register	*MOV (register)* on page C6-1120
	Move register to SP or move SP to register	*MOV (to/from SP)* on page C6-1113

C3.4.7 Shift (register)

In the Shift (register) instructions, the shift amount is the positive value in the second source register modulo the register size. The register width of the instruction controls whether the new bits are fed into the result on a right shift or rotate at bit[63] or bit[31].

Table C3-56 shows the Shift (register) instructions.

Table C3-56 Shift (register) instructions

Mnemonic	Instruction	See
ASRV	Arithmetic shift right variable	*ASRV* on page C6-809

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-56 Shift (register) instructions (continued)

Mnemonic	Instruction	See
LSLV	Logical shift left variable	*LSLV* on page C6-1101
LSRV	Logical shift right variable	*LSRV* on page C6-1107
RORV	Rotate right variable	*RORV* on page C6-1186

However, the Shift (register) instructions have a preferred set of aliases that match the shift immediate aliases described in *Shift (immediate)* on page C3-229.

Table C3-57 shows the aliases for Shift (register) instructions.

Table C3-57 Aliases for Variable shift instructions

Mnemonic	Instruction	See
ASR	Arithmetic shift right	*ASR (register)* on page C6-805
LSL	Logical shift left	*LSL (register)* on page C6-1097
LSR	Logical shift right	*LSR (register)* on page C6-1103
ROR	Rotate right	*ROR (register)* on page C6-1184

C3.4.8 Multiply and divide

This section describes the instructions used for integer multiplication and division. It contains the following subsections:

- *Multiply*.
- *Divide* on page C3-236.

Multiply

The Multiply instructions write to a single 32-bit or 64-bit destination register, and are built around the fundamental four operand multiply-add and multiply-subtract operation, together with 32-bit to 64-bit widening variants. A 64-bit to 128-bit widening multiple can be constructed with two instructions, using SMULH or UMULH to generate the upper 64 bits. Table C3-58 shows the Multiply instructions.

Table C3-58 Multiply integer instructions

Mnemonic	Instruction	See
MADD	Multiply-add	*MADD* on page C6-1109
MSUB	Multiply-subtract	*MSUB* on page C6-1133
MNEG	Multiply-negate	*MNEG* on page C6-1111
MUL	Multiply	*MUL* on page C6-1135
SMADDL	Signed multiply-add long	*SMADDL* on page C6-1203
SMSUBL	Signed multiply-subtract long	*SMSUBL* on page C6-1207
SMNEGL	Signed multiply-negate long	*SMNEGL* on page C6-1206
SMULL	Signed multiply long	*SMULL* on page C6-1210
SMULH	Signed multiply high	*SMULH* on page C6-1209

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-58 Multiply integer instructions (continued)

Mnemonic	Instruction	See
UMADDL	Unsigned multiply-add long	*UMADDL* on page C6-1382
UMSUBL	Unsigned multiply-subtract long	*UMSUBL* on page C6-1385
UMNEGL	Unsigned multiply-negate long	*UMNEGL* on page C6-1384
UMULL	Unsigned multiply long	*UMULL* on page C6-1388
UMULH	Unsigned multiply high	*UMULH* on page C6-1387

Divide

The Divide instructions compute the quotient of a division, rounded towards zero. The remainder can then be computed as (numerator - (quotient × denominator)), using the MSUB instruction.

If a signed integer division (INT_MIN / -1) is performed where INT_MIN is the most negative integer value representable in the selected register size, then the result overflows the signed integer range. No indication of this overflow is produced and the result that is written to the destination register is INT_MIN.

A division by zero results in a zero being written to the destination register, without any indication that the division by zero occurred.

Table C3-59 shows the Divide instructions.

Table C3-59 Divide instructions

Mnemonic	Instruction	See
SDIV	Signed divide	*SDIV* on page C6-1199
UDIV	Unsigned divide	*UDIV* on page C6-1381

C3.4.9 CRC32

The CRC32 instructions operate on the general-purpose register file to update a 32-bit CRC value from an input value comprising 1, 2, 4, or 8 bytes. There are two different classes of CRC instructions, CRC32, and CRC32C, that support two commonly used 32-bit polynomials, known as CRC-32 and CRC-32C.

To fit with common usage, the bit order of the values is reversed as part of the operation.

When bits[19:16] of ID_AA64ISAR0_EL1 are set to 0b0001, the CRC instructions are implemented.

These instructions are optional in an Armv8.0 implementation.

All implementations of Armv8.1 architecture and later are required to implement the CRC32 instructions.

Table C3-60 shows the CRC instructions.

Table C3-60 CRC32 instructions

Mnemonic	Instruction	See
CRC32B	CRC-32 sum from byte	*CRC32B, CRC32H, CRC32W, CRC32X* on page C6-888
CRC32H	CRC-32 sum from halfword	*CRC32B, CRC32H, CRC32W, CRC32X* on page C6-888
CRC32W	CRC-32 sum from word	*CRC32B, CRC32H, CRC32W, CRC32X* on page C6-888
CRC32X	CRC-32 sum from doubleword	*CRC32B, CRC32H, CRC32W, CRC32X* on page C6-888

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-60 CRC32 instructions (continued)

Mnemonic	Instruction	See
CRC32CB	CRC-32C sum from byte	*CRC32CB, CRC32CH, CRC32CW, CRC32CX* on page C6-890
CRC32CH	CRC-32C sum from halfword	*CRC32CB, CRC32CH, CRC32CW, CRC32CX* on page C6-890
CRC32CW	CRC-32C sum from word	*CRC32CB, CRC32CH, CRC32CW, CRC32CX* on page C6-890
CRC32CX	CRC-32C sum from doubleword	*CRC32CB, CRC32CH, CRC32CW, CRC32CX* on page C6-890

C3.4.10 Bit operation

Table C3-61 shows the Bit operation instructions.

Table C3-61 Bit operation instructions

Mnemonic	Instruction	See
CLS	Count leading sign bits	*CLS* on page C6-870
CLZ	Count leading zero bits	*CLZ* on page C6-871
RBIT	Reverse bit order	*RBIT* on page C6-1171
REV	Reverse bytes in register	*REV* on page C6-1174
REV16	Reverse bytes in halfwords	*REV16* on page C6-1176
REV32	Reverses bytes in words	*REV32* on page C6-1178
REV64	Reverse bytes in register	*REV64* on page C6-1180

C3.4.11 Conditional select

The Conditional select instructions select between the first or second source register, depending on the current state of the Condition flags. When the named condition is true, the first source register is selected and its value is copied without modification to the destination register. When the condition is false the second source register is selected and its value might be optionally inverted, negated, or incremented by one, before writing to the destination register.

Other useful conditional set and conditional unary operations are implemented as aliases of the four Conditional select instructions.

Table C3-62 shows the Conditional select instructions.

Table C3-62 Conditional select instructions

Mnemonic	Instruction	See
CSEL	Conditional select	*CSEL* on page C6-893
CSINC	Conditional select increment	*CSINC* on page C6-899
CSINV	Conditional select inversion	*CSINV* on page C6-901
CSNEG	Conditional select negation	*CSNEG* on page C6-903
CSET	Conditional set	*CSET* on page C6-895
CSETM	Conditional set mask	*CSETM* on page C6-897

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-62 Conditional select instructions (continued)

Mnemonic	Instruction	See
CINC	Conditional increment	*CINC* on page C6-865
CINV	Conditional invert	*CINV* on page C6-867
CNEG	Conditional negate	*CNEG* on page C6-885

C3.4.12 Conditional comparison

The Conditional comparison instructions provide a conditional select for the NZCV Condition flags, setting the flags to the result of an arithmetic comparison of its two source register values if the named input condition is true, or to an immediate value if the input condition is false. There are register and immediate forms. The immediate form compares the source register to a small 5-bit unsigned value.

Table C3-63 shows the Conditional comparison instructions.

Table C3-63 Conditional comparison instructions

Mnemonic	Instruction	See
CCMN	Conditional compare negative (register)	*CCMN (register)* on page C6-857
CCMN	Conditional compare negative (immediate)	*CCMN (immediate)* on page C6-855
CCMP	Conditional compare (register)	*CCMP (register)* on page C6-861
CCMP	Conditional compare (immediate)	*CCMP (immediate)* on page C6-859

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C3.5 Data processing - SIMD and floating-point

This section describes the instruction groups for data processing with SIMD and floating-point register operands.

Common features of SIMD instructions gives general information about SIMD instructions.

The following subsections describe the scalar floating-point data processing instructions:

- *Floating-point move (register)* on page C3-240.
- *Floating-point move (immediate)* on page C3-240.
- *Floating-point conversion* on page C3-241.
- *Floating-point round to integral value* on page C3-242.
- *Floating-point multiply-add* on page C3-244.
- *Floating-point arithmetic (one source)* on page C3-244.
- *Floating-point arithmetic (two sources)* on page C3-244.
- *Floating-point minimum and maximum* on page C3-245.
- *Floating-point comparison* on page C3-245.
- *Floating-point conditional select* on page C3-246.
- *BFloat16 floating-point instructions* on page C3-246.

The following subsections describe the SIMD data processing instructions:

- *SIMD move* on page C3-246
- *SIMD arithmetic* on page C3-247.
- *SIMD compare* on page C3-250.
- *SIMD widening and narrowing arithmetic* on page C3-250.
- *SIMD table lookup* on page C3-259.
- *SIMD by element arithmetic* on page C3-253.
- *SIMD permute* on page C3-255.
- *SIMD immediate* on page C3-255.
- *SIMD shift (immediate)* on page C3-255.
- *SIMD floating-point and integer conversion* on page C3-256.
- *SIMD reduce (across vector lanes)* on page C3-257.
- *SIMD pairwise arithmetic* on page C3-258.
- *SIMD dot product* on page C3-259.
- *SIMD table lookup* on page C3-259.
- *SIMD complex number arithmetic* on page C3-260.
- *SIMD BFloat16* on page C3-260.
- *SIMD matrix multiplication* on page C3-261.
- *The Cryptographic Extension* on page C3-261.

For information about the encoding structure of the instructions in this instruction group, see *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

For information about the floating-point exceptions, see *Floating-point exceptions and exception traps* on page D1-2360.

C3.5.1 Common features of SIMD instructions

A number of SIMD instructions come in three forms:

Wide Indicated by the suffix *W*. The element width of the destination register and the first source operand is double that of the second source operand.

Long Indicated by the suffix *L*. The element width of the destination register is double that of both source operands.

Narrow Indicated by the suffix *N*. The element width of the destination register is half that of both source operands.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In addition, each vector form of the instruction is part of a pair, with a second and upper half suffix of *2*, to identify the variant of the instruction:

- Where a SIMD operation widens or lengthens a 64-bit vector to a 128-bit vector, the instruction provides a second part operation that can extract the source from the upper 64 bits of the source registers.
- Where a SIMD operation narrows a 128-bit vector to a 64-bit vector, the instruction provides a second-part operation that can pack the result of a second operation into the upper part of the same destination register.

Note

This is referred to as a *lane set specifier*.

C3.5.2 Floating-point move (register)

The Floating-point move (register) instructions copy a scalar floating-point value from one register to another register without performing any conversion.

Some of the Floating-point move (register) instructions overlap with the functionality provided by the Advanced SIMD instructions `DUP`, `INS`, and `UMOV`. However, Arm recommends using the `FMOV` instructions when operating on scalar floating-point data to avoid the creation of scalar floating-point code that depends on the availability of the Advanced SIMD instruction set.

Table C3-64 shows the Floating-point move (register) instructions.

Table C3-64 Floating-point move (register) instructions

Mnemonic	Instruction	See
`FMOV`	Floating-point move register without conversion	*FMOV (register)* on page C7-1686
	Floating-point move to or from general-purpose register without conversion	*FMOV (general)* on page C7-1688

C3.5.3 Floating-point move (immediate)

The Floating-point move (immediate) instructions convert a small constant immediate floating-point value into a half-precision, single-precision, or double-precision scalar floating-point value in a SIMD and floating-point register.

The floating-point constant can be specified either in decimal notation, such as 12.0 or -1.2e1, or as a string beginning with `0x` followed by a hexadecimal representation of the IEEE 754 half-precision, single-precision, or double-precision encoding. Arm recommends that a disassembler uses the decimal notation, provided that this displays the value precisely.

Note

When ARMv8.2-FP16 is not implemented, the only half-precision instructions that are supported are floating-point conversions between half-precision, single-precision, and double-precision.

The floating-point value must be expressible as ($\pm n/16 \times 2^r$), where n is an integer in the range $16 \le n \le 31$ and r is an integer in the range of $-3 \le r \le 4$, that is a normalized binary floating-point encoding with one sign bit, four bits of fraction, and a 3-bit exponent.

Table C3-65 shows the Floating-point move (immediate) instruction:

Table C3-65 Floating-point move (immediate) instruction

Mnemonic	Instruction	See
`FMOV`	Floating-point move immediate	*FMOV (scalar, immediate)* on page C7-1691

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C3.5.4 Floating-point conversion

The following subsections describe the conversion of floating-point values:

- *Convert floating-point precision*.
- *Convert between floating-point and integer or fixed-point*.

Convert floating-point precision

These instructions convert a floating-point scalar with one precision to a floating-point scalar with a different precision, using the current rounding mode as specified by FPCR.RMode.

Table C3-66 shows the Floating-point precision conversion instruction.

Table C3-66 Floating-point precision conversion instruction

Mnemonic	Instruction	See
FCVT	Floating-point convert precision (scalar)	*FCVT* on page C7-1549

Convert between floating-point and integer or fixed-point

These instructions convert a floating-point scalar in a SIMD and floating-point register to or from a signed or unsigned integer or fixed-point value in a general-purpose register. For a fixed-point value, a final immediate operand indicates that the general-purpose register holds a fixed-point number and `fbits` indicates the number of bits after the binary point. `fbits` is in the range 1- 32 inclusive for a 32-bit general-purpose register name, and 1-64 inclusive for a 64-bit general-purpose register name.

These instructions can cause the following floating-point exceptions:

Invalid Operation

Occurs if the floating-point input is a NaN, infinity, or a numerical value that cannot be represented in the destination register. An out of range integer or fixed-point result is saturated to the size of the destination register.

Inexact

Occurs if the numeric result that differs from the input value.

Input Denormal

As *Flush-to-zero* on page A1-55 describes, when Flush-to-zero mode is enabled, occurs when zero replaces a double-precision or single-precision denormal input.

Note

When ARMv8.2-FP16 is implemented, a half-precision denormal input that is flushed to zero does not generate an Input Denormal exception.

Table C3-67 shows the Floating-point and fixed-point conversion instructions.

Table C3-67 Floating-point and integer or fixed-point conversion instructions

Mnemonic	Instruction	See
FCVTAS	Floating-point scalar convert to signed integer, rounding to nearest with ties to away (scalar form)	*FCVTAS (scalar)* on page C7-1554
FCVTAU	Floating-point scalar convert to unsigned integer, rounding to nearest with ties to away (scalar form)	*FCVTAU (scalar)* on page C7-1559
FCVTMS	Floating-point scalar convert to signed integer, rounding toward minus infinity (scalar form)	*FCVTMS (scalar)* on page C7-1566

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-67 Floating-point and integer or fixed-point conversion instructions (continued)

Mnemonic	Instruction	See
FCVTMU	Floating-point scalar convert to unsigned integer, rounding toward minus infinity (scalar form)	*FCVTMU (scalar)* on page C7-1571
FCVTNS	Floating-point scalar convert to signed integer, rounding to nearest with ties to even (scalar form)	*FCVTNS (scalar)* on page C7-1578
FCVTNU	Floating-point scalar convert to unsigned integer, rounding to nearest with ties to even (scalar form)	*FCVTNU (scalar)* on page C7-1583
FCVTPS	Floating-point scalar convert to signed integer, rounding toward positive infinity (scalar form)	*FCVTPS (scalar)* on page C7-1588
FCVTPU	Floating-point scalar convert to unsigned integer, rounding toward positive infinity (scalar form)	*FCVTPU (scalar)* on page C7-1593
FCVTZS	Floating-point scalar convert to signed integer, rounding toward zero (scalar form)	*FCVTZS (scalar, integer)* on page C7-1605
	Floating-point convert to signed fixed-point, rounding toward zero (scalar form)	*FCVTZS (scalar, fixed-point)* on page C7-1603
FCVTZU	Floating-point scalar convert to unsigned integer, rounding toward zero (scalar form)	*FCVTZU (scalar, integer)* on page C7-1615
	Floating-point scalar convert to unsigned fixed-point, rounding toward zero (scalar form)	*FCVTZU (scalar, fixed-point)* on page C7-1613
FJCVTZS	Floating-point Javascript convert to signed fixed-point, rounding toward zero	*FJCVTZS* on page C7-1621
SCVTF	Signed integer scalar convert to floating-point, using the current rounding mode (scalar form)	*SCVTF (scalar, integer)* on page C7-1931
	Signed fixed-point convert to floating-point, using the current rounding mode (scalar form)	*SCVTF (scalar, fixed-point)* on page C7-1929
UCVTF	Unsigned integer scalar convert to floating-point, using the current rounding mode (scalar form)	*UCVTF (scalar, integer)* on page C7-2205
	Unsigned fixed-point convert to floating-point, using the current rounding mode (scalar form)	*UCVTF (scalar, fixed-point)* on page C7-2203

C3.5.5 Floating-point round to integral value

The following subsections describe instructions which round a floating-point number to an integral valued floating-point number in the same format:

- *Floating-point round to an integer of the same size as the register*
- *Floating-point round to 32-bit or 64-bit integer* on page C3-243

Floating-point round to an integer of the same size as the register

The following instructions round a floating-point value to an integer floating-point value of the same size.

For these instructions:

- A zero input gives a zero result with the same sign.
- An infinite input gives an infinite result with the same sign.
- A NaN is propagated as in normal floating-point arithmetic.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

These instructions can cause the following floating-point exceptions:

Invalid Operation

Occurs in response to a floating-point input of a signaling NaN.

Inexact, `FRINTX` instruction only

Occurs if the result is numeric and does not have the same numerical value as the input.

Input Denormal

As *Flush-to-zero* on page A1-55 describes, when Flush-to-zero mode is enabled, occurs when zero replaces a double-precision or single-precision denormal input.

Note

When ARMv8.2-FP16 is implemented, a half-precision denormal input that is flushed to zero does not generate an Input Denormal exception.

Table C3-68 shows the Floating-point round to integer instructions.

Table C3-68 Floating-point round to integer instructions

Mnemonic	Instruction	See
`FRINTA`	Floating-point round to integer, to nearest with ties to away	*FRINTA (scalar)* on page C7-1746
`FRINTI`	Floating-point round to integer, using current rounding mode	*FRINTI (scalar)* on page C7-1750
`FRINTM`	Floating-point round to integer, toward minus infinity	*FRINTM (scalar)* on page C7-1754
`FRINTN`	Floating-point round to integer, to nearest with ties to even	*FRINTN (scalar)* on page C7-1758
`FRINTP`	Floating-point round to integer, toward positive infinity	*FRINTP (scalar)* on page C7-1762
`FRINTX`	Floating-point round to integer exact, using current rounding mode	*FRINTX (scalar)* on page C7-1766
`FRINTZ`	Floating-point round to integer, toward zero	*FRINTZ (scalar)* on page C7-1770

Floating-point round to 32-bit or 64-bit integer

The following instructions are present if ARMv8.5-FRINT is implemented, The instructions round to a value that fits in a 32-bit integer or a 64-bit integer size, and use either round towards zero or the ambient rounding model.

Invalid Operation

Forced to be the most negative integer representable in the target size, and occurs in response to a floating-point input of a signaling NaN, an infinite input, or an out of range input.

Inexact

Occurs if the result is numeric and does not have the same numerical value as the input.

Input Denormal

As *Flush-to-zero* on page A1-55 describes, when Flush-to-zero mode is enabled, occurs when zero replaces a double-precision or single-precision denormal input.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-69 shows the Floating-point round to 32-bit or 64-bit integer instructions.

Table C3-69 Floating-point round to integer instructions

Mnemonic	Instruction	See
FRINT32X	Floating-point round to 32-bit integer, using current rounding model	*FRINT32X (scalar)* on page C7-1730
FRINT32Z	Floating-point round to 32-bit integer, toward zero	*FRINT32Z (scalar)* on page C7-1734
FRINT64X	Floating point round to 64-bit integer using current rounding model	*FRINT64X (scalar)* on page C7-1738
FRINT64Z	Floating point round to 64-bit integer, toward zero	*FRINT64Z (scalar)* on page C7-1742

C3.5.6 Floating-point multiply-add

Table C3-70 shows the Floating-point multiply-add instructions that require three source register operands.

Table C3-70 Floating-point multiply-add instructions

Mnemonic	Instruction	See
FMADD	Floating-point scalar fused multiply-add	*FMADD* on page C7-1622
FMSUB	Floating-point scalar fused multiply-subtract	*FMSUB* on page C7-1693
FNMADD	Floating-point scalar negated fused multiply-add	*FNMADD* on page C7-1714
FNMSUB	Floating-point scalar negated fused multiply-subtract	*FNMSUB* on page C7-1716

C3.5.7 Floating-point arithmetic (one source)

Table C3-71 shows the Floating-point arithmetic instructions that require a single source register operand.

Table C3-71 Floating-point arithmetic instructions with one source register

Mnemonic	Instructions	See
FABS	Floating-point scalar absolute value	*FABS (scalar)* on page C7-1487
FNEG	Floating-point scalar negate	*FNEG (scalar)* on page C7-1712
FSQRT	Floating-point scalar square root	*FSQRT (scalar)* on page C7-1780

C3.5.8 Floating-point arithmetic (two sources)

Table C3-72 shows the Floating-point arithmetic instructions that require two source register operands.

Table C3-72 Floating-point arithmetic instructions with two source registers

Mnemonic	Instruction	See
FADD	Floating-point scalar add	*FADD (scalar)* on page C7-1499
FDIV	Floating-point scalar divide	*FDIV (scalar)* on page C7-1619
FMUL	Floating-point scalar multiply	*FMUL (scalar)* on page C7-1701
FNMUL	Floating-point scalar multiply-negate	*FNMUL (scalar)* on page C7-1718
FSUB	Floating-point scalar subtract	*FSUB (scalar)* on page C7-1784

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C3.5.9 Floating-point minimum and maximum

The `min(x,y)` and `max(x,y)` operations return a quiet NaN when either *x* or *y* is NaN.

As described in *Flush-to-zero* on page A1-55, in flush-to-zero mode, denormal operands are flushed to zero before comparison, and if the result of the comparison is the flushed value, then a zero value is returned. Where both *x* and *y* are zero, or denormal values flushed to zero, with different signs, then +0.0 is returned by `max()` and -0.0 by `min()`.

The `minNum(x,y)` and `maxNum(x,y)` operations follow the IEEE 754-2008 standard and return the numerical operand when one operand is numerical and the other a quiet NaN. Apart from this additional handling of a single quiet NaN the result is then identical to `min(x,y)` and `max(x,y)`.

Table C3-73 shows the Floating-point instructions that can perform floating-point minimum and maximum operations.

Table C3-73 Floating-point minimum and maximum instructions

Mnemonic	Instruction	See
`FMAX`	Floating-point scalar maximum	*FMAX (scalar)* on page C7-1626
`FMAXNM`	Floating-point scalar maximum number	*FMAXNM (scalar)* on page C7-1630
`FMIN`	Floating-point scalar minimum	*FMIN (scalar)* on page C7-1646
`FMINNM`	Floating-point scalar minimum number	*FMINNM (scalar)* on page C7-1650

C3.5.10 Floating-point comparison

These instructions set the NZCV Condition flags in PSTATE, based on the result of a comparison of two operands. If the floating-point comparisons are *unordered*, where one or both operands are a form of NaN, the C and V bits are set to 1 and the N and Z bits are cleared to 0.

Note

The NZCV flags in the FPSR are associated with AArch32 state. The A64 floating-point comparison instructions do not change the Condition flags in the FPSR.

For the conditional Floating-point comparison instructions, if the condition is TRUE, the flags are updated to the result of the comparison, otherwise the flags are updated to the immediate value that is defined in the instruction encoding.

The quiet compare instructions generate an Invalid Operation floating-point exception if either of the source operands is a signaling NaN. The signaling compare instructions generate an Invalid Operation floating-point exception if either of the source operands is any type of NaN.

Note

If ARMv8.5-CondM is implemented, instructions AXFLAG and XAFLAG convert between the PSTATE condition flag format used by the FCMP instruction and an alternative format. See *ARMv8.4-CondM* on page A2-85 for more information.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-74 shows the Floating-point comparison instructions.

Table C3-74 Floating-point comparison instructions

Mnemonic	Instruction	See
FCMP	Floating-point quiet compare	*FCMP* on page C7-1543
FCMPE	Floating-point signaling compare	*FCMPE* on page C7-1545
FCCMP	Floating-point conditional quiet compare	*FCCMP* on page C7-1507
FCCMPE	Floating-point conditional signaling compare	*FCCMPE* on page C7-1509

C3.5.11 Floating-point conditional select

Table C3-75 shows the Floating-point conditional select instructions.

Table C3-75 Floating-point conditional select instructions

Mnemonic	Instruction	See
FCSEL	Floating-point scalar conditional select	*FCSEL* on page C7-1547

C3.5.12 BFloat16 floating-point instructions

The BFloat16 floating-point instructions are provided by ARMv8.2-BF16. The instructions to convert single-precision floating-point values to BF16 format give a more accurate conversion than a simple truncation of F32 to BF16 by removing the least significant 16 bits of the fraction. They also honour the settings of FPCR.

Table C3-76 shows these instructions.

Table C3-76 BFloat16 floating-point instructions

Mnemonic	Instruction	See
BFCVT	BFloat16 floating-point convert from single-precision to BFloat16 format (scalar)	*BFCVT* on page C7-1419

C3.5.13 SIMD move

The functionality of some data movement instructions overlaps with that provided by the scalar floating-point FMOV instructions described in *Floating-point move (register)* on page C3-240.

Table C3-77 shows the SIMD move instructions.

Table C3-77 SIMD move instructions

Mnemonic	Instruction	See
DUP	Duplicate vector element to vector or scalar	*DUP (element)* on page C7-1472
DUP	Duplicate general-purpose register to vector	*DUP (general)* on page C7-1475
INS[a]	Insert vector element from another vector element	*INS (element)* on page C7-1786
	Insert vector element from general-purpose register	*INS (general)* on page C7-1788

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-77 SIMD move instructions (continued)

Mnemonic	Instruction	See
MOV	Move vector element to vector element	*MOV (element)* on page C7-1858
	Move general-purpose register to vector element	*MOV (from general)* on page C7-1860
	Move vector element to scalar	*MOV (scalar)* on page C7-1856
	Move vector element to general-purpose register	*MOV (to general)* on page C7-1863
UMOV	Unsigned move vector element to general-purpose register	*UMOV* on page C7-2238
SMOV	Signed move vector element to general-purpose register	*SMOV* on page C7-2009

a. Disassembles as MOV.

C3.5.14 SIMD arithmetic

Table C3-78 shows the SIMD arithmetic instructions.

Table C3-78 SIMD arithmetic instructions

Mnemonic	Instruction	See
ADD	Add (vector and scalar form)	*ADD (vector)* on page C7-1403
AND	Bitwise AND (vector form)	*AND (vector)* on page C7-1417
BIC	Bitwise bit clear (register) (vector form)	*BIC (vector, register)* on page C7-1432
BIF	Bitwise insert if false (vector form)	*BIF* on page C7-1434
BIT	Bitwise insert if true (vector form)	*BIT* on page C7-1436
BSL	Bitwise select (vector form)	*BSL* on page C7-1438
EOR	Bitwise exclusive OR (vector form)	*EOR (vector)* on page C7-1477
FABD	Floating-point absolute difference (vector and scalar form)	*FABD* on page C7-1482
FADD	Floating-point add (vector form)	*FADD (vector)* on page C7-1497
FDIV	Floating-point divide (vector form)	*FDIV (vector)* on page C7-1617
FMAX	Floating-point maximum (vector form)	*FMAXP (vector)* on page C7-1640
FMAXNM	Floating-point maximum number (vector form)	*FMAXNM (vector)* on page C7-1628
FMIN	Floating-point minimum (vector form)	*FMIN (vector)* on page C7-1644
FMINNM	Floating-point minimum number (vector form)	*FMINNM (vector)* on page C7-1648
FMLA	Floating-point fused multiply-add (vector form)	*FMLA (vector)* on page C7-1668
FMLAL, FMLAL2	Floating-point fused multiply-add long (vector form)	*FMLAL, FMLAL2 (vector)* on page C7-1672
FMLS	Floating-point fused multiply-subtract (vector form)	*FMLS (vector)* on page C7-1678
FMLSL, FMLSL2	Floating-point fused multiply-subtract long (vector form)	*FMLSL, FMLSL2 (vector)* on page C7-1682
FMUL	Floating-point multiply (vector form)	*FMUL (vector)* on page C7-1699
FMULX	Floating-point multiply extended (vector and scalar form)	*FMULX* on page C7-1707

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-78 SIMD arithmetic instructions (continued)

Mnemonic	Instruction	See
FRECPS	Floating-point reciprocal step (vector and scalar form)	*FRECPS* on page C7-1723
FRSQRTS	Floating-point reciprocal square root step (vector and scalar form)	*FRSQRTS* on page C7-1775
FSUB	Floating-point subtract (vector form)	*FSUB (vector)* on page C7-1782
MLA	Multiply-add (vector form)	*MLA (vector)* on page C7-1850
MLS	Multiply-subtract (vector form)	*MLS (vector)* on page C7-1854
MUL	Multiply (vector form)	*MUL (vector)* on page C7-1870
MOV	Move vector register (vector form)	*MOV (vector)* on page C7-1862
ORN	Bitwise inclusive OR NOT (vector form)	*ORN (vector)* on page C7-1880
ORR	Bitwise inclusive OR (register) (vector form)	*ORR (vector, register)* on page C7-1884
PMUL	Polynomial multiply (vector form)	*PMUL* on page C7-1886
SABA	Signed absolute difference and accumulate (vector form)	*SABA* on page C7-1905
SABD	Signed absolute difference (vector form)	*SABD* on page C7-1909
SHADD	Signed halving add (vector form)	*SHADD* on page C7-1955
SHSUB	Signed halving subtract (vector form)	*SHSUB* on page C7-1963
SMAX	Signed maximum (vector form)	*SMAX* on page C7-1986
SMIN	Signed minimum (vector form)	*SMIN* on page C7-1992
SQADD	Signed saturating add (vector and scalar form)	*SQADD* on page C7-2018
SQDMULH	Signed saturating doubling multiply returning high half (vector and scalar form)	*SQDMULH (vector)* on page C7-2037
SQRSHL	Signed saturating rounding shift left (register) (vector and scalar form)	*SQRSHL* on page C7-2062
SQRDMLAH	Signed saturating rounding doubling multiply accumulate returning high half	*SQRDMLAH (vector)* on page C7-2050
SQRDMLSH	Signed saturating rounding doubling multiply subtract returning high half	*SQRDMLSH (vector)* on page C7-2055
SQRDMULH	Signed saturating rounding doubling multiply returning high half (vector and scalar form)	*SQRDMULH (vector)* on page C7-2060
SQSHL	Signed saturating shift left (register) (vector and scalar form)	*SQSHL (register)* on page C7-2073
SQSUB	Signed saturating subtract (vector and scalar form)	*SQSUB* on page C7-2084
SRHADD	Signed rounding halving add (vector form)	*SRHADD* on page C7-2092
SRSHL	Signed rounding shift left (register) (vector and scalar form)	*SRSHL* on page C7-2097
SSHL	Signed shift left (register) (vector and scalar form)	*SSHL* on page C7-2103
SUB	Subtract (vector and scalar form)	*SUB (vector)* on page C7-2161
UABA	Unsigned absolute difference and accumulate (vector form)	*UABA* on page C7-2179

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-78 SIMD arithmetic instructions (continued)

Mnemonic	Instruction	See
UABD	Unsigned absolute difference (vector form)	*UABD* on page C7-2183
UHADD	Unsigned halving add (vector form)	*UHADD* on page C7-2211
UHSUB	Unsigned halving subtract (vector form)	*UHSUB* on page C7-2213
UMAX	Unsigned maximum (vector form)	*UMAX* on page C7-2215
UMIN	Unsigned minimum (vector form)	*UMIN* on page C7-2221
UQADD	Unsigned saturating add (vector and scalar form)	*UQADD* on page C7-2245
UQRSHL	Unsigned saturating rounding shift left (register) (vector and scalar form)	*UQRSHL* on page C7-2247
UQSHL	Unsigned saturating shift left (register) (vector and scalar form)	*UQSHL (register)* on page C7-2255
UQSUB	Unsigned saturating subtract (vector and scalar form)	*UQSUB* on page C7-2260
URHADD	Unsigned rounding halving add (vector form)	*URHADD* on page C7-2266
URSHL	Unsigned rounding shift left (register) (vector and scalar form)	*URSHL* on page C7-2268
USHL	Unsigned shift left (register) (vector and scalar form)	*USHL* on page C7-2279

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C3.5.15 SIMD compare

The SIMD compare instructions compare vector or scalar elements according to the specified condition and set the destination vector element to all ones if the condition holds, or to zero if the condition does not hold.

Note

Some of the comparisons, such as LS, LE, LO, and LT, can be made by reversing the operands and using the opposite comparison, HS, GE, HI, or GT.

Table C3-79 shows that SIMD compare instructions.

Table C3-79 SIMD compare instructions

Mnemonic	Instruction	See
CMEQ	Compare bitwise equal (vector and scalar form)	*CMEQ (register)* on page C7-1444
	Compare bitwise equal to zero (vector and scalar form)	*CMEQ (zero)* on page C7-1446
CMHS	Compare unsigned higher or same (vector and scalar form)	*CMHS (register)* on page C7-1461
CMGE	Compare signed greater than or equal (vector and scalar form)	*CMGE (register)* on page C7-1449
	Compare signed greater than or equal to zero (vector and scalar form)	*CMGE (zero)* on page C7-1451
CMHI	Compare unsigned higher (vector and scalar form)	*CMHI (register)* on page C7-1459
CMGT	Compare signed greater than (vector and scalar form)	*CMGT (register)* on page C7-1454
	Compare signed greater than zero (vector and scalar form)	*CMGT (zero)* on page C7-1456
CMLE	Compare signed less than or equal to zero (vector and scalar form)	*CMLE (zero)* on page C7-1463
CMLT	Compare signed less than zero (vector and scalar form)	*CMLT (zero)* on page C7-1466
CMTST	Compare bitwise test bits nonzero (vector and scalar form)	*CMTST* on page C7-1468
FCMEQ	Floating-point compare equal (vector and scalar form)	*FCMEQ (register)* on page C7-1511
	Floating-point compare equal to zero (vector and scalar form)	*FCMEQ (zero)* on page C7-1515
FCMGE	Floating-point compare greater than or equal (vector and scalar form)	*FCMGE (register)* on page C7-1518
	Floating-point compare greater than or equal to zero (vector and scalar form)	*FCMGE (zero)* on page C7-1522
FCMGT	Floating-point compare greater than (vector and scalar form)	*FCMGT (register)* on page C7-1525
	Floating-point compare greater than zero (vector and scalar form)	*FCMGT (zero)* on page C7-1529
FCMLE	Floating-point compare less than or equal to zero (vector and scalar form)	*FCMLE (zero)* on page C7-1537
FCMLT	Floating-point compare less than zero (vector and scalar form)	*FCMLT (zero)* on page C7-1540
FACGE	Floating-point absolute compare greater than or equal (vector and scalar form)	*FACGE* on page C7-1489
FACGT	Floating-point absolute compare greater than (vector and scalar form)	*FACGT* on page C7-1493

C3.5.16 SIMD widening and narrowing arithmetic

For information about the variants of these instructions, see *Common features of SIMD instructions* on page C3-239.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-80 shows the SIMD widening and narrowing arithmetic instructions.

Table C3-80 SIMD widening and narrowing arithmetic instructions

Mnemonic	Instruction	See
ADDHN, ADDHN2	Add returning high, narrow (vector form)	*ADDHN, ADDHN2* on page C7-1405
PMULL, PMULL2	Polynomial multiply long (vector form)	*PMULL, PMULL2* on page C7-1888 See also *The Cryptographic Extension* on page C3-261
RADDHN, RADDHN2	Rounding add returning high, narrow (vector form)	*RADDHN, RADDHN2* on page C7-1890
RSUBHN, RSUBHN2	Rounding subtract returning high, narrow (vector form)	*RSUBHN, RSUBHN2* on page C7-1903
SABAL, SABAL2	Signed absolute difference and accumulate long (vector form)	*SABAL, SABAL2* on page C7-1907
SABDL, SABDL2	Signed absolute difference long (vector form)	*SABDL, SABDL2* on page C7-1911
SADDL, SADDL2	Signed add long (vector form)	*SADDL, SADDL2* on page C7-1915
SADDW, SADDW2	Signed add wide (vector form)	*SADDW, SADDW2* on page C7-1921
SMLAL, SMLAL2	Signed multiply-add long (vector form)	*SMLAL, SMLAL2 (vector)* on page C7-2001
SMLSL, SMLSL2	Signed multiply-subtract long (vector form)	*SMLSL, SMLSL2 (vector)* on page C7-2006
SMULL, SMULL2	Signed multiply long (vector form)	*SMULL, SMULL2 (vector)* on page C7-2014
SQDMLAL, SQDMLAL2	Signed saturating doubling multiply-add long (vector and scalar form)	*SQDMLAL, SQDMLAL2 (vector)* on page C7-2024
SQDMLSL, SQDMLSL2	Signed saturating doubling multiply-subtract long (vector and scalar form)	*SQDMLSL, SQDMLSL2 (vector)* on page C7-2031
SQDMULL, SQDMULL2	Signed saturating doubling multiply long (vector and scalar form)	*SQDMULL, SQDMULL2 (vector)* on page C7-2042
SSUBL, SSUBL2	Signed subtract long (vector form)	*SSUBL, SSUBL2* on page C7-2114
SSUBW, SSUBW2	Signed subtract wide (vector form)	*SSUBW, SSUBW2* on page C7-2116
SUBHN, SUBHN2	Subtract returning high, narrow (vector form)	*SUBHN, SUBHN2* on page C7-2163
UABAL, UABAL2	Unsigned absolute difference and accumulate long (vector form)	*UABAL, UABAL2* on page C7-2181
UABDL, UABDL2	Unsigned absolute difference long (vector form)	*UABDL, UABDL2* on page C7-2185
UADDL, UADDL2	Unsigned add long (vector form)	*UADDL, UADDL2* on page C7-2189
UADDW, UADDW2	Unsigned add wide (vector form)	*UADDW, UADDW2* on page C7-2195
UMLAL, UMLAL2	Unsigned multiply-add long (vector form)	*UMLAL, UMLAL2 (vector)* on page C7-2230
UMLSL, UMLSL2	Unsigned multiply-subtract long (vector form)	*UMLSL, UMLSL2 (vector)* on page C7-2235
UMULL, UMULL2	Unsigned multiply long (vector form)	*UMULL, UMULL2 (vector)* on page C7-2243
USUBL, USUBL2	Unsigned subtract long (vector form)	*USUBL, USUBL2* on page C7-2293
USUBW, USUBW2	Unsigned subtract wide (vector form)	*USUBW, USUBW2* on page C7-2295

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C3.5.17 SIMD unary arithmetic

For information about the variants of these instructions, see *Common features of SIMD instructions* on page C3-239.

Table C3-81 shows the SIMD unary arithmetic instructions.

Table C3-81 SIMD unary arithmetic instructions

Mnemonic	Instruction	See
ABS	Absolute value (vector and scalar form)	*ABS* on page C7-1401
CLS	Count leading sign bits (vector form)	*CLS (vector)* on page C7-1440
CLZ	Count leading zero bits (vector form)	*CLZ (vector)* on page C7-1442
CNT	Population count per byte (vector form)	*CNT* on page C7-1470
FABS	Floating-point absolute (vector form)	*FABS (vector)* on page C7-1485
FCVTL, FCVTL2	Floating-point convert to higher precision long (vector form)	*FCVTL, FCVTL2* on page C7-1561
FCVTN, FCVTN2	Floating-point convert to lower precision narrow (vector form)	*FCVTN, FCVTN2* on page C7-1573
FCVTXN, FCVTXN2	Floating-point convert to lower precision narrow, rounding to odd (vector and scalar form)	*FCVTXN, FCVTXN2* on page C7-1595
FNEG	Floating-point negate (vector form)	*FNEG (vector)* on page C7-1710
FRECPE	Floating-point reciprocal estimate (vector and scalar form)	*FRECPE* on page C7-1720
FRECPX	Floating-point reciprocal square root (scalar form)	*FRECPX* on page C7-1726
FRINT32X	Floating-point round to 32-bit integer, using current rounding mode (vector form)	*FRINT32X (vector)* on page C7-1728
FRINT32Z	Floating-point round to 32-bit integer, toward zero (vector form)	*FRINT32Z (vector)* on page C7-1732
FRINT64X	Floating-point round to 64-bit integer, using current rounding mode (vector form)	*FRINT64X (vector)* on page C7-1736
FRINT64Z	Floating-point round to 64-bit integer, toward zero (vector form)	*FRINT64Z (vector)* on page C7-1740
FRINTA	Floating-point round to integer, to nearest with ties to away (vector form)	*FRINTA (vector)* on page C7-1744
FRINTI	Floating-point round to integer, using current rounding mode (vector form)	*FRINTI (vector)* on page C7-1748
FRINTM	Floating-point round to integer, toward minus infinity (vector form)	*FRINTM (vector)* on page C7-1752
FRINTN	Floating-point round to integer, to nearest with ties to even (vector form)	*FRINTN (vector)* on page C7-1756
FRINTP	Floating-point round to integer, toward positive infinity (vector form)	*FRINTP (vector)* on page C7-1760
FRINTX	Floating-point round to integer exact, using current rounding mode (vector form)	*FRINTX (vector)* on page C7-1764
FRINTZ	Floating-point round to integer, toward zero (vector form)	*FRINTZ (vector)* on page C7-1768
FRSQRTE	Floating-point reciprocal square root estimate (vector and scalar form)	*FRSQRTE* on page C7-1772
FSQRT	Floating-point square root (vector form)	*FSQRT (vector)* on page C7-1778
MVN	Bitwise NOT (vector form)	*MVN* on page C7-1872

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-81 SIMD unary arithmetic instructions (continued)

Mnemonic	Instruction	See
NEG	Negate (vector and scalar form)	*NEG (vector)* on page C7-1876
NOT	Bitwise NOT (vector form)	*NOT* on page C7-1878
RBIT	Bitwise reverse (vector form)	*RBIT (vector)* on page C7-1893
REV16	Reverse elements in 16-bit halfwords (vector form)	*REV16 (vector)* on page C7-1895
REV32	Reverse elements in 32-bit words (vector form)	*REV32 (vector)* on page C7-1897
REV64	Reverse elements in 64-bit doublewords (vector form)	*REV64* on page C7-1899
SADALP	Signed add and accumulate long pairwise (vector form)	*SADALP* on page C7-1913
SADDLP	Signed add long pairwise (vector form)	*SADDLP* on page C7-1917
SQABS	Signed saturating absolute value (vector and scalar form)	*SQABS* on page C7-2016
SQNEG	Signed saturating negate (vector and scalar form)	*SQNEG* on page C7-2045
SQXTN, SQXTN2	Signed saturating extract narrow (vector form)	*SQXTN, SQXTN2* on page C7-2086
SQXTUN, SQXTUN2	Signed saturating extract unsigned narrow (vector and scalar form)	*SQXTUN, SQXTUN2* on page C7-2089
SUQADD	Signed saturating accumulate of unsigned value (vector and scalar form)	*SUQADD* on page C7-2167
SXTL, SXTL2	Signed extend long	*SXTL, SXTL2* on page C7-2169
UADALP	Unsigned add and accumulate long pairwise (vector form)	*UADALP* on page C7-2187
UADDLP	Unsigned add long pairwise (vector form)	*UADDLP* on page C7-2191
UQXTN, UQXTN2	Unsigned saturating extract narrow (vector form)	*UQXTN, UQXTN2* on page C7-2262
URECPE	Unsigned reciprocal estimate (vector form)	*URECPE* on page C7-2265
URSQRTE	Unsigned reciprocal square root estimate (vector form)	*URSQRTE* on page C7-2272
USQADD	Unsigned saturating accumulate of signed value (vector and scalar form)	*USQADD* on page C7-2288
UXTL, UXTL2	Unsigned extend long	*UXTL, UXTL2* on page C7-2297
XTN, XTN2	Extract narrow (vector form)	*XTN, XTN2* on page C7-2304

C3.5.18 SIMD by element arithmetic

For information about the variants of these instructions, see *Common features of SIMD instructions* on page C3-239.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-82 shows the SIMD by element arithmetic instructions.

Table C3-82 SIMD by element arithmetic instructions

Mnemonic	Instruction	See
FMLA	Floating-point fused multiply-add (vector and scalar form)	*FMLA (by element)* on page C7-1664
FMLAL, FMLAL2	Floating-point fused multiply-add long (vector form)	*FMLAL, FMLAL2 (by element)* on page C7-1670
FMLS	Floating-point fused multiply-subtract (vector and scalar form)	*FMLS (by element)* on page C7-1674.
FMLSL, FMLSL2	Floating-point fused multiply-subtract long (vector form)	*FMLSL, FMLSL2 (by element)* on page C7-1680
FMUL	Floating-point multiply (vector and scalar form)	*FMUL (by element)* on page C7-1695
FMULX	Floating-point multiply extended (vector and scalar form)	*FMULX (by element)* on page C7-1703
MLA	Multiply-add (vector form)	*MLA (by element)* on page C7-1848
MLS	Multiply-subtract (vector form)	*MLS (by element)* on page C7-1852
MUL	Multiply (vector form)	*MUL (by element)* on page C7-1868
SMLAL, SMLAL2	Signed multiply-add long (vector form)	*SMLAL, SMLAL2 (by element)* on page C7-1998
SMLSL, SMLSL2	Signed multiply-subtract long (vector form)	*SMLSL, SMLSL2 (by element)* on page C7-2003
SMULL, SMULL2	Signed multiply long (vector form)	*SMULL, SMULL2 (by element)* on page C7-2011
SQDMLAL, SQDMLAL2	Signed saturating doubling multiply-add long (vector and scalar form)	*SQDMLAL, SQDMLAL2 (by element)* on page C7-2020
SQDMLSL, SQDMLSL2	Signed saturating doubling multiply-subtract long (vector form)	*SQDMLSL, SQDMLSL2 (by element)* on page C7-2027
SQDMULH	Signed saturating doubling multiply returning high half (vector and scalar form)	*SQDMULH (by element)* on page C7-2034
SQDMULL, SQDMULL2	Signed saturating doubling multiply long (vector and scalar form)	*SQDMULL, SQDMULL2 (by element)* on page C7-2039
SQRDMLAH	Signed saturating rounding doubling multiply accumulate returning high half	*SQRDMLSH (by element)* on page C7-2052
SQRDMLSH	Signed saturating rounding doubling multiply subtract returning high half	*SQRDMLSH (vector)* on page C7-2055
SQRDMULH	Signed saturating rounding doubling multiply returning high half (vector and scalar form)	*SQRDMULH (by element)* on page C7-2057
UMLAL, UMLAL2	Unsigned multiply-add long (vector form)	*UMLAL, UMLAL2 (by element)* on page C7-2227
UMLSL, UMLSL2	Unsigned multiply-subtract long (vector form)	*UMLSL, UMLSL2 (by element)* on page C7-2232
UMULL, UMULL2	Unsigned multiply long (vector form)	*UMULL, UMULL2 (by element)* on page C7-2240

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C3.5.19 SIMD permute

Table C3-83 shows the SIMD permute instructions.

Table C3-83 SIMD permute instructions

Mnemonic	Instruction	See
EXT	Extract vector from a pair of vectors	*EXT* on page C7-1480
TRN1	Transpose vectors (primary)	*TRN1* on page C7-2175
TRN2	Transpose vectors (secondary)	*TRN2* on page C7-2177
UZP1	Unzip vectors (primary)	*UZP1* on page C7-2299
UZP2	Unzip vectors (secondary)	*UZP2* on page C7-2301
ZIP1	Zip vectors (primary)	*ZIP1* on page C7-2306
ZIP2	Zip vectors (secondary)	*ZIP2* on page C7-2308

C3.5.20 SIMD immediate

Table C3-84 shows the SIMD immediate instructions.

Table C3-84 SIMD immediate instructions

Mnemonic	Instruction	See
BIC	Bitwise bit clear immediate	*BIC (vector, immediate)* on page C7-1430
FMOV	Floating-point move immediate	*FMOV (vector, immediate)* on page C7-1684
MOVI	Move immediate	*MOVI* on page C7-1865
MVNI	Move inverted immediate	*MVNI* on page C7-1873
ORR	Bitwise inclusive OR immediate	*ORR (vector, immediate)* on page C7-1882

C3.5.21 SIMD shift (immediate)

For information about the variants of these instructions, see *Common features of SIMD instructions* on page C3-239.

Table C3-85 shows the SIMD shift immediate instructions.

Table C3-85 SIMD shift (immediate) instructions

Mnemonic	Instruction	See
RSHRN, RSHRN2	Rounding shift right narrow immediate (vector form)	*RSHRN, RSHRN2* on page C7-1901
SHL	Shift left immediate (vector and scalar form)	*SHL* on page C7-1957
SHLL, SHLL2	Shift left long (by element size) (vector form)	*SHLL, SHLL2* on page C7-1959
SHRN, SHRN2	Shift right narrow immediate (vector form)	*SHRN, SHRN2* on page C7-1961
SLI	Shift left and insert immediate (vector and scalar form)	*SLI* on page C7-1965
SQRSHRN, SQRSHRN2	Signed saturating rounded shift right narrow immediate (vector and scalar form)	*SQRSHRN, SQRSHRN2* on page C7-2064

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-85 SIMD shift (immediate) instructions (continued)

Mnemonic	Instruction	See
SQRSHRUN, SQRSHRUN2	Signed saturating shift right unsigned narrow immediate (vector and scalar form)	*SQRSHRUN, SQRSHRUN2* on page C7-2067
SQSHL	Signed saturating shift left immediate (vector and scalar form)	*SQSHL (immediate)* on page C7-2070
SQSHLU	Signed saturating shift left unsigned immediate (vector and scalar form)	*SQSHLU* on page C7-2075
SQSHRN, SQSHRN2	Signed saturating shift right narrow immediate (vector and scalar form)	*SQSHRN, SQSHRN2* on page C7-2078
SQSHRUN, SQSHRUN2	Signed saturating shift right unsigned narrow immediate (vector and scalar form)	*SQSHRUN, SQSHRUN2* on page C7-2081
SRI	Shift right and insert immediate (vector and scalar form)	*SRI* on page C7-2094
SRSHR	Signed rounding shift right immediate (vector and scalar form)	*SRSHR* on page C7-2099
SRSRA	Signed rounding shift right and accumulate immediate (vector and scalar form)	*SRSRA* on page C7-2101.
SSHLL, SSHLL2	Signed shift left long immediate (vector form)	*SSHLL, SSHLL2* on page C7-2106
SSHR	Signed shift right immediate (vector and scalar form)	*SSHR* on page C7-2108
SSRA	Signed integer shift right and accumulate immediate (vector and scalar form)	*SSRA* on page C7-2111
SXTL, SXTL2	Signed integer extend (vector only)	*SXTL, SXTL2* on page C7-2169
UQRSHRN, UQRSHRN2	Unsigned saturating rounded shift right narrow immediate (vector and scalar form)	*UQRSHRN, UQRSHRN2* on page C7-2249
UQSHL	Unsigned saturating shift left immediate (vector and scalar form)	*UQSHL (immediate)* on page C7-2252
UQSHRN, UQSHRN2	Unsigned saturating shift right narrow immediate (vector and scalar form)	*UQSHRN, UQSHRN2* on page C7-2257
URSHR	Unsigned rounding shift right immediate (vector and scalar form)	*URSHR* on page C7-2270
URSRA	Unsigned integer rounding shift right and accumulate immediate (vector and scalar form)	*URSRA* on page C7-2273
USHLL, USHLL2	Unsigned shift left long immediate (vector form)	*USHLL, USHLL2* on page C7-2282
USHR	Unsigned shift right immediate (vector and scalar form)	*USHR* on page C7-2284
USRA	Unsigned shift right and accumulate immediate (vector and scalar form)	*USRA* on page C7-2290
UXTL, UXTL2	Unsigned integer extend (vector only)	*UXTL, UXTL2* on page C7-2297

C3.5.22 SIMD floating-point and integer conversion

The SIMD floating-point and integer conversion instructions generate the Invalid Operation floating-point exception in response to a floating-point input of NaN, infinity, or a numerical value that cannot be represented within the destination register. An out of range integer or a fixed-point result is saturated to the size of the destination register. A numeric result that differs from the input raises the Inexact floating-point exception.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-86 shows the SIMD floating-point and integer conversion instructions.

Table C3-86 SIMD floating-point and integer conversion instructions

Mnemonic	Instruction	See
FCVTAS	Floating-point convert to signed integer, rounding to nearest with ties to away (vector and scalar form)	*FCVTAS (vector)* on page C7-1551
FCVTAU	Floating-point convert to unsigned integer, rounding to nearest with ties to away (vector and scalar form)	*FCVTAU (vector)* on page C7-1556
FCVTMS	Floating-point convert to signed integer, rounding toward minus infinity (vector and scalar form)	*FCVTMS (vector)* on page C7-1563
FCVTMU	Floating-point convert to unsigned integer, rounding toward minus infinity (vector and scalar form)	*FCVTMU (vector)* on page C7-1568
FCVTNS	Floating-point convert to signed integer, rounding to nearest with ties to even (vector and scalar form)	*FCVTNS (vector)* on page C7-1575
FCVTNU	Floating-point convert to unsigned integer, rounding to nearest with ties to even (vector and scalar form)	*FCVTNU (vector)* on page C7-1580
FCVTPS	Floating-point convert to signed integer, rounding toward positive infinity (vector and scalar form)	*FCVTPS (vector)* on page C7-1585
FCVTPU	Floating-point convert to unsigned integer, rounding toward positive infinity (vector and scalar form)	*FCVTPU (vector)* on page C7-1590
FCVTZS	Floating-point convert to signed integer, rounding toward zero (vector and scalar form)	*FCVTZS (vector, integer)* on page C7-1600
	Floating-point convert to signed fixed-point, rounding toward zero (vector and scalar form)	*FCVTZS (vector, fixed-point)* on page C7-1597
FCVTZU	Floating-point convert to unsigned integer, rounding toward zero (vector and scalar form)	*FCVTZU (vector, integer)* on page C7-1610
	Floating-point convert to unsigned fixed-point, rounding toward zero, (vector and scalar form)	*FCVTZU (vector, fixed-point)* on page C7-1607
SCVTF	Signed integer convert to floating-point (vector and scalar form)	*SCVTF (vector, integer)* on page C7-1926
	Signed fixed-point convert to floating-point (vector and scalar form)	*SCVTF (vector, fixed-point)* on page C7-1923
UCVTF	Unsigned integer convert to floating-point (vector and scalar form)	*UCVTF (vector, integer)* on page C7-2200
	Unsigned fixed-point convert to floating-point (vector and scalar form)	*UCVTF (vector, fixed-point)* on page C7-2197

C3.5.23 SIMD reduce (across vector lanes)

The SIMD reduce (across vector lanes) instructions perform arithmetic operations horizontally, that is across all lanes of the input vector. They deliver a single scalar result.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-87 shows the SIMD reduce (across vector lanes) instructions.

Table C3-87 SIMD reduce (across vector lanes) instructions

Mnemonic	Instruction	See
ADDV	Add (across vector)	*ADDV* on page C7-1411
FMAXNMV	Floating-point maximum number (across vector)	*FMAXNMV* on page C7-1636
FMAXV	Floating-point maximum (across vector)	*FMAXV* on page C7-1642
FMINNMV	Floating-point minimum number (across vector)	*FMINNMV* on page C7-1656
FMINV	Floating-point minimum (across vector)	*FMINV* on page C7-1662
SADDLV	Signed add long (across vector)	*SADDLV* on page C7-1919
SMAXV	Signed maximum (across vector)	*SMAXV* on page C7-1990
SMINV	Signed minimum (across vector)	*SMINV* on page C7-1996
UADDLV	Unsigned add long (across vector)	*UADDLV* on page C7-2193
UMAXV	Unsigned maximum (across vector)	*UMAXV* on page C7-2219
UMINV	Unsigned minimum (across vector)	*UMINV* on page C7-2225

C3.5.24 SIMD pairwise arithmetic

The SIMD pairwise arithmetic instructions perform operations on pairs of adjacent elements and deliver a vector result.

Table C3-88 shows the SIMD pairwise arithmetic instructions.

Table C3-88 SIMD pairwise arithmetic instructions

Mnemonic	Instruction	See
ADDP	Add pairwise (vector and scalar form)	*ADDP (vector)* on page C7-1409
		ADDP (scalar) on page C7-1407
FADDP	Floating-point add pairwise (vector and scalar form)	*FADDP (vector)* on page C7-1503
		FADDP (scalar) on page C7-1501
FMAXNMP	Floating-point maximum number pairwise (vector and scalar form)	*FMAXNMP (vector)* on page C7-1634
		FMAXNMP (scalar) on page C7-1632
FMAXP	Floating-point maximum pairwise (vector and scalar form)	*FMAXP (vector)* on page C7-1640
		FMAXP (scalar) on page C7-1638
FMINNMP	Floating-point minimum number pairwise (vector and scalar form)	*FMINNMP (vector)* on page C7-1654
		FMINNMP (scalar) on page C7-1652
FMINP	Floating-point minimum pairwise (vector and scalar form)	*FMINP (vector)* on page C7-1660
		FMINP (scalar) on page C7-1658
SMAXP	Signed maximum pairwise	*SMAXP* on page C7-1988

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-88 SIMD pairwise arithmetic instructions (continued)

Mnemonic	Instruction	See
SMINP	Signed minimum pairwise	*SMINP* on page C7-1994
UMAXP	Unsigned maximum pairwise	*UMAXP* on page C7-2217
UMINP	Unsigned minimum pairwise	*UMINP* on page C7-2223

C3.5.25 SIMD dot product

ARMv8.2-DotProd provides SIMD instructions that perform the dot product of the four 8-bit subelements of the 32-bit elements of one vector with the four 8-bit subelements of a second vector. It provides two forms of the instructions, each with signed and unsigned versions:

Vector form The dot product is calculated for each element of the first vector with the corresponding element of the second element.

Indexed form The dot product is calculated for each element of the first vector with the element of the second vector that is indicated by the index argument to the instruction.

Note

That is, a single element from the second vector is used, and a the dot product is calculated between each element of the first vector and this single element from the second vector.

Table C3-89 SIMD dot product

Mnemonic	Instruction	See
SDOT	Signed dot product (vector form)	*SDOT (vector)* on page C7-1935
UDOT	Unsigned dot product (vector form)	*UDOT (vector)* on page C7-2209
SDOT	Signed dot product (indexed form)	*SDOT (by element)* on page C7-1933
UDOT	Unsigned dot product (indexed form)	*UDOT (by element)* on page C7-2207
USDOT	Mixed sign integer dot product (vector form)[a]	*USDOT (vector)* on page C7-2275
	Mixed sign integer dot product by indexed quadruplet[a]	*USDOT (by element)* on page C7-2277
SUDOT	Mixed sign integer dot product by indexed quadruplet[a]	*SUDOT (by element)* on page C7-2165

a. This instruction is supported when ARMv8.2-I8MM is implemented.

C3.5.26 SIMD table lookup

Table C3-90 shows the SIMD table lookup instructions.

Table C3-90 SIMD table lookup instructions

Mnemonic	Instruction	See
TBL	Table vector lookup	*TBL* on page C7-2171
TBX	Table vector lookup extension	*TBX* on page C7-2173

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C3.5.27 SIMD complex number arithmetic

ARMv8.3-CompNum provides SIMD instructions that perform arithmetic on complex numbers held in element pairs in vector registers, where the less significant element of the pair contains the real component and the more significant element contains the imaginary component.

These instructions provide double-precision and single-precision versions. If ARMv8.2-FP16 is implemented they also provide half-precision versions, otherwise the half-precision encodings are UNDEFINED.

Table C3-91 shows the ARMv8.3-CompNum SIMD instructions.

Table C3-91 SIMD complex number arithmetic

Mnemonic	Instruction	See
FCADD	Floating-point complex add	*FCADD* on page C7-1505
FCMLA	Floating-point complex multiply accumulate (vector form)	*FCMLA* on page C7-1535
FCMLA	Floating-point complex multiply accumulate (indexed form)	*FCMLA (by element)* on page C7-1532

A pair of FCMLA instructions can be used to perform a complex number multiplication. This is demonstrated in *Complex multiplication* on page K10-7958.

C3.5.28 SIMD BFloat16

The SIMD BFloat16 instructions are provided by ARMv8.2-BF16.

These instructions perform an implicit conversion of vectors of BF16 input values to IEEE 754 single-precision floating-point format, combined with an N-way dot product calculation that accumulates the products into a vector of single-precision accumulators.

All of these instructions perform arithmetic with fixed behaviors, irrespective of the values of FPCR. These behaviors are:

- Exceptional floating-point conditions produce the expected IEEE 754 default result, but do not modify the cumulative floating-point exception flags in FPSR, and cannot cause a trapped floating-point exception.
- Multiplication and addition operations are always chained and never fused. Multiplication that overflows cannot be brought back into range by a fused addition.

Note

The fractional part of the product of two BF16 inputs can be exactly represented in single-precision format, see *BFloat16 floating-point format* on page A1-48.

Table C3-92 shows these instructions.

Table C3-92 BFloat16 SIMD instructions

Mnemonic	Instruction	See
BFDOT	BFloat16 floating-point dot product (vector and indexed forms)	*BFDOT (vector)* on page C7-1424
		BFDOT (by element) on page C7-1422
BFMMLA	BFloat16 floating-point matrix multiply-accumulate into 2x2 matrix	*BFMMLA* on page C7-1429

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-92 BFloat16 SIMD instructions (continued)

Mnemonic	Instruction	See
BFMLALB	BFloat16 floating-point widening multiply-add long bottom (vector and indexed forms)	*BFMLALB, BFMLALT (vector)* on page C7-1428
		BFMLALB, BFMLALT (by element) on page C7-1426
BFMLALT	BFloat16 floating-point widening multiply-add long top (vector and index forms)	*BFMLALB, BFMLALT (vector)* on page C7-1428
		BFMLALB, BFMLALT (by element) on page C7-1426
BFCVTN, BFCVTN2	Floating-point convert from single-precision to BFloat16 format (vector form)	*BFCVTN, BFCVTN2* on page C7-1420

C3.5.29 SIMD matrix multiplication

These instructions are provided by ARMv8.2-I8MM, and include integer matrix multiply-accumulate instructions.

The matrix multiply-accumulate instructions delimit source and destination vectors into segments. Within each segment:

- The first source vector matrix is organized in row-by-row order.
- The second source vector matrix is organized in a column-by-column order.
- The destination vector matrix is organized in row-by-row order.

One matrix multiplication is performed per segment.

Table C3-93 shows these instructions.

Table C3-93 Matrix multiply SIMD instructions

Mnemonic	Instruction	See
SMMLA	Widening 8-bit signed integer matrix multiply-accumulate into 2x2 matrix	*SMMLA (vector)* on page C7-2008
UMMLA	Widening 8-bit unsigned integer matrix multiply-accumulate into 2x2 matrix	*UMMLA (vector)* on page C7-2237
USMMLA	Widening 8-bit mixed sign integer matrix multiply-accumulate into 2x2 matrix	*USMMLA (vector)* on page C7-2287

C3.5.30 The Cryptographic Extension

The instructions provided by the optional Armv8.0 Cryptographic Extension use the SIMD and floating-point register file. For more information about the functions they provide see:

- Announcing the Advanced Encryption Standard.
- *The Galois/Counter Mode of Operation*.
- *Announcing the Secure Hash Standard*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-94 shows the Armv8.0 Cryptographic Extension instructions.

Table C3-94 Cryptographic Extension instructions

Mnemonic	Instruction	See
AESD	AES single round decryption	*AESD* on page C7-1413
AESE	AES single round encryption	*AESE* on page C7-1414
AESIMC	AES inverse mix columns	*AESIMC* on page C7-1415
AESMC	AES mix columns	*AESMC* on page C7-1416
PMULL	Polynomial multiply long	*PMULL, PMULL2* on page C7-1888[a]
SHA1C	SHA1 hash update (choose)	*SHA1C* on page C7-1937
SHA1H	SHA1 fixed rotate	*SHA1H* on page C7-1938
SHA1M	SHA1 hash update (majority)	*SHA1M* on page C7-1939
SHA1P	SHA1 hash update (parity)	*SHA1P* on page C7-1940
SHA1SU0	SHA1 schedule update 0	*SHA1SU0* on page C7-1941
SHA1SU1	SHA1 schedule update 1	*SHA1SU1* on page C7-1942
SHA256H	SHA256 hash update, part 1	*SHA256H* on page C7-1944
SHA256H2	SHA256 hash update, part 2	*SHA256H2* on page C7-1943
SHA256SU0	SHA256 schedule update 0	*SHA256SU0* on page C7-1945
SHA256SU1	SHA256 schedule update 1	*SHA256SU1* on page C7-1946

a. The Cryptographic Extension adds the variant of the instruction that operates on two 64-bit polynomials.

See *The Armv8 Cryptographic Extension* on page A2-68 for information about the permitted implementation options for the Cryptographic Extension.

Armv8.2 extensions to the Cryptographic Extension

Armv8.2 provides two optional extensions to the Cryptographic Extension, see:

- *Armv8.2-SHA, SHA2-512 and SHA3*.
- *Armv8.2-SM, SM3 and SM4* on page C3-264.

Armv8.2-SHA, SHA2-512 and SHA3

ARMv8.2-SHA provides:

- Instructions to accelerate the SHA-2 hash algorithm using a digest that is larger than 256 bits. The relevant standards are SHA-384, SHA-512, SHA-512|224 and SHA-512|256. These are all based on the SHA-512 computation, and therefore this set of instructions is described as the *SHA512 instructions*.
- Instructions to accelerate the SHA-3 hash algorithm. This set of instructions is described as the *SHA3 instructions*.

Implementation of ARMv8.2-SHA requires the implementation of the SHA1 and SHA2-256 instructions from the Armv8.0 Cryptographic Extension.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

Implementation of ARMv8.2-SHA does not require the implementation of the AES instructions, and the 64-bit polynomial variants of the `PMULL` instructions, from the Armv8.0 Cryptographic Extension.

When ARMv8.2-SHA is implemented:

- The value of ID_AA64ISAR0_EL1.SHA2 is `0b0010`, indicating support for the SHA512 instructions.
- The value of ID_AA64ISAR0_EL1.SHA3 is `0b0001`, indicating support for the SHA3 instructions.

Table C3-95 shows the ARMv8.2-SHA SHA512 instructions:

Table C3-95 Armv8.2-SHA SHA512 instructions

Mnemonic	Instruction	See
`SHA512H`	SHA512 Hash update part 1	*SHA512H* on page C7-1948
`SHA512H2`	SHA512 Hash update part 2	*SHA512H2* on page C7-1950
`SHA512SU0`	SHA512 Schedule Update 0	*SHA512SU0* on page C7-1952
`SHA512SU1`	SHA512 Schedule Update 1	*SHA512SU1* on page C7-1953

Use of the SHA512 instructions on page K10-7960 shows an example of the use of these instructions to calculate a SHA512 hash iteration. This example code is not part of the architectural definition of these instructions.

Table C3-96 shows the ARMv8.2-SHA SHA3 instructions. The SHA-3 hash algorithm is based on a running digest of 1600 bytes, arranged as a five by five array of 64-bit registers. The Arm acceleration of these instructions is based on mapping the 25 64-bit values into 25 vector registers, with each 64-bit value occupying the same 64-bit element in each vector. A series of transformations is performed on these registers as part of a round of the SHA-3 hash calculation.

The SIMD nature of the vector registers means the acceleration can compute two parallel SHA3 hash calculations, where one calculation is performed using the zeroth 64-bit element of each vector, and the other calculation is performed using the first 64-bit element of each vector.

Note

This is useful because the SHA-3 family of standards is being extended to introduce a fast parallel hash algorithm that benefits from being able to calculate SHA-3 hashes in parallel.

To provide acceleration where the SIMD calculation is not required, the instructions provide variants that operate only on the zeroth 64-bit elements. These are provided as a power optimization.

Table C3-96 Armv8.2-SHA SHA3 instructions

Mnemonic	Instruction	See
`EOR3`	Three-way Exclusive OR	*EOR3* on page C7-1479
`RAX1`	Rotate and Exclusive OR	*RAX1* on page C7-1892
`XAR`	Exclusive OR and Rotate	*XAR* on page C7-2303
`BCAX`	Bit Clear and Exclusive OR	*BCAX* on page C7-1418

Use of the SHA3 instructions on page K10-7961 shows an example of the use of these instructions to calculate the combined theta, phi, rho and chi operations of a SHA3 iteration. This example code is not part of the architectural definition of these instructions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Armv8.2-SM, SM3 and SM4

ARMv8.2-SM provides:

- Instructions to accelerate the SM3 hash algorithm, the standard Chinese hash algorithm. These are described as the *SM3 instructions*.
- Instructions to accelerate the SM4 encryption algorithm, the standard Chinese encryption algorithm. This set of instructions is described as the *SM4 instructions*.

ARMv8.2-SM can be implemented independently of any part of the Armv8.0 Cryptographic Extension, and independently of ARMv8.2-SHA.

Note

This means that Armv8.2 permits an implementation of the Cryptographic Extension that provides only the ARMv8.2-SM functionality.

When ARMv8.2-SM is implemented:

- The value of ID_AA64ISAR0_EL1.SM3 is `0b0001`, indicating support for the SM3 instructions.
- The value of ID_AA64ISAR0_EL1.SM4 is `0b0001`, indicating support for the SM4 instructions.

Table C3-97 shows the ARMv8.2-SM SM3 instructions. The SM3 algorithm computes a digest of 256 bits, that can be held in two vector registers. The SM3 instructions include instructions to accelerate the computation of the hash and the schedule update.

Note

The SM3 instruction names refer to intermediate variables defined as part of the *SM3 Cryptographic Hash Algorithm* specification.

Table C3-97 Armv8.2-SM SM3 instructions

Mnemonic	Instruction	See
`SM3SS1`	SM3 SS1 calculation	*SM3SS1* on page C7-1972
`SM3TT1A`	SM3 TT1 calculation, part A	*SM3TT1A* on page C7-1974
`SM3TT1B`	SM3 TT1 calculation, part B	*SM3TT1B* on page C7-1976
`SM3TT2A`	SM3 TT2 calculation, part A	*SM3TT2A* on page C7-1978
`SM3TT2B`	SM3 TT2 calculation, part B	*SM3TT2B* on page C7-1980
`SM3PARTW1`	SM3 PARTW calculation, part 1	*SM3PARTW1* on page C7-1968
`SM3PARTW2`	SM3 PARTW calculation, part 1	*SM3PARTW2* on page C7-1970

Use of the SM3 instructions on page K10-7962 shows an example of the use of these instructions to generate an SM3 hash. This example code is not part of the architectural definition of these instructions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-98 shows the ARMv8.2-SM SM4 instructions. The SM4 algorithm is 128-bit wide block cipher. The `SM4E` instruction accelerates a single round of encryption or decryption, and the `SM4EKEY` instruction accelerates a single round of key generation:

Table C3-98 Armv8.2-SM SM4 instructions

Mnemonic	Instruction	See
`SM4E`	SM4 Encrypt	*SM4E* on page C7-1982
`SM4EKEY`	SM4 Key	*SM4EKEY* on page C7-1984

Use of the SM4 instructions on page K10-7964 shows an example of the use of these instructions to perform SM4 encryption and decryption. This example code is not part of the architectural definition of these instructions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter C4
A64 Instruction Set Encoding

This chapter describes the encoding of the A64 instruction set. It contains the following section:

- *A64 instruction set encoding* on page C4-268.

In this chapter:

- In the decode tables, an entry of - for a field value means the value of the field does not affect the decoding.
- In the decode diagrams, a shaded field indicates that the bits in that field are not used in that level of decode.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C4.1 A64 instruction set encoding

The A64 instruction encoding is:

Table C4-1 Main encoding table for the A64 instruction set

Decode fields op0	Decode group or instruction page
`0000x`	*Reserved*
`00011`	Unallocated.
`0010x`	SVE Instructions. See *The Scalable Vector Extension (SVE)* on page A2-99.
`0011x`	Unallocated.
`100xx`	*Data Processing -- Immediate*
`101xx`	*Branches, Exception Generating and System instructions* on page C4-273
`x1x0x`	*Loads and Stores* on page C4-281
`x101x`	*Data Processing -- Register* on page C4-313
`x111x`	*Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323

C4.1.1 Reserved

This section describes the encoding of the Reserved group. The encodings in this section are decoded from *A64 instruction set encoding*.

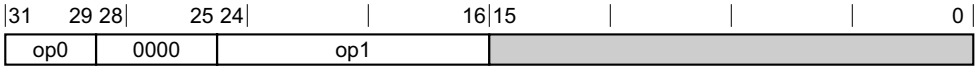


Table C4-2 Encoding table for the Reserved group

Decode fields op0	op1	Decode group or instruction page
`000`	`000000000`	UDF
-	`!= 000000000`	Unallocated.
`!= 000`	-	Unallocated.

C4.1.2 Data Processing -- Immediate

This section describes the encoding of the Data Processing -- Immediate group. The encodings in this section are decoded from *A64 instruction set encoding*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31 29	28 26	25 23	22 0
	100	op0	

Table C4-3 Encoding table for the Data Processing -- Immediate group

Decode fields op0	Decode group or instruction page
00x	*PC-rel. addressing*
010	*Add/subtract (immediate)*
011	*Add/subtract (immediate, with tags)* on page C4-270
100	*Logical (immediate)* on page C4-270
101	*Move wide (immediate)* on page C4-271
110	*Bitfield* on page C4-272
111	*Extract* on page C4-272

PC-rel. addressing

This section describes the encoding of the PC-rel. addressing instruction class. The encodings in this section are decoded from *Data Processing -- Immediate* on page C4-268.

31	30 29	28 27 26 25 24	23 5	4 0
op	immlo	1 0 0 0 0	immhi	Rd

Decode fields op	Instruction page
0	ADR
1	ADRP

Add/subtract (immediate)

This section describes the encoding of the Add/subtract (immediate) instruction class. The encodings in this section are decoded from *Data Processing -- Immediate* on page C4-268.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31	30	29	28	27	26	25	24	23	22	21	10	9	5	4	0
sf	op	S	1	0	0	0	1	0	sh	imm12			Rn		Rd

Decode fields			Instruction page
sf	op	S	
0	0	0	ADD (immediate) - 32-bit variant
0	0	1	ADDS (immediate) - 32-bit variant
0	1	0	SUB (immediate) - 32-bit variant
0	1	1	SUBS (immediate) - 32-bit variant
1	0	0	ADD (immediate) - 64-bit variant
1	0	1	ADDS (immediate) - 64-bit variant
1	1	0	SUB (immediate) - 64-bit variant
1	1	1	SUBS (immediate) - 64-bit variant

Add/subtract (immediate, with tags)

This section describes the encoding of the Add/subtract (immediate, with tags) instruction class. The encodings in this section are decoded from *Data Processing -- Immediate* on page C4-268.

31	30	29	28	27	26	25	24	23	22	21	16	15	14	13	10	9	5	4	0
sf	op	S	1	0	0	0	1	1	o2	uimm6		op3	uimm4				Rn		Rd

Decode fields				Instruction page	Architecture version
sf	op	S	o2		
-	-	-	1	Unallocated.	-
0	-	-	0	Unallocated.	-
1	-	1	0	Unallocated.	-
1	0	0	0	ADDG	ARMv8.5-MemTag
1	1	0	0	SUBG	ARMv8.5-MemTag

Logical (immediate)

This section describes the encoding of the Logical (immediate) instruction class. The encodings in this section are decoded from *Data Processing -- Immediate* on page C4-268.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31	30	29	28	27	26	25	24	23	22	21 … 16	15 … 10	9 … 5	4 … 0
sf	opc	1	0	0	1	0	0	N		immr	imms	Rn	Rd

Decode fields			Instruction page
sf	opc	N	
0	-	1	Unallocated.
0	00	0	AND (immediate) - 32-bit variant
0	01	0	ORR (immediate) - 32-bit variant
0	10	0	EOR (immediate) - 32-bit variant
0	11	0	ANDS (immediate) - 32-bit variant
1	00	-	AND (immediate) - 64-bit variant
1	01	-	ORR (immediate) - 64-bit variant
1	10	-	EOR (immediate) - 64-bit variant
1	11	-	ANDS (immediate) - 64-bit variant

Move wide (immediate)

This section describes the encoding of the Move wide (immediate) instruction class. The encodings in this section are decoded from *Data Processing -- Immediate* on page C4-268.

31	30	29	28	27	26	25	24	23	22	21	20 … 5	4 … 0
sf	opc	1	0	0	1	0	1	hw			imm16	Rd

Decode fields			Instruction page
sf	opc	hw	
-	01	-	Unallocated.
0	-	1x	Unallocated.
0	00	0x	MOVN - 32-bit variant
0	10	0x	MOVZ - 32-bit variant
0	11	0x	MOVK - 32-bit variant
1	00	-	MOVN - 64-bit variant
1	10	-	MOVZ - 64-bit variant
1	11	-	MOVK - 64-bit variant

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bitfield

This section describes the encoding of the Bitfield instruction class. The encodings in this section are decoded from *Data Processing -- Immediate* on page C4-268.

31 30 29 28	27 26 25 24	23 22 21	16	15 10	9 5	4 0
sf opc	1 0 0 1 1 0	N	immr	imms	Rn	Rd

Decode fields			Instruction page
sf	opc	N	
-	11	-	Unallocated.
0	-	1	Unallocated.
0	00	0	SBFM - 32-bit variant
0	01	0	BFM - 32-bit variant
0	10	0	UBFM - 32-bit variant
1	-	0	Unallocated.
1	00	1	SBFM - 64-bit variant
1	01	1	BFM - 64-bit variant
1	10	1	UBFM - 64-bit variant

Extract

This section describes the encoding of the Extract instruction class. The encodings in this section are decoded from *Data Processing -- Immediate* on page C4-268.

31 30 29 28	27 26 25 24	23 22 21 20	16	15 10	9 5	4 0
sf op21	1 0 0 1 1 1	N o0	Rm	imms	Rn	Rd

Decode fields					Instruction page
sf	op21	N	o0	imms	
-	x1	-	-	-	Unallocated.
-	00	-	1	-	Unallocated.
-	1x	-	-	-	Unallocated.
0	-	-	-	1xxxxx	Unallocated.
0	-	1	-	-	Unallocated.
0	00	0	0	0xxxxx	EXTR - 32-bit variant
1	-	0	-	-	Unallocated.
1	00	1	0	-	EXTR - 64-bit variant

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C4.1.3 Branches, Exception Generating and System instructions

This section describes the encoding of the Branches, Exception Generating and System instructions group. The encodings in this section are decoded from *A64 instruction set encoding* on page C4-268.

31 29	28 26	25 12	11 5	4 0
op0	101	op1		op2

Table C4-4 Encoding table for the Branches, Exception Generating and System instructions group

Decode fields			Decode group or instruction page
op0	op1	op2	
010	0xxxxxxxxxxxxx	-	*Conditional branch (immediate)*
110	00xxxxxxxxxxxx	-	*Exception generation* on page C4-274
110	01000000110010	11111	*Hints* on page C4-274
110	01000000110011	-	*Barriers* on page C4-276
110	0100000xxx0100	-	*PSTATE* on page C4-276
110	0100x01xxxxxxx	-	*System instructions* on page C4-277
110	0100x1xxxxxxxx	-	*System register move* on page C4-277
110	1xxxxxxxxxxxxx	-	*Unconditional branch (register)* on page C4-277
x00	-	-	*Unconditional branch (immediate)* on page C4-280
x01	0xxxxxxxxxxxxx	-	*Compare and branch (immediate)* on page C4-281
x01	1xxxxxxxxxxxxx	-	*Test and branch (immediate)* on page C4-281

Conditional branch (immediate)

This section describes the encoding of the Conditional branch (immediate) instruction class. The encodings in this section are decoded from *Branches, Exception Generating and System instructions*.

31 30 29 28	27 26 25 24	23 5	4	3 0
0 1 0 1	0 1 0	o1	imm19	o0	cond

Decode fields		Instruction page
o1	o0	
0	0	B.cond
0	1	Unallocated.
1	-	Unallocated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Exception generation

This section describes the encoding of the Exception generation instruction class. The encodings in this section are decoded from *Branches, Exception Generating and System instructions* on page C4-273.

31 30 29 28	27 26 25 24	23 21 20	5 4	2 1 0
1 1 0 1	0 1 0 0	opc	imm16	op2	LL

Decode fields			Instruction page	Architecture version
opc	op2	LL		
-	001	-	Unallocated.	-
-	01x	-	Unallocated.	-
-	1xx	-	Unallocated.	-
000	000	00	Unallocated.	-
000	000	01	SVC	-
000	000	10	HVC	-
000	000	11	SMC	Armv8.0-A
001	000	x1	Unallocated.	-
001	000	00	BRK	-
001	000	1x	Unallocated.	-
010	000	x1	Unallocated.	-
010	000	00	HLT	-
010	000	1x	Unallocated.	-
011	000	01	Unallocated.	-
011	000	1x	Unallocated.	-
100	000	-	Unallocated.	-
101	000	00	Unallocated.	-
101	000	01	DCPS1	-
101	000	10	DCPS2	-
101	000	11	DCPS3	Armv8.0-A
110	000	-	Unallocated.	-
111	000	-	Unallocated.	-

Hints

This section describes the encoding of the Hints instruction class. The encodings in this section are decoded from *Branches, Exception Generating and System instructions* on page C4-273.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 8	7 5	4	3 2 1 0
1 1 0 1	0 1 0 1	0 0 0 0	0 0 1 1	0 0 1 0	CRm	op2	1	1 1 1 1

Decode fields		Instruction page	Architecture version
CRm	op2		
-	-	HINT	-
0000	000	NOP	-
0000	001	YIELD	-
0000	010	WFE	-
0000	011	WFI	-
0000	100	SEV	-
0000	101	SEVL	-
0000	110	DGH	ARMv8.0-DGH
0000	111	XPACD, XPACI, XPACLRI	ARMv8.3-PAuth
0001	000	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA - PACIA1716 variant	ARMv8.3-PAuth
0001	010	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB - PACIB1716 variant	ARMv8.3-PAuth
0001	100	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA - AUTIA1716 variant	ARMv8.3-PAuth
0001	110	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB - AUTIB1716 variant	ARMv8.3-PAuth
0010	000	ESB	RAS
0010	001	PSB CSYNC	SPE
0010	010	TSB CSYNC	ARMv8.4-Trace
0010	100	CSDB	-
0011	000	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA - PACIAZ variant	ARMv8.3-PAuth
0011	001	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA - PACIASP variant	ARMv8.3-PAuth
0011	010	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB - PACIBZ variant	ARMv8.3-PAuth
0011	011	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB - PACIBSP variant	ARMv8.3-PAuth
0011	100	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA - AUTIAZ variant	ARMv8.3-PAuth
0011	101	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA - AUTIASP variant	ARMv8.3-PAuth
0011	110	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB - AUTIBZ variant	ARMv8.3-PAuth
0011	111	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB - AUTIBSP variant	ARMv8.3-PAuth
0100	xx0	BTI	ARMv8.5-BTI

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Barriers

This section describes the encoding of the Barriers instruction class. The encodings in this section are decoded from *Branches, Exception Generating and System instructions* on page C4-273.

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 8	7 5	4 0
1 1 0 1	0 1 0 1	0 0 0 0	0 0 1 1	0 0 1 1	CRm	op2	Rt

Decode fields			Instruction page
CRm	op2	Rt	
-	000	-	Unallocated.
-	001	-	Unallocated.
-	010	11111	CLREX
-	101	11111	DMB
-	110	11111	ISB
-	111	!= 11111	Unallocated.
-	111	11111	SB
!= 0x00	100	11111	DSB
0000	100	11111	SSBB
0001	011	-	Unallocated.
001x	011	-	Unallocated.
01xx	011	-	Unallocated.
0100	100	11111	PSSBB
1xxx	011	-	Unallocated.

PSTATE

This section describes the encoding of the PSTATE instruction class. The encodings in this section are decoded from *Branches, Exception Generating and System instructions* on page C4-273.

31 30 29 28	27 26 25 24	23 22 21 20	19 18 16	15 14 13 12	11 8	7 5	4 0
1 1 0 1	0 1 0 1	0 0 0 0	op1	0 1 0 0	CRm	op2	Rt

Decode fields			Instruction page	Architecture version
op1	op2	Rt		
-	-	!= 11111	Unallocated.	-
-	-	11111	MSR (immediate)	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields			Instruction page	Architecture version
op1	op2	Rt		
000	000	11111	CFINV	ARMv8.4-CondM
000	001	11111	XAFLAG	ARMv8.5-CondM
000	010	11111	AXFLAG	ARMv8.5-CondM

System instructions

This section describes the encoding of the System instructions instruction class. The encodings in this section are decoded from *Branches, Exception Generating and System instructions* on page C4-273.

31 30 29 28	27 26 25 24	23 22 21 20	19 18 16	15 12	11 8	7 5	4 0
1 1 0 1	0 1 0 1	0 0 L 0	1 op1	CRn	CRm	op2	Rt

Decode fields	Instruction page
L	
0	SYS
1	SYSL

System register move

This section describes the encoding of the System register move instruction class. The encodings in this section are decoded from *Branches, Exception Generating and System instructions* on page C4-273.

31 30 29 28	27 26 25 24	23 22 21 20	19 18 16	15 12	11 8	7 5	4 0
1 1 0 1	0 1 0 1	0 0 L 1	o0 op1	CRn	CRm	op2	Rt

Decode fields	Instruction page
L	
0	MSR (register)
1	MRS

Unconditional branch (register)

This section describes the encoding of the Unconditional branch (register) instruction class. The encodings in this section are decoded from *Branches, Exception Generating and System instructions* on page C4-273.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31 30 29 28	27 26 25 24	21 20	16	15 10	9 5	4 0
1 1 0 1	0 1 1	opc	op2	op3	Rn	op4

Decode fields					Instruction page	Architecture version
opc	op2	op3	Rn	op4		
-	!= 11111	-	-	-	Unallocated.	-
0000	11111	000000	-	!= 00000	Unallocated.	-
0000	11111	000000	-	00000	BR	-
0000	11111	000001	-	-	Unallocated.	-
0000	11111	000010	-	!= 11111	Unallocated.	-
0000	11111	000010	-	11111	BRAA, BRAAZ, BRAB, BRABZ - Key A, zero modifier variant	ARMv8.3-PAuth
0000	11111	000011	-	!= 11111	Unallocated.	-
0000	11111	000011	-	11111	BRAA, BRAAZ, BRAB, BRABZ - Key B, zero modifier variant	ARMv8.3-PAuth
0000	11111	0001xx	-	-	Unallocated.	-
0000	11111	001xxx	-	-	Unallocated.	-
0000	11111	01xxxx	-	-	Unallocated.	-
0000	11111	1xxxxx	-	-	Unallocated.	-
0001	11111	000000	-	!= 00000	Unallocated.	-
0001	11111	000000	-	00000	BLR	-
0001	11111	000001	-	-	Unallocated.	-
0001	11111	000010	-	!= 11111	Unallocated.	-
0001	11111	000010	-	11111	BLRAA, BLRAAZ, BLRAB, BLRABZ - Key A, zero modifier variant	ARMv8.3-PAuth
0001	11111	000011	-	!= 11111	Unallocated.	-
0001	11111	000011	-	11111	BLRAA, BLRAAZ, BLRAB, BLRABZ - Key B, zero modifier variant	ARMv8.3-PAuth
0001	11111	0001xx	-	-	Unallocated.	-
0001	11111	001xxx	-	-	Unallocated.	-
0001	11111	01xxxx	-	-	Unallocated.	-
0001	11111	1xxxxx	-	-	Unallocated.	-
0010	11111	000000	-	!= 00000	Unallocated.	-
0010	11111	000000	-	00000	RET	-
0010	11111	000001	-	-	Unallocated.	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields					Instruction page	Architecture version
opc	op2	op3	Rn	op4		
0010	11111	000010	!= 11111	!= 11111	Unallocated.	-
0010	11111	000010	11111	11111	RETAA, RETAB - RETAA variant	ARMv8.3-PAuth
0010	11111	000011	!= 11111	!= 11111	Unallocated.	-
0010	11111	000011	11111	11111	RETAA, RETAB - RETAB variant	ARMv8.3-PAuth
0010	11111	0001xx	-	-	Unallocated.	-
0010	11111	001xxx	-	-	Unallocated.	-
0010	11111	01xxxx	-	-	Unallocated.	-
0010	11111	1xxxxx	-	-	Unallocated.	-
0011	11111	-	-	-	Unallocated.	-
0100	11111	000000	!= 11111	!= 00000	Unallocated.	-
0100	11111	000000	!= 11111	00000	Unallocated.	-
0100	11111	000000	11111	!= 00000	Unallocated.	-
0100	11111	000000	11111	00000	ERET	-
0100	11111	000001	-	-	Unallocated.	-
0100	11111	000010	!= 11111	!= 11111	Unallocated.	-
0100	11111	000010	!= 11111	11111	Unallocated.	-
0100	11111	000010	11111	!= 11111	Unallocated.	-
0100	11111	000010	11111	11111	ERETAA, ERETAB - ERETAA variant	ARMv8.3-PAuth
0100	11111	000011	!= 11111	!= 11111	Unallocated.	-
0100	11111	000011	!= 11111	11111	Unallocated.	-
0100	11111	000011	11111	!= 11111	Unallocated.	-
0100	11111	000011	11111	11111	ERETAA, ERETAB - ERETAB variant	ARMv8.3-PAuth
0100	11111	0001xx	-	-	Unallocated.	-
0100	11111	001xxx	-	-	Unallocated.	-
0100	11111	01xxxx	-	-	Unallocated.	-
0100	11111	1xxxxx	-	-	Unallocated.	-
0101	11111	!= 000000	-	-	Unallocated.	-
0101	11111	000000	!= 11111	!= 00000	Unallocated.	-
0101	11111	000000	!= 11111	00000	Unallocated.	-
0101	11111	000000	11111	!= 00000	Unallocated.	-
0101	11111	000000	11111	00000	DRPS	-
011x	11111	-	-	-	Unallocated.	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields					Instruction page	Architecture version
opc	op2	op3	Rn	op4		
1000	11111	00000x	-	-	Unallocated.	-
1000	11111	000010	-	-	BRAA, BRAAZ, BRAB, BRABZ - Key A, register modifier variant	ARMv8.3-PAuth
1000	11111	000011	-	-	BRAA, BRAAZ, BRAB, BRABZ - Key B, register modifier variant	ARMv8.3-PAuth
1000	11111	0001xx	-	-	Unallocated.	-
1000	11111	001xxx	-	-	Unallocated.	-
1000	11111	01xxxx	-	-	Unallocated.	-
1000	11111	1xxxxx	-	-	Unallocated.	-
1001	11111	00000x	-	-	Unallocated.	-
1001	11111	000010	-	-	BLRAA, BLRAAZ, BLRAB, BLRABZ - Key A, register modifier variant	ARMv8.3-PAuth
1001	11111	000011	-	-	BLRAA, BLRAAZ, BLRAB, BLRABZ - Key B, register modifier variant	ARMv8.3-PAuth
1001	11111	0001xx	-	-	Unallocated.	-
1001	11111	001xxx	-	-	Unallocated.	-
1001	11111	01xxxx	-	-	Unallocated.	-
1001	11111	1xxxxx	-	-	Unallocated.	-
101x	11111	-	-	-	Unallocated.	-
11xx	11111	-	-	-	Unallocated.	-

Unconditional branch (immediate)

This section describes the encoding of the Unconditional branch (immediate) instruction class. The encodings in this section are decoded from *Branches, Exception Generating and System instructions* on page C4-273.

31	30	29	28	27	26	25																									0
op	0	0	1	0	1	imm26																									

Decode fields	Instruction page
op	
0	B
1	BL

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Compare and branch (immediate)

This section describes the encoding of the Compare and branch (immediate) instruction class. The encodings in this section are decoded from *Branches, Exception Generating and System instructions* on page C4-273.

31	30	29	28	27	26	25	24	23 … 5	4 … 0
sf	0	1	1	0	1	0	op	imm19	Rt

Decode fields		Instruction page
sf	op	
0	0	CBZ - 32-bit variant
0	1	CBNZ - 32-bit variant
1	0	CBZ - 64-bit variant
1	1	CBNZ - 64-bit variant

Test and branch (immediate)

This section describes the encoding of the Test and branch (immediate) instruction class. The encodings in this section are decoded from *Branches, Exception Generating and System instructions* on page C4-273.

31	30	29	28	27	26	25	24	23 … 19	18 … 5	4 … 0
b5	0	1	1	0	1	1	op	b40	imm14	Rt

Decode fields	Instruction page
op	
0	TBZ
1	TBNZ

C4.1.4 Loads and Stores

This section describes the encoding of the Loads and Stores group. The encodings in this section are decoded from *A64 instruction set encoding* on page C4-268.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C4-5 Encoding table for the Loads and Stores group

Decode fields					Decode group or instruction page
op0	op1	op2	op3	op4	
0x00	1	00	000000	-	*Advanced SIMD load/store multiple structures* on page C4-283
0x00	1	01	0xxxxx	-	*Advanced SIMD load/store multiple structures (post-indexed)* on page C4-284
0x00	1	0x	1xxxxx	-	Unallocated.
0x00	1	10	x00000	-	*Advanced SIMD load/store single structure* on page C4-285
0x00	1	11	-	-	*Advanced SIMD load/store single structure (post-indexed)* on page C4-288
0x00	1	x0	x1xxxx	-	Unallocated.
0x00	1	x0	xx1xxx	-	Unallocated.
0x00	1	x0	xxx1xx	-	Unallocated.
0x00	1	x0	xxxx1x	-	Unallocated.
0x00	1	x0	xxxxx1	-	Unallocated.
1101	0	1x	1xxxxx	-	*Load/store memory tags* on page C4-291
1x00	1	-	-	-	Unallocated.
xx00	0	0x	-	-	*Load/store exclusive* on page C4-292
xx01	0	1x	0xxxxx	00	*LDAPR/STLR (unscaled immediate)* on page C4-294
xx01	-	0x	-	-	*Load register (literal)* on page C4-295
xx10	-	00	-	-	*Load/store no-allocate pair (offset)* on page C4-296
xx10	-	01	-	-	*Load/store register pair (post-indexed)* on page C4-296
xx10	-	10	-	-	*Load/store register pair (offset)* on page C4-297
xx10	-	11	-	-	*Load/store register pair (pre-indexed)* on page C4-298
xx11	-	0x	0xxxxx	00	*Load/store register (unscaled immediate)* on page C4-298
xx11	-	0x	0xxxxx	01	*Load/store register (immediate post-indexed)* on page C4-300
xx11	-	0x	0xxxxx	10	*Load/store register (unprivileged)* on page C4-301
xx11	-	0x	0xxxxx	11	*Load/store register (immediate pre-indexed)* on page C4-301
xx11	-	0x	1xxxxx	00	*Atomic memory operations* on page C4-303
xx11	-	0x	1xxxxx	10	*Load/store register (register offset)* on page C4-310
xx11	-	0x	1xxxxx	x1	*Load/store register (pac)* on page C4-311
xx11	-	1x	-	-	*Load/store register (unsigned immediate)* on page C4-312

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Advanced SIMD load/store multiple structures

This section describes the encoding of the Advanced SIMD load/store multiple structures instruction class. The encodings in this section are decoded from *Loads and Stores* on page C4-281.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15 12	11 10	9 5	4 0
0	Q	0	0	1	1	0	0	0	L	0	0	0	0	0	0	opcode	size	Rn	Rt

Decode fields		Instruction page
L	opcode	
0	0000	ST4 (multiple structures)
0	0001	Unallocated.
0	0010	ST1 (multiple structures) - Four registers variant
0	0011	Unallocated.
0	0100	ST3 (multiple structures)
0	0101	Unallocated.
0	0110	ST1 (multiple structures) - Three registers variant
0	0111	ST1 (multiple structures) - One register variant
0	1000	ST2 (multiple structures)
0	1001	Unallocated.
0	1010	ST1 (multiple structures) - Two registers variant
0	1011	Unallocated.
0	11xx	Unallocated.
1	0000	LD4 (multiple structures)
1	0001	Unallocated.
1	0010	LD1 (multiple structures) - Four registers variant
1	0011	Unallocated.
1	0100	LD3 (multiple structures)
1	0101	Unallocated.
1	0110	LD1 (multiple structures) - Three registers variant
1	0111	LD1 (multiple structures) - One register variant
1	1000	LD2 (multiple structures)
1	1001	Unallocated.
1	1010	LD1 (multiple structures) - Two registers variant
1	1011	Unallocated.
1	11xx	Unallocated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Advanced SIMD load/store multiple structures (post-indexed)

This section describes the encoding of the Advanced SIMD load/store multiple structures (post-indexed) instruction class. The encodings in this section are decoded from *Loads and Stores* on page C4-281.

31	30	29	28	27	26	25	24	23	22	21	20–16	15–12	11	10	9–5	4–0
0	Q	0	0	1	1	0	0	1	L	0	Rm	opcode	size		Rn	Rt

Decode fields			Instruction page
L	Rm	opcode	
0	-	0001	Unallocated.
0	-	0011	Unallocated.
0	-	0101	Unallocated.
0	-	1001	Unallocated.
0	-	1011	Unallocated.
0	-	11xx	Unallocated.
0	!= 11111	0000	ST4 (multiple structures) - Register offset variant
0	!= 11111	0010	ST1 (multiple structures) - Four registers, register offset variant
0	!= 11111	0100	ST3 (multiple structures) - Register offset variant
0	!= 11111	0110	ST1 (multiple structures) - Three registers, register offset variant
0	!= 11111	0111	ST1 (multiple structures) - One register, register offset variant
0	!= 11111	1000	ST2 (multiple structures) - Register offset variant
0	!= 11111	1010	ST1 (multiple structures) - Two registers, register offset variant
0	11111	0000	ST4 (multiple structures) - Immediate offset variant
0	11111	0010	ST1 (multiple structures) - Four registers, immediate offset variant
0	11111	0100	ST3 (multiple structures) - Immediate offset variant
0	11111	0110	ST1 (multiple structures) - Three registers, immediate offset variant
0	11111	0111	ST1 (multiple structures) - One register, immediate offset variant
0	11111	1000	ST2 (multiple structures) - Immediate offset variant
0	11111	1010	ST1 (multiple structures) - Two registers, immediate offset variant
1	-	0001	Unallocated.
1	-	0011	Unallocated.
1	-	0101	Unallocated.
1	-	1001	Unallocated.
1	-	1011	Unallocated.
1	-	11xx	Unallocated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields			Instruction page
L	Rm	opcode	
1	!= 11111	0000	LD4 (multiple structures) - Register offset variant
1	!= 11111	0010	LD1 (multiple structures) - Four registers, register offset variant
1	!= 11111	0100	LD3 (multiple structures) - Register offset variant
1	!= 11111	0110	LD1 (multiple structures) - Three registers, register offset variant
1	!= 11111	0111	LD1 (multiple structures) - One register, register offset variant
1	!= 11111	1000	LD2 (multiple structures) - Register offset variant
1	!= 11111	1010	LD1 (multiple structures) - Two registers, register offset variant
1	11111	0000	LD4 (multiple structures) - Immediate offset variant
1	11111	0010	LD1 (multiple structures) - Four registers, immediate offset variant
1	11111	0100	LD3 (multiple structures) - Immediate offset variant
1	11111	0110	LD1 (multiple structures) - Three registers, immediate offset variant
1	11111	0111	LD1 (multiple structures) - One register, immediate offset variant
1	11111	1000	LD2 (multiple structures) - Immediate offset variant
1	11111	1010	LD1 (multiple structures) - Two registers, immediate offset variant

Advanced SIMD load/store single structure

This section describes the encoding of the Advanced SIMD load/store single structure instruction class. The encodings in this section are decoded from *Loads and Stores* on page C4-281.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15 13	12	11 10	9 5	4 0
0	Q	0	0	1	1	0	1	0	L	R	0	0	0	0	0	opcode	S	size	Rn	Rt

Decode fields					Instruction page
L	R	opcode	S	size	
0	-	11x	-	-	Unallocated.
0	0	000	-	-	ST1 (single structure) - 8-bit variant
0	0	001	-	-	ST3 (single structure) - 8-bit variant
0	0	010	-	x0	ST1 (single structure) - 16-bit variant
0	0	010	-	x1	Unallocated.
0	0	011	-	x0	ST3 (single structure) - 16-bit variant
0	0	011	-	x1	Unallocated.
0	0	100	-	00	ST1 (single structure) - 32-bit variant
0	0	100	-	1x	Unallocated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields					Instruction page
L	R	opcode	S	size	
0	0	100	0	01	ST1 (single structure) - 64-bit variant
0	0	100	1	01	Unallocated.
0	0	101	-	00	ST3 (single structure) - 32-bit variant
0	0	101	-	10	Unallocated.
0	0	101	0	01	ST3 (single structure) - 64-bit variant
0	0	101	0	11	Unallocated.
0	0	101	1	x1	Unallocated.
0	1	000	-	-	ST2 (single structure) - 8-bit variant
0	1	001	-	-	ST4 (single structure) - 8-bit variant
0	1	010	-	x0	ST2 (single structure) - 16-bit variant
0	1	010	-	x1	Unallocated.
0	1	011	-	x0	ST4 (single structure) - 16-bit variant
0	1	011	-	x1	Unallocated.
0	1	100	-	00	ST2 (single structure) - 32-bit variant
0	1	100	-	10	Unallocated.
0	1	100	0	01	ST2 (single structure) - 64-bit variant
0	1	100	0	11	Unallocated.
0	1	100	1	x1	Unallocated.
0	1	101	-	00	ST4 (single structure) - 32-bit variant
0	1	101	-	10	Unallocated.
0	1	101	0	01	ST4 (single structure) - 64-bit variant
0	1	101	0	11	Unallocated.
0	1	101	1	x1	Unallocated.
1	0	000	-	-	LD1 (single structure) - 8-bit variant
1	0	001	-	-	LD3 (single structure) - 8-bit variant
1	0	010	-	x0	LD1 (single structure) - 16-bit variant
1	0	010	-	x1	Unallocated.
1	0	011	-	x0	LD3 (single structure) - 16-bit variant
1	0	011	-	x1	Unallocated.
1	0	100	-	00	LD1 (single structure) - 32-bit variant
1	0	100	-	1x	Unallocated.
1	0	100	0	01	LD1 (single structure) - 64-bit variant

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields					Instruction page
L	R	opcode	S	size	
1	0	100	1	01	Unallocated.
1	0	101	-	00	LD3 (single structure) - 32-bit variant
1	0	101	-	10	Unallocated.
1	0	101	0	01	LD3 (single structure) - 64-bit variant
1	0	101	0	11	Unallocated.
1	0	101	1	x1	Unallocated.
1	0	110	0	-	LD1R
1	0	110	1	-	Unallocated.
1	0	111	0	-	LD3R
1	0	111	1	-	Unallocated.
1	1	000	-	-	LD2 (single structure) - 8-bit variant
1	1	001	-	-	LD4 (single structure) - 8-bit variant
1	1	010	-	x0	LD2 (single structure) - 16-bit variant
1	1	010	-	x1	Unallocated.
1	1	011	-	x0	LD4 (single structure) - 16-bit variant
1	1	011	-	x1	Unallocated.
1	1	100	-	00	LD2 (single structure) - 32-bit variant
1	1	100	-	10	Unallocated.
1	1	100	0	01	LD2 (single structure) - 64-bit variant
1	1	100	0	11	Unallocated.
1	1	100	1	x1	Unallocated.
1	1	101	-	00	LD4 (single structure) - 32-bit variant
1	1	101	-	10	Unallocated.
1	1	101	0	01	LD4 (single structure) - 64-bit variant
1	1	101	0	11	Unallocated.
1	1	101	1	x1	Unallocated.
1	1	110	0	-	LD2R
1	1	110	1	-	Unallocated.
1	1	111	0	-	LD4R
1	1	111	1	-	Unallocated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Advanced SIMD load/store single structure (post-indexed)

This section describes the encoding of the Advanced SIMD load/store single structure (post-indexed) instruction class. The encodings in this section are decoded from *Loads and Stores* on page C4-281.

31	30	29	28	27	26	25	24	23	22	21	20–16	15–13	12	11 10	9–5	4–0
0	Q	0	0	1	1	0	1	1	L	R	Rm	opcode	S	size	Rn	Rt

Decode fields						Instruction page
L	R	Rm	opcode	S	size	
0	-	-	11x	-	-	Unallocated.
0	0	-	010	-	x1	Unallocated.
0	0	-	011	-	x1	Unallocated.
0	0	-	100	-	1x	Unallocated.
0	0	-	100	1	01	Unallocated.
0	0	-	101	-	10	Unallocated.
0	0	-	101	0	11	Unallocated.
0	0	-	101	1	x1	Unallocated.
0	0	!= 11111	000	-	-	ST1 (single structure) - 8-bit, register offset variant
0	0	!= 11111	001	-	-	ST3 (single structure) - 8-bit, register offset variant
0	0	!= 11111	010	-	x0	ST1 (single structure) - 16-bit, register offset variant
0	0	!= 11111	011	-	x0	ST3 (single structure) - 16-bit, register offset variant
0	0	!= 11111	100	-	00	ST1 (single structure) - 32-bit, register offset variant
0	0	!= 11111	100	0	01	ST1 (single structure) - 64-bit, register offset variant
0	0	!= 11111	101	-	00	ST3 (single structure) - 32-bit, register offset variant
0	0	!= 11111	101	0	01	ST3 (single structure) - 64-bit, register offset variant
0	0	11111	000	-	-	ST1 (single structure) - 8-bit, immediate offset variant
0	0	11111	001	-	-	ST3 (single structure) - 8-bit, immediate offset variant
0	0	11111	010	-	x0	ST1 (single structure) - 16-bit, immediate offset variant
0	0	11111	011	-	x0	ST3 (single structure) - 16-bit, immediate offset variant
0	0	11111	100	-	00	ST1 (single structure) - 32-bit, immediate offset variant
0	0	11111	100	0	01	ST1 (single structure) - 64-bit, immediate offset variant
0	0	11111	101	-	00	ST3 (single structure) - 32-bit, immediate offset variant
0	0	11111	101	0	01	ST3 (single structure) - 64-bit, immediate offset variant
0	1	-	010	-	x1	Unallocated.
0	1	-	011	-	x1	Unallocated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields						Instruction page
L	R	Rm	opcode	S	size	
0	1	-	100	-	10	Unallocated.
0	1	-	100	0	11	Unallocated.
0	1	-	100	1	x1	Unallocated.
0	1	-	101	-	10	Unallocated.
0	1	-	101	0	11	Unallocated.
0	1	-	101	1	x1	Unallocated.
0	1	!= 11111	000	-	-	ST2 (single structure) - 8-bit, register offset variant
0	1	!= 11111	001	-	-	ST4 (single structure) - 8-bit, register offset variant
0	1	!= 11111	010	-	x0	ST2 (single structure) - 16-bit, register offset variant
0	1	!= 11111	011	-	x0	ST4 (single structure) - 16-bit, register offset variant
0	1	!= 11111	100	-	00	ST2 (single structure) - 32-bit, register offset variant
0	1	!= 11111	100	0	01	ST2 (single structure) - 64-bit, register offset variant
0	1	!= 11111	101	-	00	ST4 (single structure) - 32-bit, register offset variant
0	1	!= 11111	101	0	01	ST4 (single structure) - 64-bit, register offset variant
0	1	11111	000	-	-	ST2 (single structure) - 8-bit, immediate offset variant
0	1	11111	001	-	-	ST4 (single structure) - 8-bit, immediate offset variant
0	1	11111	010	-	x0	ST2 (single structure) - 16-bit, immediate offset variant
0	1	11111	011	-	x0	ST4 (single structure) - 16-bit, immediate offset variant
0	1	11111	100	-	00	ST2 (single structure) - 32-bit, immediate offset variant
0	1	11111	100	0	01	ST2 (single structure) - 64-bit, immediate offset variant
0	1	11111	101	-	00	ST4 (single structure) - 32-bit, immediate offset variant
0	1	11111	101	0	01	ST4 (single structure) - 64-bit, immediate offset variant
1	0	-	010	-	x1	Unallocated.
1	0	-	011	-	x1	Unallocated.
1	0	-	100	-	1x	Unallocated.
1	0	-	100	1	01	Unallocated.
1	0	-	101	-	10	Unallocated.
1	0	-	101	0	11	Unallocated.
1	0	-	101	1	x1	Unallocated.
1	0	-	110	1	-	Unallocated.
1	0	-	111	1	-	Unallocated.
1	0	!= 11111	000	-	-	LD1 (single structure) - 8-bit, register offset variant

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields						Instruction page
L	R	Rm	opcode	S	size	
1	0	!= 11111	001	-	-	LD3 (single structure) - 8-bit, register offset variant
1	0	!= 11111	010	-	x0	LD1 (single structure) - 16-bit, register offset variant
1	0	!= 11111	011	-	x0	LD3 (single structure) - 16-bit, register offset variant
1	0	!= 11111	100	-	00	LD1 (single structure) - 32-bit, register offset variant
1	0	!= 11111	100	0	01	LD1 (single structure) - 64-bit, register offset variant
1	0	!= 11111	101	-	00	LD3 (single structure) - 32-bit, register offset variant
1	0	!= 11111	101	0	01	LD3 (single structure) - 64-bit, register offset variant
1	0	!= 11111	110	0	-	LD1R - Register offset variant
1	0	!= 11111	111	0	-	LD3R - Register offset variant
1	0	11111	000	-	-	LD1 (single structure) - 8-bit, immediate offset variant
1	0	11111	001	-	-	LD3 (single structure) - 8-bit, immediate offset variant
1	0	11111	010	-	x0	LD1 (single structure) - 16-bit, immediate offset variant
1	0	11111	011	-	x0	LD3 (single structure) - 16-bit, immediate offset variant
1	0	11111	100	-	00	LD1 (single structure) - 32-bit, immediate offset variant
1	0	11111	100	0	01	LD1 (single structure) - 64-bit, immediate offset variant
1	0	11111	101	-	00	LD3 (single structure) - 32-bit, immediate offset variant
1	0	11111	101	0	01	LD3 (single structure) - 64-bit, immediate offset variant
1	0	11111	110	0	-	LD1R - Immediate offset variant
1	0	11111	111	0	-	LD3R - Immediate offset variant
1	1	-	010	-	x1	Unallocated.
1	1	-	011	-	x1	Unallocated.
1	1	-	100	-	10	Unallocated.
1	1	-	100	0	11	Unallocated.
1	1	-	100	1	x1	Unallocated.
1	1	-	101	-	10	Unallocated.
1	1	-	101	0	11	Unallocated.
1	1	-	101	1	x1	Unallocated.
1	1	-	110	1	-	Unallocated.
1	1	-	111	1	-	Unallocated.
1	1	!= 11111	000	-	-	LD2 (single structure) - 8-bit, register offset variant
1	1	!= 11111	001	-	-	LD4 (single structure) - 8-bit, register offset variant
1	1	!= 11111	010	-	x0	LD2 (single structure) - 16-bit, register offset variant

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields						Instruction page
L	R	Rm	opcode	S	size	
1	1	!= 11111	011	-	x0	LD4 (single structure) - 16-bit, register offset variant
1	1	!= 11111	100	-	00	LD2 (single structure) - 32-bit, register offset variant
1	1	!= 11111	100	0	01	LD2 (single structure) - 64-bit, register offset variant
1	1	!= 11111	101	-	00	LD4 (single structure) - 32-bit, register offset variant
1	1	!= 11111	101	0	01	LD4 (single structure) - 64-bit, register offset variant
1	1	!= 11111	110	0	-	LD2R - Register offset variant
1	1	!= 11111	111	0	-	LD4R - Register offset variant
1	1	11111	000	-	-	LD2 (single structure) - 8-bit, immediate offset variant
1	1	11111	001	-	-	LD4 (single structure) - 8-bit, immediate offset variant
1	1	11111	010	-	x0	LD2 (single structure) - 16-bit, immediate offset variant
1	1	11111	011	-	x0	LD4 (single structure) - 16-bit, immediate offset variant
1	1	11111	100	-	00	LD2 (single structure) - 32-bit, immediate offset variant
1	1	11111	100	0	01	LD2 (single structure) - 64-bit, immediate offset variant
1	1	11111	101	-	00	LD4 (single structure) - 32-bit, immediate offset variant
1	1	11111	101	0	01	LD4 (single structure) - 64-bit, immediate offset variant
1	1	11111	110	0	-	LD2R - Immediate offset variant
1	1	11111	111	0	-	LD4R - Immediate offset variant

Load/store memory tags

This section describes the encoding of the Load/store memory tags instruction class. The encodings in this section are decoded from *Loads and Stores* on page C4-281.

31 30 29 28	27 26 25 24	23 22 21 20		12	11 10 9	5 4	0
1 1 0 1	1 0 0 1	opc	1	imm9	op2	Rn	Rt

Decode fields			Instruction page	Architecture version
opc	imm9	op2		
00	-	01	STG - Post-index variant	ARMv8.5-MemTag
00	-	10	STG - Signed offset variant	ARMv8.5-MemTag
00	-	11	STG - Pre-index variant	ARMv8.5-MemTag
00	000000000	00	STZGM	ARMv8.5-MemTag
01	-	00	LDG	ARMv8.5-MemTag
01	-	01	STZG - Post-index variant	ARMv8.5-MemTag

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields			Instruction page	Architecture version
opc	imm9	op2		
01	-	10	STZG - Signed offset variant	ARMv8.5-MemTag
01	-	11	STZG - Pre-index variant	ARMv8.5-MemTag
10	-	01	ST2G - Post-index variant	ARMv8.5-MemTag
10	-	10	ST2G - Signed offset variant	ARMv8.5-MemTag
10	-	11	ST2G - Pre-index variant	ARMv8.5-MemTag
10	!= 000000000	00	Unallocated.	-
10	000000000	00	STGM	ARMv8.5-MemTag
11	-	01	STZ2G - Post-index variant	ARMv8.5-MemTag
11	-	10	STZ2G - Signed offset variant	ARMv8.5-MemTag
11	-	11	STZ2G - Pre-index variant	ARMv8.5-MemTag
11	!= 000000000	00	Unallocated.	-
11	000000000	00	LDGM	ARMv8.5-MemTag

Load/store exclusive

This section describes the encoding of the Load/store exclusive instruction class. The encodings in this section are decoded from *Loads and Stores* on page C4-281.

31 30	29 28	27 26	25 24	23	22	21	20 16	15	14 10	9 5	4 0
size	0 0	1 0	0 0	o2	L	o1	Rs	o0	Rt2	Rn	Rt

Decode fields						Instruction page	Architecture version
size	o2	L	o1	o0	Rt2		
-	1	-	1	-	!= 11111	Unallocated.	-
0x	0	-	1	-	!= 11111	Unallocated.	-
00	0	0	0	0	-	STXRB	-
00	0	0	0	1	-	STLXRB	-
00	0	0	1	0	11111	CASP, CASPA, CASPAL, CASPL - 32-bit CASP variant	ARMv8.1-LSE
00	0	0	1	1	11111	CASP, CASPA, CASPAL, CASPL - 32-bit CASPL variant	ARMv8.1-LSE
00	0	1	0	0	-	LDXRB	-
00	0	1	0	1	-	LDAXRB	-
00	0	1	1	0	11111	CASP, CASPA, CASPAL, CASPL - 32-bit CASPA variant	ARMv8.1-LSE
00	0	1	1	1	11111	CASP, CASPA, CASPAL, CASPL - 32-bit CASPAL variant	ARMv8.1-LSE
00	1	0	0	0	-	STLLRB	ARMv8.1-LOR

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields						Instruction page	Architecture version
size	o2	L	o1	o0	Rt2		
00	1	0	0	1	-	STLRB	-
00	1	0	1	0	11111	CASB, CASAB, CASALB, CASLB - CASB variant	ARMv8.1-LSE
00	1	0	1	1	11111	CASB, CASAB, CASALB, CASLB - CASLB variant	ARMv8.1-LSE
00	1	1	0	0	-	LDLARB	ARMv8.1-LOR
00	1	1	0	1	-	LDARB	-
00	1	1	1	0	11111	CASB, CASAB, CASALB, CASLB - CASAB variant	ARMv8.1-LSE
00	1	1	1	1	11111	CASB, CASAB, CASALB, CASLB - CASALB variant	ARMv8.1-LSE
01	0	0	0	0	-	STXRH	-
01	0	0	0	1	-	STLXRH	-
01	0	0	1	0	11111	CASP, CASPA, CASPAL, CASPL - 64-bit CASP variant	ARMv8.1-LSE
01	0	0	1	1	11111	CASP, CASPA, CASPAL, CASPL - 64-bit CASPL variant	ARMv8.1-LSE
01	0	1	0	0	-	LDXRH	-
01	0	1	0	1	-	LDAXRH	-
01	0	1	1	0	11111	CASP, CASPA, CASPAL, CASPL - 64-bit CASPA variant	ARMv8.1-LSE
01	0	1	1	1	11111	CASP, CASPA, CASPAL, CASPL - 64-bit CASPAL variant	ARMv8.1-LSE
01	1	0	0	0	-	STLLRH	ARMv8.1-LOR
01	1	0	0	1	-	STLRH	-
01	1	0	1	0	11111	CASH, CASAH, CASALH, CASLH - CASH variant	ARMv8.1-LSE
01	1	0	1	1	11111	CASH, CASAH, CASALH, CASLH - CASLH variant	ARMv8.1-LSE
01	1	1	0	0	-	LDLARH	ARMv8.1-LOR
01	1	1	0	1	-	LDARH	-
01	1	1	1	0	11111	CASH, CASAH, CASALH, CASLH - CASAH variant	ARMv8.1-LSE
01	1	1	1	1	11111	CASH, CASAH, CASALH, CASLH - CASALH variant	ARMv8.1-LSE
10	0	0	0	0	-	STXR - 32-bit variant	-
10	0	0	0	1	-	STLXR - 32-bit variant	-
10	0	0	1	0	-	STXP - 32-bit variant	-
10	0	0	1	1	-	STLXP - 32-bit variant	-
10	0	1	0	0	-	LDXR - 32-bit variant	-
10	0	1	0	1	-	LDAXR - 32-bit variant	-
10	0	1	1	0	-	LDXP - 32-bit variant	-
10	0	1	1	1	-	LDAXP - 32-bit variant	-
10	1	0	0	0	-	STLLR - 32-bit variant	ARMv8.1-LOR

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields						Instruction page	Architecture version
size	o2	L	o1	o0	Rt2		
10	1	0	0	1	-	STLR - 32-bit variant	-
10	1	0	1	0	11111	CAS, CASA, CASAL, CASL - 32-bit CAS variant	ARMv8.1-LSE
10	1	0	1	1	11111	CAS, CASA, CASAL, CASL - 32-bit CASL variant	ARMv8.1-LSE
10	1	1	0	0	-	LDLAR - 32-bit variant	ARMv8.1-LOR
10	1	1	0	1	-	LDAR - 32-bit variant	-
10	1	1	1	0	11111	CAS, CASA, CASAL, CASL - 32-bit CASA variant	ARMv8.1-LSE
10	1	1	1	1	11111	CAS, CASA, CASAL, CASL - 32-bit CASAL variant	ARMv8.1-LSE
11	0	0	0	0	-	STXR - 64-bit variant	-
11	0	0	0	1	-	STLXR - 64-bit variant	-
11	0	0	1	0	-	STXP - 64-bit variant	-
11	0	0	1	1	-	STLXP - 64-bit variant	-
11	0	1	0	0	-	LDXR - 64-bit variant	-
11	0	1	0	1	-	LDAXR - 64-bit variant	-
11	0	1	1	0	-	LDXP - 64-bit variant	-
11	0	1	1	1	-	LDAXP - 64-bit variant	-
11	1	0	0	0	-	STLLR - 64-bit variant	ARMv8.1-LOR
11	1	0	0	1	-	STLR - 64-bit variant	-
11	1	0	1	0	11111	CAS, CASA, CASAL, CASL - 64-bit CAS variant	ARMv8.1-LSE
11	1	0	1	1	11111	CAS, CASA, CASAL, CASL - 64-bit CASL variant	ARMv8.1-LSE
11	1	1	0	0	-	LDLAR - 64-bit variant	ARMv8.1-LOR
11	1	1	0	1	-	LDAR - 64-bit variant	-
11	1	1	1	0	11111	CAS, CASA, CASAL, CASL - 64-bit CASA variant	ARMv8.1-LSE
11	1	1	1	1	11111	CAS, CASA, CASAL, CASL - 64-bit CASAL variant	ARMv8.1-LSE

LDAPR/STLR (unscaled immediate)

This section describes the encoding of the LDAPR/STLR (unscaled immediate) instruction class. The encodings in this section are decoded from *Loads and Stores* on page C4-281.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31 30 29 28	27 26 25 24	23 22 21 20	12	11 10 9	5 4	0
size	0 1 1 0 0 1	opc	0	imm9	0 0	Rn	Rt

Decode fields		Instruction page	Architecture version
size	opc		
00	00	STLURB	ARMv8.4-RCPC
00	01	LDAPURB	ARMv8.4-RCPC
00	10	LDAPURSB - 64-bit variant	ARMv8.4-RCPC
00	11	LDAPURSB - 32-bit variant	ARMv8.4-RCPC
01	00	STLURH	ARMv8.4-RCPC
01	01	LDAPURH	ARMv8.4-RCPC
01	10	LDAPURSH - 64-bit variant	ARMv8.4-RCPC
01	11	LDAPURSH - 32-bit variant	ARMv8.4-RCPC
10	00	STLUR - 32-bit variant	ARMv8.4-RCPC
10	01	LDAPUR - 32-bit variant	ARMv8.4-RCPC
10	10	LDAPURSW	ARMv8.4-RCPC
10	11	Unallocated.	-
11	00	STLUR - 64-bit variant	ARMv8.4-RCPC
11	01	LDAPUR - 64-bit variant	ARMv8.4-RCPC
11	10	Unallocated.	-
11	11	Unallocated.	-

Load register (literal)

This section describes the encoding of the Load register (literal) instruction class. The encodings in this section are decoded from *Loads and Stores* on page C4-281.

31 30 29 28	27 26 25 24	23	5 4	0
opc	0 1 1 V 0 0	imm19	Rt

Decode fields		Instruction page
opc	V	
00	0	LDR (literal) - 32-bit variant
00	1	LDR (literal, SIMD&FP) - 32-bit variant
01	0	LDR (literal) - 64-bit variant

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields		Instruction page
opc	V	
01	1	LDR (literal, SIMD&FP) - 64-bit variant
10	0	LDRSW (literal)
10	1	LDR (literal, SIMD&FP) - 128-bit variant
11	0	PRFM (literal)
11	1	Unallocated.

Load/store no-allocate pair (offset)

This section describes the encoding of the Load/store no-allocate pair (offset) instruction class. The encodings in this section are decoded from *Loads and Stores* on page C4-281.

31 30	29 28 27	26	25 24 23	22	21 15	14 10	9 5	4 0
opc	1 0 1	V	0 0 0	L	imm7	Rt2	Rn	Rt

Decode fields			Instruction page
opc	V	L	
00	0	0	STNP - 32-bit variant
00	0	1	LDNP - 32-bit variant
00	1	0	STNP (SIMD&FP) - 32-bit variant
00	1	1	LDNP (SIMD&FP) - 32-bit variant
01	0	-	Unallocated.
01	1	0	STNP (SIMD&FP) - 64-bit variant
01	1	1	LDNP (SIMD&FP) - 64-bit variant
10	0	0	STNP - 64-bit variant
10	0	1	LDNP - 64-bit variant
10	1	0	STNP (SIMD&FP) - 128-bit variant
10	1	1	LDNP (SIMD&FP) - 128-bit variant
11	-	-	Unallocated.

Load/store register pair (post-indexed)

This section describes the encoding of the Load/store register pair (post-indexed) instruction class. The encodings in this section are decoded from *Loads and Stores* on page C4-281.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31 30 29 28	27 26 25 24	23 22 21	15 14	10 9	5 4	0
opc	1 0 1 V	0 0 1 L	imm7	Rt2	Rn	Rt

Decode fields			Instruction page	Architecture version
opc	V	L		
00	0	0	STP - 32-bit variant	-
00	0	1	LDP - 32-bit variant	-
00	1	0	STP (SIMD&FP) - 32-bit variant	-
00	1	1	LDP (SIMD&FP) - 32-bit variant	-
01	0	0	STGP	ARMv8.5-MemTag
01	0	1	LDPSW	-
01	1	0	STP (SIMD&FP) - 64-bit variant	-
01	1	1	LDP (SIMD&FP) - 64-bit variant	-
10	0	0	STP - 64-bit variant	-
10	0	1	LDP - 64-bit variant	-
10	1	0	STP (SIMD&FP) - 128-bit variant	-
10	1	1	LDP (SIMD&FP) - 128-bit variant	-
11	-	-	Unallocated.	-

Load/store register pair (offset)

This section describes the encoding of the Load/store register pair (offset) instruction class. The encodings in this section are decoded from *Loads and Stores* on page C4-281.

31 30 29 28	27 26 25 24	23 22 21	15 14	10 9	5 4	0
opc	1 0 1 V	0 1 0 L	imm7	Rt2	Rn	Rt

Decode fields			Instruction page	Architecture version
opc	V	L		
00	0	0	STP - 32-bit variant	-
00	0	1	LDP - 32-bit variant	-
00	1	0	STP (SIMD&FP) - 32-bit variant	-
00	1	1	LDP (SIMD&FP) - 32-bit variant	-
01	0	0	STGP	ARMv8.5-MemTag
01	0	1	LDPSW	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields			Instruction page	Architecture version
opc	V	L		
01	1	0	STP (SIMD&FP) - 64-bit variant	-
01	1	1	LDP (SIMD&FP) - 64-bit variant	-
10	0	0	STP - 64-bit variant	-
10	0	1	LDP - 64-bit variant	-
10	1	0	STP (SIMD&FP) - 128-bit variant	-
10	1	1	LDP (SIMD&FP) - 128-bit variant	-
11	-	-	Unallocated.	-

Load/store register pair (pre-indexed)

This section describes the encoding of the Load/store register pair (pre-indexed) instruction class. The encodings in this section are decoded from *Loads and Stores* on page C4-281.

31 30 29 28	27 26 25 24	23 22 21	15 14	10 9	5 4	0
opc	1 0 1 V 0 1 1	L	imm7	Rt2	Rn	Rt

Decode fields			Instruction page	Architecture version
opc	V	L		
00	0	0	STP - 32-bit variant	-
00	0	1	LDP - 32-bit variant	-
00	1	0	STP (SIMD&FP) - 32-bit variant	-
00	1	1	LDP (SIMD&FP) - 32-bit variant	-
01	0	0	STGP	ARMv8.5-MemTag
01	0	1	LDPSW	-
01	1	0	STP (SIMD&FP) - 64-bit variant	-
01	1	1	LDP (SIMD&FP) - 64-bit variant	-
10	0	0	STP - 64-bit variant	-
10	0	1	LDP - 64-bit variant	-
10	1	0	STP (SIMD&FP) - 128-bit variant	-
10	1	1	LDP (SIMD&FP) - 128-bit variant	-
11	-	-	Unallocated.	-

Load/store register (unscaled immediate)

This section describes the encoding of the Load/store register (unscaled immediate) instruction class. The encodings in this section are decoded from *Loads and Stores* on page C4-281.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31 30 29 28	27 26	25 24	23 22 21 20	12	11 10	9 5	4 0
size	1 1 1	V	0 0	opc	0	imm9	0 0	Rn	Rt

Decode fields			Instruction page
size	V	opc	
x1	1	1x	Unallocated.
00	0	00	STURB
00	0	01	LDURB
00	0	10	LDURSB - 64-bit variant
00	0	11	LDURSB - 32-bit variant
00	1	00	STUR (SIMD&FP) - 8-bit variant
00	1	01	LDUR (SIMD&FP) - 8-bit variant
00	1	10	STUR (SIMD&FP) - 128-bit variant
00	1	11	LDUR (SIMD&FP) - 128-bit variant
01	0	00	STURH
01	0	01	LDURH
01	0	10	LDURSH - 64-bit variant
01	0	11	LDURSH - 32-bit variant
01	1	00	STUR (SIMD&FP) - 16-bit variant
01	1	01	LDUR (SIMD&FP) - 16-bit variant
1x	0	11	Unallocated.
1x	1	1x	Unallocated.
10	0	00	STUR - 32-bit variant
10	0	01	LDUR - 32-bit variant
10	0	10	LDURSW
10	1	00	STUR (SIMD&FP) - 32-bit variant
10	1	01	LDUR (SIMD&FP) - 32-bit variant
11	0	00	STUR - 64-bit variant
11	0	01	LDUR - 64-bit variant
11	0	10	PRFUM
11	1	00	STUR (SIMD&FP) - 64-bit variant
11	1	01	LDUR (SIMD&FP) - 64-bit variant

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Load/store register (immediate post-indexed)

This section describes the encoding of the Load/store register (immediate post-indexed) instruction class. The encodings in this section are decoded from *Loads and Stores* on page C4-281.

31 30	29 28	27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
size	1 1	1	V	0 0	opc	0	imm9	0 1	Rn	Rt

Decode fields			Instruction page
size	V	opc	
x1	1	1x	Unallocated.
00	0	00	STRB (immediate)
00	0	01	LDRB (immediate)
00	0	10	LDRSB (immediate) - 64-bit variant
00	0	11	LDRSB (immediate) - 32-bit variant
00	1	00	STR (immediate, SIMD&FP) - 8-bit variant
00	1	01	LDR (immediate, SIMD&FP) - 8-bit variant
00	1	10	STR (immediate, SIMD&FP) - 128-bit variant
00	1	11	LDR (immediate, SIMD&FP) - 128-bit variant
01	0	00	STRH (immediate)
01	0	01	LDRH (immediate)
01	0	10	LDRSH (immediate) - 64-bit variant
01	0	11	LDRSH (immediate) - 32-bit variant
01	1	00	STR (immediate, SIMD&FP) - 16-bit variant
01	1	01	LDR (immediate, SIMD&FP) - 16-bit variant
1x	0	11	Unallocated.
1x	1	1x	Unallocated.
10	0	00	STR (immediate) - 32-bit variant
10	0	01	LDR (immediate) - 32-bit variant
10	0	10	LDRSW (immediate)
10	1	00	STR (immediate, SIMD&FP) - 32-bit variant
10	1	01	LDR (immediate, SIMD&FP) - 32-bit variant
11	0	00	STR (immediate) - 64-bit variant
11	0	01	LDR (immediate) - 64-bit variant

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields			Instruction page
size	V	opc	
11	0	10	Unallocated.
11	1	00	STR (immediate, SIMD&FP) - 64-bit variant
11	1	01	LDR (immediate, SIMD&FP) - 64-bit variant

Load/store register (unprivileged)

This section describes the encoding of the Load/store register (unprivileged) instruction class. The encodings in this section are decoded from *Loads and Stores* on page C4-281.

31 30	29 28	27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
size	1 1	1	V	0 0	opc	0	imm9	1 0	Rn	Rt

Decode fields			Instruction page
size	V	opc	
-	1	-	Unallocated.
00	0	00	STTRB
00	0	01	LDTRB
00	0	10	LDTRSB - 64-bit variant
00	0	11	LDTRSB - 32-bit variant
01	0	00	STTRH
01	0	01	LDTRH
01	0	10	LDTRSH - 64-bit variant
01	0	11	LDTRSH - 32-bit variant
1x	0	11	Unallocated.
10	0	00	STTR - 32-bit variant
10	0	01	LDTR - 32-bit variant
10	0	10	LDTRSW
11	0	00	STTR - 64-bit variant
11	0	01	LDTR - 64-bit variant
11	0	10	Unallocated.

Load/store register (immediate pre-indexed)

This section describes the encoding of the Load/store register (immediate pre-indexed) instruction class. The encodings in this section are decoded from *Loads and Stores* on page C4-281.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31 30	29 28	27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
size	1 1	1	V	0 0	opc	0	imm9	1 1	Rn	Rt

Decode fields			Instruction page
size	V	opc	
x1	1	1x	Unallocated.
00	0	00	STRB (immediate)
00	0	01	LDRB (immediate)
00	0	10	LDRSB (immediate) - 64-bit variant
00	0	11	LDRSB (immediate) - 32-bit variant
00	1	00	STR (immediate, SIMD&FP) - 8-bit variant
00	1	01	LDR (immediate, SIMD&FP) - 8-bit variant
00	1	10	STR (immediate, SIMD&FP) - 128-bit variant
00	1	11	LDR (immediate, SIMD&FP) - 128-bit variant
01	0	00	STRH (immediate)
01	0	01	LDRH (immediate)
01	0	10	LDRSH (immediate) - 64-bit variant
01	0	11	LDRSH (immediate) - 32-bit variant
01	1	00	STR (immediate, SIMD&FP) - 16-bit variant
01	1	01	LDR (immediate, SIMD&FP) - 16-bit variant
1x	0	11	Unallocated.
1x	1	1x	Unallocated.
10	0	00	STR (immediate) - 32-bit variant
10	0	01	LDR (immediate) - 32-bit variant
10	0	10	LDRSW (immediate)
10	1	00	STR (immediate, SIMD&FP) - 32-bit variant
10	1	01	LDR (immediate, SIMD&FP) - 32-bit variant
11	0	00	STR (immediate) - 64-bit variant
11	0	01	LDR (immediate) - 64-bit variant
11	0	10	Unallocated.
11	1	00	STR (immediate, SIMD&FP) - 64-bit variant
11	1	01	LDR (immediate, SIMD&FP) - 64-bit variant

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Atomic memory operations

This section describes the encoding of the Atomic memory operations instruction class. The encodings in this section are decoded from *Loads and Stores* on page C4-281.

31 30 29 28	27 26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
size	1 1 1	V	0 0	A	R	1	Rs	o3	opc	0 0	Rn	Rt

Decode fields						Instruction page	Architecture version
size	V	A	R	o3	opc		
-	0	-	-	1	001	Unallocated.	-
-	0	-	-	1	01x	Unallocated.	-
-	0	-	-	1	101	Unallocated.	-
-	0	-	-	1	11x	Unallocated.	-
-	0	0	-	1	100	Unallocated.	-
-	0	1	1	1	100	Unallocated.	-
-	1	-	-	-	-	Unallocated.	-
00	0	0	0	0	000	LDADDB, LDADDAB, LDADDALB, LDADDLB - LDADDB variant	ARMv8.1-LSE
00	0	0	0	0	001	LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB - LDCLRB variant	ARMv8.1-LSE
00	0	0	0	0	010	LDEORB, LDEORAB, LDEORALB, LDEORLB - LDEORB variant	ARMv8.1-LSE
00	0	0	0	0	011	LDSETB, LDSETAB, LDSETALB, LDSETLB - LDSETB variant	ARMv8.1-LSE
00	0	0	0	0	100	LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB - LDSMAXB variant	ARMv8.1-LSE
00	0	0	0	0	101	LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB - LDSMINB variant	ARMv8.1-LSE
00	0	0	0	0	110	LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB - LDUMAXB variant	ARMv8.1-LSE
00	0	0	0	0	111	LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB - LDUMINB variant	ARMv8.1-LSE
00	0	0	0	1	000	SWPB, SWPAB, SWPALB, SWPLB - SWPB variant	ARMv8.1-LSE
00	0	0	1	0	000	LDADDB, LDADDAB, LDADDALB, LDADDLB - LDADDLB variant	ARMv8.1-LSE
00	0	0	1	0	001	LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB - LDCLRLB variant	ARMv8.1-LSE
00	0	0	1	0	010	LDEORB, LDEORAB, LDEORALB, LDEORLB - LDEORLB variant	ARMv8.1-LSE
00	0	0	1	0	011	LDSETB, LDSETAB, LDSETALB, LDSETLB - LDSETLB variant	ARMv8.1-LSE

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields						Instruction page	Architecture version
size	V	A	R	o3	opc		
00	0	0	1	0	100	LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB - LDSMAXLB variant	ARMv8.1-LSE
00	0	0	1	0	101	LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB - LDSMINLB variant	ARMv8.1-LSE
00	0	0	1	0	110	LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB - LDUMAXLB variant	ARMv8.1-LSE
00	0	0	1	0	111	LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB - LDUMINLB variant	ARMv8.1-LSE
00	0	0	1	1	000	SWPB, SWPAB, SWPALB, SWPLB - SWPLB variant	ARMv8.1-LSE
00	0	1	0	0	000	LDADDB, LDADDAB, LDADDALB, LDADDLB - LDADDAB variant	ARMv8.1-LSE
00	0	1	0	0	001	LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB - LDCLRAB variant	ARMv8.1-LSE
00	0	1	0	0	010	LDEORB, LDEORAB, LDEORALB, LDEORLB - LDEORAB variant	ARMv8.1-LSE
00	0	1	0	0	011	LDSETB, LDSETAB, LDSETALB, LDSETLB - LDSETAB variant	ARMv8.1-LSE
00	0	1	0	0	100	LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB - LDSMAXAB variant	ARMv8.1-LSE
00	0	1	0	0	101	LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB - LDSMINAB variant	ARMv8.1-LSE
00	0	1	0	0	110	LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB - LDUMAXAB variant	ARMv8.1-LSE
00	0	1	0	0	111	LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB - LDUMINAB variant	ARMv8.1-LSE
00	0	1	0	1	000	SWPB, SWPAB, SWPALB, SWPLB - SWPAB variant	ARMv8.1-LSE
00	0	1	0	1	100	LDAPRB	ARMv8.3-RCPC
00	0	1	1	0	000	LDADDB, LDADDAB, LDADDALB, LDADDLB - LDADDALB variant	ARMv8.1-LSE
00	0	1	1	0	001	LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB - LDCLRALB variant	ARMv8.1-LSE
00	0	1	1	0	010	LDEORB, LDEORAB, LDEORALB, LDEORLB - LDEORALB variant	ARMv8.1-LSE
00	0	1	1	0	011	LDSETB, LDSETAB, LDSETALB, LDSETLB - LDSETALB variant	ARMv8.1-LSE
00	0	1	1	0	100	LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB - LDSMAXALB variant	ARMv8.1-LSE
00	0	1	1	0	101	LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB - LDSMINALB variant	ARMv8.1-LSE

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields						Instruction page	Architecture version
size	V	A	R	o3	opc		
00	0	1	1	0	110	LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB - LDUMAXALB variant	ARMv8.1-LSE
00	0	1	1	0	111	LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB - LDUMINALB variant	ARMv8.1-LSE
00	0	1	1	1	000	SWPB, SWPAB, SWPALB, SWPLB - SWPALB variant	ARMv8.1-LSE
01	0	0	0	0	000	LDADDH, LDADDAH, LDADDALH, LDADDLH - LDADDH variant	ARMv8.1-LSE
01	0	0	0	0	001	LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH - LDCLRH variant	ARMv8.1-LSE
01	0	0	0	0	010	LDEORH, LDEORAH, LDEORALH, LDEORLH - LDEORH variant	ARMv8.1-LSE
01	0	0	0	0	011	LDSETH, LDSETAH, LDSETALH, LDSETLH - LDSETH variant	ARMv8.1-LSE
01	0	0	0	0	100	LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH - LDSMAXH variant	ARMv8.1-LSE
01	0	0	0	0	101	LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH - LDSMINH variant	ARMv8.1-LSE
01	0	0	0	0	110	LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH - LDUMAXH variant	ARMv8.1-LSE
01	0	0	0	0	111	LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH - LDUMINH variant	ARMv8.1-LSE
01	0	0	0	1	000	SWPH, SWPAH, SWPALH, SWPLH - SWPH variant	ARMv8.1-LSE
01	0	0	1	0	000	LDADDH, LDADDAH, LDADDALH, LDADDLH - LDADDLH variant	ARMv8.1-LSE
01	0	0	1	0	001	LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH - LDCLRLH variant	ARMv8.1-LSE
01	0	0	1	0	010	LDEORH, LDEORAH, LDEORALH, LDEORLH - LDEORLH variant	ARMv8.1-LSE
01	0	0	1	0	011	LDSETH, LDSETAH, LDSETALH, LDSETLH - LDSETLH variant	ARMv8.1-LSE
01	0	0	1	0	100	LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH - LDSMAXLH variant	ARMv8.1-LSE
01	0	0	1	0	101	LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH - LDSMINLH variant	ARMv8.1-LSE
01	0	0	1	0	110	LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH - LDUMAXLH variant	ARMv8.1-LSE
01	0	0	1	0	111	LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH - LDUMINLH variant	ARMv8.1-LSE
01	0	0	1	1	000	SWPH, SWPAH, SWPALH, SWPLH - SWPLH variant	ARMv8.1-LSE

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields						Instruction page	Architecture version
size	V	A	R	o3	opc		
01	0	1	0	0	000	LDADDH, LDADDAH, LDADDALH, LDADDLH - LDADDAH variant	ARMv8.1-LSE
01	0	1	0	0	001	LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH - LDCLRAH variant	ARMv8.1-LSE
01	0	1	0	0	010	LDEORH, LDEORAH, LDEORALH, LDEORLH - LDEORAH variant	ARMv8.1-LSE
01	0	1	0	0	011	LDSETH, LDSETAH, LDSETALH, LDSETLH - LDSETAH variant	ARMv8.1-LSE
01	0	1	0	0	100	LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH - LDSMAXAH variant	ARMv8.1-LSE
01	0	1	0	0	101	LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH - LDSMINAH variant	ARMv8.1-LSE
01	0	1	0	0	110	LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH - LDUMAXAH variant	ARMv8.1-LSE
01	0	1	0	0	111	LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH - LDUMINAH variant	ARMv8.1-LSE
01	0	1	0	1	000	SWPH, SWPAH, SWPALH, SWPLH - SWPAH variant	ARMv8.1-LSE
01	0	1	0	1	100	LDAPRH	ARMv8.3-RCPC
01	0	1	1	0	000	LDADDH, LDADDAH, LDADDALH, LDADDLH - LDADDALH variant	ARMv8.1-LSE
01	0	1	1	0	001	LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH - LDCLRALH variant	ARMv8.1-LSE
01	0	1	1	0	010	LDEORH, LDEORAH, LDEORALH, LDEORLH - LDEORALH variant	ARMv8.1-LSE
01	0	1	1	0	011	LDSETH, LDSETAH, LDSETALH, LDSETLH - LDSETALH variant	ARMv8.1-LSE
01	0	1	1	0	100	LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH - LDSMAXALH variant	ARMv8.1-LSE
01	0	1	1	0	101	LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH - LDSMINALH variant	ARMv8.1-LSE
01	0	1	1	0	110	LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH - LDUMAXALH variant	ARMv8.1-LSE
01	0	1	1	0	111	LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH - LDUMINALH variant	ARMv8.1-LSE
01	0	1	1	1	000	SWPH, SWPAH, SWPALH, SWPLH - SWPALH variant	ARMv8.1-LSE
10	0	0	0	0	000	LDADD, LDADDA, LDADDAL, LDADDL - 32-bit LDADD variant	ARMv8.1-LSE
10	0	0	0	0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL - 32-bit LDCLR variant	ARMv8.1-LSE
10	0	0	0	0	010	LDEOR, LDEORA, LDEORAL, LDEORL - 32-bit LDEOR variant	ARMv8.1-LSE

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields						Instruction page	Architecture version
size	V	A	R	o3	opc		
10	0	0	0	0	011	LDSET, LDSETA, LDSETAL, LDSETL - 32-bit LDSET variant	ARMv8.1-LSE
10	0	0	0	0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL - 32-bit LDSMAX variant	ARMv8.1-LSE
10	0	0	0	0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL - 32-bit LDSMIN variant	ARMv8.1-LSE
10	0	0	0	0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL - 32-bit LDUMAX variant	ARMv8.1-LSE
10	0	0	0	0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL - 32-bit LDUMIN variant	ARMv8.1-LSE
10	0	0	0	1	000	SWP, SWPA, SWPAL, SWPL - 32-bit SWP variant	ARMv8.1-LSE
10	0	0	1	0	000	LDADD, LDADDA, LDADDAL, LDADDL - 32-bit LDADDL variant	ARMv8.1-LSE
10	0	0	1	0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL - 32-bit LDCLRL variant	ARMv8.1-LSE
10	0	0	1	0	010	LDEOR, LDEORA, LDEORAL, LDEORL - 32-bit LDEORL variant	ARMv8.1-LSE
10	0	0	1	0	011	LDSET, LDSETA, LDSETAL, LDSETL - 32-bit LDSETL variant	ARMv8.1-LSE
10	0	0	1	0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL - 32-bit LDSMAXL variant	ARMv8.1-LSE
10	0	0	1	0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL - 32-bit LDSMINL variant	ARMv8.1-LSE
10	0	0	1	0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL - 32-bit LDUMAXL variant	ARMv8.1-LSE
10	0	0	1	0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL - 32-bit LDUMINL variant	ARMv8.1-LSE
10	0	0	1	1	000	SWP, SWPA, SWPAL, SWPL - 32-bit SWPL variant	ARMv8.1-LSE
10	0	1	0	0	000	LDADD, LDADDA, LDADDAL, LDADDL - 32-bit LDADDA variant	ARMv8.1-LSE
10	0	1	0	0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL - 32-bit LDCLRA variant	ARMv8.1-LSE
10	0	1	0	0	010	LDEOR, LDEORA, LDEORAL, LDEORL - 32-bit LDEORA variant	ARMv8.1-LSE
10	0	1	0	0	011	LDSET, LDSETA, LDSETAL, LDSETL - 32-bit LDSETA variant	ARMv8.1-LSE
10	0	1	0	0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL - 32-bit LDSMAXA variant	ARMv8.1-LSE
10	0	1	0	0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL - 32-bit LDSMINA variant	ARMv8.1-LSE
10	0	1	0	0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL - 32-bit LDUMAXA variant	ARMv8.1-LSE

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields						Instruction page	Architecture version
size	V	A	R	o3	opc		
10	0	1	0	0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL - 32-bit LDUMINA variant	ARMv8.1-LSE
10	0	1	0	1	000	SWP, SWPA, SWPAL, SWPL - 32-bit SWPA variant	ARMv8.1-LSE
10	0	1	0	1	100	LDAPR - 32-bit variant	ARMv8.3-RCPC
10	0	1	1	0	000	LDADD, LDADDA, LDADDAL, LDADDL - 32-bit LDADDAL variant	ARMv8.1-LSE
10	0	1	1	0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL - 32-bit LDCLRAL variant	ARMv8.1-LSE
10	0	1	1	0	010	LDEOR, LDEORA, LDEORAL, LDEORL - 32-bit LDEORAL variant	ARMv8.1-LSE
10	0	1	1	0	011	LDSET, LDSETA, LDSETAL, LDSETL - 32-bit LDSETAL variant	ARMv8.1-LSE
10	0	1	1	0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL - 32-bit LDSMAXAL variant	ARMv8.1-LSE
10	0	1	1	0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL - 32-bit LDSMINAL variant	ARMv8.1-LSE
10	0	1	1	0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL - 32-bit LDUMAXAL variant	ARMv8.1-LSE
10	0	1	1	0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL - 32-bit LDUMINAL variant	ARMv8.1-LSE
10	0	1	1	1	000	SWP, SWPA, SWPAL, SWPL - 32-bit SWPAL variant	ARMv8.1-LSE
11	0	0	0	0	000	LDADD, LDADDA, LDADDAL, LDADDL - 64-bit LDADD variant	ARMv8.1-LSE
11	0	0	0	0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL - 64-bit LDCLR variant	ARMv8.1-LSE
11	0	0	0	0	010	LDEOR, LDEORA, LDEORAL, LDEORL - 64-bit LDEOR variant	ARMv8.1-LSE
11	0	0	0	0	011	LDSET, LDSETA, LDSETAL, LDSETL - 64-bit LDSET variant	ARMv8.1-LSE
11	0	0	0	0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL - 64-bit LDSMAX variant	ARMv8.1-LSE
11	0	0	0	0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL - 64-bit LDSMIN variant	ARMv8.1-LSE
11	0	0	0	0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL - 64-bit LDUMAX variant	ARMv8.1-LSE
11	0	0	0	0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL - 64-bit LDUMIN variant	ARMv8.1-LSE
11	0	0	0	1	000	SWP, SWPA, SWPAL, SWPL - 64-bit SWP variant	ARMv8.1-LSE
11	0	0	1	0	000	LDADD, LDADDA, LDADDAL, LDADDL - 64-bit LDADDL variant	ARMv8.1-LSE
11	0	0	1	0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL - 64-bit LDCLRL variant	ARMv8.1-LSE

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields						Instruction page	Architecture version
size	V	A	R	o3	opc		
11	0	0	1	0	010	LDEOR, LDEORA, LDEORAL, LDEORL - 64-bit LDEORL variant	ARMv8.1-LSE
11	0	0	1	0	011	LDSET, LDSETA, LDSETAL, LDSETL - 64-bit LDSETL variant	ARMv8.1-LSE
11	0	0	1	0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL - 64-bit LDSMAXL variant	ARMv8.1-LSE
11	0	0	1	0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL - 64-bit LDSMINL variant	ARMv8.1-LSE
11	0	0	1	0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL - 64-bit LDUMAXL variant	ARMv8.1-LSE
11	0	0	1	0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL - 64-bit LDUMINL variant	ARMv8.1-LSE
11	0	0	1	1	000	SWP, SWPA, SWPAL, SWPL - 64-bit SWPL variant	ARMv8.1-LSE
11	0	1	0	0	000	LDADD, LDADDA, LDADDAL, LDADDL - 64-bit LDADDA variant	ARMv8.1-LSE
11	0	1	0	0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL - 64-bit LDCLRA variant	ARMv8.1-LSE
11	0	1	0	0	010	LDEOR, LDEORA, LDEORAL, LDEORL - 64-bit LDEORA variant	ARMv8.1-LSE
11	0	1	0	0	011	LDSET, LDSETA, LDSETAL, LDSETL - 64-bit LDSETA variant	ARMv8.1-LSE
11	0	1	0	0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL - 64-bit LDSMAXA variant	ARMv8.1-LSE
11	0	1	0	0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL - 64-bit LDSMINA variant	ARMv8.1-LSE
11	0	1	0	0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL - 64-bit LDUMAXA variant	ARMv8.1-LSE
11	0	1	0	0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL - 64-bit LDUMINA variant	ARMv8.1-LSE
11	0	1	0	1	000	SWP, SWPA, SWPAL, SWPL - 64-bit SWPA variant	ARMv8.1-LSE
11	0	1	0	1	100	LDAPR - 64-bit variant	ARMv8.3-RCPC
11	0	1	1	0	000	LDADD, LDADDA, LDADDAL, LDADDL - 64-bit LDADDAL variant	ARMv8.1-LSE
11	0	1	1	0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL - 64-bit LDCLRAL variant	ARMv8.1-LSE
11	0	1	1	0	010	LDEOR, LDEORA, LDEORAL, LDEORL - 64-bit LDEORAL variant	ARMv8.1-LSE
11	0	1	1	0	011	LDSET, LDSETA, LDSETAL, LDSETL - 64-bit LDSETAL variant	ARMv8.1-LSE
11	0	1	1	0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL - 64-bit LDSMAXAL variant	ARMv8.1-LSE

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields						Instruction page	Architecture version
size	V	A	R	o3	opc		
11	0	1	1	0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL - 64-bit LDSMINAL variant	ARMv8.1-LSE
11	0	1	1	0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL - 64-bit LDUMAXAL variant	ARMv8.1-LSE
11	0	1	1	0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL - 64-bit LDUMINAL variant	ARMv8.1-LSE
11	0	1	1	1	000	SWP, SWPA, SWPAL, SWPL - 64-bit SWPAL variant	ARMv8.1-LSE

Load/store register (register offset)

This section describes the encoding of the Load/store register (register offset) instruction class. The encodings in this section are decoded from *Loads and Stores* on page C4-281.

31 30	29 28	27	26	25 24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
size	1 1	1	V	0 0	opc	1	Rm	option	S	1 0	Rn	Rt

Decode fields				Instruction page
size	V	opc	option	
-	-	-	x0x	Unallocated.
x1	1	1x	-	Unallocated.
00	0	00	!= 011	STRB (register) - Extended register variant
00	0	00	011	STRB (register) - Shifted register variant
00	0	01	!= 011	LDRB (register) - Extended register variant
00	0	01	011	LDRB (register) - Shifted register variant
00	0	10	!= 011	LDRSB (register) - 64-bit with extended register offset variant
00	0	10	011	LDRSB (register) - 64-bit with shifted register offset variant
00	0	11	!= 011	LDRSB (register) - 32-bit with extended register offset variant
00	0	11	011	LDRSB (register) - 32-bit with shifted register offset variant
00	1	00	!= 011	STR (register, SIMD&FP)
00	1	00	011	STR (register, SIMD&FP)
00	1	01	!= 011	LDR (register, SIMD&FP)
00	1	01	011	LDR (register, SIMD&FP)
00	1	10	-	STR (register, SIMD&FP)
00	1	11	-	LDR (register, SIMD&FP)
01	0	00	-	STRH (register)

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields				Instruction page
size	V	opc	option	
01	0	01	-	LDRH (register)
01	0	10	-	LDRSH (register) - 64-bit variant
01	0	11	-	LDRSH (register) - 32-bit variant
01	1	00	-	STR (register, SIMD&FP)
01	1	01	-	LDR (register, SIMD&FP)
1x	0	11	-	Unallocated.
1x	1	1x	-	Unallocated.
10	0	00	-	STR (register) - 32-bit variant
10	0	01	-	LDR (register) - 32-bit variant
10	0	10	-	LDRSW (register)
10	1	00	-	STR (register, SIMD&FP)
10	1	01	-	LDR (register, SIMD&FP)
11	0	00	-	STR (register) - 64-bit variant
11	0	01	-	LDR (register) - 64-bit variant
11	0	10	-	PRFM (register)
11	1	00	-	STR (register, SIMD&FP)
11	1	01	-	LDR (register, SIMD&FP)

Load/store register (pac)

This section describes the encoding of the Load/store register (pac) instruction class. The encodings in this section are decoded from *Loads and Stores* on page C4-281.

31 30 29 28	27 26 25 24	23 22 21 20	12	11 10 9	5 4	0
size	1 1 1 V 0 0	M S 1	imm9	W 1	Rn	Rt

Decode fields				Instruction page	Architecture version
size	V	M	W		
!= 11	-	-	-	Unallocated.	-
11	0	0	0	LDRAA, LDRAB - Key A, offset variant	ARMv8.3-PAuth
11	0	0	1	LDRAA, LDRAB - Key A, pre-indexed variant	ARMv8.3-PAuth
11	0	1	0	LDRAA, LDRAB - Key B, offset variant	ARMv8.3-PAuth
11	0	1	1	LDRAA, LDRAB - Key B, pre-indexed variant	ARMv8.3-PAuth
11	1	-	-	Unallocated.	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Load/store register (unsigned immediate)

This section describes the encoding of the Load/store register (unsigned immediate) instruction class. The encodings in this section are decoded from *Loads and Stores* on page C4-281.

31 30	29 28	27	26	25 24	23 22	21 10	9 5	4 0
size	1 1	1	V	0 1	opc	imm12	Rn	Rt

Decode fields			Instruction page
size	V	opc	
x1	1	1x	Unallocated.
00	0	00	STRB (immediate)
00	0	01	LDRB (immediate)
00	0	10	LDRSB (immediate) - 64-bit variant
00	0	11	LDRSB (immediate) - 32-bit variant
00	1	00	STR (immediate, SIMD&FP) - 8-bit variant
00	1	01	LDR (immediate, SIMD&FP) - 8-bit variant
00	1	10	STR (immediate, SIMD&FP) - 128-bit variant
00	1	11	LDR (immediate, SIMD&FP) - 128-bit variant
01	0	00	STRH (immediate)
01	0	01	LDRH (immediate)
01	0	10	LDRSH (immediate) - 64-bit variant
01	0	11	LDRSH (immediate) - 32-bit variant
01	1	00	STR (immediate, SIMD&FP) - 16-bit variant
01	1	01	LDR (immediate, SIMD&FP) - 16-bit variant
1x	0	11	Unallocated.
1x	1	1x	Unallocated.
10	0	00	STR (immediate) - 32-bit variant
10	0	01	LDR (immediate) - 32-bit variant
10	0	10	LDRSW (immediate)
10	1	00	STR (immediate, SIMD&FP) - 32-bit variant
10	1	01	LDR (immediate, SIMD&FP) - 32-bit variant
11	0	00	STR (immediate) - 64-bit variant
11	0	01	LDR (immediate) - 64-bit variant

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields			Instruction page
size	V	opc	
11	0	10	PRFM (immediate)
11	1	00	STR (immediate, SIMD&FP) - 64-bit variant
11	1	01	LDR (immediate, SIMD&FP) - 64-bit variant

C4.1.5 Data Processing -- Register

This section describes the encoding of the Data Processing -- Register group. The encodings in this section are decoded from *A64 instruction set encoding* on page C4-268.

Table C4-6 Encoding table for the Data Processing -- Register group

Decode fields				Decode group or instruction page
op0	op1	op2	op3	
0	1	0110	-	*Data-processing (2 source)* on page C4-314
1	1	0110	-	*Data-processing (1 source)* on page C4-315
-	0	0xxx	-	*Logical (shifted register)* on page C4-317
-	0	1xx0	-	*Add/subtract (shifted register)* on page C4-318
-	0	1xx1	-	*Add/subtract (extended register)* on page C4-318
-	1	0000	000000	*Add/subtract (with carry)* on page C4-319
-	1	0000	x00001	*Rotate right into flags* on page C4-320
-	1	0000	xx0010	*Evaluate into flags* on page C4-320
-	1	0010	xxxx0x	*Conditional compare (register)* on page C4-321
-	1	0010	xxxx1x	*Conditional compare (immediate)* on page C4-321
-	1	0100	-	*Conditional select* on page C4-322
-	1	1xxx	-	*Data-processing (3 source)* on page C4-322

Data-processing (2 source)

This section describes the encoding of the Data-processing (2 source) instruction class. The encodings in this section are decoded from *Data Processing -- Register* on page C4-313.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31	30	29	28	27	26	25	24	23	22	21	20	16	15	10	9	5	4	0
sf	0	S	1	1	0	1	0	1	1	0	Rm		opcode		Rn		Rd	

Decode fields			Instruction page	Architecture version
sf	S	opcode		
-	-	000001	Unallocated.	-
-	-	011xxx	Unallocated.	-
-	-	1xxxxx	Unallocated.	-
-	0	00011x	Unallocated.	-
-	0	001101	Unallocated.	-
-	0	00111x	Unallocated.	-
-	1	00001x	Unallocated.	-
-	1	0001xx	Unallocated.	-
-	1	001xxx	Unallocated.	-
-	1	01xxxx	Unallocated.	-
0	-	000000	Unallocated.	-
0	0	000010	UDIV - 32-bit variant	-
0	0	000011	SDIV - 32-bit variant	-
0	0	00010x	Unallocated.	-
0	0	001000	LSLV - 32-bit variant	-
0	0	001001	LSRV - 32-bit variant	-
0	0	001010	ASRV - 32-bit variant	-
0	0	001011	RORV - 32-bit variant	-
0	0	001100	Unallocated.	-
0	0	010x11	Unallocated.	-
0	0	010000	CRC32B, CRC32H, CRC32W, CRC32X - CRC32B variant	-
0	0	010001	CRC32B, CRC32H, CRC32W, CRC32X - CRC32H variant	-
0	0	010010	CRC32B, CRC32H, CRC32W, CRC32X - CRC32W variant	-
0	0	010100	CRC32CB, CRC32CH, CRC32CW, CRC32CX - CRC32CB variant	-
0	0	010101	CRC32CB, CRC32CH, CRC32CW, CRC32CX - CRC32CH variant	-
0	0	010110	CRC32CB, CRC32CH, CRC32CW, CRC32CX - CRC32CW variant	-
1	0	000000	SUBP	ARMv8.5-MemTag
1	0	000010	UDIV - 64-bit variant	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields			Instruction page	Architecture version
sf	S	opcode		
1	0	000011	SDIV - 64-bit variant	-
1	0	000100	IRG	ARMv8.5-MemTag
1	0	000101	GMI	ARMv8.5-MemTag
1	0	001000	LSLV - 64-bit variant	-
1	0	001001	LSRV - 64-bit variant	-
1	0	001010	ASRV - 64-bit variant	-
1	0	001011	RORV - 64-bit variant	-
1	0	001100	PACGA	ARMv8.3-PAuth
1	0	010xx0	Unallocated.	-
1	0	010x0x	Unallocated.	-
1	0	010011	CRC32B, CRC32H, CRC32W, CRC32X - CRC32X variant	-
1	0	010111	CRC32CB, CRC32CH, CRC32CW, CRC32CX - CRC32CX variant	-
1	1	000000	SUBPS	ARMv8.5-MemTag

Data-processing (1 source)

This section describes the encoding of the Data-processing (1 source) instruction class. The encodings in this section are decoded from *Data Processing -- Register* on page C4-313.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	10	9	5	4	0
sf	1	S	1	1	0	1	0	1	1	0	opcode2		opcode		Rn		Rd	

Decode fields					Instruction page	Architecture version
sf	S	opcode2	opcode	Rn		
-	-	-	1xxxxx	-	Unallocated.	-
-	-	xxx1x	-	-	Unallocated.	-
-	-	xx1xx	-	-	Unallocated.	-
-	-	x1xxx	-	-	Unallocated.	-
-	-	1xxxx	-	-	Unallocated.	-
-	0	00000	00011x	-	Unallocated.	-
-	0	00000	001xxx	-	Unallocated.	-
-	0	00000	01xxxx	-	Unallocated.	-
-	1	-	-	-	Unallocated.	-
0	-	00001	-	-	Unallocated.	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields					Instruction page	Architecture version
sf	S	opcode2	opcode	Rn		
0	0	00000	000000	-	RBIT - 32-bit variant	-
0	0	00000	000001	-	REV16 - 32-bit variant	-
0	0	00000	000010	-	REV - 32-bit variant	-
0	0	00000	000011	-	Unallocated.	-
0	0	00000	000100	-	CLZ - 32-bit variant	-
0	0	00000	000101	-	CLS - 32-bit variant	-
1	0	00000	000000	-	RBIT - 64-bit variant	-
1	0	00000	000001	-	REV16 - 64-bit variant	-
1	0	00000	000010	-	REV32	-
1	0	00000	000011	-	REV - 64-bit variant	-
1	0	00000	000100	-	CLZ - 64-bit variant	-
1	0	00000	000101	-	CLS - 64-bit variant	-
1	0	00001	000000	-	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA - PACIA variant	ARMv8.3-PAuth
1	0	00001	000001	-	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB - PACIB variant	ARMv8.3-PAuth
1	0	00001	000010	-	PACDA, PACDZA - PACDA variant	ARMv8.3-PAuth
1	0	00001	000011	-	PACDB, PACDZB - PACDB variant	ARMv8.3-PAuth
1	0	00001	000100	-	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA - AUTIA variant	ARMv8.3-PAuth
1	0	00001	000101	-	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB - AUTIB variant	ARMv8.3-PAuth
1	0	00001	000110	-	AUTDA, AUTDZA - AUTDA variant	ARMv8.3-PAuth
1	0	00001	000111	-	AUTDB, AUTDZB - AUTDB variant	ARMv8.3-PAuth
1	0	00001	001000	11111	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA - PACIZA variant	ARMv8.3-PAuth
1	0	00001	001001	11111	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB - PACIZB variant	ARMv8.3-PAuth
1	0	00001	001010	11111	PACDA, PACDZA - PACDZA variant	ARMv8.3-PAuth
1	0	00001	001011	11111	PACDB, PACDZB - PACDZB variant	ARMv8.3-PAuth
1	0	00001	001100	11111	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA - AUTIZA variant	ARMv8.3-PAuth
1	0	00001	001101	11111	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB - AUTIZB variant	ARMv8.3-PAuth
1	0	00001	001110	11111	AUTDA, AUTDZA - AUTDZA variant	ARMv8.3-PAuth

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields					Instruction page	Architecture version
sf	S	opcode2	opcode	Rn		
1	0	00001	001111	11111	AUTDB, AUTDZB - AUTDZB variant	ARMv8.3-PAuth
1	0	00001	010000	11111	XPACD, XPACI, XPACLRI - XPACI variant	ARMv8.3-PAuth
1	0	00001	010001	11111	XPACD, XPACI, XPACLRI - XPACD variant	ARMv8.3-PAuth
1	0	00001	01001x	-	Unallocated.	-
1	0	00001	0101xx	-	Unallocated.	-
1	0	00001	011xxx	-	Unallocated.	-

Logical (shifted register)

This section describes the encoding of the Logical (shifted register) instruction class. The encodings in this section are decoded from *Data Processing -- Register* on page C4-313.

31 30 29 28	27 26 25 24	23 22	21	20 16	15 10	9 5	4 0
sf opc	0 1 0 1 0	shift	N	Rm	imm6	Rn	Rd

Decode fields				Instruction page
sf	opc	N	imm6	
0	-	-	1xxxxx	Unallocated.
0	00	0	-	AND (shifted register) - 32-bit variant
0	00	1	-	BIC (shifted register) - 32-bit variant
0	01	0	-	ORR (shifted register) - 32-bit variant
0	01	1	-	ORN (shifted register) - 32-bit variant
0	10	0	-	EOR (shifted register) - 32-bit variant
0	10	1	-	EON (shifted register) - 32-bit variant
0	11	0	-	ANDS (shifted register) - 32-bit variant
0	11	1	-	BICS (shifted register) - 32-bit variant
1	00	0	-	AND (shifted register) - 64-bit variant
1	00	1	-	BIC (shifted register) - 64-bit variant
1	01	0	-	ORR (shifted register) - 64-bit variant
1	01	1	-	ORN (shifted register) - 64-bit variant
1	10	0	-	EOR (shifted register) - 64-bit variant
1	10	1	-	EON (shifted register) - 64-bit variant
1	11	0	-	ANDS (shifted register) - 64-bit variant
1	11	1	-	BICS (shifted register) - 64-bit variant

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Add/subtract (shifted register)

This section describes the encoding of the Add/subtract (shifted register) instruction class. The encodings in this section are decoded from *Data Processing -- Register* on page C4-313.

31	30	29	28	27	26	25	24	23 22	21	20 16	15 10	9 5	4 0
sf	op	S	0	1	0	1	1	shift	0	Rm	imm6	Rn	Rd

Decode fields					Instruction page
sf	op	S	shift	imm6	
-	-	-	11	-	Unallocated.
0	-	-	-	1xxxxx	Unallocated.
0	0	0	-	-	ADD (shifted register) - 32-bit variant
0	0	1	-	-	ADDS (shifted register) - 32-bit variant
0	1	0	-	-	SUB (shifted register) - 32-bit variant
0	1	1	-	-	SUBS (shifted register) - 32-bit variant
1	0	0	-	-	ADD (shifted register) - 64-bit variant
1	0	1	-	-	ADDS (shifted register) - 64-bit variant
1	1	0	-	-	SUB (shifted register) - 64-bit variant
1	1	1	-	-	SUBS (shifted register) - 64-bit variant

Add/subtract (extended register)

This section describes the encoding of the Add/subtract (extended register) instruction class. The encodings in this section are decoded from *Data Processing -- Register* on page C4-313.

31	30	29	28	27	26	25	24	23 22	21	20 16	15 13	12 10	9 5	4 0
sf	op	S	0	1	0	1	1	opt	1	Rm	option	imm3	Rn	Rd

Decode fields					Instruction page
sf	op	S	opt	imm3	
-	-	-	-	1x1	Unallocated.
-	-	-	-	11x	Unallocated.
-	-	-	x1	-	Unallocated.
-	-	-	1x	-	Unallocated.
0	0	0	00	-	ADD (extended register) - 32-bit variant
0	0	1	00	-	ADDS (extended register) - 32-bit variant
0	1	0	00	-	SUB (extended register) - 32-bit variant

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields					Instruction page
sf	op	S	opt	imm3	
0	1	1	00	-	SUBS (extended register) - 32-bit variant
1	0	0	00	-	ADD (extended register) - 64-bit variant
1	0	1	00	-	ADDS (extended register) - 64-bit variant
1	1	0	00	-	SUB (extended register) - 64-bit variant
1	1	1	00	-	SUBS (extended register) - 64-bit variant

Add/subtract (with carry)

This section describes the encoding of the Add/subtract (with carry) instruction class. The encodings in this section are decoded from *Data Processing -- Register* on page C4-313.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
sf	op	S	1	1	0	1	0	0	0	0	Rm		0	0	0	0	0	0	Rn		Rd	

Decode fields			Instruction page
sf	op	S	
0	0	0	ADC - 32-bit variant
0	0	1	ADCS - 32-bit variant
0	1	0	SBC - 32-bit variant
0	1	1	SBCS - 32-bit variant
1	0	0	ADC - 64-bit variant
1	0	1	ADCS - 64-bit variant
1	1	0	SBC - 64-bit variant
1	1	1	SBCS - 64-bit variant

Rotate right into flags

This section describes the encoding of the Rotate right into flags instruction class. The encodings in this section are decoded from *Data Processing -- Register* on page C4-313.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31	30	29	28	27	26	25	24	23	22	21	20	15	14	13	12	11	10	9	5	4	3	0
sf	op	S	1	1	0	1	0	0	0	0	imm6	0	0	0	0	1	Rn			o2	mask	

Decode fields				Instruction page	Architecture version
sf	op	S	o2		
0	-	-	-	Unallocated.	-
1	0	0	-	Unallocated.	-
1	0	1	0	RMIF	ARMv8.4-CondM
1	0	1	1	Unallocated.	-
1	1	-	-	Unallocated.	-

Evaluate into flags

This section describes the encoding of the Evaluate into flags instruction class. The encodings in this section are decoded from *Data Processing -- Register* on page C4-313.

31	30	29	28	27	26	25	24	23	22	21	20	15	14	13	12	11	10	9	5	4	3	0
sf	op	S	1	1	0	1	0	0	0	0	opcode2	sz	0	0	1	0	Rn			o3	mask	

Decode fields							Instruction page	Architecture version
sf	op	S	opcode2	sz	o3	mask		
0	0	0	-	-	-	-	Unallocated.	-
0	0	1	!= 000000	-	-	-	Unallocated.	-
0	0	1	000000	-	0	!= 1101	Unallocated.	-
0	0	1	000000	-	1	-	Unallocated.	-
0	0	1	000000	0	0	1101	SETF8, SETF16 - SETF8 variant	ARMv8.4-CondM
0	0	1	000000	1	0	1101	SETF8, SETF16 - SETF16 variant	ARMv8.4-CondM
0	1	-	-	-	-	-	Unallocated.	-
1	-	-	-	-	-	-	Unallocated.	-

Conditional compare (register)

This section describes the encoding of the Conditional compare (register) instruction class. The encodings in this section are decoded from *Data Processing -- Register* on page C4-313.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31	30	29	28	27	26	25	24	23	22	21	20	16	15	12	11	10	9	5	4	3	0
sf	op	S	1	1	0	1	0	0	1	0	Rm		cond		0	o2	Rn		o3	nzcv	

Decode fields					Instruction page
sf	op	S	o2	o3	
-	-	-	-	1	Unallocated.
-	-	-	1	-	Unallocated.
-	-	0	-	-	Unallocated.
0	0	1	0	0	CCMN (register) - 32-bit variant
0	1	1	0	0	CCMP (register) - 32-bit variant
1	0	1	0	0	CCMN (register) - 64-bit variant
1	1	1	0	0	CCMP (register) - 64-bit variant

Conditional compare (immediate)

This section describes the encoding of the Conditional compare (immediate) instruction class. The encodings in this section are decoded from *Data Processing -- Register* on page C4-313.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	12	11	10	9	5	4	3	0
sf	op	S	1	1	0	1	0	0	1	0	imm5		cond		1	o2	Rn		o3	nzcv	

Decode fields					Instruction page
sf	op	S	o2	o3	
-	-	-	-	1	Unallocated.
-	-	-	1	-	Unallocated.
-	-	0	-	-	Unallocated.
0	0	1	0	0	CCMN (immediate) - 32-bit variant
0	1	1	0	0	CCMP (immediate) - 32-bit variant
1	0	1	0	0	CCMN (immediate) - 64-bit variant
1	1	1	0	0	CCMP (immediate) - 64-bit variant

Conditional select

This section describes the encoding of the Conditional select instruction class. The encodings in this section are decoded from *Data Processing -- Register* on page C4-313.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31	30	29	28	27	26	25	24	23	22	21	20	16	15	12	11	10	9	5	4	0
sf	op	S	1	1	0	1	0	1	0	0	Rm		cond		op2		Rn		Rd	

Decode fields				Instruction page
sf	op	S	op2	
-	-	-	1x	Unallocated.
-	-	1	-	Unallocated.
0	0	0	00	CSEL - 32-bit variant
0	0	0	01	CSINC - 32-bit variant
0	1	0	00	CSINV - 32-bit variant
0	1	0	01	CSNEG - 32-bit variant
1	0	0	00	CSEL - 64-bit variant
1	0	0	01	CSINC - 64-bit variant
1	1	0	00	CSINV - 64-bit variant
1	1	0	01	CSNEG - 64-bit variant

Data-processing (3 source)

This section describes the encoding of the Data-processing (3 source) instruction class. The encodings in this section are decoded from *Data Processing -- Register* on page C4-313.

31	30	29	28	27	26	25	24	23	21	20	16	15	14	10	9	5	4	0
sf	op54		1	1	0	1	1	op31		Rm		o0	Ra		Rn		Rd	

Decode fields				Instruction page
sf	op54	op31	o0	
-	00	010	1	Unallocated.
-	00	011	-	Unallocated.
-	00	100	-	Unallocated.
-	00	110	1	Unallocated.
-	00	111	-	Unallocated.
-	01	-	-	Unallocated.
-	1x	-	-	Unallocated.
0	00	000	0	MADD - 32-bit variant
0	00	000	1	MSUB - 32-bit variant

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields				Instruction page
sf	op54	op31	o0	
0	00	001	0	Unallocated.
0	00	001	1	Unallocated.
0	00	010	0	Unallocated.
0	00	101	0	Unallocated.
0	00	101	1	Unallocated.
0	00	110	0	Unallocated.
1	00	000	0	MADD - 64-bit variant
1	00	000	1	MSUB - 64-bit variant
1	00	001	0	SMADDL
1	00	001	1	SMSUBL
1	00	010	0	SMULH
1	00	101	0	UMADDL
1	00	101	1	UMSUBL
1	00	110	0	UMULH

C4.1.6 Data Processing -- Scalar Floating-Point and Advanced SIMD

This section describes the encoding of the Data Processing -- Scalar Floating-Point and Advanced SIMD group. The encodings in this section are decoded from *A64 instruction set encoding* on page C4-268.

31 – 28	27 – 25	24	23 – 22	19 – 18	10 – 9	0
op0	111	op1	op2	op3		

Table C4-7 Encoding table for the Data Processing -- Scalar Floating-Point and Advanced SIMD group

Decode fields				Decode group or instruction page	Architecture version
op0	op1	op2	op3		
0000	0x	x101	00xxxxx10	Unallocated.	-
0010	0x	x101	00xxxxx10	Unallocated.	-
0100	0x	x101	00xxxxx10	*Cryptographic AES* on page C4-326	-
0101	0x	x0xx	xxx0xxx00	*Cryptographic three-register SHA* on page C4-326	-
0101	0x	x0xx	xxx0xxx10	Unallocated.	-
0101	0x	x101	00xxxxx10	*Cryptographic two-register SHA* on page C4-327	-
0110	0x	x101	00xxxxx10	Unallocated.	-
0111	0x	x0xx	xxx0xxxx0	Unallocated.	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C4-7 Encoding table for the Data Processing -- Scalar Floating-Point and Advanced SIMD group (continued)

Decode fields				Decode group or instruction page	Architecture version
op0	op1	op2	op3		
0111	0x	x101	00xxxxx10	Unallocated.	-
01x1	00	00xx	xxx0xxxx1	*Advanced SIMD scalar copy* on page C4-328	-
01x1	01	00xx	xxx0xxxx1	Unallocated.	-
01x1	0x	0111	00xxxxx10	Unallocated.	-
01x1	0x	10xx	xxx00xxx1	*Advanced SIMD scalar three same FP16* on page C4-328	-
01x1	0x	10xx	xxx01xxx1	Unallocated.	-
01x1	0x	1111	00xxxxx10	*Advanced SIMD scalar two-register miscellaneous FP16* on page C4-329	-
01x1	0x	x0xx	xxx1xxxx0	Unallocated.	-
01x1	0x	x0xx	xxx1xxxx1	*Advanced SIMD scalar three same extra* on page C4-330	-
01x1	0x	x100	00xxxxx10	*Advanced SIMD scalar two-register miscellaneous* on page C4-331	-
01x1	0x	x110	00xxxxx10	*Advanced SIMD scalar pairwise* on page C4-333	-
01x1	0x	x1xx	1xxxxxx10	Unallocated.	-
01x1	0x	x1xx	x1xxxxx10	Unallocated.	-
01x1	0x	x1xx	xxxxxxx00	*Advanced SIMD scalar three different* on page C4-334	-
01x1	0x	x1xx	xxxxxxxx1	*Advanced SIMD scalar three same* on page C4-334	-
01x1	10	-	xxxxxxxx1	*Advanced SIMD scalar shift by immediate* on page C4-336	-
01x1	11	-	xxxxxxxx1	Unallocated.	-
01x1	1x	-	xxxxxxxx0	*Advanced SIMD scalar x indexed element* on page C4-338	-
0x00	0x	x0xx	xxx0xxx00	*Advanced SIMD table lookup* on page C4-339	-
0x00	0x	x0xx	xxx0xxx10	*Advanced SIMD permute* on page C4-340	-
0x10	0x	x0xx	xxx0xxxx0	*Advanced SIMD extract* on page C4-341	-
0xx0	00	00xx	xxx0xxxx1	*Advanced SIMD copy* on page C4-341	-
0xx0	01	00xx	xxx0xxxx1	Unallocated.	-
0xx0	0x	0111	00xxxxx10	Unallocated.	-
0xx0	0x	10xx	xxx00xxx1	*Advanced SIMD three same (FP16)* on page C4-342	-
0xx0	0x	10xx	xxx01xxx1	Unallocated.	-
0xx0	0x	1111	00xxxxx10	*Advanced SIMD two-register miscellaneous (FP16)* on page C4-343	-
0xx0	0x	x0xx	xxx1xxxx0	Unallocated.	-
0xx0	0x	x0xx	xxx1xxxx1	*Advanced SIMD three-register extension* on page C4-344	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C4-7 Encoding table for the Data Processing -- Scalar Floating-Point and Advanced SIMD group (continued)

Decode fields				Decode group or instruction page	Architecture version
op0	op1	op2	op3		
0xx0	0x	x100	00xxxxx10	*Advanced SIMD two-register miscellaneous* on page C4-346	-
0xx0	0x	x110	00xxxxx10	*Advanced SIMD across lanes* on page C4-348	-
0xx0	0x	x1xx	1xxxxxx10	Unallocated.	-
0xx0	0x	x1xx	x1xxxxx10	Unallocated.	-
0xx0	0x	x1xx	xxxxxxx00	*Advanced SIMD three different* on page C4-350	-
0xx0	0x	x1xx	xxxxxxxx1	*Advanced SIMD three same* on page C4-351	-
0xx0	10	0000	xxxxxxxx1	*Advanced SIMD modified immediate* on page C4-354	-
0xx0	10	!= 0000	xxxxxxxx1	*Advanced SIMD shift by immediate* on page C4-355	-
0xx0	11	-	xxxxxxxx1	Unallocated.	-
0xx0	1x	-	xxxxxxxx0	*Advanced SIMD vector x indexed element* on page C4-357	-
1100	00	10xx	xxx10xxxx	*Cryptographic three-register, imm2* on page C4-359	-
1100	00	11xx	xxx1x00xx	*Cryptographic three-register SHA 512* on page C4-359	-
1100	00	-	xxx0xxxxx	*Cryptographic four-register* on page C4-360	-
1100	01	00xx	-	XAR	ARMv8.2-SHA3
1100	01	1000	0001000xx	*Cryptographic two-register SHA 512* on page C4-360	-
1xx0	1x	-	-	Unallocated.	-
x0x1	0x	x0xx	-	*Conversion between floating-point and fixed-point* on page C4-361	-
x0x1	0x	x1xx	xxx000000	*Conversion between floating-point and integer* on page C4-362	-
x0x1	0x	x1xx	xxxx10000	*Floating-point data-processing (1 source)* on page C4-366	-
x0x1	0x	x1xx	xxxxx1000	*Floating-point compare* on page C4-368	-
x0x1	0x	x1xx	xxxxxx100	*Floating-point immediate* on page C4-369	-
x0x1	0x	x1xx	xxxxxxx01	*Floating-point conditional compare* on page C4-370	-
x0x1	0x	x1xx	xxxxxxx10	*Floating-point data-processing (2 source)* on page C4-370	-
x0x1	0x	x1xx	xxxxxxx11	*Floating-point conditional select* on page C4-372	-
x0x1	1x	-	-	*Floating-point data-processing (3 source)* on page C4-372	-

Cryptographic AES

This section describes the encoding of the Cryptographic AES instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31 30 29 28	27 26 25 24	23 22	21 20	19 18 17 16	12	11 10	9 5	4 0
0 1 0 0	1 1 1 0	size	1 0	1 0 0	opcode	1 0	Rn	Rd

Decode fields		Instruction page
size	opcode	
-	x1xxx	Unallocated.
-	000xx	Unallocated.
-	1xxxx	Unallocated.
x1	-	Unallocated.
00	00100	AESE
00	00101	AESD
00	00110	AESMC
00	00111	AESIMC
1x	-	Unallocated.

Cryptographic three-register SHA

This section describes the encoding of the Cryptographic three-register SHA instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

31 30 29 28	27 26 25 24	23 22	21 20	16	15	14 12	11 10	9 5	4 0
0 1 0 1	1 1 1 0	size	0	Rm	0	opcode	0 0	Rn	Rd

Decode fields		Instruction page
size	opcode	
-	111	Unallocated.
x1	-	Unallocated.
00	000	SHA1C
00	001	SHA1P
00	010	SHA1M
00	011	SHA1SU0
00	100	SHA256H
00	101	SHA256H2
00	110	SHA256SU1
1x	-	Unallocated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Cryptographic two-register SHA

This section describes the encoding of the Cryptographic two-register SHA instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

31 30 29 28	27 26 25 24	23 22	21 20 19 18 17 16	12 11 10	9 5	4 0
0 1 0 1	1 1 1 0	size	1 0 1 0 0	opcode	1 0	Rn	Rd

Decode fields		Instruction page
size	opcode	
-	xx1xx	Unallocated.
-	x1xxx	Unallocated.
-	1xxxx	Unallocated.
x1	-	Unallocated.
00	00000	SHA1H
00	00001	SHA1SU1
00	00010	SHA256SU0
00	00011	Unallocated.
1x	-	Unallocated.

Advanced SIMD scalar copy

This section describes the encoding of the Advanced SIMD scalar copy instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

31 30	29	28 27 26 25 24	23 22 21 20	16	15	14 11	10	9 5	4 0
0 1	op	1 1 1 1	0 0 0 0	imm5	0	imm4	1	Rn	Rd

Decode fields			Instruction page
op	imm5	imm4	
0	-	xxx1	Unallocated.
0	-	xx1x	Unallocated.
0	-	x1xx	Unallocated.
0	-	0000	DUP (element)
0	-	1xxx	Unallocated.
0	x0000	0000	Unallocated.
1	-	-	Unallocated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Advanced SIMD scalar three same FP16

This section describes the encoding of the Advanced SIMD scalar three same FP16 instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

31 30	29	28 27 26 25	24	23	22 21	20 16	15 14	13 11	10	9 5	4 0
0 1	U	1 1 1 1	0	a	1 0	Rm	0 0	opcode	1	Rn	Rd

Decode fields			Instruction page	Architecture version
U	a	opcode		
-	-	110	Unallocated.	-
-	1	011	Unallocated.	-
0	0	011	FMULX	ARMv8.2-FP16
0	0	100	FCMEQ (register)	ARMv8.2-FP16
0	0	101	Unallocated.	-
0	0	111	FRECPS	ARMv8.2-FP16
0	1	100	Unallocated.	-
0	1	101	Unallocated.	-
0	1	111	FRSQRTS	ARMv8.2-FP16
1	0	011	Unallocated.	-
1	0	100	FCMGE (register)	ARMv8.2-FP16
1	0	101	FACGE	ARMv8.2-FP16
1	0	111	Unallocated.	-
1	1	010	FABD	ARMv8.2-FP16
1	1	100	FCMGT (register)	ARMv8.2-FP16
1	1	101	FACGT	ARMv8.2-FP16
1	1	111	Unallocated.	-

Advanced SIMD scalar two-register miscellaneous FP16

This section describes the encoding of the Advanced SIMD scalar two-register miscellaneous FP16 instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31 30	29	28 27 26 25	24	23 22 21 20	19 18 17 16	12	11 10	9 5	4 0
0 1	U	1 1 1 1	0	a	1 1 1 1	0 0	opcode	1 0	Rn	Rd

Decode fields			Instruction page	Architecture version
U	a	opcode		
-	-	00xxx	Unallocated.	-
-	-	010xx	Unallocated.	-
-	-	10xxx	Unallocated.	-
-	-	1100x	Unallocated.	-
-	-	11110	Unallocated.	-
-	0	011xx	Unallocated.	-
-	0	11111	Unallocated.	-
-	1	01111	Unallocated.	-
-	1	11100	Unallocated.	-
0	0	11010	FCVTNS (vector)	ARMv8.2-FP16
0	0	11011	FCVTMS (vector)	ARMv8.2-FP16
0	0	11100	FCVTAS (vector)	ARMv8.2-FP16
0	0	11101	SCVTF (vector, integer)	ARMv8.2-FP16
0	1	01100	FCMGT (zero)	ARMv8.2-FP16
0	1	01101	FCMEQ (zero)	ARMv8.2-FP16
0	1	01110	FCMLT (zero)	ARMv8.2-FP16
0	1	11010	FCVTPS (vector)	ARMv8.2-FP16
0	1	11011	FCVTZS (vector, integer)	ARMv8.2-FP16
0	1	11101	FRECPE	ARMv8.2-FP16
0	1	11111	FRECPX	ARMv8.2-FP16
1	0	11010	FCVTNU (vector)	ARMv8.2-FP16
1	0	11011	FCVTMU (vector)	ARMv8.2-FP16
1	0	11100	FCVTAU (vector)	ARMv8.2-FP16
1	0	11101	UCVTF (vector, integer)	ARMv8.2-FP16
1	1	01100	FCMGE (zero)	ARMv8.2-FP16
1	1	01101	FCMLE (zero)	ARMv8.2-FP16
1	1	01110	Unallocated.	-
1	1	11010	FCVTPU (vector)	ARMv8.2-FP16

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields			Instruction page	Architecture version
U	a	opcode		
1	1	11011	FCVTZU (vector, integer)	ARMv8.2-FP16
1	1	11101	FRSQRTE	ARMv8.2-FP16
1	1	11111	Unallocated.	-

Advanced SIMD scalar three same extra

This section describes the encoding of the Advanced SIMD scalar three same extra instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

31 30	29	28 27 26 25	24	23 22	21	20 16	15	14 11	10	9 5	4 0
0 1	U	1 1 1 1	0	size	0	Rm	1	opcode	1	Rn	Rd

Decode fields		Instruction page	Architecture version
U	opcode		
-	001x	Unallocated.	-
-	01xx	Unallocated.	-
-	1xxx	Unallocated.	-
0	0000	Unallocated.	-
0	0001	Unallocated.	-
1	0000	SQRDMLAH (vector)	ARMv8.1-RDMA
1	0001	SQRDMLSH (vector)	ARMv8.1-RDMA

Advanced SIMD scalar two-register miscellaneous

This section describes the encoding of the Advanced SIMD scalar two-register miscellaneous instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

31 30	29	28 27 26 25	24	23 22	21 20	19 18 17 16	12	11 10	9 5	4 0
0 1	U	1 1 1 1	0	size	1 0 0 0	0	opcode	1 0	Rn	Rd

Decode fields			Instruction page
U	size	opcode	
-	-	0000x	Unallocated.
-	-	00010	Unallocated.
-	-	0010x	Unallocated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields			Instruction page
U	size	opcode	
-	-	00110	Unallocated.
-	-	01111	Unallocated.
-	-	1000x	Unallocated.
-	-	10011	Unallocated.
-	-	10101	Unallocated.
-	-	10111	Unallocated.
-	-	1100x	Unallocated.
-	-	11110	Unallocated.
-	0x	011xx	Unallocated.
-	0x	11111	Unallocated.
-	1x	10110	Unallocated.
-	1x	11100	Unallocated.
0	-	00011	SUQADD
0	-	00111	SQABS
0	-	01000	CMGT (zero)
0	-	01001	CMEQ (zero)
0	-	01010	CMLT (zero)
0	-	01011	ABS
0	-	10010	Unallocated.
0	-	10100	SQXTN, SQXTN2
0	0x	10110	Unallocated.
0	0x	11010	FCVTNS (vector)
0	0x	11011	FCVTMS (vector)
0	0x	11100	FCVTAS (vector)
0	0x	11101	SCVTF (vector, integer)
0	1x	01100	FCMGT (zero)
0	1x	01101	FCMEQ (zero)
0	1x	01110	FCMLT (zero)
0	1x	11010	FCVTPS (vector)
0	1x	11011	FCVTZS (vector, integer)
0	1x	11101	FRECPE
0	1x	11111	FRECPX

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields			Instruction page
U	size	opcode	
1	-	00011	USQADD
1	-	00111	SQNEG
1	-	01000	CMGE (zero)
1	-	01001	CMLE (zero)
1	-	01010	Unallocated.
1	-	01011	NEG (vector)
1	-	10010	SQXTUN, SQXTUN2
1	-	10100	UQXTN, UQXTN2
1	0x	10110	FCVTXN, FCVTXN2
1	0x	11010	FCVTNU (vector)
1	0x	11011	FCVTMU (vector)
1	0x	11100	FCVTAU (vector)
1	0x	11101	UCVTF (vector, integer)
1	1x	01100	FCMGE (zero)
1	1x	01101	FCMLE (zero)
1	1x	01110	Unallocated.
1	1x	11010	FCVTPU (vector)
1	1x	11011	FCVTZU (vector, integer)
1	1x	11101	FRSQRTE
1	1x	11111	Unallocated.

Advanced SIMD scalar pairwise

This section describes the encoding of the Advanced SIMD scalar pairwise instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

31 30	29	28 27 26 25 24	23 22	21 20 19 18 17 16	12 11 10	9 5	4 0
0 1	U	1 1 1 1 0	size	1 1 0 0 0	opcode	1 0	Rn	Rd

Decode fields			Instruction page	Architecture version
U	size	opcode		
-	-	00xxx	Unallocated.	-
-	-	010xx	Unallocated.	-
-	-	01110	Unallocated.	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields			Instruction page	Architecture version
U	size	opcode		
-	-	10xxx	Unallocated.	-
-	-	1100x	Unallocated.	-
-	-	11010	Unallocated.	-
-	-	111xx	Unallocated.	-
-	1x	01101	Unallocated.	-
0	-	11011	ADDP (scalar)	-
0	00	01100	FMAXNMP (scalar) - Half-precision variant	ARMv8.2-FP16
0	00	01101	FADDP (scalar) - Half-precision variant	ARMv8.2-FP16
0	00	01111	FMAXP (scalar) - Half-precision variant	ARMv8.2-FP16
0	01	01100	Unallocated.	-
0	01	01101	Unallocated.	-
0	01	01111	Unallocated.	-
0	10	01100	FMINNMP (scalar) - Half-precision variant	ARMv8.2-FP16
0	10	01111	FMINP (scalar) - Half-precision variant	ARMv8.2-FP16
0	11	01100	Unallocated.	-
0	11	01111	Unallocated.	-
1	-	11011	Unallocated.	-
1	0x	01100	FMAXNMP (scalar) - Single-precision and double-precision variant	-
1	0x	01101	FADDP (scalar) - Single-precision and double-precision variant	-
1	0x	01111	FMAXP (scalar) - Single-precision and double-precision variant	-
1	1x	01100	FMINNMP (scalar) - Single-precision and double-precision variant	-
1	1x	01111	FMINP (scalar) - Single-precision and double-precision variant	-

Advanced SIMD scalar three different

This section describes the encoding of the Advanced SIMD scalar three different instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31 30	29	28 27 26 25	24	23 22	21	20 16	15 12	11 10	9 5	4 0
0 1	U	1 1 1 1	0	size	1	Rm	opcode	0 0	Rn	Rd

Decode fields		Instruction page
U	opcode	
-	00xx	Unallocated.
-	01xx	Unallocated.
-	1000	Unallocated.
-	1010	Unallocated.
-	1100	Unallocated.
-	111x	Unallocated.
0	1001	SQDMLAL, SQDMLAL2 (vector)
0	1011	SQDMLSL, SQDMLSL2 (vector)
0	1101	SQDMULL, SQDMULL2 (vector)
1	1001	Unallocated.
1	1011	Unallocated.
1	1101	Unallocated.

Advanced SIMD scalar three same

This section describes the encoding of the Advanced SIMD scalar three same instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

31 30	29	28 27 26 25	24	23 22	21	20 16	15	11 10	9 5	4 0
0 1	U	1 1 1 1	0	size	1	Rm	opcode	1	Rn	Rd

Decode fields			Instruction page
U	size	opcode	
-	-	00000	Unallocated.
-	-	0001x	Unallocated.
-	-	00100	Unallocated.
-	-	011xx	Unallocated.
-	-	1001x	Unallocated.
-	1x	11011	Unallocated.
0	-	00001	SQADD

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields			Instruction page
U	size	opcode	
0	-	00101	SQSUB
0	-	00110	CMGT (register)
0	-	00111	CMGE (register)
0	-	01000	SSHL
0	-	01001	SQSHL (register)
0	-	01010	SRSHL
0	-	01011	SQRSHL
0	-	10000	ADD (vector)
0	-	10001	CMTST
0	-	10100	Unallocated.
0	-	10101	Unallocated.
0	-	10110	SQDMULH (vector)
0	-	10111	Unallocated.
0	0x	11000	Unallocated.
0	0x	11001	Unallocated.
0	0x	11010	Unallocated.
0	0x	11011	FMULX
0	0x	11100	FCMEQ (register)
0	0x	11101	Unallocated.
0	0x	11110	Unallocated.
0	0x	11111	FRECPS
0	1x	11000	Unallocated.
0	1x	11001	Unallocated.
0	1x	11010	Unallocated.
0	1x	11100	Unallocated.
0	1x	11101	Unallocated.
0	1x	11110	Unallocated.
0	1x	11111	FRSQRTS
1	-	00001	UQADD
1	-	00101	UQSUB
1	-	00110	CMHI (register)
1	-	00111	CMHS (register)

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields			Instruction page
U	size	opcode	
1	-	01000	USHL
1	-	01001	UQSHL (register)
1	-	01010	URSHL
1	-	01011	UQRSHL
1	-	10000	SUB (vector)
1	-	10001	CMEQ (register)
1	-	10100	Unallocated.
1	-	10101	Unallocated.
1	-	10110	SQRDMULH (vector)
1	-	10111	Unallocated.
1	0x	11000	Unallocated.
1	0x	11001	Unallocated.
1	0x	11010	Unallocated.
1	0x	11011	Unallocated.
1	0x	11100	FCMGE (register)
1	0x	11101	FACGE
1	0x	11110	Unallocated.
1	0x	11111	Unallocated.
1	1x	11000	Unallocated.
1	1x	11001	Unallocated.
1	1x	11010	FABD
1	1x	11100	FCMGT (register)
1	1x	11101	FACGT
1	1x	11110	Unallocated.
1	1x	11111	Unallocated.

Advanced SIMD scalar shift by immediate

This section describes the encoding of the Advanced SIMD scalar shift by immediate instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31 30	29	28 27 26 25 24	23 22	19 18 16	15 11	10 9	5 4	0
0 1	U	1 1 1 1 1 0	immh	immb	opcode	1	Rn	Rd

Decode fields			Instruction page
U	immh	opcode	
-	!= 0000	00001	Unallocated.
-	!= 0000	00011	Unallocated.
-	!= 0000	00101	Unallocated.
-	!= 0000	00111	Unallocated.
-	!= 0000	01001	Unallocated.
-	!= 0000	01011	Unallocated.
-	!= 0000	01101	Unallocated.
-	!= 0000	01111	Unallocated.
-	!= 0000	101xx	Unallocated.
-	!= 0000	110xx	Unallocated.
-	!= 0000	11101	Unallocated.
-	!= 0000	11110	Unallocated.
-	0000	-	Unallocated.
0	!= 0000	00000	SSHR
0	!= 0000	00010	SSRA
0	!= 0000	00100	SRSHR
0	!= 0000	00110	SRSRA
0	!= 0000	01000	Unallocated.
0	!= 0000	01010	SHL
0	!= 0000	01100	Unallocated.
0	!= 0000	01110	SQSHL (immediate)
0	!= 0000	10000	Unallocated.
0	!= 0000	10001	Unallocated.
0	!= 0000	10010	SQSHRN, SQSHRN2
0	!= 0000	10011	SQRSHRN, SQRSHRN2
0	!= 0000	11100	SCVTF (vector, fixed-point)
0	!= 0000	11111	FCVTZS (vector, fixed-point)
1	!= 0000	00000	USHR

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields			Instruction page
U	immh	opcode	
1	!= 0000	00010	USRA
1	!= 0000	00100	URSHR
1	!= 0000	00110	URSRA
1	!= 0000	01000	SRI
1	!= 0000	01010	SLI
1	!= 0000	01100	SQSHLU
1	!= 0000	01110	UQSHL (immediate)
1	!= 0000	10000	SQSHRUN, SQSHRUN2
1	!= 0000	10001	SQRSHRUN, SQRSHRUN2
1	!= 0000	10010	UQSHRN, UQSHRN2
1	!= 0000	10011	UQRSHRN, UQRSHRN2
1	!= 0000	11100	UCVTF (vector, fixed-point)
1	!= 0000	11111	FCVTZU (vector, fixed-point)

Advanced SIMD scalar x indexed element

This section describes the encoding of the Advanced SIMD scalar x indexed element instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

31 30	29	28 27 26 25	24 23 22	21	20	19 16	15 12	11	10	9 5	4 0
0 1	U	1 1 1 1	size	L	M	Rm	opcode	H	0	Rn	Rd

Decode fields			Instruction page	Architecture version
U	size	opcode		
-	-	0000	Unallocated.	-
-	-	0010	Unallocated.	-
-	-	0100	Unallocated.	-
-	-	0110	Unallocated.	-
-	-	1000	Unallocated.	-
-	-	1010	Unallocated.	-
-	-	1110	Unallocated.	-
-	01	0001	Unallocated.	-
-	01	0101	Unallocated.	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields			Instruction page	Architecture version
U	size	opcode		
-	01	1001	Unallocated.	-
0	-	0011	SQDMLAL, SQDMLAL2 (by element)	-
0	-	0111	SQDMLSL, SQDMLSL2 (by element)	-
0	-	1011	SQDMULL, SQDMULL2 (by element)	-
0	-	1100	SQDMULH (by element)	-
0	-	1101	SQRDMULH (by element)	-
0	-	1111	Unallocated.	-
0	00	0001	FMLA (by element) - Scalar, half-precision variant	ARMv8.2-FP16
0	00	0101	FMLS (by element) - Scalar, half-precision variant	ARMv8.2-FP16
0	00	1001	FMUL (by element) - Scalar, half-precision variant	ARMv8.2-FP16
0	1x	0001	FMLA (by element) - Scalar, single-precision and double-precision variant	-
0	1x	0101	FMLS (by element) - Scalar, single-precision and double-precision variant	-
0	1x	1001	FMUL (by element) - Scalar, single-precision and double-precision variant	-
1	-	0011	Unallocated.	-
1	-	0111	Unallocated.	-
1	-	1011	Unallocated.	-
1	-	1100	Unallocated.	-
1	-	1101	SQRDMLAH (by element)	ARMv8.1-RDMA
1	-	1111	SQRDMLSH (by element)	ARMv8.1-RDMA
1	00	0001	Unallocated.	-
1	00	0101	Unallocated.	-
1	00	1001	FMULX (by element) - Scalar, half-precision variant	ARMv8.2-FP16
1	1x	0001	Unallocated.	-
1	1x	0101	Unallocated.	-
1	1x	1001	FMULX (by element) - Scalar, single-precision and double-precision variant	-

Advanced SIMD table lookup

This section describes the encoding of the Advanced SIMD table lookup instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31	30	29	28	27	26	25	24	23 22	21	20 16	15	14 13	12	11 10	9 5	4 0
0	Q	0	0	1	1	1	0	op2	0	Rm	0	len	op	0 0	Rn	Rd

Decode fields			Instruction page
op2	len	op	
x1	-	-	Unallocated.
00	00	0	TBL - Single register table variant
00	00	1	TBX - Single register table variant
00	01	0	TBL - Two register table variant
00	01	1	TBX - Two register table variant
00	10	0	TBL - Three register table variant
00	10	1	TBX - Three register table variant
00	11	0	TBL - Four register table variant
00	11	1	TBX - Four register table variant
1x	-	-	Unallocated.

Advanced SIMD permute

This section describes the encoding of the Advanced SIMD permute instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

31	30	29	28	27	26	25	24	23 22	21	20 16	15	14 12	11 10	9 5	4 0
0	Q	0	0	1	1	1	0	size	0	Rm	0	opcode	1 0	Rn	Rd

Decode fields	Instruction page
opcode	
000	Unallocated.
001	UZP1
010	TRN1
011	ZIP1
100	Unallocated.
101	UZP2
110	TRN2
111	ZIP2

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Advanced SIMD extract

This section describes the encoding of the Advanced SIMD extract instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

31	30	29	28	27	26	25	24	23 22	21	20 16	15	14 11	10	9 5	4 0
0	Q	1	0	1	1	1	0	op2	0	Rm	0	imm4	0	Rn	Rd

Decode fields op2	Instruction page
x1	Unallocated.
00	EXT
1x	Unallocated.

Advanced SIMD copy

This section describes the encoding of the Advanced SIMD copy instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

31	30	29	28	27	26	25	24	23	22	21	20 16	15	14 11	10	9 5	4 0
0	Q	op	0	1	1	1	0	0	0	0	imm5	0	imm4	1	Rn	Rd

Decode fields Q	op	imm5	imm4	Instruction page
-	-	x0000	-	Unallocated.
-	0	-	0000	DUP (element)
-	0	-	0001	DUP (general)
-	0	-	0010	Unallocated.
-	0	-	0100	Unallocated.
-	0	-	0110	Unallocated.
-	0	-	1xxx	Unallocated.
0	0	-	0011	Unallocated.
0	0	-	0101	SMOV
0	0	-	0111	UMOV
0	1	-	-	Unallocated.
1	0	-	0011	INS (general)

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields				Instruction page
Q	op	imm5	imm4	
1	0	-	0101	SMOV
1	0	x1000	0111	UMOV
1	1	-	-	INS (element)

Advanced SIMD three same (FP16)

This section describes the encoding of the Advanced SIMD three same (FP16) instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	11	10	9	5	4	0
0	Q	U	0	1	1	1	0	a	1	0	Rm		0	0	opcode	1	Rn			Rd	

Decode fields			Instruction page	Architecture version
U	a	opcode		
0	0	000	FMAXNM (vector)	ARMv8.2-FP16
0	0	001	FMLA (vector)	ARMv8.2-FP16
0	0	010	FADD (vector)	ARMv8.2-FP16
0	0	011	FMULX	ARMv8.2-FP16
0	0	100	FCMEQ (register)	ARMv8.2-FP16
0	0	101	Unallocated.	-
0	0	110	FMAX (vector)	ARMv8.2-FP16
0	0	111	FRECPS	ARMv8.2-FP16
0	1	000	FMINNM (vector)	ARMv8.2-FP16
0	1	001	FMLS (vector)	ARMv8.2-FP16
0	1	010	FSUB (vector)	ARMv8.2-FP16
0	1	011	Unallocated.	-
0	1	100	Unallocated.	-
0	1	101	Unallocated.	-
0	1	110	FMIN (vector)	ARMv8.2-FP16
0	1	111	FRSQRTS	ARMv8.2-FP16
1	0	000	FMAXNMP (vector)	ARMv8.2-FP16
1	0	001	Unallocated.	-
1	0	010	FADDP (vector)	ARMv8.2-FP16
1	0	011	FMUL (vector)	ARMv8.2-FP16

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields			Instruction page	Architecture version
U	a	opcode		
1	0	100	FCMGE (register)	ARMv8.2-FP16
1	0	101	FACGE	ARMv8.2-FP16
1	0	110	FMAXP (vector)	ARMv8.2-FP16
1	0	111	FDIV (vector)	ARMv8.2-FP16
1	1	000	FMINNMP (vector)	ARMv8.2-FP16
1	1	001	Unallocated.	-
1	1	010	FABD	ARMv8.2-FP16
1	1	011	Unallocated.	-
1	1	100	FCMGT (register)	ARMv8.2-FP16
1	1	101	FACGT	ARMv8.2-FP16
1	1	110	FMINP (vector)	ARMv8.2-FP16
1	1	111	Unallocated.	-

Advanced SIMD two-register miscellaneous (FP16)

This section describes the encoding of the Advanced SIMD two-register miscellaneous (FP16) instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

31	30	29	28 27 26 25	24	23	22 21 20	19 18 17 16	12	11 10	9 5	4 0
0	Q	U	0 1 1 1	0	a	1 1 1	1 0 0	opcode	1 0	Rn	Rd

Decode fields			Instruction page	Architecture version
U	a	opcode		
-	-	00xxx	Unallocated.	-
-	-	010xx	Unallocated.	-
-	-	10xxx	Unallocated.	-
-	-	11110	Unallocated.	-
-	0	011xx	Unallocated.	-
-	0	11111	Unallocated.	-
-	1	11100	Unallocated.	-
0	0	11000	FRINTN (vector)	ARMv8.2-FP16
0	0	11001	FRINTM (vector)	ARMv8.2-FP16
0	0	11010	FCVTNS (vector)	ARMv8.2-FP16

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields			Instruction page	Architecture version
U	a	opcode		
0	0	11011	FCVTMS (vector)	ARMv8.2-FP16
0	0	11100	FCVTAS (vector)	ARMv8.2-FP16
0	0	11101	SCVTF (vector, integer)	ARMv8.2-FP16
0	1	01100	FCMGT (zero)	ARMv8.2-FP16
0	1	01101	FCMEQ (zero)	ARMv8.2-FP16
0	1	01110	FCMLT (zero)	ARMv8.2-FP16
0	1	01111	FABS (vector)	ARMv8.2-FP16
0	1	11000	FRINTP (vector)	ARMv8.2-FP16
0	1	11001	FRINTZ (vector)	ARMv8.2-FP16
0	1	11010	FCVTPS (vector)	ARMv8.2-FP16
0	1	11011	FCVTZS (vector, integer)	ARMv8.2-FP16
0	1	11101	FRECPE	ARMv8.2-FP16
0	1	11111	Unallocated.	-
1	0	11000	FRINTA (vector)	ARMv8.2-FP16
1	0	11001	FRINTX (vector)	ARMv8.2-FP16
1	0	11010	FCVTNU (vector)	ARMv8.2-FP16
1	0	11011	FCVTMU (vector)	ARMv8.2-FP16
1	0	11100	FCVTAU (vector)	ARMv8.2-FP16
1	0	11101	UCVTF (vector, integer)	ARMv8.2-FP16
1	1	01100	FCMGE (zero)	ARMv8.2-FP16
1	1	01101	FCMLE (zero)	ARMv8.2-FP16
1	1	01110	Unallocated.	-
1	1	01111	FNEG (vector)	ARMv8.2-FP16
1	1	11000	Unallocated.	-
1	1	11001	FRINTI (vector)	ARMv8.2-FP16
1	1	11010	FCVTPU (vector)	ARMv8.2-FP16
1	1	11011	FCVTZU (vector, integer)	ARMv8.2-FP16
1	1	11101	FRSQRTE	ARMv8.2-FP16
1	1	11111	FSQRT (vector)	ARMv8.2-FP16

Advanced SIMD three-register extension

This section describes the encoding of the Advanced SIMD three-register extension instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31	30	29	28	27	26	25	24	23 22	21	20 16	15	14 11	10	9 5	4 0
0	Q	U	0	1	1	1	0	size	0	Rm	1	opcode	1	Rn	Rd

Decode fields				Instruction page	Architecture version
Q	U	size	opcode		
-	-	0x	0011	Unallocated.	-
-	-	11	0011	Unallocated.	-
-	0	-	0000	Unallocated.	-
-	0	-	0001	Unallocated.	-
-	0	-	0010	SDOT (vector)	ARMv8.2-DotProd
-	0	-	1xxx	Unallocated.	-
-	0	10	0011	USDOT (vector)	ARMv8.2-I8MM
-	1	-	0000	SQRDMLAH (vector)	ARMv8.1-RDMA
-	1	-	0001	SQRDMLSH (vector)	ARMv8.1-RDMA
-	1	-	0010	UDOT (vector)	ARMv8.2-DotProd
-	1	-	10xx	FCMLA	ARMv8.3-CompNum
-	1	-	11x0	FCADD	ARMv8.3-CompNum
-	1	00	1101	Unallocated.	-
-	1	00	1111	Unallocated.	-
-	1	01	1111	BFDOT (vector)	ARMv8.2-BF16
-	1	1x	1101	Unallocated.	-
-	1	10	0011	Unallocated.	-
-	1	10	1111	Unallocated.	-
-	1	11	1111	BFMLALB, BFMLALT (vector)	ARMv8.2-BF16
0	-	-	01xx	Unallocated.	-
0	1	01	1101	Unallocated.	-
1	-	0x	01xx	Unallocated.	-
1	-	1x	011x	Unallocated.	-
1	0	10	0100	SMMLA (vector)	ARMv8.2-I8MM
1	0	10	0101	USMMLA (vector)	ARMv8.2-I8MM
1	1	01	1101	BFMMLA	ARMv8.2-BF16
1	1	10	0100	UMMLA (vector)	ARMv8.2-I8MM
1	1	10	0101	Unallocated.	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Advanced SIMD two-register miscellaneous

This section describes the encoding of the Advanced SIMD two-register miscellaneous instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	17	16	12	11	10	9	5	4	0
0	Q	U	0	1	1	1	0	size	1	0	0	0	0	opcode		1	0	Rn		Rd	

Decode fields			Instruction page	Architecture version
U	size	opcode		
-	-	1000x	Unallocated.	-
-	-	10101	Unallocated.	-
-	0x	011xx	Unallocated.	-
-	1x	10111	Unallocated.	-
-	1x	11110	Unallocated.	-
-	11	10110	Unallocated.	-
0	-	00000	REV64	-
0	-	00001	REV16 (vector)	-
0	-	00010	SADDLP	-
0	-	00011	SUQADD	-
0	-	00100	CLS (vector)	-
0	-	00101	CNT	-
0	-	00110	SADALP	-
0	-	00111	SQABS	-
0	-	01000	CMGT (zero)	-
0	-	01001	CMEQ (zero)	-
0	-	01010	CMLT (zero)	-
0	-	01011	ABS	-
0	-	10010	XTN, XTN2	-
0	-	10011	Unallocated.	-
0	-	10100	SQXTN, SQXTN2	-
0	0x	10110	FCVTN, FCVTN2	-
0	0x	10111	FCVTL, FCVTL2	-
0	0x	11000	FRINTN (vector)	-
0	0x	11001	FRINTM (vector)	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields			Instruction page	Architecture version
U	size	opcode		
0	0x	11010	FCVTNS (vector)	-
0	0x	11011	FCVTMS (vector)	-
0	0x	11100	FCVTAS (vector)	-
0	0x	11101	SCVTF (vector, integer)	-
0	0x	11110	FRINT32Z (vector)	ARMv8.5-FRINT
0	0x	11111	FRINT64Z (vector)	ARMv8.5-FRINT
0	1x	01100	FCMGT (zero)	-
0	1x	01101	FCMEQ (zero)	-
0	1x	01110	FCMLT (zero)	-
0	1x	01111	FABS (vector)	-
0	1x	11000	FRINTP (vector)	-
0	1x	11001	FRINTZ (vector)	-
0	1x	11010	FCVTPS (vector)	-
0	1x	11011	FCVTZS (vector, integer)	-
0	1x	11100	URECPE	-
0	1x	11101	FRECPE	-
0	1x	11111	Unallocated.	-
0	10	10110	BFCVTN, BFCVTN2	ARMv8.2-BF16
1	-	00000	REV32 (vector)	-
1	-	00001	Unallocated.	-
1	-	00010	UADDLP	-
1	-	00011	USQADD	-
1	-	00100	CLZ (vector)	-
1	-	00110	UADALP	-
1	-	00111	SQNEG	-
1	-	01000	CMGE (zero)	-
1	-	01001	CMLE (zero)	-
1	-	01010	Unallocated.	-
1	-	01011	NEG (vector)	-
1	-	10010	SQXTUN, SQXTUN2	-
1	-	10011	SHLL, SHLL2	-
1	-	10100	UQXTN, UQXTN2	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields			Instruction page	Architecture version
U	size	opcode		
1	0x	10110	FCVTXN, FCVTXN2	-
1	0x	10111	Unallocated.	-
1	0x	11000	FRINTA (vector)	-
1	0x	11001	FRINTX (vector)	-
1	0x	11010	FCVTNU (vector)	-
1	0x	11011	FCVTMU (vector)	-
1	0x	11100	FCVTAU (vector)	-
1	0x	11101	UCVTF (vector, integer)	-
1	0x	11110	FRINT32X (vector)	ARMv8.5-FRINT
1	0x	11111	FRINT64X (vector)	ARMv8.5-FRINT
1	00	00101	NOT	-
1	01	00101	RBIT (vector)	-
1	1x	00101	Unallocated.	-
1	1x	01100	FCMGE (zero)	-
1	1x	01101	FCMLE (zero)	-
1	1x	01110	Unallocated.	-
1	1x	01111	FNEG (vector)	-
1	1x	11000	Unallocated.	-
1	1x	11001	FRINTI (vector)	-
1	1x	11010	FCVTPU (vector)	-
1	1x	11011	FCVTZU (vector, integer)	-
1	1x	11100	URSQRTE	-
1	1x	11101	FRSQRTE	-
1	1x	11111	FSQRT (vector)	-
1	10	10110	Unallocated.	-

Advanced SIMD across lanes

This section describes the encoding of the Advanced SIMD across lanes instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	12	11	10	9	5	4	0
0	Q	U	0	1	1	1	0	size		1	1	0	0	0	opcode		1	0	Rn		Rd	

Decode fields			Instruction page	Architecture version
U	size	opcode		
-	-	0000x	Unallocated.	-
-	-	00010	Unallocated.	-
-	-	001xx	Unallocated.	-
-	-	0100x	Unallocated.	-
-	-	01011	Unallocated.	-
-	-	01101	Unallocated.	-
-	-	01110	Unallocated.	-
-	-	10xxx	Unallocated.	-
-	-	1100x	Unallocated.	-
-	-	111xx	Unallocated.	-
0	-	00011	SADDLV	-
0	-	01010	SMAXV	-
0	-	11010	SMINV	-
0	-	11011	ADDV	-
0	00	01100	FMAXNMV - Half-precision variant	ARMv8.2-FP16
0	00	01111	FMAXV - Half-precision variant	ARMv8.2-FP16
0	01	01100	Unallocated.	-
0	01	01111	Unallocated.	-
0	10	01100	FMINNMV - Half-precision variant	ARMv8.2-FP16
0	10	01111	FMINV - Half-precision variant	ARMv8.2-FP16
0	11	01100	Unallocated.	-
0	11	01111	Unallocated.	-
1	-	00011	UADDLV	-
1	-	01010	UMAXV	-
1	-	11010	UMINV	-
1	-	11011	Unallocated.	-
1	0x	01100	FMAXNMV - Single-precision and double-precision variant	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields			Instruction page	Architecture version
U	size	opcode		
1	0x	01111	FMAXV - Single-precision and double-precision variant	-
1	1x	01100	FMINNMV - Single-precision and double-precision variant	-
1	1x	01111	FMINV - Single-precision and double-precision variant	-

Advanced SIMD three different

This section describes the encoding of the Advanced SIMD three different instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

31	30	29	28	27	26	25	24	23 22	21	20 16	15 12	11 10	9 5	4 0
0	Q	U	0	1	1	1	0	size	1	Rm	opcode	0 0	Rn	Rd

Decode fields		Instruction page
U	opcode	
-	1111	Unallocated.
0	0000	SADDL, SADDL2
0	0001	SADDW, SADDW2
0	0010	SSUBL, SSUBL2
0	0011	SSUBW, SSUBW2
0	0100	ADDHN, ADDHN2
0	0101	SABAL, SABAL2
0	0110	SUBHN, SUBHN2
0	0111	SABDL, SABDL2
0	1000	SMLAL, SMLAL2 (vector)
0	1001	SQDMLAL, SQDMLAL2 (vector)
0	1010	SMLSL, SMLSL2 (vector)
0	1011	SQDMLSL, SQDMLSL2 (vector)
0	1100	SMULL, SMULL2 (vector)
0	1101	SQDMULL, SQDMULL2 (vector)
0	1110	PMULL, PMULL2
1	0000	UADDL, UADDL2
1	0001	UADDW, UADDW2
1	0010	USUBL, USUBL2

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields		Instruction page
U	opcode	
1	0011	USUBW, USUBW2
1	0100	RADDHN, RADDHN2
1	0101	UABAL, UABAL2
1	0110	RSUBHN, RSUBHN2
1	0111	UABDL, UABDL2
1	1000	UMLAL, UMLAL2 (vector)
1	1001	Unallocated.
1	1010	UMLSL, UMLSL2 (vector)
1	1011	Unallocated.
1	1100	UMULL, UMULL2 (vector)
1	1101	Unallocated.
1	1110	Unallocated.

Advanced SIMD three same

This section describes the encoding of the Advanced SIMD three same instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

31	30	29	28	27	26	25	24	23 22	21	20 16	15 11	10	9 5	4 0
0	Q	U	0	1	1	1	0	size	1	Rm	opcode	1	Rn	Rd

Decode fields			Instruction page	Architecture version
U	size	opcode		
0	-	00000	SHADD	-
0	-	00001	SQADD	-
0	-	00010	SRHADD	-
0	-	00100	SHSUB	-
0	-	00101	SQSUB	-
0	-	00110	CMGT (register)	-
0	-	00111	CMGE (register)	-
0	-	01000	SSHL	-
0	-	01001	SQSHL (register)	-
0	-	01010	SRSHL	-
0	-	01011	SQRSHL	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields			Instruction page	Architecture version
U	size	opcode		
0	-	01100	SMAX	-
0	-	01101	SMIN	-
0	-	01110	SABD	-
0	-	01111	SABA	-
0	-	10000	ADD (vector)	-
0	-	10001	CMTST	-
0	-	10010	MLA (vector)	-
0	-	10011	MUL (vector)	-
0	-	10100	SMAXP	-
0	-	10101	SMINP	-
0	-	10110	SQDMULH (vector)	-
0	-	10111	ADDP (vector)	-
0	0x	11000	FMAXNM (vector)	-
0	0x	11001	FMLA (vector)	-
0	0x	11010	FADD (vector)	-
0	0x	11011	FMULX	-
0	0x	11100	FCMEQ (register)	-
0	0x	11110	FMAX (vector)	-
0	0x	11111	FRECPS	-
0	00	00011	AND (vector)	-
0	00	11101	FMLAL, FMLAL2 (vector) - FMLAL variant	ARMv8.2-FHM
0	01	00011	BIC (vector, register)	-
0	01	11101	Unallocated.	-
0	1x	11000	FMINNM (vector)	-
0	1x	11001	FMLS (vector)	-
0	1x	11010	FSUB (vector)	-
0	1x	11011	Unallocated.	-
0	1x	11100	Unallocated.	-
0	1x	11110	FMIN (vector)	-
0	1x	11111	FRSQRTS	-
0	10	00011	ORR (vector, register)	-
0	10	11101	FMLSL, FMLSL2 (vector) - FMLSL variant	ARMv8.2-FHM

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields			Instruction page	Architecture version
U	size	opcode		
0	11	00011	ORN (vector)	-
0	11	11101	Unallocated.	-
1	-	00000	UHADD	-
1	-	00001	UQADD	-
1	-	00010	URHADD	-
1	-	00100	UHSUB	-
1	-	00101	UQSUB	-
1	-	00110	CMHI (register)	-
1	-	00111	CMHS (register)	-
1	-	01000	USHL	-
1	-	01001	UQSHL (register)	-
1	-	01010	URSHL	-
1	-	01011	UQRSHL	-
1	-	01100	UMAX	-
1	-	01101	UMIN	-
1	-	01110	UABD	-
1	-	01111	UABA	-
1	-	10000	SUB (vector)	-
1	-	10001	CMEQ (register)	-
1	-	10010	MLS (vector)	-
1	-	10011	PMUL	-
1	-	10100	UMAXP	-
1	-	10101	UMINP	-
1	-	10110	SQRDMULH (vector)	-
1	-	10111	Unallocated.	-
1	0x	11000	FMAXNMP (vector)	-
1	0x	11010	FADDP (vector)	-
1	0x	11011	FMUL (vector)	-
1	0x	11100	FCMGE (register)	-
1	0x	11101	FACGE	-
1	0x	11110	FMAXP (vector)	-
1	0x	11111	FDIV (vector)	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields			Instruction page	Architecture version
U	size	opcode		
1	00	00011	EOR (vector)	-
1	00	11001	FMLAL, FMLAL2 (vector) - FMLAL2 variant	ARMv8.2-FHM
1	01	00011	BSL	-
1	01	11001	Unallocated.	-
1	1x	11000	FMINNMP (vector)	-
1	1x	11010	FABD	-
1	1x	11011	Unallocated.	-
1	1x	11100	FCMGT (register)	-
1	1x	11101	FACGT	-
1	1x	11110	FMINP (vector)	-
1	1x	11111	Unallocated.	-
1	10	00011	BIT	-
1	10	11001	FMLSL, FMLSL2 (vector) - FMLSL2 variant	ARMv8.2-FHM
1	11	00011	BIF	-
1	11	11001	Unallocated.	-

Advanced SIMD modified immediate

This section describes the encoding of the Advanced SIMD modified immediate instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15 – 12	11	10	9	8	7	6	5	4	0
0	Q	op	0	1	1	1	1	0	0	0	0	0	a	b	c	cmode	o2	1	d	e	f	g	h	Rd	

Decode fields				Instruction page	Architecture version
Q	op	cmode	o2		
-	0	0xxx	1	Unallocated.	-
-	0	0xx0	0	MOVI - 32-bit shifted immediate variant	-
-	0	0xx1	0	ORR (vector, immediate) - 32-bit variant	-
-	0	10xx	1	Unallocated.	-
-	0	10x0	0	MOVI - 16-bit shifted immediate variant	-
-	0	10x1	0	ORR (vector, immediate) - 16-bit variant	-
-	0	110x	0	MOVI - 32-bit shifting ones variant	-
-	0	110x	1	Unallocated.	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields				Instruction page	Architecture version
Q	op	cmode	o2		
-	0	1110	0	MOVI - 8-bit variant	-
-	0	1110	1	Unallocated.	-
-	0	1111	0	FMOV (vector, immediate) - Single-precision variant	-
-	0	1111	1	FMOV (vector, immediate) - Half-precision variant	ARMv8.2-FP16
-	1	-	1	Unallocated.	-
-	1	0xx0	0	MVNI - 32-bit shifted immediate variant	-
-	1	0xx1	0	BIC (vector, immediate) - 32-bit variant	-
-	1	10x0	0	MVNI - 16-bit shifted immediate variant	-
-	1	10x1	0	BIC (vector, immediate) - 16-bit variant	-
-	1	110x	0	MVNI - 32-bit shifting ones variant	-
0	1	1110	0	MOVI - 64-bit scalar variant	-
0	1	1111	0	Unallocated.	-
1	1	1110	0	MOVI - 64-bit vector variant	-
1	1	1111	0	FMOV (vector, immediate) - Double-precision variant	-

Advanced SIMD shift by immediate

This section describes the encoding of the Advanced SIMD shift by immediate instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

31	30	29	28	27	26	25	24	23 22	19 18 16	15 11	10	9 5	4 0
0	Q	U	0	1	1	1	1	0	!=0000	immb	opcode	1	Rn	Rd
									immh					

Decode fields		Instruction page
U	opcode	
-	00001	Unallocated.
-	00011	Unallocated.
-	00101	Unallocated.
-	00111	Unallocated.
-	01001	Unallocated.
-	01011	Unallocated.
-	01101	Unallocated.
-	01111	Unallocated.
-	10101	Unallocated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields		Instruction page
U	opcode	
-	1011x	Unallocated.
-	110xx	Unallocated.
-	11101	Unallocated.
-	11110	Unallocated.
0	00000	SSHR
0	00010	SSRA
0	00100	SRSHR
0	00110	SRSRA
0	01000	Unallocated.
0	01010	SHL
0	01100	Unallocated.
0	01110	SQSHL (immediate)
0	10000	SHRN, SHRN2
0	10001	RSHRN, RSHRN2
0	10010	SQSHRN, SQSHRN2
0	10011	SQRSHRN, SQRSHRN2
0	10100	SSHLL, SSHLL2
0	11100	SCVTF (vector, fixed-point)
0	11111	FCVTZS (vector, fixed-point)
1	00000	USHR
1	00010	USRA
1	00100	URSHR
1	00110	URSRA
1	01000	SRI
1	01010	SLI
1	01100	SQSHLU
1	01110	UQSHL (immediate)
1	10000	SQSHRUN, SQSHRUN2
1	10001	SQRSHRUN, SQRSHRUN2
1	10010	UQSHRN, UQSHRN2
1	10011	UQRSHRN, UQRSHRN2

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields		Instruction page
U	opcode	
1	10100	USHLL, USHLL2
1	11100	UCVTF (vector, fixed-point)
1	11111	FCVTZU (vector, fixed-point)

Advanced SIMD vector x indexed element

This section describes the encoding of the Advanced SIMD vector x indexed element instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

31	30	29	28	27	26	25	24	23 22	21	20	19 16	15 12	11	10	9 5	4 0
0	Q	U	0	1	1	1	1	size	L	M	Rm	opcode	H	0	Rn	Rd

Decode fields			Instruction page	Architecture version
U	size	opcode		
-	01	1001	Unallocated.	-
0	-	0010	SMLAL, SMLAL2 (by element)	-
0	-	0011	SQDMLAL, SQDMLAL2 (by element)	-
0	-	0110	SMLSL, SMLSL2 (by element)	-
0	-	0111	SQDMLSL, SQDMLSL2 (by element)	-
0	-	1000	MUL (by element)	-
0	-	1010	SMULL, SMULL2 (by element)	-
0	-	1011	SQDMULL, SQDMULL2 (by element)	-
0	-	1100	SQDMULH (by element)	-
0	-	1101	SQRDMULH (by element)	-
0	-	1110	SDOT (by element)	ARMv8.2-DotProd
0	0x	0000	Unallocated.	-
0	0x	0100	Unallocated.	-
0	00	0001	FMLA (by element) - Vector, half-precision variant	ARMv8.2-FP16
0	00	0101	FMLS (by element) - Vector, half-precision variant	ARMv8.2-FP16
0	00	1001	FMUL (by element) - Vector, half-precision variant	ARMv8.2-FP16
0	00	1111	SUDOT (by element)	ARMv8.2-I8MM
0	01	0001	Unallocated.	-
0	01	0101	Unallocated.	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

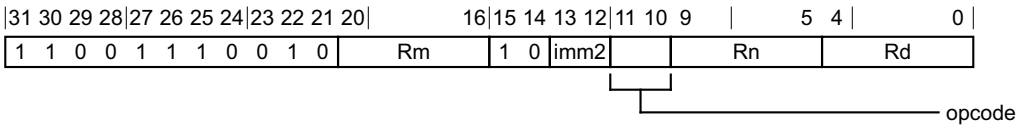
Decode fields			Instruction page	Architecture version
U	size	opcode		
0	01	1111	BFDOT (by element)	ARMv8.2-BF16
0	1x	0001	FMLA (by element) - Vector, single-precision and double-precision variant	-
0	1x	0101	FMLS (by element) - Vector, single-precision and double-precision variant	-
0	1x	1001	FMUL (by element) - Vector, single-precision and double-precision variant	-
0	10	0000	FMLAL, FMLAL2 (by element) - FMLAL variant	ARMv8.2-FHM
0	10	0100	FMLSL, FMLSL2 (by element) - FMLSL variant	ARMv8.2-FHM
0	10	1111	USDOT (by element)	ARMv8.2-I8MM
0	11	0000	Unallocated.	-
0	11	0100	Unallocated.	-
0	11	1111	BFMLALB, BFMLALT (by element)	ARMv8.2-BF16
1	-	0000	MLA (by element)	-
1	-	0010	UMLAL, UMLAL2 (by element)	-
1	-	0100	MLS (by element)	-
1	-	0110	UMLSL, UMLSL2 (by element)	-
1	-	1010	UMULL, UMULL2 (by element)	-
1	-	1011	Unallocated.	-
1	-	1101	SQRDMLAH (by element)	ARMv8.1-RDMA
1	-	1110	UDOT (by element)	ARMv8.2-DotProd
1	-	1111	SQRDMLSH (by element)	ARMv8.1-RDMA
1	0x	1000	Unallocated.	-
1	0x	1100	Unallocated.	-
1	00	0001	Unallocated.	-
1	00	0011	Unallocated.	-
1	00	0101	Unallocated.	-
1	00	0111	Unallocated.	-
1	00	1001	FMULX (by element) - Vector, half-precision variant	ARMv8.2-FP16
1	01	0xx1	FCMLA (by element)	ARMv8.3-CompNum
1	1x	1001	FMULX (by element) - Vector, single-precision and double-precision variant	-
1	10	0xx1	FCMLA (by element)	ARMv8.3-CompNum
1	10	1000	FMLAL, FMLAL2 (by element) - FMLAL2 variant	ARMv8.2-FHM
1	10	1100	FMLSL, FMLSL2 (by element) - FMLSL2 variant	ARMv8.2-FHM
1	11	0001	Unallocated.	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields			Instruction page	Architecture version
U	size	opcode		
1	11	0011	Unallocated.	-
1	11	0101	Unallocated.	-
1	11	0111	Unallocated.	-
1	11	1000	Unallocated.	-
1	11	1100	Unallocated.	-

Cryptographic three-register, imm2

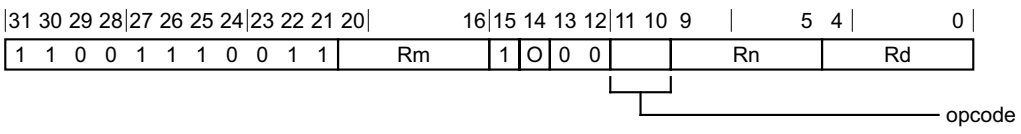
This section describes the encoding of the Cryptographic three-register, imm2 instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.



Decode fields	Instruction page	Architecture version
opcode		
00	SM3TT1A	ARMv8.2-SM3
01	SM3TT1B	ARMv8.2-SM3
10	SM3TT2A	ARMv8.2-SM3
11	SM3TT2B	ARMv8.2-SM3

Cryptographic three-register SHA 512

This section describes the encoding of the Cryptographic three-register SHA 512 instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.



Decode fields		Instruction page	Architecture version
O	opcode		
0	00	SHA512H	ARMv8.2-SHA2
0	01	SHA512H2	ARMv8.2-SHA2
0	10	SHA512SU1	ARMv8.2-SHA2

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields		Instruction page	Architecture version
O	opcode		
0	11	RAX1	ARMv8.2-SHA3
1	00	SM3PARTW1	ARMv8.2-SM3
1	01	SM3PARTW2	ARMv8.2-SM3
1	10	SM4EKEY	ARMv8.2-SM4
1	11	Unallocated.	-

Cryptographic four-register

This section describes the encoding of the Cryptographic four-register instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

31 30 29 28	27 26 25 24	23 22 21 20	16	15 14	10 9	5 4	0
1 1 0 0	1 1 1 0 0	Op0	Rm	0	Ra	Rn	Rd

Decode fields	Instruction page	Architecture version
Op0		
00	EOR3	ARMv8.2-SHA3
01	BCAX	ARMv8.2-SHA3
10	SM3SS1	ARMv8.2-SM3
11	Unallocated.	-

Cryptographic two-register SHA 512

This section describes the encoding of the Cryptographic two-register SHA 512 instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10	9	5	4	0
1 1 0 0	1 1 1 0	1 1 0 0	0 0 0 0	1 0 0 0		Rn		Rd	

opcode

Decode fields	Instruction page	Architecture version
opcode		
00	SHA512SU0	ARMv8.2-SHA2
01	SM4E	ARMv8.2-SM4
1x	Unallocated.	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Conversion between floating-point and fixed-point

This section describes the encoding of the Conversion between floating-point and fixed-point instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

31	30	29	28	27	26	25	24	23 22	21	20 19	18 16	15 10	9 5	4 0
sf	0	S	1	1	1	1	0	ptype	0	rmode	opcode	scale	Rn	Rd

Decode fields						Instruction page	Architecture version
sf	S	ptype	rmode	opcode	scale		
-	-	-	-	1xx	-	Unallocated.	-
-	-	-	x0	00x	-	Unallocated.	-
-	-	-	x1	01x	-	Unallocated.	-
-	-	-	0x	00x	-	Unallocated.	-
-	-	-	1x	01x	-	Unallocated.	-
-	-	10	-	-	-	Unallocated.	-
-	1	-	-	-	-	Unallocated.	-
0	-	-	-	-	0xxxxx	Unallocated.	-
0	0	00	00	010	-	SCVTF (scalar, fixed-point) - 32-bit to single-precision variant	-
0	0	00	00	011	-	UCVTF (scalar, fixed-point) - 32-bit to single-precision variant	-
0	0	00	11	000	-	FCVTZS (scalar, fixed-point) - Single-precision to 32-bit variant	-
0	0	00	11	001	-	FCVTZU (scalar, fixed-point) - Single-precision to 32-bit variant	-
0	0	01	00	010	-	SCVTF (scalar, fixed-point) - 32-bit to double-precision variant	-
0	0	01	00	011	-	UCVTF (scalar, fixed-point) - 32-bit to double-precision variant	-
0	0	01	11	000	-	FCVTZS (scalar, fixed-point) - Double-precision to 32-bit variant	-
0	0	01	11	001	-	FCVTZU (scalar, fixed-point) - Double-precision to 32-bit variant	-
0	0	11	00	010	-	SCVTF (scalar, fixed-point) - 32-bit to half-precision variant	ARMv8.2-FP16
0	0	11	00	011	-	UCVTF (scalar, fixed-point) - 32-bit to half-precision variant	ARMv8.2-FP16
0	0	11	11	000	-	FCVTZS (scalar, fixed-point) - Half-precision to 32-bit variant	ARMv8.2-FP16

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields						Instruction page	Architecture version
sf	S	ptype	rmode	opcode	scale		
0	0	11	11	001	-	FCVTZU (scalar, fixed-point) - Half-precision to 32-bit variant	ARMv8.2-FP16
1	0	00	00	010	-	SCVTF (scalar, fixed-point) - 64-bit to single-precision variant	-
1	0	00	00	011	-	UCVTF (scalar, fixed-point) - 64-bit to single-precision variant	-
1	0	00	11	000	-	FCVTZS (scalar, fixed-point) - Single-precision to 64-bit variant	-
1	0	00	11	001	-	FCVTZU (scalar, fixed-point) - Single-precision to 64-bit variant	-
1	0	01	00	010	-	SCVTF (scalar, fixed-point) - 64-bit to double-precision variant	-
1	0	01	00	011	-	UCVTF (scalar, fixed-point) - 64-bit to double-precision variant	-
1	0	01	11	000	-	FCVTZS (scalar, fixed-point) - Double-precision to 64-bit variant	-
1	0	01	11	001	-	FCVTZU (scalar, fixed-point) - Double-precision to 64-bit variant	-
1	0	11	00	010	-	SCVTF (scalar, fixed-point) - 64-bit to half-precision variant	ARMv8.2-FP16
1	0	11	00	011	-	UCVTF (scalar, fixed-point) - 64-bit to half-precision variant	ARMv8.2-FP16
1	0	11	11	000	-	FCVTZS (scalar, fixed-point) - Half-precision to 64-bit variant	ARMv8.2-FP16
1	0	11	11	001	-	FCVTZU (scalar, fixed-point) - Half-precision to 64-bit variant	ARMv8.2-FP16

Conversion between floating-point and integer

This section describes the encoding of the Conversion between floating-point and integer instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31	30	29	28	27	26	25	24	23 22	21	20 19	18 16	15	14	13	12	11	10	9 5	4 0
sf	0	S	1	1	1	1	0	ptype	1	rmode	opcode	0	0	0	0	0	0	Rn	Rd

Decode fields					Instruction page	Architecture version
sf	S	ptype	rmode	opcode		
-	-	-	x1	01x	Unallocated.	-
-	-	-	x1	10x	Unallocated.	-
-	-	-	1x	01x	Unallocated.	-
-	-	-	1x	10x	Unallocated.	-
-	0	10	-	0xx	Unallocated.	-
-	0	10	-	10x	Unallocated.	-
-	1	-	-	-	Unallocated.	-
0	0	00	x1	11x	Unallocated.	-
0	0	00	00	000	FCVTNS (scalar) - Single-precision to 32-bit variant	-
0	0	00	00	001	FCVTNU (scalar) - Single-precision to 32-bit variant	-
0	0	00	00	010	SCVTF (scalar, integer) - 32-bit to single-precision variant	-
0	0	00	00	011	UCVTF (scalar, integer) - 32-bit to single-precision variant	-
0	0	00	00	100	FCVTAS (scalar) - Single-precision to 32-bit variant	-
0	0	00	00	101	FCVTAU (scalar) - Single-precision to 32-bit variant	-
0	0	00	00	110	FMOV (general) - Single-precision to 32-bit variant	-
0	0	00	00	111	FMOV (general) - 32-bit to single-precision variant	-
0	0	00	01	000	FCVTPS (scalar) - Single-precision to 32-bit variant	-
0	0	00	01	001	FCVTPU (scalar) - Single-precision to 32-bit variant	-
0	0	00	1x	11x	Unallocated.	-
0	0	00	10	000	FCVTMS (scalar) - Single-precision to 32-bit variant	-
0	0	00	10	001	FCVTMU (scalar) - Single-precision to 32-bit variant	-
0	0	00	11	000	FCVTZS (scalar, integer) - Single-precision to 32-bit variant	-
0	0	00	11	001	FCVTZU (scalar, integer) - Single-precision to 32-bit variant	-
0	0	01	0x	11x	Unallocated.	-
0	0	01	00	000	FCVTNS (scalar) - Double-precision to 32-bit variant	-
0	0	01	00	001	FCVTNU (scalar) - Double-precision to 32-bit variant	-
0	0	01	00	010	SCVTF (scalar, integer) - 32-bit to double-precision variant	-
0	0	01	00	011	UCVTF (scalar, integer) - 32-bit to double-precision variant	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields					Instruction page	Architecture version
sf	S	ptype	rmode	opcode		
0	0	01	00	100	FCVTAS (scalar) - Double-precision to 32-bit variant	-
0	0	01	00	101	FCVTAU (scalar) - Double-precision to 32-bit variant	-
0	0	01	01	000	FCVTPS (scalar) - Double-precision to 32-bit variant	-
0	0	01	01	001	FCVTPU (scalar) - Double-precision to 32-bit variant	-
0	0	01	10	000	FCVTMS (scalar) - Double-precision to 32-bit variant	-
0	0	01	10	001	FCVTMU (scalar) - Double-precision to 32-bit variant	-
0	0	01	10	11x	Unallocated.	-
0	0	01	11	000	FCVTZS (scalar, integer) - Double-precision to 32-bit variant	-
0	0	01	11	001	FCVTZU (scalar, integer) - Double-precision to 32-bit variant	-
0	0	01	11	110	FJCVTZS	ARMv8.3-JConv
0	0	01	11	111	Unallocated.	-
0	0	10	-	11x	Unallocated.	-
0	0	11	00	000	FCVTNS (scalar) - Half-precision to 32-bit variant	ARMv8.2-FP16
0	0	11	00	001	FCVTNU (scalar) - Half-precision to 32-bit variant	ARMv8.2-FP16
0	0	11	00	010	SCVTF (scalar, integer) - 32-bit to half-precision variant	ARMv8.2-FP16
0	0	11	00	011	UCVTF (scalar, integer) - 32-bit to half-precision variant	ARMv8.2-FP16
0	0	11	00	100	FCVTAS (scalar) - Half-precision to 32-bit variant	ARMv8.2-FP16
0	0	11	00	101	FCVTAU (scalar) - Half-precision to 32-bit variant	ARMv8.2-FP16
0	0	11	00	110	FMOV (general) - Half-precision to 32-bit variant	ARMv8.2-FP16
0	0	11	00	111	FMOV (general) - 32-bit to half-precision variant	ARMv8.2-FP16
0	0	11	01	000	FCVTPS (scalar) - Half-precision to 32-bit variant	ARMv8.2-FP16
0	0	11	01	001	FCVTPU (scalar) - Half-precision to 32-bit variant	ARMv8.2-FP16
0	0	11	10	000	FCVTMS (scalar) - Half-precision to 32-bit variant	ARMv8.2-FP16
0	0	11	10	001	FCVTMU (scalar) - Half-precision to 32-bit variant	ARMv8.2-FP16
0	0	11	11	000	FCVTZS (scalar, integer) - Half-precision to 32-bit variant	ARMv8.2-FP16
0	0	11	11	001	FCVTZU (scalar, integer) - Half-precision to 32-bit variant	ARMv8.2-FP16
1	0	00	-	11x	Unallocated.	-
1	0	00	00	000	FCVTNS (scalar) - Single-precision to 64-bit variant	-
1	0	00	00	001	FCVTNU (scalar) - Single-precision to 64-bit variant	-
1	0	00	00	010	SCVTF (scalar, integer) - 64-bit to single-precision variant	-
1	0	00	00	011	UCVTF (scalar, integer) - 64-bit to single-precision variant	-
1	0	00	00	100	FCVTAS (scalar) - Single-precision to 64-bit variant	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields					Instruction page	Architecture version
sf	S	ptype	rmode	opcode		
1	0	00	00	101	FCVTAU (scalar) - Single-precision to 64-bit variant	-
1	0	00	01	000	FCVTPS (scalar) - Single-precision to 64-bit variant	-
1	0	00	01	001	FCVTPU (scalar) - Single-precision to 64-bit variant	-
1	0	00	10	000	FCVTMS (scalar) - Single-precision to 64-bit variant	-
1	0	00	10	001	FCVTMU (scalar) - Single-precision to 64-bit variant	-
1	0	00	11	000	FCVTZS (scalar, integer) - Single-precision to 64-bit variant	-
1	0	00	11	001	FCVTZU (scalar, integer) - Single-precision to 64-bit variant	-
1	0	01	x1	11x	Unallocated.	-
1	0	01	00	000	FCVTNS (scalar) - Double-precision to 64-bit variant	-
1	0	01	00	001	FCVTNU (scalar) - Double-precision to 64-bit variant	-
1	0	01	00	010	SCVTF (scalar, integer) - 64-bit to double-precision variant	-
1	0	01	00	011	UCVTF (scalar, integer) - 64-bit to double-precision variant	-
1	0	01	00	100	FCVTAS (scalar) - Double-precision to 64-bit variant	-
1	0	01	00	101	FCVTAU (scalar) - Double-precision to 64-bit variant	-
1	0	01	00	110	FMOV (general) - Double-precision to 64-bit variant	-
1	0	01	00	111	FMOV (general) - 64-bit to double-precision variant	-
1	0	01	01	000	FCVTPS (scalar) - Double-precision to 64-bit variant	-
1	0	01	01	001	FCVTPU (scalar) - Double-precision to 64-bit variant	-
1	0	01	1x	11x	Unallocated.	-
1	0	01	10	000	FCVTMS (scalar) - Double-precision to 64-bit variant	-
1	0	01	10	001	FCVTMU (scalar) - Double-precision to 64-bit variant	-
1	0	01	11	000	FCVTZS (scalar, integer) - Double-precision to 64-bit variant	-
1	0	01	11	001	FCVTZU (scalar, integer) - Double-precision to 64-bit variant	-
1	0	10	x0	11x	Unallocated.	-
1	0	10	01	110	FMOV (general) - Top half of 128-bit to 64-bit variant	-
1	0	10	01	111	FMOV (general) - 64-bit to top half of 128-bit variant	-
1	0	10	1x	11x	Unallocated.	-
1	0	11	00	000	FCVTNS (scalar) - Half-precision to 64-bit variant	ARMv8.2-FP16
1	0	11	00	001	FCVTNU (scalar) - Half-precision to 64-bit variant	ARMv8.2-FP16
1	0	11	00	010	SCVTF (scalar, integer) - 64-bit to half-precision variant	ARMv8.2-FP16
1	0	11	00	011	UCVTF (scalar, integer) - 64-bit to half-precision variant	ARMv8.2-FP16
1	0	11	00	100	FCVTAS (scalar) - Half-precision to 64-bit variant	ARMv8.2-FP16

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields					Instruction page	Architecture version
sf	S	ptype	rmode	opcode		
1	0	11	00	101	FCVTAU (scalar) - Half-precision to 64-bit variant	ARMv8.2-FP16
1	0	11	00	110	FMOV (general) - Half-precision to 64-bit variant	ARMv8.2-FP16
1	0	11	00	111	FMOV (general) - 64-bit to half-precision variant	ARMv8.2-FP16
1	0	11	01	000	FCVTPS (scalar) - Half-precision to 64-bit variant	ARMv8.2-FP16
1	0	11	01	001	FCVTPU (scalar) - Half-precision to 64-bit variant	ARMv8.2-FP16
1	0	11	10	000	FCVTMS (scalar) - Half-precision to 64-bit variant	ARMv8.2-FP16
1	0	11	10	001	FCVTMU (scalar) - Half-precision to 64-bit variant	ARMv8.2-FP16
1	0	11	11	000	FCVTZS (scalar, integer) - Half-precision to 64-bit variant	ARMv8.2-FP16
1	0	11	11	001	FCVTZU (scalar, integer) - Half-precision to 64-bit variant	ARMv8.2-FP16

Floating-point data-processing (1 source)

This section describes the encoding of the Floating-point data-processing (1 source) instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

31	30	29	28	27	26	25	24	23 22	21	20 15	14	13	12	11	10	9 5	4 0
M	0	S	1	1	1	1	0	ptype	1	opcode	1	0	0	0	0	Rn	Rd

Decode fields				Instruction page	Architecture version
M	S	ptype	opcode		
-	-	-	1xxxxx	Unallocated.	-
-	1	-	-	Unallocated.	-
0	0	00	000000	FMOV (register) - Single-precision variant	-
0	0	00	000001	FABS (scalar) - Single-precision variant	-
0	0	00	000010	FNEG (scalar) - Single-precision variant	-
0	0	00	000011	FSQRT (scalar) - Single-precision variant	-
0	0	00	000100	Unallocated.	-
0	0	00	000101	FCVT - Single-precision to double-precision variant	-
0	0	00	000110	Unallocated.	-
0	0	00	000111	FCVT - Single-precision to half-precision variant	-
0	0	00	001000	FRINTN (scalar) - Single-precision variant	-
0	0	00	001001	FRINTP (scalar) - Single-precision variant	-
0	0	00	001010	FRINTM (scalar) - Single-precision variant	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields				Instruction page	Architecture version
M	S	ptype	opcode		
0	0	00	001011	FRINTZ (scalar) - Single-precision variant	-
0	0	00	001100	FRINTA (scalar) - Single-precision variant	-
0	0	00	001101	Unallocated.	-
0	0	00	001110	FRINTX (scalar) - Single-precision variant	-
0	0	00	001111	FRINTI (scalar) - Single-precision variant	-
0	0	00	010000	FRINT32Z (scalar) - Single-precision variant	ARMv8.5-FRINT
0	0	00	010001	FRINT32X (scalar) - Single-precision variant	ARMv8.5-FRINT
0	0	00	010010	FRINT64Z (scalar) - Single-precision variant	ARMv8.5-FRINT
0	0	00	010011	FRINT64X (scalar) - Single-precision variant	ARMv8.5-FRINT
0	0	00	0101xx	Unallocated.	-
0	0	00	011xxx	Unallocated.	-
0	0	01	000000	FMOV (register) - Double-precision variant	-
0	0	01	000001	FABS (scalar) - Double-precision variant	-
0	0	01	000010	FNEG (scalar) - Double-precision variant	-
0	0	01	000011	FSQRT (scalar) - Double-precision variant	-
0	0	01	000100	FCVT - Double-precision to single-precision variant	-
0	0	01	000101	Unallocated.	-
0	0	01	000110	BFCVT	ARMv8.2-BF16
0	0	01	000111	FCVT - Double-precision to half-precision variant	-
0	0	01	001000	FRINTN (scalar) - Double-precision variant	-
0	0	01	001001	FRINTP (scalar) - Double-precision variant	-
0	0	01	001010	FRINTM (scalar) - Double-precision variant	-
0	0	01	001011	FRINTZ (scalar) - Double-precision variant	-
0	0	01	001100	FRINTA (scalar) - Double-precision variant	-
0	0	01	001101	Unallocated.	-
0	0	01	001110	FRINTX (scalar) - Double-precision variant	-
0	0	01	001111	FRINTI (scalar) - Double-precision variant	-
0	0	01	010000	FRINT32Z (scalar) - Double-precision variant	ARMv8.5-FRINT
0	0	01	010001	FRINT32X (scalar) - Double-precision variant	ARMv8.5-FRINT
0	0	01	010010	FRINT64Z (scalar) - Double-precision variant	ARMv8.5-FRINT
0	0	01	010011	FRINT64X (scalar) - Double-precision variant	ARMv8.5-FRINT
0	0	01	0101xx	Unallocated.	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields				Instruction page	Architecture version
M	S	ptype	opcode		
0	0	01	011xxx	Unallocated.	-
0	0	10	0xxxxx	Unallocated.	-
0	0	11	000000	FMOV (register) - Half-precision variant	ARMv8.2-FP16
0	0	11	000001	FABS (scalar) - Half-precision variant	ARMv8.2-FP16
0	0	11	000010	FNEG (scalar) - Half-precision variant	ARMv8.2-FP16
0	0	11	000011	FSQRT (scalar) - Half-precision variant	ARMv8.2-FP16
0	0	11	000100	FCVT - Half-precision to single-precision variant	-
0	0	11	000101	FCVT - Half-precision to double-precision variant	-
0	0	11	00011x	Unallocated.	-
0	0	11	001000	FRINTN (scalar) - Half-precision variant	ARMv8.2-FP16
0	0	11	001001	FRINTP (scalar) - Half-precision variant	ARMv8.2-FP16
0	0	11	001010	FRINTM (scalar) - Half-precision variant	ARMv8.2-FP16
0	0	11	001011	FRINTZ (scalar) - Half-precision variant	ARMv8.2-FP16
0	0	11	001100	FRINTA (scalar) - Half-precision variant	ARMv8.2-FP16
0	0	11	001101	Unallocated.	-
0	0	11	001110	FRINTX (scalar) - Half-precision variant	ARMv8.2-FP16
0	0	11	001111	FRINTI (scalar) - Half-precision variant	ARMv8.2-FP16
0	0	11	01xxxx	Unallocated.	-
1	-	-	-	Unallocated.	-

Floating-point compare

This section describes the encoding of the Floating-point compare instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
M	0	S	1	1	1	1	0	ptype		1	Rm		op	1	0	0	0	Rn			opcode2	

Decode fields					Instruction page	Architecture version
M	S	ptype	op	opcode2		
-	-	-	-	xxxx1	Unallocated.	-
-	-	-	-	xxx1x	Unallocated.	-
-	-	-	-	xx1xx	Unallocated.	-
-	-	-	x1	-	Unallocated.	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields					Instruction page	Architecture version
M	S	ptype	op	opcode2		
-	-	-	1x	-	Unallocated.	-
-	-	10	-	-	Unallocated.	-
-	1	-	-	-	Unallocated.	-
0	0	00	00	00000	FCMP	-
0	0	00	00	01000	FCMP	-
0	0	00	00	10000	FCMPE	-
0	0	00	00	11000	FCMPE	-
0	0	01	00	00000	FCMP	-
0	0	01	00	01000	FCMP	-
0	0	01	00	10000	FCMPE	-
0	0	01	00	11000	FCMPE	-
0	0	11	00	00000	FCMP	ARMv8.2-FP16
0	0	11	00	01000	FCMP	ARMv8.2-FP16
0	0	11	00	10000	FCMPE	ARMv8.2-FP16
0	0	11	00	11000	FCMPE	ARMv8.2-FP16
1	-	-	-	-	Unallocated.	-

Floating-point immediate

This section describes the encoding of the Floating-point immediate instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

31	30	29	28	27	26	25	24	23 22	21	20 … 13	12	11	10	9 … 5	4 … 0
M	0	S	1	1	1	1	0	ptype	1	imm8	1	0	0	imm5	Rd

Decode fields				Instruction page	Architecture version
M	S	ptype	imm5		
-	-	-	xxxx1	Unallocated.	-
-	-	-	xxx1x	Unallocated.	-
-	-	-	xx1xx	Unallocated.	-
-	-	-	x1xxx	Unallocated.	-
-	-	-	1xxxx	Unallocated.	-
-	-	10	-	Unallocated.	-
-	1	-	-	Unallocated.	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields				Instruction page	Architecture version
M	S	ptype	imm5		
0	0	00	00000	FMOV (scalar, immediate) - Single-precision variant	-
0	0	01	00000	FMOV (scalar, immediate) - Double-precision variant	-
0	0	11	00000	FMOV (scalar, immediate) - Half-precision variant	ARMv8.2-FP16
1	-	-	-	Unallocated.	-

Floating-point conditional compare

This section describes the encoding of the Floating-point conditional compare instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

31	30	29	28	27	26	25	24	23 22	21	20 16	15 12	11	10	9 5	4	3 0
M	0	S	1	1	1	1	0	ptype	1	Rm	cond	0	1	Rn	op	nzcv

Decode fields				Instruction page	Architecture version
M	S	ptype	op		
-	-	10	-	Unallocated.	-
-	1	-	-	Unallocated.	-
0	0	00	0	FCCMP - Single-precision variant	-
0	0	00	1	FCCMPE - Single-precision variant	-
0	0	01	0	FCCMP - Double-precision variant	-
0	0	01	1	FCCMPE - Double-precision variant	-
0	0	11	0	FCCMP - Half-precision variant	ARMv8.2-FP16
0	0	11	1	FCCMPE - Half-precision variant	ARMv8.2-FP16
1	-	-	-	Unallocated.	-

Floating-point data-processing (2 source)

This section describes the encoding of the Floating-point data-processing (2 source) instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31	30	29	28	27	26	25	24	23 22	21	20 16	15 12	11 10	9 5	4 0
M	0	S	1	1	1	1	0	ptype	1	Rm	opcode	1 0	Rn	Rd

Decode fields				Instruction page	Architecture version
M	S	ptype	opcode		
-	-	-	1xx1	Unallocated.	-
-	-	-	1x1x	Unallocated.	-
-	-	-	11xx	Unallocated.	-
-	-	10	-	Unallocated.	-
-	1	-	-	Unallocated.	-
0	0	00	0000	FMUL (scalar) - Single-precision variant	-
0	0	00	0001	FDIV (scalar) - Single-precision variant	-
0	0	00	0010	FADD (scalar) - Single-precision variant	-
0	0	00	0011	FSUB (scalar) - Single-precision variant	-
0	0	00	0100	FMAX (scalar) - Single-precision variant	-
0	0	00	0101	FMIN (scalar) - Single-precision variant	-
0	0	00	0110	FMAXNM (scalar) - Single-precision variant	-
0	0	00	0111	FMINNM (scalar) - Single-precision variant	-
0	0	00	1000	FNMUL (scalar) - Single-precision variant	-
0	0	01	0000	FMUL (scalar) - Double-precision variant	-
0	0	01	0001	FDIV (scalar) - Double-precision variant	-
0	0	01	0010	FADD (scalar) - Double-precision variant	-
0	0	01	0011	FSUB (scalar) - Double-precision variant	-
0	0	01	0100	FMAX (scalar) - Double-precision variant	-
0	0	01	0101	FMIN (scalar) - Double-precision variant	-
0	0	01	0110	FMAXNM (scalar) - Double-precision variant	-
0	0	01	0111	FMINNM (scalar) - Double-precision variant	-
0	0	01	1000	FNMUL (scalar) - Double-precision variant	-
0	0	11	0000	FMUL (scalar) - Half-precision variant	ARMv8.2-FP16
0	0	11	0001	FDIV (scalar) - Half-precision variant	ARMv8.2-FP16
0	0	11	0010	FADD (scalar) - Half-precision variant	ARMv8.2-FP16
0	0	11	0011	FSUB (scalar) - Half-precision variant	ARMv8.2-FP16
0	0	11	0100	FMAX (scalar) - Half-precision variant	ARMv8.2-FP16

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields				Instruction page	Architecture version
M	S	ptype	opcode		
0	0	11	0101	FMIN (scalar) - Half-precision variant	ARMv8.2-FP16
0	0	11	0110	FMAXNM (scalar) - Half-precision variant	ARMv8.2-FP16
0	0	11	0111	FMINNM (scalar) - Half-precision variant	ARMv8.2-FP16
0	0	11	1000	FNMUL (scalar) - Half-precision variant	ARMv8.2-FP16
1	-	-	-	Unallocated.	-

Floating-point conditional select

This section describes the encoding of the Floating-point conditional select instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

31	30	29	28	27	26	25	24	23 22	21	20 16	15 12	11	10	9 5	4 0
M	0	S	1	1	1	1	0	ptype	1	Rm	cond	1	1	Rn	Rd

Decode fields			Instruction page	Architecture version
M	S	ptype		
-	-	10	Unallocated.	-
-	1	-	Unallocated.	-
0	0	00	FCSEL - Single-precision variant	-
0	0	01	FCSEL - Double-precision variant	-
0	0	11	FCSEL - Half-precision variant	ARMv8.2-FP16
1	-	-	Unallocated.	-

Floating-point data-processing (3 source)

This section describes the encoding of the Floating-point data-processing (3 source) instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-323.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31	30	29	28	27	26	25	24	23 22	21	20 16	15	14 10	9 5	4 0
M	0	S	1	1	1	1	1	ptype	o1	Rm	o0	Ra	Rn	Rd

Decode fields					Instruction page	Architecture version
M	S	ptype	o1	o0		
-	-	10	-	-	Unallocated.	-
-	1	-	-	-	Unallocated.	-
0	0	00	0	0	FMADD - Single-precision variant	-
0	0	00	0	1	FMSUB - Single-precision variant	-
0	0	00	1	0	FNMADD - Single-precision variant	-
0	0	00	1	1	FNMSUB - Single-precision variant	-
0	0	01	0	0	FMADD - Double-precision variant	-
0	0	01	0	1	FMSUB - Double-precision variant	-
0	0	01	1	0	FNMADD - Double-precision variant	-
0	0	01	1	1	FNMSUB - Double-precision variant	-
0	0	11	0	0	FMADD - Half-precision variant	ARMv8.2-FP16
0	0	11	0	1	FMSUB - Half-precision variant	ARMv8.2-FP16
0	0	11	1	0	FNMADD - Half-precision variant	ARMv8.2-FP16
0	0	11	1	1	FNMSUB - Half-precision variant	ARMv8.2-FP16
1	-	-	-	-	Unallocated.	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter C5
The A64 System Instruction Class

This chapter describes the A64 System instruction class, and the System instruction class encoding space, that is a subset of the System registers encoding space. It contains the following sections:

- *The System instruction class encoding space* on page C5-376.
- *Special-purpose registers* on page C5-389.
- *A64 System instructions for cache maintenance* on page C5-475.
- *A64 System instructions for address translation* on page C5-536.
- *A64 System instructions for TLB maintenance* on page C5-561.
- *A64 System instructions for prediction restriction* on page C5-760.

See *General information about the A64 instruction descriptions* on page C2-197 for information about entries used in the instruction encoding descriptions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.1 The System instruction class encoding space

Part of the A64 instruction encoding space is assigned to instructions that access the System register encoding space. These instructions provide:

- Access to *System registers*, including the debug registers, that provide system control, and system status information.
- Access to Special-purpose registers such as SPSR_ELx, ELR_ELx, and the equivalent fields of the Process State.
- The cache and TLB maintenance instructions and address translation instructions.
- Barriers and the `CLREX` instruction.
- Architectural hint instructions.

This section describes the general model for accessing this functionality.

Note

- See *Fixed values in AArch64 instruction and System register descriptions* on page C2-197 for information about abbreviations used in the System instruction descriptions.
- In AArch32 state much of this functionality is provided through the System register interface described in *The AArch32 System register interface* on page G1-5769. In AArch64 state, the parameters used to characterize the System register encoding space are {`op0`, `op1`, `CRn`, `CRm`, `op2`}. These are based on the parameters that characterize the AArch32 System register encoding space, which reflect the original implementation of these registers, as described in *Background to the System register interface* on page G1-5770. In Armv8, there is no particular significance to the naming of these parameters, and no functional distinction between the `op`*n* parameters and the `CR`*x* parameters.

Principles of the System instruction class encoding describes some general properties of these encodings. *System instruction class encoding overview* on page C5-377 then describes the top-level encoding of these instructions, and the following sections then describe the next level of the encoding hierarchy of System instructions and Special-purpose registers:

- *op0==0b00, architectural hints, barriers and CLREX, and PSTATE access* on page C5-378.
- *op0==0b01, cache maintenance, TLB maintenance, and address translation instructions* on page C5-380.
- *op0==0b11, Moves to and from Special-purpose registers* on page C5-387.

For the description of the next level of encoding hierarchy of System registers, see:

- *op0==0b10, Moves to and from debug and trace System registers* on page D12-2851.
- *op0==0b11, Moves to and from non-debug System registers, Special-purpose registers* on page D12-2853.
- *Reserved encodings for IMPLEMENTATION DEFINED registers* on page D12-2866.

C5.1.1 Principles of the System instruction class encoding

In Armv8, an encoding in the System instruction space is identified by a set of arguments, {`op0`, `op1`, `CRn`, `CRm`, `op2`}. These form an encoding hierarchy, where:

`op0` Defines the top-level division of the encoding space, see *System instruction class encoding overview* on page C5-377.

`op1` Identifies the lowest Exception level at which the encoding is accessible, as follows:

Accessible at EL0 `op1` has the value 3.

Accessible at EL1 `op1` has the value 0, 1, or 2. The value is the same as the `op1` value used to access the equivalent AArch32 register.

Accessible at EL2 `op1` has the value 4.

Accessible at EL3 `op1` has the value 6.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Arm strongly recommends that implementers adopt this use of op1 when using the IMPLEMENTATION DEFINED regions of the encoding space described in *Reserved encodings for IMPLEMENTATION DEFINED registers* on page D12-2866.

C5.1.2 System instruction class encoding overview

The encoding of the System instruction class describes each instruction as being either:

- A transfer to a System register. This is a System instruction with the semantics of a write.
- A transfer from a System register. This is a System instruction with the semantics of a read.

A System instruction that initiates an operation operates as if it was making a transfer to a register.

In the AArch64 instruction set, the decode structure for the System instruction class is:

31 30 29 28	27 26 25 24	23 22	21	20	19 18	16	15	12	11	8	7	5	4	0
1 1 0 1	0 1 0 1	0 0	L	op0	op1		CRn		CRm		op2		Rt	

The value of L indicates the transfer direction:

0 Transfer to System register.

1 Transfer from System register.

The op0 field is the top level encoding of the System instruction type. Its possible values are:

0b00 These encodings provide:

- Instructions with an immediate field for accessing PSTATE, the current PE state.
- The architectural hint instructions.
- Barriers and the CLREX instruction.

For more information about these encodings, see *op0==0b00, architectural hints, barriers and CLREX, and PSTATE access* on page C5-378.

0b01 These encodings provide the cache maintenance, TLB maintenance, and address translation instructions.

Note

These are equivalent to operations in the AArch32 (coproc==0b1111) encoding space.

For more information, see *op0==0b01, cache maintenance, TLB maintenance, and address translation instructions* on page C5-380.

0b10 These encodings provide moves to and from:

- Legacy AArch32 System registers for execution environments, to provide access to these registers from higher Exception levels that are using AArch64.
- Debug and trace registers.

Note

These are equivalent to the registers in the AArch32 (coproc==0b1110) encoding space.

For more information, see *op0==0b10, Moves to and from debug and trace System registers* on page D12-2851.

0b11 These encodings provide:

- Moves to and from Non-debug System registers. The accessed registers provide system control, and system status information.

  Note

  The accessed registers are equivalent to the registers in the AArch32 (coproc==0b1111) encoding space.

- Access to Special-purpose registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For more information, see *Instructions for accessing Special-purpose registers* on page C5-387 and *Instructions for accessing non-debug System registers* on page D12-2853.

UNDEFINED behaviors

In the System register instruction encoding space, the following principles apply:

- All unallocated encodings are treated as UNDEFINED.
- All encodings with `L==1` and `op0==0b0x` are UNDEFINED, except for encodings in the area reserved for IMPLEMENTATION DEFINED use, see *Reserved encodings for IMPLEMENTATION DEFINED registers* on page D12-2866.

For registers and operations that are accessible from a particular Exception level, any attempt to access those registers from a lower Exception level is UNDEFINED.

If a particular Exception level:

- Defines a register to be RO, then any attempt to write to that register, at that Exception level, is UNDEFINED. This means that any access to that register with `L==0` is UNDEFINED.
- Defines a register to be WO, then any attempt to read from that register, at that Exception level, is UNDEFINED. This means that any access to that register with `L==1` is UNDEFINED.

For IMPLEMENTATION DEFINED encoding spaces, the treatment of the encodings is IMPLEMENTATION DEFINED, but see the recommendation in *Principles of the System instruction class encoding* on page C5-376.

C5.1.3 `op0==0b00`, architectural hints, barriers and `CLREX`, and PSTATE access

The different groups of System register instructions with `op0==0b00`:

- Are identified by the value of `CRn`.
- Are always encoded with a value of `0b11111` in the `Rt` field.

The encoding of these instructions is:

31	30	29	28	27	26	25	24	23	22	21	20	19	18 … 16	15 … 12	11 … 8	7 … 5	4 … 0
1	1	0	1	0	1	0	1	0	0	L	0	0	op1	CRn	CRm	op2	1 1 1 1 1
											op0						Rt

The encoding of the `CRn` field is as follows:

0b0010 See *Architectural hint instructions*.

0b0011 See *Barriers and CLREX* on page C5-379.

0b0100 See *Instructions for accessing the PSTATE fields* on page C5-380.

Architectural hint instructions

Within the `op0==0b00` encodings, the architectural hint instructions are identified by `CRn` having the value `0b0010`. The encoding of these instructions is:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11 … 5	4 … 0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	Op<6:0>	1 1 1 1 1
											op0		op1			CRn				CRm op2	Rt

The value of `op<6:0>`, formed by concatenating the `CRm` and `op2` fields, determines the hint instruction as follows:

`0b0000000` `NOP` instruction.

`0b0000001` `YIELD` instruction.

`0b0000010` `WFE` instruction.

`0b0000011` `WFI` instruction.

`0b0000100` `SEV` instruction.

`0b0000101` `SEVL` instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b0000110` `DGH` instruction.

`0b0000111` `XPACD, XPACI, XPACLRI` instruction.

`0b0001000` `PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA` instruction, PACIA1716 variant.

`0b0001010` `PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB` instruction, PACIB1716 variant.

`0b0001100` `AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA` instruction, AUTIA1716 variant.

`0b0001110` `AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB` instruction, AUTIB1716 variant.

`0b0010000` `ESB` instruction.

`0b0010001` `PSB CSYNC` instruction.

`0b0010010` `TSB CSYNC` instruction.

`0b0010100` `CSDB` instruction.

`0b0011000` `PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA` instruction, PACIAZ variant.

`0b0011001` `PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA` instruction, PACIASP variant.

`0b0011010` `PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB` instruction, PACIBZ variant.

`0b0011011` `PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB` instruction, PACIBSP variant.

`0b0011100` `AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA` instruction, AUTIAZ variant.

`0b0011101` `AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA` instruction, AUTIASP variant.

`0b0011110` `AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB` instruction, AUTIBZ variant.

`0b0011111` `AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB` instruction, AUTIBSP variant.

`0b0100xx0` `BTI` instruction.

These instructions are described in Chapter C6 *A64 Base Instruction Descriptions*.

Note

- Instruction encodings with bits[4:0] not set to `0b11111` are UNDEFINED.
- The operation of the A64 instructions for architectural hints are identical to the corresponding A32 and T32 instructions.

For more information about:

- The `WFE`, `WFI`, `SEV`, and `SEVL` instructions, see *Mechanisms for entering a low-power state* on page D1-2397.
- The `YIELD` instruction, see *Software control features and EL0* on page B1-112.

Barriers and `CLREX`

Within the `op0==0b00` encodings, the barriers and `CLREX` instructions are identified by `CRn` having the value `0b0011`. The encoding of these instructions is:

31 30 29 28	27 26 25 24	23 22 21	20	19	18	16	15	12	11	8	7	5	4	0
1 1 0 1	0 1 0 1	0 0 0	0	0	0 1	1	0 0 1	1	CRm		op2		1	1 1 1 1
			op0		op1		CRn							Rt

The value of `op2` determines the instruction, as follows.

`0b010` `CLREX` instruction.

`0b100` `DSB` instruction.

`0b101` `DMB` instruction.

`0b110` `ISB` instruction.

`0b000`, `0b001`, `0b011`, `0b111`
UNDEFINED.

These instructions are described in Chapter C6 *A64 Base Instruction Descriptions*.

Note

- Instruction encodings with bits[4:0] not set to `0b11111` are UNDEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The operation of the A64 instructions for barriers and `CLREX` are identical to the corresponding A32 and T32 instructions.

For more information about:

- The barrier instructions, see *Memory barriers* on page B2-135.
- The `CLREX` instruction, see *Synchronization and semaphores* on page B2-167.

Instructions for accessing the PSTATE fields

Within the `op0==0b00` encodings, the instructions that can be used to modify PSTATE fields directly are identified by `CRn` having the value `0b0100`. The encoding of these instructions is:

31	30	29	28	27	26	25	24	23	22	21	20	19	18 … 16	15	14	13	12	11 … 8	7 … 5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	op1	0	1	0	0	Imm4	op2	1	1	1	1	1
											op0			CRn				CRm		Rt				

These instructions are:

```
CFINV                 ; Inverts the value of PSTATE.C
MSR DAIFSet, #Imm4    ; Used to set any or all of DAIF to 1
MSR DAIFClr, #Imm4    ; Used to clear any or all of DAIF to 0
MSR SPSel, #Imm4      ; Used to select the Stack Pointer, between SP_EL0 and SP_ELx
MSR UAO, #Imm4        ; Used to set the value of PSTATE.UAO
MSR PAN, #Imm4        ; Used to set the value of PSTATE.PAN
```

The value of `op2` selects the instruction form, which defines the constraints on the values of the `op1` and `Imm4` arguments, as follows:

`op2==0b000` Selects the `CFINV` instruction.

`op2==0b011` Selects the `MSR UAO` instruction.

`op2==0b100` Selects the `MSR PAN` instruction.

`op2==0b101` Selects the `MSR SPSel` instruction.

`op2==0b110` Selects the `MSR DAIFSet` instruction, that sets the specified PSTATE.{D, A, I, F} bits to 1.

`op2==0b111` Selects the `MSR DAIFClr` instruction, that clears the specified PSTATE.{D, A, I, F} bits to 0.

All other combinations of `op1` and `op2` are reserved, and the corresponding instructions are UNDEFINED.

Note

For PSTATE updates, instruction encodings with bits[4:0] not set to `0b11111` are UNDEFINED.

Writes to PSTATE.{PAN, D, A, I, F} occur in program order without the need for additional synchronization. Changing PSTATE.SPSel to use SP_EL0 synchronizes any updates to SP_EL0 that have been written by an `MSR` to SP_EL0, without the need for additional synchronization.

C5.1.4 `op0==0b01`, cache maintenance, TLB maintenance, and address translation instructions

The System instructions are encoded with `op0==0b01`. The different groups of System instructions are identified by the values of `CRn` and `CRm`, except that some of this encoding space is reserved for IMPLEMENTATION DEFINED functionality. The encoding of these instructions is:

31	30	29	28	27	26	25	24	23	22	21	20	19	18 … 16	15 … 12	11 … 8	7 … 5	4 … 0
1	1	0	1	0	1	0	1	0	0	0	0	1	op1	CRn	CRm	op2	Xt
											op0						

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The grouping of these instructions depending on the `CRn` and `CRm` fields is as follows:

`CRn`==**7** The instruction group is determined by the value of `CRm`, as follows:

- `CRm`==**{1, 5}** Instruction cache maintenance instructions.
  See *Cache maintenance instructions, and data cache zero operation*.
- `CRm`==**3** Prediction restriction instructions.
  See *Prediction restriction instructions* on page C5-382.
- `CRm`==**4** Data cache zero operation.
  See *Cache maintenance instructions, and data cache zero operation*.
- `CRm`==**{6, 10, 11, 12, 14}** Data cache maintenance instructions.
  See *Cache maintenance instructions, and data cache zero operation*.
- `CRm`==**8** See *Address translation instructions* on page C5-383.

`CRn`==**8** See *TLB maintenance instructions* on page C5-383.

`CRn`==**{11, 15}** See *Reserved encoding space for IMPLEMENTATION DEFINED instructions* on page C5-386.

Cache maintenance instructions, and data cache zero operation

Table C5-1 lists the Cache maintenance instructions and their encodings. Instructions that take an argument include `Xt` in the instruction syntax. For instructions that do not take an argument, the Xt field is encoded as `0b11111`. For these instructions, if the Xt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Xt field is set to `0b11111`.

Table C5-1 Cache maintenance instructions and data cache zero operation

Instruction	Access instruction encoding					Notes
	op0	op1	CRn	CRm	op2	
Instruction cache maintenance instructions						
IC IALLUIS	1	0	7	1	0	Accessible from EL1 or higher.
IC IALLU				5	0	
IC IVAU, `Xt`		3	7	5	1	When SCTLR_EL1.UCI == 1, accessible from EL0 or higher. Otherwise, accessible from EL1 or higher.
Data cache maintenance instructions						

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C5-1 Cache maintenance instructions and data cache zero operation (continued)

Instruction	Access instruction encoding					Notes
	op0	op1	CRn	CRm	op2	
DC IVAC, Xt	1	0	7	6	1	Accessible from EL1 or higher.
DC ISW, Xt					2	
DC CSW, Xt				10	2	
DC CISW, Xt				14	2	
DC CVAC, Xt		3	7	10	1	When SCTLR_EL1.UCI == 1, accessible from EL0 or higher. Otherwise, accessible from EL1 or higher.
DC CVAU, Xt				11	1	
DC CVAP, Xt				12	1	
DC CIVAC, Xt				14	1	
Data cache zero operation						
DC ZVA, Xt	1	3	7	4	1	When SCTLR_EL1.DZE == 1, accessible from EL0 or higher. Otherwise, accessible from EL1 or higher.

For more information about these instructions, see *About cache maintenance in AArch64 state* on page D4-2506 and *A64 Cache maintenance instructions* on page D4-2510.

Prediction restriction instructions

Table C5-2 lists the Prediction restriction instructions and their encodings. Instructions that take an argument include Xt in the instruction syntax.

Table C5-2 Prediction restriction instructions

Instruction	Prediction restriction encoding					Notes
	op0	op1	CRn	CRm	op2	
CFP RCTX, Xt	1	3	7	3	4	When ARMv8.0-PredInv is implemented, accessible from EL0 or higher.
CPP RCTX, Xt					5	
DVP RCTX, Xt					7	

For more information about these instructions, see *Execution and data prediction restriction System instructions* on page D4-2522.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Address translation instructions

Table C5-3 lists the Address translation instructions and their encodings. The syntax of the instructions includes `Xt`, that provides the address to be translated.

Table C5-3 Address translation instructions

Instruction	Access instruction encoding					Notes
	op0	op1	CRn	CRm	op2	
AT S1E1R, `Xt`	1	0	7	8	0	Accessible from EL1 or higher.
AT S1E1W, `Xt`					1	
AT S1E0R, `Xt`					2	
AT S1E0W, `Xt`					3	
AT S1E1RP, `Xt`				9	0	
AT S1E1WP, `Xt`					1	
AT S1E2R, `Xt`		4	7	8	0	Accessible from EL2 or higher.
AT S1E2W, `Xt`					1	
AT S12E1R, `Xt`					4	
AT S12E1W, `Xt`					5	
AT S12E0R, `Xt`					6	
AT S12E0W, `Xt`					7	
AT S1E3R, `Xt`		6	7	8	0	Accessible only from EL3.
AT S1E3W, `Xt`					1	

For more information about these instructions, see *Address translation instructions* on page D5-2589.

TLB maintenance instructions

Table C5-4 on page C5-384 lists the TLB maintenance instructions and their encodings. Instructions that take an argument include `Xt` in the instruction syntax. For instructions that do not take an argument, the Xt field is encoded as `0b11111`. For these instructions, if the Xt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The instruction behaves as if the Xt field is set to `0b11111`.

Table C5-4 TLB maintenance instructions

Instruction	Access instruction encoding					Notes
	op0	op1	CRn	CRm	op2	
TLBI VMALLE1OS, `Xt`	1	0	8	1	0	Accessible from EL1 or higher.
TLBI VAE1OS, `Xt`					1	
TLBI ASIDE1OS, `Xt`					2	
TLBI VAAE1OS, `Xt`					3	
TLBI VALE1OS, `Xt`					5	
TLBI VAALE1OS, `Xt`					7	
TLBI RVAE1IS, `Xt`				2	1	
TLBI RVAAE1IS, `Xt`					3	
TLBI RVALE1IS, `Xt`					5	
TLBI RVAALE1IS, `Xt`					7	
TLBI VMALLE1IS				3	0	
TLBI VAE1IS, `Xt`					1	
TLBI ASIDE1IS, `Xt`					2	
TLBI VAAE1IS, `Xt`					3	
TLBI VALE1IS, `Xt`					5	
TLBI VAALE1IS, `Xt`					7	
TLBI RVAE1OS, `Xt`				5	1	
TLBI RVAAE1OS, `Xt`					3	
TLBI RVALE1OS, `Xt`					5	
TLBI RVAALE1OS, `Xt`					7	
TLBI RVAE1, `Xt`				6	1	
TLBI RVAAE1, `Xt`					3	
TLBI RVALE1, `Xt`					5	
TLBI RVAALE1, `Xt`					7	
TLBI VMALLE1				7	0	
TLBI VAE1, `Xt`					1	

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C5-4 TLB maintenance instructions (continued)

Instruction	Access instruction encoding					Notes
	op0	op1	CRn	CRm	op2	
TLBI ASIDE1, Xt	1	0	8	7	2	Accessible from EL1 or higher.
TLBI VAAE1, Xt					3	
TLBI VALE1, Xt					5	
TLBI VAALE1, Xt					7	
TLBI IPAS2E1IS, Xt		4	8	0	1	Accessible from EL2 or higher.
TLBI RIPAS2E1IS, Xt					2	
TLBI IPAS2LE1IS, Xt					5	
TLBI RIPAS2LE1IS, Xt					6	
TLBI ALLE2OS				1	0	
TLBI VAE2OS, Xt					1	
TLBI ALLE1OS					4	
TLBI VALE2OS, Xt					5	
TLBI VMALLS12E1OS					6	
TLBI RVAE2IS, Xt				2	1	
TLBI RVALE2IS, Xt					5	
TLBI ALLE2IS				3	0	
TLBI VAE2IS, Xt					1	
TLBI ALLE1IS					4	
TLBI VALE2IS, Xt					5	
TLBI VMALLS12E1IS					6	
TLBI IPAS2E1OS, Xt				4	0	
TLBI IPAS2E1, Xt					1	
TLBI RIPAS2E1, Xt					2	
TLBI RIPAS2E1OS, Xt					3	
TLBI IPAS2LE1OS, Xt					4	
TLBI IPAS2LE1, Xt					5	
TLBI RIPAS2LE1, Xt					6	
TLBI RIPAS2LE1OS, Xt					7	
TLBI RVAE2OS, Xt				5	1	
TLBI RVALE2OS, Xt					5	
TLBI RVAE2, Xt				6	1	

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C5-4 TLB maintenance instructions (continued)

Instruction	Access instruction encoding					Notes
	op0	op1	CRn	CRm	op2	
TLBI RVALE2, Xt	1	4	8	6	5	Accessible from EL2 or higher.
TLBI ALLE2				7	0	
TLBI VAE2, Xt					1	
TLBI ALLE1					4	
TLBI VALE2, Xt					5	
TLBI VMALLS12E1					6	
TLBI ALLE3OS		6	8	1	0	Accessible only from EL3.
TLBI VAE3OS, Xt					1	
TLBI VALE3OS, Xt					5	
TLBI RVAE3IS, Xt				2	1	
TLBI RVALE3IS, Xt					5	
TLBI ALLE3IS				3	0	
TLBI VAE3IS, Xt					1	
TLBI VALE3IS, Xt					5	
TLBI RVAE3OS, Xt				5	1	
TLBI RVALE3OS, Xt					5	
TLBI RVAE3, Xt				6	1	
TLBI RVALE3, Xt					5	
TLBI ALLE3				7	0	
TLBI VAE3, Xt					1	
TLBI VALE3, Xt					5	

For more information about these instructions, see *TLB maintenance instructions* on page D5-2671.

Reserved encoding space for IMPLEMENTATION DEFINED instructions

The A64 instruction set reserves the following encoding space for IMPLEMENTATION DEFINED instructions:

31 30 29 28	27 26 25 24	23 22	21	20 19	18 16	15 12	11 8	7 5	4 0
1 1 0 1	0 1 0 1	0 0	L	0 1	op1	1 x 1 1	CRm	op2	Rt
				op0		CRn			

The value of L defines the use of Rt as follows:

0 Rt is an argument supplied to the instruction.

1 Rt is a result returned by the instruction.

IMPLEMENTATION DEFINED instructions in this encoding space are accessed using the SYS and SYSL instructions, see *SYS* on page C6-1362 and *SYSL* on page C6-1364.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

See also *Reserved encodings for IMPLEMENTATION DEFINED registers* on page D12-2866.

C5.1.5 `op0==0b11`, Moves to and from Special-purpose registers

The instructions that move data to and from non-debug System registers are encoded with `op0==0b11`, except that some of this encoding space is reserved for IMPLEMENTATION DEFINED functionality. The encoding of these instructions is:

31	30	29	28	27	26	25	24	23	22	21	20	19	18 … 16	15 … 12	11 … 8	7 … 5	4 … 0
1	1	0	1	0	1	0	1	0	0	L	1	1	op1	CRn	CRm	op2	Rt
											op0						

Instructions for accessing Special-purpose registers

The value of `CRn` provides the next level of decode of these instructions. For Special-purpose registers, the value of CRn is 4.

The A64 instructions for accessing Special-purpose registers are:

```
MSR <Special-purpose register>, Xt    ; Write to Special-purpose register
MRS Xt, <Special-purpose register>    ; Read from Special-purpose register
```

For these accesses, `CRn` has the value 4. The encoding for Special-purpose register accesses is:

31	30	29	28	27	26	25	24	23	22	21	20	19	18 … 16	15	14	13	12	11 … 8	7 … 5	4 … 0
1	1	0	1	0	1	0	1	0	0	L	1	1	op1	0	1	0	0	CRm	op2	Rt
											op0			CRn						

The full list of Special-purpose registers is in Table C5-5. The characteristic of a Special-purpose register is that all direct and indirect reads and writes to the register appear to occur in program order relative to other instructions, without the need for any explicit synchronization.

Table C5-5 lists the encodings for `op1`, `CRm`, and `op2` fields for accesses to the Special-purpose registers in AArch64.

Table C5-5 Special-purpose register accesses

Register	Access instruction encoding					Notes
	op0	op1	CRn	CRm	op2	
SPSR_EL1	3	0	4	0	0	Accessible from EL1 or higher
ELR_EL1					1	
SP_EL0				1	0	Accessible from EL1 or higher. If SP_EL0 is the current stack pointer then the access is UNDEFINED.
SPSel				2	0	Accessible from EL1 or higher.
CurrentEL					2	RO. Accessible from EL1 or higher.
PAN					3	Accessible from EL1 or higher.
UAO					4	
NZCV		3	4	2	0	Accessible from EL0 or higher.
DAIF					1	Configurable whether accesses at EL0 are permitted.
DIT					5	Accessible from EL0 or higher.
SSBS					6	
TCO					7	

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C5-5 Special-purpose register accesses (continued)

Register	Access instruction encoding					Notes
	op0	op1	CRn	CRm	op2	
FPCR	3	3	4	4	0	Accessible from EL0 or higher.
FPSR					1	
DSPSR_EL0				5	0	Accessible only in Debug state, from EL0 or higher.
DLR_EL0					1	
SPSR_EL2		4	4	0	0	Accessible from EL2 or higher.
ELR_EL2					1	
SP_EL1				1	0	
SPSR_irq				3	0	
SPSR_abt					1	
SPSR_und					2	
SPSR_fiq					3	
*_EL12		5	4	{0-15}	{0-7}	Reserved for EL2 aliases of EL1 Special-purpose registers, see Table D5-48 on page D5-2643.
SPSR_EL3		6	4	0	0	Accessible from EL3 or higher.
ELR_EL3					1	
SP_EL2				1	0	

All direct and indirect reads and writes to Special-purpose registers appear to occur in program order relative to other instructions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.2 Special-purpose registers

This section describes the following Special-purpose registers:

- CurrentEL, that holds PSTATE.EL, and that software can read to determine the current Exception level.
- DAIF, that holds the current PSTATE.{D, A, I, F} interrupt mask bits.
- DIT, that holds the PSTATE.DIT bit.
- ELR_EL1, that holds the address to return to for an exception return from EL1.
- ELR_EL2, that holds the address to return to for an exception return from EL2.
- ELR_EL3, that holds the address to return to for an exception return from EL3.
- FPCR, that provides control of floating-point operation.
- FPSR, that provides floating-point status information.
- NZCV, that holds the PSTATE.{N, Z, C, V} condition flags.
- PAN, that holds the PSTATE.PAN state bit.
- SP_EL0, that holds the stack pointer for EL0.
- SP_EL1, that holds the stack pointer for EL1.
- SP_EL2, that holds the stack pointer for EL2.
- SP_EL3, that holds the stack pointer for EL3.
- SPSel, that holds PSTATE.SP, that at EL1 or higher selects the current SP.
- SPSR_abt, that holds process state on taking an exception to AArch32 Abort mode.
- SPSR_EL1, that holds process state on taking an exception to AArch64 EL1.
- SPSR_EL2, that holds process state on taking an exception to AArch64 EL2.
- SPSR_EL3, that holds process state on taking an exception to AArch64 EL3.
- SPSR_fiq, that holds process state on taking an exception to AArch32 FIQ mode.
- SPSR_irq, that holds process state on taking an exception to AArch32 IRQ mode.
- SPSR_und, that holds process state on taking an exception to AArch32 Undefined mode.
- SSBS, that holds the PSTATE.SSBS bit.
- TCO, that holds the PSTATE.TCO bit.
- UAO, that holds the PSTATE.UAO bit.

The following registers are also Special-purpose registers:

- DLR_EL0, that holds the address to return to for a return from Debug state.
- DSPSR_EL0, that holds process state on entry to Debug state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.2.1 CurrentEL, Current Exception Level

The CurrentEL characteristics are:

Purpose

Holds the current Exception level.

Configurations

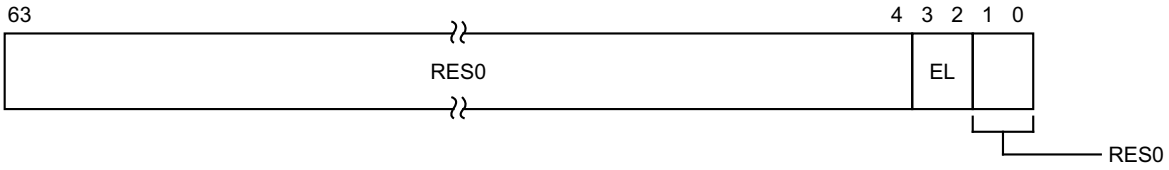
There are no configuration notes.

Attributes

CurrentEL is a 64-bit register.

Field descriptions

The CurrentEL bit assignments are:



Bits [63:4]

Reserved, RES0.

EL, bits [3:2]

Current Exception level. Possible values of this field are:

`0b00` EL0

`0b01` EL1

`0b10` EL2

`0b11` EL3

When the HCR_EL2.NV bit is 1, EL1 read accesses to the CurrentEL register return the value of `0b10` in this field.

This field resets to the highest implemented Exception Level.

Bits [1:0]

Reserved, RES0.

Accessing the CurrentEL

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, CurrentEL

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0010	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        return Zeros(60):'10':Zeros(2);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        return Zeros(60):PSTATE.EL:Zeros(2);
elsif PSTATE.EL == EL2 then
    return Zeros(60):PSTATE.EL:Zeros(2);
elsif PSTATE.EL == EL3 then
    return Zeros(60):PSTATE.EL:Zeros(2);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.2.2 DAIF, Interrupt Mask Bits

The DAIF characteristics are:

Purpose

Allows access to the interrupt mask bits.

Configurations

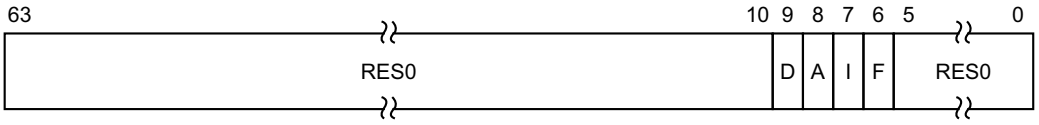
There are no configuration notes.

Attributes

DAIF is a 64-bit register.

Field descriptions

The DAIF bit assignments are:



Bits [63:10]

Reserved, RES0.

D, bit [9]

Process state D mask. The possible values of this bit are:

`0b0` Watchpoint, Breakpoint, and Software Step exceptions targeted at the current Exception level are not masked.

`0b1` Watchpoint, Breakpoint, and Software Step exceptions targeted at the current Exception level are masked.

When the target Exception level of the debug exception is higher than the current Exception level, the exception is not masked by this bit.

This field resets to `1`.

A, bit [8]

SError interrupt mask bit. The possible values of this bit are:

`0b0` Exception not masked.

`0b1` Exception masked.

This field resets to `1`.

I, bit [7]

IRQ mask bit. The possible values of this bit are:

`0b0` Exception not masked.

`0b1` Exception masked.

This field resets to `1`.

F, bit [6]

FIQ mask bit. The possible values of this bit are:

`0b0` Exception not masked.

`0b1` Exception masked.

This field resets to `1`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bits [5:0]

Reserved, RES0.

Accessing the DAIF

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, DAIF

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && ((EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') || SCTLR_EL1.UMA == '0') then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        return Zeros(54):PSTATE.<D,A,I,F>:Zeros(6);
elsif PSTATE.EL == EL1 then
    return Zeros(54):PSTATE.<D,A,I,F>:Zeros(6);
elsif PSTATE.EL == EL2 then
    return Zeros(54):PSTATE.<D,A,I,F>:Zeros(6);
elsif PSTATE.EL == EL3 then
    return Zeros(54):PSTATE.<D,A,I,F>:Zeros(6);
```

MSR DAIF, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && ((EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') || SCTLR_EL1.UMA == '0') then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        PSTATE.<D,A,I,F> = X[t]<9:6>;
elsif PSTATE.EL == EL1 then
    PSTATE.<D,A,I,F> = X[t]<9:6>;
elsif PSTATE.EL == EL2 then
    PSTATE.<D,A,I,F> = X[t]<9:6>;
elsif PSTATE.EL == EL3 then
    PSTATE.<D,A,I,F> = X[t]<9:6>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR DAIFSet, #<imm>

op0	op1	CRn	op2
0b00	0b011	0b0100	0b110

MSR DAIFClr, #<imm>

op0	op1	CRn	op2
0b00	0b011	0b0100	0b111

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.2.3 DIT, Data Independent Timing

The DIT characteristics are:

Purpose

Allows access to the Data Independent Timing bit.

Configurations

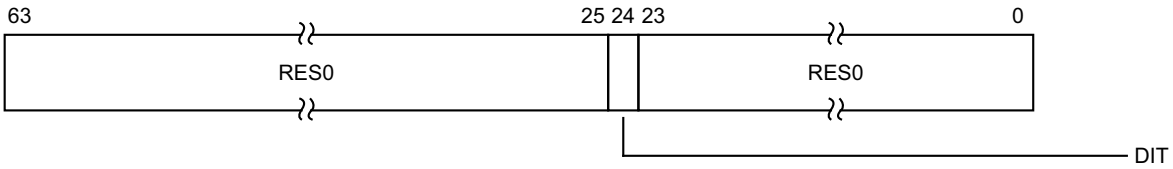
This register is present only when ARMv8.4-DIT is implemented. Otherwise, direct accesses to DIT are UNDEFINED.

Attributes

DIT is a 64-bit register.

Field descriptions

The DIT bit assignments are:



Bits [63:25]

Reserved, RES0.

DIT, bit [24]

Data Independent Timing.

`0b0` The architecture makes no statement about the timing properties of any instructions.

`0b1` The architecture requires that:

- The timing of every load and store instruction is insensitive to the value of the data being loaded or stored.
- For certain data processing instructions, the instruction takes a time which is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- For certain data processing instructions, the response of the instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

The data processing instructions affected by this bit are:

- All cryptographic instructions. These instructions are:
  - `AESD`, `AESE`, `AESIMC`, `AESMC`, `SHA1C`, `SHA1H`, `SHA1M`, `SHA1P`, `SHA1SU0`, `SHA1SU1`, `SHA256H`, `SHA256H2`, `SHA256SU0`, `SHA256SU1`, `SHA512H`, `SHA512H2`, `SHA512SU0`, `SHA512SU1`, `EOR3`, `RAX1`, `XAR`, `BCAX`, `SM3SS1`, `SM3TT1A`, `SM3TT1B`, `SM3TT2A`, `SM3TT2B`, `SM3PARTW1`, `SM3PARTW2`, `SM4E`, and `SM4EKEY`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- A subset of those instructions which use the general-purpose register file. These instructions are:
  - ADC, ADCS, ADD, ADDS, AND, ANDS, ASR, ASRV, BFC, BFI, BFM, BFXIL, BIC, BICS, CCMN, CCMP, CFINV, CINC, CINV, CLS, CLZ, CMN, CMP, CNEG, CSEL, CSET, CSETM, CSINC, CSINV, CSNEG, EON, EOR, EXTR, LSL, LSLV, LSR, LSRV, MADD, MNEG, MOV, MOVK, MOVN, MOVZ, MSUB, MUL, MVN, NEG, NEGS, NGC, NGCS, NOP, ORN, ORR, RBIT, RET, REV, REV16, REV32, REV64, RMIF, ROR, RORV, SBC, SBCS, SBFIZ, SBFM, SBFX, SETF8, SETF16, SMADDL, SMNEGL, SMSUBL, SMULH, SMULL, SUB, SUBS, SXTB, SXTH, SXTW, TST, UBFIZ, UBFM, UBFX, UMADDL, UMNEGL, UMSUBL, UMULH, UMULL, UXTB, and UXTH.
- A subset of those instructions which use the SIMD&FP register file. These instructions are:
  - ABS, ADD, ADDHN, ADDHN2, ADDP, ADDV, AND, BIC, BIF, BIT, BSL, CLS, CLZ, CMEQ, CMGE, CMGT, CMHI, CMHS, CMLE, CMLT, CMTST, CNT, CRC32B, CRC32H, CRC32W, CRC32X, CRC32CB, CRC32CH, CRC32CW, CRC32CX, DUP, EOR, EXT, FCSEL, INS, MLA, MLS, MOV, MOVI, MUL, MVN, MVNI, NEG, NOT, ORN, ORR, PMUL, PMULL, PMULL2, RADDHN, RADDHN2, RBIT, REV16, REV32, RSHRN, RSHRN2, RSUBHN, RSUBHN2, SABA, SABD, SABAL, SABAL2, SABDL, SABDL2, SADALP, SADDL, SADDL2, SADDLP, SADDLV, SADDW, SADDW2, SHADD, SHL, SHLL, SHLL2, SHRN, SHRN2, SHSUB, SLI, SMAX, SMAXP, SMAXV, SMIN, SMINP, SMINV, SMLAL, SMLAL2, SMLSL, SMLSL2, SMOV, SMULL, SMULL2, SRI, SSHL, SSHLL, SSHLL2, SSHR, SSRA, SSUBL, SSUBL2, SSUBW, SSUBW2, SUB, SUBHN, SUBHN2, SXTL, SXTL2, TBL, TBX, TRN1, TRN2, UABA, UABAL, UABAL2, UABD, UABDL, UABDL2, UADALP, UADDL, UADDL2, UADDLP, UADDLV, UADDW, UADDW2, UHADD, UHSUB, UMAX, UMAXP, UMAXV, UMIN, UMINP, UMINV, UMLAL, UMLAL2, UMLSL, UMOV, UMLSL2, UMULL, UMULL2, USHL, USHLL, USHLL2, USHR, USRA, USUBL, USUBL2, USUBW, USUBW2, UXTL, UXTL2, UZP1, UZP2, XTN, XTN2, ZIP1, and ZIP2.

Note

The architecture makes no statement about the timing properties when the PSTATE.DIT bit is not set. However, it is likely that many of these instructions have timing that is invariant of the data in many situations.

In particular, Arm strongly recommends that the Armv8.3 pointer authentication instructions do not have their timing dependent on the key value used in the pointer authentication in all cases, regardless of the PSTATE.DIT bit.

This field resets to 0.

Bits [23:0]

Reserved, RES0.

Accessing the DIT

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, DIT

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b101

```
if PSTATE.EL == EL0 then
    return Zeros(39):PSTATE.DIT:Zeros(24);
elsif PSTATE.EL == EL1 then
    return Zeros(39):PSTATE.DIT:Zeros(24);
elsif PSTATE.EL == EL2 then
    return Zeros(39):PSTATE.DIT:Zeros(24);
elsif PSTATE.EL == EL3 then
    return Zeros(39):PSTATE.DIT:Zeros(24);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR DIT, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b101

```
if PSTATE.EL == EL0 then
    PSTATE.DIT = X[t]<24>;
elsif PSTATE.EL == EL1 then
    PSTATE.DIT = X[t]<24>;
elsif PSTATE.EL == EL2 then
    PSTATE.DIT = X[t]<24>;
elsif PSTATE.EL == EL3 then
    PSTATE.DIT = X[t]<24>;
```

MSR DIT, #<imm>

op0	op1	CRn	op2
0b00	0b011	0b0100	0b010

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.2.4 ELR_EL1, Exception Link Register (EL1)

The ELR_EL1 characteristics are:

Purpose

When taking an exception to EL1, holds the address to return to.

Configurations

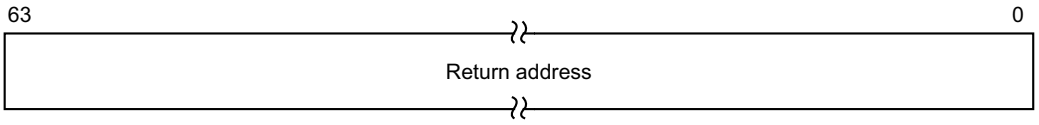
There are no configuration notes.

Attributes

ELR_EL1 is a 64-bit register.

Field descriptions

The ELR_EL1 bit assignments are:



Bits [63:0]

Return address.

An exception return from EL1 using AArch64 makes ELR_EL1 become UNKNOWN.

This field resets to an architecturally UNKNOWN value.

Accessing the ELR_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic ELR_EL1 or ELR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ELR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1> == '01' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x230];
    else
        return ELR_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return ELR_EL2;
    else
        return ELR_EL1;
elsif PSTATE.EL == EL3 then
    return ELR_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR ELR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1> == '01' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x230] = X[t];
    else
        ELR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        ELR_EL2 = X[t];
    else
        ELR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    ELR_EL1 = X[t];
```

MRS <Xt>, ELR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0100	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        return NVMem[0x230];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return ELR_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        return ELR_EL1;
    else
        UNDEFINED;
```

MSR ELR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0100	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        NVMem[0x230] = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        ELR_EL1 = X[t];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        ELR_EL1 = X[t];
    else
        UNDEFINED;
```

MRS <Xt>, ELR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        return ELR_EL1;
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return ELR_EL2;
elsif PSTATE.EL == EL3 then
    return ELR_EL2;
```

MSR ELR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        ELR_EL1 = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    ELR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    ELR_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.2.5 ELR_EL2, Exception Link Register (EL2)

The ELR_EL2 characteristics are:

Purpose

When taking an exception to EL2, holds the address to return to.

Configurations

AArch64 System register ELR_EL2[31:0] is architecturally mapped to AArch32 System register ELR_hyp[31:0].

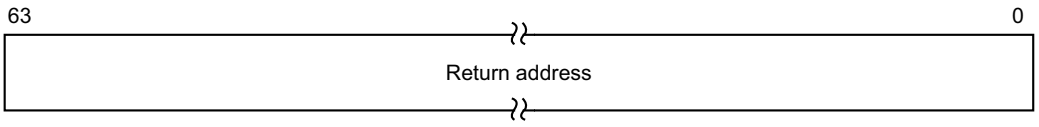
This register has no effect if EL2 is not enabled in the current Security state.

Attributes

ELR_EL2 is a 64-bit register.

Field descriptions

The ELR_EL2 bit assignments are:



Bits [63:0]

Return address.

An exception return from EL2 using AArch64 makes ELR_EL2 become UNKNOWN.

When EL2 is in AArch32 Execution state and an exception is taken from EL0, EL1, or EL2 to EL3 and AArch64 execution, the upper 32-bits of ELR_EL2 are either set to 0 or hold the same value that they did before AArch32 execution. Which option is adopted is determined by an implementation, and might vary dynamically within an implementation. Correspondingly software must regard the value as being an UNKNOWN choice between the two values.

This field resets to an architecturally UNKNOWN value.

Accessing the ELR_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic ELR_EL2 or ELR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ELR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        return ELR_EL1;
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    return ELR_EL2;
elsif PSTATE.EL == EL3 then
    return ELR_EL2;
```

MSR ELR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        ELR_EL1 = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    ELR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    ELR_EL2 = X[t];
```

MRS <Xt>, ELR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1> == '01' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x230];
    else
        return ELR_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return ELR_EL2;
    else
        return ELR_EL1;
elsif PSTATE.EL == EL3 then
    return ELR_EL1;
```

MSR ELR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1> == '01' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x230] = X[t];
    else
        ELR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        ELR_EL2 = X[t];
    else
        ELR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    ELR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.2.6 ELR_EL3, Exception Link Register (EL3)

The ELR_EL3 characteristics are:

Purpose

When taking an exception to EL3, holds the address to return to.

Configurations

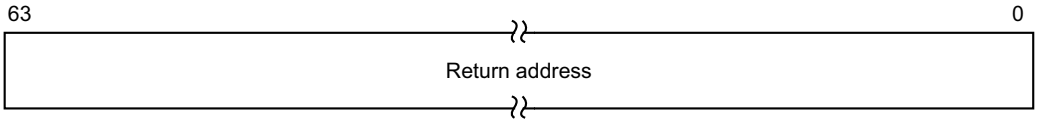
This register is present only when EL3 is implemented. Otherwise, direct accesses to ELR_EL3 are UNDEFINED.

Attributes

ELR_EL3 is a 64-bit register.

Field descriptions

The ELR_EL3 bit assignments are:



Bits [63:0]

Return address.

An exception return from EL3 using AArch64 makes ELR_EL3 become UNKNOWN.

This field resets to an architecturally UNKNOWN value.

Accessing the ELR_EL3

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ELR_EL3

op0	op1	CRn	CRm	op2
0b11	0b110	0b0100	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    return ELR_EL3;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR ELR_EL3, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b110	0b0100	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    ELR_EL3 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.2.7 FPCR, Floating-point Control Register

The FPCR characteristics are:

Purpose

Controls floating-point behavior.

Configurations

The named fields in this register map to the equivalent fields in the AArch32 FPSCR.

It is IMPLEMENTATION DEFINED whether the Len and Stride fields can be programmed to non-zero values, which will cause some AArch32 floating-point instruction encodings to be UNDEFINED, or whether these fields are RAZ.

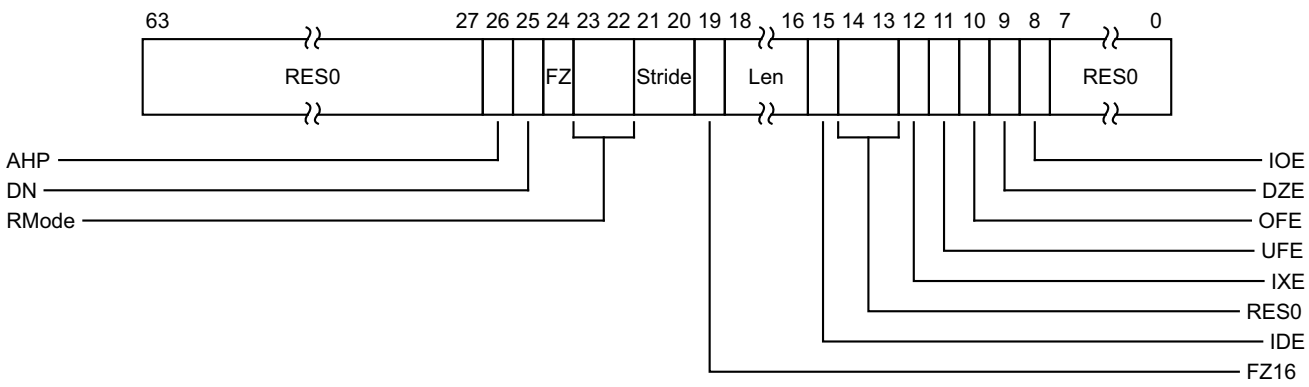
RW fields in this register reset to architecturally UNKNOWN values.

Attributes

FPCR is a 64-bit register.

Field descriptions

The FPCR bit assignments are:



Bits [63:27]

Reserved, RES0.

AHP, bit [26]

Alternative half-precision control bit:

`0b0` IEEE half-precision format selected.

`0b1` Alternative half-precision format selected.

This bit is only used for conversions between half-precision floating-point and other floating-point formats.

The data-processing instructions added as part of the ARMv8.2-FP16 extension always use the IEEE half-precision format, and ignore the value of this bit.

This field resets to an architecturally UNKNOWN value.

DN, bit [25]

Default NaN mode control bit:

`0b0` NaN operands propagate through to the output of a floating-point operation.

`0b1` Any operation involving one or more NaNs returns the Default NaN.

The value of this bit controls both scalar and Advanced SIMD floating-point arithmetic.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

FZ, bit [24]

Flush-to-zero mode control bit:

`0b0` Flush-to-zero mode disabled. Behavior of the floating-point system is fully compliant with the IEEE 754 standard.

`0b1` Flush-to-zero mode enabled.

The value of this bit controls both scalar and Advanced SIMD floating-point arithmetic.

This bit has no effect on half-precision calculations.

This field resets to an architecturally UNKNOWN value.

RMode, bits [23:22]

Rounding Mode control field. The encoding of this field is:

`0b00` Round to Nearest (RN) mode.

`0b01` Round towards Plus Infinity (RP) mode.

`0b10` Round towards Minus Infinity (RM) mode.

`0b11` Round towards Zero (RZ) mode.

The specified rounding mode is used by both scalar and Advanced SIMD floating-point instructions.

This field resets to an architecturally UNKNOWN value.

Stride, bits [21:20]

This field has no function in AArch64 state, and non-zero values are ignored during execution in AArch64 state. It is included only for context saving and restoration of the AArch32 FPSCR.Stride field.

This field resets to an architecturally UNKNOWN value.

FZ16, bit [19]

When ARMv8.2-FP16 is implemented:

Flush-to-zero mode control bit on half-precision data-processing instructions:

`0b0` Flush-to-zero mode disabled. Behavior of the floating-point system is fully compliant with the IEEE 754 standard.

`0b1` Flush-to-zero mode enabled.

The value of this bit applies to both scalar and Advanced SIMD floating-point half-precision calculations. A half-precision floating-point number that is flushed to zero as a result of the value of the FZ16 bit does not generate an Input Denormal exception.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Len, bits [18:16]

This field has no function in AArch64 state, and non-zero values are ignored during execution in AArch64 state. It is included only for context saving and restoration of the AArch32 FPSCR.Len field.

This field resets to an architecturally UNKNOWN value.

IDE, bit [15]

Input Denormal floating-point exception trap enable. Possible values are:

`0b0` Untrapped exception handling selected. If the floating-point exception occurs then the FPSR.IDC bit is set to 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.IDC bit. The trap handling software can decide whether to set the FPSR.IDC bit to 1.

The value of this bit controls both scalar and Advanced SIMD floating-point arithmetic.

If the implementation does not support this exception, this bit is RAZ/WI.

This field resets to an architecturally UNKNOWN value.

Bits [14:13]

Reserved, RES0.

IXE, bit [12]

Inexact floating-point exception trap enable. Possible values are:

`0b0` Untrapped exception handling selected. If the floating-point exception occurs then the FPSR.IXC bit is set to 1.

`0b1` Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.IXC bit. The trap handling software can decide whether to set the FPSR.IXC bit to 1.

The value of this bit controls both scalar and Advanced SIMD floating-point arithmetic.

If the implementation does not support this exception, this bit is RAZ/WI.

This field resets to an architecturally UNKNOWN value.

UFE, bit [11]

Underflow floating-point exception trap enable. Possible values are:

`0b0` Untrapped exception handling selected. If the floating-point exception occurs then the FPSR.UFC bit is set to 1.

`0b1` Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.UFC bit. The trap handling software can decide whether to set the FPSR.UFC bit to 1.

The value of this bit controls both scalar and Advanced SIMD floating-point arithmetic.

If the implementation does not support this exception, this bit is RAZ/WI.

This field resets to an architecturally UNKNOWN value.

OFE, bit [10]

Overflow floating-point exception trap enable. Possible values are:

`0b0` Untrapped exception handling selected. If the floating-point exception occurs then the FPSR.OFC bit is set to 1.

`0b1` Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.OFC bit. The trap handling software can decide whether to set the FPSR.OFC bit to 1.

The value of this bit controls both scalar and Advanced SIMD floating-point arithmetic.

If the implementation does not support this exception, this bit is RAZ/WI.

This field resets to an architecturally UNKNOWN value.

DZE, bit [9]

Divide by Zero floating-point exception trap enable. Possible values are:

`0b0` Untrapped exception handling selected. If the floating-point exception occurs then the FPSR.DZC bit is set to 1.

`0b1` Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.DZC bit. The trap handling software can decide whether to set the FPSR.DZC bit to 1.

The value of this bit controls both scalar and Advanced SIMD floating-point arithmetic.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the implementation does not support this exception, this bit is RAZ/WI.

This field resets to an architecturally UNKNOWN value.

IOE, bit [8]

Invalid Operation floating-point exception trap enable. Possible values are:

`0b0` Untrapped exception handling selected. If the floating-point exception occurs then the FPSR.IOC bit is set to 1.

`0b1` Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.IOC bit. The trap handling software can decide whether to set the FPSR.IOC bit to 1.

The value of this bit controls both scalar and Advanced SIMD floating-point arithmetic.

If the implementation does not support this exception, this bit is RAZ/WI.

This field resets to an architecturally UNKNOWN value.

Bits [7:0]

Reserved, RES0.

Accessing the FPCR

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, FPCR

op0	CRn	op1	op2	CRm
0b11	0b0100	0b011	0b000	0b0100

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.FPEN != '11'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x00);
        else
            AArch64.SystemAccessTrap(EL1, 0x07);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.FPEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
        AArch64.SystemAccessTrap(EL3, 0x07);
    else
        return FPCR;
elsif PSTATE.EL == EL1 then
    if CPACR_EL1.FPEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x07);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
        AArch64.SystemAccessTrap(EL3, 0x07);
    else
        return FPCR;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x07);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
        AArch64.SystemAccessTrap(EL3, 0x07);
    else
        return FPCR;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TFP == '1' then
        AArch64.SystemAccessTrap(EL3, 0x07);
    else
        return FPCR;
```

MSR FPCR, <Xt>

op0	CRn	op1	op2	CRm
0b11	0b0100	0b011	0b000	0b0100

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.FPEN != '11'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x00);
        else
            AArch64.SystemAccessTrap(EL1, 0x07);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.FPEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
        AArch64.SystemAccessTrap(EL3, 0x07);
    else
        FPCR = X[t];
elsif PSTATE.EL == EL1 then
    if CPACR_EL1.FPEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x07);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
        AArch64.SystemAccessTrap(EL3, 0x07);
    else
        FPCR = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
        AArch64.SystemAccessTrap(EL3, 0x07);
    else
        FPCR = X[t];
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TFP == '1' then
        AArch64.SystemAccessTrap(EL3, 0x07);
    else
        FPCR = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.2.8 FPSR, Floating-point Status Register

The FPSR characteristics are:

Purpose

Provides floating-point system status information.

Configurations

The named fields in this register map to the equivalent fields in the AArch32 FPSCR.

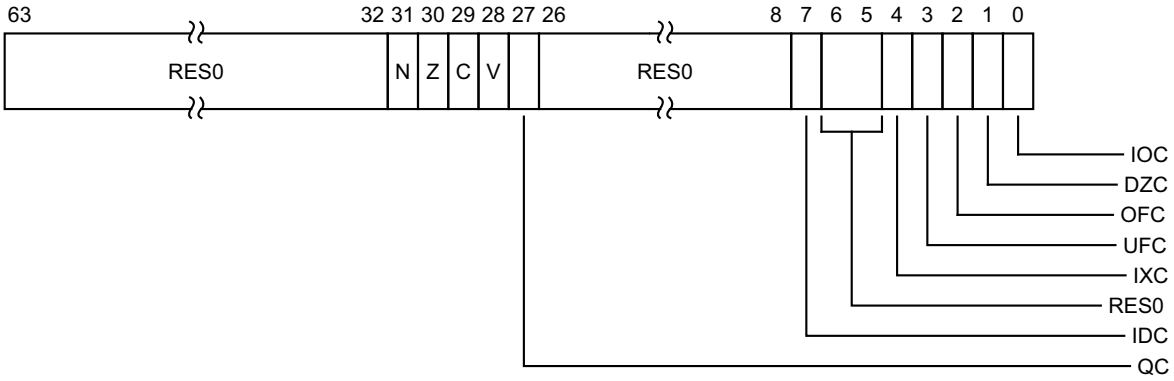
RW fields in this register reset to architecturally UNKNOWN values.

Attributes

FPSR is a 64-bit register.

Field descriptions

The FPSR bit assignments are:



Bits [63:32]

Reserved, RES0.

N, bit [31]

When AArch32 is supported at any Exception level and AArch32 floating-point is implemented:

Negative condition flag for AArch32 floating-point comparison operations.

AArch64 floating-point comparisons set the PSTATE.N flag instead.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Z, bit [30]

When AArch32 is supported at any Exception level and AArch32 floating-point is implemented:

Zero condition flag for AArch32 floating-point comparison operations.

AArch64 floating-point comparisons set the PSTATE.Z flag instead.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C, bit [29]

When AArch32 is supported at any Exception level and AArch32 floating-point is implemented:

Carry condition flag for AArch32 floating-point comparison operations.

AArch64 floating-point comparisons set the PSTATE.C flag instead.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

V, bit [28]

When AArch32 is supported at any Exception level and AArch32 floating-point is implemented:

Overflow condition flag for AArch32 floating-point comparison operations.

AArch64 floating-point comparisons set the PSTATE.V flag instead.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

QC, bit [27]

Cumulative saturation bit, Advanced SIMD only. This bit is set to 1 to indicate that an Advanced SIMD integer operation has saturated since 0 was last written to this bit.

This field resets to an architecturally UNKNOWN value.

Bits [26:8]

Reserved, RES0.

IDC, bit [7]

Input Denormal cumulative floating-point exception bit. This bit is set to 1 to indicate that the Input Denormal floating-point exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.IDE bit. This bit is only set to 1 to indicate a floating-point exception if FPCR.IDE is 0, or if trapping software sets it.

This field resets to an architecturally UNKNOWN value.

Bits [6:5]

Reserved, RES0.

IXC, bit [4]

Inexact cumulative floating-point exception bit. This bit is set to 1 to indicate that the Inexact exception floating-point has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.IXE bit. This bit is only set to 1 to indicate a floating-point exception if FPCR.IXE is 0, or if trapping software sets it.

This field resets to an architecturally UNKNOWN value.

UFC, bit [3]

Underflow cumulative floating-point exception bit. This bit is set to 1 to indicate that the Underflow floating-point exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.UFE bit. This bit is only set to 1 to indicate a floating-point exception if FPCR.UFE is 0, or if trapping software sets it.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

OFC, bit [2]

Overflow cumulative floating-point exception bit. This bit is set to 1 to indicate that the Overflow floating-point exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.OFE bit. This bit is only set to 1 to indicate a floating-point exception if FPCR.OFE is 0, or if trapping software sets it.

This field resets to an architecturally UNKNOWN value.

DZC, bit [1]

Divide by Zero cumulative floating-point exception bit. This bit is set to 1 to indicate that the Divide by Zero floating-point exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.DZE bit. This bit is only set to 1 to indicate a floating-point exception if FPCR.DZE is 0, or if trapping software sets it.

This field resets to an architecturally UNKNOWN value.

IOC, bit [0]

Invalid Operation cumulative floating-point exception bit. This bit is set to 1 to indicate that the Invalid Operation floating-point exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.IOE bit. This bit is only set to 1 to indicate a floating-point exception if FPCR.IOE is 0, or if trapping software sets it.

This field resets to an architecturally UNKNOWN value.

Accessing the FPSR

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, FPSR

op0	CRn	op1	op2	CRm
0b11	0b0100	0b011	0b001	0b0100

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.FPEN != '11'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x00);
        else
            AArch64.SystemAccessTrap(EL1, 0x07);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.FPEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
        AArch64.SystemAccessTrap(EL3, 0x07);
    else
        return FPSR;
elsif PSTATE.EL == EL1 then
    if CPACR_EL1.FPEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x07);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.SystemAccessTrap(EL3, 0x07);
    else
        return FPSR;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
        AArch64.SystemAccessTrap(EL3, 0x07);
    else
        return FPSR;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TFP == '1' then
        AArch64.SystemAccessTrap(EL3, 0x07);
    else
        return FPSR;
```

MSR FPSR, <Xt>

op0	CRn	op1	op2	CRm
0b11	0b0100	0b011	0b001	0b0100

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.FPEN != '11'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x00);
        else
            AArch64.SystemAccessTrap(EL1, 0x07);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.FPEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
        AArch64.SystemAccessTrap(EL3, 0x07);
    else
        FPSR = X[t];
elsif PSTATE.EL == EL1 then
    if CPACR_EL1.FPEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x07);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
        AArch64.SystemAccessTrap(EL3, 0x07);
    else
        FPSR = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
        AArch64.SystemAccessTrap(EL3, 0x07);
    else
        FPSR = X[t];
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TFP == '1' then
        AArch64.SystemAccessTrap(EL3, 0x07);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
else
    FPSR = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.2.9 NZCV, Condition Flags

The NZCV characteristics are:

Purpose

Allows access to the condition flags.

Configurations

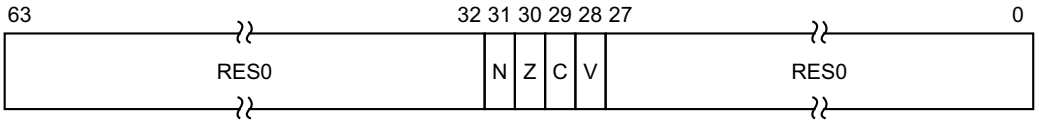
There are no configuration notes.

Attributes

NZCV is a 64-bit register.

Field descriptions

The NZCV bit assignments are:



Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative condition flag. Set to 1 if the result of the last flag-setting instruction was negative.

Z, bit [30]

Zero condition flag. Set to 1 if the result of the last flag-setting instruction was zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.

C, bit [29]

Carry condition flag. Set to 1 if the last flag-setting instruction resulted in a carry condition, for example an unsigned overflow on an addition.

V, bit [28]

Overflow condition flag. Set to 1 if the last flag-setting instruction resulted in an overflow condition, for example a signed overflow on an addition.

Bits [27:0]

Reserved, RES0.

Accessing the NZCV

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, NZCV

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b000

```
if PSTATE.EL == EL0 then
    return Zeros(32):PSTATE.<N,Z,C,V>:Zeros(28);
elsif PSTATE.EL == EL1 then
    return Zeros(32):PSTATE.<N,Z,C,V>:Zeros(28);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL2 then
    return Zeros(32):PSTATE.<N,Z,C,V>:Zeros(28);
elsif PSTATE.EL == EL3 then
    return Zeros(32):PSTATE.<N,Z,C,V>:Zeros(28);
```

MSR NZCV, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b000

```
if PSTATE.EL == EL0 then
    PSTATE.<N,Z,C,V> = X[t]<31:28>;
elsif PSTATE.EL == EL1 then
    PSTATE.<N,Z,C,V> = X[t]<31:28>;
elsif PSTATE.EL == EL2 then
    PSTATE.<N,Z,C,V> = X[t]<31:28>;
elsif PSTATE.EL == EL3 then
    PSTATE.<N,Z,C,V> = X[t]<31:28>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.2.10 PAN, Privileged Access Never

The PAN characteristics are:

Purpose

Allows access to the Privileged Access Never bit.

Configurations

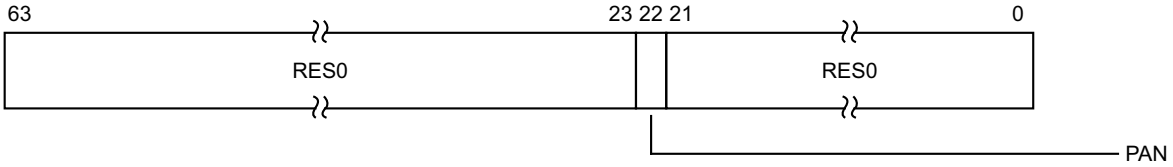
This register is present only when ARMv8.1-PAN is implemented. Otherwise, direct accesses to PAN are UNDEFINED.

Attributes

PAN is a 64-bit register.

Field descriptions

The PAN bit assignments are:



Bits [63:23]

Reserved, RES0.

PAN, bit [22]

Privileged Access Never.

`0b0` Privileged reads and write are not disabled by this mechanism.

`0b1` Disables privileged read and write accesses to addresses accessible at EL0 for an enabled stage 1 translation regime that defines the EL0 permissions.

The value of this bit is usually preserved on taking an exception, except in the following situations:

- When the target of the exception is EL1, and the value of the SCTLR_EL1.SPAN bit is 0, this bit is set to 1.
- When the target of the exception is EL2, HCR_EL2.{E2H, TGE} is {1, 1}, and the value of the SCTLR_EL2.SPAN bit is 0, this bit is set to 1.

Bits [21:0]

Reserved, RES0.

Accessing the PAN

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PAN

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0010	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    return Zeros(41):PSTATE.PAN:Zeros(22);
elsif PSTATE.EL == EL2 then
    return Zeros(41):PSTATE.PAN:Zeros(22);
elsif PSTATE.EL == EL3 then
    return Zeros(41):PSTATE.PAN:Zeros(22);
```

MSR PAN, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0010	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    PSTATE.PAN = X[t]<22>;
elsif PSTATE.EL == EL2 then
    PSTATE.PAN = X[t]<22>;
elsif PSTATE.EL == EL3 then
    PSTATE.PAN = X[t]<22>;
```

MSR PAN, #<imm>

op0	op1	CRn	op2
0b00	0b000	0b0100	0b100

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.2.11 SP_EL0, Stack Pointer (EL0)

The SP_EL0 characteristics are:

Purpose

Holds the stack pointer associated with EL0. At higher Exception levels, this is used as the current stack pointer when the value of SPSel.SP is 0.

Configurations

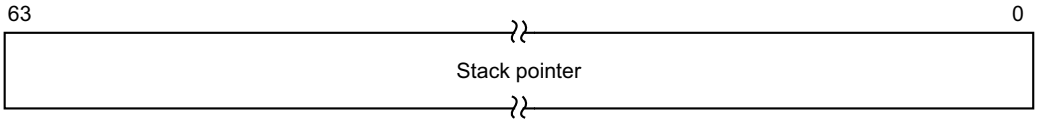
There are no configuration notes.

Attributes

SP_EL0 is a 64-bit register.

Field descriptions

The SP_EL0 bit assignments are:



Bits [63:0]

Stack pointer.

This field resets to an architecturally UNKNOWN value.

Accessing the SP_EL0

When the value of PSTATE.SP is 0, this register is accessible at all Exception levels as the current stack pointer.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, SP_EL0

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    else
        return SP_EL0;
elsif PSTATE.EL == EL2 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    else
        return SP_EL0;
elsif PSTATE.EL == EL3 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    else
        return SP_EL0;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR SP_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    else
        SP_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    else
        SP_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    else
        SP_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.2.12 SP_EL1, Stack Pointer (EL1)

The SP_EL1 characteristics are:

Purpose

Holds the stack pointer associated with EL1. When executing at EL1, the value of SPSel.SP determines the current stack pointer:

SPSel.SP	Current stack pointer
0b0	SP_EL0
0b1	SP_EL1

Configurations

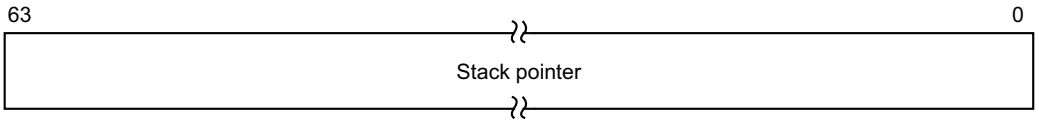
There are no configuration notes.

Attributes

SP_EL1 is a 64-bit register.

Field descriptions

The SP_EL1 bit assignments are:



Bits [63:0]

Stack pointer.

This field resets to an architecturally UNKNOWN value.

Accessing the SP_EL1

This accessibility information only applies to accesses using the MRS or MSR instructions.

When the value of SPSel.SP is 1, this register is also accessible at EL1 as the current stack pointer.

Note

When the value of SPSel.SP is 0, SP_EL0 is used as the current stack pointer at all Exception levels.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, SP_EL1

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        return NVMem[0x240];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return SP_EL1;
elsif PSTATE.EL == EL3 then
    return SP_EL1;
```

MSR SP_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NVMem[0x240] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    SP_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    SP_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.2.13 SP_EL2, Stack Pointer (EL2)

The SP_EL2 characteristics are:

Purpose

Holds the stack pointer associated with EL2. When executing at EL2, the value of SPSel. SP determines the current stack pointer:

SPSel.SP	Current stack pointer
0b0	SP_EL0
0b1	SP_EL2

Configurations

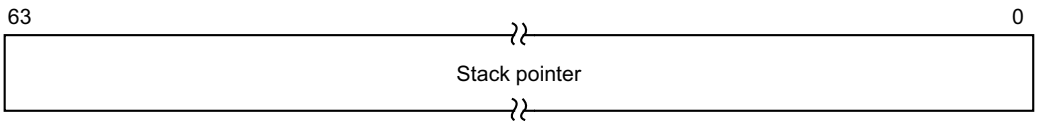
This register has no effect if EL2 is not enabled in the current Security state.

Attributes

SP_EL2 is a 64-bit register.

Field descriptions

The SP_EL2 bit assignments are:



Bits [63:0]

Stack pointer.

This field resets to an architecturally UNKNOWN value.

Accessing the SP_EL2

This accessibility information only applies to accesses using the MRS or MSR instructions.

When the value of SPSel.SP is 1, this register is also accessible at EL2 as the current stack pointer.

Note

When the value of SPSel.SP is 0, SP_EL0 is used as the current stack pointer at all Exception levels.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, SP_EL2

op0	op1	CRn	CRm	op2
0b11	0b110	0b0100	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    return SP_EL2;
```

MSR SP_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b110	0b0100	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    SP_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.2.14 SP_EL3, Stack Pointer (EL3)

The SP_EL3 characteristics are:

Purpose

Holds the stack pointer associated with EL3. When executing at EL3, the value of SPSel.SP determines the current stack pointer:

SPSel.SP	Current stack pointer
`0b0`	SP_EL0
`0b1`	SP_EL3

Configurations

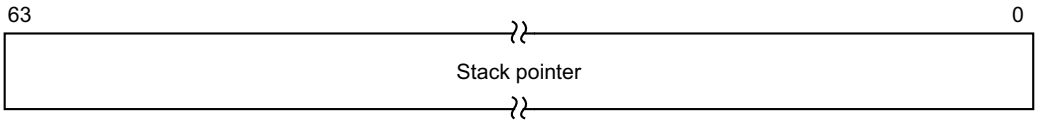
This register is present only when EL3 is implemented. Otherwise, direct accesses to SP_EL3 are UNDEFINED.

Attributes

SP_EL3 is a 64-bit register.

Field descriptions

The SP_EL3 bit assignments are:



Bits [63:0]

Stack pointer.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.2.15 SPSel, Stack Pointer Select

The SPSel characteristics are:

Purpose

Allows the Stack Pointer to be selected between SP_EL0 and SP_ELx.

Configurations

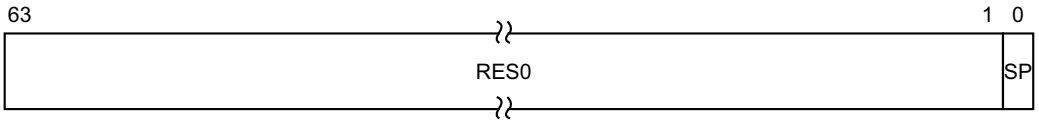
There are no configuration notes.

Attributes

SPSel is a 64-bit register.

Field descriptions

The SPSel bit assignments are:



Bits [63:1]

Reserved, RES0.

SP, bit [0]

Stack pointer to use. Possible values of this bit are:

`0b0` Use SP_EL0 at all Exception levels.

`0b1` Use SP_ELx for Exception level ELx.

This field resets to `1`.

Accessing the SPSel

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, SPSel

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    return Zeros(63):PSTATE.SP;
elsif PSTATE.EL == EL2 then
    return Zeros(63):PSTATE.SP;
elsif PSTATE.EL == EL3 then
    return Zeros(63):PSTATE.SP;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR SPSel, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    PSTATE.SP = X[t]<0>;
elsif PSTATE.EL == EL2 then
    PSTATE.SP = X[t]<0>;
elsif PSTATE.EL == EL3 then
    PSTATE.SP = X[t]<0>;
```

MSR SPSel, #<imm>

op0	op1	CRn	op2
0b00	0b000	0b0100	0b101

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.2.16 SPSR_abt, Saved Program Status Register (Abort mode)

The SPSR_abt characteristics are:

Purpose

Holds the saved process state when an exception is taken to Abort mode.

Configurations

AArch64 System register SPSR_abt[31:0] is architecturally mapped to AArch32 System register SPSR_abt[31:0].

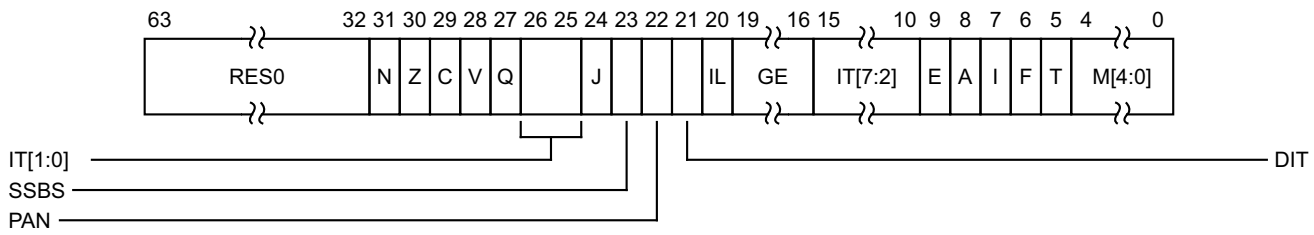
If EL1 only supports execution in AArch64 state, this register is RES0 from EL2 and EL3.

Attributes

SPSR_abt is a 64-bit register.

Field descriptions

The SPSR_abt bit assignments are:



Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to Abort mode, and copied to PSTATE.N on executing an exception return operation in Abort mode.

This field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to Abort mode, and copied to PSTATE.Z on executing an exception return operation in Abort mode.

This field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to Abort mode, and copied to PSTATE.C on executing an exception return operation in Abort mode.

This field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to Abort mode, and copied to PSTATE.V on executing an exception return operation in Abort mode.

This field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to Abort mode, and copied to PSTATE.Q on executing an exception return operation in Abort mode.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

IT[1:0], bits [26:25]

If-Then. Set to the value of PSTATE.IT[1:0] on taking an exception to Abort mode, and copied to PSTATE.IT[1:0] on executing an exception return operation in Abort mode.

On executing an exception return operation in Abort mode SPSR_abt.IT must contain a value that is valid for the instruction being returned to.

This field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.

Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When ARMv8.0-SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to Abort mode, and copied to PSTATE.SSBS on executing an exception return operation in Abort mode.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When ARMv8.1-PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to Abort mode, and copied to PSTATE.PAN on executing an exception return operation in Abort mode.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When ARMv8.4-DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to Abort mode, and copied to PSTATE.DIT on executing an exception return operation in Abort mode.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to Abort mode, and copied to PSTATE.IL on executing an exception return operation in Abort mode.

This field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to Abort mode, and copied to PSTATE.GE on executing an exception return operation in Abort mode.

This field resets to an architecturally UNKNOWN value.

IT[7:2], bits [15:10]

If-Then. Set to the value of PSTATE.IT[7:2] on taking an exception to Abort mode, and copied to PSTATE.IT[7:2] on executing an exception return operation in Abort mode.

SPSR_abt.IT must contain a value that is valid for the instruction being returned to.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to Abort mode, and copied to PSTATE.E on executing an exception return operation in Abort mode.

If the implementation does not support big-endian operation, SPSR_abt.E is RES0. If the implementation does not support little-endian operation, SPSR_abt.E is RES1. On executing an exception return operation in Abort mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_abt.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_abt.E is RES1.

This field resets to an architecturally UNKNOWN value.

A, bit [8]

SError interrupt mask. Set to the value of PSTATE.A on taking an exception to Abort mode, and copied to PSTATE.A on executing an exception return operation in Abort mode.

This field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to Abort mode, and copied to PSTATE.I on executing an exception return operation in Abort mode.

This field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to Abort mode, and copied to PSTATE.F on executing an exception return operation in Abort mode.

This field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to Abort mode, and copied to PSTATE.T on executing an exception return operation in Abort mode.

This field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to Abort mode, and copied to PSTATE.M[4:0] on executing an exception return operation in Abort mode.

`0b10000` User.

`0b10001` FIQ.

`0b10010` IRQ.

`0b10011` Supervisor.

`0b10111` Abort.

`0b11011` Undefined.

`0b11111` System.

Other values are reserved. If SPSR_abt.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in Abort mode is an illegal return event, as described in *Illegal return events from AArch32 state* on page G1-5726.

This field resets to an architecturally UNKNOWN value.

Accessing the SPSR_abt

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, SPSR_abt

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0011	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return SPSR_abt;
elsif PSTATE.EL == EL3 then
    return SPSR_abt;
```

MSR SPSR_abt, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0011	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    SPSR_abt = X[t];
elsif PSTATE.EL == EL3 then
    SPSR_abt = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.2.17 SPSR_EL1, Saved Program Status Register (EL1)

The SPSR_EL1 characteristics are:

Purpose

Holds the saved process state when an exception is taken to EL1.

Configurations

AArch64 System register SPSR_EL1[31:0] is architecturally mapped to AArch32 System register SPSR_svc[31:0].

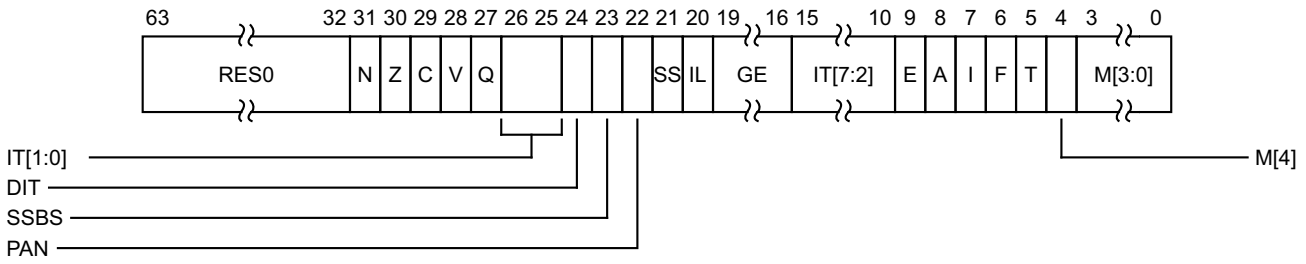
Attributes

SPSR_EL1 is a 64-bit register.

Field descriptions

The SPSR_EL1 bit assignments are:

When AArch32 is supported at any Exception level and exception taken from AArch32 state:



An exception return from EL1 using AArch64 makes SPSR_EL1 become UNKNOWN.

Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to EL1, and copied to PSTATE.N on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to EL1, and copied to PSTATE.Z on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to EL1, and copied to PSTATE.C on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to EL1, and copied to PSTATE.V on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to EL1, and copied to PSTATE.Q on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

IT[1:0], bits [26:25]

If-Then. Set to the value of PSTATE.IT[1:0] on taking an exception to EL1, and copied to PSTATE.IT[1:0] on executing an exception return operation in EL1.

On executing an exception return operation in EL1 SPSR_EL1.IT must contain a value that is valid for the instruction being returned to.

This field resets to an architecturally UNKNOWN value.

DIT, bit [24]

When ARMv8.4-DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to EL1, and copied to PSTATE.DIT on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SSBS, bit [23]

When ARMv8.0-SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to EL1, and copied to PSTATE.SSBS on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When ARMv8.1-PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to EL1, and copied to PSTATE.PAN on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SS, bit [21]

Software Step. Set to the value of PSTATE.SS on taking an exception to EL1, and conditionally copied to PSTATE.SS on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to EL1, and copied to PSTATE.IL on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to EL1, and copied to PSTATE.GE on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

IT[7:2], bits [15:10]

If-Then. Set to the value of PSTATE.IT[7:2] on taking an exception to EL1, and copied to PSTATE.IT[7:2] on executing an exception return operation in EL1.

SPSR_EL1.IT must contain a value that is valid for the instruction being returned to.

This field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to EL1, and copied to PSTATE.E on executing an exception return operation in EL1.

If the implementation does not support big-endian operation, SPSR_EL1.E is RES0. If the implementation does not support little-endian operation, SPSR_EL1.E is RES1. On executing an exception return operation in EL1, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_EL1.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_EL1.E is RES1.

This field resets to an architecturally UNKNOWN value.

A, bit [8]

SError interrupt mask. Set to the value of PSTATE.A on taking an exception to EL1, and copied to PSTATE.A on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to EL1, and copied to PSTATE.I on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to EL1, and copied to PSTATE.F on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to EL1, and copied to PSTATE.T on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

M[4], bit [4]

Execution state. Set to `0b1`, the value of PSTATE.nRW, on taking an exception to EL1 from AArch32 state, and copied to PSTATE.nRW on executing an exception return operation in EL1.

`0b1` AArch32 execution state.

This field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]

AArch32 Mode. Set to the value of PSTATE.M[3:0] on taking an exception to EL1, and copied to PSTATE.M[3:0] on executing an exception return operation in EL1.

`0b0000` User.

`0b0001` FIQ.

`0b0010` IRQ.

`0b0011` Supervisor.

`0b0111` Abort.

`0b1011` Undefined.

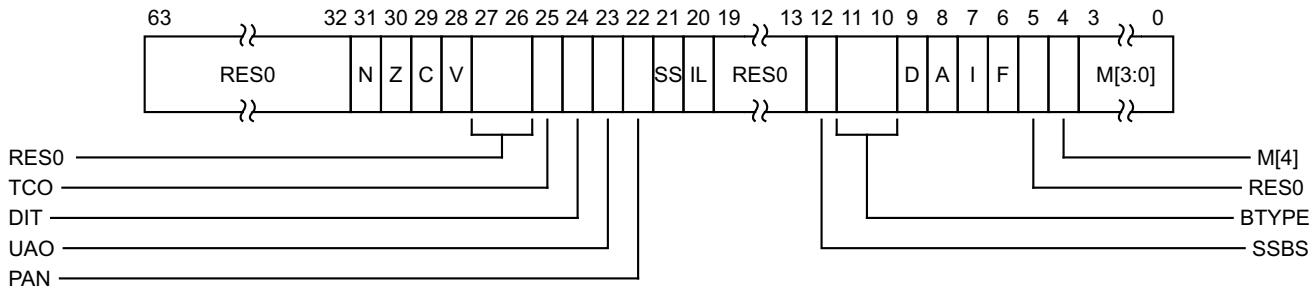
`0b1111` System.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Other values are reserved. If SPSR_EL1.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in EL1 is an illegal return event, as described in *Illegal return events from AArch64 state* on page D1-2351.

This field resets to an architecturally UNKNOWN value.

When exception taken from AArch64 state:



An exception return from EL1 using AArch64 makes SPSR_EL1 become UNKNOWN.

Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to EL1, and copied to PSTATE.N on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to EL1, and copied to PSTATE.Z on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to EL1, and copied to PSTATE.C on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to EL1, and copied to PSTATE.V on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

Bits [27:26]

Reserved, RES0.

TCO, bit [25]

When ARMv8.5-MemTag is implemented:

Tag Check Override. Set to the value of PSTATE.TCO on taking an exception to EL1, and copied to PSTATE.TCO on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

DIT, bit [24]

When ARMv8.4-DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to EL1, and copied to PSTATE.DIT on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

UAO, bit [23]

When ARMv8.2-UAO is implemented:

User Access Override. Set to the value of PSTATE.UAO on taking an exception to EL1, and copied to PSTATE.UAO on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When ARMv8.1-PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to EL1, and copied to PSTATE.PAN on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SS, bit [21]

Software Step. Set to the value of PSTATE.SS on taking an exception to EL1, and conditionally copied to PSTATE.SS on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to EL1, and copied to PSTATE.IL on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

Bits [19:13]

Reserved, RES0.

SSBS, bit [12]

When ARMv8.0-SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to EL1, and copied to PSTATE.SSBS on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

BTYPE, bits [11:10]

When ARMv8.5-BTI is implemented:

Branch Type Indicator. Set to the value of PSTATE.BTYPE on taking an exception to EL1, and copied to PSTATE.BTYPE on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

Reserved, RES0.

D, bit [9]

Debug exception mask. Set to the value of PSTATE.D on taking an exception to EL1, and copied to PSTATE.D on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

A, bit [8]

SError interrupt mask. Set to the value of PSTATE.A on taking an exception to EL1, and copied to PSTATE.A on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to EL1, and copied to PSTATE.I on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to EL1, and copied to PSTATE.F on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

M[4], bit [4]

Execution state. Set to `0b0`, the value of PSTATE.nRW, on taking an exception to EL1 from AArch64 state, and copied to PSTATE.nRW on executing an exception return operation in EL1.

`0b0` AArch64 execution state.

This field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]

AArch64 Exception level and selected Stack Pointer.

`0b0000` EL0t.

`0b0100` EL1t.

`0b0101` EL1h.

Other values are reserved. If SPSR_EL1.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in EL1 is an illegal return event, as described in *Illegal return events from AArch64 state* on page D1-2351.

The bits in this field are interpreted as follows:

- M[3:2] is set to the value of PSTATE.EL on taking an exception to EL1 and copied to PSTATE.EL on executing an exception return operation in EL1.
- M[1] is unused and is 0 for all non-reserved values.
- M[0] is set to the value of PSTATE.SP on taking an exception to EL1 and copied to PSTATE.SP on executing an exception return operation in EL1

This field resets to an architecturally UNKNOWN value.

Accessing the SPSR_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic SPSR_EL1 or SPSR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, SPSR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1> == '01' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x160];
    else
        return SPSR_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return SPSR_EL2;
    else
        return SPSR_EL1;
elsif PSTATE.EL == EL3 then
    return SPSR_EL1;
```

MSR SPSR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1> == '01' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x160] = X[t];
    else
        SPSR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        SPSR_EL2 = X[t];
    else
        SPSR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    SPSR_EL1 = X[t];
```

MRS <Xt>, SPSR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0100	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        return NVMem[0x160];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return SPSR_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        return SPSR_EL1;
    else
        UNDEFINED;
```

MSR SPSR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0100	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        NVMem[0x160] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        SPSR_EL1 = X[t];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        SPSR_EL1 = X[t];
    else
        UNDEFINED;
```

MRS <Xt>, SPSR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        return SPSR_EL1;
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return SPSR_EL2;
elsif PSTATE.EL == EL3 then
    return SPSR_EL2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR SPSR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        SPSR_EL1 = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    SPSR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    SPSR_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.2.18 SPSR_EL2, Saved Program Status Register (EL2)

The SPSR_EL2 characteristics are:

Purpose

Holds the saved process state when an exception is taken to EL2.

Configurations

AArch64 System register SPSR_EL2[31:0] is architecturally mapped to AArch32 System register SPSR_hyp[31:0].

This register has no effect if EL2 is not enabled in the current Security state.

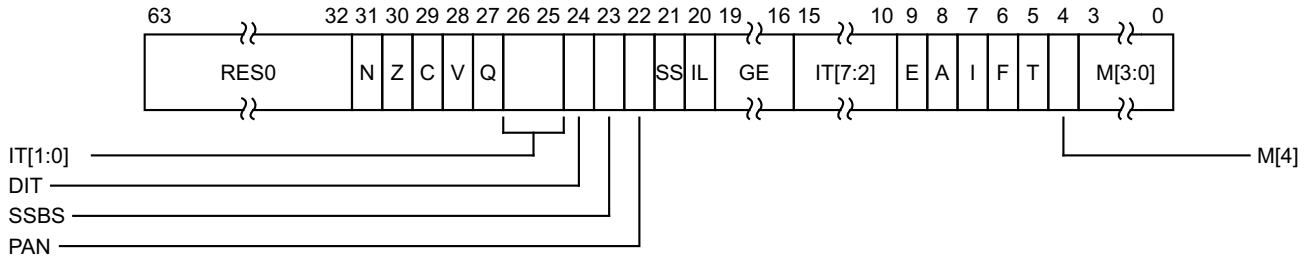
Attributes

SPSR_EL2 is a 64-bit register.

Field descriptions

The SPSR_EL2 bit assignments are:

When AArch32 is supported at any Exception level and exception taken from AArch32 state:



An exception return from EL2 using AArch64 makes SPSR_EL2 become UNKNOWN.

Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to EL2, and copied to PSTATE.N on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to EL2, and copied to PSTATE.Z on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to EL2, and copied to PSTATE.C on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to EL2, and copied to PSTATE.V on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to EL2, and copied to PSTATE.Q on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

IT[1:0], bits [26:25]

If-Then. Set to the value of PSTATE.IT[1:0] on taking an exception to EL2, and copied to PSTATE.IT[1:0] on executing an exception return operation in EL2.

On executing an exception return operation in EL2 SPSR_EL2.IT must contain a value that is valid for the instruction being returned to.

This field resets to an architecturally UNKNOWN value.

DIT, bit [24]

When ARMv8.4-DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to EL2, and copied to PSTATE.DIT on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SSBS, bit [23]

When ARMv8.0-SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to EL2, and copied to PSTATE.SSBS on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When ARMv8.1-PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to EL2, and copied to PSTATE.PAN on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SS, bit [21]

Software Step. Set to the value of PSTATE.SS on taking an exception to EL2, and conditionally copied to PSTATE.SS on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to EL2, and copied to PSTATE.IL on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to EL2, and copied to PSTATE.GE on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

IT[7:2], bits [15:10]

If-Then. Set to the value of PSTATE.IT[7:2] on taking an exception to EL2, and copied to PSTATE.IT[7:2] on executing an exception return operation in EL2.

SPSR_EL2.IT must contain a value that is valid for the instruction being returned to.

This field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to EL2, and copied to PSTATE.E on executing an exception return operation in EL2.

If the implementation does not support big-endian operation, SPSR_EL2.E is RES0. If the implementation does not support little-endian operation, SPSR_EL2.E is RES1. On executing an exception return operation in EL2, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_EL2.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_EL2.E is RES1.

This field resets to an architecturally UNKNOWN value.

A, bit [8]

SError interrupt mask. Set to the value of PSTATE.A on taking an exception to EL2, and copied to PSTATE.A on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to EL2, and copied to PSTATE.I on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to EL2, and copied to PSTATE.F on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to EL2, and copied to PSTATE.T on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

M[4], bit [4]

Execution state. Set to `0b1`, the value of PSTATE.nRW, on taking an exception to EL2 from AArch32 state, and copied to PSTATE.nRW on executing an exception return operation in EL2.

`0b1` AArch32 execution state.

This field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]

AArch32 Mode. Set to the value of PSTATE.M[3:0] on taking an exception to EL2, and copied to PSTATE.M[3:0] on executing an exception return operation in EL2.

`0b0000` User.

`0b0001` FIQ.

`0b0010` IRQ.

`0b0011` Supervisor.

`0b0111` Abort.

`0b1010` Hyp.

`0b1011` Undefined.

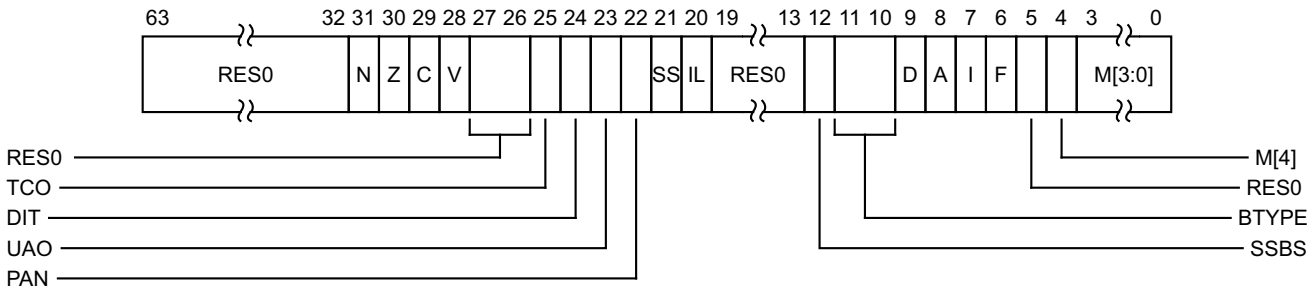
Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1111` System.

Other values are reserved. If SPSR_EL2.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in EL2 is an illegal return event, as described in *Illegal return events from AArch64 state* on page D1-2351.

This field resets to an architecturally UNKNOWN value.

When exception taken from AArch64 state:



An exception return from EL2 using AArch64 makes SPSR_EL2 become UNKNOWN.

Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to EL2, and copied to PSTATE.N on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to EL2, and copied to PSTATE.Z on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to EL2, and copied to PSTATE.C on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to EL2, and copied to PSTATE.V on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

Bits [27:26]

Reserved, RES0.

TCO, bit [25]

When ARMv8.5-MemTag is implemented:

Tag Check Override. Set to the value of PSTATE.TCO on taking an exception to EL2, and copied to PSTATE.TCO on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

DIT, bit [24]

When ARMv8.4-DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to EL2, and copied to PSTATE.DIT on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

UAO, bit [23]

When ARMv8.2-UAO is implemented:

User Access Override. Set to the value of PSTATE.UAO on taking an exception to EL2, and copied to PSTATE.UAO on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When ARMv8.1-PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to EL2, and copied to PSTATE.PAN on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SS, bit [21]

Software Step. Set to the value of PSTATE.SS on taking an exception to EL2, and conditionally copied to PSTATE.SS on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to EL2, and copied to PSTATE.IL on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

Bits [19:13]

Reserved, RES0.

SSBS, bit [12]

When ARMv8.0-SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to EL2, and copied to PSTATE.SSBS on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

BTYPE, bits [11:10]

When ARMv8.5-BTI is implemented:

Branch Type Indicator. Set to the value of PSTATE.BTYPE on taking an exception to EL2, and copied to PSTATE.BTYPE on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

Reserved, RES0.

D, bit [9]

Debug exception mask. Set to the value of PSTATE.D on taking an exception to EL2, and copied to PSTATE.D on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

A, bit [8]

SError interrupt mask. Set to the value of PSTATE.A on taking an exception to EL2, and copied to PSTATE.A on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to EL2, and copied to PSTATE.I on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to EL2, and copied to PSTATE.F on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

M[4], bit [4]

Execution state. Set to `0b0`, the value of PSTATE.nRW, on taking an exception to EL2 from AArch64 state, and copied to PSTATE.nRW on executing an exception return operation in EL2.

`0b0` AArch64 execution state.

This field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]

AArch64 Exception level and selected Stack Pointer.

`0b0000` EL0t.

`0b0100` EL1t.

`0b0101` EL1h.

`0b1000` EL2t.

`0b1001` EL2h.

Other values are reserved. If SPSR_EL2.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in EL2 is an illegal return event, as described in *Illegal return events from AArch64 state* on page D1-2351.

The bits in this field are interpreted as follows:

- M[3:2] is set to the value of PSTATE.EL on taking an exception to EL2 and copied to PSTATE.EL on executing an exception return operation in EL2.
- M[1] is unused and is 0 for all non-reserved values.
- M[0] is set to the value of PSTATE.SP on taking an exception to EL2 and copied to PSTATE.SP on executing an exception return operation in EL2

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the SPSR_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic SPSR_EL2 or SPSR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, SPSR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        return SPSR_EL1;
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return SPSR_EL2;
elsif PSTATE.EL == EL3 then
    return SPSR_EL2;
```

MSR SPSR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        SPSR_EL1 = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    SPSR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    SPSR_EL2 = X[t];
```

MRS <Xt>, SPSR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1> == '01' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        return NVMem[0x160];
    else
        return SPSR_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return SPSR_EL2;
    else
        return SPSR_EL1;
elsif PSTATE.EL == EL3 then
    return SPSR_EL1;
```

MSR SPSR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1> == '01' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x160] = X[t];
    else
        SPSR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        SPSR_EL2 = X[t];
    else
        SPSR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    SPSR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.2.19 SPSR_EL3, Saved Program Status Register (EL3)

The SPSR_EL3 characteristics are:

Purpose

Holds the saved process state when an exception is taken to EL3.

Configurations

AArch64 System register SPSR_EL3[31:0] can be mapped to AArch32 System register SPSR_mon[31:0], but this is not architecturally mandated.

This register is present only when EL3 is implemented. Otherwise, direct accesses to SPSR_EL3 are UNDEFINED.

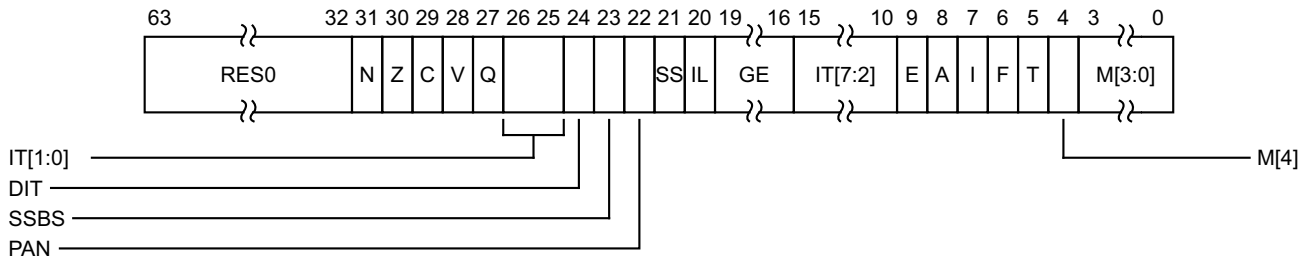
Attributes

SPSR_EL3 is a 64-bit register.

Field descriptions

The SPSR_EL3 bit assignments are:

When AArch32 is supported at any Exception level and exception taken from AArch32 state:



An exception return from EL3 using AArch64 makes SPSR_EL1 become UNKNOWN.

Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to EL3, and copied to PSTATE.N on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to EL3, and copied to PSTATE.Z on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to EL3, and copied to PSTATE.C on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to EL3, and copied to PSTATE.V on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to EL3, and copied to PSTATE.Q on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

IT[1:0], bits [26:25]

If-Then. Set to the value of PSTATE.IT[1:0] on taking an exception to EL3, and copied to PSTATE.IT[1:0] on executing an exception return operation in EL3.

On executing an exception return operation in EL3 SPSR_EL1.IT must contain a value that is valid for the instruction being returned to.

This field resets to an architecturally UNKNOWN value.

DIT, bit [24]

When ARMv8.4-DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to EL3, and copied to PSTATE.DIT on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SSBS, bit [23]

When ARMv8.0-SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to EL3, and copied to PSTATE.SSBS on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When ARMv8.1-PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to EL3, and copied to PSTATE.PAN on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SS, bit [21]

Software Step. Set to the value of PSTATE.SS on taking an exception to EL3, and conditionally copied to PSTATE.SS on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to EL3, and copied to PSTATE.IL on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to EL3, and copied to PSTATE.GE on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

IT[7:2], bits [15:10]

If-Then. Set to the value of PSTATE.IT[7:2] on taking an exception to EL3, and copied to PSTATE.IT[7:2] on executing an exception return operation in EL3.

SPSR_EL1.IT must contain a value that is valid for the instruction being returned to.

This field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to EL3, and copied to PSTATE.E on executing an exception return operation in EL3.

If the implementation does not support big-endian operation, SPSR_EL1.E is RES0. If the implementation does not support little-endian operation, SPSR_EL1.E is RES1. On executing an exception return operation in EL3, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_EL1.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_EL1.E is RES1.

This field resets to an architecturally UNKNOWN value.

A, bit [8]

SError interrupt mask. Set to the value of PSTATE.A on taking an exception to EL3, and copied to PSTATE.A on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to EL3, and copied to PSTATE.I on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to EL3, and copied to PSTATE.F on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to EL3, and copied to PSTATE.T on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

M[4], bit [4]

Execution state. Set to `0b1`, the value of PSTATE.nRW, on taking an exception to EL3 from AArch32 state, and copied to PSTATE.nRW on executing an exception return operation in EL3.

`0b1` AArch32 execution state.

This field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]

AArch32 Mode. Set to the value of PSTATE.M[3:0] on taking an exception to EL3, and copied to PSTATE.M[3:0] on executing an exception return operation in EL3.

`0b0000` User.

`0b0001` FIQ.

`0b0010` IRQ.

`0b0011` Supervisor.

`0b0110` Monitor.

`0b0111` Abort.

`0b1010` Hyp.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

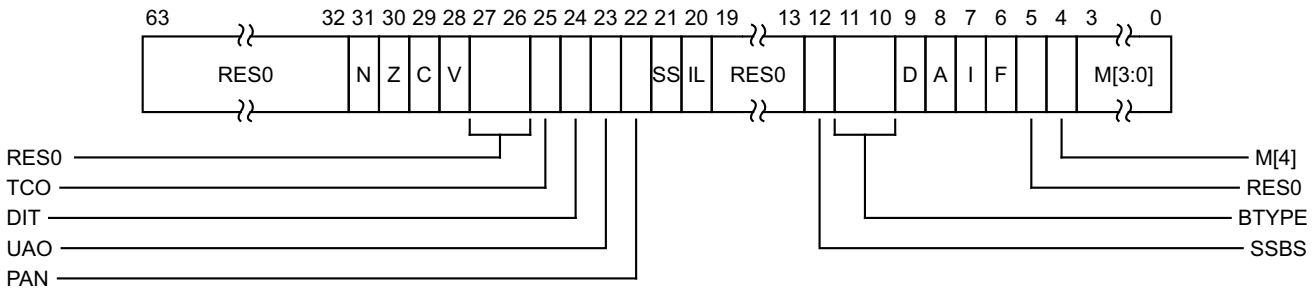
`0b1011` Undefined.

`0b1111` System.

Other values are reserved. If SPSR_EL1.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in EL3 is an illegal return event, as described in *Illegal return events from AArch64 state* on page D1-2351.

This field resets to an architecturally UNKNOWN value.

When exception taken from AArch64 state:



An exception return from EL3 using AArch64 makes SPSR_EL1 become UNKNOWN.

Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to EL3, and copied to PSTATE.N on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to EL3, and copied to PSTATE.Z on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to EL3, and copied to PSTATE.C on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to EL3, and copied to PSTATE.V on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

Bits [27:26]

Reserved, RES0.

TCO, bit [25]

When ARMv8.5-MemTag is implemented:

Tag Check Override. Set to the value of PSTATE.TCO on taking an exception to EL3, and copied to PSTATE.TCO on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

Reserved, RES0.

DIT, bit [24]

When ARMv8.4-DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to EL3, and copied to PSTATE.DIT on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

UAO, bit [23]

When ARMv8.2-UAO is implemented:

User Access Override. Set to the value of PSTATE.UAO on taking an exception to EL3, and copied to PSTATE.UAO on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When ARMv8.1-PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to EL3, and copied to PSTATE.PAN on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SS, bit [21]

Software Step. Set to the value of PSTATE.SS on taking an exception to EL3, and conditionally copied to PSTATE.SS on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to EL3, and copied to PSTATE.IL on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

Bits [19:13]

Reserved, RES0.

SSBS, bit [12]

When ARMv8.0-SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to EL3, and copied to PSTATE.SSBS on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

BTYPE, bits [11:10]

When ARMv8.5-BTI is implemented:

Branch Type Indicator. Set to the value of PSTATE.BTYPE on taking an exception to EL3, and copied to PSTATE.BTYPE on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

D, bit [9]

Debug exception mask. Set to the value of PSTATE.D on taking an exception to EL3, and copied to PSTATE.D on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

A, bit [8]

SError interrupt mask. Set to the value of PSTATE.A on taking an exception to EL3, and copied to PSTATE.A on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to EL3, and copied to PSTATE.I on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to EL3, and copied to PSTATE.F on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

M[4], bit [4]

Execution state. Set to `0b0`, the value of PSTATE.nRW, on taking an exception to EL3 from AArch64 state, and copied to PSTATE.nRW on executing an exception return operation in EL3.

`0b0` AArch64 execution state.

This field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]

AArch64 Exception level and selected Stack Pointer.

`0b0000` EL0t.

`0b0100` EL1t.

`0b0101` EL1h.

`0b1000` EL2t.

`0b1001` EL2h.

`0b1100` EL3t.

`0b1101` EL3h.

Other values are reserved. If SPSR_EL1.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in EL3 is an illegal return event, as described in *Illegal return events from AArch64 state* on page D1-2351.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The bits in this field are interpreted as follows:

- M[3:2] is set to the value of PSTATE.EL on taking an exception to EL3 and copied to PSTATE.EL on executing an exception return operation in EL3.
- M[1] is unused and is 0 for all non-reserved values.
- M[0] is set to the value of PSTATE.SP on taking an exception to EL3 and copied to PSTATE.SP on executing an exception return operation in EL3

This field resets to an architecturally UNKNOWN value.

Accessing the SPSR_EL3

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, SPSR_EL3

op0	op1	CRn	CRm	op2
0b11	0b110	0b0100	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    return SPSR_EL3;
```

MSR SPSR_EL3, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b110	0b0100	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    SPSR_EL3 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.2.20 SPSR_fiq, Saved Program Status Register (FIQ mode)

The SPSR_fiq characteristics are:

Purpose

Holds the saved process state when an exception is taken to FIQ mode.

Configurations

AArch64 System register SPSR_fiq[31:0] is architecturally mapped to AArch32 System register SPSR_fiq[31:0].

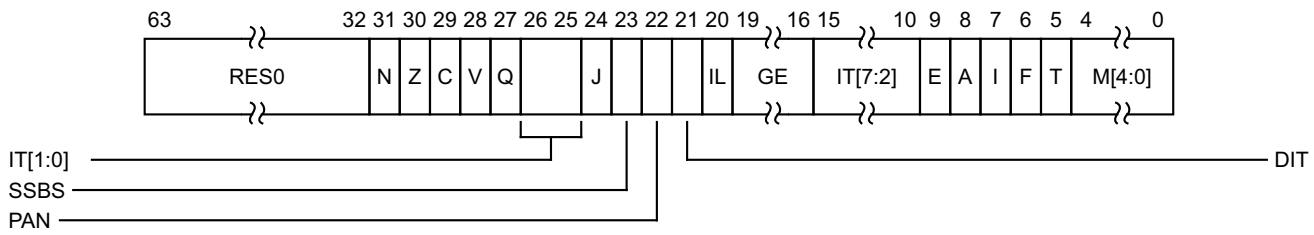
If EL1 only supports execution in AArch64 state, this register is RES0 from EL2 and EL3.

Attributes

SPSR_fiq is a 64-bit register.

Field descriptions

The SPSR_fiq bit assignments are:



Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to FIQ mode, and copied to PSTATE.N on executing an exception return operation in FIQ mode.

This field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to FIQ mode, and copied to PSTATE.Z on executing an exception return operation in FIQ mode.

This field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to FIQ mode, and copied to PSTATE.C on executing an exception return operation in FIQ mode.

This field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to FIQ mode, and copied to PSTATE.V on executing an exception return operation in FIQ mode.

This field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to FIQ mode, and copied to PSTATE.Q on executing an exception return operation in FIQ mode.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

IT[1:0], bits [26:25]

If-Then. Set to the value of PSTATE.IT[1:0] on taking an exception to FIQ mode, and copied to PSTATE.IT[1:0] on executing an exception return operation in FIQ mode.

On executing an exception return operation in FIQ mode SPSR_fiq.IT must contain a value that is valid for the instruction being returned to.

This field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.

Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When ARMv8.0-SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to FIQ mode, and copied to PSTATE.SSBS on executing an exception return operation in FIQ mode.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When ARMv8.1-PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to FIQ mode, and copied to PSTATE.PAN on executing an exception return operation in FIQ mode.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When ARMv8.4-DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to FIQ mode, and copied to PSTATE.DIT on executing an exception return operation in FIQ mode.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to FIQ mode, and copied to PSTATE.IL on executing an exception return operation in FIQ mode.

This field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to FIQ mode, and copied to PSTATE.GE on executing an exception return operation in FIQ mode.

This field resets to an architecturally UNKNOWN value.

IT[7:2], bits [15:10]

If-Then. Set to the value of PSTATE.IT[7:2] on taking an exception to FIQ mode, and copied to PSTATE.IT[7:2] on executing an exception return operation in FIQ mode.

SPSR_fiq.IT must contain a value that is valid for the instruction being returned to.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to FIQ mode, and copied to PSTATE.E on executing an exception return operation in FIQ mode.

If the implementation does not support big-endian operation, SPSR_fiq.E is RES0. If the implementation does not support little-endian operation, SPSR_fiq.E is RES1. On executing an exception return operation in FIQ mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_fiq.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_fiq.E is RES1.

This field resets to an architecturally UNKNOWN value.

A, bit [8]

SError interrupt mask. Set to the value of PSTATE.A on taking an exception to FIQ mode, and copied to PSTATE.A on executing an exception return operation in FIQ mode.

This field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to FIQ mode, and copied to PSTATE.I on executing an exception return operation in FIQ mode.

This field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to FIQ mode, and copied to PSTATE.F on executing an exception return operation in FIQ mode.

This field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to FIQ mode, and copied to PSTATE.T on executing an exception return operation in FIQ mode.

This field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to FIQ mode, and copied to PSTATE.M[4:0] on executing an exception return operation in FIQ mode.

`0b10000` User.

`0b10001` FIQ.

`0b10010` IRQ.

`0b10011` Supervisor.

`0b10111` Abort.

`0b11011` Undefined.

`0b11111` System.

Other values are reserved. If SPSR_fiq.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in FIQ mode is an illegal return event, as described in *Illegal return events from AArch32 state* on page G1-5726.

This field resets to an architecturally UNKNOWN value.

Accessing the SPSR_fiq

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, SPSR_fiq

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0011	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return SPSR_fiq;
elsif PSTATE.EL == EL3 then
    return SPSR_fiq;
```

MSR SPSR_fiq, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0011	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    SPSR_fiq = X[t];
elsif PSTATE.EL == EL3 then
    SPSR_fiq = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.2.21 SPSR_irq, Saved Program Status Register (IRQ mode)

The SPSR_irq characteristics are:

Purpose

Holds the saved process state when an exception is taken to IRQ mode.

Configurations

AArch64 System register SPSR_irq[31:0] is architecturally mapped to AArch32 System register SPSR_irq[31:0].

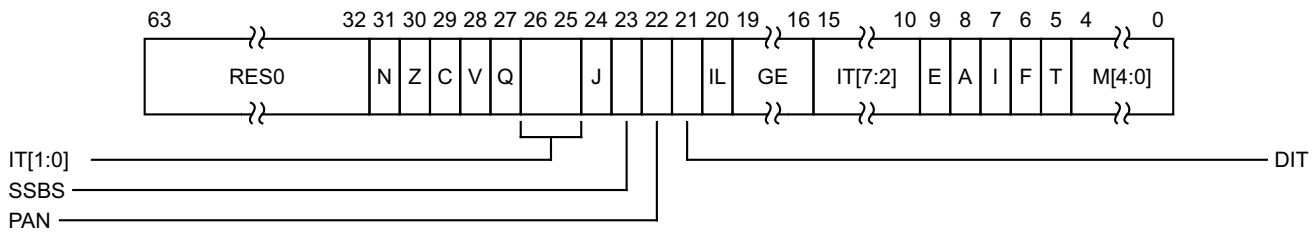
If EL1 only supports execution in AArch64 state, this register is RES0 from EL2 and EL3.

Attributes

SPSR_irq is a 64-bit register.

Field descriptions

The SPSR_irq bit assignments are:



Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to IRQ mode, and copied to PSTATE.N on executing an exception return operation in IRQ mode.

This field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to IRQ mode, and copied to PSTATE.Z on executing an exception return operation in IRQ mode.

This field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to IRQ mode, and copied to PSTATE.C on executing an exception return operation in IRQ mode.

This field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to IRQ mode, and copied to PSTATE.V on executing an exception return operation in IRQ mode.

This field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to IRQ mode, and copied to PSTATE.Q on executing an exception return operation in IRQ mode.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

IT[1:0], bits [26:25]

If-Then. Set to the value of PSTATE.IT[1:0] on taking an exception to IRQ mode, and copied to PSTATE.IT[1:0] on executing an exception return operation in IRQ mode.

On executing an exception return operation in IRQ mode SPSR_irq.IT must contain a value that is valid for the instruction being returned to.

This field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.

Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When ARMv8.0-SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to IRQ mode, and copied to PSTATE.SSBS on executing an exception return operation in IRQ mode.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When ARMv8.1-PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to IRQ mode, and copied to PSTATE.PAN on executing an exception return operation in IRQ mode.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When ARMv8.4-DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to IRQ mode, and copied to PSTATE.DIT on executing an exception return operation in IRQ mode.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to IRQ mode, and copied to PSTATE.IL on executing an exception return operation in IRQ mode.

This field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to IRQ mode, and copied to PSTATE.GE on executing an exception return operation in IRQ mode.

This field resets to an architecturally UNKNOWN value.

IT[7:2], bits [15:10]

If-Then. Set to the value of PSTATE.IT[7:2] on taking an exception to IRQ mode, and copied to PSTATE.IT[7:2] on executing an exception return operation in IRQ mode.

SPSR_irq.IT must contain a value that is valid for the instruction being returned to.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to IRQ mode, and copied to PSTATE.E on executing an exception return operation in IRQ mode.

If the implementation does not support big-endian operation, SPSR_irq.E is RES0. If the implementation does not support little-endian operation, SPSR_irq.E is RES1. On executing an exception return operation in IRQ mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_irq.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_irq.E is RES1.

This field resets to an architecturally UNKNOWN value.

A, bit [8]

SError interrupt mask. Set to the value of PSTATE.A on taking an exception to IRQ mode, and copied to PSTATE.A on executing an exception return operation in IRQ mode.

This field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to IRQ mode, and copied to PSTATE.I on executing an exception return operation in IRQ mode.

This field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to IRQ mode, and copied to PSTATE.F on executing an exception return operation in IRQ mode.

This field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to IRQ mode, and copied to PSTATE.T on executing an exception return operation in IRQ mode.

This field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to IRQ mode, and copied to PSTATE.M[4:0] on executing an exception return operation in IRQ mode.

`0b10000` User.

`0b10001` FIQ.

`0b10010` IRQ.

`0b10011` Supervisor.

`0b10111` Abort.

`0b11011` Undefined.

`0b11111` System.

Other values are reserved. If SPSR_irq.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in IRQ mode is an illegal return event, as described in *Illegal return events from AArch32 state* on page G1-5726.

This field resets to an architecturally UNKNOWN value.

Accessing the SPSR_irq

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, SPSR_irq

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return SPSR_irq;
elsif PSTATE.EL == EL3 then
    return SPSR_irq;
```

MSR SPSR_irq, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    SPSR_irq = X[t];
elsif PSTATE.EL == EL3 then
    SPSR_irq = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.2.22 SPSR_und, Saved Program Status Register (Undefined mode)

The SPSR_und characteristics are:

Purpose

Holds the saved process state when an exception is taken to Undefined mode.

Configurations

AArch64 System register SPSR_und[31:0] is architecturally mapped to AArch32 System register SPSR_und[31:0].

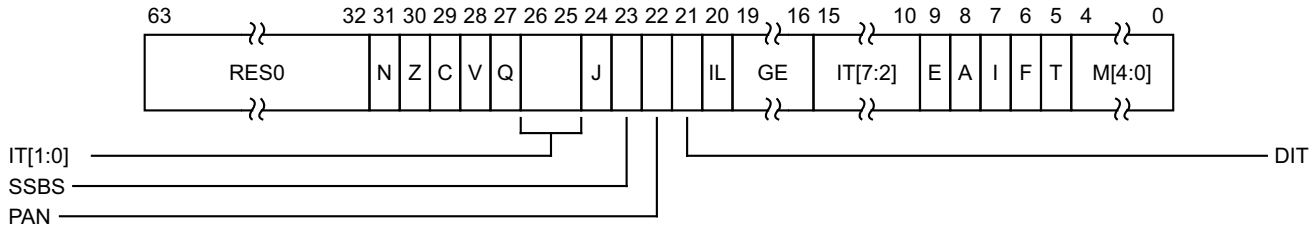
If EL1 only supports execution in AArch64 state, this register is RES0 from EL2 and EL3.

Attributes

SPSR_und is a 64-bit register.

Field descriptions

The SPSR_und bit assignments are:



Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to Undefined mode, and copied to PSTATE.N on executing an exception return operation in Undefined mode.

This field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to Undefined mode, and copied to PSTATE.Z on executing an exception return operation in Undefined mode.

This field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to Undefined mode, and copied to PSTATE.C on executing an exception return operation in Undefined mode.

This field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to Undefined mode, and copied to PSTATE.V on executing an exception return operation in Undefined mode.

This field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to Undefined mode, and copied to PSTATE.Q on executing an exception return operation in Undefined mode.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

IT[1:0], bits [26:25]

If-Then. Set to the value of PSTATE.IT[1:0] on taking an exception to Undefined mode, and copied to PSTATE.IT[1:0] on executing an exception return operation in Undefined mode.

On executing an exception return operation in Undefined mode SPSR_und.IT must contain a value that is valid for the instruction being returned to.

This field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.

Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When ARMv8.0-SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to Undefined mode, and copied to PSTATE.SSBS on executing an exception return operation in Undefined mode.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When ARMv8.1-PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to Undefined mode, and copied to PSTATE.PAN on executing an exception return operation in Undefined mode.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When ARMv8.4-DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to Undefined mode, and copied to PSTATE.DIT on executing an exception return operation in Undefined mode.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to Undefined mode, and copied to PSTATE.IL on executing an exception return operation in Undefined mode.

This field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to Undefined mode, and copied to PSTATE.GE on executing an exception return operation in Undefined mode.

This field resets to an architecturally UNKNOWN value.

IT[7:2], bits [15:10]

If-Then. Set to the value of PSTATE.IT[7:2] on taking an exception to Undefined mode, and copied to PSTATE.IT[7:2] on executing an exception return operation in Undefined mode.

SPSR_und.IT must contain a value that is valid for the instruction being returned to.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to Undefined mode, and copied to PSTATE.E on executing an exception return operation in Undefined mode.

If the implementation does not support big-endian operation, SPSR_und.E is RES0. If the implementation does not support little-endian operation, SPSR_und.E is RES1. On executing an exception return operation in Undefined mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_und.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_und.E is RES1.

This field resets to an architecturally UNKNOWN value.

A, bit [8]

SError interrupt mask. Set to the value of PSTATE.A on taking an exception to Undefined mode, and copied to PSTATE.A on executing an exception return operation in Undefined mode.

This field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to Undefined mode, and copied to PSTATE.I on executing an exception return operation in Undefined mode.

This field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to Undefined mode, and copied to PSTATE.F on executing an exception return operation in Undefined mode.

This field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to Undefined mode, and copied to PSTATE.T on executing an exception return operation in Undefined mode.

This field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to Undefined mode, and copied to PSTATE.M[4:0] on executing an exception return operation in Undefined mode.

`0b10000` User.

`0b10001` FIQ.

`0b10010` IRQ.

`0b10011` Supervisor.

`0b10111` Abort.

`0b11011` Undefined.

`0b11111` System.

Other values are reserved. If SPSR_und.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in Undefined mode is an illegal return event, as described in *Illegal return events from AArch32 state* on page G1-5726.

This field resets to an architecturally UNKNOWN value.

Accessing the SPSR_und

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, SPSR_und

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0011	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return SPSR_und;
elsif PSTATE.EL == EL3 then
    return SPSR_und;
```

MSR SPSR_und, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0011	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    SPSR_und = X[t];
elsif PSTATE.EL == EL3 then
    SPSR_und = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.2.23 SSBS, Speculative Store Bypass Safe

The SSBS characteristics are:

Purpose

Allows access to the Speculative Store Bypass Safe bit.

Configurations

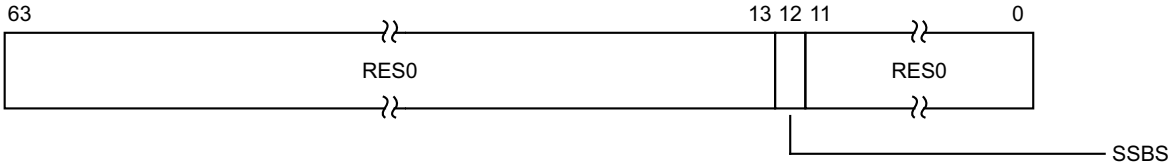
This register is present only when ARMv8.0-SSBS is implemented. Otherwise, direct accesses to SSBS are UNDEFINED.

Attributes

SSBS is a 64-bit register.

Field descriptions

The SSBS bit assignments are:



Bits [63:13]

Reserved, RES0.

SSBS, bit [12]

Speculative Store Bypass Safe.

Prohibits speculative loads or stores which might practically allow a cache timing side channel.

A cache timing side channel might be exploited where a load or store uses an address that is derived from a register that is being loaded from memory using a load instruction speculatively read from a memory location. If PSTATE.SSBS is enabled, the address derived from the load instruction might be from earlier in the coherence order than the latest store to that memory location with the same virtual address.

`0b0` Hardware is not permitted to load or store speculatively, in a manner that could practically give rise to a cache timing side channel, using an address derived from a register value that has been loaded from memory using a load instruction (L) that speculatively reads an entry from earlier in the coherence order from that location being loaded from than the entry generated by the latest store (S) to that location using the same virtual address as L.

`0b1` Hardware is permitted to load or store speculatively, in a manner that could practically give rise to a cache timing side channel, using an address derived from a register value that has been loaded from memory using a load instruction (L) that speculatively reads an entry from earlier in the coherence order fro that location being loaded from than the entry generated by the latest store (S) to that location using the same virtual address as L.

The value of this bit is set to the value in the SCTLR_ELx.DSSBS field on taking an exception to ELx.

This field resets to an IMPLEMENTATION DEFINED value.

Bits [11:0]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the SSBS

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, SSBS

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b110

```
if PSTATE.EL == EL0 then
    return Zeros(51):PSTATE.SSBS:Zeros(12);
elsif PSTATE.EL == EL1 then
    return Zeros(51):PSTATE.SSBS:Zeros(12);
elsif PSTATE.EL == EL2 then
    return Zeros(51):PSTATE.SSBS:Zeros(12);
elsif PSTATE.EL == EL3 then
    return Zeros(51):PSTATE.SSBS:Zeros(12);
```

MSR SSBS, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b110

```
if PSTATE.EL == EL0 then
    PSTATE.SSBS = X[t]<12>;
elsif PSTATE.EL == EL1 then
    PSTATE.SSBS = X[t]<12>;
elsif PSTATE.EL == EL2 then
    PSTATE.SSBS = X[t]<12>;
elsif PSTATE.EL == EL3 then
    PSTATE.SSBS = X[t]<12>;
```

MSR SSBS, #<imm>

op0	op1	CRn	op2
0b00	0b011	0b0100	0b001

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.2.24 TCO, Tag Check Override

The TCO characteristics are:

Purpose

When ARMv8.5-MemTag is implemented, this register allows tag checks to be disabled globally.

Configurations

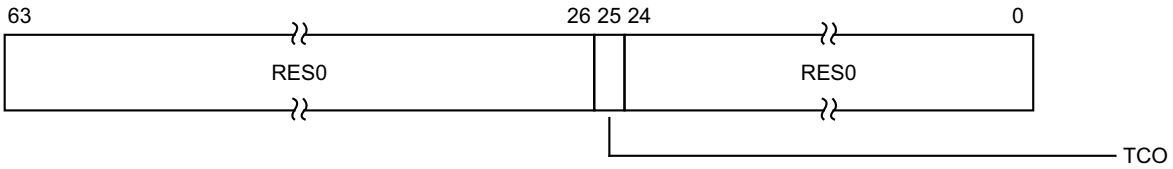
This register is present only when ARMv8.5-MemTag is implemented. Otherwise, direct accesses to TCO are UNDEFINED.

Attributes

TCO is a 64-bit register.

Field descriptions

The TCO bit assignments are:



Bits [63:26]

Reserved, RES0.

TCO, bit [25]

Allows memory tag checks to be globally disabled.

`0b0` Loads and Stores are not affected by this control.

`0b1` Loads and Stores are unchecked.

Bits [24:0]

Reserved, RES0.

Accessing the TCO

For details on the operation of the MSR (immediate) accessor, see MSR (immediate).

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, TCO

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b111

```
if PSTATE.EL == EL0 then
    return Zeros(38):PSTATE.TCO:Zeros(25);
elsif PSTATE.EL == EL1 then
    return Zeros(38):PSTATE.TCO:Zeros(25);
elsif PSTATE.EL == EL2 then
    return Zeros(38):PSTATE.TCO:Zeros(25);
elsif PSTATE.EL == EL3 then
    return Zeros(38):PSTATE.TCO:Zeros(25);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR TCO, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b111

```
if PSTATE.EL == EL0 then
    PSTATE.TCO = X[t]<25>;
elsif PSTATE.EL == EL1 then
    PSTATE.TCO = X[t]<25>;
elsif PSTATE.EL == EL2 then
    PSTATE.TCO = X[t]<25>;
elsif PSTATE.EL == EL3 then
    PSTATE.TCO = X[t]<25>;
```

MSR TCO, #<imm>

op0	op1	CRn	op2
0b00	0b011	0b0100	0b100

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.2.25 UAO, User Access Override

The UAO characteristics are:

Purpose

Allows access to the User Access Override bit.

Configurations

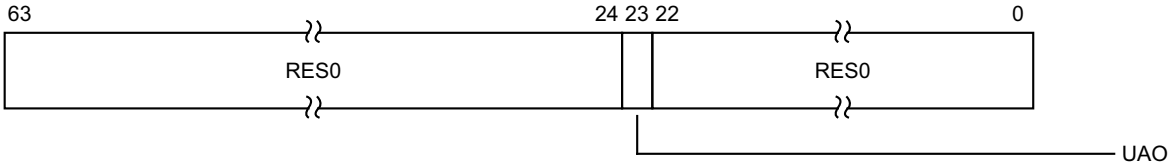
This register is present only when ARMv8.2-UAO is implemented. Otherwise, direct accesses to UAO are UNDEFINED.

Attributes

UAO is a 64-bit register.

Field descriptions

The UAO bit assignments are:



Bits [63:24]

Reserved, RES0.

UAO, bit [23]

User Access Override.

`0b0` The behavior of LDTR* and STTR* instructions is as defined in the base Armv8 architecture.

`0b1` When executed at EL1, or at EL2 with HCR_EL2.{E2H, TGE} == {1, 1}, LDTR* and STTR* instructions behave as the equivalent `LDR*` and `STR*` instructions.

When executed at EL3, or at EL2 with HCR_EL2.E2H == 0 or HCR_EL2.TGE == 0, the LDTR* and STTR* instructions behave as the equivalent `LDR*` and `STR*` instructions, regardless of the setting of the PSTATE.UAO bit.

Bits [22:0]

Reserved, RES0.

Accessing the UAO

For details on the operation of the MSR (immediate) accessor, see MSR (immediate)in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, UAO

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0010	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    return Zeros(40):PSTATE.UAO:Zeros(23);
elsif PSTATE.EL == EL2 then
    return Zeros(40):PSTATE.UAO:Zeros(23);
elsif PSTATE.EL == EL3 then
    return Zeros(40):PSTATE.UAO:Zeros(23);
```

MSR UAO, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0010	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    PSTATE.UAO = X[t]<23>;
elsif PSTATE.EL == EL2 then
    PSTATE.UAO = X[t]<23>;
elsif PSTATE.EL == EL3 then
    PSTATE.UAO = X[t]<23>;
```

MSR UAO, #<imm>

op0	op1	CRn	op2
0b00	0b000	0b0100	0b011

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.3 A64 System instructions for cache maintenance

This section lists the A64 System instructions for cache maintenance.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.3.1 DC CGDSW, Data, Allocation Tag or unified Cache line Clean of Data and Allocation Tags by Set/Way

The DC CGDSW characteristics are:

Purpose

Clean data and Allocation Tags in data cache by set/way.

Configurations

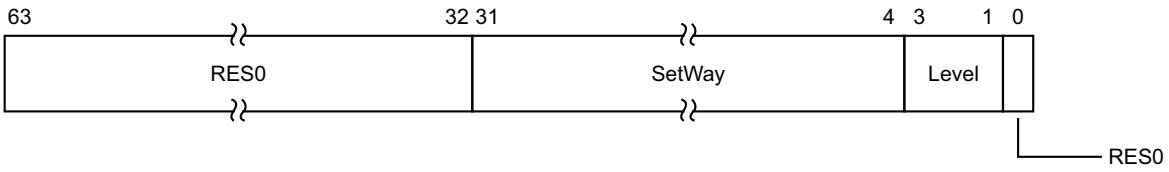
This instruction is present only when ARMv8.5-MemTag is implemented and ID_AA64PFR1_EL1.MTE != 0b0001. Otherwise, direct accesses to DC CGDSW are UNDEFINED.

Attributes

DC CGDSW is a 64-bit System instruction.

Field descriptions

The DC CGDSW input value bit assignments are:



Bits [63:32]

Reserved, RES0.

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log_2(ASSOCIATIVITY), L = Log_2(LINELEN), B = (L + S), S = Log_2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing the DC CGDSW instruction

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— A single arbitrary cache line.

— Multiple arbitrary cache lines.

Accesses to this register use the following encodings in the System instruction encoding space:

DC CGDSW, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1010	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TSW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.DCCSW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_CGDSW(X[t]);
elsif PSTATE.EL == EL2 then
    DC_CGDSW(X[t]);
elsif PSTATE.EL == EL3 then
    DC_CGDSW(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.3.2 DC CGDVAC, Data, Allocation Tag or unified Cache line Clean of Allocation Tags by VA to PoC

The DC CGDVAC characteristics are:

Purpose

Clean data and Allocation Tags in data cache by address to Point of Coherency.

Configurations

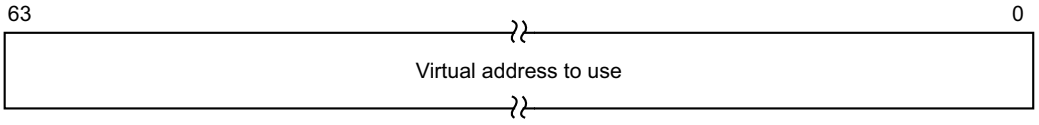
This instruction is present only when ARMv8.5-MemTag is implemented. Otherwise, direct accesses to DC CGDVAC are UNDEFINED.

Attributes

DC CGDVAC is a 64-bit System instruction.

Field descriptions

The DC CGDVAC input value bit assignments are:



Bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing the DC CGDVAC instruction

If EL0 access is enabled, when executed at EL0, this instruction requires read access permission to the VA, otherwise it generates a Permission Fault, subject to the constraints described in *Permission fault* on page D5-2653.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see *The data cache maintenance instruction (DC)* on page D4-2511.

Accesses to this register use the following encodings in the System instruction encoding space:

DC CGDVAC, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1010	0b101

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.UCI == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.UCI == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_CGDVAC(X[t]);
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TPCP == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.DCCVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_CGDVAC(X[t]);
elsif PSTATE.EL == EL2 then
    DC_CGDVAC(X[t]);
elsif PSTATE.EL == EL3 then
    DC_CGDVAC(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.3.3 DC CGDVADP, Data, Allocation Tag or unified Cache line Clean of Allocation Tags by VA to PoDP

The DC CGDVADP characteristics are:

Purpose

Clean Allocation Tags and data in data cache by address to Point of Deep Persistence.

If the memory system does not identify a Point of Deep Persistence, then this instruction behaves as a DC CGDVAP.

Configurations

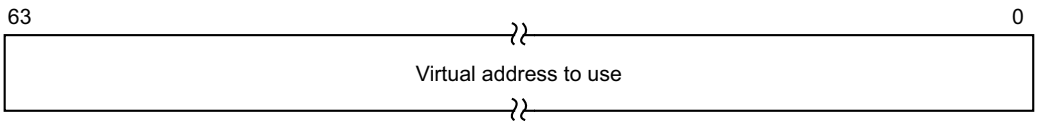
This instruction is present only when ARMv8.2-DCCVADP is implemented and ARMv8.5-MemTag is implemented. Otherwise, direct accesses to DC CGDVADP are UNDEFINED.

Attributes

DC CGDVADP is a 64-bit System instruction.

Field descriptions

The DC CGDVADP input value bit assignments are:



Bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing the DC CGDVADP instruction

If EL0 access is enabled, when executed at EL0, this instruction requires read access permission to the VA, otherwise it generates a Permission Fault, see *Permission fault* on page D5-2653.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see *The data cache maintenance instruction (DC)* on page D4-2511.

Accesses to this register use the following encodings in the System instruction encoding space:

DC CGDVADP, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1101	0b101

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.UCI == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCVADP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.UCI == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        DC_CGDVADP(X[t]);
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.DCCVADP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_CGDVADP(X[t]);
elsif PSTATE.EL == EL2 then
    DC_CGDVADP(X[t]);
elsif PSTATE.EL == EL3 then
    DC_CGDVADP(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.3.4 DC CGDVAP, Data, Allocation Tag or unified Cache line Clean of Data and Allocation Tags by VA to PoP

The DC CGDVAP characteristics are:

Purpose

Clean data and Allocation Tags in data cache by address to Point of Persistence.

If the memory system does not identify a Point of Persistence, then this instruction behaves as a DC CGDVAC.

Configurations

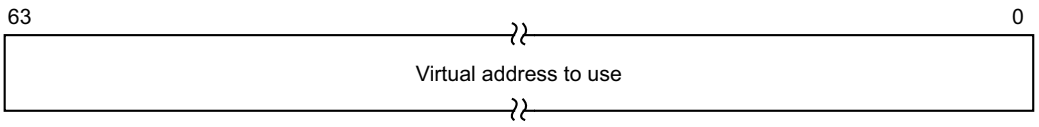
This instruction is present only when ARMv8.5-MemTag is implemented. Otherwise, direct accesses to DC CGDVAP are UNDEFINED.

Attributes

DC CGDVAP is a 64-bit System instruction.

Field descriptions

The DC CGDVAP input value bit assignments are:



Bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing the DC CGDVAP instruction

If EL0 access is enabled, when executed at EL0, this instruction requires read access permission to the VA, otherwise it generates a Permission Fault, see *Permission fault* on page D5-2653.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see *The data cache maintenance instruction (DC)* on page D4-2511.

Accesses to this register use the following encodings in the System instruction encoding space:

DC CGDVAP, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1100	0b101

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.UCI == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCVAP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.UCI == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        DC_CGDVAP(X[t]);
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.DCCVAP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_CGDVAP(X[t]);
elsif PSTATE.EL == EL2 then
    DC_CGDVAP(X[t]);
elsif PSTATE.EL == EL3 then
    DC_CGDVAP(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.3.5 DC CGSW, Data, Allocation Tag or unified Cache line Clean of Allocation Tags by Set/Way

The DC CGSW characteristics are:

Purpose

Clean Allocation Tags in data cache by set/way.

Configurations

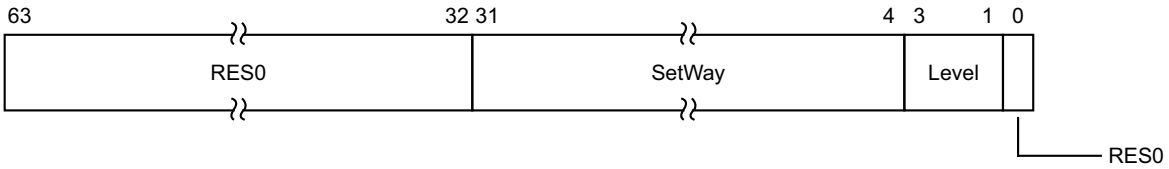
This instruction is present only when ARMv8.5-MemTag is implemented and ID_AA64PFR1_EL1.MTE != 0b0001. Otherwise, direct accesses to DC CGSW are UNDEFINED.

Attributes

DC CGSW is a 64-bit System instruction.

Field descriptions

The DC CGSW input value bit assignments are:



Bits [63:32]

Reserved, RES0.

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log_2(ASSOCIATIVITY), L = Log_2(LINELEN), B = (L + S), S = Log_2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing the DC CGSW instruction

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— Multiple arbitrary cache lines.

Accesses to this register use the following encodings in the System instruction encoding space:

DC CGSW, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1010	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TSW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.DCCSW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_CGSW(X[t]);
elsif PSTATE.EL == EL2 then
    DC_CGSW(X[t]);
elsif PSTATE.EL == EL3 then
    DC_CGSW(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.3.6 DC CGVAC, Data, Allocation Tag or unified Cache line Clean of Allocation Tags by VA to PoC

The DC CGVAC characteristics are:

Purpose

Clean Allocation Tags in data cache by address to Point of Coherency.

Configurations

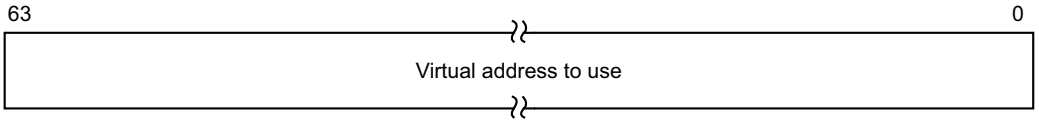
This instruction is present only when ARMv8.5-MemTag is implemented. Otherwise, direct accesses to DC CGVAC are UNDEFINED.

Attributes

DC CGVAC is a 64-bit System instruction.

Field descriptions

The DC CGVAC input value bit assignments are:



Bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing the DC CGVAC instruction

If EL0 access is enabled, when executed at EL0, this instruction requires read access permission to the VA, otherwise it generates a Permission Fault, subject to the constraints described in *Permission fault* on page D5-2653.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see *The data cache maintenance instruction (DC)* on page D4-2511.

Accesses to this register use the following encodings in the System instruction encoding space:

DC CGVAC, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1010	0b011

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.UCI == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.UCI == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_CGVAC(X[t]);
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TPCP == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.DCCVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_CGVAC(X[t]);
elsif PSTATE.EL == EL2 then
    DC_CGVAC(X[t]);
elsif PSTATE.EL == EL3 then
    DC_CGVAC(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.3.7 DC CGVADP, Data, Allocation Tag or unified Cache line Clean of Data and Allocation Tags by VA to PoDP

The DC CGVADP characteristics are:

Purpose

Clean data and Allocation Tags in data cache by address to Point of Deep Persistence.

If the memory system does not identify a Point of Deep Persistence, then this instruction behaves as a DC CGDVAP.

Configurations

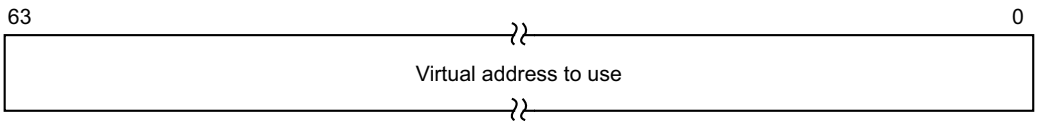
This instruction is present only when ARMv8.2-DCCVADP is implemented and ARMv8.5-MemTag is implemented. Otherwise, direct accesses to DC CGVADP are UNDEFINED.

Attributes

DC CGVADP is a 64-bit System instruction.

Field descriptions

The DC CGVADP input value bit assignments are:



Bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing the DC CGVADP instruction

If EL0 access is enabled, when executed at EL0, this instruction requires read access permission to the VA, otherwise it generates a Permission Fault, see *Permission fault* on page D5-2653.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see *The data cache maintenance instruction (DC)* on page D4-2511.

Accesses to this register use the following encodings in the System instruction encoding space:

DC CGVADP, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1101	0b011

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.UCI == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCVADP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.UCI == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        DC_CGVADP(X[t]);
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.DCCVADP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_CGVADP(X[t]);
elsif PSTATE.EL == EL2 then
    DC_CGVADP(X[t]);
elsif PSTATE.EL == EL3 then
    DC_CGVADP(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.3.8 DC CGVAP, Data, Allocation Tag or unified Cache line Clean of Allocation Tags by VA to PoP

The DC CGVAP characteristics are:

Purpose

Clean Allocation Tags in data cache by address to Point of Persistence.

If the memory system does not identify a Point of Persistence, then this instruction behaves as a DC CGVAC.

Configurations

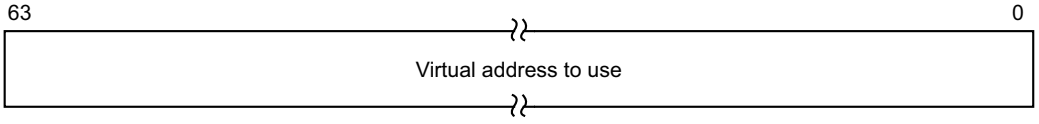
This instruction is present only when ARMv8.5-MemTag is implemented. Otherwise, direct accesses to DC CGVAP are UNDEFINED.

Attributes

DC CGVAP is a 64-bit System instruction.

Field descriptions

The DC CGVAP input value bit assignments are:



Bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing the DC CGVAP instruction

If EL0 access is enabled, when executed at EL0, this instruction requires read access permission to the VA, otherwise it generates a Permission Fault, see *Permission fault* on page D5-2653.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see *The data cache maintenance instruction (DC)* on page D4-2511.

Accesses to this register use the following encodings in the System instruction encoding space:

DC CGVAP, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1100	0b011

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.UCI == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCVAP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.UCI == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_CGVAP(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.DCCVAP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_CGVAP(X[t]);
elsif PSTATE.EL == EL2 then
    DC_CGVAP(X[t]);
elsif PSTATE.EL == EL3 then
    DC_CGVAP(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.3.9 DC CIGDSW, Data, Allocation Tag or unified Cache line Clean and Invalidate of Data and Allocation Tags by Set/Way

The DC CIGDSW characteristics are:

Purpose

Clean and Invalidate data and Allocation Tags in data cache by set/way.

Configurations

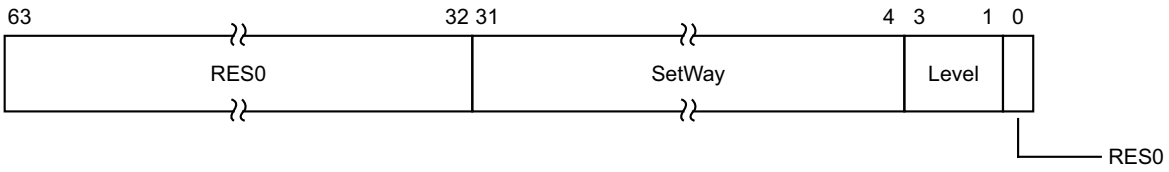
This instruction is present only when ARMv8.5-MemTag is implemented and ID_AA64PFR1_EL1.MTE != 0b0001. Otherwise, direct accesses to DC CIGDSW are UNDEFINED.

Attributes

DC CIGDSW is a 64-bit System instruction.

Field descriptions

The DC CIGDSW input value bit assignments are:



Bits [63:32]

Reserved, RES0.

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log_2(ASSOCIATIVITY), L = Log_2(LINELEN), B = (L + S), S = Log_2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing the DC CIGDSW instruction

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— A single arbitrary cache line.

— Multiple arbitrary cache lines.

Accesses to this register use the following encodings in the System instruction encoding space:

DC CIGDSW, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1110	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TSW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.DCCISW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_CIGDSW(X[t]);
elsif PSTATE.EL == EL2 then
    DC_CIGDSW(X[t]);
elsif PSTATE.EL == EL3 then
    DC_CIGDSW(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.3.10 DC CIGDVAC, Data, Allocation Tag or unified Cache line Clean and Invalidate of Data and Allocation Tags by VA to PoC

The DC CIGDVAC characteristics are:

Purpose

Clean and Invalidate data and Allocation Tags in data cache by address to Point of Coherency.

Configurations

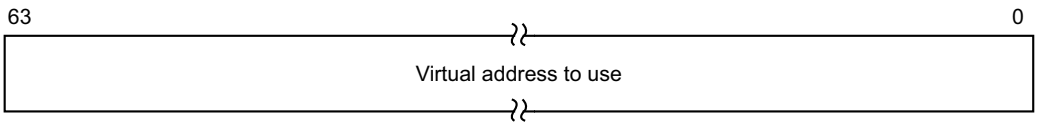
This instruction is present only when ARMv8.5-MemTag is implemented. Otherwise, direct accesses to DC CIGDVAC are UNDEFINED.

Attributes

DC CIGDVAC is a 64-bit System instruction.

Field descriptions

The DC CIGDVAC input value bit assignments are:



Bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing the DC CIGDVAC instruction

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see *The data cache maintenance instruction (DC)* on page D4-2511.

If EL0 access is enabled, when executed at EL0, this instruction requires read access permission to the VA, otherwise it generates a Permission Fault, subject to the constraints described in *Permission fault* on page D5-2653.

Accesses to this register use the following encodings in the System instruction encoding space:

DC CIGDVAC, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1110	0b101

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.UCI == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCIVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.UCI == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_CIGDVAC(X[t]);
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.DCCIVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_CIGDVAC(X[t]);
elsif PSTATE.EL == EL2 then
    DC_CIGDVAC(X[t]);
elsif PSTATE.EL == EL3 then
    DC_CIGDVAC(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.3.11 DC CIGSW, Data, Allocation Tag or unified Cache line Clean and Invalidate of Allocation Tags by Set/Way

The DC CIGSW characteristics are:

Purpose

Clean and Invalidate Allocation Tags in data cache by set/way.

Configurations

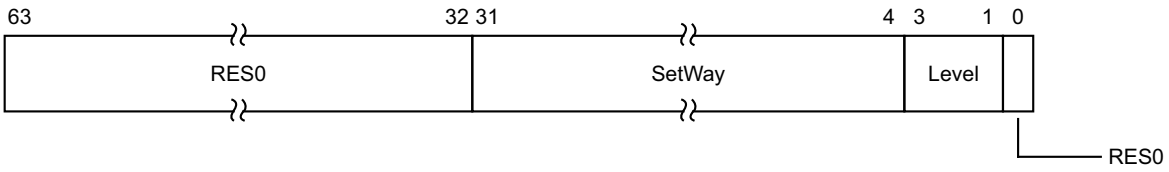
This instruction is present only when ARMv8.5-MemTag is implemented and ID_AA64PFR1_EL1.MTE != 0b0001. Otherwise, direct accesses to DC CIGSW are UNDEFINED.

Attributes

DC CIGSW is a 64-bit System instruction.

Field descriptions

The DC CIGSW input value bit assignments are:



Bits [63:32]

Reserved, RES0.

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log_2(ASSOCIATIVITY), L = Log_2(LINELEN), B = (L + S), S = Log_2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing the DC CIGSW instruction

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— A single arbitrary cache line.

— Multiple arbitrary cache lines.

Accesses to this register use the following encodings in the System instruction encoding space:

DC CIGSW, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1110	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TSW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.DCCISW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_CIGSW(X[t]);
elsif PSTATE.EL == EL2 then
    DC_CIGSW(X[t]);
elsif PSTATE.EL == EL3 then
    DC_CIGSW(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.3.12 DC CIGVAC, Data, Allocation Tag or unified Cache line Clean and Invalidate of Allocation Tags by VA to PoC

The DC CIGVAC characteristics are:

Purpose

Clean and Invalidate Allocation Tags in data cache by address to Point of Coherency.

Configurations

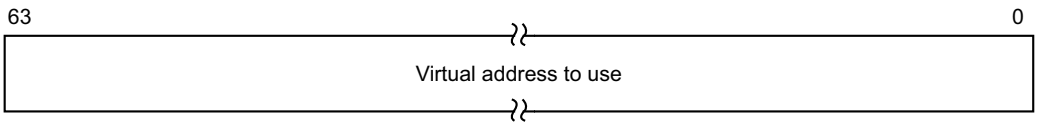
This instruction is present only when ARMv8.5-MemTag is implemented. Otherwise, direct accesses to DC CIGVAC are UNDEFINED.

Attributes

DC CIGVAC is a 64-bit System instruction.

Field descriptions

The DC CIGVAC input value bit assignments are:



Bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing the DC CIGVAC instruction

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see *The data cache maintenance instruction (DC)* on page D4-2511.

If EL0 access is enabled, when executed at EL0, this instruction requires read access permission to the VA, otherwise it generates a Permission Fault, subject to the constraints described in *Permission fault* on page D5-2653.

Accesses to this register use the following encodings in the System instruction encoding space:

DC CIGVAC, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1110	0b011

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.UCI == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCIVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.UCI == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_CIGVAC(X[t]);
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.DCCIVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_CIGVAC(X[t]);
elsif PSTATE.EL == EL2 then
    DC_CIGVAC(X[t]);
elsif PSTATE.EL == EL3 then
    DC_CIGVAC(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.3.13 DC CISW, Data or unified Cache line Clean and Invalidate by Set/Way

The DC CISW characteristics are:

Purpose

Clean and Invalidate data cache by set/way.

When ARMv8.5-MemTag is implemented, this instruction might clean and invalidate Allocation Tags from caches.

Configurations

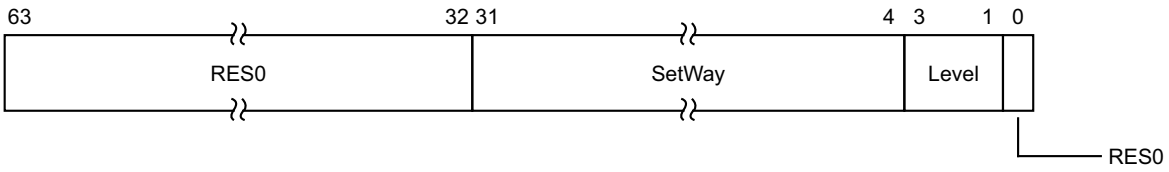
AArch64 System instruction DC CISW performs the same function as AArch32 System instruction DCCISW.

Attributes

DC CISW is a 64-bit System instruction.

Field descriptions

The DC CISW input value bit assignments are:



Bits [63:32]

Reserved, RES0.

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log_2(ASSOCIATIVITY), L = Log_2(LINELEN), B = (L + S), S = Log_2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing the DC CISW instruction

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

Accesses to this register use the following encodings in the System instruction encoding space:

DC CISW, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1110	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TSW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.DCCISW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_CISW(X[t]);
elsif PSTATE.EL == EL2 then
    DC_CISW(X[t]);
elsif PSTATE.EL == EL3 then
    DC_CISW(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.3.14 DC CIVAC, Data or unified Cache line Clean and Invalidate by VA to PoC

The DC CIVAC characteristics are:

Purpose

Clean and Invalidate data cache by address to Point of Coherency.

When ARMv8.5-MemTag is implemented, this instruction might clean and invalidate Allocation Tags from caches.

Configurations

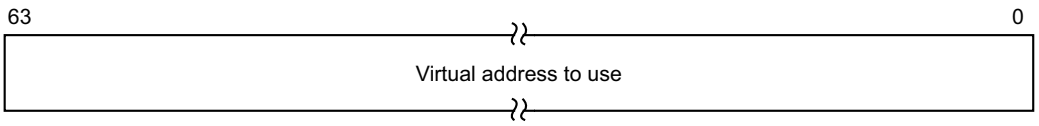
AArch64 System instruction DC CIVAC performs the same function as AArch32 System instruction DCCIMVAC.

Attributes

DC CIVAC is a 64-bit System instruction.

Field descriptions

The DC CIVAC input value bit assignments are:



Bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing the DC CIVAC instruction

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see *The data cache maintenance instruction (DC)* on page D4-2511.

If EL0 access is enabled, when executed at EL0, this instruction requires read access permission to the VA, otherwise it generates a Permission Fault, subject to the constraints described in *Permission fault* on page D5-2653.

Accesses to this register use the following encodings in the System instruction encoding space:

DC CIVAC, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1110	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.UCI == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCIVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.UCI == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_CIVAC(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.DCCIVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_CIVAC(X[t]);
elsif PSTATE.EL == EL2 then
    DC_CIVAC(X[t]);
elsif PSTATE.EL == EL3 then
    DC_CIVAC(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.3.15 DC CSW, Data or unified Cache line Clean by Set/Way

The DC CSW characteristics are:

Purpose

Clean data cache by set/way.

When ARMv8.5-MemTag is implemented, this instruction might clean Allocation Tags from caches.

Configurations

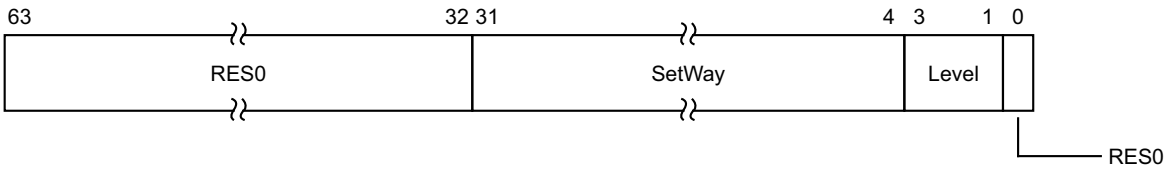
AArch64 System instruction DC CSW performs the same function as AArch32 System instruction DCCSW.

Attributes

DC CSW is a 64-bit System instruction.

Field descriptions

The DC CSW input value bit assignments are:



Bits [63:32]

Reserved, RES0.

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log_2(ASSOCIATIVITY), L = Log_2(LINELEN), B = (L + S), S = Log_2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing the DC CSW instruction

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

Accesses to this register use the following encodings in the System instruction encoding space:

DC CSW, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1010	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TSW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.DCCSW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_CSW(X[t]);
elsif PSTATE.EL == EL2 then
    DC_CSW(X[t]);
elsif PSTATE.EL == EL3 then
    DC_CSW(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.3.16 DC CVAC, Data or unified Cache line Clean by VA to PoC

The DC CVAC characteristics are:

Purpose

Clean data cache by address to Point of Coherency.

When ARMv8.5-MemTag is implemented, this instruction might clean Allocation Tags from caches.

Configurations

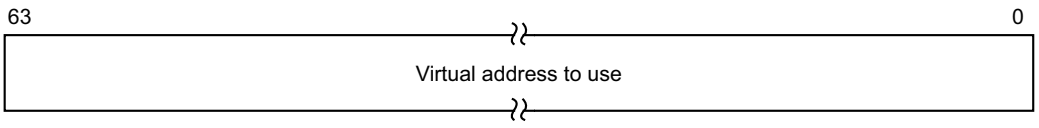
AArch64 System instruction DC CVAC performs the same function as AArch32 System instruction DCCMVAC.

Attributes

DC CVAC is a 64-bit System instruction.

Field descriptions

The DC CVAC input value bit assignments are:



Bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing the DC CVAC instruction

If EL0 access is enabled, when executed at EL0, this instruction requires read access permission to the VA, otherwise it generates a Permission Fault, subject to the constraints described in *Permission fault* on page D5-2653.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see *The data cache maintenance instruction (DC)* on page D4-2511.

Accesses to this register use the following encodings in the System instruction encoding space:

DC CVAC, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1010	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.UCI == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.UCI == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_CVAC(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.DCCVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_CVAC(X[t]);
elsif PSTATE.EL == EL2 then
    DC_CVAC(X[t]);
elsif PSTATE.EL == EL3 then
    DC_CVAC(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.3.17 DC CVADP, Data or unified Cache line Clean by VA to PoDP

The DC CVADP characteristics are:

Purpose

Clean data cache by address to Point of Deep Persistence.

If the memory system does not identify a Point of Deep Persistence, then this instruction behaves as a DC CVAP.

When ARMv8.5-MemTag is implemented, this instruction might clean Allocation Tags from caches.

Configurations

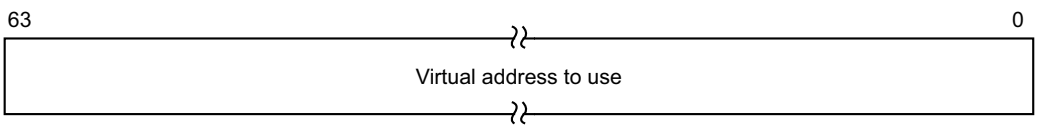
This instruction is present only when ARMv8.2-DCCVADP is implemented. Otherwise, direct accesses to DC CVADP are UNDEFINED.

Attributes

DC CVADP is a 64-bit System instruction.

Field descriptions

The DC CVADP input value bit assignments are:



Bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing the DC CVADP instruction

If EL0 access is enabled, when executed at EL0, this instruction requires read access permission to the VA, otherwise it generates a Permission Fault, see *Permission fault* on page D5-2653.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see *The data cache maintenance instruction (DC)* on page D4-2511.

Accesses to this register use the following encodings in the System instruction encoding space:

DC CVADP, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1101	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.UCI == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCVADP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.UCI == '0' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_CVADP(X[t]);
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.DCCVADP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_CVADP(X[t]);
elsif PSTATE.EL == EL2 then
    DC_CVADP(X[t]);
elsif PSTATE.EL == EL3 then
    DC_CVADP(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.3.18 DC CVAP, Data or unified Cache line Clean by VA to PoP

The DC CVAP characteristics are:

Purpose

Clean data cache by address to Point of Persistence.

If the memory system does not identify a Point of Persistence, then this instruction behaves as a DC CVAC.

When ARMv8.5-MemTag is implemented, this instruction might clean Allocation Tags from caches.

Configurations

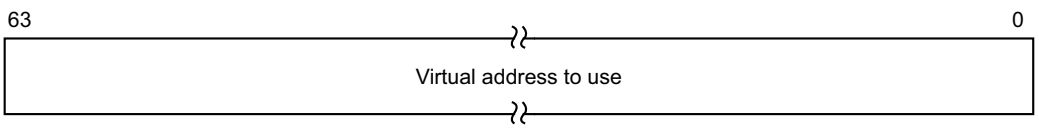
This instruction is present only when ARMv8.2-DCPoP is implemented. Otherwise, direct accesses to DC CVAP are UNDEFINED.

Attributes

DC CVAP is a 64-bit System instruction.

Field descriptions

The DC CVAP input value bit assignments are:



Bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing the DC CVAP instruction

If EL0 access is enabled, when executed at EL0, this instruction requires read access permission to the VA, otherwise it generates a Permission Fault, see *Permission fault* on page D5-2653.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see *The data cache maintenance instruction (DC)* on page D4-2511.

Accesses to this register use the following encodings in the System instruction encoding space:

DC CVAP, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1100	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.UCI == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCVAP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.UCI == '0' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_CVAP(X[t]);
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.DCCVAP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_CVAP(X[t]);
elsif PSTATE.EL == EL2 then
    DC_CVAP(X[t]);
elsif PSTATE.EL == EL3 then
    DC_CVAP(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.3.19 DC CVAU, Data or unified Cache line Clean by VA to PoU

The DC CVAU characteristics are:

Purpose

Clean data cache by address to Point of Unification.

Configurations

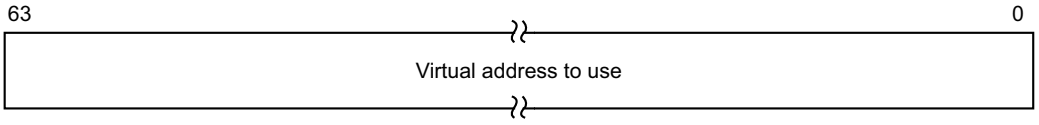
AArch64 System instruction DC CVAU performs the same function as AArch32 System instruction DCCMVAU.

Attributes

DC CVAU is a 64-bit System instruction.

Field descriptions

The DC CVAU input value bit assignments are:



Bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing the DC CVAU instruction

If EL0 access is enabled, when executed at EL0, this instruction requires read access permission to the VA, otherwise it generates a Permission Fault, subject to the constraints described in *Permission fault* on page D5-2653.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see *The data cache maintenance instruction (DC)* on page D4-2511.

Accesses to this register use the following encodings in the System instruction encoding space:

DC CVAU, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1011	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.UCI == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TPU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TOCU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCVAU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.UCI == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_CVAU(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TPU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TOCU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.DCCVAU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_CVAU(X[t]);
elsif PSTATE.EL == EL2 then
    DC_CVAU(X[t]);
elsif PSTATE.EL == EL3 then
    DC_CVAU(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.3.20 DC GVA, Data Cache set Allocation Tag by VA

The DC GVA characteristics are:

Purpose

Write a value to the Allocation Tags of a naturally aligned block of N bytes, where the size of N is identified in DCZID_EL0. The Allocation Tag used is determined by the input address.

Configurations

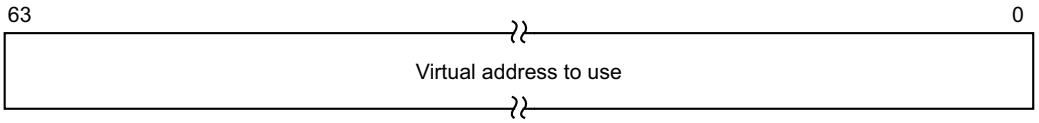
This instruction is present only when ARMv8.5-MemTag is implemented. Otherwise, direct accesses to DC GVA are UNDEFINED.

Attributes

DC GVA is a 64-bit System instruction.

Field descriptions

The DC GVA input value bit assignments are:



Bits [63:0]

Virtual address to use. There is no alignment restriction on the address within the block of N bytes that is used.

Executing the DC GVA instruction

When this instruction is executed, it can generate memory faults or watchpoints which are prioritized in the same way as other memory-related faults or watchpoints. If a synchronous data abort fault or a watchpoint is generated, the CM bit in the ESR_ELx.ISS field is not set.

If the memory region being zeroed is any type of Device memory, this instruction can give an alignment fault which is prioritized in the same way as other alignment faults that are determined by the memory type.

This instruction applies to Normal memory regardless of cacheability attributes.

This instruction behaves as a set of Stores to each byte within the block being accessed, and so it:

- Generates a Permission Fault if the translation system does not permit writes to the locations.
- Requires the same considerations for ordering and the management of coherency as any other store instructions.

Accesses to this register use the following encodings in the System instruction encoding space:

DC GVA, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0100	0b011

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.DZE == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TDZ == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCZVA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.DZE == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_GVA(X[t]);
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TDZ == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.DCZVA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_GVA(X[t]);
elsif PSTATE.EL == EL2 then
    DC_GVA(X[t]);
elsif PSTATE.EL == EL3 then
    DC_GVA(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.3.21 DC GZVA, Data Cache set Allocation Tags and Zero by VA

The DC GZVA characteristics are:

Purpose

Zero data and write a value to the Allocation Tags of a naturally aligned block of N bytes, where the size of N is identified in DCZID_EL0. The Allocation Tag used is determined by the input address.

Configurations

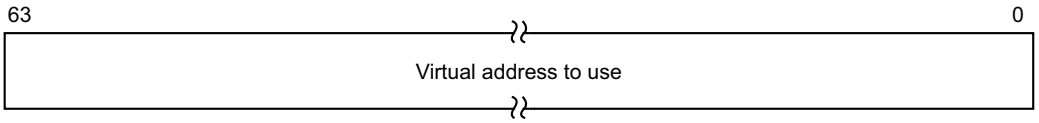
This instruction is present only when ARMv8.5-MemTag is implemented. Otherwise, direct accesses to DC GZVA are UNDEFINED.

Attributes

DC GZVA is a 64-bit System instruction.

Field descriptions

The DC GZVA input value bit assignments are:



Bits [63:0]

Virtual address to use. There is no alignment restriction on the address within the block of N bytes that is used.

Executing the DC GZVA instruction

When this instruction is executed, it can generate memory faults or watchpoints which are prioritized in the same way as other memory-related faults or watchpoints. If a synchronous data abort fault or a watchpoint is generated, the CM bit in the ESR_ELx.ISS field is not set.

If the memory region being zeroed is any type of Device memory, this instruction can give an alignment fault which is prioritized in the same way as other alignment faults that are determined by the memory type.

This instruction applies to Normal memory regardless of cacheability attributes.

This instruction behaves as a set of Stores to each byte within the block being accessed, and so it:

- Generates a Permission Fault if the translation system does not permit writes to the locations.
- Requires the same considerations for ordering and the management of coherency as any other store instructions.

Accesses to this register use the following encodings in the System instruction encoding space:

DC GZVA, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0100	0b100

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.DZE == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TDZ == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCZVA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.DZE == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_GZVA(X[t]);
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TDZ == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.DCZVA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_GZVA(X[t]);
elsif PSTATE.EL == EL2 then
    DC_GZVA(X[t]);
elsif PSTATE.EL == EL3 then
    DC_GZVA(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.3.22 DC IGDSW, Data, Allocation Tag or unified Cache line Invalidate of Data and Allocation Tags by Set/Way

The DC IGDSW characteristics are:

Purpose

Invalidate data and Allocation Tags in data cache by set/way.

Configurations

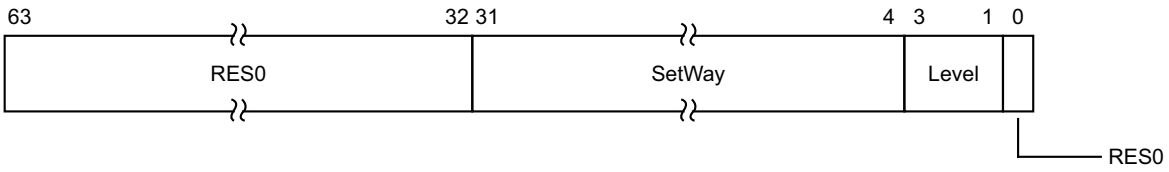
This instruction is present only when ARMv8.5-MemTag is implemented and ID_AA64PFR1_EL1.MTE != 0b0001. Otherwise, direct accesses to DC IGDSW are UNDEFINED.

Attributes

DC IGDSW is a 64-bit System instruction.

Field descriptions

The DC IGDSW input value bit assignments are:



Bits [63:32]

Reserved, RES0.

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log_2(ASSOCIATIVITY), L = Log_2(LINELEN), B = (L + S), S = Log_2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing the DC IGDSW instruction

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— A single arbitrary cache line.

— Multiple arbitrary cache lines.

Accesses to this register use the following encodings in the System instruction encoding space:

DC IGDSW, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b0110	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TSW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.DCISW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.SWIO == '1' then
        DC_CIGDSW(X[t]);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<DC,VM> != '00' then
        DC_CIGDSW(X[t]);
    else
        DC_IGDSW(X[t]);
elsif PSTATE.EL == EL2 then
    DC_IGDSW(X[t]);
elsif PSTATE.EL == EL3 then
    DC_IGDSW(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.3.23 DC IGDVAC, Data, Allocation Tag or unified Cache line Invalidate of Allocation Tags by VA to PoC

The DC IGDVAC characteristics are:

Purpose

Invalidate data and Allocation Tags in data cache by address to Point of Coherency.

Configurations

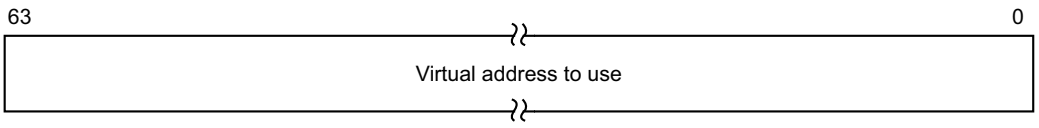
This instruction is present only when ARMv8.5-MemTag is implemented and ID_AA64PFR1_EL1.MTE != 0b0001. Otherwise, direct accesses to DC IGDVAC are UNDEFINED.

Attributes

DC IGDVAC is a 64-bit System instruction.

Field descriptions

The DC IGDVAC input value bit assignments are:



Bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing the DC IGDVAC instruction

When the instruction is executed, it can generate a watchpoint, which is prioritized in the same way as other watchpoints. If a watchpoint is generated, the CM bit in the ESR_ELx.ISS field is set to 1.

This instruction requires write access permission to the VA, otherwise it generates a Permission Fault, subject to the constraints described in *Permission fault* on page D5-2653.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see *The data cache maintenance instruction (DC)* on page D4-2511.

Accesses to this register use the following encodings in the System instruction encoding space:

DC IGDVAC, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b0110	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.DCIVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<DC,VM> != '00' then
        DC_CIGDVAC(X[t]);
    else
        DC_IGDVAC(X[t]);
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    DC_IGDVAC(X[t]);
elsif PSTATE.EL == EL3 then
    DC_IGDVAC(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.3.24 DC IGSW, Data, Allocation Tag or unified Cache line Invalidate of Allocation Tags by Set/Way

The DC IGSW characteristics are:

Purpose

Invalidate Allocation Tags in data cache by set/way.

Configurations

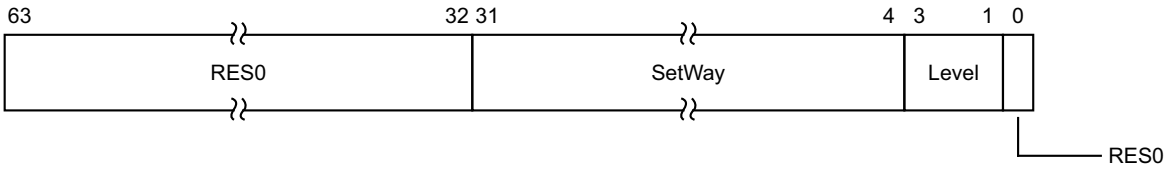
This instruction is present only when ARMv8.5-MemTag is implemented and ID_AA64PFR1_EL1.MTE != 0b0001. Otherwise, direct accesses to DC IGSW are UNDEFINED.

Attributes

DC IGSW is a 64-bit System instruction.

Field descriptions

The DC IGSW input value bit assignments are:



Bits [63:32]

Reserved, RES0.

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log_2(ASSOCIATIVITY), L = Log_2(LINELEN), B = (L + S), S = Log_2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing the DC IGSW instruction

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— Multiple arbitrary cache lines.

Accesses to this register use the following encodings in the System instruction encoding space:

DC IGSW, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b0110	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TSW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.DCISW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.SWIO == '1' then
        DC_CIGSW(X[t]);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<DC,VM> != '00' then
        DC_CIGSW(X[t]);
    else
        DC_IGSW(X[t]);
elsif PSTATE.EL == EL2 then
    DC_IGSW(X[t]);
elsif PSTATE.EL == EL3 then
    DC_IGSW(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.3.25 DC IGVAC, Data, Allocation Tag or unified Cache line Invalidate of Allocation Tags by VA to PoC

The DC IGVAC characteristics are:

Purpose

Invalidate Allocation Tags in data cache by address to Point of Coherency.

Configurations

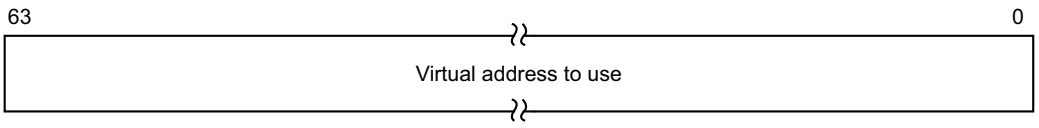
This instruction is present only when ARMv8.5-MemTag is implemented and ID_AA64PFR1_EL1.MTE != 0b0001. Otherwise, direct accesses to DC IGVAC are UNDEFINED.

Attributes

DC IGVAC is a 64-bit System instruction.

Field descriptions

The DC IGVAC input value bit assignments are:



Bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing the DC IGVAC instruction

When the instruction is executed, it can generate a watchpoint, which is prioritized in the same way as other watchpoints. If a watchpoint is generated, the CM bit in the ESR_ELx.ISS field is set to 1.

This instruction requires write access permission to the VA, otherwise it generates a Permission Fault, subject to the constraints described in *Permission fault* on page D5-2653.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see *The data cache maintenance instruction (DC)* on page D4-2511.

Accesses to this register use the following encodings in the System instruction encoding space:

DC IGVAC, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b0110	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.DCIVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<DC,VM> != '00' then
        DC_CIGVAC(X[t]);
    else
        DC_IGVAC(X[t]);
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    DC_IGVAC(X[t]);
elsif PSTATE.EL == EL3 then
    DC_IGVAC(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.3.26 DC ISW, Data or unified Cache line Invalidate by Set/Way

The DC ISW characteristics are:

Purpose

Invalidate data cache by set/way.

When ARMv8.5-MemTag is implemented, this instruction might invalidate Allocation Tags from caches. When it invalidates Allocation Tags from caches, it also cleans them.

Configurations

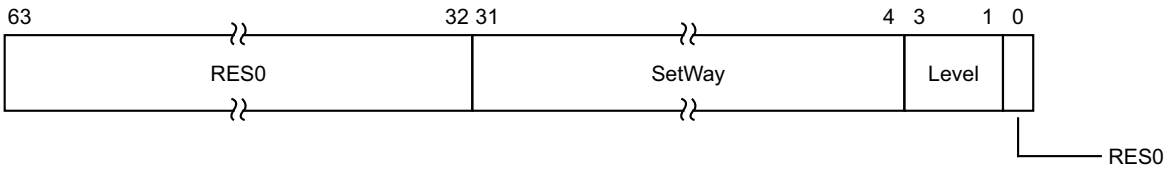
AArch64 System instruction DC ISW performs the same function as AArch32 System instruction DCISW.

Attributes

DC ISW is a 64-bit System instruction.

Field descriptions

The DC ISW input value bit assignments are:



Bits [63:32]

Reserved, RES0.

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log_2(ASSOCIATIVITY), L = Log_2(LINELEN), B = (L + S), S = Log_2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing the DC ISW instruction

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

Accesses to this register use the following encodings in the System instruction encoding space:

DC ISW, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b0110	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TSW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.DCISW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.SWIO == '1' then
        DC_CISW(X[t]);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<DC,VM> != '00' then
        DC_CISW(X[t]);
    else
        DC_ISW(X[t]);
elsif PSTATE.EL == EL2 then
    DC_ISW(X[t]);
elsif PSTATE.EL == EL3 then
    DC_ISW(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.3.27 DC IVAC, Data or unified Cache line Invalidate by VA to PoC

The DC IVAC characteristics are:

Purpose

Invalidate data cache by address to Point of Coherency.

When ARMv8.5-MemTag is implemented, this instruction might invalidate Allocation Tags from caches. When it invalidates Allocation Tags from caches, it also cleans them.

Configurations

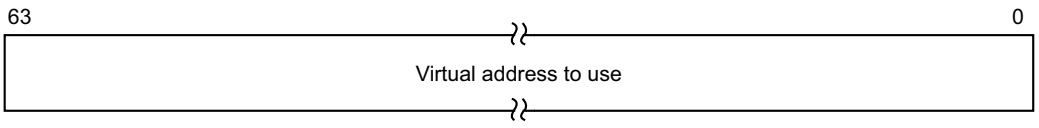
AArch64 System instruction DC IVAC performs the same function as AArch32 System instruction DCIMVAC.

Attributes

DC IVAC is a 64-bit System instruction.

Field descriptions

The DC IVAC input value bit assignments are:



Bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing the DC IVAC instruction

When the instruction is executed, it can generate a watchpoint, which is prioritized in the same way as other watchpoints. If a watchpoint is generated, the CM bit in the ESR_ELx.ISS field is set to 1.

This instruction requires write access permission to the VA, otherwise it generates a Permission Fault, subject to the constraints described in *Permission fault* on page D5-2653.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see *The data cache maintenance instruction (DC)* on page D4-2511.

Accesses to this register use the following encodings in the System instruction encoding space:

DC IVAC, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b0110	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.DCIVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<DC,VM> != '00' then
        DC_CIVAC(X[t]);
    else
        DC_IVAC(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL2 then
    DC_IVAC(X[t]);
elsif PSTATE.EL == EL3 then
    DC_IVAC(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.3.28 DC ZVA, Data Cache Zero by VA

The DC ZVA characteristics are:

Purpose

Zero data cache by address. Zeroes a naturally aligned block of N bytes, where the size of N is identified in DCZID_EL0.

Configurations

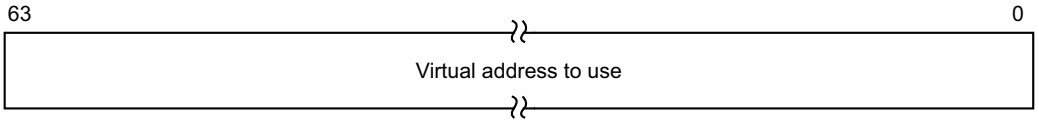
There are no configuration notes.

Attributes

DC ZVA is a 64-bit System instruction.

Field descriptions

The DC ZVA input value bit assignments are:



Bits [63:0]

Virtual address to use. There is no alignment restriction on the address within the block of N bytes that is used.

Executing the DC ZVA instruction

When this instruction is executed, it can generate memory faults or watchpoints which are prioritized in the same way as other memory-related faults or watchpoints. If a synchronous data abort fault or a watchpoint is generated, the CM bit in the ESR_ELx.ISS field is set to 0.

If the memory region being zeroed is any type of Device memory, this instruction can give an Alignment fault which is prioritized in the same way as other Alignment faults that are determined by the memory type.

This instruction applies to Normal memory regardless of cacheability attributes.

This instruction behaves as a set of Stores to each byte within the block being accessed, and so it:

- Generates a Permission Fault if the translation system does not permit writes to the locations.
- Requires the same considerations for ordering and the management of coherency as any other store instructions.

Accesses to this register use the following encodings in the System instruction encoding space:

DC ZVA, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0100	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.DZE == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TDZ == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCZVA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.DZE == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_ZVA(X[t]);
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TDZ == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.DCZVA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DC_ZVA(X[t]);
elsif PSTATE.EL == EL2 then
    DC_ZVA(X[t]);
elsif PSTATE.EL == EL3 then
    DC_ZVA(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.3.29 IC IALLU, Instruction Cache Invalidate All to PoU

The IC IALLU characteristics are:

Purpose

Invalidate all instruction caches to Point of Unification.

Configurations

AArch64 System instruction IC IALLU performs the same function as AArch32 System instruction ICIALLU.

Attributes

IC IALLU is a 64-bit System instruction.

Field descriptions

IC IALLU ignores the value in the register specified by the instruction. Software does not have to write a value to the register before issuing this instruction.

Executing the IC IALLU instruction

When executing this instruction Xt should be encoded as `0b11111`. If the Xt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Xt field is set to `0b11111`.

Accesses to this register use the following encodings in the System instruction encoding space:

IC IALLU{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b0101	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TPU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TOCU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.ICIALLU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.FB == '1' then
        IC_IALLUIS();
    else
        IC_IALLU();
elsif PSTATE.EL == EL2 then
    IC_IALLU();
elsif PSTATE.EL == EL3 then
    IC_IALLU();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.3.30 IC IALLUIS, Instruction Cache Invalidate All to PoU, Inner Shareable

The IC IALLUIS characteristics are:

Purpose

Invalidate all instruction caches in Inner Shareable domain to Point of Unification.

Configurations

AArch64 System instruction IC IALLUIS performs the same function as AArch32 System instruction ICIALLUIS.

Attributes

IC IALLUIS is a 64-bit System instruction.

Field descriptions

IC IALLUIS ignores the value in the register specified by the instruction. Software does not have to write a value to the register before issuing this instruction.

Executing the IC IALLUIS instruction

When executing this instruction Xt should be encoded as `0b11111`. If the Xt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Xt field is set to `0b11111`.

Accesses to this register use the following encodings in the System instruction encoding space:

IC IALLUIS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TPU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TICAB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.ICIALLUIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        IC_IALLUIS();
elsif PSTATE.EL == EL2 then
    IC_IALLUIS();
elsif PSTATE.EL == EL3 then
    IC_IALLUIS();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.3.31 IC IVAU, Instruction Cache line Invalidate by VA to PoU

The IC IVAU characteristics are:

Purpose

Invalidate instruction cache by address to Point of Unification.

Configurations

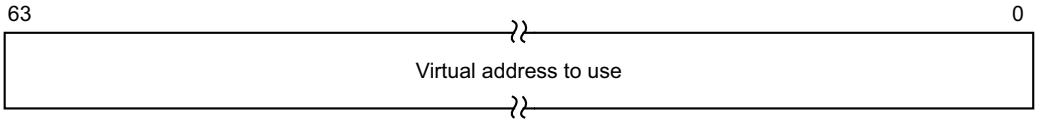
AArch64 System instruction IC IVAU performs the same function as AArch32 System instruction ICIMVAU.

Attributes

IC IVAU is a 64-bit System instruction.

Field descriptions

The IC IVAU input value bit assignments are:



Bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing the IC IVAU instruction

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see *The instruction cache maintenance instruction (IC)* on page D4-2511.

If EL0 access is enabled, when executed at EL0, this instruction requires read access permission to the VA, otherwise it is IMPLEMENTATION DEFINED whether it generates a Permission Fault, see *Permission fault* on page D5-2653.

Accesses to this register use the following encodings in the System instruction encoding space:

IC IVAU{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0101	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.UCI == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TPU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TOCU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HFGITR_EL2.ICIVAU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.UCI == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        IC_IVAU(X[t]);
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TPU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TOCU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.ICIVAU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        IC_IVAU(X[t]);
elsif PSTATE.EL == EL2 then
    IC_IVAU(X[t]);
elsif PSTATE.EL == EL3 then
    IC_IVAU(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.4 A64 System instructions for address translation

This section lists the A64 System instructions for address translation.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.4.1 AT S12E0R, Address Translate Stages 1 and 2 EL0 Read

The AT S12E0R characteristics are:

Purpose

Performs stage 1 and 2 address translations from EL0, with permissions as if reading from the given virtual address from EL0, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - — If HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
  - — If HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime.
- Otherwise, the EL1&0 translation regime.

Configurations

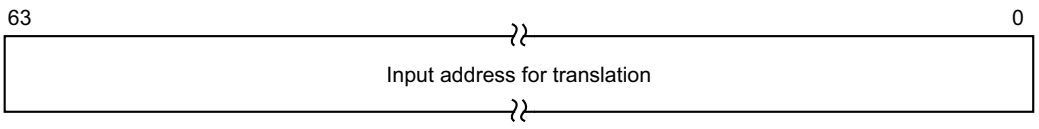
There are no configuration notes.

Attributes

AT S12E0R is a 64-bit System instruction.

Field descriptions

The AT S12E0R input value bit assignments are:



Bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing the AT S12E0R instruction

Accesses to this register use the following encodings in the System instruction encoding space:

AT S12E0R, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b100	0b0111	0b1000	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.<E2H,TGE> == '11' || HCR_EL2.<DC,VM> == '00' then
        AT_S1E0R(X[t]);
    else
        AT_S12E0R(X[t]);
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if !EL2Enabled() then
    AT_S1E0R(X[t]);
elsif EL2Enabled() && (HCR_EL2.<E2H,TGE> == '11' || HCR_EL2.<DC,VM> == '00') then
    AT_S1E0R(X[t]);
else
    AT_S12E0R(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.4.2 AT S12E0W, Address Translate Stages 1 and 2 EL0 Write

The AT S12E0W characteristics are:

Purpose

Performs stage 1 and 2 address translations from EL0, with permissions as if writing to the given virtual address from EL0, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - — If HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
  - — If HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime.
- Otherwise, the EL1&0 translation regime.

Configurations

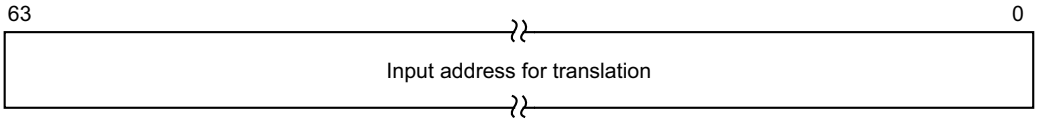
There are no configuration notes.

Attributes

AT S12E0W is a 64-bit System instruction.

Field descriptions

The AT S12E0W input value bit assignments are:



Bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing the AT S12E0W instruction

Accesses to this register use the following encodings in the System instruction encoding space:

AT S12E0W, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b100	0b0111	0b1000	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.<E2H,TGE> == '11' || HCR_EL2.<DC,VM> == '00' then
        AT_S1E0W(X[t]);
    else
        AT_S12E0W(X[t]);
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if !EL2Enabled() then
    AT_S1E0W(X[t]);
elsif EL2Enabled() && (HCR_EL2.<E2H,TGE> == '11' || HCR_EL2.<DC,VM> == '00') then
    AT_S1E0W(X[t]);
else
    AT_S12E0W(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.4.3 AT S12E1R, Address Translate Stages 1 and 2 EL1 Read

The AT S12E1R characteristics are:

Purpose

Performs stage 1 and 2 address translation, with permissions as if reading from the given virtual address from EL1, or from EL2 if the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - If HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime, accessed from EL1.
  - If HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime, accessed from EL2.
- Otherwise, the EL1&0 translation regime, accessed from EL1.

Configurations

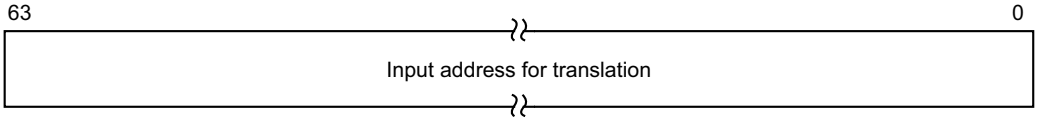
There are no configuration notes.

Attributes

AT S12E1R is a 64-bit System instruction.

Field descriptions

The AT S12E1R input value bit assignments are:



Bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing the AT S12E1R instruction

Accesses to this register use the following encodings in the System instruction encoding space:

AT S12E1R, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b100	0b0111	0b1000	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.<E2H,TGE> == '11' || HCR_EL2.<DC,VM> == '00' then
        AT_S1E1R(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        else
            AT_S12E1R(X[t]);
    elsif PSTATE.EL == EL3 then
        if !EL2Enabled() then
            AT_S1E1R(X[t]);
        elsif EL2Enabled() && (HCR_EL2.<E2H,TGE> == '11' || HCR_EL2.<DC,VM> == '00') then
            AT_S1E1R(X[t]);
        else
            AT_S12E1R(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.4.4 AT S12E1W, Address Translate Stages 1 and 2 EL1 Write

The AT S12E1W characteristics are:

Purpose

Performs stage 1 and 2 address translation, with permissions as if writing to the given virtual address from EL1, or from EL2 if the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - — If HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime, accessed from EL1.
  - — If HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime, accessed from EL2.
- Otherwise, the EL1&0 translation regime, accessed from EL1.

Configurations

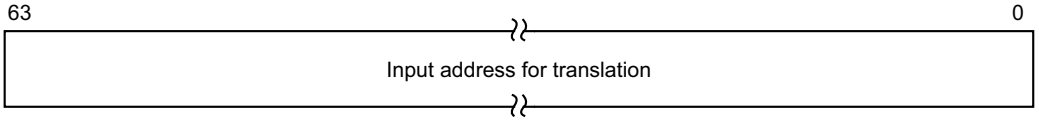
There are no configuration notes.

Attributes

AT S12E1W is a 64-bit System instruction.

Field descriptions

The AT S12E1W input value bit assignments are:



Bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing the AT S12E1W instruction

Accesses to this register use the following encodings in the System instruction encoding space:

AT S12E1W, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b100	0b0111	0b1000	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.<E2H,TGE> == '11' || HCR_EL2.<DC,VM> == '00' then
        AT_S1E1W(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        else
            AT_S12E1W(X[t]);
    elsif PSTATE.EL == EL3 then
        if !EL2Enabled() then
            AT_S1E1W(X[t]);
        elsif EL2Enabled() && (HCR_EL2.<E2H,TGE> == '11' || HCR_EL2.<DC,VM> == '00') then
            AT_S1E1W(X[t]);
        else
            AT_S12E1W(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.4.5 AT S1E0R, Address Translate Stage 1 EL0 Read

The AT S1E0R characteristics are:

Purpose

Performs stage 1 address translation from EL0, with permissions as if reading from the given virtual address from EL0, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - — If HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
  - — If HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime.
- Otherwise, the EL1&0 translation regime.

Configurations

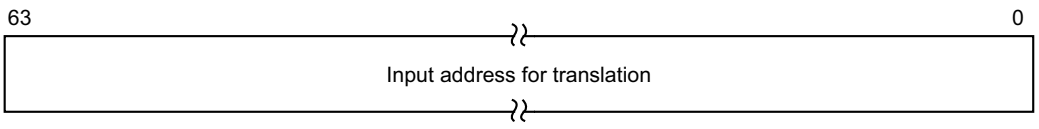
There are no configuration notes.

Attributes

AT S1E0R is a 64-bit System instruction.

Field descriptions

The AT S1E0R input value bit assignments are:



Bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing the AT S1E0R instruction

Accesses to this register use the following encodings in the System instruction encoding space:

AT S1E0R, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.AT == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.ATS1E0R == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AT_S1E0R(X[t]);
elsif PSTATE.EL == EL2 then
    AT_S1E0R(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    AT_S1E0R(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.4.6 AT S1E0W, Address Translate Stage 1 EL0 Write

The AT S1E0W characteristics are:

Purpose

Performs stage 1 address translation from EL0, with permissions as if writing to the given virtual address from EL0, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - — If HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
  - — If HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime.
- Otherwise, the EL1&0 translation regime.

Configurations

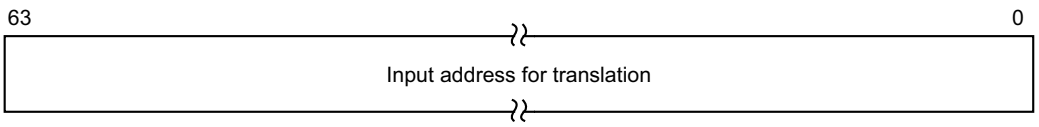
There are no configuration notes.

Attributes

AT S1E0W is a 64-bit System instruction.

Field descriptions

The AT S1E0W input value bit assignments are:



Bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing the AT S1E0W instruction

Accesses to this register use the following encodings in the System instruction encoding space:

AT S1E0W, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1000	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.AT == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.ATS1E0W == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AT_S1E0W(X[t]);
elsif PSTATE.EL == EL2 then
    AT_S1E0W(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    AT_S1E0W(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.4.7 AT S1E1R, Address Translate Stage 1 EL1 Read

The AT S1E1R characteristics are:

Purpose

Performs stage 1 address translation, with permissions as if reading from the given virtual address from EL1, or from EL2 if the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - — If HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime, accessed from EL1.
  - — If HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime, accessed from EL2.
- Otherwise, the EL1&0 translation regime, accessed from EL1.

Configurations

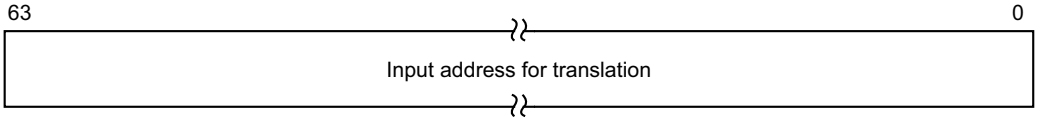
There are no configuration notes.

Attributes

AT S1E1R is a 64-bit System instruction.

Field descriptions

The AT S1E1R input value bit assignments are:



Bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing the AT S1E1R instruction

Accesses to this register use the following encodings in the System instruction encoding space:

AT S1E1R, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.AT == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.ATS1E1R == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AT_S1E1R(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL2 then
    AT_S1E1R(X[t]);
elsif PSTATE.EL == EL3 then
    AT_S1E1R(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.4.8 AT S1E1RP, Address Translate Stage 1 EL1 Read PAN

The AT S1E1RP characteristics are:

Purpose

Performs a stage 1 address translation, where the value of PSTATE.PAN determines if a read from a location will generate a permission fault for a privileged access, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - If HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime, accessed from EL1.
  - If HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime, accessed from EL2.
- Otherwise, the EL1&0 translation regime, accessed from EL1.

Configurations

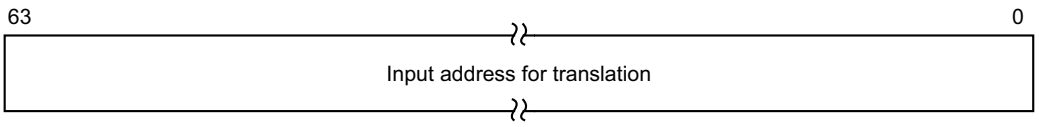
This instruction is present only when ARMv8.2-ATS1E1 is implemented. Otherwise, direct accesses to AT S1E1RP are UNDEFINED.

Attributes

AT S1E1RP is a 64-bit System instruction.

Field descriptions

The AT S1E1RP input value bit assignments are:



Bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing the AT S1E1RP instruction

Accesses to this register use the following encodings in the System instruction encoding space:

AT S1E1RP, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.AT == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.ATS1E1RP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AT_S1E1RP(X[t]);
elsif PSTATE.EL == EL2 then
    AT_S1E1RP(X[t]);
elsif PSTATE.EL == EL3 then
    AT_S1E1RP(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.4.9 AT S1E1W, Address Translate Stage 1 EL1 Write

The AT S1E1W characteristics are:

Purpose

Performs stage 1 address translation, with permissions as if writing to the given virtual address from EL1, or from EL2 if the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - — If HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime, accessed from EL1.
  - — If HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime, accessed from EL2.
- Otherwise, the EL1&0 translation regime, accessed from EL1.

Configurations

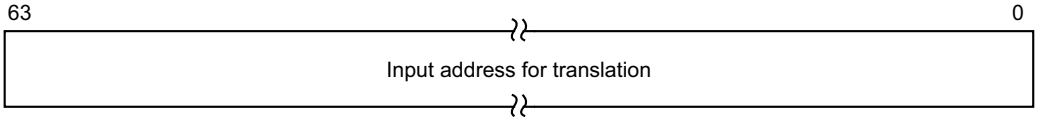
There are no configuration notes.

Attributes

AT S1E1W is a 64-bit System instruction.

Field descriptions

The AT S1E1W input value bit assignments are:



Bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing the AT S1E1W instruction

Accesses to this register use the following encodings in the System instruction encoding space:

AT S1E1W, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.AT == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.ATS1E1W == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AT_S1E1W(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL2 then
    AT_S1E1W(X[t]);
elsif PSTATE.EL == EL3 then
    AT_S1E1W(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.4.10 AT S1E1WP, Address Translate Stage 1 EL1 Write PAN

The AT S1E1WP characteristics are:

Purpose

Performs a stage 1 address translation, where the value of PSTATE.PAN determines if a write to a location will generate a permission fault for a privileged access, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - If HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime, accessed from EL1.
  - If HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime, accessed from EL2.
- Otherwise, the EL1&0 translation regime, accessed from EL1.

Configurations

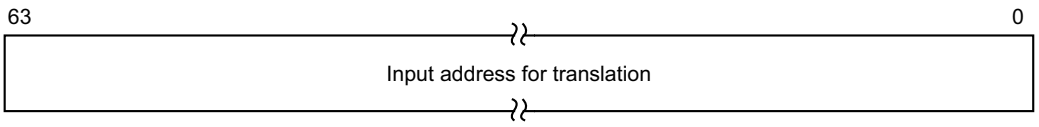
This instruction is present only when ARMv8.2-ATS1E1 is implemented. Otherwise, direct accesses to AT S1E1WP are UNDEFINED.

Attributes

AT S1E1WP is a 64-bit System instruction.

Field descriptions

The AT S1E1WP input value bit assignments are:



Bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing the AT S1E1WP instruction

Accesses to this register use the following encodings in the System instruction encoding space:

AT S1E1WP, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.AT == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.ATS1E1WP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AT_S1E1WP(X[t]);
elsif PSTATE.EL == EL2 then
    AT_S1E1WP(X[t]);
elsif PSTATE.EL == EL3 then
    AT_S1E1WP(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.4.11 AT S1E2R, Address Translate Stage 1 EL2 Read

The AT S1E2R characteristics are:

Purpose

Performs stage 1 address translation as defined for EL2, with permissions as if reading from the given virtual address.

Configurations

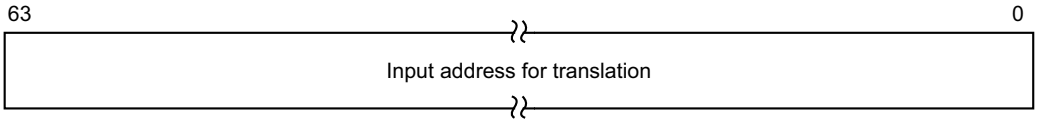
There are no configuration notes.

Attributes

AT S1E2R is a 64-bit System instruction.

Field descriptions

The AT S1E2R input value bit assignments are:



Bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing the AT S1E2R instruction

Accesses to this register use the following encodings in the System instruction encoding space:

AT S1E2R, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b100	0b0111	0b1000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AT_S1E2R(X[t]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    else
        AT_S1E2R(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.4.12 AT S1E2W, Address Translate Stage 1 EL2 Write

The AT S1E2W characteristics are:

Purpose

Performs stage 1 address translation as defined for EL2, with permissions as if writing to the given virtual address.

Configurations

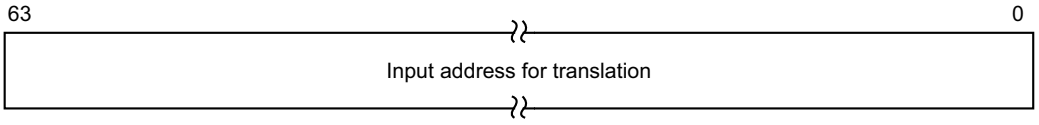
There are no configuration notes.

Attributes

AT S1E2W is a 64-bit System instruction.

Field descriptions

The AT S1E2W input value bit assignments are:



Bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing the AT S1E2W instruction

Accesses to this register use the following encodings in the System instruction encoding space:

AT S1E2W, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b100	0b0111	0b1000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AT_S1E2W(X[t]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    else
        AT_S1E2W(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.4.13 AT S1E3R, Address Translate Stage 1 EL3 Read

The AT S1E3R characteristics are:

Purpose

Performs stage 1 address translation as defined for EL3, with permissions as if reading from the given virtual address.

Configurations

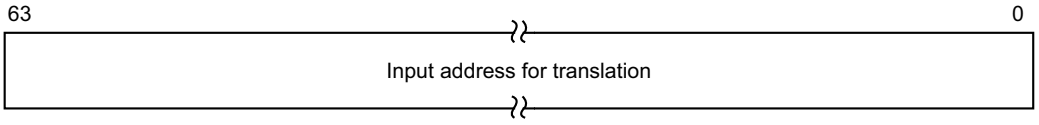
There are no configuration notes.

Attributes

AT S1E3R is a 64-bit System instruction.

Field descriptions

The AT S1E3R input value bit assignments are:



Bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing the AT S1E3R instruction

Accesses to this register use the following encodings in the System instruction encoding space:

AT S1E3R, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b110	0b0111	0b1000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    AT_S1E3R(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.4.14 AT S1E3W, Address Translate Stage 1 EL3 Write

The AT S1E3W characteristics are:

Purpose

Performs stage 1 address translation as defined for EL3, with permissions as if writing to the given virtual address.

Configurations

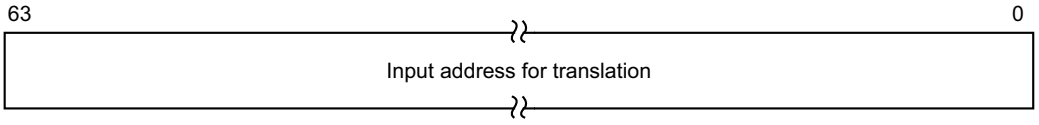
There are no configuration notes.

Attributes

AT S1E3W is a 64-bit System instruction.

Field descriptions

The AT S1E3W input value bit assignments are:



Bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing the AT S1E3W instruction

Accesses to this register use the following encodings in the System instruction encoding space:

AT S1E3W, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b110	0b0111	0b1000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    AT_S1E3W(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5 A64 System instructions for TLB maintenance

This section lists the A64 System instructions for TLB maintenance.

For more information about these instructions see *TLB maintenance instructions* on page D5-2671. In particular, for the full description of the scope of each instruction see *Scope of the A64 TLB maintenance instructions* on page D5-2676.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.1 TLBI ALLE1, TLB Invalidate All, EL1

The TLBI ALLE1 characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 or stage 2 translation table entry, from any level of the translation table walk.
- If SCR_EL3.NS is 0 and the entry would be required to translate an address using the Secure EL1&0 translation regime.
- If SCR_EL3.NS is 1 and the entry would be required to translate an address using the Non-secure EL1&0 translation regime.

The invalidation applies to entries with any VMID.

The invalidation only applies to the PE that executes this System instruction.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries, and non-global entries with any ASID.

Configurations

There are no configuration notes.

Attributes

TLBI ALLE1 is a 64-bit System instruction.

Field descriptions

TLBI ALLE1 ignores the value in the register specified by the instruction. Software does not have to write a value to the register before issuing this instruction.

Executing the TLBI ALLE1 instruction

When executing this instruction Xt should be encoded as `0b11111`. If the Xt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Xt field is set to `0b11111`.

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI ALLE1{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0111	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBI_ALLE1();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    TLBI_ALLE1();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.2 TLBI ALLE1IS, TLB Invalidate All, EL1, Inner Shareable

The TLBI ALLE1IS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 or stage 2 translation table entry, from any level of the translation table walk.
- If SCR_EL3.NS is 0 and the entry would be required to translate an address using the Secure EL1&0 translation regime.
- If SCR_EL3.NS is 1 and the entry would be required to translate an address using the Non-secure EL1&0 translation regime.

The invalidation applies to entries with any VMID.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries, and non-global entries with any ASID.

Configurations

There are no configuration notes.

Attributes

TLBI ALLE1IS is a 64-bit System instruction.

Field descriptions

TLBI ALLE1IS ignores the value in the register specified by the instruction. Software does not have to write a value to the register before issuing this instruction.

Executing the TLBI ALLE1IS instruction

When executing this instruction Xt should be encoded as `0b11111`. If the Xt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Xt field is set to `0b11111`.

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI ALLE1IS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0011	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    TLBI_ALLE1IS();
elsif PSTATE.EL == EL3 then
    TLBI_ALLE1IS();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.3 TLBI ALLE1OS, TLB Invalidate All, EL1, Outer Shareable

The TLBI ALLE1OS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 or stage 2 translation table entry, from any level of the translation table walk.
- If SCR_EL3.NS is 0 and the entry would be required to translate an address using the Secure EL1&0 translation regime.
- If SCR_EL3.NS is 1 and the entry would be required to translate an address using the Non-secure EL1&0 translation regime.

The invalidation applies to entries with any VMID.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries, and non-global entries with any ASID.

Configurations

This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI ALLE1OS are UNDEFINED.

Attributes

TLBI ALLE1OS is a 64-bit System instruction.

Field descriptions

TLBI ALLE1OS ignores the value in the register specified by the instruction. Software does not have to write a value to the register before issuing this instruction.

Executing the TLBI ALLE1OS instruction

When executing this instruction Xt should be encoded as `0b11111`. If the Xt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Xt field is set to `0b11111`.

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI ALLE1OS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0001	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL2 then
    TLBI_ALLE1OS();
elsif PSTATE.EL == EL3 then
    TLBI_ALLE1OS();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.4 TLBI ALLE2, TLB Invalidate All, EL2

The TLBI ALLE2 characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- If SCR_EL3.NS is 1 and the entry would be required to translate an address using the Non-secure EL2 or Non-secure EL2&0 translation regime.
- If SCR_EL3.NS is 0 and the entry would be required to translate an address using the Secure EL2 or Secure EL2&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Configurations

There are no configuration notes.

Attributes

TLBI ALLE2 is a 64-bit System instruction.

Field descriptions

TLBI ALLE2 ignores the value in the register specified by the instruction. Software does not have to write a value to the register before issuing this instruction.

Executing the TLBI ALLE2 instruction

When executing this instruction Xt should be encoded as `0b11111`. If the Xt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Xt field is set to `0b11111`.

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI ALLE2{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0111	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBI_ALLE2();
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    else
        TLBI_ALLE2();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.5 TLBI ALLE2IS, TLB Invalidate All, EL2, Inner Shareable

The TLBI ALLE2IS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- If SCR_EL3.NS is 1 and the entry would be required to translate an address using the Non-secure EL2 or Non-secure EL2&0 translation regime.
- If SCR_EL3.NS is 0 and the entry would be required to translate an address using the Secure EL2 or Secure EL2&0 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configurations

There are no configuration notes.

Attributes

TLBI ALLE2IS is a 64-bit System instruction.

Field descriptions

TLBI ALLE2IS ignores the value in the register specified by the instruction. Software does not have to write a value to the register before issuing this instruction.

Executing the TLBI ALLE2IS instruction

When executing this instruction Xt should be encoded as `0b11111`. If the Xt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Xt field is set to `0b11111`.

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI ALLE2IS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBI_ALLE2IS();
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    else
        TLBI_ALLE2IS();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.6 TLBI ALLE2OS, TLB Invalidate All, EL2, Outer Shareable

The TLBI ALLE2OS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- If SCR_EL3.NS is 1 and the entry would be required to translate an address using the Non-secure EL2 or Non-secure EL2&0 translation regime.
- If SCR_EL3.NS is 0 and the entry would be required to translate an address using the Secure EL2 or Secure EL2&0 translation regime.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Configurations

This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI ALLE2OS are UNDEFINED.

Attributes

TLBI ALLE2OS is a 64-bit System instruction.

Field descriptions

TLBI ALLE2OS ignores the value in the register specified by the instruction. Software does not have to write a value to the register before issuing this instruction.

Executing the TLBI ALLE2OS instruction

When executing this instruction Xt should be encoded as `0b11111`. If the Xt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Xt field is set to `0b11111`.

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI ALLE2OS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBI_ALLE2OS();
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    else
        TLBI_ALLE2OS();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.7 TLBI ALLE3, TLB Invalidate All, EL3

The TLBI ALLE3 characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be required to translate an address using the EL3 translation regime.

The invalidation applies to the PE that executes this System instruction.

Configurations

There are no configuration notes.

Attributes

TLBI ALLE3 is a 64-bit System instruction.

Field descriptions

TLBI ALLE3 ignores the value in the register specified by the instruction. Software does not have to write a value to the register before issuing this instruction.

Executing the TLBI ALLE3 instruction

When executing this instruction Xt should be encoded as `0b11111`. If the Xt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Xt field is set to `0b11111`.

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI ALLE3{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0111	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    TLBI_ALLE3();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.8 TLBI ALLE3IS, TLB Invalidate All, EL3, Inner Shareable

The TLBI ALLE3IS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be required to translate an address using the EL3 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configurations

There are no configuration notes.

Attributes

TLBI ALLE3IS is a 64-bit System instruction.

Field descriptions

TLBI ALLE3IS ignores the value in the register specified by the instruction. Software does not have to write a value to the register before issuing this instruction.

Executing the TLBI ALLE3IS instruction

When executing this instruction Xt should be encoded as `0b11111`. If the Xt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Xt field is set to `0b11111`.

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI ALLE3IS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    TLBI_ALLE3IS();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.9 TLBI ALLE3OS, TLB Invalidate All, EL3, Outer Shareable

The TLBI ALLE3OS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be required to translate an address using the EL3 translation regime.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Configurations

This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI ALLE3OS are UNDEFINED.

Attributes

TLBI ALLE3OS is a 64-bit System instruction.

Field descriptions

TLBI ALLE3OS ignores the value in the register specified by the instruction. Software does not have to write a value to the register before issuing this instruction.

Executing the TLBI ALLE3OS instruction

When executing this instruction Xt should be encoded as `0b11111`. If the Xt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Xt field is set to `0b11111`.

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI ALLE3OS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    TLBI_ALLE3OS();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.10 TLBI ASIDE1, TLB Invalidate by ASID, EL1

The TLBI ASIDE1 characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used for the specified ASID, and either:
  - — Is from a level of lookup above the final level.
  - — Is a non-global entry from the final level of lookup.
- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - — If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID, and would be required to translate an address using the EL1&0 translation regime.
  - — If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate an address using the EL2&0 translation regime.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate an address using the EL1&0 translation regime.

The invalidation applies to the PE that executes this System instruction.

Configurations

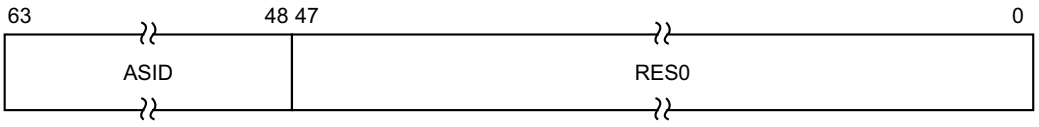
There are no configuration notes.

Attributes

TLBI ASIDE1 is a 64-bit System instruction.

Field descriptions

The TLBI ASIDE1 input value bit assignments are:



ASID, bits [63:48]

ASID value to match. Any appropriate TLB entries that match the ASID values will be affected by this System instruction.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are RES0.

Bits [47:0]

Reserved, RES0.

Executing the TLBI ASIDE1 instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TLBI ASIDE1{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0111	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.TLBIASIDE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.FB == '1' then
        TLBI_ASIDE1IS(X[t]);
    else
        TLBI_ASIDE1(X[t]);
elsif PSTATE.EL == EL2 then
    TLBI_ASIDE1(X[t]);
elsif PSTATE.EL == EL3 then
    TLBI_ASIDE1(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.11 TLBI ASIDE1IS, TLB Invalidate by ASID, EL1, Inner Shareable

The TLBI ASIDE1IS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used for the specified ASID, and either:
  - — Is from a level of lookup above the final level.
  - — Is a non-global entry from the final level of lookup.
- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - — If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID, and would be required to translate an address using the EL1&0 translation regime.
  - — If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate an address using the EL2&0 translation regime.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate an address using the EL1&0 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configurations

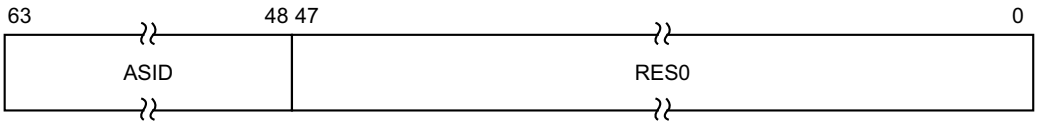
There are no configuration notes.

Attributes

TLBI ASIDE1IS is a 64-bit System instruction.

Field descriptions

The TLBI ASIDE1IS input value bit assignments are:



ASID, bits [63:48]

ASID value to match. Any appropriate TLB entries that match the ASID values will be affected by this System instruction.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are RES0.

Bits [47:0]

Reserved, RES0.

Executing the TLBI ASIDE1IS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TLBI ASIDE1IS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0011	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.TLBIASIDE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        TLBI_ASIDE1IS(X[t]);
elsif PSTATE.EL == EL2 then
    TLBI_ASIDE1IS(X[t]);
elsif PSTATE.EL == EL3 then
    TLBI_ASIDE1IS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.12 TLBI ASIDE1OS, TLB Invalidate by ASID, EL1, Outer Shareable

The TLBI ASIDE1OS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used for the specified ASID, and either:
  - — Is from a level of lookup above the final level.
  - — Is a non-global entry from the final level of lookup.
- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - — If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID, and would be required to translate an address using the EL1&0 translation regime.
  - — If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate an address using the EL2&0 translation regime.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate an address using the EL1&0 translation regime.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Configurations

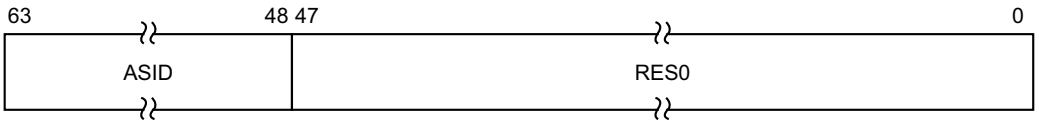
This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI ASIDE1OS are UNDEFINED.

Attributes

TLBI ASIDE1OS is a 64-bit System instruction.

Field descriptions

The TLBI ASIDE1OS input value bit assignments are:



ASID, bits [63:48]

ASID value to match. Any appropriate TLB entries that match the ASID values will be affected by this System instruction.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are RES0.

Bits [47:0]

Reserved, RES0.

Executing the TLBI ASIDE1OS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TLBI ASIDE1OS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0001	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.TLBIASIDE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        TLBI_ASIDE1OS(X[t]);
elsif PSTATE.EL == EL2 then
    TLBI_ASIDE1OS(X[t]);
elsif PSTATE.EL == EL3 then
    TLBI_ASIDE1OS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.13 TLBI IPAS2E1, TLB Invalidate by Intermediate Physical Address, Stage 2, EL1

The TLBI IPAS2E1 characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from any level of the translation table walk.
- One of the following applies:
  - SCR_EL3.NS==1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.NS==0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction see *Invalidation of TLB entries from stage 2 translations* on page D5-2680.

Configurations

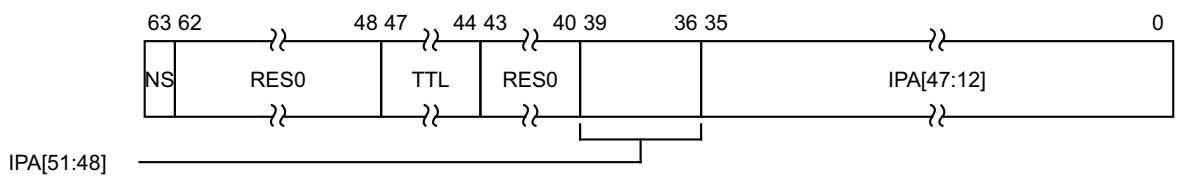
There are no configuration notes.

Attributes

TLBI IPAS2E1 is a 64-bit System instruction.

Field descriptions

The TLBI IPAS2E1 input value bit assignments are:



NS, bit [63]

When ARMv8.4-SecEL2 is implemented:

Not Secure. Specifies the IPA space.

`0b0` IPA is in the Secure IPA space.

`0b1` IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When ARMv8.4-SecEL2 is not implemented or is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TTL, bits [47:44]

When ARMv8.4-TTL is implemented:

Translation Table Level. Indicates the level of the page table walk that holds the leaf entry for the address being invalidated.

`0b00xx` No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

`0b01xx` The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

`0b10xx` The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b10` : Level 2.

`0b11` : Level 3.

`0b11xx` The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:40]

Reserved, RES0.

IPA[51:48], bits [39:36]

When ARMv8.2-LPA is implemented:

Extension to IPA[47:12]. See IPA[47:12] for more details.

Otherwise:

Reserved, RES0.

IPA[47:12], bits [35:0]

Bits[47:12] of the intermediate physical address to match. For implementations with fewer than 48 bits, the upper bits of this field are RES0.

When ARMv8.2-LPA is implemented, and 52-bit addresses and a 64KB translation granule are in use, IPA[51:48] form the upper part of the address value. Otherwise, IPA[51:48] are RES0.

Executing the TLBI IPAS2E1 instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TLBI IPAS2E1{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBI_IPAS2E1(X[t]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        //no operation
    else
        TLBI_IPAS2E1(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.14 TLBI IPAS2E1IS, TLB Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable

The TLBI IPAS2E1IS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from any level of the translation table walk.
- One of the following applies:
  - SCR_EL3.NS==1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.NS==0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction see *Invalidation of TLB entries from stage 2 translations* on page D5-2680.

Configurations

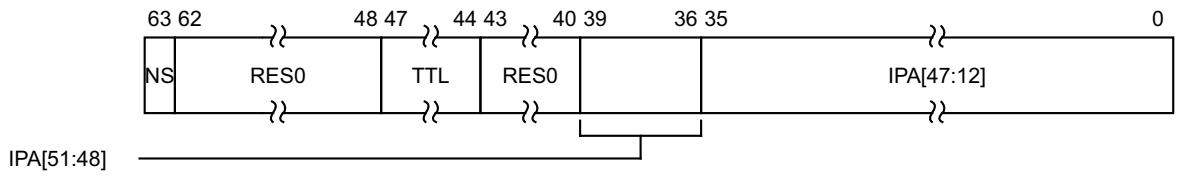
There are no configuration notes.

Attributes

TLBI IPAS2E1IS is a 64-bit System instruction.

Field descriptions

The TLBI IPAS2E1IS input value bit assignments are:



NS, bit [63]

When ARMv8.4-SecEL2 is implemented:

Not Secure. Specifies the IPA space.

`0b0` IPA is in the Secure IPA space.

`0b1` IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When ARMv8.4-SecEL2 is not implemented or is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When ARMv8.4-TTL is implemented:

Translation Table Level. Indicates the level of the page table walk that holds the leaf entry for the address being invalidated.

`0b00xx` No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

`0b01xx` The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

`0b10xx` The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b10` : Level 2.

`0b11` : Level 3.

`0b11xx` The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:40]

Reserved, RES0.

IPA[51:48], bits [39:36]

When ARMv8.2-LPA is implemented:

Extension to IPA[47:12]. See IPA[47:12] for more details.

Otherwise:

Reserved, RES0.

IPA[47:12], bits [35:0]

Bits[47:12] of the intermediate physical address to match. For implementations with fewer than 48 bits, the upper bits of this field are RES0.

When ARMv8.2-LPA is implemented, and 52-bit addresses and a 64KB translation granule are in use, IPA[51:48] form the upper part of the address value. Otherwise, IPA[51:48] are RES0.

Executing the TLBI IPAS2E1IS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TLBI IPAS2E1IS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBI_IPAS2E1IS(X[t]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        //no operation
    else
        TLBI_IPAS2E1IS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.15 TLBI IPAS2E1OS, TLB Invalidate by Intermediate Physical Address, Stage 2, EL1, Outer Shareable

The TLBI IPAS2E1OS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from any level of the translation table walk.
- One of the following applies:
  - SCR_EL3.NS==1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.NS==0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction see *Invalidation of TLB entries from stage 2 translations* on page D5-2680.

Configurations

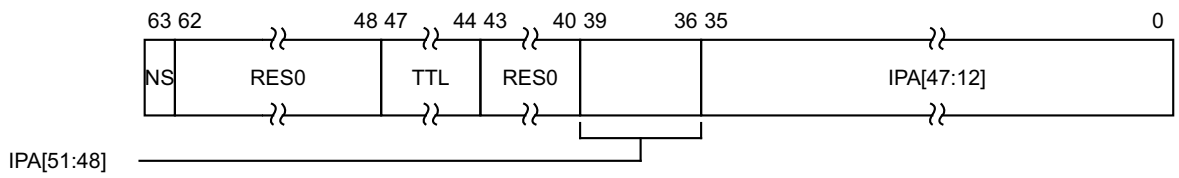
This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI IPAS2E1OS are UNDEFINED.

Attributes

TLBI IPAS2E1OS is a 64-bit System instruction.

Field descriptions

The TLBI IPAS2E1OS input value bit assignments are:



NS, bit [63]

Not Secure. Specifies the IPA space.

`0b0` IPA is in the Secure IPA space.

`0b1` IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When ARMv8.4-SecEL2 is not implemented or is disabled in the current Security state, this field is RES0.

Bits [62:48]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TTL, bits [47:44]

When ARMv8.4-TTL is implemented:

Translation Table Level. Indicates the level of the page table walk that holds the leaf entry for the address being invalidated.

`0b00xx` No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

`0b01xx` The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

`0b10xx` The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b10` : Level 2.

`0b11` : Level 3.

`0b11xx` The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:40]

Reserved, RES0.

IPA[51:48], bits [39:36]

Extension to IPA[47:12]. See IPA[47:12] for more details.

IPA[47:12], bits [35:0]

Bits[47:12] of the intermediate physical address to match. For implementations with fewer than 48 bits, the upper bits of this field are RES0.

When ARMv8.2-LPA is implemented, and 52-bit addresses and a 64KB translation granule are in use, IPA[51:48] form the upper part of the address value. Otherwise, IPA[51:48] are RES0.

Executing the TLBI IPAS2E1OS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TLBI IPAS2E1OS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBI_IPAS2E1OS(X[t]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        //no operation
    else
        TLBI_IPAS2E1OS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.16 TLBI IPAS2LE1, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1

The TLBI IPAS2LE1 characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from the final level of the translation table walk.
- One of the following applies:
  - SCR_EL3.NS==1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.NS==0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction see *Invalidation of TLB entries from stage 2 translations* on page D5-2680.

Configurations

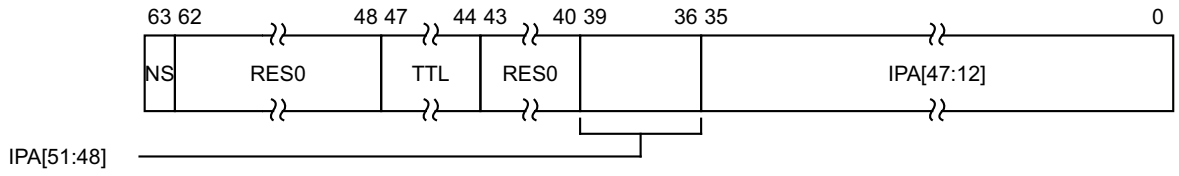
There are no configuration notes.

Attributes

TLBI IPAS2LE1 is a 64-bit System instruction.

Field descriptions

The TLBI IPAS2LE1 input value bit assignments are:



NS, bit [63]

When ARMv8.4-SecEL2 is implemented:

Not Secure. Specifies the IPA space.

`0b0` IPA is in the Secure IPA space.

`0b1` IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When ARMv8.4-SecEL2 is not implemented or is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TTL, bits [47:44]

When ARMv8.4-TTL is implemented:

Translation Table Level. Indicates the level of the page table walk that holds the leaf entry for the address being invalidated.

`0b00xx` No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

`0b01xx` The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

`0b10xx` The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b10` : Level 2.

`0b11` : Level 3.

`0b11xx` The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:40]

Reserved, RES0.

IPA[51:48], bits [39:36]

When ARMv8.2-LPA is implemented:

Extension to IPA[47:12]. See IPA[47:12] for more details.

Otherwise:

Reserved, RES0.

IPA[47:12], bits [35:0]

Bits[47:12] of the intermediate physical address to match. For implementations with fewer than 48 bits, the upper bits of this field are RES0.

When ARMv8.2-LPA is implemented, and 52-bit addresses and a 64KB translation granule are in use, IPA[51:48] form the upper part of the address value. Otherwise, IPA[51:48] are RES0.

Executing the TLBI IPAS2LE1 instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TLBI IPAS2LE1{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBI_IPAS2LE1(X[t]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        //no operation
    else
        TLBI_IPAS2LE1(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.17 TLBI IPAS2LE1IS, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable

The TLBI IPAS2LE1IS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from the final level of the translation table walk.
- One of the following applies:
  - SCR_EL3.NS==1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.NS==0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction see *Invalidation of TLB entries from stage 2 translations* on page D5-2680.

Configurations

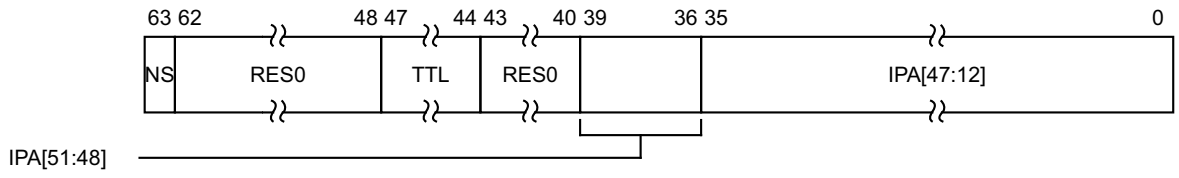
There are no configuration notes.

Attributes

TLBI IPAS2LE1IS is a 64-bit System instruction.

Field descriptions

The TLBI IPAS2LE1IS input value bit assignments are:



NS, bit [63]

When ARMv8.4-SecEL2 is implemented:

Not Secure. Specifies the IPA space.

`0b0` IPA is in the Secure IPA space.

`0b1` IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When ARMv8.4-SecEL2 is not implemented or is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When ARMv8.4-TTL is implemented:

Translation Table Level. Indicates the level of the page table walk that holds the leaf entry for the address being invalidated.

`0b00xx` No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

`0b01xx` The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

`0b10xx` The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b10` : Level 2.

`0b11` : Level 3.

`0b11xx` The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:40]

Reserved, RES0.

IPA[51:48], bits [39:36]

When ARMv8.2-LPA is implemented:

Extension to IPA[47:12]. See IPA[47:12] for more details.

Otherwise:

Reserved, RES0.

IPA[47:12], bits [35:0]

Bits[47:12] of the intermediate physical address to match. For implementations with fewer than 48 bits, the upper bits of this field are RES0.

When ARMv8.2-LPA is implemented, and 52-bit addresses and a 64KB translation granule are in use, IPA[51:48] form the upper part of the address value. Otherwise, IPA[51:48] are RES0.

Executing the TLBI IPAS2LE1IS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TLBI IPAS2LE1IS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0000	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBI_IPAS2LE1IS(X[t]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        //no operation
    else
        TLBI_IPAS2LE1IS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.18 TLBI IPAS2LE1OS, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable

The TLBI IPAS2LE1OS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from the final level of the translation table walk.
- One of the following applies:
  - SCR_EL3.NS==1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.NS==0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction see *Invalidation of TLB entries from stage 2 translations* on page D5-2680.

Configurations

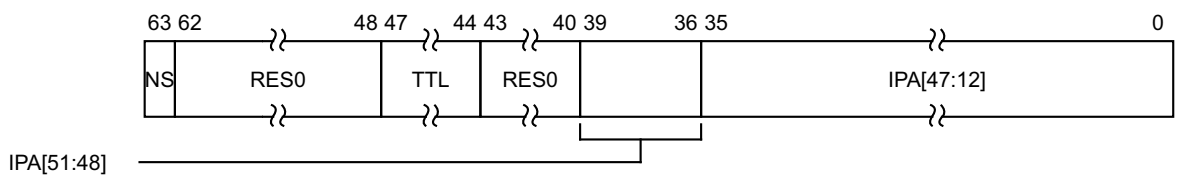
This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI IPAS2LE1OS are UNDEFINED.

Attributes

TLBI IPAS2LE1OS is a 64-bit System instruction.

Field descriptions

The TLBI IPAS2LE1OS input value bit assignments are:



NS, bit [63]

When ARMv8.4-SecEL2 is implemented:

Not Secure. Specifies the IPA space.

`0b0` IPA is in the Secure IPA space.

`0b1` IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When ARMv8.4-SecEL2 is not implemented or is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When ARMv8.4-TTL is implemented:

Translation Table Level. Indicates the level of the page table walk that holds the leaf entry for the address being invalidated.

`0b00xx` No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

`0b01xx` The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

`0b10xx` The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b10` : Level 2.

`0b11` : Level 3.

`0b11xx` The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:40]

Reserved, RES0.

IPA[51:48], bits [39:36]

Extension to IPA[47:12]. See IPA[47:12] for more details.

IPA[47:12], bits [35:0]

Bits[47:12] of the intermediate physical address to match. For implementations with fewer than 48 bits, the upper bits of this field are RES0.

When ARMv8.2-LPA is implemented, and 52-bit addresses and a 64KB translation granule are in use, IPA[51:48] form the upper part of the address value. Otherwise, IPA[51:48] are RES0.

Executing the TLBI IPAS2LE1OS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TLBI IPAS2LE1OS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBI_IPAS2LE1OS(X[t]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        //no operation
    else
        TLBI_IPAS2LE1OS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.19 TLBI RIPAS2E1, TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1

The TLBI RIPAS2E1 characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from any level of the translation table walk.
- One of the following applies:
  - SCR_EL3.NS==1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.NS==0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*2^(5*SCALE +1) * Translation_Granule_Size)].

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to the PE that executes this System instruction.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

For more information about the architectural requirements for this System instruction see *Invalidation of TLB entries from stage 2 translations* on page D5-2680.

Configurations

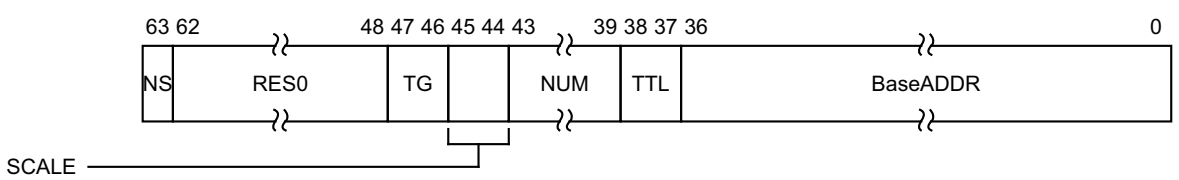
This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI RIPAS2E1 are UNDEFINED.

Attributes

TLBI RIPAS2E1 is a 64-bit System instruction.

Field descriptions

The TLBI RIPAS2E1 input value bit assignments are:



Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

NS, bit [63]

When ARMv8.4-SecEL2 is implemented:

Not Secure. Specifies the IPA space.

`0b0` IPA is in the Secure IPA space.

`0b1` IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When ARMv8.4-SecEL2 is not implemented or is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

`0b00` Reserved.

`0b01` 4K translation granule.

`0b10` 16K translation granule.

`0b11` 64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

`0b00` The entries in the range can be using any level for the translation table entries.

`0b01` When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries.

When using a 16KB translation granule, this value is reserved. Hardware should treat this field as `0b00`.

`0b10` All entries to invalidate are Level 2 translation table entries.

`0b11` All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing the TLBI RIPAS2E1 instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TLBI RIPAS2E1{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBI_RIPAS2E1(X[t]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        //no operation
    else
        TLBI_RIPAS2E1(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.20 TLBI RIPAS2E1IS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable

The TLBI RIPAS2E1IS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from any level of the translation table walk.
- One of the following applies:
  - SCR_EL3.NS==1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.NS==0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*2^(5*SCALE +1) * Translation_Granule_Size)].

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

For more information about the architectural requirements for this System instruction see *Invalidation of TLB entries from stage 2 translations* on page D5-2680.

Configurations

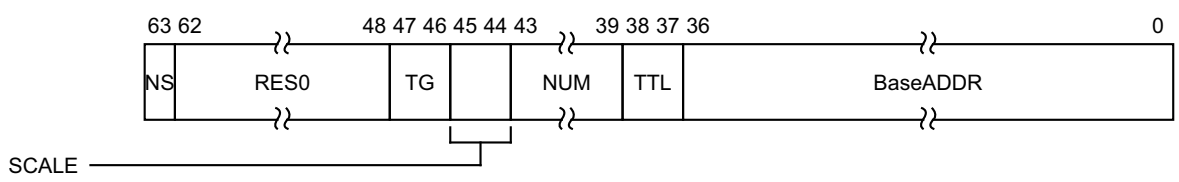
This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI RIPAS2E1IS are UNDEFINED.

Attributes

TLBI RIPAS2E1IS is a 64-bit System instruction.

Field descriptions

The TLBI RIPAS2E1IS input value bit assignments are:



Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

NS, bit [63]

When ARMv8.4-SecEL2 is implemented:

Not Secure. Specifies the IPA space.

`0b0` IPA is in the Secure IPA space.

`0b1` IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When ARMv8.4-SecEL2 is not implemented or is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

`0b00` Reserved.

`0b01` 4K translation granule.

`0b10` 16K translation granule.

`0b11` 64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

`0b00` The entries in the range can be using any level for the translation table entries.

`0b01` When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries.

When using a 16KB translation granule, this value is reserved. Hardware should treat this field as `0b00`.

`0b10` All entries to invalidate are Level 2 translation table entries.

`0b11` All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing the TLBI RIPAS2E1IS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TLBI RIPAS2E1IS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBI_RIPAS2E1IS(X[t]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        //no operation
    else
        TLBI_RIPAS2E1IS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.21 TLBI RIPAS2E1OS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Outer Shareable

The TLBI RIPAS2E1OS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from any level of the translation table walk.
- One of the following applies:
  - SCR_EL3.NS==1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.NS==0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*2^(5*SCALE +1) * Translation_Granule_Size)].

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

For more information about the architectural requirements for this System instruction see *Invalidation of TLB entries from stage 2 translations* on page D5-2680.

Configurations

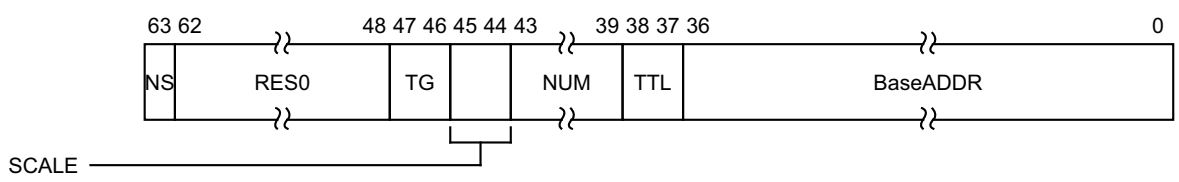
This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI RIPAS2E1OS are UNDEFINED.

Attributes

TLBI RIPAS2E1OS is a 64-bit System instruction.

Field descriptions

The TLBI RIPAS2E1OS input value bit assignments are:



Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

NS, bit [63]

Not Secure. Specifies the IPA space.

`0b0` IPA is in the Secure IPA space.

`0b1` IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When ARMv8.4-SecEL2 is not implemented or is disabled in the current Security state, this field is RES0.

Bits [62:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

`0b00` Reserved.

`0b01` 4K translation granule.

`0b10` 16K translation granule.

`0b11` 64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

`0b00` The entries in the range can be using any level for the translation table entries.

`0b01` When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries.

When using a 16KB translation granule, this value is reserved. Hardware should treat this field as `0b00`.

`0b10` All entries to invalidate are Level 2 translation table entries.

`0b11` All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing the TLBI RIPAS2E1OS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TLBI RIPAS2E1OS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBI_RIPAS2E1OS(X[t]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        //no operation
    else
        TLBI_RIPAS2E1OS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.22 TLBI RIPAS2LE1, TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1

The TLBI RIPAS2LE1 characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from the final level of the translation table walk.
- One of the following applies:
  - SCR_EL3.NS==1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.NS==0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*2^(5*SCALE +1) * Translation_Granule_Size)].

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation only applies to the PE that executes this System instruction.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

For more information about the architectural requirements for this System instruction see *Invalidation of TLB entries from stage 2 translations* on page D5-2680.

Configurations

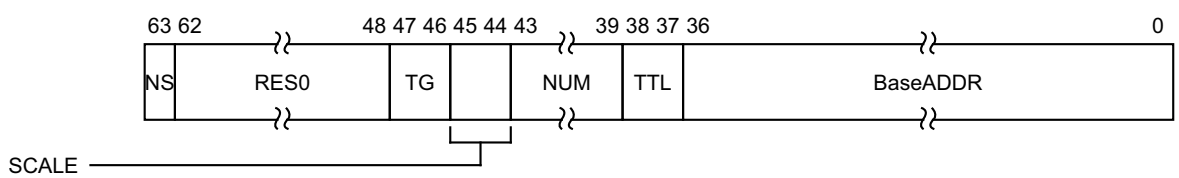
This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI RIPAS2LE1 are UNDEFINED.

Attributes

TLBI RIPAS2LE1 is a 64-bit System instruction.

Field descriptions

The TLBI RIPAS2LE1 input value bit assignments are:



Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

NS, bit [63]

When ARMv8.4-SecEL2 is implemented:

Not Secure. Specifies the IPA space.

`0b0` IPA is in the Secure IPA space.

`0b1` IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When ARMv8.4-SecEL2 is not implemented or is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

`0b00` Reserved.

`0b01` 4K translation granule.

`0b10` 16K translation granule.

`0b11` 64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

`0b00` The entries in the range can be using any level for the translation table entries.

`0b01` When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries.

When using a 16KB translation granule, this value is reserved. Hardware should treat this field as `0b00`.

`0b10` All entries to invalidate are Level 2 translation table entries.

`0b11` All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing the TLBI RIPAS2LE1 instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TLBI RIPAS2LE1{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBI_RIPAS2LE1(X[t]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        //no operation
    else
        TLBI_RIPAS2LE1(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.23 TLBI RIPAS2LE1IS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable

The TLBI RIPAS2LE1IS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from the final level of the translation table walk.
- One of the following applies:
  - SCR_EL3.NS==1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.NS==0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*2^(5*SCALE +1) * Translation_Granule_Size)].

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

For more information about the architectural requirements for this System instruction see *Invalidation of TLB entries from stage 2 translations* on page D5-2680.

Configurations

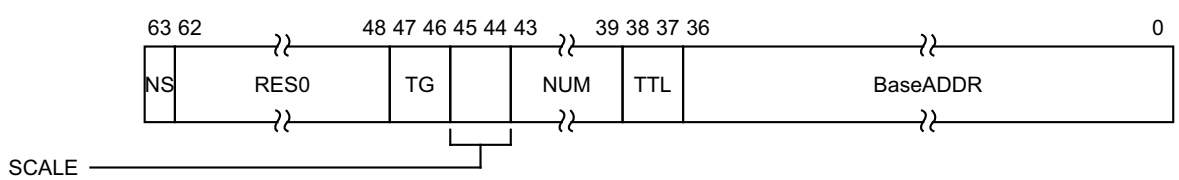
This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI RIPAS2LE1IS are UNDEFINED.

Attributes

TLBI RIPAS2LE1IS is a 64-bit System instruction.

Field descriptions

The TLBI RIPAS2LE1IS input value bit assignments are:



Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

NS, bit [63]

When ARMv8.4-SecEL2 is implemented:

Not Secure. Specifies the IPA space.

`0b0` IPA is in the Secure IPA space.

`0b1` IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When ARMv8.4-SecEL2 is not implemented or is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

`0b00` Reserved.

`0b01` 4K translation granule.

`0b10` 16K translation granule.

`0b11` 64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

`0b00` The entries in the range can be using any level for the translation table entries.

`0b01` When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries.

When using a 16KB translation granule, this value is reserved. Hardware should treat this field as `0b00`.

`0b10` All entries to invalidate are Level 2 translation table entries.

`0b11` All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing the TLBI RIPAS2LE1IS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TLBI RIPAS2LE1IS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0000	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBI_RIPAS2LE1IS(X[t]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        //no operation
    else
        TLBI_RIPAS2LE1IS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.24 TLBI RIPAS2LE1OS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable

The TLBI RIPAS2LE1OS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from any level of the translation table walk.
- One of the following applies:
  - SCR_EL3.NS==1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.NS==0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*2^(5*SCALE +1) * Translation_Granule_Size)].

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

For more information about the architectural requirements for this System instruction see *Invalidation of TLB entries from stage 2 translations* on page D5-2680.

Configurations

This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI RIPAS2LE1OS are UNDEFINED.

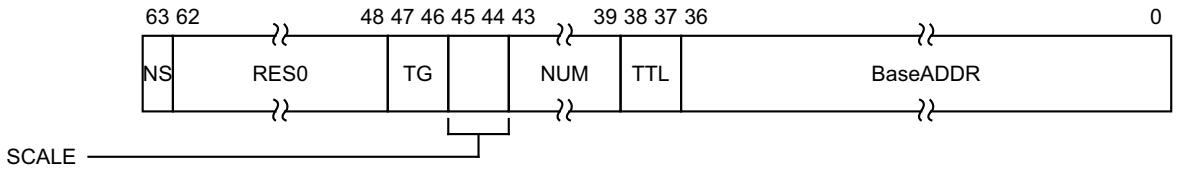
Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Attributes

TLBI RIPAS2LE1OS is a 64-bit System instruction.

Field descriptions

The TLBI RIPAS2LE1OS input value bit assignments are:



NS, bit [63]

When ARMv8.4-SecEL2 is implemented:

Not Secure. Specifies the IPA space.

`0b0` IPA is in the Secure IPA space.

`0b1` IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When ARMv8.4-SecEL2 is not implemented or is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

`0b00` Reserved.

`0b01` 4K translation granule.

`0b10` 16K translation granule.

`0b11` 64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

`0b00` The entries in the range can be using any level for the translation table entries.

`0b01` When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When using a 16KB translation granule, this value is reserved. Hardware should treat this field as `0b00`.

`0b10` All entries to invalidate are Level 2 translation table entries.

`0b11` All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing the TLBI RIPAS2LE1OS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI RIPAS2LE1OS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBI_RIPAS2LE1OS(X[t]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        //no operation
    else
        TLBI_RIPAS2LE1OS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.25 TLBI RVAAE1, TLB Range Invalidate by VA, All ASID, EL1

The TLBI RVAAE1 characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime.
  - If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*2^(5*SCALE +1) * Translation_Granule_Size)].

The invalidation applies to the PE that executes this System instruction.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both:

- Global entries.
- Non-global entries with any ASID.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 0000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

Configurations

This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI RVAAE1 are UNDEFINED.

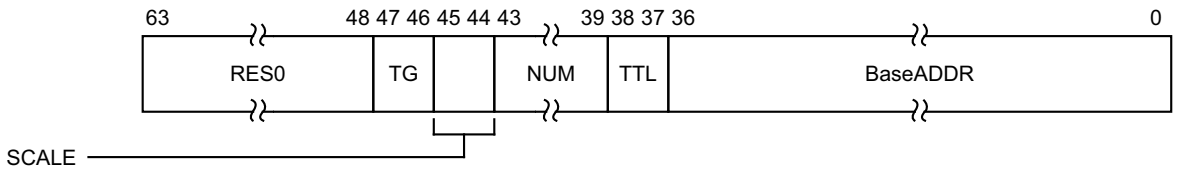
Attributes

TLBI RVAAE1 is a 64-bit System instruction.

Field descriptions

The TLBI RVAAE1 input value bit assignments are:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta



Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

`0b00` Reserved.

`0b01` 4K translation granule.

`0b10` 16K translation granule.

`0b11` 64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

`0b00` The entries in the range can be using any level for the translation table entries.

`0b01` When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries.

When using a 16KB translation granule, this value is reserved and hardware should treat this field as `0b00`.

`0b10` All entries to invalidate are Level 2 translation table entries.

`0b11` All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing the TLBI RVAAE1 instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TLBI RVAAE1{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0110	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.TLBIRVAAE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.FB == '1' then
        TLBI_RVAAE1IS(X[t]);
    else
        TLBI_RVAAE1(X[t]);
elsif PSTATE.EL == EL2 then
    TLBI_RVAAE1(X[t]);
elsif PSTATE.EL == EL3 then
    TLBI_RVAAE1(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.26 TLBI RVAAE1IS, TLB Range Invalidate by VA, All ASID, EL1, Inner Shareable

The TLBI RVAAE1IS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime.
  - If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*2^(5*SCALE +1) * Translation_Granule_Size)].

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both:

- Global entries.
- Non-global entries with any ASID.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

Configurations

This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI RVAAE1IS are UNDEFINED.

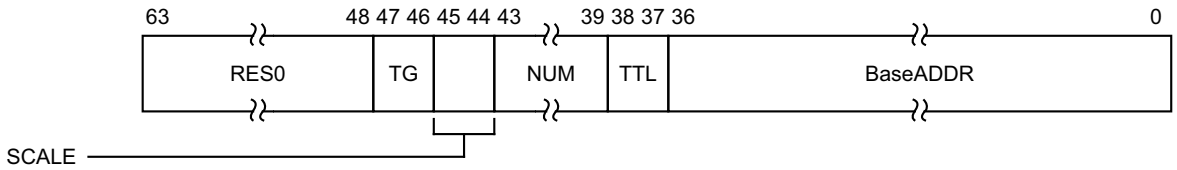
Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Attributes

TLBI RVAAE1IS is a 64-bit System instruction.

Field descriptions

The TLBI RVAAE1IS input value bit assignments are:



Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

`0b00` Reserved.

`0b01` 4K translation granule.

`0b10` 16K translation granule.

`0b11` 64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

`0b00` The entries in the range can be using any level for the translation table entries.

`0b01` When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries.

When using a 16KB translation granule, this value is reserved and hardware should treat this field as `0b00`.

`0b10` All entries to invalidate are Level 2 translation table entries.

`0b11` All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Executing the TLBI RVAAE1IS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI RVAAE1IS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0010	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.TLBIRVAAE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        TLBI_RVAAE1IS(X[t]);
elsif PSTATE.EL == EL2 then
    TLBI_RVAAE1IS(X[t]);
elsif PSTATE.EL == EL3 then
    TLBI_RVAAE1IS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.27 TLBI RVAAE1OS, TLB Range Invalidate by VA, All ASID, EL1, Outer Shareable

The TLBI RVAAE1OS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime.
  - If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*2^(5*SCALE +1) * Translation_Granule_Size)].

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both:

- Global entries.
- Non-global entries with any ASID.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

Configurations

This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI RVAAE1OS are UNDEFINED.

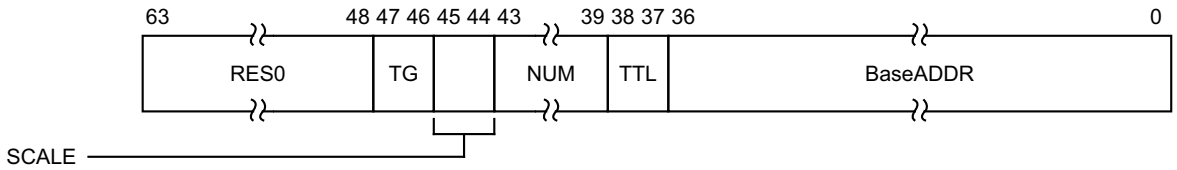
Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Attributes

TLBI RVAAE1OS is a 64-bit System instruction.

Field descriptions

The TLBI RVAAE1OS input value bit assignments are:



Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

`0b00` Reserved.

`0b01` 4K translation granule.

`0b10` 16K translation granule.

`0b11` 64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

`0b00` The entries in the range can be using any level for the translation table entries.

`0b01` When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries.

When using a 16KB translation granule, this value is reserved and hardware should treat this field as `0b00`.

`0b10` All entries to invalidate are Level 2 translation table entries.

`0b11` All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Executing the TLBI RVAAE1OS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI RVAAE1OS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0101	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.TLBIRVAAE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        TLBI_RVAAE1OS(X[t]);
elsif PSTATE.EL == EL2 then
    TLBI_RVAAE1OS(X[t]);
elsif PSTATE.EL == EL3 then
    TLBI_RVAAE1OS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.28 TLBI RVAALE1, TLB Range Invalidate by VA, All ASID, Last level, EL1

The TLBI RVAALE1 characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.
- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime.
  - If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*2^(5*SCALE +1) * Translation_Granule_Size)].

The invalidation applies to the PE that executes this System instruction.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both:

- Global entries.
- Non-global entries with any ASID.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 0000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

Configurations

This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI RVAALE1 are UNDEFINED.

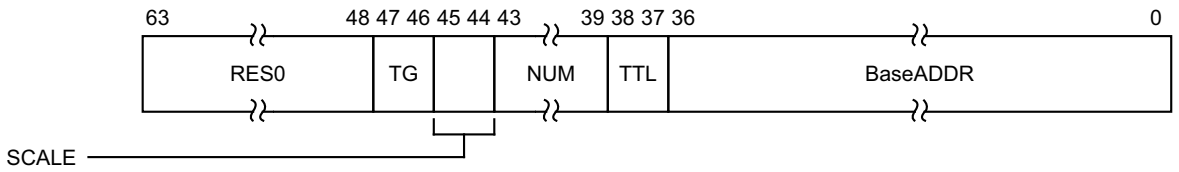
Attributes

TLBI RVAALE1 is a 64-bit System instruction.

Field descriptions

The TLBI RVAALE1 input value bit assignments are:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta



Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

`0b00` Reserved.

`0b01` 4K translation granule.

`0b10` 16K translation granule.

`0b11` 64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

`0b00` The entries in the range can be using any level for the translation table entries.

`0b01` When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries.

When using a 16KB translation granule, this value is reserved and hardware should treat this field as `0b00`.

`0b10` All entries to invalidate are Level 2 translation table entries.

`0b11` All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing the TLBI RVAALE1 instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TLBI RVAALE1{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0110	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.TLBIRVAALE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.FB == '1' then
        TLBI_RVAAE1IS(X[t]);
    else
        TLBI_RVAALE1(X[t]);
elsif PSTATE.EL == EL2 then
    TLBI_RVAALE1(X[t]);
elsif PSTATE.EL == EL3 then
    TLBI_RVAALE1(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.29 TLBI RVAALE1IS, TLB Range Invalidate by VA, All ASID, Last Level, EL1, Inner Shareable

The TLBI RVAALE1IS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.
- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime.
  - If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*2^(5*SCALE +1) * Translation_Granule_Size)].

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both:

- Global entries.
- Non-global entries with any ASID.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

Configurations

This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI RVAALE1IS are UNDEFINED.

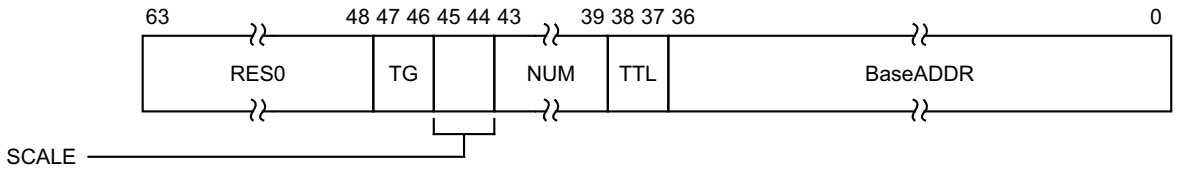
Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Attributes

TLBI RVAALE1IS is a 64-bit System instruction.

Field descriptions

The TLBI RVAALE1IS input value bit assignments are:



Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

`0b00` Reserved.

`0b01` 4K translation granule.

`0b10` 16K translation granule.

`0b11` 64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

`0b00` The entries in the range can be using any level for the translation table entries.

`0b01` When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries.

When using a 16KB translation granule, this value is reserved and hardware should treat this field as `0b00`.

`0b10` All entries to invalidate are Level 2 translation table entries.

`0b11` All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Executing the TLBI RVAALE1IS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI RVAALE1IS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0010	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.TLBIRVAALE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        TLBI_RVAALE1IS(X[t]);
elsif PSTATE.EL == EL2 then
    TLBI_RVAALE1IS(X[t]);
elsif PSTATE.EL == EL3 then
    TLBI_RVAALE1IS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.30 TLBI RVAALE1OS, TLB Range Invalidate by VA, All ASID, Last Level, EL1, Outer Shareable

The TLBI RVAALE1OS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.
- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime.
  - If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*2^(5*SCALE +1) * Translation_Granule_Size)].

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both:

- Global entries.
- Non-global entries with any ASID.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

Configurations

This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI RVAALE1OS are UNDEFINED.

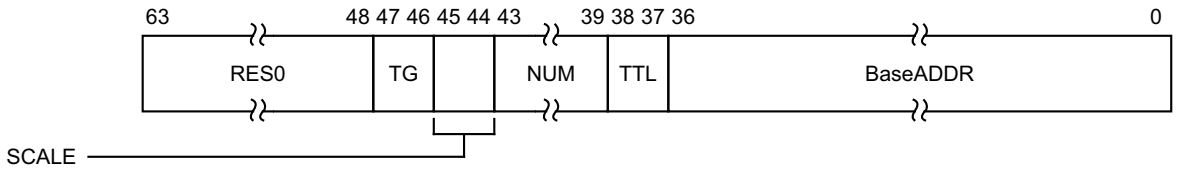
Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Attributes

TLBI RVAALE1OS is a 64-bit System instruction.

Field descriptions

The TLBI RVAALE1OS input value bit assignments are:



Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

`0b00` Reserved.

`0b01` 4K translation granule.

`0b10` 16K translation granule.

`0b11` 64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

`0b00` The entries in the range can be using any level for the translation table entries.

`0b01` When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries.

When using a 16KB translation granule, this value is reserved and hardware should treat this field as `0b00`.

`0b10` All entries to invalidate are Level 2 translation table entries.

`0b11` All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Executing the TLBI RVAALE1OS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI RVAALE1OS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0101	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.TLBIRVAALE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        TLBI_RVAALE1OS(X[t]);
elsif PSTATE.EL == EL2 then
    TLBI_RVAALE1OS(X[t]);
elsif PSTATE.EL == EL3 then
    TLBI_RVAALE1OS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.31 TLBI RVAE1, TLB Range Invalidate by VA, EL1

The TLBI RVAE1 characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA, and one of the following applies:
  - — The entry is from a level of lookup above the final level and matches the specified ASID.
  - — The entry is a global entry from the final level of lookup.
  - — The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - — If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime.
  - — If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*2^(5*SCALE +1) * Translation_Granule_Size)].

The invalidation applies to the PE that executes this System instruction.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - — If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - — If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - — If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - — If TTL==01 and BaseADDR[41:16] is not equal to 0000000000000000000000000000.
  - — If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

Configurations

This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI RVAE1 are UNDEFINED.

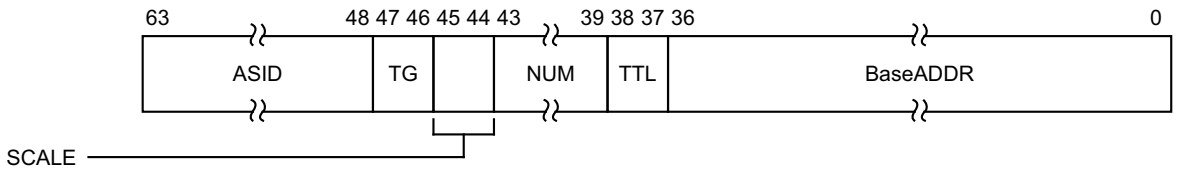
Attributes

TLBI RVAE1 is a 64-bit System instruction.

Field descriptions

The TLBI RVAE1 input value bit assignments are:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta



ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are RES0.

TG, bits [47:46]

Translation granule size.

`0b00` Reserved.

`0b01` 4K translation granule.

`0b10` 16K translation granule.

`0b11` 64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

`0b00` The entries in the range can be using any level for the translation table entries.

`0b01` When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries.

When using a 16KB translation granule, this value is reserved and hardware should treat this field as `0b00`.

`0b10` All entries to invalidate are Level 2 translation table entries.

`0b11` All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing the TLBI RVAE1 instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TLBI RVAE1{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0110	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.TLBIRVAE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.FB == '1' then
        TLBI_RVAE1IS(X[t]);
    else
        TLBI_RVAE1(X[t]);
elsif PSTATE.EL == EL2 then
    TLBI_RVAE1(X[t]);
elsif PSTATE.EL == EL3 then
    TLBI_RVAE1(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.32 TLBI RVAE1IS, TLB Range Invalidate by VA, EL1, Inner Shareable

The TLBI RVAE1IS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA, and one of the following applies:
  - — The entry is from a level of lookup above the final level and matches the specified ASID.
  - — The entry is a global entry from the final level of lookup.
  - — The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - — If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime.
  - — If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*2^(5*SCALE +1) * Translation_Granule_Size)].

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - — If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - — If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - — If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - — If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - — If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Configurations

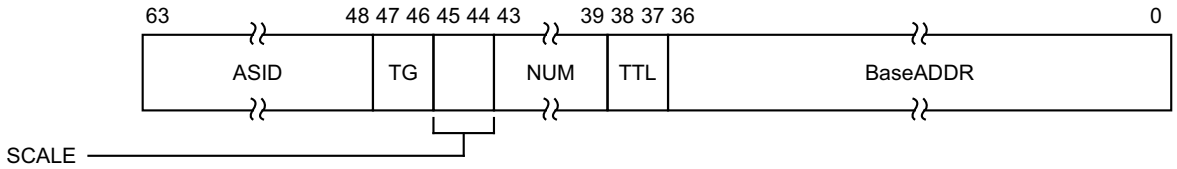
This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI RVAE1IS are UNDEFINED.

Attributes

TLBI RVAE1IS is a 64-bit System instruction.

Field descriptions

The TLBI RVAE1IS input value bit assignments are:



ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are RES0.

TG, bits [47:46]

Translation granule size.

`0b00` Reserved.

`0b01` 4K translation granule.

`0b10` 16K translation granule.

`0b11` 64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

`0b00` The entries in the range can be using any level for the translation table entries.

`0b01` When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries.

When using a 16KB translation granule, this value is reserved and hardware should treat this field as `0b00`.

`0b10` All entries to invalidate are Level 2 translation table entries.

`0b11` All entries to invalidate are Level 3 translation table entries.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing the TLBI RVAE1IS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI RVAE1IS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.TLBIRVAE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        TLBI_RVAE1IS(X[t]);
elsif PSTATE.EL == EL2 then
    TLBI_RVAE1IS(X[t]);
elsif PSTATE.EL == EL3 then
    TLBI_RVAE1IS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.33 TLBI RVAE1OS, TLB Range Invalidate by VA, EL1, Outer Shareable

The TLBI RVAE1OS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA, and one of the following applies:
  - — The entry is from a level of lookup above the final level and matches the specified ASID.
  - — The entry is a global entry from the final level of lookup.
  - — The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - — If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime.
  - — If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*2^(5*SCALE +1) * Translation_Granule_Size)].

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - — If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - — If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - — If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - — If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - — If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Configurations

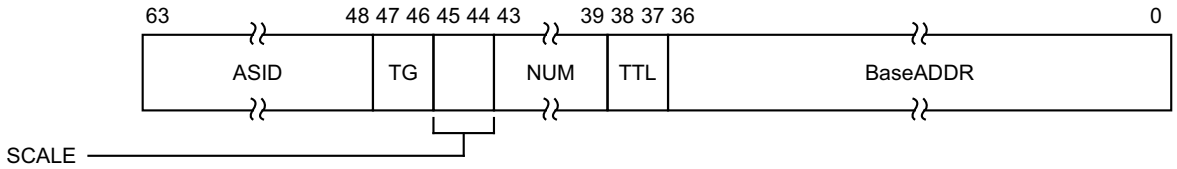
This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI RVAE1OS are UNDEFINED.

Attributes

TLBI RVAE1OS is a 64-bit System instruction.

Field descriptions

The TLBI RVAE1OS input value bit assignments are:



ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are RES0.

TG, bits [47:46]

Translation granule size.

`0b00` Reserved.

`0b01` 4K translation granule.

`0b10` 16K translation granule.

`0b11` 64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

`0b00` The entries in the range can be using any level for the translation table entries.

`0b01` When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries.

When using a 16KB translation granule, this value is reserved and hardware should treat this field as `0b00`.

`0b10` All entries to invalidate are Level 2 translation table entries.

`0b11` All entries to invalidate are Level 3 translation table entries.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing the TLBI RVAE1OS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI RVAE1OS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0101	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.TLBIRVAE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        TLBI_RVAE1OS(X[t]);
elsif PSTATE.EL == EL2 then
    TLBI_RVAE1OS(X[t]);
elsif PSTATE.EL == EL3 then
    TLBI_RVAE1OS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.34 TLBI RVAE2, TLB Range Invalidate by VA, EL2

The TLBI RVAE2 characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA in the specified range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*2^(5*SCALE +1) * Translation_Granule_Size)], using the EL2 or EL2&0 translation regime.
- If HCR_EL2.E2H == 0, the entry is from any level of the translation table walk.
- If HCR_EL2.E2H == 1, one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk and matches the specified ASID.

The invalidation applies to the PE that executes this System instruction.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

Configurations

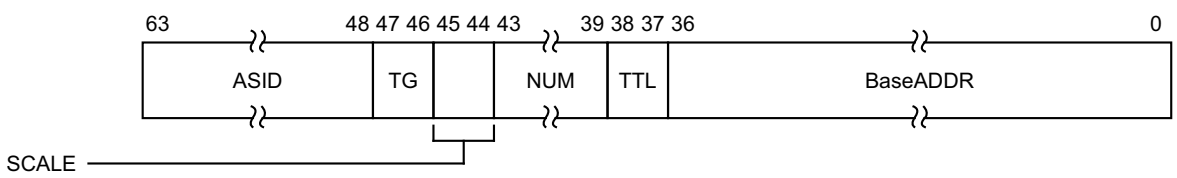
This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI RVAE2 are UNDEFINED.

Attributes

TLBI RVAE2 is a 64-bit System instruction.

Field descriptions

The TLBI RVAE2 input value bit assignments are:



ASID, bits [63:48]

When HCR_EL2.E2H == 1:

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are RES0.

Otherwise:

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

`0b00` Reserved.

`0b01` 4K translation granule.

`0b10` 16K translation granule.

`0b11` 64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

`0b00` The entries in the range can be using any level for the translation table entries.

`0b01` When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries.

When using a 16KB translation granule, this value is reserved and hardware should treat this field as `0b00`.

`0b10` All entries to invalidate are Level 2 translation table entries.

`0b11` All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing the TLBI RVAE2 instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI RVAE2{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0110	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBI_RVAE2(X[t]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    else
        TLBI_RVAE2(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.35 TLBI RVAE2IS, TLB Range Invalidate by VA, EL2, Inner Shareable

The TLBI RVAE2IS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA in the specified range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*2^(5*SCALE +1) * Translation_Granule_Size)], using the EL2 or EL2&0 translation regime.
- If HCR_EL2.E2H == 0, the entry is from any level of the translation table walk.
- If HCR_EL2.E2H == 1, one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk and matches the specified ASID.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

Configurations

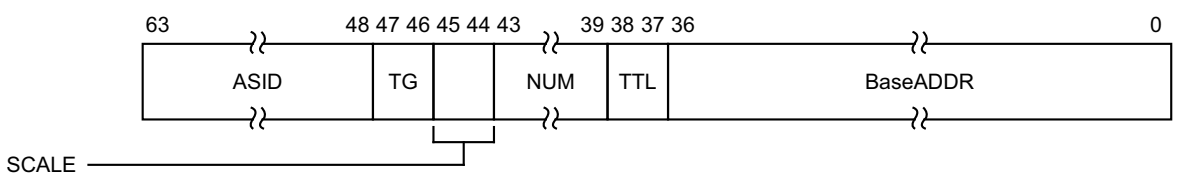
This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI RVAE2IS are UNDEFINED.

Attributes

TLBI RVAE2IS is a 64-bit System instruction.

Field descriptions

The TLBI RVAE2IS input value bit assignments are:



Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ASID, bits [63:48]

When HCR_EL2.E2H == 1:

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are RES0.

Otherwise:

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

`0b00` Reserved.

`0b01` 4K translation granule.

`0b10` 16K translation granule.

`0b11` 64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

`0b00` The entries in the range can be using any level for the translation table entries.

`0b01` When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries.

When using a 16KB translation granule, this value is reserved and hardware should treat this field as `0b00`.

`0b10` All entries to invalidate are Level 2 translation table entries.

`0b11` All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing the TLBI RVAE2IS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TLBI RVAE2IS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBI_RVAE2IS(X[t]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    else
        TLBI_RVAE2IS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.36 TLBI RVAE2OS, TLB Range Invalidate by VA, EL2, Outer Shareable

The TLBI RVAE2OS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA in the specified range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*2^(5*SCALE +1) * Translation_Granule_Size)], using the EL2 or EL2&0 translation regime.
- If HCR_EL2.E2H == 0, the entry is from any level of the translation table walk.
- If HCR_EL2.E2H == 1, one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk and matches the specified ASID.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

Configurations

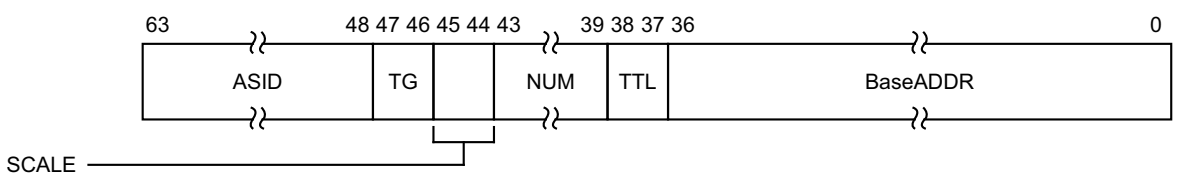
This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI RVAE2OS are UNDEFINED.

Attributes

TLBI RVAE2OS is a 64-bit System instruction.

Field descriptions

The TLBI RVAE2OS input value bit assignments are:



Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ASID, bits [63:48]

When HCR_EL2.E2H == 1:

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are RES0.

Otherwise:

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

`0b00` Reserved.

`0b01` 4K translation granule.

`0b10` 16K translation granule.

`0b11` 64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

`0b00` The entries in the range can be using any level for the translation table entries.

`0b01` When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries.

When using a 16KB translation granule, this value is reserved and hardware should treat this field as `0b00`.

`0b10` All entries to invalidate are Level 2 translation table entries.

`0b11` All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing the TLBI RVAE2OS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TLBI RVAE2OS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0101	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBI_RVAE2OS(X[t]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    else
        TLBI_RVAE2OS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.37 TLBI RVAE3, TLB Range Invalidate by VA, EL3

The TLBI RVAE3 characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be used to translate the specified VA using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*2^(5*SCALE +1) * Translation_Granule_Size)].

The invalidation applies to the PE that executes this System instruction.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

Configurations

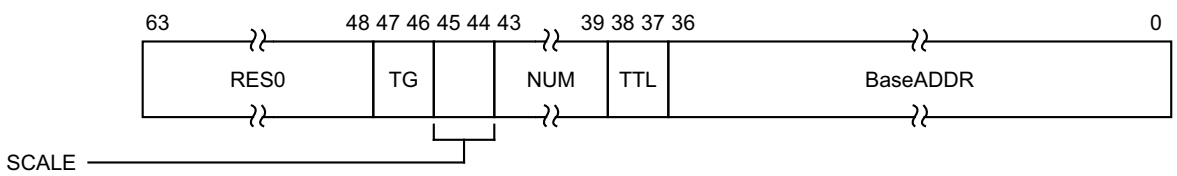
This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI RVAE3 are UNDEFINED.

Attributes

TLBI RVAE3 is a 64-bit System instruction.

Field descriptions

The TLBI RVAE3 input value bit assignments are:



Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

`0b00` Reserved.

`0b01` 4K translation granule.

`0b10` 16K translation granule.

`0b11` 64K translation granule.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

`0b00` The entries in the range can be using any level for the translation table entries.

`0b01` When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries.

When using a 16KB translation granule, this value is reserved and hardware should treat this field as `0b00`.

`0b10` All entries to invalidate are Level 2 translation table entries.

`0b11` All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing the TLBI RVAE3 instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI RVAE3{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0110	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    TLBI_RVAE3(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.38 TLBI RVAE3IS, TLB Range Invalidate by VA, EL3, Inner Shareable

The TLBI RVAE3IS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be used to translate the specified VA using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*2^(5*SCALE +1) * Translation_Granule_Size)].

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

Configurations

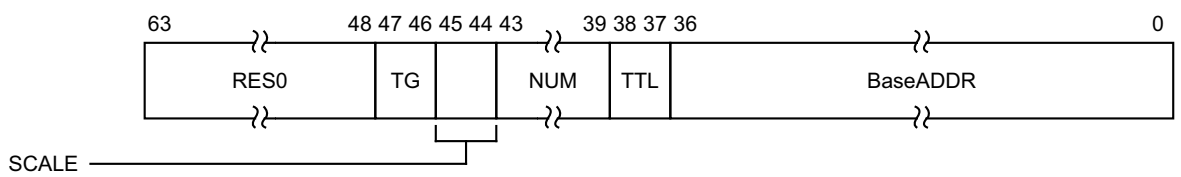
This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI RVAE3IS are UNDEFINED.

Attributes

TLBI RVAE3IS is a 64-bit System instruction.

Field descriptions

The TLBI RVAE3IS input value bit assignments are:



Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

`0b00` Reserved.

`0b01` 4K translation granule.

`0b10` 16K translation granule.

`0b11` 64K translation granule.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

`0b00` The entries in the range can be using any level for the translation table entries.

`0b01` When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries.

When using a 16KB translation granule, this value is reserved and hardware should treat this field as `0b00`.

`0b10` All entries to invalidate are Level 2 translation table entries.

`0b11` All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing the TLBI RVAE3IS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI RVAE3IS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    TLBI_RVAE3IS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.39 TLBI RVAE3OS, TLB Range Invalidate by VA, EL3, Outer Shareable

The TLBI RVAE3OS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be used to translate the specified VA using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*2^(5*SCALE +1) * Translation_Granule_Size)].

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

Configurations

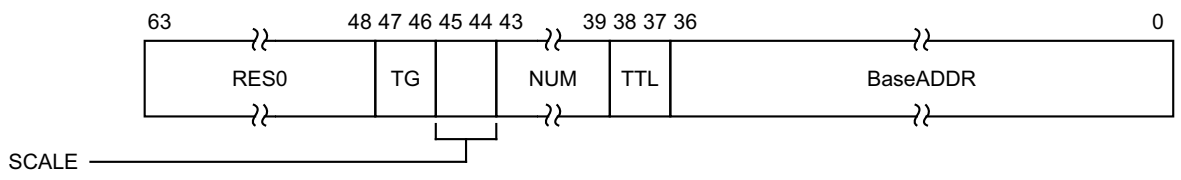
This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI RVAE3OS are UNDEFINED.

Attributes

TLBI RVAE3OS is a 64-bit System instruction.

Field descriptions

The TLBI RVAE3OS input value bit assignments are:



Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

`0b00` Reserved.

`0b01` 4K translation granule.

`0b10` 16K translation granule.

`0b11` 64K translation granule.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

`0b00` The entries in the range can be using any level for the translation table entries.

`0b01` When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries.

When using a 16KB translation granule, this value is reserved and hardware should treat this field as `0b00`.

`0b10` All entries to invalidate are Level 2 translation table entries.

`0b11` All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing the TLBI RVAE3OS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI RVAE3OS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0101	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    TLBI_RVAE3OS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.40 TLBI RVALE1, TLB Range Invalidate by VA, Last level, EL1

The TLBI RVALE1 characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA, and one of the following applies:
  - — The entry is a global entry from the final level of lookup.
  - — The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - — If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime.
  - — If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*2^(5*SCALE +1) * Translation_Granule_Size)].

The invalidation applies to the PE that executes this System instruction.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - — If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - — If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - — If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - — If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - — If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

For more information about the architectural requirements for this System instruction see *Invalidation of TLB entries from stage 2 translations* on page D5-2680.

Configurations

This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI RVALE1 are UNDEFINED.

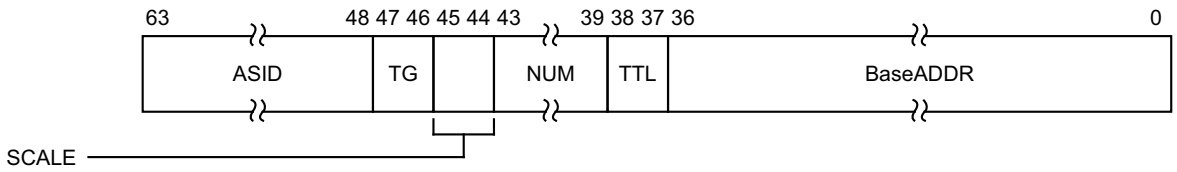
Attributes

TLBI RVALE1 is a 64-bit System instruction.

Field descriptions

The TLBI RVALE1 input value bit assignments are:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta



ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are RES0.

TG, bits [47:46]

Translation granule size.

`0b00` Reserved.

`0b01` 4K translation granule.

`0b10` 16K translation granule.

`0b11` 64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

`0b00` The entries in the range can be using any level for the translation table entries.

`0b01` When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries.

When using a 16KB translation granule, this value is reserved and hardware should treat this field as `0b00`.

`0b10` All entries to invalidate are Level 2 translation table entries.

`0b11` All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing the TLBI RVALE1 instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TLBI RVALE1{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0110	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.TLBIRVALE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.FB == '1' then
        TLBI_RVALE1IS(X[t]);
    else
        TLBI_RVALE1(X[t]);
elsif PSTATE.EL == EL2 then
    TLBI_RVALE1(X[t]);
elsif PSTATE.EL == EL3 then
    TLBI_RVALE1(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.41 TLBI RVALE1IS, TLB Range Invalidate by VA, Last level, EL1, Inner Shareable

The TLBI RVALE1IS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA, and one of the following applies:
  - — The entry is a global entry from the final level of lookup.
  - — The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - — If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime.
  - — If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*2^(5*SCALE +1) * Translation_Granule_Size)].

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - — If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - — If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - — If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - — If TTL==01 and BaseADDR[41:16] is not equal to 0000000000000000000000000.
  - — If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

Configurations

This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI RVALE1IS are UNDEFINED.

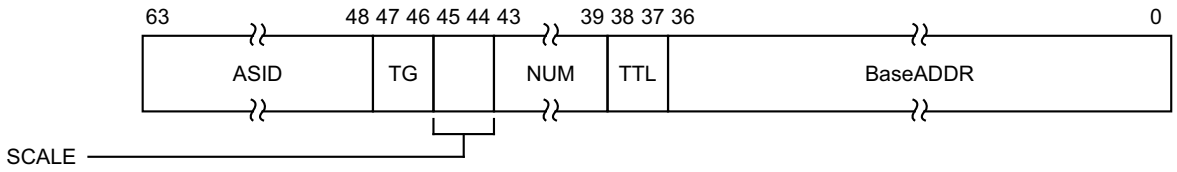
Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Attributes

TLBI RVALE1IS is a 64-bit System instruction.

Field descriptions

The TLBI RVALE1IS input value bit assignments are:



ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are RES0.

TG, bits [47:46]

Translation granule size.

`0b00` Reserved.

`0b01` 4K translation granule.

`0b10` 16K translation granule.

`0b11` 64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

`0b00` The entries in the range can be using any level for the translation table entries.

`0b01` When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries.

When using a 16KB translation granule, this value is reserved and hardware should treat this field as `0b00`.

`0b10` All entries to invalidate are Level 2 translation table entries.

`0b11` All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing the TLBI RVALE1IS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI RVALE1IS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0010	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.TLBIRVALE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        TLBI_RVALE1IS(X[t]);
elsif PSTATE.EL == EL2 then
    TLBI_RVALE1IS(X[t]);
elsif PSTATE.EL == EL3 then
    TLBI_RVALE1IS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.42 TLBI RVALE1OS, TLB Range Invalidate by VA, Last level, EL1, Outer Shareable

The TLBI RVALE1OS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime.
  - If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*2^(5*SCALE +1) * Translation_Granule_Size)].

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

Configurations

This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI RVALE1OS are UNDEFINED.

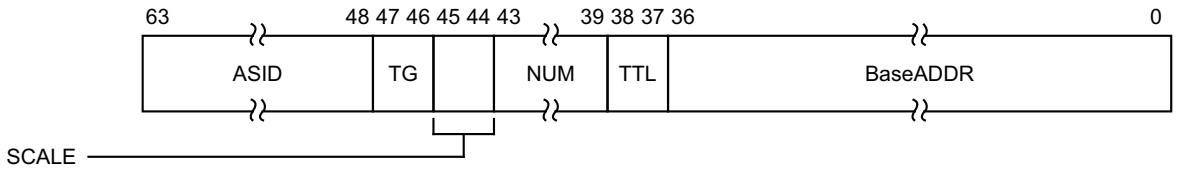
Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Attributes

TLBI RVALE1OS is a 64-bit System instruction.

Field descriptions

The TLBI RVALE1OS input value bit assignments are:



ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are RES0.

TG, bits [47:46]

Translation granule size.

`0b00` Reserved.

`0b01` 4K translation granule.

`0b10` 16K translation granule.

`0b11` 64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

`0b00` The entries in the range can be using any level for the translation table entries.

`0b01` When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries.

When using a 16KB translation granule, this value is reserved and hardware should treat this field as `0b00`.

`0b10` All entries to invalidate are Level 2 translation table entries.

`0b11` All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing the TLBI RVALE1OS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI RVALE1OS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0101	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.TLBIRVALE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        TLBI_RVALE1OS(X[t]);
elsif PSTATE.EL == EL2 then
    TLBI_RVALE1OS(X[t]);
elsif PSTATE.EL == EL3 then
    TLBI_RVALE1OS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.43 TLBI RVALE2, TLB Range Invalidate by VA, Last level, EL2

The TLBI RVALE2 characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA in the specified range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*2^(5*SCALE +1) * Translation_Granule_Size)] using the EL2 or EL2&0 translation regime.
- If HCR_EL2.E2H == 0, the entry is from the final level of the translation table walk.
- If HCR_EL2.E2H == 1, one of the following applies:
  - The entry is a global entry from the final level of translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk and matches the specified ASID.

The invalidation applies to the PE that executes this System instruction.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

Configurations

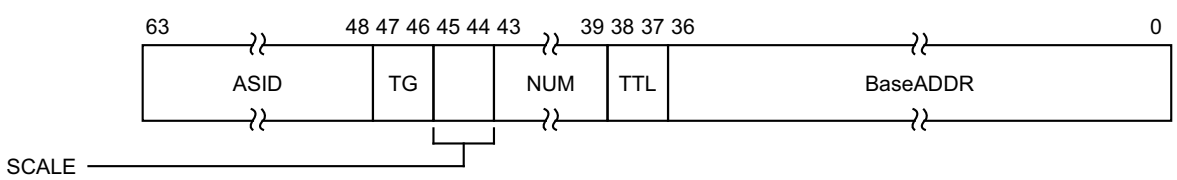
This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI RVALE2 are UNDEFINED.

Attributes

TLBI RVALE2 is a 64-bit System instruction.

Field descriptions

The TLBI RVALE2 input value bit assignments are:



ASID, bits [63:48]

When HCR_EL2.E2H == 1:

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are RES0.

Otherwise:

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

`0b00` Reserved.

`0b01` 4K translation granule.

`0b10` 16K translation granule.

`0b11` 64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

`0b00` The entries in the range can be using any level for the translation table entries.

`0b01` When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries.

When using a 16KB translation granule, this value is reserved and hardware should treat this field as `0b00`.

`0b10` All entries to invalidate are Level 2 translation table entries.

`0b11` All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing the TLBI RVALE2 instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI RVALE2{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0110	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBI_RVALE2(X[t]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    else
        TLBI_RVALE2(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.44 TLBI RVALE2IS, TLB Range Invalidate by VA, Last level, EL2, Inner Shareable

The TLBI RVALE2IS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA in the specified range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*2^(5*SCALE +1) * Translation_Granule_Size)] using the EL2 or EL2&0 translation regime.
- If HCR_EL2.E2H == 0, the entry is from the final level of the translation table walk.
- If HCR_EL2.E2H == 1, one of the following applies:
  - The entry is a global entry from the final level of translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk and matches the specified ASID.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

Configurations

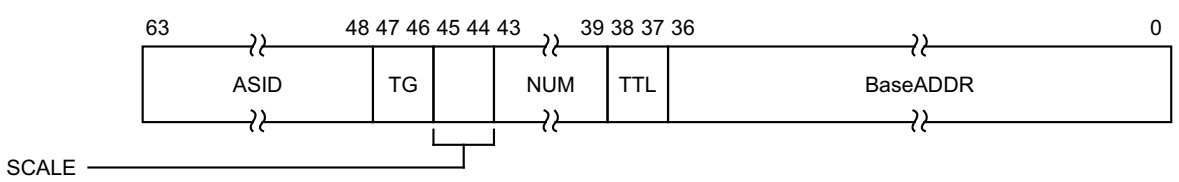
This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI RVALE2IS are UNDEFINED.

Attributes

TLBI RVALE2IS is a 64-bit System instruction.

Field descriptions

The TLBI RVALE2IS input value bit assignments are:



ASID, bits [63:48]

When HCR_EL2.E2H == 1:

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are RES0.

Otherwise:

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

`0b00` Reserved.

`0b01` 4K translation granule.

`0b10` 16K translation granule.

`0b11` 64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

`0b00` The entries in the range can be using any level for the translation table entries.

`0b01` When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries.

When using a 16KB translation granule, this value is reserved and hardware should treat this field as `0b00`.

`0b10` All entries to invalidate are Level 2 translation table entries.

`0b11` All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing the TLBI RVALE2IS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI RVALE2IS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0010	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        if EL2Enabled() && HCR_EL2.NV == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            UNDEFINED;
    elsif PSTATE.EL == EL2 then
        TLBI_RVALE2IS(X[t]);
    elsif PSTATE.EL == EL3 then
        if !EL2Enabled() then
            UNDEFINED;
        else
            TLBI_RVALE2IS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.45 TLBI RVALE2OS, TLB Range Invalidate by VA, Last level, EL2, Outer Shareable

The TLBI RVALE2OS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA in the specified range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*2^(5*SCALE +1) * Translation_Granule_Size)] using the EL2 or EL2&0 translation regime.
- If HCR_EL2.E2H == 0, the entry is from the final level of the translation table walk.
- If HCR_EL2.E2H == 1, one of the following applies:
  - The entry is a global entry from the final level of translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk and matches the specified ASID.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

Configurations

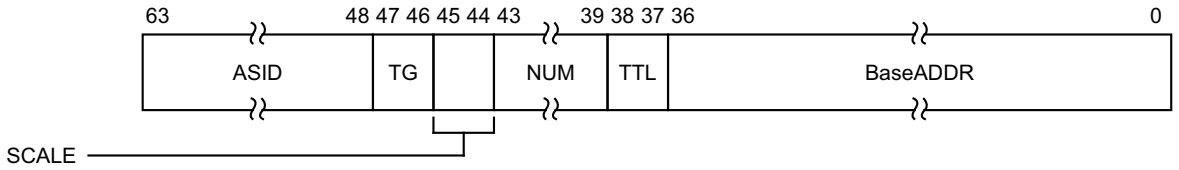
This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI RVALE2OS are UNDEFINED.

Attributes

TLBI RVALE2OS is a 64-bit System instruction.

Field descriptions

The TLBI RVALE2OS input value bit assignments are:



ASID, bits [63:48]

When HCR_EL2.E2H == 1:

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are RES0.

Otherwise:

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

`0b00` Reserved.

`0b01` 4K translation granule.

`0b10` 16K translation granule.

`0b11` 64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

`0b00` The entries in the range can be using any level for the translation table entries.

`0b01` When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries.

When using a 16KB translation granule, this value is reserved and hardware should treat this field as `0b00`.

`0b10` All entries to invalidate are Level 2 translation table entries.

`0b11` All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing the TLBI RVALE2OS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI RVALE2OS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0101	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBI_RVALE2OS(X[t]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    else
        TLBI_RVALE2OS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.46 TLBI RVALE3, TLB Range Invalidate by VA, Last level, EL3

The TLBI RVALE3 characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be used to translate the specified VA using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*2^(5*SCALE +1) * Translation_Granule_Size)].

The invalidation applies to the PE that executes this System instruction.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - — If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - — If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - — If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - — If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - — If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

Configurations

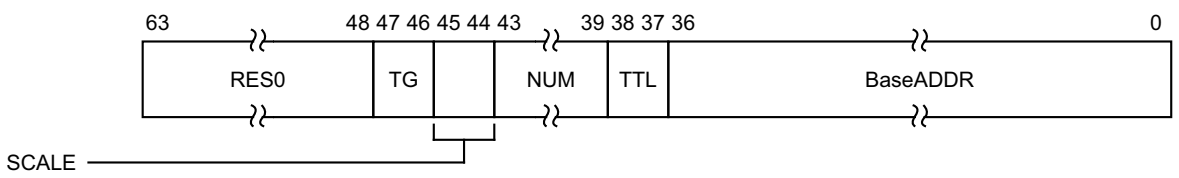
This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI RVALE3 are UNDEFINED.

Attributes

TLBI RVALE3 is a 64-bit System instruction.

Field descriptions

The TLBI RVALE3 input value bit assignments are:



Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

`0b00` Reserved.

`0b01` 4K translation granule.

`0b10` 16K translation granule.

`0b11` 64K translation granule.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

`0b00` The entries in the range can be using any level for the translation table entries.

`0b01` When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries.

When using a 16KB translation granule, this value is reserved and hardware should treat this field as `0b00`.

`0b10` All entries to invalidate are Level 2 translation table entries.

`0b11` All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing the TLBI RVALE3 instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI RVALE3{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0110	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    TLBI_RVALE3(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.47 TLBI RVALE3IS, TLB Range Invalidate by VA, Last level, EL3, Inner Shareable

The TLBI RVALE3IS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be used to translate the specified VA using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*2^(5*SCALE +1) * Translation_Granule_Size)].

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

Configurations

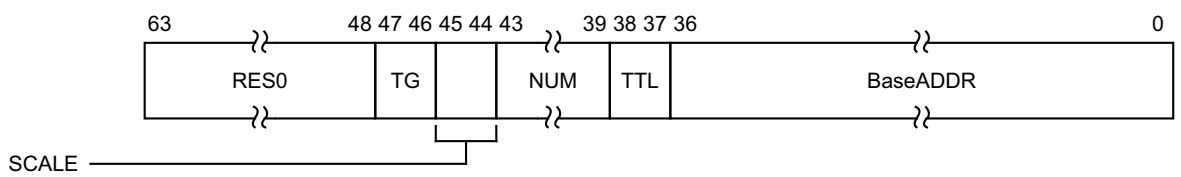
This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI RVALE3IS are UNDEFINED.

Attributes

TLBI RVALE3IS is a 64-bit System instruction.

Field descriptions

The TLBI RVALE3IS input value bit assignments are:



Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

`0b00` Reserved.

`0b01` 4K translation granule.

`0b10` 16K translation granule.

`0b11` 64K translation granule.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

`0b00` The entries in the range can be using any level for the translation table entries.

`0b01` When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries.

When using a 16KB translation granule, this value is reserved and hardware should treat this field as `0b00`.

`0b10` All entries to invalidate are Level 2 translation table entries.

`0b11` All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing the TLBI RVALE3IS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI RVALE3IS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0010	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    TLBI_RVALE3IS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.48 TLBI RVALE3OS, TLB Range Invalidate by VA, Last level, EL3, Outer Shareable

The TLBI RVALE3OS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be used to translate the specified VA using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*2^(5*SCALE +1) * Translation_Granule_Size)].

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

Configurations

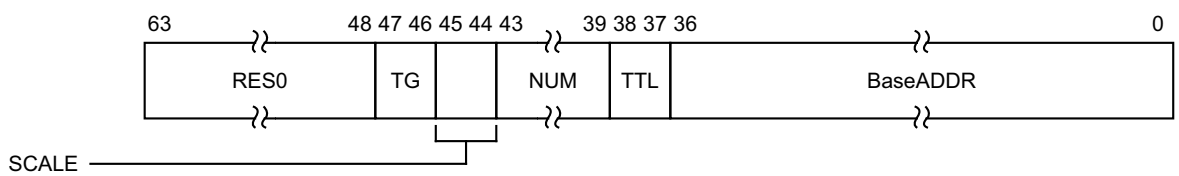
This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI RVALE3OS are UNDEFINED.

Attributes

TLBI RVALE3OS is a 64-bit System instruction.

Field descriptions

The TLBI RVALE3OS input value bit assignments are:



Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

`0b00` Reserved.

`0b01` 4K translation granule.

`0b10` 16K translation granule.

`0b11` 64K translation granule.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

`0b00` The entries in the range can be using any level for the translation table entries.

`0b01` When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries.

When using a 16KB translation granule, this value is reserved and hardware should treat this field as `0b00`.

`0b10` All entries to invalidate are Level 2 translation table entries.

`0b11` All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing the TLBI RVALE3OS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI RVALE3OS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0101	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    TLBI_RVALE3OS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.49 TLBI VAAE1, TLB Invalidate by VA, All ASID, EL1

The TLBI VAAE1 characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - — If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime.
  - — If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime.

The invalidation applies to the PE that executes this System instruction.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries, and non-global entries with any ASID.

Configurations

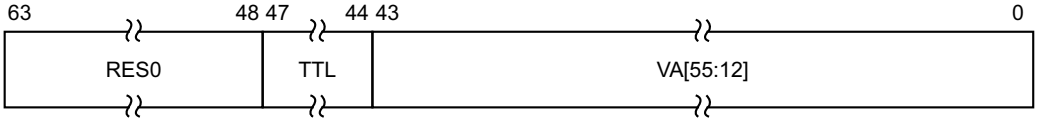
There are no configuration notes.

Attributes

TLBI VAAE1 is a 64-bit System instruction.

Field descriptions

The TLBI VAAE1 input value bit assignments are:



Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When ARMv8.4-TTL is implemented:

Translation Table Level. Indicates the level of the page table walk that holds the leaf entry for the address being invalidated.

`0b00xx` No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

`0b01xx` The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b10xx` The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b10` : Level 2.

`0b11` : Level 3.

`0b11xx` The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing the TLBI VAAE1 instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI VAAE1{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0111	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.TLBIVAAE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.FB == '1' then
        TLBI_VAAE1IS(X[t]);
    else
        TLBI_VAAE1(X[t]);
elsif PSTATE.EL == EL2 then
    TLBI_VAAE1(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    TLBI_VAAE1(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.50 TLBI VAAE1IS, TLB Invalidate by VA, All ASID, EL1, Inner Shareable

The TLBI VAAE1IS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - — If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime.
  - — If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries, and non-global entries with any ASID.

Configurations

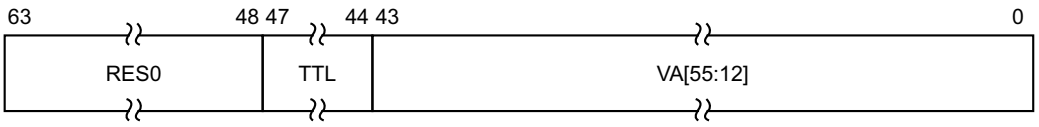
There are no configuration notes.

Attributes

TLBI VAAE1IS is a 64-bit System instruction.

Field descriptions

The TLBI VAAE1IS input value bit assignments are:



Bits [63:48]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TTL, bits [47:44]

When ARMv8.4-TTL is implemented:

Translation Table Level. Indicates the level of the page table walk that holds the leaf entry for the address being invalidated.

`0b00xx` No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

`0b01xx` The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

`0b10xx` The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b10` : Level 2.

`0b11` : Level 3.

`0b11xx` The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing the TLBI VAAE1IS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TLBI VAAE1IS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0011	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.TLBIVAAE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        TLBI_VAAE1IS(X[t]);
elsif PSTATE.EL == EL2 then
    TLBI_VAAE1IS(X[t]);
elsif PSTATE.EL == EL3 then
    TLBI_VAAE1IS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.51 TLBI VAAE1OS, TLB Invalidate by VA, All ASID, EL1, Outer Shareable

The TLBI VAAE1OS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime.
  - If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries, and non-global entries with any ASID.

Configurations

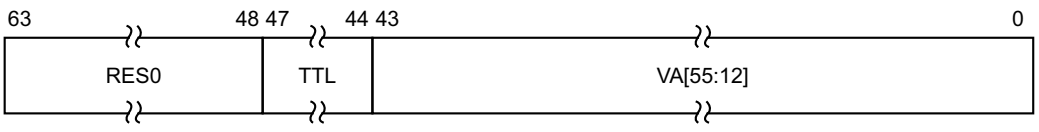
This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI VAAE1OS are UNDEFINED.

Attributes

TLBI VAAE1OS is a 64-bit System instruction.

Field descriptions

The TLBI VAAE1OS input value bit assignments are:



Bits [63:48]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TTL, bits [47:44]

When ARMv8.4-TTL is implemented:

Translation Table Level. Indicates the level of the page table walk that holds the leaf entry for the address being invalidated.

`0b00xx` No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

`0b01xx` The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

`0b10xx` The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b10` : Level 2.

`0b11` : Level 3.

`0b11xx` The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing the TLBI VAAE1OS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TLBI VAAE1OS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0001	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.TLBIVAAE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        TLBI_VAAE1OS(X[t]);
elsif PSTATE.EL == EL2 then
    TLBI_VAAE1OS(X[t]);
elsif PSTATE.EL == EL3 then
    TLBI_VAAE1OS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.52 TLBI VAALE1, TLB Invalidate by VA, All ASID, Last level, EL1

The TLBI VAALE1 characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.
- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime.
  - If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime.

The invalidation applies to the PE that executes this System instruction.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries, and non-global entries with any ASID.

Configurations

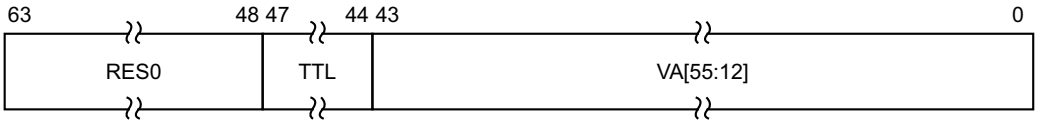
There are no configuration notes.

Attributes

TLBI VAALE1 is a 64-bit System instruction.

Field descriptions

The TLBI VAALE1 input value bit assignments are:



Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When ARMv8.4-TTL is implemented:

Translation Table Level. Indicates the level of the page table walk that holds the leaf entry for the address being invalidated.

`0b00xx` No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

`0b01xx` The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b10xx` The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b10` : Level 2.

`0b11` : Level 3.

`0b11xx` The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing the TLBI VAALE1 instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI VAALE1{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0111	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.TLBIVAALE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.FB == '1' then
        TLBI_VAALE1IS(X[t]);
    else
        TLBI_VAALE1(X[t]);
elsif PSTATE.EL == EL2 then
    TLBI_VAALE1(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    TLBI_VAALE1(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.53 TLBI VAALE1IS, TLB Invalidate by VA, All ASID, Last Level, EL1, Inner Shareable

The TLBI VAALE1IS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.
- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime.
  - If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries, and non-global entries with any ASID.

Configurations

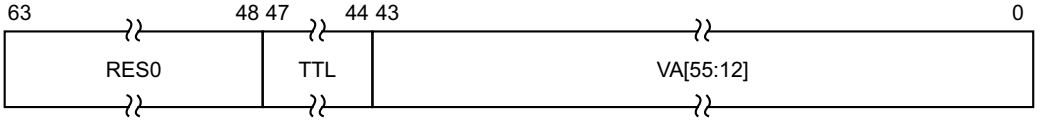
There are no configuration notes.

Attributes

TLBI VAALE1IS is a 64-bit System instruction.

Field descriptions

The TLBI VAALE1IS input value bit assignments are:



Bits [63:48]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TTL, bits [47:44]

When ARMv8.4-TTL is implemented:

Translation Table Level. Indicates the level of the page table walk that holds the leaf entry for the address being invalidated.

`0b00xx` No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

`0b01xx` The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

`0b10xx` The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b10` : Level 2.

`0b11` : Level 3.

`0b11xx` The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing the TLBI VAALE1IS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TLBI VAALE1IS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0011	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.TLBIVAALE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        TLBI_VAALE1IS(X[t]);
elsif PSTATE.EL == EL2 then
    TLBI_VAALE1IS(X[t]);
elsif PSTATE.EL == EL3 then
    TLBI_VAALE1IS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.54 TLBI VAALE1OS, TLB Invalidate by VA, All ASID, Last Level, EL1, Outer Shareable

The TLBI VAALE1OS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.
- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - — If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime.
  - — If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries, and non-global entries with any ASID.

Configurations

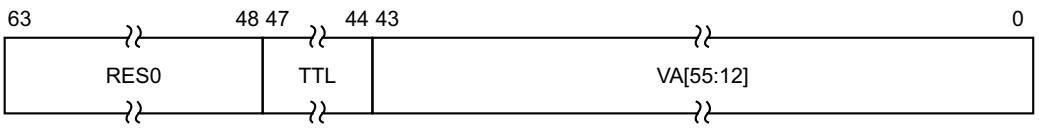
This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI VAALE1OS are UNDEFINED.

Attributes

TLBI VAALE1OS is a 64-bit System instruction.

Field descriptions

The TLBI VAALE1OS input value bit assignments are:



Bits [63:48]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TTL, bits [47:44]

When ARMv8.4-TTL is implemented:

Translation Table Level. Indicates the level of the page table walk that holds the leaf entry for the address being invalidated.

`0b00xx` No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

`0b01xx` The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

`0b10xx` The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b10` : Level 2.

`0b11` : Level 3.

`0b11xx` The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing the TLBI VAALE1OS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TLBI VAALE1OS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0001	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.TLBIVAALE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        TLBI_VAALE1OS(X[t]);
elsif PSTATE.EL == EL2 then
    TLBI_VAALE1OS(X[t]);
elsif PSTATE.EL == EL3 then
    TLBI_VAALE1OS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.55 TLBI VAE1, TLB Invalidate by VA, EL1

The TLBI VAE1 characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA, and one of the following applies:
  - — The entry is from a level of lookup above the final level and matches the specified ASID.
  - — The entry is a global entry from the final level of lookup.
  - — The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - — If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime.
  - — If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime.

The invalidation applies to the PE that executes this System instruction.

Configurations

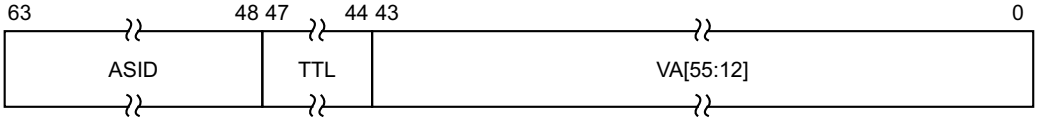
There are no configuration notes.

Attributes

TLBI VAE1 is a 64-bit System instruction.

Field descriptions

The TLBI VAE1 input value bit assignments are:



ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TTL, bits [47:44]

When ARMv8.4-TTL is implemented:

Translation Table Level. Indicates the level of the page table walk that holds the leaf entry for the address being invalidated.

`0b00xx` No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

`0b01xx` The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

`0b10xx` The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b10` : Level 2.

`0b11` : Level 3.

`0b11xx` The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing the TLBI VAE1 instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TLBI VAE1{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0111	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.TLBIVAE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.FB == '1' then
        TLBI_VAE1IS(X[t]);
    else
        TLBI_VAE1(X[t]);
elsif PSTATE.EL == EL2 then
    TLBI_VAE1(X[t]);
elsif PSTATE.EL == EL3 then
    TLBI_VAE1(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.56 TLBI VAE1IS, TLB Invalidate by VA, EL1, Inner Shareable

The TLBI VAE1IS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA, and one of the following applies:
  - — The entry is from a level of lookup above the final level and matches the specified ASID.
  - — The entry is a global entry from the final level of lookup.
  - — The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - — If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime.
  - — If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Configurations

There are no configuration notes.

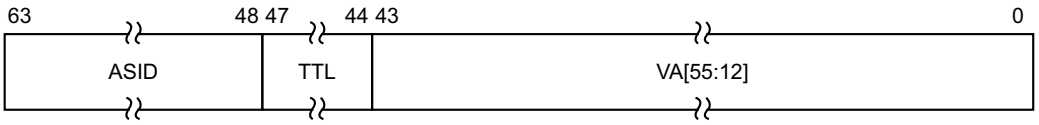
Attributes

TLBI VAE1IS is a 64-bit System instruction.

Field descriptions

The TLBI VAE1IS input value bit assignments are:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta



ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are RES0.

TTL, bits [47:44]

When ARMv8.4-TTL is implemented:

Translation Table Level. Indicates the level of the page table walk that holds the leaf entry for the address being invalidated.

`0b00xx` No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

`0b01xx` The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

`0b10xx` The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b10` : Level 2.

`0b11` : Level 3.

`0b11xx` The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing the TLBI VAE1IS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI VAE1IS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0011	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.TLBIVAE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        TLBI_VAE1IS(X[t]);
elsif PSTATE.EL == EL2 then
    TLBI_VAE1IS(X[t]);
elsif PSTATE.EL == EL3 then
    TLBI_VAE1IS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.57 TLBI VAE1OS, TLB Invalidate by VA, EL1, Outer Shareable

The TLBI VAE1OS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA, and one of the following applies:
  - — The entry is from a level of lookup above the final level and matches the specified ASID.
  - — The entry is a global entry from the final level of lookup.
  - — The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - — If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime.
  - — If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Configurations

This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI VAE1OS are UNDEFINED.

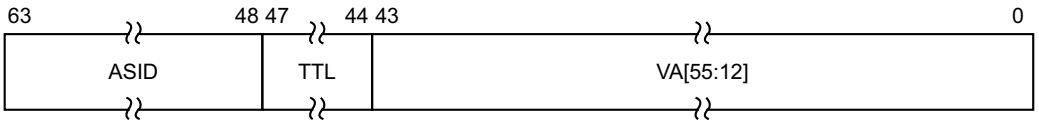
Attributes

TLBI VAE1OS is a 64-bit System instruction.

Field descriptions

The TLBI VAE1OS input value bit assignments are:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta



ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are RES0.

TTL, bits [47:44]

When ARMv8.4-TTL is implemented:

Translation Table Level. Indicates the level of the page table walk that holds the leaf entry for the address being invalidated.

`0b00xx` No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

`0b01xx` The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

`0b10xx` The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b10` : Level 2.

`0b11` : Level 3.

`0b11xx` The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing the TLBI VAE1OS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI VAE1OS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.TLBIVAE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        TLBI_VAE1OS(X[t]);
elsif PSTATE.EL == EL2 then
    TLBI_VAE1OS(X[t]);
elsif PSTATE.EL == EL3 then
    TLBI_VAE1OS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.58 TLBI VAE2, TLB Invalidate by VA, EL2

The TLBI VAE2 characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be required to translate the specified VA using the EL2 or the EL2&0 translation regime.
- If HCR_EL2.E2H == 0, the entry is from any level of the translation table walk.
- If HCR_EL2.E2H == 1 one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk and matches the specified ASID.

The invalidation applies to the PE that executes this System instruction.

Configurations

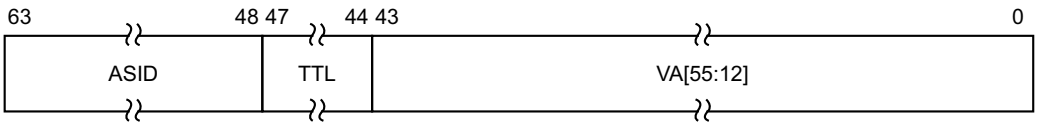
There are no configuration notes.

Attributes

TLBI VAE2 is a 64-bit System instruction.

Field descriptions

The TLBI VAE2 input value bit assignments are:



ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are RES0.

TTL, bits [47:44]

When ARMv8.4-TTL is implemented:

Translation Table Level. Indicates the level of the page table walk that holds the leaf entry for the address being invalidated.

`0b00xx` No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

`0b01xx` The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b10` : Level 2.

`0b11` : Level 3.

`0b10xx` The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b10` : Level 2.

`0b11` : Level 3.

`0b11xx` The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing the TLBI VAE2 instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI VAE2{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0111	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBI_VAE2(X[t]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    UNDEFINED;
else
    TLBI_VAE2(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.59 TLBI VAE2IS, TLB Invalidate by VA, EL2, Inner Shareable

The TLBI VAE2IS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be required to translate the specified VA using the EL2 or the EL2&0 translation regime.
- If HCR_EL2.E2H == 0, the entry is from any level of the translation table walk.
- If HCR_EL2.E2H == 1 one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk and matches the specified ASID.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configurations

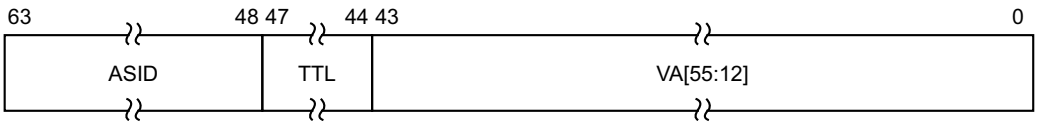
There are no configuration notes.

Attributes

TLBI VAE2IS is a 64-bit System instruction.

Field descriptions

The TLBI VAE2IS input value bit assignments are:



ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are RES0.

TTL, bits [47:44]

When ARMv8.4-TTL is implemented:

Translation Table Level. Indicates the level of the page table walk that holds the leaf entry for the address being invalidated.

`0b00xx` No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

`0b01xx` The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b10` : Level 2.

`0b11` : Level 3.

`0b10xx` The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b10` : Level 2.

`0b11` : Level 3.

`0b11xx` The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing the TLBI VAE2IS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI VAE2IS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0011	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBI_VAE2IS(X[t]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        UNDEFINED;
    else
        TLBI_VAE2IS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.60 TLBI VAE2OS, TLB Invalidate by VA, EL2, Outer Shareable

The TLBI VAE2OS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be required to translate the specified VA using the EL2 or the EL2&0 translation regime.
- If HCR_EL2.E2H == 0, the entry is from any level of the translation table walk.
- If HCR_EL2.E2H == 1 one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk and matches the specified ASID.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Configurations

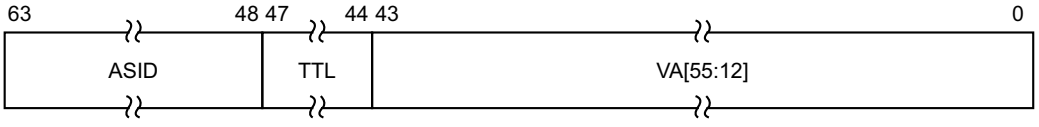
This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI VAE2OS are UNDEFINED.

Attributes

TLBI VAE2OS is a 64-bit System instruction.

Field descriptions

The TLBI VAE2OS input value bit assignments are:



ASID, bits [63:48]

When HCR_EL2.E2H == 1:

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are RES0.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TTL, bits [47:44]

When ARMv8.4-TTL is implemented:

Translation Table Level. Indicates the level of the page table walk that holds the leaf entry for the address being invalidated.

`0b00xx` No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

`0b01xx` The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

`0b10xx` The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b10` : Level 2.

`0b11` : Level 3.

`0b11xx` The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing the TLBI VAE2OS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TLBI VAE2OS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBI_VAE2OS(X[t]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    else
        TLBI_VAE2OS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.61 TLBI VAE3, TLB Invalidate by VA, EL3

The TLBI VAE3 characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to the PE that executes this System instruction.

Configurations

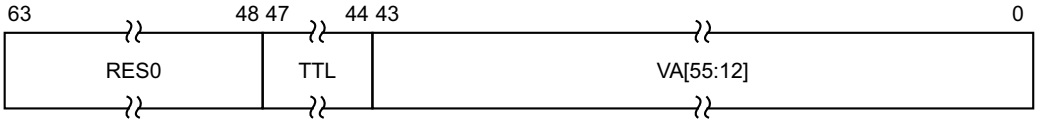
There are no configuration notes.

Attributes

TLBI VAE3 is a 64-bit System instruction.

Field descriptions

The TLBI VAE3 input value bit assignments are:



Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When ARMv8.4-TTL is implemented:

Translation Table Level. Indicates the level of the page table walk that holds the leaf entry for the address being invalidated.

`0b00xx` No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

`0b01xx` The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

`0b10xx` The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b10` : Level 2.

`0b11` : Level 3.

`0b11xx` The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing the TLBI VAE3 instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI VAE3{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0111	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    TLBI_VAE3(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.62 TLBI VAE3IS, TLB Invalidate by VA, EL3, Inner Shareable

The TLBI VAE3IS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configurations

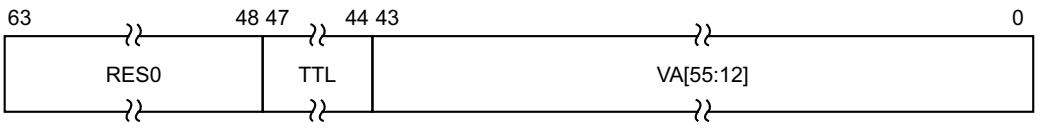
There are no configuration notes.

Attributes

TLBI VAE3IS is a 64-bit System instruction.

Field descriptions

The TLBI VAE3IS input value bit assignments are:



Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When ARMv8.4-TTL is implemented:

Translation Table Level. Indicates the level of the page table walk that holds the leaf entry for the address being invalidated.

`0b00xx` No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

`0b01xx` The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

`0b10xx` The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b10` : Level 2.

`0b11` : Level 3.

`0b11xx` The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing the TLBI VAE3IS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI VAE3IS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0011	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    TLBI_VAE3IS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.63 TLBI VAE3OS, TLB Invalidate by VA, EL3, Outer Shareable

The TLBI VAE3OS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Configurations

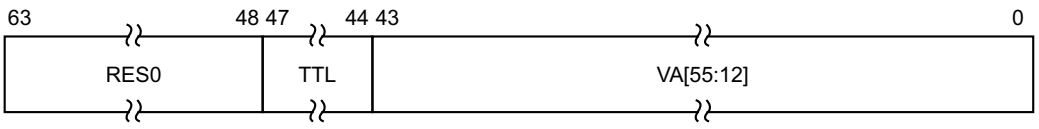
This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI VAE3OS are UNDEFINED.

Attributes

TLBI VAE3OS is a 64-bit System instruction.

Field descriptions

The TLBI VAE3OS input value bit assignments are:



Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When ARMv8.4-TTL is implemented:

Translation Table Level. Indicates the level of the page table walk that holds the leaf entry for the address being invalidated.

`0b00xx` No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

`0b01xx` The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

`0b10xx` The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b10` : Level 2.

`0b11` : Level 3.

`0b11xx` The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b10` : Level 2.

`0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing the TLBI VAE3OS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI VAE3OS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    TLBI_VAE3OS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.64 TLBI VALE1, TLB Invalidate by VA, Last level, EL1

The TLBI VALE1 characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA, and one of the following applies:
  - — The entry is a global entry from the final level of lookup.
  - — The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - — If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime.
  - — If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime.

The invalidation applies to the PE that executes this System instruction.

Configurations

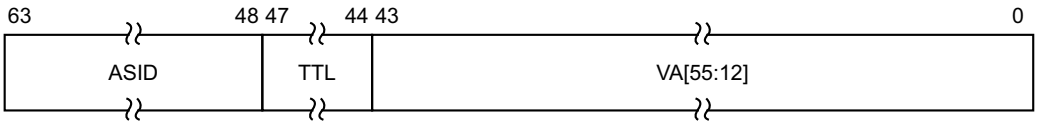
There are no configuration notes.

Attributes

TLBI VALE1 is a 64-bit System instruction.

Field descriptions

The TLBI VALE1 input value bit assignments are:



ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are RES0.

TTL, bits [47:44]

When ARMv8.4-TTL is implemented:

Translation Table Level. Indicates the level of the page table walk that holds the leaf entry for the address being invalidated.

`0b00xx` No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b01xx` The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

`0b10xx` The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b10` : Level 2.

`0b11` : Level 3.

`0b11xx` The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing the TLBI VALE1 instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI VALE1{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0111	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
HFGITR_EL2.TLBIVALE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.FB == '1' then
        TLBI_VALE1IS(X[t]);
    else
        TLBI_VAAE1(X[t]);
elsif PSTATE.EL == EL2 then
    TLBI_VAAE1(X[t]);
elsif PSTATE.EL == EL3 then
    TLBI_VAAE1(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.65 TLBI VALE1IS, TLB Invalidate by VA, Last level, EL1, Inner Shareable

The TLBI VALE1IS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA, and one of the following applies:
  - — The entry is a global entry from the final level of lookup.
  - — The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - — If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime.
  - — If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Configurations

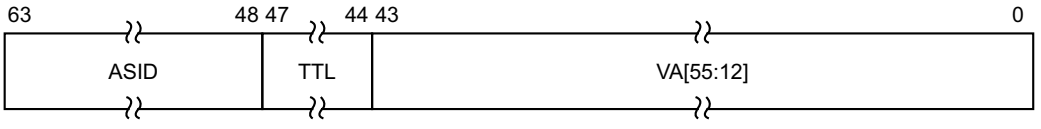
There are no configuration notes.

Attributes

TLBI VALE1IS is a 64-bit System instruction.

Field descriptions

The TLBI VALE1IS input value bit assignments are:



Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are RES0.

TTL, bits [47:44]

When ARMv8.4-TTL is implemented:

Translation Table Level. Indicates the level of the page table walk that holds the leaf entry for the address being invalidated.

`0b00xx` No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

`0b01xx` The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

`0b10xx` The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b10` : Level 2.

`0b11` : Level 3.

`0b11xx` The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Executing the TLBI VALE1IS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI VALE1IS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0011	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.TLBIVALE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        TLBI_VALE1IS(X[t]);
elsif PSTATE.EL == EL2 then
    TLBI_VALE1IS(X[t]);
elsif PSTATE.EL == EL3 then
    TLBI_VALE1IS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.66 TLBI VALE1OS, TLB Invalidate by VA, Last level, EL1, Outer Shareable

The TLBI VALE1OS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA, and one of the following applies:
  - — The entry is a global entry from the final level of lookup.
  - — The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - — If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime.
  - — If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime.
- When EL2 is not implemented or disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Configurations

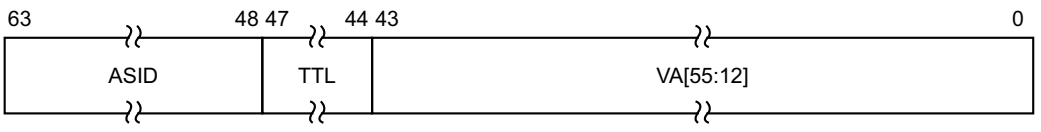
This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI VALE1OS are UNDEFINED.

Attributes

TLBI VALE1OS is a 64-bit System instruction.

Field descriptions

The TLBI VALE1OS input value bit assignments are:



Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are RES0.

TTL, bits [47:44]

When ARMv8.4-TTL is implemented:

Translation Table Level. Indicates the level of the page table walk that holds the leaf entry for the address being invalidated.

`0b00xx` No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

`0b01xx` The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

`0b10xx` The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b10` : Level 2.

`0b11` : Level 3.

`0b11xx` The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Executing the TLBI VALE1OS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI VALE1OS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0001	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.TLBIVALE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        TLBI_VALE1OS(X[t]);
elsif PSTATE.EL == EL2 then
    TLBI_VALE1OS(X[t]);
elsif PSTATE.EL == EL3 then
    TLBI_VALE1OS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.67 TLBI VALE2, TLB Invalidate by VA, Last level, EL2

The TLBI VALE2 characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA using the EL2 or EL2&0 translation regime.
- If HCR_EL2.E2H == 0, the entry is from the final level of the translation table walk.
- If HCR_EL2.E2H == 1, one of the following applies:
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of translation table walk that matches the specified ASID.

The invalidation applies to the PE that executes this System instruction.

Configurations

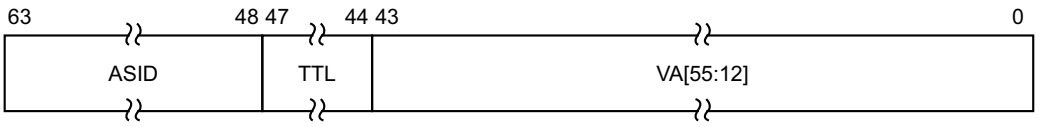
There are no configuration notes.

Attributes

TLBI VALE2 is a 64-bit System instruction.

Field descriptions

The TLBI VALE2 input value bit assignments are:



ASID, bits [63:48]

When HCR_EL2.E2H == 1:

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are RES0.

Otherwise:

Reserved, RES0.

TTL, bits [47:44]

When ARMv8.4-TTL is implemented:

Translation Table Level. Indicates the level of the page table walk that holds the leaf entry for the address being invalidated.

`0b00xx` No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

`0b01xx` The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

`0b10xx` The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b10` : Level 2.

`0b11` : Level 3.

`0b11xx` The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing the TLBI VALE2 instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI VALE2{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0111	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBI_VALE2(X[t]);
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if !EL2Enabled() then
    UNDEFINED;
else
    TLBI_VALE2(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.68 TLBI VALE2IS, TLB Invalidate by VA, Last level, EL2, Inner Shareable

The TLBI VALE2IS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA using the EL2 or EL2&0 translation regime.
- If HCR_EL2.E2H == 0, the entry is from the final level of the translation table walk.
- If HCR_EL2.E2H == 1, one of the following applies:
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of translation table walk that matches the specified ASID.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configurations

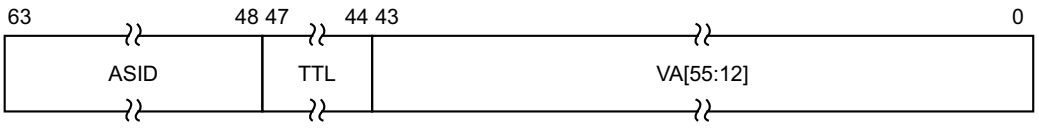
There are no configuration notes.

Attributes

TLBI VALE2IS is a 64-bit System instruction.

Field descriptions

The TLBI VALE2IS input value bit assignments are:



ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are RES0.

TTL, bits [47:44]

When ARMv8.4-TTL is implemented:

Translation Table Level. Indicates the level of the page table walk that holds the leaf entry for the address being invalidated.

`0b00xx` No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

`0b01xx` The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b10xx` The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b10` : Level 2.

`0b11` : Level 3.

`0b11xx` The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing the TLBI VALE2IS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI VALE2IS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0011	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBI_VALE2IS(X[t]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        else
                TLBI_VALE2IS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.69 TLBI VALE2OS, TLB Invalidate by VA, Last level, EL2, Outer Shareable

The TLBI VALE2OS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA using the EL2 or EL2&0 translation regime.
- If HCR_EL2.E2H == 0, the entry is from the final level of the translation table walk.
- If HCR_EL2.E2H == 1, one of the following applies:
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of translation table walk that matches the specified ASID.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Configurations

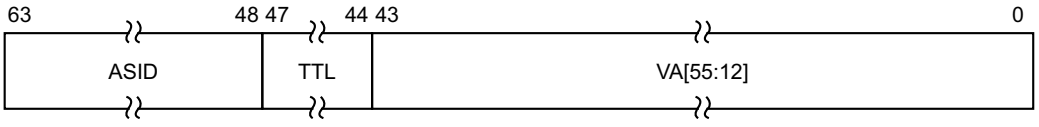
This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI VALE2OS are UNDEFINED.

Attributes

TLBI VALE2OS is a 64-bit System instruction.

Field descriptions

The TLBI VALE2OS input value bit assignments are:



ASID, bits [63:48]

When HCR_EL2.E2H == 1:

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are RES0.

Otherwise:

Reserved, RES0.

TTL, bits [47:44]

When ARMv8.4-TTL is implemented:

Translation Table Level. Indicates the level of the page table walk that holds the leaf entry for the address being invalidated.

`0b00xx` No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b01xx` The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

`0b10xx` The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b10` : Level 2.

`0b11` : Level 3.

`0b11xx` The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing the TLBI VALE2OS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI VALE2OS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0001	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBI_VALE2OS(X[t]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    else
        TLBI_VALE2OS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.70 TLBI VALE3, TLB Invalidate by VA, Last level, EL3

The TLBI VALE3 characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to the PE that executes this System instruction.

Configurations

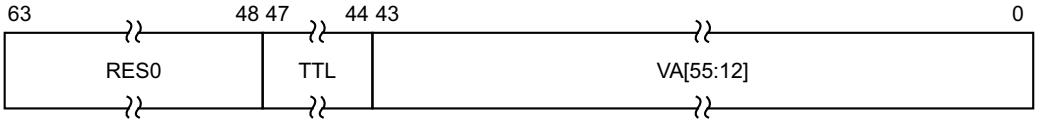
There are no configuration notes.

Attributes

TLBI VALE3 is a 64-bit System instruction.

Field descriptions

The TLBI VALE3 input value bit assignments are:



Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When ARMv8.4-TTL is implemented:

Translation Table Level. Indicates the level of the page table walk that holds the leaf entry for the address being invalidated.

`0b00xx` No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

`0b01xx` The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

`0b10xx` The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b10` : Level 2.

`0b11` : Level 3.

`0b11xx` The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing the TLBI VALE3 instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI VALE3{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0111	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    TLBI_VALE3(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.71 TLBI VALE3IS, TLB Invalidate by VA, Last level, EL3, Inner Shareable

The TLBI VALE3IS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configurations

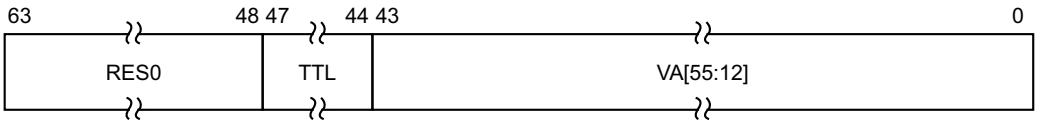
There are no configuration notes.

Attributes

TLBI VALE3IS is a 64-bit System instruction.

Field descriptions

The TLBI VALE3IS input value bit assignments are:



Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When ARMv8.4-TTL is implemented:

Translation Table Level. Indicates the level of the page table walk that holds the leaf entry for the address being invalidated.

`0b00xx` No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

`0b01xx` The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

`0b10xx` The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b10` : Level 2.

`0b11` : Level 3.

`0b11xx` The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing the TLBI VALE3IS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI VALE3IS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0011	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    TLBI_VALE3IS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.72 TLBI VALE3OS, TLB Invalidate by VA, Last level, EL3, Outer Shareable

The TLBI VALE3OS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Configurations

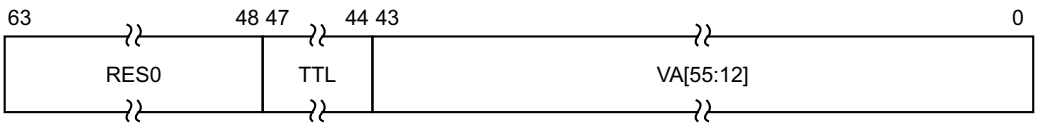
This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI VALE3OS are UNDEFINED.

Attributes

TLBI VALE3OS is a 64-bit System instruction.

Field descriptions

The TLBI VALE3OS input value bit assignments are:



Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When ARMv8.4-TTL is implemented:

Translation Table Level. Indicates the level of the page table walk that holds the leaf entry for the address being invalidated.

`0b00xx` No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

`0b01xx` The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

`0b10` : Level 2.

`0b11` : Level 3.

`0b10xx` The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b10` : Level 2.

`0b11` : Level 3.

`0b11xx` The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as:

`0b00` : Reserved. Treat as if TTL<3:2> is `0b00`.

`0b01` : Level 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b10` : Level 2.

`0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing the TLBI VALE3OS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI VALE3OS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0001	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    TLBI_VALE3OS(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.73 TLBI VMALLE1, TLB Invalidate by VMID, All at stage 1, EL1

The TLBI VMALLE1 characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime.
  - If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime.

The invalidation applies to the PE that executes this System instruction.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries, and non-global entries with any ASID.

Configurations

There are no configuration notes.

Attributes

TLBI VMALLE1 is a 64-bit System instruction.

Field descriptions

TLBI VMALLE1 ignores the value in the register specified by the instruction. Software does not have to write a value to the register before issuing this instruction.

Executing the TLBI VMALLE1 instruction

When executing this instruction Xt should be encoded as `0b11111`. If the Xt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Xt field is set to `0b11111`.

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI VMALLE1{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0111	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
HFGITR_EL2.TLBIVMALLE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.FB == '1' then
        TLBI_VMALLE1IS();
    else
        TLBI_VMALLE1();
elsif PSTATE.EL == EL2 then
    TLBI_VMALLE1();
elsif PSTATE.EL == EL3 then
    TLBI_VMALLE1();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.74 TLBI VMALLE1IS, TLB Invalidate by VMID, All at stage 1, EL1, Inner Shareable

The TLBI VMALLE1IS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime.
  - If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

For the EL1&0 translation regimes, the invalidation applies to both global entries, and non-global entries with any ASID.

Configurations

There are no configuration notes.

Attributes

TLBI VMALLE1IS is a 64-bit System instruction.

Field descriptions

TLBI VMALLE1IS ignores the value in the register specified by the instruction. Software does not have to write a value to the register before issuing this instruction.

Executing the TLBI VMALLE1IS instruction

When executing this instruction Xt should be encoded as `0b11111`. If the Xt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Xt field is set to `0b11111`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI VMALLE1IS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.TLBIVMALLE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        TLBI_VMALLE1IS();
elsif PSTATE.EL == EL2 then
    TLBI_VMALLE1IS();
elsif PSTATE.EL == EL3 then
    TLBI_VMALLE1IS();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.75 TLBI VMALLE1OS, TLB Invalidate by VMID, All at stage 1, EL1, Outer Shareable

The TLBI VMALLE1OS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime.
  - If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

For the EL1&0 translation regimes, the invalidation applies to both global entries, and non-global entries with any ASID.

Configurations

This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI VMALLE1OS are UNDEFINED.

Attributes

TLBI VMALLE1OS is a 64-bit System instruction.

Field descriptions

TLBI VMALLE1OS ignores the value in the register specified by the instruction. Software does not have to write a value to the register before issuing this instruction.

Executing the TLBI VMALLE1OS instruction

When executing this instruction Xt should be encoded as `0b11111`. If the Xt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Xt field is set to `0b11111`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI VMALLE1OS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.TLBIVMALLE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        TLBI_VMALLE1OS();
elsif PSTATE.EL == EL2 then
    TLBI_VMALLE1OS();
elsif PSTATE.EL == EL3 then
    TLBI_VMALLE1OS();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.76 TLBI VMALLS12E1, TLB Invalidate by VMID, All at Stage 1 and 2, EL1

The TLBI VMALLS12E1 characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 or stage 2 translation table entry, from any level of the translation table walk.
- If SCR_EL3.NS is 0, then
  - The entry would be required to translate an address using the Secure EL1&0 translation regime.
  - If ARMv8.4-SecEL2 is implemented and enabled, the entry would be used with the current VMID.
- If SCR_EL3.NS is 1, then:
  - The entry would be required to translate an address using the Non-secure EL1&0 translation regime.
  - If Non-secure EL2 is implemented, the entry would be used with the current VMID.

The invalidation applies to the PE that executes this System instruction.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries, and non-global entries with any ASID.

Configurations

There are no configuration notes.

Attributes

TLBI VMALLS12E1 is a 64-bit System instruction.

Field descriptions

TLBI VMALLS12E1 ignores the value in the register specified by the instruction. Software does not have to write a value to the register before issuing this instruction.

Executing the TLBI VMALLS12E1 instruction

When executing this instruction Xt should be encoded as `0b11111`. If the Xt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Xt field is set to `0b11111`.

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI VMALLS12E1{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0111	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBI_VMALLS12E1();
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        TLBI_VMALLE1();
    else
        TLBI_VMALLS12E1();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.77 TLBI VMALLS12E1IS, TLB Invalidate by VMID, All at Stage 1 and 2, EL1, Inner Shareable

The TLBI VMALLS12E1IS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 or stage 2 translation table entry, from any level of the translation table walk.
- If SCR_EL3.NS is 0, then
  - The entry would be required to translate an address using the Secure EL1&0 translation regime.
  - If ARMv8.4-SecEL2 is implemented and enabled, the entry would be used with the current VMID.
- If SCR_EL3.NS is 1, then:
  - The entry would be required to translate an address using the Non-secure EL1&0 translation regime.
  - If Non-secure EL2 is implemented, the entry would be used with the current VMID.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

For the EL1&0 translation regimes, the invalidation applies to both global entries, and non-global entries with any ASID.

Configurations

There are no configuration notes.

Attributes

TLBI VMALLS12E1IS is a 64-bit System instruction.

Field descriptions

TLBI VMALLS12E1IS ignores the value in the register specified by the instruction. Software does not have to write a value to the register before issuing this instruction.

Executing the TLBI VMALLS12E1IS instruction

When executing this instruction Xt should be encoded as `0b11111`. If the Xt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The instruction behaves as if the Xt field is set to `0b11111`.

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI VMALLS12E1IS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0011	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBI_VMALLS12E1IS();
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        TLBI_VMALLE1IS();
    else
        TLBI_VMALLS12E1IS();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.5.78 TLBI VMALLS12E1OS, TLB Invalidate by VMID, All at Stage 1 and 2, EL1, Outer Shareable

The TLBI VMALLS12E1OS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 or stage 2 translation table entry, from any level of the translation table walk.
- If SCR_EL3.NS is 0, then
  - The entry would be required to translate an address using the Secure EL1&0 translation regime.
  - If ARMv8.4-SecEL2 is implemented and enabled, the entry would be used with the current VMID.
- If SCR_EL3.NS is 1, then:
  - The entry would be required to translate an address using the Non-secure EL1&0 translation regime.
  - If Non-secure EL2 is implemented, the entry would be used with the current VMID.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

For the EL1&0 translation regimes, the invalidation applies to both global entries, and non-global entries with any ASID.

Configurations

This instruction is present only when ARMv8.4-TLBI is implemented. Otherwise, direct accesses to TLBI VMALLS12E1OS are UNDEFINED.

Attributes

TLBI VMALLS12E1OS is a 64-bit System instruction.

Field descriptions

TLBI VMALLS12E1OS ignores the value in the register specified by the instruction. Software does not have to write a value to the register before issuing this instruction.

Executing the TLBI VMALLS12E1OS instruction

When executing this instruction Xt should be encoded as `0b11111`. If the Xt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The instruction behaves as if the Xt field is set to `0b11111`.

Accesses to this register use the following encodings in the System instruction encoding space:

TLBI VMALLS12E1OS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0001	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBI_VMALLS12E1OS();
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        TLBI_VMALLE1OS();
    else
        TLBI_VMALLS12E1OS();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.6 A64 System instructions for prediction restriction

This section lists the A64 System instructions for prediction restriction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.6.1 CFP RCTX, Control Flow Prediction Restriction by Context

The CFP RCTX characteristics are:

Purpose

Control Flow Prediction Restriction by Context applies to all Control Flow Prediction Resources that predict execution based on information gathered within the target execution context or contexts.

When this instruction is complete and synchronized, control flow prediction does not permit later speculative execution within the target execution context to be observable through side channels.

This instruction is guaranteed to be complete following a DSB that covers both read and write behavior on the same PE as executed the original restriction instruction, and a subsequent context synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

Note

This instruction does not require the invalidation of prediction structures so long as the behavior described for completion of this instruction is met by the implementation.

On some implementations the instruction is likely to take a significant number of cycles to execute. This instruction is expected to be used very rarely, such as on the roll-over of an ASID or VMID, but should not be used on every context switch.

Configurations

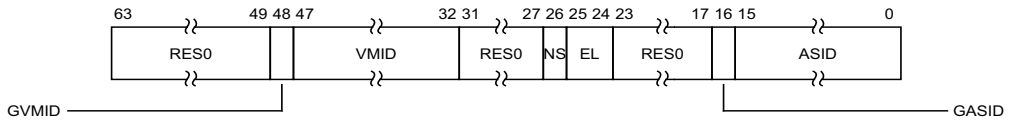
This instruction is present only when ARMv8.0-PredInv is implemented. Otherwise, direct accesses to CFP RCTX are UNDEFINED.

Attributes

CFP RCTX is a 64-bit System instruction.

Field descriptions

The CFP RCTX input value bit assignments are:



Bits [63:49]

Reserved, RES0.

GVMID, bit [48]

Execution of this instruction applies to all VMIDs or a specified VMID.

`0b0` Applies to specified VMID for an EL0 or EL1 context. For all other contexts this field is RES0.

`0b1` Applies to all VMIDs for an EL0 or EL1 context. For all other contexts this field is RES0.

If the instruction is executed at EL0 or EL1, then this field has an Effective value of 0.

VMID, bits [47:32]

Only applies when bit[48] is 0 and either:

- an EL1 context.
- an EL0 context when (HCR_EL2.E2H==0 or HCR_EL2.TGE==0).

Otherwise this field is RES0.

When the instruction is executed at EL1 then this field is treated as the current VMID.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When the instruction is executed at EL0 and (HCR_EL2.E2H==0 or HCR_EL2.TGE==0) then this field is treated as the current VMID.

When the instruction is executed at EL0 and (HCR_EL2.E2H==1 and HCR_EL2.TGE==1) then this field is ignored.

Bits [31:27]

Reserved, RES0.

NS, bit [26]

Security State

`0b0` Secure state

`0b1` Non-secure state

If the instruction is executed in Non-secure state, this field has an Effective value of 1.

EL, bits [25:24]

Exception Level

`0b00` EL0.

`0b01` EL1.

`0b10` EL2.

`0b11` EL3.

If the instruction is executed at an exception level lower than the specified level, this instruction is treated as a NOP.

Bits [23:17]

Reserved, RES0.

GASID, bit [16]

Execution of this instruction applies to all ASIDs or a specified ASID.

`0b0` Applies to specified ASID for an EL0 context. For all other contexts this field is RES0.

`0b1` Applies to all ASID for an EL0 context. For all other contexts this field is RES0.

If the instruction is executed at EL0, then this field has an Effective value of 0.

ASID, bits [15:0]

Only applies for an EL0 context and when bit[16] is 0.

Otherwise this field is RES0.

When the instruction is executed at EL0 then this field is treated as the current ASID.

Executing the CFP RCTX instruction

Accesses to this register use the following encodings in the System instruction encoding space:

CFP RCTX, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0011	0b100

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.EnRCTX == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HFGITR_EL2.CFPRCTX == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.EnRCTX == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        CFP_RCTX(X[t]);
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.CFPRCTX == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        CFP_RCTX(X[t]);
elsif PSTATE.EL == EL2 then
    CFP_RCTX(X[t]);
elsif PSTATE.EL == EL3 then
    CFP_RCTX(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.6.2 CPP RCTX, Cache Prefetch Prediction Restriction by Context

The CPP RCTX characteristics are:

Purpose

Cache Prefetch Prediction Restriction by Context applies to all Cache Allocation Resources that predict cache allocations based on information gathered within the target execution context or contexts.

When this instruction is complete and synchronized, cache prefetch prediction does not permit later speculative execution within the target execution context to be observable through side channels.

This instruction applies to all:

- Instruction caches.
- Data caches.
- TLB prefetching hardware used by the executing PE that applies to the supplied context or contexts.

This instruction is guaranteed to be complete following a DSB that covers both read and write behavior on the same PE as executed the original restriction instruction, and a subsequent context synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

Note

This instruction does not require the invalidation of Cache Allocation Resources so long as the behavior described for completion of this instruction is met by the implementation.

On some implementations the instruction is likely to take a significant number of cycles to execute. This instruction is expected to be used very rarely, such as on the roll-over of an ASID or VMID, but should not be used on every context switch.

Configurations

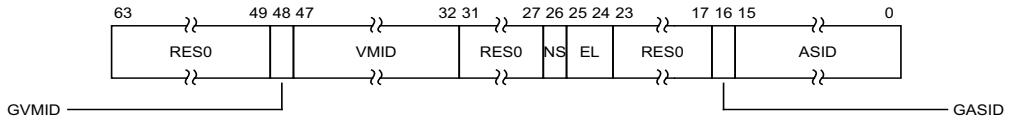
This instruction is present only when ARMv8.0-PredInv is implemented. Otherwise, direct accesses to CPP RCTX are UNDEFINED.

Attributes

CPP RCTX is a 64-bit System instruction.

Field descriptions

The CPP RCTX input value bit assignments are:



Bits [63:49]

Reserved, RES0.

GVMID, bit [48]

Execution of this instruction applies to all VMIDs or a specified VMID.

`0b0` Applies to specified VMID for an EL0 or EL1 context. For all other contexts this field is RES0.

`0b1` Applies to all VMIDs for an EL0 or EL1 context. For all other contexts this field is RES0.

If the instruction is executed at EL0 or EL1, then this field has an Effective value of 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

VMID, bits [47:32]

Only applies when bit[48] is 0 and one of:

- an EL1 context.
- an EL0 context when (HCR_EL2.E2H==0 or HCR_EL2.TGE==0).

Otherwise this field is RES0.

When the instruction is executed at EL1 then this field is treated as the current VMID.

When the instruction is executed at EL0 and (HCR_EL2.E2H==0 or HCR_EL2.TGE==0) then this field is treated as the current VMID.

When the instruction is executed at EL0 and (HCR_EL2.E2H==1 and HCR_EL2.TGE==1) then this field is ignored.

Bits [31:27]

Reserved, RES0.

NS, bit [26]

Security State.

`0b0` Secure state.

`0b1` Non-secure state.

If the instruction is executed in Non-secure state, this field has an Effective value of 1.

EL, bits [25:24]

Exception Level.

`0b00` EL0.

`0b01` EL1.

`0b10` EL2.

`0b11` EL3.

If the instruction is executed at an exception level lower than the specified level, this instruction is treated as a NOP.

Bits [23:17]

Reserved, RES0.

GASID, bit [16]

Execution of this instruction applies to all ASIDs or a specified ASID.

`0b0` Applies to specified ASID for an EL0 context. For all other contexts this field is RES0.

`0b1` Applies to all ASID for an EL0 context. For all other contexts this field is RES0.

If the instruction is executed at EL0, then this field has an Effective value of 0.

ASID, bits [15:0]

Only applies for an EL0 context and when bit[16] is 0.

Otherwise this field is RES0.

When the instruction is executed at EL0 then this field is treated as the current ASID.

Executing the CPP RCTX instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CPP RCTX, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0011	0b111

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.EnRCTX == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HFGITR_EL2.CPPRCTX == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.EnRCTX == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        CPP_RCTX(X[t]);
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.CPPRCTX == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        CPP_RCTX(X[t]);
elsif PSTATE.EL == EL2 then
    CPP_RCTX(X[t]);
elsif PSTATE.EL == EL3 then
    CPP_RCTX(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C5.6.3 DVP RCTX, Data Value Prediction Restriction by Context

The DVP RCTX characteristics are:

Purpose

Data Value Prediction Restriction by Context applies to all Data Value Prediction Resources that predict execution based on information gathered within the target execution context or contexts.

When this instruction is complete and synchronized, data value prediction does not permit later speculative execution within the target execution context to be observable through side channels.

This instruction is guaranteed to be complete following a DSB that covers both read and write behavior on the same PE as executed the original restriction instruction, and a subsequent context synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

Note

This instruction does not require the invalidation of prediction structures so long as the behavior described for completion of this instruction is met by the implementation.

On some implementations the instruction is likely to take a significant number of cycles to execute. This instruction is expected to be used very rarely, such as on the roll-over of an ASID or VMID, but should not be used on every context switch.

Configurations

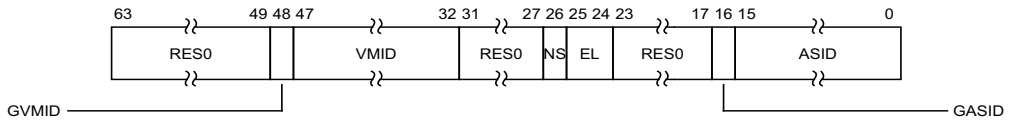
This instruction is present only when ARMv8.0-PredInv is implemented. Otherwise, direct accesses to DVP RCTX are UNDEFINED.

Attributes

DVP RCTX is a 64-bit System instruction.

Field descriptions

The DVP RCTX input value bit assignments are:



Bits [63:49]

Reserved, RES0.

GVMID, bit [48]

Execution of this instruction applies to all VMIDs or a specified VMID.

`0b0` Applies to specified VMID for an EL0 or EL1 context. For all other contexts this field is RES0.

`0b1` Applies to all VMIDs for an EL0 or EL1 context. For all other contexts this field is RES0.

If the instruction is executed at EL0 or EL1, then this field has an Effective value of 0.

VMID, bits [47:32]

Only applies when bit[48] is 0 and one of:

- an EL1 context.
- an EL0 context when (HCR_EL2.E2H==0 or HCR_EL2.TGE==0).

Otherwise this field is RES0.

When the instruction is executed at EL1 then this field is treated as the current VMID.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When the instruction is executed at EL0 and (HCR_EL2.E2H==0 or HCR_EL2.TGE==0) then this field is treated as the current VMID.

When the instruction is executed at EL0 and (HCR_EL2.E2H==1 and HCR_EL2.TGE==1) then this field is ignored.

Bits [31:27]

Reserved, RES0.

NS, bit [26]

Security State.

`0b0` Secure state.

`0b1` Non-secure state.

If the instruction is executed in Non-secure state, this field has an Effective value of 1.

EL, bits [25:24]

Exception Level

`0b00` EL0.

`0b01` EL1.

`0b10` EL2.

`0b11` EL3.

If the instruction is executed at an exception level lower than the specified level, this instruction is treated as a NOP.

Bits [23:17]

Reserved, RES0.

GASID, bit [16]

Execution of this instruction applies to all ASIDs or a specified ASID.

`0b0` Applies to specified ASID for an EL0 context. For all other contexts this field is RES0.

`0b1` Applies to all ASID for an EL0 context. For all other contexts this field is RES0.

If the instruction is executed at EL0, then this field has an Effective value of 0.

ASID, bits [15:0]

Only applies for an EL0 context and when bit[16] is 0.

Otherwise this field is RES0.

When the instruction is executed at EL0 then this field is treated as the current ASID.

Executing the DVP RCTX instruction

Accesses to this register use the following encodings in the System instruction encoding space:

DVP RCTX, <Xt>

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0011	0b101

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.EnRCTX == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HFGITR_EL2.DVPRCTX == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.EnRCTX == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DVP_RCTX(X[t]);
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.DVPRCTX == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DVP_RCTX(X[t]);
elsif PSTATE.EL == EL2 then
    DVP_RCTX(X[t]);
elsif PSTATE.EL == EL3 then
    DVP_RCTX(X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter C6
A64 Base Instruction Descriptions

This chapter describes the A64 base instructions.

It contains the following sections:

- *About the A64 base instructions* on page C6-772.
- *Alphabetical list of A64 base instructions* on page C6-775.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.1 About the A64 base instructions

Alphabetical list of A64 base instructions on page C6-775 gives full descriptions of the A64 instructions that are in the following instruction groups:

- Branch, Exception generation, and System instructions.
- Loads and stores associated with the general-purpose registers.
- Data processing (immediate).
- Data processing (register).

A64 instruction set encoding on page C4-268 provides an overview of the instruction encodings as well as of the instruction classes within their functional groups.

The rest of this section is general description of the base instructions. It contains the following subsections:

- *Register size*.
- *Use of the PC*.
- *Use of the stack pointer* on page C6-773.
- *Condition flags and related instructions* on page C6-773.

C6.1.1 Register size

Most data processing, comparison, and conversion instructions that use the general-purpose registers as the source or destination operand have two instruction variants that operate on either a 32-bit or a 64-bit value.

Where a 32-bit instruction form is selected, the following holds:

- The upper 32 bits of the source registers are ignored.
- The upper 32 bits of the destination register are set to zero.
- Right shifts and right rotates inject at bit[31], not at bit[63].
- The Condition flags, where set by the instruction, are computed from the lower 32 bits.

This distinction applies even when the results of a 32-bit instruction form are indistinguishable from the lower 32 bits computed by the equivalent 64-bit instruction form. For example, a 32-bit bitwise `ORR` could be performed using a 64-bit `ORR` and simply ignoring the top 32 bits of the result. However, the A64 instruction set includes separate 32-bit and 64-bit forms of the `ORR` instruction.

As well as distinct sign-extend or zero-extend instructions, the A64 instruction set also provides the ability to extend and shift the final source register of an `ADD`, `SUB`, `ADDS`, `or SUBS` instruction and the index register of a Load/Store instruction. This enables array index calculations involving a 64-bit array pointer and a 32-bit array index to be implemented efficiently.

The assembly language notation enables the distinct identification of registers holding 32-bit values and registers holding 64-bit values. See *Register names* on page C1-186 and *Register indexed addressing* on page C1-189.

C6.1.2 Use of the PC

A64 instructions have limited access to the PC. The only instructions that can read the PC are those that generate a PC relative address:

- ADR and ADRP.
- The Load register (literal) instruction class.
- Direct branches that use an immediate offset.
- The unconditional branch with link instructions, BL and BLR, that use the PC to create the return link address.

Only explicit control flow instructions can modify the PC:

- Conditional and unconditional branch and return instructions.
- Exception generation and exception return instructions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For more details of instructions that can modify the PC, see *Branches, Exception generating, and System instructions* on page C3-202.

C6.1.3 Use of the stack pointer

A64 instructions can use the stack pointer only in a limited number of cases:

- Load/Store instructions use the current stack pointer as the base address:
  - When stack alignment checking is enabled by system software and the base register is SP, the current stack pointer must be initially quadword aligned, That is, it must be aligned to 16 bytes. Misalignment generates an SP alignment fault. See *SP alignment checking* on page D1-2333 for more information.
- Add and subtract data processing instructions in their immediate and extended register forms, use the current stack pointer as a source register or the destination register or both.
- Logical data processing instructions in their immediate form use the current stack pointer as the destination register.

C6.1.4 Condition flags and related instructions

The A64 base instructions that use the Condition flags as an input are:

- Conditional branch. The conditional branch instruction is `B.cond`.
- Add or subtract with carry. These instruction types include instructions to perform multi-precision arithmetic and calculate checksums. The add or subtract with carry instructions are `ADC`, `ADCS`, `SBC`, and `SBCS`, or an architectural alias for these instructions.
- Conditional select with increment, negate, or invert.This instruction type conditionally selects between one source register and a second, incremented, negated, inverted, or unmodified source register. The conditional select with increment, negate, or invert instructions are `CSINC`, `CSINV`, and `CSNEG`.

  These instructions also implement:
  - Conditional select or move. The Condition flags select one of two source registers as the destination register. Short conditional sequences can be replaced by unconditional instructions followed by a conditional select, `CSEL`.
  - Conditional set. Conditionally selects between 0 and 1, or 0 and -1. This can be used to convert the Condition flags to a Boolean value or mask in a general-purpose register, for example. These instructions include `CSET` and `CSETM`.
- Conditional compare. This instruction type sets the Condition flags to the result of a comparison if the original condition is true, otherwise it sets the Condition flags to an immediate value. It permits the flattening of nested conditional expressions without using conditional branches or performing Boolean arithmetic within the general-purpose registers.The conditional compare instructions are `CCMP` and `CCMN`.

The A64 base instructions that update the Condition flags as an output are:

- Flag-setting data processing instructions, such as `ADCS`, `ADDS`, `ANDS`, `BICS`, `RMIF`, `SBCS`, `SETF8`, `SETF16`, and `SUBS`, and the aliases `CMN`, `CMP`, and `TST`.
- Conditional compare instructions such as `CCMN`, `CCMP`.
- The random number generation instructions `MRS RNDR` and `MRS RNDRRS`, see *Effect of random number generation instructions on Condition flags* on page C6-774.

The A64 base instructions that manipulate the Condition flags are:

- The flag manipulation instruction `CFINV`, which inverts the value of the Carry flag.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If ARMv8.5-CondM is implemented, the base instructions `AXFLAG` and `XAFLAG`. These instructions convert between the Arm floating point comparison PSTATE condition flag format and an alternative format shown in Table C6-1.

Table C6-1 Relationship between ARM format and alternative format PSTATE condition flags

	Arm format				Alternative format			
Result	N	Z	C	V	N	Z	C	V
Greater than	0	0	1	0	0	0	1	0
Less than	1	0	0	0	0	0	0	0
Equal	0	1	1	0	0	1	1	0
Unordered	0	0	1	1	0	1	0	0

The flags can be directly accessed for a read/write using the *NZCV, Condition Flags* on page C5-416.

The A64 base instructions also include conditional branch instructions that do not use the Condition flags as an input:

- Compare and branch if a register is zero or nonzero, `CBZ` and `CBNZ`.
- Test a single bit in a register and branch if the bit is zero or nonzero, `TBZ` and `TBNZ`.

Effect of random number generation instructions on Condition flags

If ARMv8.5-RNG is implemented, then:

- When a valid random number is returned, the PSTATE.NZCV flags are set to `0b0000`.
- If the random number hardware is not capable of returning a random number in a reasonable period of time, the PSTATE.NZCV flags are set to `0b0100`, and the random number generation instructions return the value 0.

Note

The definition of "reasonable period of time" is IMPLEMENTATION DEFINED. The expectation is that software might use this as an opportunity to reschedule or run a different routine, perhaps after a small number of retries have failed to return a valid value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2 Alphabetical list of A64 base instructions

This section lists every instruction in the base category of the A64 instruction set. For details of the format used, see *Understanding the A64 instruction descriptions* on page C2-194.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.1 ADC

Add with Carry adds two register values and the Carry flag value, and writes the result to the destination register.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
sf	0	0	1	1	0	1	0	0	0	0	Rm		0	0	0	0	0	0	Rn		Rd	
	op	S																				

32-bit variant

Applies when `sf == 0`.

```
ADC <Wd>, <Wn>, <Wm>
```

64-bit variant

Applies when `sf == 1`.

```
ADC <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
```

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Wm>`	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];

(result, -) = AddWithCarry(operand1, operand2, PSTATE.C);

X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.2 ADCS

Add with Carry, setting flags, adds two register values and the Carry flag value, and writes the result to the destination register. It updates the condition flags based on the result.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
sf	0	1	1	1	0	1	0	0	0	0	Rm		0	0	0	0	0	0	Rn		Rd	
	op	S																				

32-bit variant

Applies when `sf == 0`.

```
ADCS <Wd>, <Wn>, <Wm>
```

64-bit variant

Applies when `sf == 1`.

```
ADCS <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
```

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Wm>`	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(4) nzcv;

(result, nzcv) = AddWithCarry(operand1, operand2, PSTATE.C);

PSTATE.<N,Z,C,V> = nzcv;

X[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.3 ADD (extended register)

Add (extended register) adds a register value and a sign or zero-extended register value, followed by an optional left shift amount, and writes the result to the destination register. The argument that is extended from the `<Rm>` register can be a byte, halfword, word, or doubleword.

31	30	29	28 27 26 25 24	23 22	21	20 16	15 13	12 10	9 5	4 0
sf	0	0	0 1 0 1 1	0 0	1	Rm	option	imm3	Rn	Rd
	op	S								

32-bit variant

Applies when `sf == 0`.

```
ADD <Wd|WSP>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

64-bit variant

Applies when `sf == 1`.

```
ADD <Xd|SP>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then UNDEFINED;
```

Assembler symbols

`<Wd|WSP>` Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

`<Wn|WSP>` Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<Xd|SP>` Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

`<Xn|SP>` Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

`<R>` Is a width specifier, encoded in the "option" field. It can have the following values:

`W`	when `option = 00x`
`W`	when `option = 010`
`X`	when `option = x11`
`W`	when `option = 10x`
`W`	when `option = 110`

`<m>` Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

UXTB when option = 000

UXTH when option = 001

LSL|UXTW when option = 010

UXTX when option = 011

SXTB when option = 100

SXTH when option = 101

SXTW when option = 110

SXTX when option = 111

If "Rd" or "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

UXTB when option = 000

UXTH when option = 001

UXTW when option = 010

LSL|UXTX when option = 011

SXTB when option = 100

SXTH when option = 101

SXTW when option = 110

SXTX when option = 111

If "Rd" or "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);

(result, -) = AddWithCarry(operand1, operand2, '0');

if d == 31 then
    SP[] = result;
else
    X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.4 ADD (immediate)

Add (immediate) adds a register value and an optionally-shifted immediate value, and writes the result to the destination register.

This instruction is used by the alias MOV (to/from SP). See *Alias conditions* for details of when each alias is preferred.

31	30	29	28	27	26	25	24	23	22	21	10	9	5	4	0
sf	0	0	1	0	0	0	1	0	sh	imm12			Rn		Rd
	op	S													

32-bit variant

Applies when `sf == 0`.

```
ADD <Wd|WSP>, <Wn|WSP>, #<imm>{, <shift>}
```

64-bit variant

Applies when `sf == 1`.

```
ADD <Xd|SP>, <Xn|SP>, #<imm>{, <shift>}
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;

case sh of
    when '0' imm = ZeroExtend(imm12, datasize);
    when '1' imm = ZeroExtend(imm12:Zeros(12), datasize);
```

Alias conditions

Alias	is preferred when
MOV (to/from SP)	`sh == '0' && imm12 == '000000000000' && (Rd == '11111' || Rn == '11111')`

Assembler symbols

`<Wd|WSP>` Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

`<Wn|WSP>` Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

`<Xd|SP>` Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

`<Xn|SP>` Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

`<imm>` Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.

`<shift>` Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "sh" field. It can have the following values:

`LSL #0` when `sh = 0`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`LSL #12` when `sh = 1`

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];

(result, -) = AddWithCarry(operand1, imm, '0');

if d == 31 then
    SP[] = result;
else
    X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.5 ADD (shifted register)

Add (shifted register) adds a register value and an optionally-shifted register value, and writes the result to the destination register.

31	30	29	28 27 26 25 24	23 22	21	20 16	15 10	9 5	4 0
sf	0	0	0 1 0 1 1	shift	0	Rm	imm6	Rn	Rd
	op	S							

32-bit variant

Applies when `sf == 0`.

```
ADD <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit variant

Applies when `sf == 1`.

```
ADD <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;

if shift == '11' then UNDEFINED;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in the "shift" field. It can have the following values:

`LSL` when `shift = 00`

`LSR` when `shift = 01`

`ASR` when `shift = 10`

The encoding `shift = 11` is reserved.

`<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

(result, -) = AddWithCarry(operand1, operand2, '0');

X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.6 ADDG

Add with Tag adds an immediate value scaled by the Tag granule to the address in the source register, modifies the Logical Address Tag of the address using an immediate value, and writes the result to the destination register. Tags specified in GCR_EL1.Exclude are excluded from the possible outputs when modifying the Logical Address Tag.

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21 … 16	15	14	13 … 10	9 … 5	4 … 0
1	0	0	1	0	0	0	1	1	0	uimm6	(0)	(0)	uimm4	Xn	Xd
											op3				

Integer variant

```
ADDG <Xd|SP>, <Xn|SP>, #<uimm6>, #<uimm4>
```

Decode for this encoding

```
integer d = UInt(Xd);
integer n = UInt(Xn);
bits(64) offset = LSL(ZeroExtend(uimm6, 64), LOG2_TAG_GRANULE);
```

Assembler symbols

`<Xd|SP>` Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Xd" field.

`<Xn|SP>` Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Xn" field.

`<uimm6>` Is an unsigned immediate, a multiple of 16 in the range 0 to 1008, encoded in the "uimm6" field.

`<uimm4>` Is an unsigned immediate, in the range 0 to 15, encoded in the "uimm4" field.

Operation

```
bits(64) operand1 = if n == 31 then SP[] else X[n];
bits(4) start_tag = AArch64.AllocationTagFromAddress(operand1);
bits(16) exclude = GCR_EL1.Exclude;
bits(64) result;
bits(4) rtag;

if AArch64.AllocationTagAccessIsEnabled() then
    rtag = AArch64.ChooseNonExcludedTag(start_tag, uimm4, exclude);
else
    rtag = '0000';

(result, -) = AddWithCarry(operand1, offset, '0');
result = AArch64.AddressWithAllocationTag(result, rtag);

if d == 31 then
    SP[] = result;
else
    X[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.7 ADDS (extended register)

Add (extended register), setting flags, adds a register value and a sign or zero-extended register value, followed by an optional left shift amount, and writes the result to the destination register. The argument that is extended from the `<Rm>` register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result.

This instruction is used by the alias CMN (extended register). See Alias conditions for details of when each alias is preferred.

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15 … 13	12 … 10	9 … 5	4 … 0
sf	0	1	0	1	0	1	1	0	0	1	Rm	option	imm3	Rn	Rd
	op	S													

32-bit variant

Applies when `sf == 0`.

```
ADDS <Wd>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

64-bit variant

Applies when `sf == 1`.

```
ADDS <Xd>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then UNDEFINED;
```

Alias conditions

Alias	is preferred when
CMN (extended register)	`Rd == '11111'`

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn|WSP>` Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn|SP>` Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

`<R>` Is a width specifier, encoded in the "option" field. It can have the following values:

- `W` when `option = 00x`
- `W` when `option = 010`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

X	when option = x11
W	when option = 10x
W	when option = 110

<m> Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

UXTB	when option = 000
UXTH	when option = 001
LSL|UXTW	when option = 010
UXTX	when option = 011
SXTB	when option = 100
SXTH	when option = 101
SXTW	when option = 110
SXTX	when option = 111

If "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

UXTB	when option = 000
UXTH	when option = 001
UXTW	when option = 010
LSL|UXTX	when option = 011
SXTB	when option = 100
SXTH	when option = 101
SXTW	when option = 110
SXTX	when option = 111

If "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);
bits(4) nzcv;

(result, nzcv) = AddWithCarry(operand1, operand2, '0');

PSTATE.<N,Z,C,V> = nzcv;

X[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.8 ADDS (immediate)

Add (immediate), setting flags, adds a register value and an optionally-shifted immediate value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias CMN (immediate). See *Alias conditions* for details of when each alias is preferred.

31	30	29	28	27	26	25	24	23	22	21	10	9	5	4	0
sf	0	1	1	0	0	0	1	0	sh	imm12			Rn		Rd
	op	S													

32-bit variant

Applies when `sf == 0`.

```
ADDS <Wd>, <Wn|WSP>, #<imm>{, <shift>}
```

64-bit variant

Applies when `sf == 1`.

```
ADDS <Xd>, <Xn|SP>, #<imm>{, <shift>}
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;

case sh of
    when '0' imm = ZeroExtend(imm12, datasize);
    when '1' imm = ZeroExtend(imm12:Zeros(12), datasize);
```

Alias conditions

Alias	is preferred when
CMN (immediate)	`Rd == '11111'`

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn|WSP>` Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn|SP>` Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

`<imm>` Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.

`<shift>` Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "sh" field. It can have the following values:

`LSL #0` when `sh = 0`

`LSL #12` when `sh = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(4) nzcv;

(result, nzcv) = AddWithCarry(operand1, imm, '0');

PSTATE.<N,Z,C,V> = nzcv;

X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.9 ADDS (shifted register)

Add (shifted register), setting flags, adds a register value and an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias CMN (shifted register). See *Alias conditions* for details of when each alias is preferred.

31	30	29	28	27	26	25	24	23 22	21	20 16	15 10	9 5	4 0
sf	0	1	0	1	0	1	1	shift	0	Rm	imm6	Rn	Rd
	op	S											

32-bit variant

Applies when `sf == 0`.

```
ADDS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit variant

Applies when `sf == 1`.

```
ADDS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;

if shift == '11' then UNDEFINED;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

Alias conditions

Alias	is preferred when
CMN (shifted register)	`Rd == '11111'`

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Wm>`	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<shift>` Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in the "shift" field. It can have the following values:

`LSL` when `shift = 00`

`LSR` when `shift = 01`

`ASR` when `shift = 10`

The encoding `shift = 11` is reserved.

`<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
bits(4) nzcv;

(result, nzcv) = AddWithCarry(operand1, operand2, '0');

PSTATE.<N,Z,C,V> = nzcv;

X[d] = result;
```

Operational information

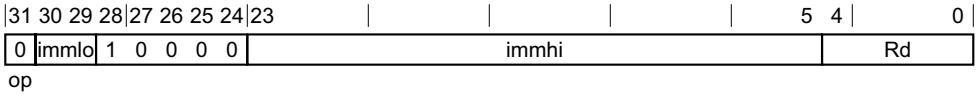
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.10 ADR

Form PC-relative address adds an immediate value to the PC value to form a PC-relative address, and writes the result to the destination register.



Literal variant

```
ADR <Xd>, <label>
```

Decode for this encoding

```
integer d = UInt(Rd);
bits(64) imm;

imm = SignExtend(immhi:immlo, 64);
```

Assembler symbols

`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<label>`	Is the program label whose address is to be calculated. Its offset from the address of this instruction, in the range +/-1MB, is encoded in "immhi:immlo".

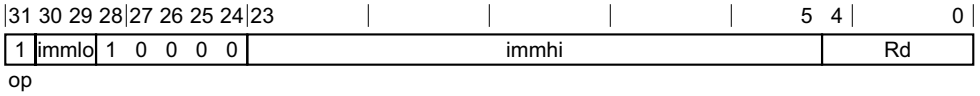
Operation

```
bits(64) base = PC[];

X[d] = base + imm;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.11 ADRP

Form PC-relative address to 4KB page adds an immediate value that is shifted left by 12 bits, to the PC value to form a PC-relative address, with the bottom 12 bits masked out, and writes the result to the destination register.



Literal variant

```
ADRP <Xd>, <label>
```

Decode for this encoding

```
integer d = UInt(Rd);
bits(64) imm;

imm = SignExtend(immhi:immlo:Zeros(12), 64);
```

Assembler symbols

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<label>` Is the program label whose 4KB page address is to be calculated. Its offset from the page address of this instruction, in the range +/-4GB, is encoded as "immhi:immlo" times 4096.

Operation

```
bits(64) base = PC[];

base<11:0> = Zeros(12);

X[d] = base + imm;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.12 AND (immediate)

Bitwise AND (immediate) performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register.

31	30 29	28 27 26 25 24	23	22 21	16 15	10 9	5 4	0
sf	0 0	1 0 0 1 0 0	N	immr	imms	Rn	Rd	
	opc							

32-bit variant

Applies when `sf == 0 && N == 0`.

```
AND <Wd|WSP>, <Wn>, #<imm>
```

64-bit variant

Applies when `sf == 1`.

```
AND <Xd|SP>, <Xn>, #<imm>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;
if sf == '0' && N != '0' then UNDEFINED;
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);
```

Assembler symbols

`<Wd|WSP>`	Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
`<Xd|SP>`	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
`<imm>`	For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr". For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];

result = operand1 AND imm;
if d == 31 then
    SP[] = result;
else
    X[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.13 AND (shifted register)

Bitwise AND (shifted register) performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register.

31	30 29	28 27 26 25	24 23 22	21	20 16	15 10	9 5	4 0
sf	0 0	0 1 0 1 0	shift	0	Rm	imm6	Rn	Rd
	opc			N				

32-bit variant

Applies when `sf == 0`.

```
AND <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit variant

Applies when `sf == 1`.

```
AND <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:

`LSL`	when `shift = 00`
`LSR`	when `shift = 01`
`ASR`	when `shift = 10`
`ROR`	when `shift = 11`

`<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

result = operand1 AND operand2;
X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.14 ANDS (immediate)

Bitwise AND (immediate), setting flags, performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias TST (immediate). See *Alias conditions* for details of when each alias is preferred.

31	30 29	28	27 26 25 24	23 22	21 … 16	15 … 10	9 … 5	4 … 0
sf	1 1 (opc)	1	0 0 1 0 0	N	immr	imms	Rn	Rd

32-bit variant

Applies when `sf == 0 && N == 0`.

```
ANDS <Wd>, <Wn>, #<imm>
```

64-bit variant

Applies when `sf == 1`.

```
ANDS <Xd>, <Xn>, #<imm>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;

bits(datasize) imm;
if sf == '0' && N != '0' then UNDEFINED;
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);
```

Alias conditions

Alias	is preferred when
TST (immediate)	`Rd == '11111'`

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
`<imm>`	For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr". For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
result = operand1 AND imm;
PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.15 ANDS (shifted register)

Bitwise AND (shifted register), setting flags, performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias TST (shifted register). See *Alias conditions* for details of when each alias is preferred.

31	30 29	28	27 26 25 24	23 22	21	20 16	15 10	9 5	4 0
sf	1 1	0	1 0 1 0	shift	0	Rm	imm6	Rn	Rd
	opc				N				

32-bit variant

Applies when `sf == 0`.

```
ANDS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit variant

Applies when `sf == 1`.

```
ANDS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;

if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

Alias conditions

Alias	is preferred when
TST (shifted register)	`Rd == '11111'`

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:

`LSL` when `shift = 00`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

LSR	when shift = 01
ASR	when shift = 10
ROR	when shift = 11

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

result = operand1 AND operand2;
PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.16 ASR (register)

Arithmetic Shift Right (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is an alias of the ASRV instruction. This means that:

- The encodings in this description are named to match the encodings of ASRV.
- The description of ASRV gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
sf	0	0	1	1	0	1	0	1	1	0	Rm	0	0	1	0	1	0	Rn	Rd
																op2			

32-bit variant

Applies when `sf == 0`.

```
ASR <Wd>, <Wn>, <Wm>
```

is equivalent to

```
ASRV <Wd>, <Wn>, <Wm>
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

```
ASR <Xd>, <Xn>, <Xm>
```

is equivalent to

```
ASRV <Xd>, <Xn>, <Xm>
```

and is always the preferred disassembly.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Wm>`	Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

The description of ASRV gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.17 ASR (immediate)

Arithmetic Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in copies of the sign bit in the upper bits and zeros in the lower bits, and writes the result to the destination register.

This instruction is an alias of the SBFM instruction. This means that:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

31	30 29	28 27 26 25 24	23	22 21 16	15	10 9 5	4 0
sf	0 0	1 0 0 1 1 0	N	immr	x 1 1 1 1 1	Rn	Rd
	opc				imms		

32-bit variant

Applies when `sf == 0 && N == 0 && imms == 011111`.

```
ASR <Wd>, <Wn>, #<shift>
```

is equivalent to

```
SBFM <Wd>, <Wn>, #<shift>, #31
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1 && N == 1 && imms == 111111`.

```
ASR <Xd>, <Xn>, #<shift>
```

is equivalent to

```
SBFM <Xd>, <Xn>, #<shift>, #63
```

and is always the preferred disassembly.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
`<shift>`	For the 32-bit variant: is the shift amount, in the range 0 to 31, encoded in the "immr" field. For the 64-bit variant: is the shift amount, in the range 0 to 63, encoded in the "immr" field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.18 ASRV

Arithmetic Shift Right Variable shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is used by the alias ASR (register). The alias is always the preferred disassembly.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
sf	0	0	1	1	0	1	0	1	1	0	Rm		0	0	1	0	1	0	Rn		Rd	
																	op2					

32-bit variant

Applies when `sf == 0`.

```
ASRV <Wd>, <Wn>, <Wm>
```

64-bit variant

Applies when `sf == 1`.

```
ASRV <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ShiftType shift_type = DecodeShift(op2);
```

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

```
bits(datasize) result;
bits(datasize) operand2 = X[m];

result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize);
X[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.19 AT

Address Translate. For more information, see *op0==0b01, cache maintenance, TLB maintenance, and address translation instructions* on page C5-380.

This instruction is an alias of the SYS instruction. This means that:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode for this instruction.

31 30 29 28	27 26 25 24	23 22	21	20	19 18 16	15 12	11 8	7 5	4 0
1 1 0 1	0 1 0 1	0 0	0	0 1	op1	0 1 1 1	1 0 0 x	op2	Rt
			L			CRn	CRm		

System variant

```
AT <at_op>, <Xt>
```

is equivalent to

```
SYS #<op1>, C7, <Cm>, #<op2>, <Xt>
```

and is the preferred disassembly when `SysOp(op1,'0111',CRm,op2) == Sys_AT`.

Assembler symbols

`<at_op>` Is an AT instruction name, as listed for the AT system instruction group, encoded in the "op1:CRm<0>:op2" field. It can have the following values:

`S1E1R`	when `op1 = 000`, `CRm<0> = 0`, `op2 = 000`
`S1E1W`	when `op1 = 000`, `CRm<0> = 0`, `op2 = 001`
`S1E0R`	when `op1 = 000`, `CRm<0> = 0`, `op2 = 010`
`S1E0W`	when `op1 = 000`, `CRm<0> = 0`, `op2 = 011`
`S1E2R`	when `op1 = 100`, `CRm<0> = 0`, `op2 = 000`
`S1E2W`	when `op1 = 100`, `CRm<0> = 0`, `op2 = 001`
`S12E1R`	when `op1 = 100`, `CRm<0> = 0`, `op2 = 100`
`S12E1W`	when `op1 = 100`, `CRm<0> = 0`, `op2 = 101`
`S12E0R`	when `op1 = 100`, `CRm<0> = 0`, `op2 = 110`
`S12E0W`	when `op1 = 100`, `CRm<0> = 0`, `op2 = 111`
`S1E3R`	when `op1 = 110`, `CRm<0> = 0`, `op2 = 000`
`S1E3W`	when `op1 = 110`, `CRm<0> = 0`, `op2 = 001`

When ARMv8.2-ATS1E1 is implemented, the following values are also valid:

`S1E1RP`	when `op1 = 000`, `CRm<0> = 1`, `op2 = 000`
`S1E1WP`	when `op1 = 000`, `CRm<0> = 1`, `op2 = 001`

`<op1>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

`<Cm>` Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

`<op2>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

`<Xt>` Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.20 AUTDA, AUTDZA

Authenticate Data address, using key A. This instruction authenticates a data address, using a modifier and key A.

The address is in the general-purpose register that is specified by `<Xd>`.

The modifier is:

- In the general-purpose register or stack pointer that is specified by `<Xn|SP>` for `AUTDA`.
- The value zero, for `AUTDZA`.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

ARMv8.3

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	0	Z	1	1	0	Rn		Rd	

AUTDA variant

Applies when `Z == 0`.

```
AUTDA <Xd>, <Xn|SP>
```

AUTDZA variant

Applies when `Z == 1 && Rn == 11111`.

```
AUTDZA <Xd>
```

Decode for all variants of this encoding

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UNDEFINED;

if Z == '0' then // AUTDA
    if n == 31 then source_is_sp = TRUE;
else // AUTDZA
    if n != 31 then UNDEFINED;
```

Assembler symbols

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

```
if HavePACExt() then
    if source_is_sp then
        X[d] = AuthDA(X[d], SP[], FALSE);
    else
        X[d] = AuthDA(X[d], X[n], FALSE);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.21 AUTDB, AUTDZB

Authenticate Data address, using key B. This instruction authenticates a data address, using a modifier and key B.

The address is in the general-purpose register that is specified by `<Xd>`.

The modifier is:

- In the general-purpose register or stack pointer that is specified by `<Xn|SP>` for `AUTDB`.
- The value zero, for `AUTDZB`.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

ARMv8.3

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	0	Z	1	1	1	Rn		Rd	

AUTDB variant

Applies when `Z == 0`.

```
AUTDB <Xd>, <Xn|SP>
```

AUTDZB variant

Applies when `Z == 1 && Rn == 11111`.

```
AUTDZB <Xd>
```

Decode for all variants of this encoding

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UNDEFINED;

if Z == '0' then // AUTDB
    if n == 31 then source_is_sp = TRUE;
else // AUTDZB
    if n != 31 then UNDEFINED;
```

Assembler symbols

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

```
if HavePACExt() then
    if source_is_sp then
        X[d] = AuthDB(X[d], SP[], FALSE);
    else
        X[d] = AuthDB(X[d], X[n], FALSE);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.22 AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA

Authenticate Instruction address, using key A. This instruction authenticates an instruction address, using a modifier and key A.

The address is:

- In the general-purpose register that is specified by `<Xd>` for `AUTIA` and `AUTIZA`.
- In X17, for `AUTIA1716`.
- In X30, for `AUTIASP` and `AUTIAZ`.

The modifier is:

- In the general-purpose register or stack pointer that is specified by `<Xn|SP>` for `AUTIA`.
- The value zero, for `AUTIZA` and `AUTIAZ`.
- In X16, for `AUTIA1716`.
- In SP, for `AUTIASP`.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

Integer

ARMv8.3

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	0	Z	1	0	0	Rn		Rd	

AUTIA variant

Applies when `Z == 0`.

```
AUTIA <Xd>, <Xn|SP>
```

AUTIZA variant

Applies when `Z == 1 && Rn == 11111`.

```
AUTIZA <Xd>
```

Decode for all variants of this encoding

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UNDEFINED;

if Z == '0' then // AUTIA
    if n == 31 then source_is_sp = TRUE;
else // AUTIZA
    if n != 31 then UNDEFINED;
```

System

ARMv8.3

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11			8	7			5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	x	1	1	0	x		1	1	1	1	1
																				CRm				op2								

AUTIA1716 variant

Applies when `CRm == 0001 && op2 == 100`.

```
AUTIA1716
```

AUTIASP variant

Applies when `CRm == 0011 && op2 == 101`.

```
AUTIASP
```

AUTIAZ variant

Applies when `CRm == 0011 && op2 == 100`.

```
AUTIAZ
```

Decode for all variants of this encoding

```
integer d;
integer n;
boolean source_is_sp = FALSE;

case CRm:op2 of
    when '0011 100'    // AUTIAZ
        d = 30;
        n = 31;
    when '0011 101'    // AUTIASP
        d = 30;
        source_is_sp = TRUE;
    when '0001 100'    // AUTIA1716
        d = 17;
        n = 16;
    when '0001 000' SEE "PACIA";
    when '0001 010' SEE "PACIB";
    when '0001 110' SEE "AUTIB";
    when '0011 00x' SEE "PACIA";
    when '0011 01x' SEE "PACIB";
    when '0011 11x' SEE "AUTIB";
    when '0000 111' SEE "XPACLRI";
    otherwise SEE "HINT";
```

Assembler symbols

`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation for all encodings

```
if HavePACExt() then
    if source_is_sp then
        X[d] = AuthIA(X[d], SP[], FALSE);
    else
        X[d] = AuthIA(X[d], X[n], FALSE);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.23 AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB

Authenticate Instruction address, using key B. This instruction authenticates an instruction address, using a modifier and key B.

The address is:

- In the general-purpose register that is specified by `<Xd>` for `AUTIB` and `AUTIZB`.
- In X17, for `AUTIB1716`.
- In X30, for `AUTIBSP` and `AUTIBZ`.

The modifier is:

- In the general-purpose register or stack pointer that is specified by `<Xn|SP>` for `AUTIB`.
- The value zero, for `AUTIZB` and `AUTIBZ`.
- In X16, for `AUTIB1716`.
- In SP, for `AUTIBSP`.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

Integer

ARMv8.3

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	0	Z	1	0	1	Rn		Rd	

AUTIB variant

Applies when `Z == 0`.

```
AUTIB <Xd>, <Xn|SP>
```

AUTIZB variant

Applies when `Z == 1 && Rn == 11111`.

```
AUTIZB <Xd>
```

Decode for all variants of this encoding

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UNDEFINED;

if Z == '0' then // AUTIB
    if n == 31 then source_is_sp = TRUE;
else // AUTIZB
    if n != 31 then UNDEFINED;
```

System

ARMv8.3

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11			8	7			5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	x	1	1	1	x		1	1	1	1	1
																				CRm				op2								

AUTIB1716 variant

Applies when `CRm == 0001 && op2 == 110`.

```
AUTIB1716
```

AUTIBSP variant

Applies when `CRm == 0011 && op2 == 111`.

```
AUTIBSP
```

AUTIBZ variant

Applies when `CRm == 0011 && op2 == 110`.

```
AUTIBZ
```

Decode for all variants of this encoding

```
integer d;
integer n;
boolean source_is_sp = FALSE;

case CRm:op2 of
    when '0011 110'    // AUTIBZ
        d = 30;
        n = 31;
    when '0011 111'    // AUTIBSP
        d = 30;
        source_is_sp = TRUE;
    when '0001 110'    // AUTIB1716
        d = 17;
        n = 16;
    when '0001 000' SEE "PACIA";
    when '0001 010' SEE "PACIB";
    when '0001 100' SEE "AUTIA";
    when '0011 00x' SEE "PACIA";
    when '0011 01x' SEE "PACIB";
    when '0011 10x' SEE "AUTIA";
    when '0000 111' SEE "XPACLRI";
    otherwise SEE "HINT";
```

Assembler symbols

`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation for all encodings

```
if HavePACExt() then
    if source_is_sp then
        X[d] = AuthIB(X[d], SP[], FALSE);
    else
        X[d] = AuthIB(X[d], X[n], FALSE);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.24 AXFLAG

Convert floating-point condition flags from Arm to external format. This instruction converts the state of the PSTATE.{N,Z,C,V} flags from a form representing the result of an Arm floating-point scalar compare instruction to an alternative representation required by some software.

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11			8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	1	0	0	(0)	(0)	(0)	(0)	0	1	0	1	1	1	1	1
																				CRm											

System variant

```
AXFLAG
```

Decode for this encoding

```
if !HaveFlagFormatExt() then UNDEFINED;
```

Operation

```
bit Z = PSTATE.Z OR PSTATE.V;
bit C = PSTATE.C AND NOT(PSTATE.V);

PSTATE.N = '0';
PSTATE.Z = Z;
PSTATE.C = C;
PSTATE.V = '0';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.25 B.cond

Branch conditionally to a label at a PC-relative offset, with a hint that this is not a subroutine call or return.

31 30 29 28	27 26 25 24	23 … 5	4	3 … 0
0 1 0 1	0 1 0 0	imm19	0	cond

19-bit signed PC-relative branch offset variant

```
B.<cond> <label>
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm19:'00', 64);
```

Assembler symbols

`<cond>` Is one of the standard conditions, encoded in the "cond" field in the standard way.

`<label>` Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

```
if ConditionHolds(cond) then
    BranchTo(PC[] + offset, BranchType_DIR);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.26 B

Branch causes an unconditional branch to a label at a PC-relative offset, with a hint that this is not a subroutine call or return.

31	30 29 28 27 26 25	0
op	0 0 1 0 1	imm26

26-bit signed PC-relative branch offset variant

```
B <label>
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm26:'00', 64);
```

Assembler symbols

`<label>` Is the program label to be unconditionally branched to. Its offset from the address of this instruction, in the range +/-128MB, is encoded as "imm26" times 4.

Operation

```
BranchTo(PC[] + offset, BranchType_DIR);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.27 BFC

Bitfield Clear sets a bitfield of `<width>` bits at bit position `<lsb>` of the destination register to zero, leaving the other destination bits unchanged.

This instruction is an alias of the BFM instruction. This means that:

- The encodings in this description are named to match the encodings of BFM.
- The description of BFM gives the operational pseudocode for this instruction.

ARMv8.2

31	30 29	28 27 26 25 24	23	22 21	16	15 10	9 5	4 0
sf	0 1	1 0 0 1 1 0	N	immr		imms	1 1 1 1 1	Rd
	opc						Rn	

32-bit variant

Applies when `sf == 0 && N == 0`.

```
BFC <Wd>, #<lsb>, #<width>
```

is equivalent to

```
BFM <Wd>, WZR, #(-<lsb> MOD 32), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

64-bit variant

Applies when `sf == 1 && N == 1`.

```
BFC <Xd>, #<lsb>, #<width>
```

is equivalent to

```
BFM <Xd>, XZR, #(-<lsb> MOD 64), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<lsb>`	For the 32-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31. For the 64-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.
`<width>`	For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>. For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of BFM gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.28 BFI

Bitfield Insert copies a bitfield of `<width>` bits from the least significant bits of the source register to bit position `<lsb>` of the destination register, leaving the other destination bits unchanged.

This instruction is an alias of the BFM instruction. This means that:

- The encodings in this description are named to match the encodings of BFM.
- The description of BFM gives the operational pseudocode for this instruction.

31	30 29	28 27 26 25 24	23	22 21	16 15	10 9	5 4	0
sf	0 1	1 0 0 1 1 0	N	immr	imms	!=11111	Rd	
	opc					Rn		

32-bit variant

Applies when `sf == 0 && N == 0`.

```
BFI <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
BFM  <Wd>, <Wn>, #(-<lsb> MOD 32), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

64-bit variant

Applies when `sf == 1 && N == 1`.

```
BFI <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
BFM  <Xd>, <Xn>, #(-<lsb> MOD 64), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
`<lsb>`	For the 32-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31. For the 64-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.
`<width>`	For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>. For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of BFM gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.29 BFM

Bitfield Move is usually accessed via one of its aliases, which are always preferred for disassembly.

If `<imms>` is greater than or equal to `<immr>`, this copies a bitfield of (`<imms>`-`<immr>`+1) bits starting from bit position `<immr>` in the source register to the least significant bits of the destination register.

If `<imms>` is less than `<immr>`, this copies a bitfield of (`<imms>`+1) bits from the least significant bits of the source register to bit position (regsize-`<immr>`) of the destination register, where regsize is the destination register size of 32 or 64 bits.

In both cases the other bits of the destination register remain unchanged.

This instruction is used by the aliases BFC, BFI, and BFXIL. See *Alias conditions* on page C6-827 for details of when each alias is preferred.

31	30 29	28 27 26 25 24	23	22 21	16 15	10 9	5 4	0
sf	0 1 (opc)	1 0 0 1 1 0	N	immr	imms	Rn		Rd

32-bit variant

Applies when `sf == 0 && N == 0`.

```
BFM <Wd>, <Wn>, #<immr>, #<imms>
```

64-bit variant

Applies when `sf == 1 && N == 1`.

```
BFM <Xd>, <Xn>, #<immr>, #<imms>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;

integer R;
bits(datasize) wmask;
bits(datasize) tmask;

if sf == '1' && N != '1' then UNDEFINED;
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then UNDEFINED;

R = UInt(immr);
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Alias conditions

Alias	is preferred when
BFC	`Rn == '11111' && UInt(imms) < UInt(immr)`
BFI	`Rn != '11111' && UInt(imms) < UInt(immr)`
BFXIL	`UInt(imms) >= UInt(immr)`

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

`<immr>` For the 32-bit variant: is the right rotate amount, in the range 0 to 31, encoded in the "immr" field.

For the 64-bit variant: is the right rotate amount, in the range 0 to 63, encoded in the "immr" field.

`<imms>` For the 32-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field.

For the 64-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the "imms" field.

Operation

```
bits(datasize) dst = X[d];
bits(datasize) src = X[n];

// perform bitfield move on low bits
bits(datasize) bot = (dst AND NOT(wmask)) OR (ROR(src, R) AND wmask);

// combine extension bits and result bits
X[d] = (dst AND NOT(tmask)) OR (bot AND tmask);
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.30 BFXIL

Bitfield Extract and Insert Low copies a bitfield of `<width>` bits starting from bit position `<lsb>` in the source register to the least significant bits of the destination register, leaving the other destination bits unchanged.

This instruction is an alias of the BFM instruction. This means that:

- The encodings in this description are named to match the encodings of BFM.
- The description of BFM gives the operational pseudocode for this instruction.

31	30 29	28	27 26 25 24	23	22 21	16	15 10	9 5	4 0
sf	0 1	1	0 0 1 1	0	N	immr	imms	Rn	Rd
	opc								

32-bit variant

Applies when `sf == 0 && N == 0`.

```
BFXIL <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
BFM  <Wd>, <Wn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `UInt(imms) >= UInt(immr)`.

64-bit variant

Applies when `sf == 1 && N == 1`.

```
BFXIL <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
BFM  <Xd>, <Xn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `UInt(imms) >= UInt(immr)`.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
`<lsb>`	For the 32-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31. For the 64-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.
`<width>`	For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>. For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of BFM gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.31 BIC (shifted register)

Bitwise Bit Clear (shifted register) performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.

31	30 29	28	27 26 25 24	23 22	21	20 16	15 10	9 5	4 0
sf	0 0	0	1 0 1 0	shift	1	Rm	imm6	Rn	Rd
	opc				N				

32-bit variant

Applies when `sf == 0`.

```
BIC <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit variant

Applies when `sf == 1`.

```
BIC <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:

`LSL`	when `shift = 00`
`LSR`	when `shift = 01`
`ASR`	when `shift = 10`
`ROR`	when `shift = 11`

`<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

operand2 = NOT(operand2);

result = operand1 AND operand2;
X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.32 BICS (shifted register)

Bitwise Bit Clear (shifted register), setting flags, performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

31	30 29	28	27 26 25 24	23 22	21	20 16	15 10	9 5	4 0
sf	1 1	0	1 0 1 0	shift	1	Rm	imm6	Rn	Rd
	opc				N				

32-bit variant

Applies when `sf == 0`.

```
BICS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit variant

Applies when `sf == 1`.

```
BICS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;

if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:

`LSL`	when `shift = 00`
`LSR`	when `shift = 01`
`ASR`	when `shift = 10`
`ROR`	when `shift = 11`

`<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

operand2 = NOT(operand2);

result = operand1 AND operand2;
PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.33 BL

Branch with Link branches to a PC-relative offset, setting the register X30 to PC+4. It provides a hint that this is a subroutine call.

31	30 29 28	27 26 25					0
1	0 0 1 0 1	imm26					
op							

26-bit signed PC-relative branch offset variant

```
BL <label>
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm26:'00', 64);
```

Assembler symbols

`<label>` Is the program label to be unconditionally branched to. Its offset from the address of this instruction, in the range +/-128MB, is encoded as "imm26" times 4.

Operation

```
X[30] = PC[] + 4;

BranchTo(PC[] + offset, BranchType_DIRCALL);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.34 BLR

Branch with Link to Register calls a subroutine at an address in a register, setting register X30 to PC+4.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
1	1	0	1	0	1	1	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	Rn		0 0 0 0 0	
							Z		op											A	M			Rm	

Integer variant

```
BLR <Xn>
```

Decode for this encoding

```
integer n = UInt(Rn);
```

Assembler symbols

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

Operation

```
bits(64) target = X[n];

X[30] = PC[] + 4;

BranchTo(target, BranchType_INDCALL);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.35 BLRAA, BLRAAZ, BLRAB, BLRABZ

Branch with Link to Register, with pointer authentication. This instruction authenticates the address in the general-purpose register that is specified by `<Xn>`, using a modifier and the specified key, and calls a subroutine at the authenticated address, setting register X30 to PC+4.

The modifier is:

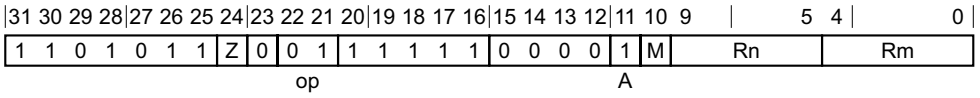
- In the general-purpose register or stack pointer that is specified by `<Xm|SP>` for `BLRAA` and `BLRAB`.
- The value zero, for `BLRAAZ` and `BLRABZ`.

Key A is used for `BLRAA` and `BLRAAZ`, and key B is used for `BLRAB` and `BLRABZ`.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to the general-purpose register.

ARMv8.3



Key A, zero modifier variant

Applies when `Z == 0 && M == 0 && Rm == 11111`.

```
BLRAAZ <Xn>
```

Key A, register modifier variant

Applies when `Z == 1 && M == 0`.

```
BLRAA <Xn>, <Xm|SP>
```

Key B, zero modifier variant

Applies when `Z == 0 && M == 1 && Rm == 11111`.

```
BLRABZ <Xn>
```

Key B, register modifier variant

Applies when `Z == 1 && M == 1`.

```
BLRAB <Xn>, <Xm|SP>
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean use_key_a = (M == '0');
boolean source_is_sp = ((Z == '1') && (m == 31));

if !HavePACExt() then
    UNDEFINED;

if Z == '0' && m != 31 then
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

<Xm|SP> Is the 64-bit name of the general-purpose source register or stack pointer holding the modifier, encoded in the "Rm" field.

Operation

```
bits(64) target = X[n];

bits(64) modifier = if source_is_sp then SP[] else X[m];

if use_key_a then
    target = AuthIA(target, modifier, TRUE);
else
    target = AuthIB(target, modifier, TRUE);

X[30] = PC[] + 4;

BranchTo(target, BranchType_INDCALL);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.36 BR

Branch to Register branches unconditionally to an address in a register, with a hint that this is not a subroutine return.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
1	1	0	1	0	1	1	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	Rn		0 0 0 0	0
							Z		op											A	M			Rm	

Integer variant

```
BR <Xn>
```

Decode for this encoding

```
integer n = UInt(Rn);
```

Assembler symbols

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

Operation

```
bits(64) target = X[n];

BranchTo(target, BranchType_INDIR);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.37 BRAA, BRAAZ, BRAB, BRABZ

Branch to Register, with pointer authentication. This instruction authenticates the address in the general-purpose register that is specified by `<Xn>`, using a modifier and the specified key, and branches to the authenticated address.

The modifier is:

- In the general-purpose register or stack pointer that is specified by `<Xm|SP>` for `BRAA` and `BRAB`.
- The value zero, for `BRAAZ` and `BRABZ`.

Key A is used for `BRAA` and `BRAAZ`, and key B is used for `BRAB` and `BRABZ`.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to the general-purpose register.

ARMv8.3

31 30 29 28	27 26 25 24	23	22 21 20	19 18 17 16	15 14 13 12	11	10	9 5	4 0
1 1 0 1	0 1 1 Z	0	0 0	1 1 1 1 1	0 0 0 0	1	M	Rn	Rm
			op			A			

Key A, zero modifier variant

Applies when `Z == 0 && M == 0 && Rm == 11111`.

```
BRAAZ <Xn>
```

Key A, register modifier variant

Applies when `Z == 1 && M == 0`.

```
BRAA <Xn>, <Xm|SP>
```

Key B, zero modifier variant

Applies when `Z == 0 && M == 1 && Rm == 11111`.

```
BRABZ <Xn>
```

Key B, register modifier variant

Applies when `Z == 1 && M == 1`.

```
BRAB <Xn>, <Xm|SP>
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean use_key_a = (M == '0');
boolean source_is_sp = ((Z == '1') && (m == 31));

if !HavePACExt() then
    UNDEFINED;

if Z == '0' && m != 31 then
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<Xn>` Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

`<Xm|SP>` Is the 64-bit name of the general-purpose source register or stack pointer holding the modifier, encoded in the "Rm" field.

Operation

```
bits(64) target = X[n];

bits(64) modifier = if source_is_sp then SP[] else X[m];

if use_key_a then
    target = AuthIA(target, modifier, TRUE);
else
    target = AuthIB(target, modifier, TRUE);

BranchTo(target, BranchType_INDIR);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.38 BRK

Breakpoint instruction. A `BRK` instruction generates a Breakpoint Instruction exception. The PE records the exception in ESR_ELx, using the EC value `0x3c`, and captures the value of the immediate argument in ESR_ELx.ISS.

31 30 29 28	27 26 25 24	23 22 21 20	5 4	3 2 1 0
1 1 0 1	0 1 0 0	0 0 1	imm16	0 0 0 0 0

System variant

```
BRK #<imm>
```

Decode for this encoding

```
if HaveBTIExt() then
    SetBTypeCompatible(TRUE);
```

Assembler symbols

`<imm>` Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

Operation

```
AArch64.SoftwareBreakpoint(imm16);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.39 BTI

Branch Target Identification. A `BTI` instruction is used to guard against the execution of instructions which are not the intended target of a branch.

Outside of a guarded memory region, a `BTI` instruction executes as a `NOP`. Within a guarded memory region while PSTATE.BTYPE != `0b00`, a `BTI` instruction compatible with the current value of PSTATE.BTYPE will not generate a Branch Target Exception and will allow execution of subsequent instructions within the memory region.

The operand `<targets>` passed to a `BTI` instruction determines the values of PSTATE.BTYPE which the `BTI` instruction is compatible with.

Note

Within a guarded memory region, while PSTATE.BTYPE != `0b00`, all instructions will generate a Branch Target Exception, other than `BRK`, `BTI`, `HLT`, `PACIASP`, and `PACIBSP`, which may not. See the individual instructions for details.

ARMv8.5

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 8	7 5	4	3 2 1 0
1 1 0 1	0 1 0 1	0 0 0 0	0 1 1 0	0 0 1 0	0 1 0 0	x x 0	1	1 1 1 1
					CRm	op2		

System variant

```
BTI {<targets>}
```

Decode for this encoding

```
SystemHintOp op;

if CRm:op2 == '0100 xx0' then
    op = SystemHintOp_BTI;
    // Check branch target compatibility between BTI instruction and PSTATE.BTYPE
    SetBTypeCompatible(BTypeCompatible_BTI(op2<2:1>));
else
    EndOfInstruction();
```

Assembler symbols

`<targets>` Is the type of indirection, encoded in the "op2<2:1>" field. It can have the following values:

`(omitted)`	when `op2<2:1> = 00`
`c`	when `op2<2:1> = 01`
`j`	when `op2<2:1> = 10`
`jc`	when `op2<2:1> = 11`

Operation

```
case op of
    when SystemHintOp_YIELD
        Hint_Yield();

    when SystemHintOp_DGH
        Hint_DGH();

    when SystemHintOp_WFE
        if IsEventRegisterSet() then
            ClearEventRegister();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        if PSTATE.EL == EL0 then
            // Check for traps described by the OS which may be EL1 or EL2.
            AArch64.CheckForWFxTrap(EL1, TRUE);
        if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
            // Check for traps described by the Hypervisor.
            AArch64.CheckForWFxTrap(EL2, TRUE);
        if HaveEL(EL3) && PSTATE.EL != EL3 then
            // Check for traps described by the Secure Monitor.
            AArch64.CheckForWFxTrap(EL3, TRUE);
        WaitForEvent();

when SystemHintOp_WFI
    if !InterruptPending() then
        if PSTATE.EL == EL0 then
            // Check for traps described by the OS which may be EL1 or EL2.
            AArch64.CheckForWFxTrap(EL1, FALSE);
        if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
            // Check for traps described by the Hypervisor.
            AArch64.CheckForWFxTrap(EL2, FALSE);
        if HaveEL(EL3) && PSTATE.EL != EL3 then
            // Check for traps described by the Secure Monitor.
            AArch64.CheckForWFxTrap(EL3, FALSE);
        WaitForInterrupt();

when SystemHintOp_SEV
    SendEvent();

when SystemHintOp_SEVL
    SendEventLocal();

when SystemHintOp_ESB
    SynchronizeErrors();
    AArch64.ESBOperation();
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch64.vESBOperation();
    TakeUnmaskedSErrorInterrupts();

when SystemHintOp_PSB
    ProfilingSynchronizationBarrier();

when SystemHintOp_TSB
    TraceSynchronizationBarrier();

when SystemHintOp_CSDB
    ConsumptionOfSpeculativeDataBarrier();

when SystemHintOp_BTI
    SetBTypeNext('00');

otherwise    // do nothing
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.40 CASB, CASAB, CASALB, CASLB

Compare and Swap byte in memory reads an 8-bit byte from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- `CASAB` and `CASALB` load from memory with acquire semantics.
- `CASLB` and `CASALB` store to memory with release semantics.
- `CASB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is `<Ws>`, is restored to the values held in the register before the instruction was executed.

ARMv8.1

31 30	29 28	27 26 25 24	23	22	21	20 16	15	14 13 12	11 10	9 5	4 0
0 0	0 0	1 0 0 0	1	L	1	Rs	o0	1 1 1	1 1	Rn	Rt
size											

CASAB variant

Applies when `L == 1 && o0 == 0`.

```
CASAB <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

CASALB variant

Applies when `L == 1 && o0 == 1`.

```
CASALB <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

CASB variant

Applies when `L == 0 && o0 == 0`.

```
CASB <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

CASLB variant

Applies when `L == 0 && o0 == 1`.

```
CASLB <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
AccType ldacctype = if L == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.
`<Wt>`	Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) comparevalue;
bits(8) newvalue;
bits(8) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

comparevalue = X[s];
newvalue = X[t];

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomicCompareAndSwap(address, comparevalue, newvalue, ldacctype, stacctype);

X[s] = ZeroExtend(data, 32);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.41 CASH, CASAH, CASALH, CASLH

Compare and Swap halfword in memory reads a 16-bit halfword from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- `CASAH` and `CASALH` load from memory with acquire semantics.
- `CASLH` and `CASALH` store to memory with release semantics.
- `CAS` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is `<Ws>`, is restored to the values held in the register before the instruction was executed.

ARMv8.1

31 30	29 28	27 26 25 24	23	22	21	20 16	15	14 13 12	11 10	9 5	4 0
size	0 0	1 0 0 0	1	L	1	Rs	o0	1 1 1	1 1	Rn	Rt

CASAH variant

Applies when `L == 1 && o0 == 0`.

```
CASAH <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

CASALH variant

Applies when `L == 1 && o0 == 1`.

```
CASALH <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

CASH variant

Applies when `L == 0 && o0 == 0`.

```
CASH <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

CASLH variant

Applies when `L == 0 && o0 == 1`.

```
CASLH <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
AccType ldacctype = if L == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.
`<Wt>`	Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) comparevalue;
bits(16) newvalue;
bits(16) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

comparevalue = X[s];
newvalue = X[t];

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomicCompareAndSwap(address, comparevalue, newvalue, ldacctype, stacctype);

X[s] = ZeroExtend(data, 32);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.42 CASP, CASPA, CASPAL, CASPL

Compare and Swap Pair of words or doublewords in memory reads a pair of 32-bit words or 64-bit doublewords from memory, and compares them against the values held in the first pair of registers. If the comparison is equal, the values in the second pair of registers are written to memory. If the writes are performed, the reads and writes occur atomically such that no other modification of the memory location can take place between the reads and writes.

- `CASPA` and `CASPAL` load from memory with acquire semantics.
- `CASPL` and `CASPAL` store to memory with release semantics.
- `CAS` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the registers which are compared and loaded, that is `<Ws>` and `<W(s+1)>`, or `<Xs>` and `<X(s+1)>`, are restored to the values held in the registers before the instruction was executed.

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14 … 10	9 … 5	4 … 0
0	sz	0	0	1	0	0	0	0	L	1	Rs	o0	1 1 1 1 1 (Rt2)	Rn	Rt

32-bit CASP variant

Applies when `sz == 0 && L == 0 && o0 == 0`.

```
CASP <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{,#0}]
```

32-bit CASPA variant

Applies when `sz == 0 && L == 1 && o0 == 0`.

```
CASPA <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{,#0}]
```

32-bit CASPAL variant

Applies when `sz == 0 && L == 1 && o0 == 1`.

```
CASPAL <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{,#0}]
```

32-bit CASPL variant

Applies when `sz == 0 && L == 0 && o0 == 1`.

```
CASPL <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{,#0}]
```

64-bit CASP variant

Applies when `sz == 1 && L == 0 && o0 == 0`.

```
CASP <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

64-bit CASPA variant

Applies when `sz == 1 && L == 1 && o0 == 0`.

```
CASPA <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}]
```

64-bit CASPAL variant

Applies when `sz == 1 && L == 1 && o0 == 1`.

```
CASPAL <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}]
```

64-bit CASPL variant

Applies when `sz == 1 && L == 0 && o0 == 1`.

```
CASPL <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UNDEFINED;
if Rs<0> == '1' then UNDEFINED;
if Rt<0> == '1' then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);

integer datasize = 32 << UInt(sz);
AccType ldacctype = if L == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler symbols

`<Ws>`	Is the 32-bit name of the first general-purpose register to be compared and loaded, encoded in the "Rs" field. <Ws> must be an even-numbered register.
`<W(s+1)>`	Is the 32-bit name of the second general-purpose register to be compared and loaded.
`<Wt>`	Is the 32-bit name of the first general-purpose register to be conditionally stored, encoded in the "Rt" field. <Wt> must be an even-numbered register.
`<W(t+1)>`	Is the 32-bit name of the second general-purpose register to be conditionally stored.
`<Xs>`	Is the 64-bit name of the first general-purpose register to be compared and loaded, encoded in the "Rs" field. <Xs> must be an even-numbered register.
`<X(s+1)>`	Is the 64-bit name of the second general-purpose register to be compared and loaded.
`<Xt>`	Is the 64-bit name of the first general-purpose register to be conditionally stored, encoded in the "Rt" field. <Xt> must be an even-numbered register.
`<X(t+1)>`	Is the 64-bit name of the second general-purpose register to be conditionally stored.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(2*datasize) comparevalue;
bits(2*datasize) newvalue;
bits(2*datasize) data;

bits(datasize) s1 = X[s];
bits(datasize) s2 = X[s+1];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(datasize) t1 = X[t];
bits(datasize) t2 = X[t+1];
comparevalue = if BigEndian() then s1:s2 else s2:s1;
newvalue = if BigEndian() then t1:t2 else t2:t1;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomicCompareAndSwap(address, comparevalue, newvalue, ldacctype, stacctype);

if BigEndian() then
    X[s] = ZeroExtend(data<2*datasize-1:datasize>, datasize);
    X[s+1] = ZeroExtend(data<datasize-1:0>, datasize);
else
    X[s] = ZeroExtend(data<datasize-1:0>, datasize);
    X[s+1] = ZeroExtend(data<2*datasize-1:datasize>, datasize);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.43 CAS, CASA, CASAL, CASL

Compare and Swap word or doubleword in memory reads a 32-bit word or 64-bit doubleword from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- `CASA` and `CASAL` load from memory with acquire semantics.
- `CASL` and `CASAL` store to memory with release semantics.
- `CAS` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is `<Ws>`, or `<Xs>`, is restored to the value held in the register before the instruction was executed.

ARMv8.1

31 30	29 28 27 26 25 24	23	22	21	20 16	15	14 13 12 11 10	9 5	4 0
1 x	0 0 1 0 0 0	1	L	1	Rs	o0	1 1 1 1 1	Rn	Rt
size									

32-bit CAS variant

Applies when `size == 10 && L == 0 && o0 == 0`.

```
CAS <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

32-bit CASA variant

Applies when `size == 10 && L == 1 && o0 == 0`.

```
CASA <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

32-bit CASAL variant

Applies when `size == 10 && L == 1 && o0 == 1`.

```
CASAL <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

32-bit CASL variant

Applies when `size == 10 && L == 0 && o0 == 1`.

```
CASL <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

64-bit CAS variant

Applies when `size == 11 && L == 0 && o0 == 0`.

```
CAS <Xs>, <Xt>, [<Xn|SP>{,#0}]
```

64-bit CASA variant

Applies when `size == 11 && L == 1 && o0 == 0`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
CASA <Xs>, <Xt>, [<Xn|SP>{,#0}]
```

64-bit CASAL variant

Applies when `size == 11 && L == 1 && o0 == 1`.

```
CASAL <Xs>, <Xt>, [<Xn|SP>{,#0}]
```

64-bit CASL variant

Applies when `size == 11 && L == 0 && o0 == 1`.

```
CASL <Xs>, <Xt>, [<Xn|SP>{,#0}]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if L == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.
`<Wt>`	Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.
`<Xs>`	Is the 64-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) comparevalue;
bits(datasize) newvalue;
bits(datasize) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

comparevalue = X[s];
newvalue = X[t];

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomicCompareAndSwap(address, comparevalue, newvalue, ldacctype, stacctype);

X[s] = ZeroExtend(data, regsize);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.44 CBNZ

Compare and Branch on Nonzero compares the value in a register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is not equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

31	30	29	28	27	26	25	24	23 … 5	4 … 0
sf	0	1	1	0	1	0	1 (op)	imm19	Rt

32-bit variant

Applies when `sf == 0`.

```
CBNZ <Wt>, <label>
```

64-bit variant

Applies when `sf == 1`.

```
CBNZ <Xt>, <label>
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer datasize = if sf == '1' then 64 else 32;
bits(64) offset = SignExtend(imm19:'00', 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
`<label>`	Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

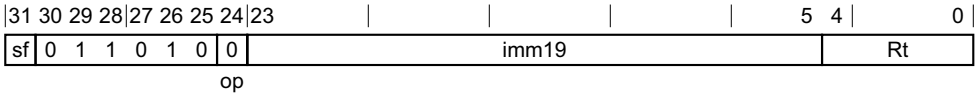
Operation

```
bits(datasize) operand1 = X[t];

if IsZero(operand1) == FALSE then
    BranchTo(PC[] + offset, BranchType_DIR);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.45 CBZ

Compare and Branch on Zero compares the value in a register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.



32-bit variant

Applies when `sf == 0`.

```
CBZ <Wt>, <label>
```

64-bit variant

Applies when `sf == 1`.

```
CBZ <Xt>, <label>
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer datasize = if sf == '1' then 64 else 32;
bits(64) offset = SignExtend(imm19:'00', 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
`<label>`	Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

```
bits(datasize) operand1 = X[t];

if IsZero(operand1) == TRUE then
    BranchTo(PC[] + offset, BranchType_DIR);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.46 CCMN (immediate)

Conditional Compare Negative (immediate) sets the value of the condition flags to the result of the comparison of a register value and a negated immediate value if the condition is TRUE, and an immediate value otherwise.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	12	11	10	9	5	4	3	0
sf	0	1	1	1	0	1	0	0	1	0	imm5		cond		1	0	Rn		0	nzcv	
	op																				

32-bit variant

Applies when `sf == 0`.

```
CCMN <Wn>, #<imm>, #<nzcv>, <cond>
```

64-bit variant

Applies when `sf == 1`.

```
CCMN <Xn>, #<imm>, #<nzcv>, <cond>
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(4) flags = nzcv;
bits(datasize) imm = ZeroExtend(imm5, datasize);
```

Assembler symbols

`<Wn>`	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Xn>`	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<imm>`	Is a five bit unsigned (positive) immediate encoded in the "imm5" field.
`<nzcv>`	Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.
`<cond>`	Is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

```
bits(datasize) operand1 = X[n];

if ConditionHolds(cond) then
    (-, flags) = AddWithCarry(operand1, imm, '0');
PSTATE.<N,Z,C,V> = flags;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.47 CCMN (register)

Conditional Compare Negative (register) sets the value of the condition flags to the result of the comparison of a register value and the inverse of another register value if the condition is TRUE, and an immediate value otherwise.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	12	11	10	9	5	4	3	0
sf	0	1	1	1	0	1	0	0	1	0	Rm		cond		0	0	Rn		0	nzcv	
	op																				

32-bit variant

Applies when `sf == 0`.

```
CCMN <Wn>, <Wm>, #<nzcv>, <cond>
```

64-bit variant

Applies when `sf == 1`.

```
CCMN <Xn>, <Xm>, #<nzcv>, <cond>
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
bits(4) flags = nzcv;
```

Assembler symbols

`<Wn>`	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Wm>`	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
`<Xn>`	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
`<nzcv>`	Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.
`<cond>`	Is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];

if ConditionHolds(cond) then
    (-, flags) = AddWithCarry(operand1, operand2, '0');
PSTATE.<N,Z,C,V> = flags;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.48 CCMP (immediate)

Conditional Compare (immediate) sets the value of the condition flags to the result of the comparison of a register value and an immediate value if the condition is TRUE, and an immediate value otherwise.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	12	11	10	9	5	4	3	0
sf	1	1	1	1	0	1	0	0	1	0	imm5		cond		1	0	Rn		0	nzcv	
	op																				

32-bit variant

Applies when `sf == 0`.

```
CCMP <Wn>, #<imm>, #<nzcv>, <cond>
```

64-bit variant

Applies when `sf == 1`.

```
CCMP <Xn>, #<imm>, #<nzcv>, <cond>
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(4) flags = nzcv;
bits(datasize) imm = ZeroExtend(imm5, datasize);
```

Assembler symbols

`<Wn>`	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Xn>`	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<imm>`	Is a five bit unsigned (positive) immediate encoded in the "imm5" field.
`<nzcv>`	Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.
`<cond>`	Is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2;

if ConditionHolds(cond) then
    operand2 = NOT(imm);
    (-, flags) = AddWithCarry(operand1, operand2, '1');
PSTATE.<N,Z,C,V> = flags;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.49 CCMP (register)

Conditional Compare (register) sets the value of the condition flags to the result of the comparison of two registers if the condition is TRUE, and an immediate value otherwise.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	12	11	10	9	5	4	3	0
sf	1	1	1	1	0	1	0	0	1	0	Rm		cond		0	0	Rn		0	nzcv	
	op																				

32-bit variant

Applies when `sf == 0`.

```
CCMP <Wn>, <Wm>, #<nzcv>, <cond>
```

64-bit variant

Applies when `sf == 1`.

```
CCMP <Xn>, <Xm>, #<nzcv>, <cond>
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
bits(4) flags = nzcv;
```

Assembler symbols

`<Wn>`	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Wm>`	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
`<Xn>`	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
`<nzcv>`	Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.
`<cond>`	Is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];

if ConditionHolds(cond) then
    operand2 = NOT(operand2);
    (-, flags) = AddWithCarry(operand1, operand2, '1');
PSTATE.<N,Z,C,V> = flags;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.50 CFINV

Invert Carry Flag. This instruction inverts the value of the PSTATE.C flag.

ARMv8.4

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11			8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	1	0	0	(0)	(0)	(0)	(0)	0	0	0	1	1	1	1	1
																				CRm											

System variant

```
CFINV
```

Decode for this encoding

```
if !HaveFlagManipulateExt() then UNDEFINED;
```

Operation

```
PSTATE.C = NOT(PSTATE.C);
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.51 CFP

Control Flow Prediction Restriction by Context prevents control flow predictions that predict execution addresses, based on information gathered from earlier execution within a particular execution context, from allowing later speculative execution within that context to be observable through side-channels.

For more information, see CFP RCTX.

This instruction is an alias of the SYS instruction. This means that:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode for this instruction.

ARMv8.5

31 30 29 28	27 26 25 24	23 22	21	20 19	18 16	15 12	11 8	7 5	4 0
1 1 0 1	0 1 0 1	0 0	0	0 1	0 1 1	0 1 1 1	0 0 1 1	1 0 0	Rt
			L		op1	CRn	CRm	op2	

System variant

```
CFP RCTX, <Xt>
```

is equivalent to

```
SYS #3, C8, C3, #4, <Xt>
```

and is always the preferred disassembly.

Assembler symbols

`<Xt>` Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.52 CINC

Conditional Increment returns, in the destination register, the value of the source register incremented by 1 if the condition is TRUE, and otherwise returns the value of the source register.

This instruction is an alias of the CSINC instruction. This means that:

- The encodings in this description are named to match the encodings of CSINC.
- The description of CSINC gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	12	11	10	9	5	4	0
sf	0	0	1	1	0	1	0	1	0	0	!=11111		!=111x		0	1	!=11111		Rd	
op											Rm		cond		o2		Rn			

32-bit variant

Applies when `sf == 0`.

```
CINC <Wd>, <Wn>, <cond>
```

is equivalent to

```
CSINC <Wd>, <Wn>, <Wn>, invert(<cond>)
```

and is the preferred disassembly when `Rn == Rm`.

64-bit variant

Applies when `sf == 1`.

```
CINC <Xd>, <Xn>, <cond>
```

is equivalent to

```
CSINC <Xd>, <Xn>, <Xn>, invert(<cond>)
```

and is the preferred disassembly when `Rn == Rm`.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
`<cond>`	Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

The description of CSINC gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.53 CINV

Conditional Invert returns, in the destination register, the bitwise inversion of the value of the source register if the condition is TRUE, and otherwise returns the value of the source register.

This instruction is an alias of the CSINV instruction. This means that:

- The encodings in this description are named to match the encodings of CSINV.
- The description of CSINV gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	12	11	10	9	5	4	0
sf	1	0	1	1	0	1	0	1	0	0	!=11111		!=111x		0	0	!=11111		Rd	
op											Rm		cond		o2		Rn			

32-bit variant

Applies when `sf == 0`.

```
CINV <Wd>, <Wn>, <cond>
```

is equivalent to

```
CSINV <Wd>, <Wn>, <Wn>, invert(<cond>)
```

and is the preferred disassembly when `Rn == Rm`.

64-bit variant

Applies when `sf == 1`.

```
CINV <Xd>, <Xn>, <cond>
```

is equivalent to

```
CSINV <Xd>, <Xn>, <Xn>, invert(<cond>)
```

and is the preferred disassembly when `Rn == Rm`.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
`<cond>`	Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

The description of CSINV gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.54 CLREX

Clear Exclusive clears the local monitor of the executing PE.

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 8	7 6 5 4	3 2 1 0
1 1 0 1	0 1 0 1	0 0 0 0	0 1 1 1	0 0 1 1	CRm	0 1 0 1	1 1 1 1

System variant

```
CLREX {#<imm>}
```

Decode for this encoding

```
// CRm field is ignored
```

Assembler symbols

`<imm>` Is an optional 4-bit unsigned immediate, in the range 0 to 15, defaulting to 15 and encoded in the "CRm" field.

Operation

```
ClearExclusiveLocal(ProcessorID());
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.55 CLS

Count Leading Sign bits counts the number of leading bits of the source register that have the same value as the most significant bit of the register, and writes the result to the destination register. This count does not include the most significant bit of the source register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
sf	1	0	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	1	0	1 (op)	Rn	Rd

32-bit variant

Applies when `sf == 0`.

```
CLS <Wd>, <Wn>
```

64-bit variant

Applies when `sf == 1`.

```
CLS <Xd>, <Xn>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
```

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
integer result;
bits(datasize) operand1 = X[n];

result = CountLeadingSignBits(operand1);

X[d] = result<datasize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.56 CLZ

Count Leading Zeros counts the number of binary zero bits before the first binary one bit in the value of the source register, and writes the result to the destination register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
sf	1	0	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	Rn		Rd	
																					op				

32-bit variant

Applies when `sf == 0`.

```
CLZ <Wd>, <Wn>
```

64-bit variant

Applies when `sf == 1`.

```
CLZ <Xd>, <Xn>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
```

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
integer result;
bits(datasize) operand1 = X[n];

result = CountLeadingZeroBits(operand1);
X[d] = result<datasize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.57 CMN (extended register)

Compare Negative (extended register) adds a register value and a sign or zero-extended register value, followed by an optional left shift amount. The argument that is extended from the `<Rm>` register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result, and discards the result.

This instruction is an alias of the ADDS (extended register) instruction. This means that:

- The encodings in this description are named to match the encodings of ADDS (extended register).
- The description of ADDS (extended register) gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15 … 13	12 … 10	9 … 5	4 … 0
sf	0	1	0	1	0	1	1	0	0	1	Rm	option	imm3	Rn	1 1 1 1 1
	op	S													Rd

32-bit variant

Applies when `sf == 0`.

```
CMN <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

is equivalent to

```
ADDS WZR, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

```
CMN <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

is equivalent to

```
ADDS XZR, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

and is always the preferred disassembly.

Assembler symbols

`<Wn|WSP>` Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<Xn|SP>` Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

`<R>` Is a width specifier, encoded in the "option" field. It can have the following values:

`W`	when `option = 00x`
`W`	when `option = 010`
`X`	when `option = x11`
`W`	when `option = 10x`
`W`	when `option = 110`

`<m>` Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<extend>` For the 32-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

`UXTB`	when `option = 000`
`UXTH`	when `option = 001`
`LSL|UXTW`	when `option = 010`
`UXTX`	when `option = 011`
`SXTB`	when `option = 100`
`SXTH`	when `option = 101`
`SXTW`	when `option = 110`
`SXTX`	when `option = 111`

If "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

`UXTB`	when `option = 000`
`UXTH`	when `option = 001`
`UXTW`	when `option = 010`
`LSL|UXTX`	when `option = 011`
`SXTB`	when `option = 100`
`SXTH`	when `option = 101`
`SXTW`	when `option = 110`
`SXTX`	when `option = 111`

If "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

`<amount>` Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

Operation

The description of ADDS (extended register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.58 CMN (immediate)

Compare Negative (immediate) adds a register value and an optionally-shifted immediate value. It updates the condition flags based on the result, and discards the result.

This instruction is an alias of the ADDS (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of ADDS (immediate).
- The description of ADDS (immediate) gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	10	9	5	4	0
sf	0	1	1	0	0	0	1	0	sh	imm12			Rn	1 1 1 1 1	
	op	S												Rd	

32-bit variant

Applies when `sf == 0`.

```
CMN <Wn|WSP>, #<imm>{, <shift>}
```

is equivalent to

```
ADDS WZR, <Wn|WSP>, #<imm> {, <shift>}
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

```
CMN <Xn|SP>, #<imm>{, <shift>}
```

is equivalent to

```
ADDS XZR, <Xn|SP>, #<imm> {, <shift>}
```

and is always the preferred disassembly.

Assembler symbols

`<Wn|WSP>` Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

`<Xn|SP>` Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

`<imm>` Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.

`<shift>` Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "sh" field. It can have the following values:

`LSL #0` when `sh = 0`

`LSL #12` when `sh = 1`

Operation

The description of ADDS (immediate) gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.59 CMN (shifted register)

Compare Negative (shifted register) adds a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

This instruction is an alias of the ADDS (shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of ADDS (shifted register).
- The description of ADDS (shifted register) gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23 22	21	20 16	15 10	9 5	4 0
sf	0	1	0	1	0	1	1	shift	0	Rm	imm6	Rn	1 1 1 1 1
	op	S											Rd

32-bit variant

Applies when `sf == 0`.

```
CMN <Wn>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
ADDS WZR, <Wn>, <Wm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

```
CMN <Xn>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
ADDS XZR, <Xn>, <Xm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler symbols

`<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in the "shift" field. It can have the following values:

`LSL`	when `shift = 00`
`LSR`	when `shift = 01`
`ASR`	when `shift = 10`

The encoding `shift = 11` is reserved.

`<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

The description of ADDS (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.60 CMP (extended register)

Compare (extended register) subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value. The argument that is extended from the `<Rm>` register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result, and discards the result.

This instruction is an alias of the SUBS (extended register) instruction. This means that:

- The encodings in this description are named to match the encodings of SUBS (extended register).
- The description of SUBS (extended register) gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15 … 13	12 … 10	9 … 5	4 … 0
sf	1	1	0	1	0	1	1	0	0	1	Rm	option	imm3	Rn	1 1 1 1 1
	op	S													Rd

32-bit variant

Applies when `sf == 0`.

```
CMP <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

is equivalent to

```
SUBS WZR, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

```
CMP <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

is equivalent to

```
SUBS XZR, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

and is always the preferred disassembly.

Assembler symbols

`<Wn|WSP>` Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<Xn|SP>` Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

`<R>` Is a width specifier, encoded in the "option" field. It can have the following values:

`W`	when `option = 00x`
`W`	when `option = 010`
`X`	when `option = x11`
`W`	when `option = 10x`
`W`	when `option = 110`

`<m>` Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<extend>` For the 32-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

`UXTB`	when `option = 000`
`UXTH`	when `option = 001`
`LSL|UXTW`	when `option = 010`
`UXTX`	when `option = 011`
`SXTB`	when `option = 100`
`SXTH`	when `option = 101`
`SXTW`	when `option = 110`
`SXTX`	when `option = 111`

If "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

`UXTB`	when `option = 000`
`UXTH`	when `option = 001`
`UXTW`	when `option = 010`
`LSL|UXTX`	when `option = 011`
`SXTB`	when `option = 100`
`SXTH`	when `option = 101`
`SXTW`	when `option = 110`
`SXTX`	when `option = 111`

If "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

`<amount>` Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

Operation

The description of SUBS (extended register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.61 CMP (immediate)

Compare (immediate) subtracts an optionally-shifted immediate value from a register value. It updates the condition flags based on the result, and discards the result.

This instruction is an alias of the SUBS (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of SUBS (immediate).
- The description of SUBS (immediate) gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	10	9	5	4	0
sf	1	1	1	0	0	0	1	0	sh	imm12			Rn	1 1 1 1 1	
	op	S												Rd	

32-bit variant

Applies when `sf == 0`.

```
CMP <Wn|WSP>, #<imm>{, <shift>}
```

is equivalent to

```
SUBS WZR, <Wn|WSP>, #<imm> {, <shift>}
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

```
CMP <Xn|SP>, #<imm>{, <shift>}
```

is equivalent to

```
SUBS XZR, <Xn|SP>, #<imm> {, <shift>}
```

and is always the preferred disassembly.

Assembler symbols

`<Wn|WSP>` Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

`<Xn|SP>` Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

`<imm>` Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.

`<shift>` Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "sh" field. It can have the following values:

`LSL #0` when `sh = 0`

`LSL #12` when `sh = 1`

Operation

The description of SUBS (immediate) gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.62 CMP (shifted register)

Compare (shifted register) subtracts an optionally-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.

This instruction is an alias of the SUBS (shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of SUBS (shifted register).
- The description of SUBS (shifted register) gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23 22	21	20 16	15 10	9 5	4 0
sf	1	1	0	1	0	1	1	shift	0	Rm	imm6	Rn	1 1 1 1 1
	op	S											Rd

32-bit variant

Applies when `sf == 0`.

```
CMP <Wn>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
SUBS WZR, <Wn>, <Wm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

```
CMP <Xn>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
SUBS XZR, <Xn>, <Xm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler symbols

`<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in the "shift" field. It can have the following values:

`LSL`	when `shift = 00`
`LSR`	when `shift = 01`
`ASR`	when `shift = 10`

The encoding `shift = 11` is reserved.

`<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

The description of SUBS (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.63 CMPP

Compare with Tag subtracts the 56-bit address held in the second source register from the 56-bit address held in the first source register, updates the condition flags based on the result of the subtraction, and discards the result.

This instruction is an alias of the SUBPS instruction. This means that:

- The encodings in this description are named to match the encodings of SUBPS.
- The description of SUBPS gives the operational pseudocode for this instruction.

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
1	0	1	1	1	0	1	0	1	1	0	Xm		0	0	0	0	0	0	Xn		1 1 1 1 1 (Xd)	

Integer variant

```
CMPP <Xn|SP>, <Xm|SP>
```

is equivalent to

```
SUBPS XZR, <Xn|SP>, <Xm|SP>
```

and is always the preferred disassembly.

Assembler symbols

`<Xn|SP>` Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.

`<Xm|SP>` Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the "Xm" field.

Operation

The description of SUBPS gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.64 CNEG

Conditional Negate returns, in the destination register, the negated value of the source register if the condition is TRUE, and otherwise returns the value of the source register.

This instruction is an alias of the CSNEG instruction. This means that:

- The encodings in this description are named to match the encodings of CSNEG.
- The description of CSNEG gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	16	15 12	11	10	9 5	4 0
sf	1	0	1	1	0	1	0	1	0	0	Rm		!=111x	0	1	Rn	Rd
op													cond		o2		

32-bit variant

Applies when `sf == 0`.

```
CNEG <Wd>, <Wn>, <cond>
```

is equivalent to

```
CSNEG <Wd>, <Wn>, <Wn>, invert(<cond>)
```

and is the preferred disassembly when `Rn == Rm`.

64-bit variant

Applies when `sf == 1`.

```
CNEG <Xd>, <Xn>, <cond>
```

is equivalent to

```
CSNEG <Xd>, <Xn>, <Xn>, invert(<cond>)
```

and is the preferred disassembly when `Rn == Rm`.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
`<cond>`	Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

The description of CSNEG gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.65 CPP

Cache Prefetch Prediction Restriction by Context prevents cache allocation predictions, based on information gathered from earlier execution within a particular execution context, from allowing later speculative execution within that context to be observable through side-channels.

For more information, see CPP RCTX.

This instruction is an alias of the SYS instruction. This means that:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode for this instruction.

ARMv8.5

31 30 29 28	27 26 25 24	23 22	21	20	19 18	16	15	12	11	8	7	5	4	0
1 1 0 1	0 1 0 1	0 0	0	0 1	0 1	1	0 1 1 1		0 0 1 1		1 1 1		Rt	
			L		op1		CRn		CRm		op2			

System variant

```
CPP RCTX, <Xt>
```

is equivalent to

```
SYS #3, C8, C3, #7, <Xt>
```

and is always the preferred disassembly.

Assembler symbols

`<Xt>` Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.66 CRC32B, CRC32H, CRC32W, CRC32X

`CRC32` checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, 32, or 64 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial `0x04C11DB7` is used for the CRC calculation.

In Armv8-A, this is an OPTIONAL instruction, and in Armv8.1 it is mandatory for all implementations to implement it.

Note

ID_AA64ISAR0_EL1.CRC32 indicates whether this instruction is supported.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
sf	0	0	1	1	0	1	0	1	1	0	Rm		0	1	0	0	sz		Rn		Rd	
																C						

CRC32B variant

Applies when `sf == 0 && sz == 00`.

```
CRC32B <Wd>, <Wn>, <Wm>
```

CRC32H variant

Applies when `sf == 0 && sz == 01`.

```
CRC32H <Wd>, <Wn>, <Wm>
```

CRC32W variant

Applies when `sf == 0 && sz == 10`.

```
CRC32W <Wd>, <Wn>, <Wm>
```

CRC32X variant

Applies when `sf == 1 && sz == 11`.

```
CRC32X <Wd>, <Wn>, <Xm>
```

Decode for all variants of this encoding

```
if !HaveCRCExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sf == '1' && sz != '11' then UNDEFINED;
if sf == '0' && sz == '11' then UNDEFINED;
integer size = 8 << UInt(sz);
```

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose accumulator output register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the general-purpose accumulator input register, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the general-purpose data source register, encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Wm>` Is the 32-bit name of the general-purpose data source register, encoded in the "Rm" field.

Operation

```
bits(32) acc = X[n];    // accumulator
bits(size) val = X[m];    // input value
bits(32) poly = 0x04C11DB7<31:0>;

bits(32+size) tempacc = BitReverse(acc):Zeros(size);
bits(size+32) tempval = BitReverse(val):Zeros(32);

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
X[d] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.67 CRC32CB, CRC32CH, CRC32CW, CRC32CX

`CRC32` checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, 32, or 64 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial `0x1EDC6F41` is used for the CRC calculation.

In Armv8-A, this is an OPTIONAL instruction, and in Armv8.1 it is mandatory for all implementations to implement it.

Note

ID_AA64ISAR0_EL1.CRC32 indicates whether this instruction is supported.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
sf	0	0	1	1	0	1	0	1	1	0	Rm		0	1	0	1	sz		Rn		Rd	
																C						

CRC32CB variant

Applies when `sf == 0 && sz == 00`.

```
CRC32CB <Wd>, <Wn>, <Wm>
```

CRC32CH variant

Applies when `sf == 0 && sz == 01`.

```
CRC32CH <Wd>, <Wn>, <Wm>
```

CRC32CW variant

Applies when `sf == 0 && sz == 10`.

```
CRC32CW <Wd>, <Wn>, <Wm>
```

CRC32CX variant

Applies when `sf == 1 && sz == 11`.

```
CRC32CX <Wd>, <Wn>, <Xm>
```

Decode for all variants of this encoding

```
if !HaveCRCExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sf == '1' && sz != '11' then UNDEFINED;
if sf == '0' && sz == '11' then UNDEFINED;
integer size = 8 << UInt(sz);
```

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose accumulator output register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the general-purpose accumulator input register, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the general-purpose data source register, encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Wm>` Is the 32-bit name of the general-purpose data source register, encoded in the "Rm" field.

Operation

```
bits(32) acc = X[n];    // accumulator
bits(size) val = X[m];    // input value
bits(32) poly = 0x1EDC6F41<31:0>;

bits(32+size) tempacc = BitReverse(acc):Zeros(size);
bits(size+32) tempval = BitReverse(val):Zeros(32);

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
X[d] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.68 CSDB

Consumption of Speculative Data Barrier is a memory barrier that controls speculative execution and data value prediction.

No instruction other than branch instructions appearing in program order after the CSDB can be speculatively executed using the results of any:

- Data value predictions of any instructions.
- PSTATE.{N,Z,C,V} predictions of any instructions other than conditional branch instructions appearing in program order before the CSDB that have not been architecturally resolved.
- Predictions of SVE predication state for any SVE instructions.

Note

For purposes of the definition of CSDB, PSTATE.{N,Z,C,V} is not considered a data value. This definition permits:

- Control flow speculation before and after the CSDB.
- Speculative execution of conditional data processing instructions after the CSDB, unless they use the results of data value or PSTATE.{N,Z,C,V} predictions of instructions appearing in program order before the CSDB that have not been architecturally resolved.

31 30 29 28	27 26 25 24	23 22	21	20	19 18 17 16	15 14 13 12	11 8	7 5	4	3 2 1 0
1 1 0 1	0 1 0 1	0 0	0	0 0	0 1 1	0 0 1 0	0 0 1 0	1 0 0	1	1 1 1 1
							CRm	op2		

System variant

```
CSDB
```

Decode for this encoding

```
// Empty.
```

Operation

```
ConsumptionOfSpeculativeDataBarrier();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.69 CSEL

Conditional Select returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the value of the second source register.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	12	11	10	9	5	4	0
sf	0	0	1	1	0	1	0	1	0	0	Rm		cond		0	0	Rn		Rd	
	op															o2				

32-bit variant

Applies when `sf == 0`.

```
CSEL <Wd>, <Wn>, <Wm>, <cond>
```

64-bit variant

Applies when `sf == 1`.

```
CSEL <Xd>, <Xn>, <Xm>, <cond>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
```

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Wm>`	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
`<cond>`	Is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];

if ConditionHolds(cond) then
    result = operand1;
else
    result = operand2;

X[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.70 CSET

Conditional Set sets the destination register to 1 if the condition is TRUE, and otherwise sets it to 0.

This instruction is an alias of the CSINC instruction. This means that:

- The encodings in this description are named to match the encodings of CSINC.
- The description of CSINC gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	16	15 12	11	10	9 5	4 0
sf	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	!=111x	0	1	1	1	1	1	1	Rd
op												Rm		cond		o2	Rn	

32-bit variant

Applies when `sf == 0`.

```
CSET <Wd>, <cond>
```

is equivalent to

```
CSINC <Wd>, WZR, WZR, invert(<cond>)
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

```
CSET <Xd>, <cond>
```

is equivalent to

```
CSINC <Xd>, XZR, XZR, invert(<cond>)
```

and is always the preferred disassembly.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<cond>`	Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

The description of CSINC gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.71 CSETM

Conditional Set Mask sets all bits of the destination register to 1 if the condition is TRUE, and otherwise sets all bits to 0.

This instruction is an alias of the CSINV instruction. This means that:

- The encodings in this description are named to match the encodings of CSINV.
- The description of CSINV gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	12	11	10	9	5	4	0
sf	1	0	1	1	0	1	0	1	0	0	1	1	1	1	1	!=111x	0	0	1	1	1	1	1	Rd
op												Rm				cond		o2		Rn				

32-bit variant

Applies when `sf == 0`.

```
CSETM <Wd>, <cond>
```

is equivalent to

```
CSINV <Wd>, WZR, WZR, invert(<cond>)
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

```
CSETM <Xd>, <cond>
```

is equivalent to

```
CSINV <Xd>, XZR, XZR, invert(<cond>)
```

and is always the preferred disassembly.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<cond>`	Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

The description of CSINV gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.72 CSINC

Conditional Select Increment returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the value of the second source register incremented by 1.

This instruction is used by the aliases CINC and CSET. See *Alias conditions* for details of when each alias is preferred.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	12	11	10	9	5	4	0
sf	0	0	1	1	0	1	0	1	0	0	Rm		cond		0	1	Rn		Rd	
	op															o2				

32-bit variant

Applies when `sf == 0`.

```
CSINC <Wd>, <Wn>, <Wm>, <cond>
```

64-bit variant

Applies when `sf == 1`.

```
CSINC <Xd>, <Xn>, <Xm>, <cond>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
```

Alias conditions

Alias	is preferred when
CINC	`Rm != '11111' && cond != '111x' && Rn != '11111' && Rn == Rm`
CSET	`Rm == '11111' && cond != '111x' && Rn == '11111'`

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Wm>`	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
`<cond>`	Is one of the standard conditions, encoded in the "cond" field in the standard way.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];

if ConditionHolds(cond) then
    result = operand1;
else
    result = operand2 + 1;

X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.73 CSINV

Conditional Select Invert returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the bitwise inversion value of the second source register.

This instruction is used by the aliases CINV and CSETM. See *Alias conditions* for details of when each alias is preferred.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	12	11	10	9	5	4	0
sf	1	0	1	1	0	1	0	1	0	0	Rm		cond		0	0	Rn		Rd	
	op															o2				

32-bit variant

Applies when `sf == 0`.

```
CSINV <Wd>, <Wn>, <Wm>, <cond>
```

64-bit variant

Applies when `sf == 1`.

```
CSINV <Xd>, <Xn>, <Xm>, <cond>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
```

Alias conditions

Alias	is preferred when
CINV	`Rm != '11111' && cond != '111x' && Rn != '11111' && Rn == Rm`
CSETM	`Rm == '11111' && cond != '111x' && Rn == '11111'`

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Wm>`	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
`<cond>`	Is one of the standard conditions, encoded in the "cond" field in the standard way.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];

if ConditionHolds(cond) then
    result = operand1;
else
    result = NOT(operand2);

X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.74 CSNEG

Conditional Select Negation returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the negated value of the second source register.

This instruction is used by the alias CNEG. See *Alias conditions* for details of when each alias is preferred.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	12	11	10	9	5	4	0
sf	1	0	1	1	0	1	0	1	0	0	Rm		cond		0	1	Rn		Rd	
	op															o2				

32-bit variant

Applies when `sf == 0`.

```
CSNEG <Wd>, <Wn>, <Wm>, <cond>
```

64-bit variant

Applies when `sf == 1`.

```
CSNEG <Xd>, <Xn>, <Xm>, <cond>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
```

Alias conditions

Alias	is preferred when
CNEG	`cond != '111x' && Rn == Rm`

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Wm>`	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
`<cond>`	Is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];

if ConditionHolds(cond) then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    result = operand1;
else
    result = NOT(operand2);
    result = result + 1;

X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.75 DC

Data Cache operation. For more information, see *op0==0b01, cache maintenance, TLB maintenance, and address translation instructions* on page C5-380.

This instruction is an alias of the SYS instruction. This means that:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode for this instruction.

31 30 29 28	27 26 25 24	23 22	21	20	19 18 16	15 12	11 8	7 5	4 0
1 1 0 1	0 1 0 1	0 0	0	0 1	op1	0 1 1 1	CRm	op2	Rt
			L			CRn			

System variant

```
DC <dc_op>, <Xt>
```

is equivalent to

```
SYS #<op1>, C7, <Cm>, #<op2>, <Xt>
```

and is the preferred disassembly when `SysOp(op1,'0111',CRm,op2) == Sys_DC`.

Assembler symbols

`<dc_op>` Is a DC instruction name, as listed for the DC system instruction group, encoded in the "op1:CRm:op2" field. It can have the following values:

IVAC	when op1 = 000, CRm = 0110, op2 = 001
ISW	when op1 = 000, CRm = 0110, op2 = 010
CSW	when op1 = 000, CRm = 1010, op2 = 010
CISW	when op1 = 000, CRm = 1110, op2 = 010
ZVA	when op1 = 011, CRm = 0100, op2 = 001
CVAC	when op1 = 011, CRm = 1010, op2 = 001
CVAU	when op1 = 011, CRm = 1011, op2 = 001
CIVAC	when op1 = 011, CRm = 1110, op2 = 001

When ARMv8.5-MemTag is implemented, the following values are also valid:

IGVAC	when op1 = 000, CRm = 0110, op2 = 011
IGSW	when op1 = 000, CRm = 0110, op2 = 100
IGDVAC	when op1 = 000, CRm = 0110, op2 = 101
IGDSW	when op1 = 000, CRm = 0110, op2 = 110
CGSW	when op1 = 000, CRm = 1010, op2 = 100
CGDSW	when op1 = 000, CRm = 1010, op2 = 110
CIGSW	when op1 = 000, CRm = 1110, op2 = 100
CIGDSW	when op1 = 000, CRm = 1110, op2 = 110
GVA	when op1 = 011, CRm = 0100, op2 = 011
GZVA	when op1 = 011, CRm = 0100, op2 = 100
CGVAC	when op1 = 011, CRm = 1010, op2 = 011
CGDVAC	when op1 = 011, CRm = 1010, op2 = 101

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`CGVAP` when `op1 = 011`, `CRm = 1100`, `op2 = 011`

`CGDVAP` when `op1 = 011`, `CRm = 1100`, `op2 = 101`

`CGVADP` when `op1 = 011`, `CRm = 1101`, `op2 = 011`

`CGDVADP` when `op1 = 011`, `CRm = 1101`, `op2 = 101`

`CIGVAC` when `op1 = 011`, `CRm = 1110`, `op2 = 011`

`CIGDVAC` when `op1 = 011`, `CRm = 1110`, `op2 = 101`

When ARMv8.2-DCPoP is implemented, the following value is also valid:

`CVAP` when `op1 = 011`, `CRm = 1100`, `op2 = 001`

When ARMv8.2-DCCVADP is implemented, the following value is also valid:

`CVADP` when `op1 = 011`, `CRm = 1101`, `op2 = 001`

`<op1>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

`<Cm>` Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

`<op2>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

`<Xt>` Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.76 DCPS1

Debug Change PE State to EL1, when executed in Debug state:

- If executed at EL0 changes the current Exception level and SP to EL1 using SP_EL1.
- Otherwise, if executed at ELx, selects SP_ELx.

The target exception level of a DCPS1 instruction is:

- EL1 if the instruction is executed at EL0.
- Otherwise, the Exception level at which the instruction is executed.

When the target Exception level of a DCPS1 instruction is ELx, on executing this instruction:

- ELR_ELx becomes UNKNOWN.
- SPSR_ELx becomes UNKNOWN.
- ESR_ELx becomes UNKNOWN.
- DLR_EL0 and DSPSR_EL0 become UNKNOWN.
- The endianness is set according to SCTLR_ELx.EE.

This instruction is UNDEFINED at EL0 in Non-secure state if EL2 is implemented and HCR_EL2.TGE == 1.

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPSn instructions, see *DCPS<n>* on page H2-6954.

31 30 29 28	27 26 25 24	23 22 21 20	5 4	3 2 1 0
1 1 0 1	0 1 0 0	1 0 1	imm16	0 0 0 0 1
				LL

System variant

```
DCPS1 {#<imm>}
```

Decode for this encoding

```
if !Halted() then UNDEFINED;
```

Assembler symbols

`<imm>` Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the "imm16" field.

Operation

```
DCPSInstruction(LL);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.77 DCPS2

Debug Change PE State to EL2, when executed in Debug state:

- If executed at EL0 or EL1 changes the current Exception level and SP to EL2 using SP_EL2.
- Otherwise, if executed at ELx, selects SP_ELx.

The target exception level of a DCPS2 instruction is:

- EL2 if the instruction is executed at an exception level that is not EL3.
- EL3 if the instruction is executed at EL3.

When the target Exception level of a DCPS2 instruction is ELx, on executing this instruction:

- ELR_ELx becomes UNKNOWN.
- SPSR_ELx becomes UNKNOWN.
- ESR_ELx becomes UNKNOWN.
- DLR_EL0 and DSPSR_EL0 become UNKNOWN.
- The endianness is set according to SCTLR_ELx.EE.

This instruction is UNDEFINED at the following exception levels:

- All exception levels if EL2 is not implemented.
- At EL0 and EL1 if EL2 is disabled in the current Security state.

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPSn instructions, see *DCPS<n>* on page H2-6954.

31 30 29 28	27 26 25 24	23 22 21 20	5 4	3 2 1 0
1 1 0 1	0 1 0 0	1 0 1	imm16	0 0 0 1 0
				LL

System variant

```
DCPS2 {#<imm>}
```

Decode for this encoding

```
if !Halted() then UNDEFINED;
```

Assembler symbols

`<imm>` Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the "imm16" field.

Operation

```
DCPSInstruction(LL);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.78 DCPS3

Debug Change PE State to EL3, when executed in Debug state:

- If executed at EL3 selects SP_EL3.
- Otherwise, changes the current Exception level and SP to EL3 using SP_EL3.

The target exception level of a DCPS3 instruction is EL3.

On executing a DCPS3 instruction:

- ELR_EL3 becomes UNKNOWN.
- SPSR_EL3 becomes UNKNOWN.
- ESR_EL3 becomes UNKNOWN.
- DLR_EL0 and DSPSR_EL0 become UNKNOWN.
- The endianness is set according to SCTLR_EL3.EE.

This instruction is UNDEFINED at all exception levels if either:

- EDSCR.SDD == 1.
- EL3 is not implemented.

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPSn instructions, see *DCPS<n>* on page H2-6954.

31 30 29 28	27 26 25 24	23 22 21 20	5 4	3 2 1 0
1 1 0 1	0 1 0 0	1 0 1	imm16	0 0 0 1 1
				LL

System variant

```
DCPS3 {#<imm>}
```

Decode for this encoding

```
if !Halted() then UNDEFINED;
```

Assembler symbols

`<imm>` Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the "imm16" field.

Operation

```
DCPSInstruction(LL);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.79 DGH

DGH is a hint instruction. A `DGH` instruction is not expected to be performance optimal to merge memory accesses with Normal Non-cacheable or Device-GRE attributes appearing in program order before the hint instruction with any memory accesses appearing after the hint instruction into a single memory transaction on an interconnect.

ARMv8.6

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	1	1	0	1	1	1	1	1
																				CRm				op2							

System variant

`DGH`

Decode for this encoding

```
if !HaveDGHExt() then EndOfInstruction();
```

Operation

```
Hint_DGH();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.80 DMB

Data Memory Barrier is a memory barrier that ensures the ordering of observations of memory accesses, see *Data Memory Barrier (DMB)* on page B2-135.

31 30 29 28	27 26 25 24	23 22	21	20	19 18 17 16	15 14 13 12	11 8	7	6 5	4	3 2 1 0
1 1 0 1	0 1 0 1	0 0	0	0 0	0 1 1	0 0 1 1	CRm	1	0 1	1	1 1 1 1
									opc		

System variant

```
DMB <option>|#<imm>
```

Decode for this encoding

```
case CRm<3:2> of
    when '00' domain = MBReqDomain_OuterShareable;
    when '01' domain = MBReqDomain_Nonshareable;
    when '10' domain = MBReqDomain_InnerShareable;
    when '11' domain = MBReqDomain_FullSystem;
case CRm<1:0> of
    when '00' types = MBReqTypes_All; domain = MBReqDomain_FullSystem;
    when '01' types = MBReqTypes_Reads;
    when '10' types = MBReqTypes_Writes;
    when '11' types = MBReqTypes_All;
```

Assembler symbols

`<option>` Specifies the limitation on the barrier operation. Values are:

- `SY` Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as CRm = `0b1111`.
- `ST` Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = `0b1110`.
- `LD` Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = `0b1101`.
- `ISH` Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = `0b1011`.
- `ISHST` Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = `0b1010`.
- `ISHLD` Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = `0b1001`.
- `NSH` Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as CRm = `0b0111`.
- `NSHST` Non-shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = `0b0110`.
- `NSHLD` Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = `0b0101`.
- `OSH` Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = `0b0011`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`OSHST` Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = `0b0010`.

`OSHLD` Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = `0b0001`.

All other encodings of `CRm` that are not listed above are reserved, and can be encoded using the `#<imm>` syntax. All unsupported and reserved options must execute as a full system barrier operation, but software must not rely on this behavior. For more information on whether an access is before or after a barrier instruction, see *Data Memory Barrier (DMB)* on page B2-135 or see *Data Synchronization Barrier (DSB)* on page B2-138.

`<imm>` Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

Operation

```
DataMemoryBarrier(domain, types);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.81 DRPS

Debug restore process state

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	1	0	1	0	1	1	1	1	1	1	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0

System variant

```
DRPS
```

Decode for this encoding

```
if !Halted() || PSTATE.EL == EL0 then UNDEFINED;
```

Operation

```
DRPSInstruction();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.82 DSB

Data Synchronization Barrier is a memory barrier that ensures the completion of memory accesses, see *Data Synchronization Barrier (DSB)* on page B2-138.

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 8	7	6 5	4	3 2 1 0
1 1 0 1	0 1 0 1	0 0 0 0	0 1 1	0 0 1 1	!=0x00	1	0 0	1	1 1 1 1
					CRm		opc		

System variant

```
DSB <option>|#<imm>
```

Decode for this encoding

```
case CRm<3:2> of
    when '00' domain = MBReqDomain_OuterShareable;
    when '01' domain = MBReqDomain_Nonshareable;
    when '10' domain = MBReqDomain_InnerShareable;
    when '11' domain = MBReqDomain_FullSystem;
case CRm<1:0> of
    when '00' types = MBReqTypes_All; domain = MBReqDomain_FullSystem;
    when '01' types = MBReqTypes_Reads;
    when '10' types = MBReqTypes_Writes;
    when '11' types = MBReqTypes_All;
```

Assembler symbols

`<option>` Specifies the limitation on the barrier operation. Values are:

- `SY` Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as CRm = `0b1111`.
- `ST` Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = `0b1110`.
- `LD` Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = `0b1101`.
- `ISH` Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = `0b1011`.
- `ISHST` Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = `0b1010`.
- `ISHLD` Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = `0b1001`.
- `NSH` Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as CRm = `0b0111`.
- `NSHST` Non-shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = `0b0110`.
- `NSHLD` Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = `0b0101`.
- `OSH` Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = `0b0011`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`OSHST` Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = `0b0010`.

`OSHLD` Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = `0b0001`.

All other encodings of `CRm`, other than the values `0b0000` and `0b0100`, that are not listed above are reserved, and can be encoded using the `#<imm>` syntax. All unsupported and reserved options must execute as a full system barrier operation, but software must not rely on this behavior. For more information on whether an access is before or after a barrier instruction, see *Data Memory Barrier (DMB)* on page B2-135 or see *Data Synchronization Barrier (DSB)* on page B2-138.

Note

The value `0b0000` is used to encode SSBB and the value `0b0100` is used to encode PSSBB.

`<imm>` Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

Operation

```
DataSynchronizationBarrier(domain, types);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.83 DVP

Data Value Prediction Restriction by Context prevents data value predictions, based on information gathered from earlier execution within an particular execution context, from allowing later speculative execution within that context to be observable through side-channels.

For more information, see DVP RCTX.

This instruction is an alias of the SYS instruction. This means that:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode for this instruction.

ARMv8.5

31 30 29 28	27 26 25 24	23 22	21	20	19 18	16	15	12	11	8	7	5	4	0
1 1 0 1	0 1 0 1	0 0	0	0 1	0 1	1	0 1 1	1	0 0 1	1	1 0	1	Rt	
			L		op1		CRn		CRm		op2			

System variant

```
DVP RCTX, <Xt>
```

is equivalent to

```
SYS #3, C8, C3, #5, <Xt>
```

and is always the preferred disassembly.

Assembler symbols

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.84 EON (shifted register)

Bitwise Exclusive OR NOT (shifted register) performs a bitwise Exclusive OR NOT of a register value and an optionally-shifted register value, and writes the result to the destination register.

31	30 29	28	27 26 25 24	23 22	21	20 16	15 10	9 5	4 0
sf	1 0	0	1 0 1 0	shift	1	Rm	imm6	Rn	Rd
	opc				N				

32-bit variant

Applies when `sf == 0`.

```
EON <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit variant

Applies when `sf == 1`.

```
EON <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:

`LSL`	when `shift = 00`
`LSR`	when `shift = 01`
`ASR`	when `shift = 10`
`ROR`	when `shift = 11`

`<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

operand2 = NOT(operand2);

result = operand1 EOR operand2;

X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.85 EOR (immediate)

Bitwise Exclusive OR (immediate) performs a bitwise Exclusive OR of a register value and an immediate value, and writes the result to the destination register.

31	30 29	28 27 26 25 24	23	22 21	16 15	10 9	5 4	0
sf	1 0 (opc)	1 0 0 1 0 0	N	immr	imms	Rn	Rd	

32-bit variant

Applies when `sf == 0 && N == 0`.

```
EOR <Wd|WSP>, <Wn>, #<imm>
```

64-bit variant

Applies when `sf == 1`.

```
EOR <Xd|SP>, <Xn>, #<imm>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;
if sf == '0' && N != '0' then UNDEFINED;
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);
```

Assembler symbols

`<Wd|WSP>`	Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
`<Xd|SP>`	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
`<imm>`	For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr". For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];

result = operand1 EOR imm;

if d == 31 then
    SP[] = result;
else
    X[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.86 EOR (shifted register)

Bitwise Exclusive OR (shifted register) performs a bitwise Exclusive OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

31	30 29	28	27 26 25 24	23 22	21	20 16	15 10	9 5	4 0
sf	1 0	0	1 0 1 0	shift	0	Rm	imm6	Rn	Rd
	opc				N				

32-bit variant

Applies when `sf == 0`.

```
EOR <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit variant

Applies when `sf == 1`.

```
EOR <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:

- `LSL` when `shift = 00`
- `LSR` when `shift = 01`
- `ASR` when `shift = 10`
- `ROR` when `shift = 11`

`<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

result = operand1 EOR operand2;

X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.87 ERET

Exception Return using the ELR and SPSR for the current Exception level. When executed, the PE restores PSTATE from the SPSR, and branches to the address held in the ELR.

The PE checks the SPSR for the current Exception level for an illegal return event. See *Illegal return events from AArch64 state* on page D1-2351.

ERET is UNDEFINED at EL0.

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11	10	9 5	4 0
1 1 0 1	0 1 1 1	0 0 1 0	1 1 1 1	0 0 0 0	0	0	1 1 1 1 1	0 0 0 0 0
					A	M	Rn	op4

System variant

```
ERET
```

Decode for this encoding

```
if PSTATE.EL == EL0 then UNDEFINED;
```

Operation

```
AArch64.CheckForERetTrap(FALSE, TRUE);
bits(64) target = ELR[];

AArch64.ExceptionReturn(target, SPSR[]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.88 ERETAA, ERETAB

Exception Return, with pointer authentication. This instruction authenticates the address in ELR, using SP as the modifier and the specified key, the PE restores PSTATE from the SPSR for the current Exception level, and branches to the authenticated address.

Key A is used for `ERETAA`, and key B is used for `ERETAB`.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to ELR.

The PE checks the SPSR for the current Exception level for an illegal return event. See *Illegal return events from AArch64 state* on page D1-2351.

`ERETAA` and `ERETAB` are UNDEFINED at EL0.

ARMv8.3

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9		5	4		0
1	1	0	1	0	1	1	0	1	0	0	1	1	1	1	1	0	0	0	0	1	M	1	1	1	1	1	1	1	1	1	1
																				A		Rn				op4	

ERETAA variant

Applies when `M == 0`.

`ERETAA`

ERETAB variant

Applies when `M == 1`.

`ERETAB`

Decode for all variants of this encoding

```
if PSTATE.EL == EL0 then UNDEFINED;
boolean use_key_a = (M == '0');

if !HavePACExt() then
    UNDEFINED;
```

Operation

```
AArch64.CheckForERetTrap(TRUE, use_key_a);
bits(64) target;

if use_key_a then
    target = AuthIA(ELR[], SP[], TRUE);
else
    target = AuthIB(ELR[], SP[], TRUE);

AArch64.ExceptionReturn(target, SPSR[]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.89 ESB

Error Synchronization Barrier is an error synchronization event that might also update DISR_EL1 and VDISR_EL2.

This instruction can be used at all Exception levels and in Debug state.

In Debug state, this instruction behaves as if SError interrupts are masked at all Exception levels. See Error Synchronization Barrier in the Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for Armv8-A architecture profile.

If the RAS Extension is not implemented, this instruction executes as a `NOP`.

ARMv8.2

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 8	7 5	4	3 2 1 0
1 1 0 1	0 1 0 1	0 0 0 0	0 0	0 1 1	0 0 1 0	0 0 0	1	1 1 1 1
					CRm	op2		

System variant

```
ESB
```

Decode for this encoding

```
if !HaveRASExt() then EndOfInstruction();
```

Operation

```
SynchronizeErrors();
AArch64.ESBOperation();
if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch64.vESBOperation();
TakeUnmaskedSErrorInterrupts();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.90 EXTR

Extract register extracts a register from a pair of registers.

This instruction is used by the alias ROR (immediate). See *Alias conditions* for details of when each alias is preferred.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	10	9	5	4	0
sf	0	0	1	0	0	1	1	1	N	0	Rm		imms		Rn		Rd	

32-bit variant

Applies when `sf == 0 && N == 0 && imms == 0xxxxx`.

```
EXTR <Wd>, <Wn>, <Wm>, #<lsb>
```

64-bit variant

Applies when `sf == 1 && N == 1`.

```
EXTR <Xd>, <Xn>, <Xm>, #<lsb>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
integer lsb;

if N != sf then UNDEFINED;
if sf == '0' && imms<5> == '1' then UNDEFINED;
lsb = UInt(imms);
```

Alias conditions

Alias	is preferred when
ROR (immediate)	`Rn == Rm`

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Wm>`	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
`<lsb>`	For the 32-bit variant: is the least significant bit position from which to extract, in the range 0 to 31, encoded in the "imms" field. For the 64-bit variant: is the least significant bit position from which to extract, in the range 0 to 63, encoded in the "imms" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(2*datasize) concat = operand1:operand2;

result = concat<lsb+datasize-1:lsb>;

X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.91 GMI

Tag Mask Insert inserts the tag in the first source register into the excluded set specified in the second source register, writing the new excluded set to the destination register.

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
1	0	0	1	1	0	1	0	1	1	0	Xm		0	0	0	1	0	1	Xn		Xd	

Integer variant

```
GMI <Xd>, <Xn|SP>, <Xm>
```

Decode for this encoding

```
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);
```

Assembler symbols

`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Xd" field.
`<Xn|SP>`	Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.
`<Xm>`	Is the 64-bit name of the second general-purpose source register, encoded in the "Xm" field.

Operation

```
bits(64) address = if n == 31 then SP[] else X[n];
bits(64) mask = X[m];
bits(4) tag = AArch64.AllocationTagFromAddress(address);

mask<UInt(tag)> = '1';
X[d] = mask;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.92 HINT

Hint instruction is for the instruction set space that is reserved for architectural hint instructions.

Some encodings described here are not allocated in this revision of the architecture, and behave as NOPs. These encodings might be allocated to other hint functionality in future revisions of the architecture and therefore must not be used by software.

31 30 29 28	27 26 25 24	23 22	21	20 19	18 17 16	15 14 13 12	11 8	7 5	4	3 2 1 0
1 1 0 1	0 1 0 1	0 0	0	0 0	0 1 1	0 0 1 0	CRm	op2	1	1 1 1 1

System variant

```
HINT #<imm>
```

Decode for this encoding

```
SystemHintOp op;

case CRm:op2 of
    when '0000 000' op = SystemHintOp_NOP;
    when '0000 001' op = SystemHintOp_YIELD;
    when '0000 010' op = SystemHintOp_WFE;
    when '0000 011' op = SystemHintOp_WFI;
    when '0000 100' op = SystemHintOp_SEV;
    when '0000 101' op = SystemHintOp_SEVL;
    when '0000 110'
        if !HaveDGHExt() then EndOfInstruction();    // Instruction executes as NOP
        op = SystemHintOp_DGH;
    when '0000 111' SEE "XPACLRI";
    when '0001 xxx'
        case op2 of
            when '000' SEE "PACIA1716";
            when '010' SEE "PACIB1716";
            when '100' SEE "AUTIA1716";
            when '110' SEE "AUTIB1716";
            otherwise EndOfInstruction();
    when '0010 000'
        if !HaveRASExt() then EndOfInstruction();    // Instruction executes as NOP
        op = SystemHintOp_ESB;
    when '0010 001'
        if !HaveStatisticalProfiling() then EndOfInstruction();    // Instruction executes as NOP
        op = SystemHintOp_PSB;
    when '0010 010'
        if !HaveSelfHostedTrace() then EndOfInstruction();    // Instruction executes as NOP
        op = SystemHintOp_TSB;
    when '0010 100'
        op = SystemHintOp_CSDB;
    when '0011 xxx'
        case op2 of
            when '000' SEE "PACIAZ";
            when '001' SEE "PACIASP";
            when '010' SEE "PACIBZ";
            when '011' SEE "PACIBSP";
            when '100' SEE "AUTIAZ";
            when '101' SEE "AUTHASP";
            when '110' SEE "AUTIBZ";
            when '111' SEE "AUTIBSP";
    when '0100 xx0'
        op = SystemHintOp_BTI;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        // Check branch target compatibility between BTI instruction and PSTATE.BTYPE
        SetBTypeCompatible(BTypeCompatible_BTI(op2<2:1>));
    otherwise EndOfInstruction();
```

Assembler symbols

`<imm>` Is a 7-bit unsigned immediate, in the range 0 to 127 encoded in the "CRm:op2" field.

The encodings that are allocated to architectural hint functionality are described in the "Hints" table in the "Index by Encoding".

Note

For allocated encodings of "CRm:op2":

- A disassembler will disassemble the allocated instruction, rather than the `HINT` instruction.
- An assembler may support assembly of allocated encodings using `HINT` with the corresponding `<imm>` value, but it is not required to do so.

Operation

```
case op of
    when SystemHintOp_YIELD
        Hint_Yield();

    when SystemHintOp_DGH
        Hint_DGH();

    when SystemHintOp_WFE
        if IsEventRegisterSet() then
            ClearEventRegister();
        else
            if PSTATE.EL == EL0 then
                // Check for traps described by the OS which may be EL1 or EL2.
                AArch64.CheckForWFxTrap(EL1, TRUE);
            if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
                // Check for traps described by the Hypervisor.
                AArch64.CheckForWFxTrap(EL2, TRUE);
            if HaveEL(EL3) && PSTATE.EL != EL3 then
                // Check for traps described by the Secure Monitor.
                AArch64.CheckForWFxTrap(EL3, TRUE);
            WaitForEvent();

    when SystemHintOp_WFI
        if !InterruptPending() then
            if PSTATE.EL == EL0 then
                // Check for traps described by the OS which may be EL1 or EL2.
                AArch64.CheckForWFxTrap(EL1, FALSE);
            if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
                // Check for traps described by the Hypervisor.
                AArch64.CheckForWFxTrap(EL2, FALSE);
            if HaveEL(EL3) && PSTATE.EL != EL3 then
                // Check for traps described by the Secure Monitor.
                AArch64.CheckForWFxTrap(EL3, FALSE);
            WaitForInterrupt();

    when SystemHintOp_SEV
        SendEvent();

    when SystemHintOp_SEVL
        SendEventLocal();

    when SystemHintOp_ESB
        SynchronizeErrors();
        AArch64.ESBOperation();
        if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch64.vESBOperation();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    TakeUnmaskedSErrorInterrupts();

when SystemHintOp_PSB
    ProfilingSynchronizationBarrier();

when SystemHintOp_TSB
    TraceSynchronizationBarrier();

when SystemHintOp_CSDB
    ConsumptionOfSpeculativeDataBarrier();

when SystemHintOp_BTI
    SetBTypeNext('00');

otherwise    // do nothing
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.93 HLT

Halt instruction. A HLTinstruction can generate a Halt Instruction debug event, which causes entry into Debug state.

31 30 29 28	27 26 25 24	23 22 21 20	5 4	3 2 1 0
1 1 0 1	0 1 0 0	0 1 0	imm16	0 0 0 0 0

System variant

```
HLT #<imm>
```

Decode for this encoding

```
if EDSCR.HDE == '0' || !HaltingAllowed() then UNDEFINED;
if HaveBTIExt() then
    SetBTypeCompatible(TRUE);
```

Assembler symbols

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

Operation

```
Halt(DebugHalt_HaltInstruction);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.94 HVC

Hypervisor Call causes an exception to EL2. Non-secure software executing at EL1 can use this instruction to call the hypervisor to request a service.

The `HVC` instruction is UNDEFINED:

- At EL0.
- At EL1 if EL2 is not enabled in the current Security state.
- When SCR_EL3.HCE is set to 0.

On executing an `HVC` instruction, the PE records the exception as a Hypervisor Call exception in ESR_ELx, using the EC value `0x16`, and the value of the immediate argument.

31 30 29 28	27 26 25 24	23 22 21 20	5 4	3 2 1 0
1 1 0 1	0 1 0 0	0 0 0	imm16	0 0 0 1 0

System variant

```
HVC #<imm>
```

Decode for this encoding

```
// Empty.
```

Assembler symbols

`<imm>` Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

Operation

```
if !HaveEL(EL2) || PSTATE.EL == EL0 || (PSTATE.EL == EL1 && (!IsSecureEL2Enabled() && IsSecure())) then
    UNDEFINED;

hvc_enable = if HaveEL(EL3) then SCR_EL3.HCE else NOT(HCR_EL2.HCD);

if hvc_enable == '0' then
    UNDEFINED;
else
    AArch64.CallHypervisor(imm16);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.95 IC

Instruction Cache operation. For more information, see *op0==0b01, cache maintenance, TLB maintenance, and address translation instructions* on page C5-380.

This instruction is an alias of the SYS instruction. This means that:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode for this instruction.

31 30 29 28	27 26 25 24	23 22	21	20	19 18 16	15 12	11 8	7 5	4 0
1 1 0 1	0 1 0 1	0 0	0	0 1	op1	0 1 1 1	CRm	op2	Rt
			L			CRn			

System variant

```
IC <ic_op>{, <Xt>}
```

is equivalent to

```
SYS #<op1>, C7, <Cm>, #<op2>{, <Xt>}
```

and is the preferred disassembly when `SysOp(op1,'0111',CRm,op2) == Sys_IC`.

Assembler symbols

`<ic_op>` Is an IC instruction name, as listed for the IC system instruction pages, encoded in the "op1:CRm:op2" field. It can have the following values:

`IALLUIS`	when `op1 = 000`, `CRm = 0001`, `op2 = 000`
`IALLU`	when `op1 = 000`, `CRm = 0101`, `op2 = 000`
`IVAU`	when `op1 = 011`, `CRm = 0101`, `op2 = 001`

`<op1>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

`<Cm>` Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

`<op2>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

`<Xt>` Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.96 IRG

Insert Random Tag inserts a random Logical Address Tag into the address in the first source register, and writes the result to the destination register. Any tags specified in the optional second source register or in GCR_EL1.Exclude are excluded from the selection of the random Logical Address Tag.

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
1	0	0	1	1	0	1	0	1	1	0	Xm		0	0	0	1	0	0	Xn		Xd	

Integer variant

```
IRG <Xd|SP>, <Xn|SP>{, <Xm>}
```

Decode for this encoding

```
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);
```

Assembler symbols

`<Xd|SP>` Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Xd" field.

`<Xn|SP>` Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Xm" field. Defaults to XZR if absent.

Operation

```
bits(64) operand = if n == 31 then SP[] else X[n];
bits(64) exclude_reg = X[m];
bits(16) exclude = exclude_reg<15:0> OR GCR_EL1.Exclude;

if AArch64.AllocationTagAccessIsEnabled() then
    if GCR_EL1.RRND == '1' then
        rtag = _ChooseRandomNonExcludedTag(exclude);
    else
        bits(4) start = RGSR_EL1.TAG;
        bits(4) offset = AArch64.RandomTag();

        rtag = AArch64.ChooseNonExcludedTag(start, offset, exclude);

        RGSR_EL1.TAG = rtag;
else
    rtag = '0000';

bits(64) result = AArch64.AddressWithAllocationTag(operand, rtag);

if d == 31 then
    SP[] = result;
else
    X[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.97 ISB

Instruction Synchronization Barrier flushes the pipeline in the PE and is a context synchronization event. For more information, see *Instruction Synchronization Barrier (ISB)* on page B2-135.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	1	CRm		1	1	0	1	1	1	1	1
																							opc						

System variant

```
ISB {<option>|#<imm>}
```

Decode for this encoding

```
// No additional decoding required
```

Assembler symbols

`<option>` Specifies an optional limitation on the barrier operation. Values are:

`SY` Full system barrier operation, encoded as `CRm` = `0b1111`. Can be omitted.

All other encodings of `CRm` are reserved. The corresponding instructions execute as full system barrier operations, but must not be relied upon by software.

`<imm>` Is an optional 4-bit unsigned immediate, in the range 0 to 15, defaulting to 15 and encoded in the "CRm" field.

Operation

```
InstructionSynchronizationBarrier();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.98 LDADDB, LDADDAB, LDADDALB, LDADDLB

Atomic add on byte in memory atomically loads an 8-bit byte from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDADDAB` and `LDADDALB` load from memory with acquire semantics.
- `LDADDLB` and `LDADDALB` store to memory with release semantics.
- `LDADDB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is used by the alias STADDB, STADDLB. See *Alias conditions* on page C6-938 for details of when each alias is preferred.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	A	R	1	Rs	0	0 0 0	0 0	Rn	Rt
size									opc			

LDADDAB variant

Applies when `A == 1 && R == 0`.

```
LDADDAB <Ws>, <Wt>, [<Xn|SP>]
```

LDADDALB variant

Applies when `A == 1 && R == 1`.

```
LDADDALB <Ws>, <Wt>, [<Xn|SP>]
```

LDADDB variant

Applies when `A == 0 && R == 0`.

```
LDADDB <Ws>, <Wt>, [<Xn|SP>]
```

LDADDLB variant

Applies when `A == 0 && R == 1`.

```
LDADDLB <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Alias conditions

Alias	is preferred when
STADDB, STADDLB	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_ADD, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.99 LDADDH, LDADDAH, LDADDALH, LDADDLH

Atomic add on halfword in memory atomically loads a 16-bit halfword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDADDAH` and `LDADDALH` load from memory with acquire semantics.
- `LDADDLH` and `LDADDALH` store to memory with release semantics.
- `LDADDH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is used by the alias STADDH, STADDLH. See *Alias conditions* on page C6-940 for details of when each alias is preferred.

ARMv8.1

31 30	29 28	27 26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1	1	0 0	A	R	1	Rs	0	0 0 0	0 0	Rn	Rt
size								opc				

LDADDAH variant

Applies when `A == 1 && R == 0`.

```
LDADDAH <Ws>, <Wt>, [<Xn|SP>]
```

LDADDALH variant

Applies when `A == 1 && R == 1`.

```
LDADDALH <Ws>, <Wt>, [<Xn|SP>]
```

LDADDH variant

Applies when `A == 0 && R == 0`.

```
LDADDH <Ws>, <Wt>, [<Xn|SP>]
```

LDADDLH variant

Applies when `A == 0 && R == 1`.

```
LDADDLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Alias conditions

Alias	is preferred when
STADDH, STADDLH	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_ADD, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.100 LDADD, LDADDA, LDADDAL, LDADDL

Atomic add on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDADDA` and `LDADDAL` load from memory with acquire semantics.
- `LDADDL` and `LDADDAL` store to memory with release semantics.
- `LDADD` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is used by the alias STADD, STADDL. See *Alias conditions* on page C6-942 for details of when each alias is preferred.

ARMv8.1

31 30	29 28	27 26	25 24	23	22	21	20 … 16	15	14 … 12	11 10	9 … 5	4 … 0
1 x	1 1	1	0 0	A	R	1	Rs	0	0 0 0	0 0	Rn	Rt
size									opc			

32-bit LDADD variant

Applies when `size == 10 && A == 0 && R == 0`.

```
LDADD <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDADDA variant

Applies when `size == 10 && A == 1 && R == 0`.

```
LDADDA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDADDAL variant

Applies when `size == 10 && A == 1 && R == 1`.

```
LDADDAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDADDL variant

Applies when `size == 10 && A == 0 && R == 1`.

```
LDADDL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit LDADD variant

Applies when `size == 11 && A == 0 && R == 0`.

```
LDADD <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDADDA variant

Applies when `size == 11 && A == 1 && R == 0`.

```
LDADDA <Xs>, <Xt>, [<Xn|SP>]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

64-bit LDADDAL variant

Applies when `size == 11 && A == 1 && R == 1`.

```
LDADDAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDADDL variant

Applies when `size == 11 && A == 0 && R == 1`.

```
LDADDL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Alias conditions

Alias	is preferred when
STADD, STADDL	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Wt>`	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
`<Xs>`	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_ADD, value, ldacctype, stacctype);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.101 LDAPR

Load-Acquire RCpc Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from the derived address in memory, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

ARMv8.3

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
1	x	1	1	1	0	0	0	1	0	1	(1)(1)(1)(1)(1)		1	1	0	0	0	0	Rn		Rt	
size											Rs											

32-bit variant

Applies when `size == 10`.

```
LDAPR <Wt>, [<Xn|SP> {,#0}]
```

64-bit variant

Applies when `size == 11`.

```
LDAPR <Xt>, [<Xn|SP> {,#0}]
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
boolean tag_checked = n != 31;
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = Mem[address, dbytes, AccType_ORDERED];
X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.102 LDAPRB

Load-Acquire RCpc Register Byte derives an address from a base register value, loads a byte from the derived address in memory, zero-extends it and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

ARMv8.3

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 13 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	1	0	1	(1)(1)(1)(1)(1)	1	1 0 0	0 0	Rn	Rt
size							Rs					

Integer variant

```
LDAPRB <Wt>, [<Xn|SP> {,#0}]
```

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = Mem[address, 1, AccType_ORDERED];
X[t] = ZeroExtend(data, 32);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.103 LDAPRH

Load-Acquire RCpc Register Halfword derives an address from a base register value, loads a halfword from the derived address in memory, zero-extends it and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

ARMv8.3

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 13 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	1	0	1	(1)(1)(1)(1)(1)	1	1 0 0	0 0	Rn	Rt
size							Rs					

Integer variant

```
LDAPRH <Wt>, [<Xn|SP> {,#0}]
```

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = Mem[address, 2, AccType_ORDERED];
X[t] = ZeroExtend(data, 32);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.104 LDAPUR

Load-Acquire RCpc Register (unscaled) calculates an address from a base register and an immediate offset, loads a 32-bit word or 64-bit doubleword from memory, zero-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

ARMv8.4

31 30	29 28 27 26 25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
1 x (size)	0 1 1 0 0 1	0 1 (opc)	0	imm9	0 0	Rn	Rt

32-bit variant

Applies when `size == 10`.

```
LDAPUR <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit variant

Applies when `size == 11`.

```
LDAPUR <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
boolean tag_checked = n != 31;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, datasize DIV 8, AccType_ORDERED];
X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.105 LDAPURB

Load-Acquire RCpc Register Byte (unscaled) calculates an address from a base register and an immediate offset, loads a byte from memory, zero-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

ARMv8.4

31 30	29 28 27 26 25 24	23 22	21	20 12	11 10	9 5	4 0
0 0	0 1 1 0 0 1	0 1	0	imm9	0 0	Rn	Rt
size		opc					

Unscaled offset variant

```
LDAPURB <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    address = X[n];

address = address + offset;

data = Mem[address, 1, AccType_ORDERED];
X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.106 LDAPURH

Load-Acquire RCpc Register Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a halfword from memory, zero-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

ARMv8.4

31 30	29 28 27	26 25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 1	0 1 1	0 0 1	0 1	0	imm9	0 0	Rn	Rt
size			opc					

Unscaled offset variant

```
LDAPURH <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    address = X[n];

address = address + offset;

data = Mem[address, 2, AccType_ORDERED];
X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.107 LDAPURSB

Load-Acquire RCpc Register Signed Byte (unscaled) calculates an address from a base register and an immediate offset, loads a signed byte from memory, sign-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

ARMv8.4

31 30	29 28 27 26 25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 0	0 1 1 0 0 1	1 x	0	imm9	0 0	Rn	Rt
size		opc					

32-bit variant

Applies when `opc == 11`.

```
LDAPURSB <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit variant

Applies when `opc == 10`.

```
LDAPURSB <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);
```

Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 1, AccType_ORDERED] = data;

    when MemOp_LOAD
        data = Mem[address, 1, AccType_ORDERED];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.108 LDAPURSH

Load-Acquire RCpc Register Signed Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a signed halfword from memory, sign-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

ARMv8.4

31 30	29 28 27 26 25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 1	0 1 1 0 0 1	1 x	0	imm9	0 0	Rn	Rt
size		opc					

32-bit variant

Applies when `opc == 11`.

```
LDAPURSH <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit variant

Applies when `opc == 10`.

```
LDAPURSH <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);
```

Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 2, AccType_ORDERED] = data;

    when MemOp_LOAD
        data = Mem[address, 2, AccType_ORDERED];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.109 LDAPURSW

Load-Acquire RCpc Register Signed Word (unscaled) calculates an address from a base register and an immediate offset, loads a signed word from memory, sign-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

ARMv8.4

31 30	29 28 27 26 25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
1 0	0 1 1 0 0 1	1 0	0	imm9	0 0	Rn	Rt
size		opc					

Unscaled offset variant

```
LDAPURSW <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(32) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    address = X[n];

address = address + offset;

data = Mem[address, 4, AccType_ORDERED];
X[t] = SignExtend(data, 64);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.110 LDAR

Load-Acquire Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

Note

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

31 30	29 28 27 26 25 24	23	22	21	20 16	15	14 10	9 5	4 0
1 x	0 0 1 0 0 0	1	1	0	(1)(1)(1)(1)(1)	1	(1)(1)(1)(1)(1)	Rn	Rt
size			L		Rs	o0	Rt2		

32-bit variant

Applies when `size == 10`.

```
LDAR <Wt>, [<Xn|SP>{,#0}]
```

64-bit variant

Applies when `size == 11`.

```
LDAR <Xt>, [<Xn|SP>{,#0}]
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
boolean tag_checked = n != 31;
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
data = Mem[address, dbytes, AccType_ORDERED];
X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.111 LDARB

Load-Acquire Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

Note

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

31 30	29 28 27 26 25 24	23	22	21	20 16	15	14 10	9 5	4 0
0 0	0 0 1 0 0 0	1	1	0	(1)(1)(1)(1)(1)	1	(1)(1)(1)(1)(1)	Rn	Rt
size			L		Rs	o0	Rt2		

No offset variant

```
LDARB <Wt>, [<Xn|SP>{,#0}]
```

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = Mem[address, 1, AccType_ORDERED];
X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.112 LDARH

Load-Acquire Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it, and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

Note

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

31 30	29 28 27 26 25 24	23	22	21	20 16	15	14 10	9 5	4 0
0 1	0 0 1 0 0 0	1	1	0	(1)(1)(1)(1)(1)	1	(1)(1)(1)(1)(1)	Rn	Rt
size			L		Rs	o0	Rt2		

No offset variant

```
LDARH <Wt>, [<Xn|SP>{,#0}]
```

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = Mem[address, 2, AccType_ORDERED];
X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.113 LDAXP

Load-Acquire Exclusive Pair of Registers derives an address from a base register value, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and is single-copy atomic for each doubleword at doubleword granularity. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores* on page B2-167. The instruction also has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140. For information about memory accesses see *Load/Store addressing modes* on page C1-189.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	10	9	5	4	0
1	sz	0	0	1	0	0	0	0	1	1	(1)	(1)	(1)	(1)	(1)	1	Rt2	Rn	Rt
									L		Rs					o0			

32-bit variant

Applies when `sz == 0`.

```
LDAXP <Wt1>, <Wt2>, [<Xn|SP>{,#0}]
```

64-bit variant

Applies when `sz == 1`.

```
LDAXP <Xt1>, <Xt2>, [<Xn|SP>{,#0}]
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);

integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2;
boolean tag_checked = n != 31;
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *LDAXP* on page K1-7862.

Assembler symbols

`<Wt1>`	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
`<Wt2>`	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
`<Xt1>`	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt2>`	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    X[t] = bits(datasize) UNKNOWN;    // In this case t = t2
elsif elsize == 32 then
    // 32-bit load exclusive pair (atomic)
    data = Mem[address, dbytes, AccType_ORDEREDATOMIC];
    if BigEndian() then
        X[t] = data<datasize-1:elsize>;
        X[t2] = data<elsize-1:0>;
    else
        X[t] = data<elsize-1:0>;
        X[t2] = data<datasize-1:elsize>;
else // elsize == 64
    // 64-bit load exclusive pair (not atomic),
    // but must be 128-bit aligned
    if address != Align(address, dbytes) then
        AArch64.Abort(address, AArch64.AlignmentFault(AccType_ORDEREDATOMIC, FALSE, FALSE));
    X[t] = Mem[address, 8, AccType_ORDEREDATOMIC];
    X[t2] = Mem[address+8, 8, AccType_ORDEREDATOMIC];
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.114 LDAXR

Load-Acquire Exclusive Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores* on page B2-167. The instruction also has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140. For information about memory accesses see *Load/Store addressing modes* on page C1-189.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	10	9	5	4	0
1	x	0	0	1	0	0	0	0	1	0	(1)(1)(1)(1)(1)		1	(1)(1)(1)(1)(1)		Rn		Rt	
size									L		Rs		o0	Rt2					

32-bit variant

Applies when `size == 10`.

```
LDAXR <Wt>, [<Xn|SP>{,#0}]
```

64-bit variant

Applies when `size == 11`.

```
LDAXR <Xt>, [<Xn|SP>{,#0}]
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
boolean tag_checked = n != 31;
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

data = Mem[address, dbytes, AccType_ORDEREDATOMIC];
X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.115 LDAXRB

Load-Acquire Exclusive Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores* on page B2-167. The instruction also has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140. For information about memory accesses see *Load/Store addressing modes* on page C1-189.

31 30	29 28	27 26	25 24	23	22	21	20 16	15	14 10	9 5	4 0
0 0	0 0	1 0	0 0	0	1	0	(1)(1)(1)(1)(1)	1	(1)(1)(1)(1)(1)	Rn	Rt
size					L		Rs	o0	Rt2		

No offset variant

```
LDAXRB <Wt>, [<Xn|SP>{,#0}]
```

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 1);

data = Mem[address, 1, AccType_ORDEREDATOMIC];
X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.116 LDAXRH

Load-Acquire Exclusive Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores* on page B2-167. The instruction also has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140. For information about memory accesses see *Load/Store addressing modes* on page C1-189.

31 30	29 28	27 26	25 24	23	22	21	20 16	15	14 10	9 5	4 0
0 1	0 0	1 0	0 0	0	1	0	(1)(1)(1)(1)(1)	1	(1)(1)(1)(1)(1)	Rn	Rt
size					L		Rs	o0	Rt2		

No offset variant

```
LDAXRH <Wt>, [<Xn|SP>{,#0}]
```

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 2);

data = Mem[address, 2, AccType_ORDEREDATOMIC];
X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.117 LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB

Atomic bit clear on byte in memory atomically loads an 8-bit byte from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDCLRAB` and `LDCLRALB` load from memory with acquire semantics.
- `LDCLRLB` and `LDCLRALB` store to memory with release semantics.
- `LDCLRB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is used by the alias STCLRB, STCLRLB. See *Alias conditions* on page C6-973 for details of when each alias is preferred.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	A	R	1	Rs	0	0 0 1	0 0	Rn	Rt
size								opc				

LDCLRAB variant

Applies when `A == 1 && R == 0`.

```
LDCLRAB <Ws>, <Wt>, [<Xn|SP>]
```

LDCLRALB variant

Applies when `A == 1 && R == 1`.

```
LDCLRALB <Ws>, <Wt>, [<Xn|SP>]
```

LDCLRB variant

Applies when `A == 0 && R == 0`.

```
LDCLRB <Ws>, <Wt>, [<Xn|SP>]
```

LDCLRLB variant

Applies when `A == 0 && R == 1`.

```
LDCLRLB <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Alias conditions

Alias	is preferred when
STCLRB, STCLRLB	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_BIC, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.118 LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH

Atomic bit clear on halfword in memory atomically loads a 16-bit halfword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDCLRAH` and `LDCLRALH` load from memory with acquire semantics.
- `LDCLRLH` and `LDCLRALH` store to memory with release semantics.
- `LDCLRH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is used by the alias STCLRH, STCLRLH. See *Alias conditions* on page C6-975 for details of when each alias is preferred.

ARMv8.1

31 30	29 28	27	26	25	24	23	22	21	20 … 16	15	14 … 12	11 10	9 … 5	4 … 0
0 1	1 1	1	0	0	0	A	R	1	Rs	0	0 0 1	0 0	Rn	Rt
size										opc				

LDCLRAH variant

Applies when `A == 1 && R == 0`.

```
LDCLRAH <Ws>, <Wt>, [<Xn|SP>]
```

LDCLRALH variant

Applies when `A == 1 && R == 1`.

```
LDCLRALH <Ws>, <Wt>, [<Xn|SP>]
```

LDCLRH variant

Applies when `A == 0 && R == 0`.

```
LDCLRH <Ws>, <Wt>, [<Xn|SP>]
```

LDCLRLH variant

Applies when `A == 0 && R == 1`.

```
LDCLRLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Alias conditions

Alias	is preferred when
STCLRH, STCLRLH	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_BIC, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.119 LDCLR, LDCLRA, LDCLRAL, LDCLRL

Atomic bit clear on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDCLRA` and `LDCLRAL` load from memory with acquire semantics.
- `LDCLRL` and `LDCLRAL` store to memory with release semantics.
- `LDCLR` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is used by the alias STCLR, STCLRL. See *Alias conditions* on page C6-977 for details of when each alias is preferred.

ARMv8.1

31 30	29 28	27 26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
1 x	1 1	1	0 0	A	R	1	Rs	0	0 0 1	0 0	Rn	Rt
size									opc			

32-bit LDCLR variant

Applies when `size == 10 && A == 0 && R == 0`.

```
LDCLR <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDCLRA variant

Applies when `size == 10 && A == 1 && R == 0`.

```
LDCLRA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDCLRAL variant

Applies when `size == 10 && A == 1 && R == 1`.

```
LDCLRAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDCLRL variant

Applies when `size == 10 && A == 0 && R == 1`.

```
LDCLRL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit LDCLR variant

Applies when `size == 11 && A == 0 && R == 0`.

```
LDCLR <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDCLRA variant

Applies when `size == 11 && A == 1 && R == 0`.

```
LDCLRA <Xs>, <Xt>, [<Xn|SP>]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

64-bit LDCLRAL variant

Applies when `size == 11 && A == 1 && R == 1`.

```
LDCLRAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDCLRL variant

Applies when `size == 11 && A == 0 && R == 1`.

```
LDCLRL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Alias conditions

Alias	is preferred when
STCLR, STCLRL	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Wt>`	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
`<Xs>`	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_BIC, value, ldacctype, stacctype);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.120 LDEORB, LDEORAB, LDEORALB, LDEORLB

Atomic exclusive OR on byte in memory atomically loads an 8-bit byte from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDEORAB` and `LDEORALB` load from memory with acquire semantics.
- `LDEORLB` and `LDEORALB` store to memory with release semantics.
- `LDEORB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is used by the alias STEORB, STEORLB. See *Alias conditions* on page C6-980 for details of when each alias is preferred.

ARMv8.1

31 30	29 28	27 26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1	1	0 0	A	R	1	Rs	0	0 1 0	0 0	Rn	Rt
size									opc			

LDEORAB variant

Applies when `A == 1 && R == 0`.

```
LDEORAB <Ws>, <Wt>, [<Xn|SP>]
```

LDEORALB variant

Applies when `A == 1 && R == 1`.

```
LDEORALB <Ws>, <Wt>, [<Xn|SP>]
```

LDEORB variant

Applies when `A == 0 && R == 0`.

```
LDEORB <Ws>, <Wt>, [<Xn|SP>]
```

LDEORLB variant

Applies when `A == 0 && R == 1`.

```
LDEORLB <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Alias conditions

Alias	is preferred when
STEORB, STEORLB	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_EOR, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.121 LDEORH, LDEORAH, LDEORALH, LDEORLH

Atomic exclusive OR on halfword in memory atomically loads a 16-bit halfword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDEORAH` and `LDEORALH` load from memory with acquire semantics.
- `LDEORLH` and `LDEORALH` store to memory with release semantics.
- `LDEORH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is used by the alias STEORH, STEORLH. See *Alias conditions* on page C6-982 for details of when each alias is preferred.

ARMv8.1

31 30	29 28	27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1	1	0	0 0	A	R	1	Rs	0	0 1 0	0 0	Rn	Rt
size										opc			

LDEORAH variant

Applies when `A == 1 && R == 0`.

```
LDEORAH <Ws>, <Wt>, [<Xn|SP>]
```

LDEORALH variant

Applies when `A == 1 && R == 1`.

```
LDEORALH <Ws>, <Wt>, [<Xn|SP>]
```

LDEORH variant

Applies when `A == 0 && R == 0`.

```
LDEORH <Ws>, <Wt>, [<Xn|SP>]
```

LDEORLH variant

Applies when `A == 0 && R == 1`.

```
LDEORLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Alias conditions

Alias	is preferred when
STEORH, STEORLH	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_EOR, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.122 LDEOR, LDEORA, LDEORAL, LDEORL

Atomic exclusive OR on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDEORA` and `LDEORAL` load from memory with acquire semantics.
- `LDEORL` and `LDEORAL` store to memory with release semantics.
- `LDEOR` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is used by the alias STEOR, STEORL. See *Alias conditions* on page C6-984 for details of when each alias is preferred.

ARMv8.1

31 30	29 28	27 26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
1 x	1 1	1	0 0	A	R	1	Rs	0	0 1 0	0 0	Rn	Rt
size									opc			

32-bit LDEOR variant

Applies when `size == 10 && A == 0 && R == 0`.

```
LDEOR <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDEORA variant

Applies when `size == 10 && A == 1 && R == 0`.

```
LDEORA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDEORAL variant

Applies when `size == 10 && A == 1 && R == 1`.

```
LDEORAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDEORL variant

Applies when `size == 10 && A == 0 && R == 1`.

```
LDEORL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit LDEOR variant

Applies when `size == 11 && A == 0 && R == 0`.

```
LDEOR <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDEORA variant

Applies when `size == 11 && A == 1 && R == 0`.

```
LDEORA <Xs>, <Xt>, [<Xn|SP>]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

64-bit LDEORAL variant

Applies when `size == 11 && A == 1 && R == 1`.

```
LDEORAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDEORL variant

Applies when `size == 11 && A == 0 && R == 1`.

```
LDEORL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Alias conditions

Alias	is preferred when
STEOR, STEORL	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Wt>`	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
`<Xs>`	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_EOR, value, ldacctype, stacctype);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.123 LDG

Load Allocation Tag loads an Allocation Tag from a memory address, generates a Logical Address Tag from the Allocation Tag and merges it into the destination register. The address used for the load is calculated from the base register and an immediate signed offset scaled by the Tag granule.

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21	20 – 12	11	10	9 – 5	4 – 0
1	1	0	1	1	0	0	1	0	1	1	imm9	0	0	Xn	Xt

Integer variant

```
LDG <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
integer t = UInt(Xt);
integer n = UInt(Xn);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
```

Assembler symbols

`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
`<simm>`	Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

Operation

```
bits(64) address;
bits(4) tag;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
address = Align(address, TAG_GRANULE);

tag = AArch64.MemTag[address, AccType_NORMAL];
X[t] = AArch64.AddressWithAllocationTag(X[t], tag);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.124 LDGM

Load Tag Multiple reads a naturally aligned block of N Allocation Tags, where the size of N is identified in GMID_EL1.BS, and writes the Allocation Tag read from address A to the destination register at 4*A<7:4>+3:4*A<7:4>. Bits of the destination register not written with an Allocation Tag are set to 0.

This instruction is UNDEFINED at EL0.

This instruction generates an Unchecked access.

If ID_AA64PFR1_EL1.MTE != 0b0010, this instruction is UNDEFINED.

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
1	1	0	1	1	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	Xn		Xt	

Integer variant

```
LDGM <Xt>, [<Xn|SP>]
```

Decode for this encoding

```
integer t = UInt(Xt);
integer n = UInt(Xn);
```

Assembler symbols

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

Operation

```
if PSTATE.EL == EL0 then
    UNDEFINED;

bits(64) data = Zeros(64);
bits(64) address;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

integer size = 4 * (2 ^ (UInt(GMID_EL1.BS)));
address = Align(address, size);
integer count = size >> LOG2_TAG_GRANULE;
integer index = UInt(address<LOG2_TAG_GRANULE+3:LOG2_TAG_GRANULE>);

for i = 0 to count-1
    bits(4) tag = AArch64.MemTag[address, AccType_NORMAL];
    data<(index*4)+3:index*4> = tag;
    address = address + TAG_GRANULE;
    index = index + 1;

X[t] = data;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.125 LDLARB

Load LOAcquire Register Byte loads a byte from memory, zero-extends it and writes it to a register. The instruction also has memory ordering semantics as described in *LoadLOAcquire, StoreLORelease* on page B2-141. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

Note

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

ARMv8.1

31 30	29 28	27 26	25 24	23	22	21	20 16	15	14 10	9 5	4 0
0 0	0 0	1 0	0 0	1	1	0	(1)(1)(1)(1)(1)	0	(1)(1)(1)(1)(1)	Rn	Rt
size					L		Rs	o0	Rt2		

No offset variant

```
LDLARB <Wt>, [<Xn|SP>{,#0}]
```

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = Mem[address, 1, AccType_LIMITEDORDERED];
X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.126 LDLARH

Load LOAcquire Register Halfword loads a halfword from memory, zero-extends it, and writes it to a register. The instruction also has memory ordering semantics as described in *LoadLOAcquire, StoreLORelease* on page B2-141. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

Note

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

ARMv8.1

31 30	29 28	27 26	25 24	23	22	21	20 16	15	14 10	9 5	4 0
0 1	0 0	1 0	0 0	1	1	0	(1)(1)(1)(1)(1)	0	(1)(1)(1)(1)(1)	Rn	Rt
size					L		Rs	o0	Rt2		

No offset variant

```
LDLARH <Wt>, [<Xn|SP>{,#0}]
```

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = Mem[address, 2, AccType_LIMITEDORDERED];
X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.127 LDLAR

Load LOAcquire Register loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The instruction also has memory ordering semantics as described in *LoadLOAcquire, StoreLORelease* on page B2-141. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

Note

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

ARMv8.1

31 30	29 28	27 26	25 24	23	22	21	20 16	15	14 10	9 5	4 0
1 x	0 0	1 0	0 0	1	1	0	(1)(1)(1)(1)(1)	0	(1)(1)(1)(1)(1)	Rn	Rt
size					L		Rs	o0	Rt2		

32-bit variant

Applies when `size == 10`.

```
LDLAR <Wt>, [<Xn|SP>{,#0}]
```

64-bit variant

Applies when `size == 11`.

```
LDLAR <Xt>, [<Xn|SP>{,#0}]
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
boolean tag_checked = n != 31;
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
data = Mem[address, dbytes, AccType_LIMITEDORDERED];
X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.128 LDNP

Load Pair of Registers, with non-temporal hint, calculates an address from a base register value and an immediate offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers.

For information about memory accesses, see *Load/Store addressing modes* on page C1-189. For information about Non-temporal pair instructions, see *Load/Store Non-temporal Pair* on page C3-212.

31 30	29 28 27	26	25 24 23	22	21 15	14 10	9 5	4 0
x 0	1 0 1	0	0 0 0	1	imm7	Rt2	Rn	Rt
opc				L				

32-bit variant

Applies when `opc == 00`.

```
LDNP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

64-bit variant

Applies when `opc == 10`.

```
LDNP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for all variants of this encoding

```
// Empty.
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *LDNP* on page K1-7862.

Assembler symbols

`<Wt1>` Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

`<Wt2>` Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

`<Xt1>` Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

`<Xt2>` Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<imm>` For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc<0> == '1' then UNDEFINED;
integer scale = 2 + UInt(opc<1>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = n != 31;
```

Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data1 = Mem[address, dbytes, AccType_STREAM];
data2 = Mem[address+dbytes, dbytes, AccType_STREAM];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
X[t] = data1;
X[t2] = data2;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.129 LDP

Load Pair of Registers calculates an address from a base register value and an immediate offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

Post-index

31 30	29 28 27	26	25 24 23	22	21 15	14 10	9 5	4 0
x 0	1 0 1	0	0 0 1	1	imm7	Rt2	Rn	Rt
opc				L				

32-bit variant

Applies when `opc == 00`.

```
LDP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>
```

64-bit variant

Applies when `opc == 10`.

```
LDP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
```

Pre-index

31 30	29 28 27	26	25 24 23	22	21 15	14 10	9 5	4 0
x 0	1 0 1	0	0 1 1	1	imm7	Rt2	Rn	Rt
opc				L				

32-bit variant

Applies when `opc == 00`.

```
LDP <Wt1>, <Wt2>, [<Xn|SP>, #<imm>]!
```

64-bit variant

Applies when `opc == 10`.

```
LDP <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
```

Signed offset

31 30	29 28 27	26	25 24 23	22	21 15	14 10	9 5	4 0
x 0	1 0 1	0	0 1 0	1	imm7	Rt2	Rn	Rt
opc				L				

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

32-bit variant

Applies when `opc == 00`.

```
LDP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

64-bit variant

Applies when `opc == 10`.

```
LDP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for all variants of this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *LDP* on page K1-7863.

Assembler symbols

`<Wt1>`	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
`<Wt2>`	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
`<Xt1>`	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt2>`	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<imm>`	For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4. For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4. For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8. For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if L:opc<0> == '01' || opc == '11' then UNDEFINED;
boolean signed = (opc<0> != '0');
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = wback || n != 31;
```

Operation for all encodings

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

data1 = Mem[address, dbytes, AccType_NORMAL];
data2 = Mem[address+dbytes, dbytes, AccType_NORMAL];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
if signed then
    X[t] = SignExtend(data1, 64);
    X[t2] = SignExtend(data2, 64);
else
    X[t] = data1;
    X[t2] = data2;

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.130 LDPSW

Load Pair of Registers Signed Word calculates an address from a base register value and an immediate offset, loads two 32-bit words from memory, sign-extends them, and writes them to two registers. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

Post-index

31 30	29 28 27	26	25 24 23	22	21 … 15	14 … 10	9 … 5	4 … 0
0 1	1 0 1	0	0 0 1	1	imm7	Rt2	Rn	Rt
opc				L				

Post-index variant

```
LDPSW <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
```

Pre-index

31 30	29 28 27	26	25 24 23	22	21 … 15	14 … 10	9 … 5	4 … 0
0 1	1 0 1	0	0 1 1	1	imm7	Rt2	Rn	Rt
opc				L				

Pre-index variant

```
LDPSW <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
```

Signed offset

31 30	29 28 27	26	25 24 23	22	21 … 15	14 … 10	9 … 5	4 … 0
0 1	1 0 1	0	0 1 0	1	imm7	Rt2	Rn	Rt
opc				L				

Signed offset variant

```
LDPSW <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *LDPSW* on page K1-7864.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the post-index and pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4. For the signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
bits(64) offset = LSL(SignExtend(imm7, 64), 2);
boolean tag_checked = wback || n != 31;
```

Operation for all encodings

```
bits(64) address;
bits(32) data1;
bits(32) data2;
boolean rt_unknown = FALSE;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

data1 = Mem[address, 4, AccType_NORMAL];
data2 = Mem[address+4, 4, AccType_NORMAL];
if rt_unknown then
    data1 = bits(32) UNKNOWN;
    data2 = bits(32) UNKNOWN;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
X[t] = SignExtend(data1, 64);
X[t2] = SignExtend(data2, 64);
if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.131 LDR (immediate)

Load Register (immediate) loads a word or doubleword from memory and writes it to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes* on page C1-189. The Unsigned offset variant scales the immediate offset value by the size of the value accessed before adding it to the base register value.

Post-index

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
1	x	1	1	1	0	0	0	0	1	0	imm9	0	1	Rn	Rt
size								opc							

32-bit variant

Applies when `size == 10`.

```
LDR <Wt>, [<Xn|SP>], #<simm>
```

64-bit variant

Applies when `size == 11`.

```
LDR <Xt>, [<Xn|SP>], #<simm>
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
1	x	1	1	1	0	0	0	0	1	0	imm9	1	1	Rn	Rt
size								opc							

32-bit variant

Applies when `size == 10`.

```
LDR <Wt>, [<Xn|SP>, #<simm>]!
```

64-bit variant

Applies when `size == 11`.

```
LDR <Xt>, [<Xn|SP>, #<simm>]!
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Unsigned offset

31 30	29 28 27	26	25 24	23 22	21 … 10	9 … 5	4 … 0
1 x	1 1 1	0	0 1	0 1	imm12	Rn	Rt
size				opc			

32-bit variant

Applies when `size == 10`.

```
LDR <Wt>, [<Xn|SP>{, #<pimm>}]
```

64-bit variant

Applies when `size == 11`.

```
LDR <Xt>, [<Xn|SP>{, #<pimm>}]
```

Decode for all variants of this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *LDR (immediate)* on page K1-7864.

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
`<pimm>`	For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4. For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
boolean tag_checked = wback || n != 31;
```

Operation for all encodings

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(datasize) data;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
boolean wb_unknown = FALSE;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

data = Mem[address, datasize DIV 8, AccType_NORMAL];
X[t] = ZeroExtend(data, regsize);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.132 LDR (literal)

Load Register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

31	30	29	28	27	26	25	24	23 … 5	4 … 0
0	x	0	1	1	0	0	0	imm19	Rt
opc									

32-bit variant

Applies when `opc == 00`.

```
LDR <Wt>, <label>
```

64-bit variant

Applies when `opc == 01`.

```
LDR <Xt>, <label>
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
MemOp memop = MemOp_LOAD;
boolean signed = FALSE;
integer size;
bits(64) offset;

case opc of
    when '00'
        size = 4;
    when '01'
        size = 8;
    when '10'
        size = 4;
        signed = TRUE;
    when '11'
        memop = MemOp_PREFETCH;

offset = SignExtend(imm19:'00', 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
`<label>`	Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

```
bits(64) address = PC[] + offset;
bits(size*8) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(FALSE);

case memop of
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
when MemOp_LOAD
    data = Mem[address, size, AccType_NORMAL];
    if signed then
        X[t] = SignExtend(data, 64);
    else
        X[t] = data;

when MemOp_PREFETCH
    Prefetch(address, t<4:0>);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.133 LDR (register)

Load Register (register) calculates an address from a base register value and an offset register value, loads a word from memory, and writes it to a register. The offset register value can optionally be shifted and extended. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

31 30	29 28 27	26	25 24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
1 x	1 1 1	0	0 0	0 1	1	Rm	option	S	1 0	Rn	Rt
size				opc							

32-bit variant

Applies when `size == 10`.

```
LDR <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

64-bit variant

Applies when `size == 11`.

```
LDR <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for all variants of this encoding

```
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xt>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<Wm>` When `option<0>` is set to `0`, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

`<Xm>` When `option<0>` is set to `1`, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

`<extend>` Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in the "option" field. It can have the following values:

- `UXTW` when `option = 010`
- `LSL` when `option = 011`
- `SXTW` when `option = 110`
- `SXTX` when `option = 111`

`<amount>` For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

- `#0` when `S = 0`
- `#2` when `S = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

`#0` when `S = 0`

`#3` when `S = 1`

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
    SetTagCheckedInstruction(TRUE);

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, datasize DIV 8, AccType_NORMAL];
X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.134 LDRAA, LDRAB

Load Register, with pointer authentication. This instruction authenticates an address from a base register using a modifier of zero and the specified key, adds an immediate offset to the authenticated address, and loads a 64-bit doubleword from memory at this resulting address into a register.

Key A is used for `LDRAA`, and key B is used for `LDRAB`.

If the authentication passes, the PE behaves the same as for an `LDR` instruction. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to the base register, unless the pre-indexed variant of the instruction is used. In this case, the address that is written back to the base register does not include the pointer authentication code.

For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

ARMv8.3

31 30	29 28	27 26	25 24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
1 1	1 1	1	0	0	0	M	S	1	imm9	W	1	Rn	Rt
size													

Key A, offset variant

Applies when `M == 0 && W == 0`.

```
LDRAA <Xt>, [<Xn|SP>{, #<simm>}]
```

Key A, pre-indexed variant

Applies when `M == 0 && W == 1`.

```
LDRAA <Xt>, [<Xn|SP>{, #<simm>}]!
```

Key B, offset variant

Applies when `M == 1 && W == 0`.

```
LDRAB <Xt>, [<Xn|SP>{, #<simm>}]
```

Key B, pre-indexed variant

Applies when `M == 1 && W == 1`.

```
LDRAB <Xt>, [<Xn|SP>{, #<simm>}]!
```

Decode for all variants of this encoding

```
if !HavePACExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
boolean wback = (W == '1');
boolean use_key_a = (M == '0');
bits(10) S10 = S:imm9;
bits(64) offset = LSL(SignExtend(S10, 64), 3);
boolean tag_checked = wback || n != 31;
```

Assembler symbols

`<Xt>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, a multiple of 8 in the range -4096 to 4088, defaulting to 0 and encoded in the "S:imm9" field as <simm>/8.

Operation

```
bits(64) address;
bits(64) data;
boolean wb_unknown = FALSE;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    address = SP[];
else
    address = X[n];

if use_key_a then
    address = AuthDA(address, X[31], TRUE);
else
    address = AuthDB(address, X[31], TRUE);

if n == 31 then
    CheckSPAlignment();

address = address + offset;
data = Mem[address, 8, AccType_NORMAL];
X[t] = data;

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.135 LDRB (immediate)

Load Register Byte (immediate) loads a byte from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

Post-index

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 0	1 1 1	0	0 0	0 1	0	imm9	0 1	Rn	Rt
size				opc					

Post-index variant

```
LDRB <Wt>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 0	1 1 1	0	0 0	0 1	0	imm9	1 1	Rn	Rt
size				opc					

Pre-index variant

```
LDRB <Wt>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31 30	29 28 27	26	25 24	23 22	21 … 10	9 … 5	4 … 0
0 0	1 1 1	0	0 1	0 1	imm12	Rn	Rt
size				opc			

Unsigned offset variant

```
LDRB <Wt>, [<Xn|SP>{, #<pimm>}]
```

Decode for this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *LDRB (immediate)* on page K1-7864.

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
`<pimm>`	Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = wback || n != 31;
```

Operation for all encodings

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;

boolean wb_unknown = FALSE;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

data = Mem[address, 1, AccType_NORMAL];
X[t] = ZeroExtend(data, 32);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.136 LDRB (register)

Load Register Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

31 30	29 28 27	26	25 24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	0 1	1	Rm	option	S	1 0	Rn	Rt
size				opc							

Extended register variant

Applies when `option != 011`.

```
LDRB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

Shifted register variant

Applies when `option == 011`.

```
LDRB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

Decode for all variants of this encoding

```
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
```

Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<Wm>` When `option<0>` is set to `0`, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

`<Xm>` When `option<0>` is set to `1`, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

`<extend>` Is the index extend specifier, encoded in the "option" field. It can have the following values:

- `UXTW` when `option = 010`
- `SXTW` when `option = 110`
- `SXTX` when `option = 111`

`<amount>` Is the index shift amount, it must be `#0`, encoded in "S" as `0` if omitted, or as `1` if present.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, 0);
if HaveMTEExt() then
    SetTagCheckedInstruction(TRUE);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(64) address;
bits(8) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, 1, AccType_NORMAL];
X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.137 LDRH (immediate)

Load Register Halfword (immediate) loads a halfword from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

Post-index

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 1	1 1 1	0	0 0	0 1	0	imm9	0 1	Rn	Rt
size				opc					

Post-index variant

```
LDRH <Wt>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 1	1 1 1	0	0 0	0 1	0	imm9	1 1	Rn	Rt
size				opc					

Pre-index variant

```
LDRH <Wt>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31 30	29 28 27	26	25 24	23 22	21 … 10	9 … 5	4 … 0
0 1	1 1 1	0	0 1	0 1	imm12	Rn	Rt
size				opc			

Unsigned offset variant

```
LDRH <Wt>, [<Xn|SP>{, #<pimm>}]
```

Decode for this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *LDRH (immediate)* on page K1-7865.

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
`<pimm>`	Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = wback || n != 31;
```

Operation for all encodings

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;

boolean wb_unknown = FALSE;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

data = Mem[address, 2, AccType_NORMAL];
X[t] = ZeroExtend(data, 32);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.138 LDRH (register)

Load Register Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

31 30	29 28 27	26	25 24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	0 1	1	Rm	option	S	1 0	Rn	Rt
size				opc							

32-bit variant

```
LDRH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for this encoding

```
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 1 else 0;
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<Wm>` When `option<0>` is set to `0`, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

`<Xm>` When `option<0>` is set to `1`, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

`<extend>` Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in the "option" field. It can have the following values:

- `UXTW` when `option = 010`
- `LSL` when `option = 011`
- `SXTW` when `option = 110`
- `SXTX` when `option = 111`

`<amount>` Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

- `#0` when `S = 0`
- `#1` when `S = 1`

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
    SetTagCheckedInstruction(TRUE);

bits(64) address;
bits(16) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, 2, AccType_NORMAL];
X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.139 LDRSB (immediate)

Load Register Signed Byte (immediate) loads a byte from memory, sign-extends it to either 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

Post-index

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 0	1 1 1	0	0 0	1 x	0	imm9	0 1	Rn	Rt
size				opc					

32-bit variant

Applies when `opc == 11`.

```
LDRSB <Wt>, [<Xn|SP>], #<simm>
```

64-bit variant

Applies when `opc == 10`.

```
LDRSB <Xt>, [<Xn|SP>], #<simm>
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 0	1 1 1	0	0 0	1 x	0	imm9	1 1	Rn	Rt
size				opc					

32-bit variant

Applies when `opc == 11`.

```
LDRSB <Wt>, [<Xn|SP>, #<simm>]!
```

64-bit variant

Applies when `opc == 10`.

```
LDRSB <Xt>, [<Xn|SP>, #<simm>]!
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31 30	29 28 27	26	25 24	23 22	21 … 10	9 … 5	4 … 0
0 0	1 1 1	0	0 1	1 x	imm12	Rn	Rt
size				opc			

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

32-bit variant

Applies when `opc == 11`.

```
LDRSB <Wt>, [<Xn|SP>{, #<pimm>}]
```

64-bit variant

Applies when `opc == 10`.

```
LDRSB <Xt>, [<Xn|SP>{, #<pimm>}]
```

Decode for all variants of this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *LDRSB (immediate)* on page K1-7865.

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
`<pimm>`	Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);
```

Operation for all encodings

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(8) UNKNOWN;
        else
            data = X[t];
        Mem[address, 1, AccType_NORMAL] = data;

    when MemOp_LOAD
        data = Mem[address, 1, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.140 LDRSB (register)

Load Register Signed Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

31 30	29 28 27	26	25 24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	1 x	1	Rm	option	S	1 0	Rn	Rt
size				opc							

32-bit with extended register offset variant

Applies when `opc == 11 && option != 011`.

```
LDRSB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

32-bit with shifted register offset variant

Applies when `opc == 11 && option == 011`.

```
LDRSB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

64-bit with extended register offset variant

Applies when `opc == 10 && option != 011`.

```
LDRSB <Xt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

64-bit with shifted register offset variant

Applies when `opc == 10 && option == 011`.

```
LDRSB <Xt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

Decode for all variants of this encoding

```
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
```

Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xt>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<Wm>` When `option<0>` is set to `0`, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

`<Xm>` When `option<0>` is set to `1`, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

`<extend>` Is the index extend specifier, encoded in the "option" field. It can have the following values:

- `UXTW` when `option = 010`
- `SXTW` when `option = 110`
- `SXTX` when `option = 111`

`<amount>` Is the index shift amount, it must be `#0`, encoded in "S" as `0` if omitted, or as `1` if present.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, 0);
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 1, AccType_NORMAL] = data;

    when MemOp_LOAD
        data = Mem[address, 1, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.141 LDRSH (immediate)

Load Register Signed Halfword (immediate) loads a halfword from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

Post-index

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 1	1 1 1	0	0 0	1 x	0	imm9	0 1	Rn	Rt
size				opc					

32-bit variant

Applies when `opc == 11`.

```
LDRSH <Wt>, [<Xn|SP>], #<simm>
```

64-bit variant

Applies when `opc == 10`.

```
LDRSH <Xt>, [<Xn|SP>], #<simm>
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 1	1 1 1	0	0 0	1 x	0	imm9	1 1	Rn	Rt
size				opc					

32-bit variant

Applies when `opc == 11`.

```
LDRSH <Wt>, [<Xn|SP>, #<simm>]!
```

64-bit variant

Applies when `opc == 10`.

```
LDRSH <Xt>, [<Xn|SP>, #<simm>]!
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31 30	29 28 27	26	25 24	23 22	21 … 10	9 … 5	4 … 0
0 1	1 1 1	0	0 1	1 x	imm12	Rn	Rt
size				opc			

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

32-bit variant

Applies when `opc == 11`.

```
LDRSH <Wt>, [<Xn|SP>{, #<pimm>}]
```

64-bit variant

Applies when `opc == 10`.

```
LDRSH <Xt>, [<Xn|SP>{, #<pimm>}]
```

Decode for all variants of this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *LDRSH (immediate)* on page K1-7866.

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
`<pimm>`	Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);
```

Operation for all encodings

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(16) UNKNOWN;
        else
            data = X[t];
        Mem[address, 2, AccType_NORMAL] = data;

    when MemOp_LOAD
        data = Mem[address, 2, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.142 LDRSH (register)

Load Register Signed Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, sign-extends it, and writes it to a register. For information about memory accesses see *Load/Store addressing modes* on page C1-189.

31 30	29 28 27	26	25 24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	1 x	1	Rm	option	S	1 0	Rn	Rt
size				opc							

32-bit variant

Applies when `opc == 11`.

```
LDRSH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

64-bit variant

Applies when `opc == 10`.

```
LDRSH <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for all variants of this encoding

```
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 1 else 0;
```

Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xt>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<Wm>` When `option<0>` is set to `0`, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

`<Xm>` When `option<0>` is set to `1`, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

`<extend>` Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in the "option" field. It can have the following values:

- `UXTW` when `option = 010`
- `LSL` when `option = 011`
- `SXTW` when `option = 110`
- `SXTX` when `option = 111`

`<amount>` Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

- `#0` when `S = 0`
- `#1` when `S = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 2, AccType_NORMAL] = data;

    when MemOp_LOAD
        data = Mem[address, 2, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.143 LDRSW (immediate)

Load Register Signed Word (immediate) loads a word from memory, sign-extends it to 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

Post-index

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
1 0	1 1 1	0	0 0	1 0	0	imm9	0 1	Rn	Rt
size				opc					

Post-index variant

```
LDRSW <Xt>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
1 0	1 1 1	0	0 0	1 0	0	imm9	1 1	Rn	Rt
size				opc					

Pre-index variant

```
LDRSW <Xt>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31 30	29 28 27	26	25 24	23 22	21 … 10	9 … 5	4 … 0
1 0	1 1 1	0	0 1	1 0	imm12	Rn	Rt
size				opc			

Unsigned offset variant

```
LDRSW <Xt>, [<Xn|SP>{, #<pimm>}]
```

Decode for this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 2);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *LDRSW (immediate)* on page K1-7866.

Assembler symbols

`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
`<pimm>`	Is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = wback || n != 31;
```

Operation for all encodings

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(32) data;

boolean wb_unknown = FALSE;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

data = Mem[address, 4, AccType_NORMAL];
X[t] = SignExtend(data, 64);
if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.144 LDRSW (literal)

Load Register Signed Word (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

31 30	29 28 27	26	25 24	23 … 5	4 … 0
1 0	0 1 1	0	0 0	imm19	Rt
opc					

Literal variant

```
LDRSW <Xt>, <label>
```

Decode for this encoding

```
integer t = UInt(Rt);
bits(64) offset;

offset = SignExtend(imm19:'00', 64);
```

Assembler symbols

`<Xt>` Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

`<label>` Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

```
bits(64) address = PC[] + offset;
bits(32) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(FALSE);

data = Mem[address, 4, AccType_NORMAL];
X[t] = SignExtend(data, 64);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.145 LDRSW (register)

Load Register Signed Word (register) calculates an address from a base register value and an offset register value, loads a word from memory, sign-extends it to form a 64-bit value, and writes it to a register. The offset register value can be shifted left by 0 or 2 bits. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

31 30	29 28 27	26	25 24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
1 0	1 1 1	0	0 0	1 0	1	Rm	option	S	1 0	Rn	Rt
size				opc							

64-bit variant

```
LDRSW <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for this encoding

```
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 2 else 0;
```

Assembler symbols

`<Xt>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<Wm>` When `option<0>` is set to `0`, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

`<Xm>` When `option<0>` is set to `1`, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

`<extend>` Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in the "option" field. It can have the following values:

- `UXTW` when `option = 010`
- `LSL` when `option = 011`
- `SXTW` when `option = 110`
- `SXTX` when `option = 111`

`<amount>` Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

- `#0` when `S = 0`
- `#2` when `S = 1`

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
    SetTagCheckedInstruction(TRUE);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(64) address;
bits(32) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, 4, AccType_NORMAL];
X[t] = SignExtend(data, 64);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.146 LDSETB, LDSETAB, LDSETALB, LDSETLB

Atomic bit set on byte in memory atomically loads an 8-bit byte from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDSETAB` and `LDSETALB` load from memory with acquire semantics.
- `LDSETLB` and `LDSETALB` store to memory with release semantics.
- `LDSETB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is used by the alias STSETB, STSETLB. See *Alias conditions* on page C6-1036 for details of when each alias is preferred.

ARMv8.1

31 30	29 28	27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1	1	0	0 0	A	R	1	Rs	0	0 1 1	0 0	Rn	Rt
size										opc			

LDSETAB variant

Applies when `A == 1 && R == 0`.

```
LDSETAB <Ws>, <Wt>, [<Xn|SP>]
```

LDSETALB variant

Applies when `A == 1 && R == 1`.

```
LDSETALB <Ws>, <Wt>, [<Xn|SP>]
```

LDSETB variant

Applies when `A == 0 && R == 0`.

```
LDSETB <Ws>, <Wt>, [<Xn|SP>]
```

LDSETLB variant

Applies when `A == 0 && R == 1`.

```
LDSETLB <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Alias conditions

Alias	is preferred when
STSETB, STSETLB	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_ORR, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.147 LDSETH, LDSETAH, LDSETALH, LDSETLH

Atomic bit set on halfword in memory atomically loads a 16-bit halfword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDSETAH` and `LDSETALH` load from memory with acquire semantics.
- `LDSETLH` and `LDSETALH` store to memory with release semantics.
- `LDSETH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is used by the alias STSETH, STSETLH. See *Alias conditions* on page C6-1038 for details of when each alias is preferred.

ARMv8.1

31 30	29 28	27	26	25	24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1	1	0	0	0	A	R	1	Rs	0	0 1 1	0 0	Rn	Rt
size										opc				

LDSETAH variant

Applies when `A == 1 && R == 0`.

```
LDSETAH <Ws>, <Wt>, [<Xn|SP>]
```

LDSETALH variant

Applies when `A == 1 && R == 1`.

```
LDSETALH <Ws>, <Wt>, [<Xn|SP>]
```

LDSETH variant

Applies when `A == 0 && R == 0`.

```
LDSETH <Ws>, <Wt>, [<Xn|SP>]
```

LDSETLH variant

Applies when `A == 0 && R == 1`.

```
LDSETLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Alias conditions

Alias	is preferred when
STSETH, STSETLH	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_ORR, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.148 LDSET, LDSETA, LDSETAL, LDSETL

Atomic bit set on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDSETA` and `LDSETAL` load from memory with acquire semantics.
- `LDSETL` and `LDSETAL` store to memory with release semantics.
- `LDSET` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is used by the alias STSET, STSETL. See *Alias conditions* on page C6-1040 for details of when each alias is preferred.

ARMv8.1

31 30	29 28 27	26 25 24	23	22	21	20 … 16	15	14 … 12	11 10	9 … 5	4 … 0
1 x	1 1 1	0 0 0	A	R	1	Rs	0	0 1 1	0 0	Rn	Rt
size								opc			

32-bit LDSET variant

Applies when `size == 10 && A == 0 && R == 0`.

```
LDSET <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSETA variant

Applies when `size == 10 && A == 1 && R == 0`.

```
LDSETA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSETAL variant

Applies when `size == 10 && A == 1 && R == 1`.

```
LDSETAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSETL variant

Applies when `size == 10 && A == 0 && R == 1`.

```
LDSETL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit LDSET variant

Applies when `size == 11 && A == 0 && R == 0`.

```
LDSET <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDSETA variant

Applies when `size == 11 && A == 1 && R == 0`.

```
LDSETA <Xs>, <Xt>, [<Xn|SP>]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

64-bit LDSETAL variant

Applies when `size == 11 && A == 1 && R == 1`.

```
LDSETAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDSETL variant

Applies when `size == 11 && A == 0 && R == 1`.

```
LDSETL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Alias conditions

Alias	is preferred when
STSET, STSETL	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Wt>`	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
`<Xs>`	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_ORR, value, ldacctype, stacctype);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.149 LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB

Atomic signed maximum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDSMAXAB` and `LDSMAXALB` load from memory with acquire semantics.
- `LDSMAXLB` and `LDSMAXALB` store to memory with release semantics.
- `LDSMAXB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is used by the alias STSMAXB, STSMAXLB. See *Alias conditions* on page C6-1043 for details of when each alias is preferred.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	A	R	1	Rs	0	1 0 0	0 0	Rn	Rt
size									opc			

LDSMAXAB variant

Applies when `A == 1 && R == 0`.

```
LDSMAXAB <Ws>, <Wt>, [<Xn|SP>]
```

LDSMAXALB variant

Applies when `A == 1 && R == 1`.

```
LDSMAXALB <Ws>, <Wt>, [<Xn|SP>]
```

LDSMAXB variant

Applies when `A == 0 && R == 0`.

```
LDSMAXB <Ws>, <Wt>, [<Xn|SP>]
```

LDSMAXLB variant

Applies when `A == 0 && R == 1`.

```
LDSMAXLB <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Alias conditions

Alias	is preferred when
STSMAXB, STSMAXLB	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_SMAX, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.150 LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH

Atomic signed maximum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDSMAXAH` and `LDSMAXALH` load from memory with acquire semantics.
- `LDSMAXLH` and `LDSMAXALH` store to memory with release semantics.
- `LDSMAXH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is used by the alias STSMAXH, STSMAXLH. See *Alias conditions* on page C6-1045 for details of when each alias is preferred.

ARMv8.1

31 30	29 28	27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1	1	0	0 0	A	R	1	Rs	0	1 0 0	0 0	Rn	Rt
size										opc			

LDSMAXAH variant

Applies when `A == 1 && R == 0`.

```
LDSMAXAH <Ws>, <Wt>, [<Xn|SP>]
```

LDSMAXALH variant

Applies when `A == 1 && R == 1`.

```
LDSMAXALH <Ws>, <Wt>, [<Xn|SP>]
```

LDSMAXH variant

Applies when `A == 0 && R == 0`.

```
LDSMAXH <Ws>, <Wt>, [<Xn|SP>]
```

LDSMAXLH variant

Applies when `A == 0 && R == 1`.

```
LDSMAXLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Alias conditions

Alias	is preferred when
STSMAXH, STSMAXLH	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_SMAX, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.151 LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL

Atomic signed maximum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDSMAXA` and `LDSMAXAL` load from memory with acquire semantics.
- `LDSMAXL` and `LDSMAXAL` store to memory with release semantics.
- `LDSMAX` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is used by the alias STSMAX, STSMAXL. See *Alias conditions* on page C6-1047 for details of when each alias is preferred.

ARMv8.1

31 30	29 28	27 26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
1 x	1 1	1	0 0	A	R	1	Rs	0	1 0 0	0 0	Rn	Rt
size									opc			

32-bit LDSMAX variant

Applies when `size == 10 && A == 0 && R == 0`.

```
LDSMAX <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSMAXA variant

Applies when `size == 10 && A == 1 && R == 0`.

```
LDSMAXA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSMAXAL variant

Applies when `size == 10 && A == 1 && R == 1`.

```
LDSMAXAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSMAXL variant

Applies when `size == 10 && A == 0 && R == 1`.

```
LDSMAXL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit LDSMAX variant

Applies when `size == 11 && A == 0 && R == 0`.

```
LDSMAX <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDSMAXA variant

Applies when `size == 11 && A == 1 && R == 0`.

```
LDSMAXA <Xs>, <Xt>, [<Xn|SP>]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

64-bit LDSMAXAL variant

Applies when `size == 11 && A == 1 && R == 1`.

```
LDSMAXAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDSMAXL variant

Applies when `size == 11 && A == 0 && R == 1`.

```
LDSMAXL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Alias conditions

Alias	is preferred when
STSMAX, STSMAXL	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Wt>`	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
`<Xs>`	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_SMAX, value, ldacctype, stacctype);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.152 LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB

Atomic signed minimum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDSMINAB` and `LDSMINALB` load from memory with acquire semantics.
- `LDSMINLB` and `LDSMINALB` store to memory with release semantics.
- `LDSMINB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is used by the alias STSMINB, STSMINLB. See *Alias conditions* on page C6-1050 for details of when each alias is preferred.

ARMv8.1

31 30	29 28	27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1	1	0	0 0	A	R	1	Rs	0	1 0 1	0 0	Rn	Rt
size										opc			

LDSMINAB variant

Applies when `A == 1 && R == 0`.

```
LDSMINAB <Ws>, <Wt>, [<Xn|SP>]
```

LDSMINALB variant

Applies when `A == 1 && R == 1`.

```
LDSMINALB <Ws>, <Wt>, [<Xn|SP>]
```

LDSMINB variant

Applies when `A == 0 && R == 0`.

```
LDSMINB <Ws>, <Wt>, [<Xn|SP>]
```

LDSMINLB variant

Applies when `A == 0 && R == 1`.

```
LDSMINLB <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Alias conditions

Alias	is preferred when
STSMINB, STSMINLB	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_SMIN, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.153 LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH

Atomic signed minimum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDSMINAH` and `LDSMINALH` load from memory with acquire semantics.
- `LDSMINLH` and `LDSMINALH` store to memory with release semantics.
- `LDSMINH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is used by the alias STSMINH, STSMINLH. See *Alias conditions* on page C6-1052 for details of when each alias is preferred.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	A	R	1	Rs	0	1 0 1	0 0	Rn	Rt
size								opc				

LDSMINAH variant

Applies when `A == 1 && R == 0`.

```
LDSMINAH <Ws>, <Wt>, [<Xn|SP>]
```

LDSMINALH variant

Applies when `A == 1 && R == 1`.

```
LDSMINALH <Ws>, <Wt>, [<Xn|SP>]
```

LDSMINH variant

Applies when `A == 0 && R == 0`.

```
LDSMINH <Ws>, <Wt>, [<Xn|SP>]
```

LDSMINLH variant

Applies when `A == 0 && R == 1`.

```
LDSMINLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Alias conditions

Alias	is preferred when
STSMINH, STSMINLH	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_SMIN, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.154 LDSMIN, LDSMINA, LDSMINAL, LDSMINL

Atomic signed minimum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDSMINA` and `LDSMINAL` load from memory with acquire semantics.
- `LDSMINL` and `LDSMINAL` store to memory with release semantics.
- `LDSMIN` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is used by the alias STSMIN, STSMINL. See *Alias conditions* on page C6-1054 for details of when each alias is preferred.

ARMv8.1

31 30	29 28	27 26	25 24	23	22	21	20 … 16	15	14 … 12	11 10	9 … 5	4 … 0
1 x	1 1	1	0 0	A	R	1	Rs	0	1 0 1	0 0	Rn	Rt
size									opc			

32-bit LDSMIN variant

Applies when `size == 10 && A == 0 && R == 0`.

```
LDSMIN <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSMINA variant

Applies when `size == 10 && A == 1 && R == 0`.

```
LDSMINA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSMINAL variant

Applies when `size == 10 && A == 1 && R == 1`.

```
LDSMINAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSMINL variant

Applies when `size == 10 && A == 0 && R == 1`.

```
LDSMINL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit LDSMIN variant

Applies when `size == 11 && A == 0 && R == 0`.

```
LDSMIN <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDSMINA variant

Applies when `size == 11 && A == 1 && R == 0`.

```
LDSMINA <Xs>, <Xt>, [<Xn|SP>]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

64-bit LDSMINAL variant

Applies when `size == 11 && A == 1 && R == 1`.

```
LDSMINAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDSMINL variant

Applies when `size == 11 && A == 0 && R == 1`.

```
LDSMINL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Alias conditions

Alias	is preferred when
STSMIN, STSMINL	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Wt>`	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
`<Xs>`	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_SMIN, value, ldacctype, stacctype);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.155 LDTR

Load Register (unprivileged) loads a word or doubleword from memory, and writes it to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
1 x	1 1 1	0	0 0	0 1	0	imm9	1 0	Rn	Rt
size				opc					

32-bit variant

Applies when `size == 10`.

```
LDTR <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit variant

Applies when `size == 11`.

```
LDTR <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
boolean tag_checked = n != 31;
```

Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, datasize DIV 8, acctype];
X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.156 LDTRB

Load Register Byte (unprivileged) loads a byte from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 0	1 1 1	0	0 0	0 1	0	imm9	1 0	Rn	Rt
size				opc					

Unscaled offset variant

```
LDTRB <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

boolean tag_checked = n != 31;
```

Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, 1, acctype];
X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.157 LDTRH

Load Register Halfword (unprivileged) loads a halfword from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 1	1 1 1	0	0 0	0 1	0	imm9	1 0	Rn	Rt
size				opc					

Unscaled offset variant

```
LDTRH <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

boolean tag_checked = n != 31;
```

Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, 2, acctype];
X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.158 LDTRSB

Load Register Signed Byte (unprivileged) loads a byte from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 0	1 1 1	0	0 0	1 x	0	imm9	1 0	Rn	Rt
size				opc					

32-bit variant

Applies when `opc == 11`.

```
LDTRSB <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit variant

Applies when `opc == 10`.

```
LDTRSB <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

MemOp memop;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);
```

Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 1, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, 1, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.159 LDTRSH

Load Register Signed Halfword (unprivileged) loads a halfword from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

31 30	29 28	27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 1	1 1	1	0	0 0	1 x	0	imm9	1 0	Rn	Rt
size					opc					

32-bit variant

Applies when `opc == 11`.

```
LDTRSH <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit variant

Applies when `opc == 10`.

```
LDTRSH <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);
```

Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 2, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, 2, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.160 LDTRSW

Load Register Signed Word (unprivileged) loads a word from memory, sign-extends it to 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

31 30	29 28	27 26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
1 0	1 1	1 0	0 0	1 0	0	imm9	1 0	Rn	Rt
size				opc					

Unscaled offset variant

```
LDTRSW <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

boolean tag_checked = n != 31;
```

Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(32) data;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, 4, acctype];
X[t] = SignExtend(data, 64);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.161 LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB

Atomic unsigned maximum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDUMAXAB` and `LDUMAXALB` load from memory with acquire semantics.
- `LDUMAXLB` and `LDUMAXALB` store to memory with release semantics.
- `LDUMAXB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is used by the alias STUMAXB, STUMAXLB. See *Alias conditions* on page C6-1069 for details of when each alias is preferred.

ARMv8.1

31 30	29 28	27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1	1	0	0 0	A	R	1	Rs	0	1 1 0	0 0	Rn	Rt
size										opc			

LDUMAXAB variant

Applies when `A == 1 && R == 0`.

```
LDUMAXAB <Ws>, <Wt>, [<Xn|SP>]
```

LDUMAXALB variant

Applies when `A == 1 && R == 1`.

```
LDUMAXALB <Ws>, <Wt>, [<Xn|SP>]
```

LDUMAXB variant

Applies when `A == 0 && R == 0`.

```
LDUMAXB <Ws>, <Wt>, [<Xn|SP>]
```

LDUMAXLB variant

Applies when `A == 0 && R == 1`.

```
LDUMAXLB <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Alias conditions

Alias	is preferred when
STUMAXB, STUMAXLB	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_UMAX, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.162 LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH

Atomic unsigned maximum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDUMAXAH` and `LDUMAXALH` load from memory with acquire semantics.
- `LDUMAXLH` and `LDUMAXALH` store to memory with release semantics.
- `LDUMAXH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is used by the alias STUMAXH, STUMAXLH. See *Alias conditions* on page C6-1071 for details of when each alias is preferred.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	A	R	1	Rs	0	1 1 0	0 0	Rn	Rt
size								opc				

LDUMAXAH variant

Applies when `A == 1 && R == 0`.

```
LDUMAXAH <Ws>, <Wt>, [<Xn|SP>]
```

LDUMAXALH variant

Applies when `A == 1 && R == 1`.

```
LDUMAXALH <Ws>, <Wt>, [<Xn|SP>]
```

LDUMAXH variant

Applies when `A == 0 && R == 0`.

```
LDUMAXH <Ws>, <Wt>, [<Xn|SP>]
```

LDUMAXLH variant

Applies when `A == 0 && R == 1`.

```
LDUMAXLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Alias conditions

Alias	is preferred when
STUMAXH, STUMAXLH	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_UMAX, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.163 LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL

Atomic unsigned maximum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDUMAXA` and `LDUMAXAL` load from memory with acquire semantics.
- `LDUMAXL` and `LDUMAXAL` store to memory with release semantics.
- `LDUMAX` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is used by the alias STUMAX, STUMAXL. See *Alias conditions* on page C6-1073 for details of when each alias is preferred.

ARMv8.1

31 30	29 28 27	26 25 24	23	22	21	20 16	15	14 13 12	11 10	9 5	4 0
1 x	1 1 1	0	0 0	A	R	1	Rs	0	1 1 0	0 0	Rn	Rt
size								opc				

32-bit LDUMAX variant

Applies when `size == 10 && A == 0 && R == 0`.

```
LDUMAX <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDUMAXA variant

Applies when `size == 10 && A == 1 && R == 0`.

```
LDUMAXA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDUMAXAL variant

Applies when `size == 10 && A == 1 && R == 1`.

```
LDUMAXAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDUMAXL variant

Applies when `size == 10 && A == 0 && R == 1`.

```
LDUMAXL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit LDUMAX variant

Applies when `size == 11 && A == 0 && R == 0`.

```
LDUMAX <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDUMAXA variant

Applies when `size == 11 && A == 1 && R == 0`.

```
LDUMAXA <Xs>, <Xt>, [<Xn|SP>]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

64-bit LDUMAXAL variant

Applies when `size == 11 && A == 1 && R == 1`.

```
LDUMAXAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDUMAXL variant

Applies when `size == 11 && A == 0 && R == 1`.

```
LDUMAXL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Alias conditions

Alias	is preferred when
STUMAX, STUMAXL	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Wt>`	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
`<Xs>`	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_UMAX, value, ldacctype, stacctype);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.164 LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB

Atomic unsigned minimum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDUMINAB` and `LDUMINALB` load from memory with acquire semantics.
- `LDUMINLB` and `LDUMINALB` store to memory with release semantics.
- `LDUMINB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is used by the alias STUMINB, STUMINLB. See *Alias conditions* on page C6-1076 for details of when each alias is preferred.

ARMv8.1

31 30	29 28	27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1	1	0	0 0	A	R	1	Rs	0	1 1 1	0 0	Rn	Rt
size										opc			

LDUMINAB variant

Applies when `A == 1 && R == 0`.

```
LDUMINAB <Ws>, <Wt>, [<Xn|SP>]
```

LDUMINALB variant

Applies when `A == 1 && R == 1`.

```
LDUMINALB <Ws>, <Wt>, [<Xn|SP>]
```

LDUMINB variant

Applies when `A == 0 && R == 0`.

```
LDUMINB <Ws>, <Wt>, [<Xn|SP>]
```

LDUMINLB variant

Applies when `A == 0 && R == 1`.

```
LDUMINLB <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Alias conditions

Alias	is preferred when
STUMINB, STUMINLB	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_UMIN, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.165 LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH

Atomic unsigned minimum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDUMINAH` and `LDUMINALH` load from memory with acquire semantics.
- `LDUMINLH` and `LDUMINALH` store to memory with release semantics.
- `LDUMINH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is used by the alias STUMINH, STUMINLH. See *Alias conditions* on page C6-1078 for details of when each alias is preferred.

ARMv8.1

31 30	29 28	27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1	1	0	0 0	A	R	1	Rs	0	1 1 1	0 0	Rn	Rt
size										opc			

LDUMINAH variant

Applies when `A == 1 && R == 0`.

```
LDUMINAH <Ws>, <Wt>, [<Xn|SP>]
```

LDUMINALH variant

Applies when `A == 1 && R == 1`.

```
LDUMINALH <Ws>, <Wt>, [<Xn|SP>]
```

LDUMINH variant

Applies when `A == 0 && R == 0`.

```
LDUMINH <Ws>, <Wt>, [<Xn|SP>]
```

LDUMINLH variant

Applies when `A == 0 && R == 1`.

```
LDUMINLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Alias conditions

Alias	is preferred when
STUMINH, STUMINLH	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_UMIN, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.166 LDUMIN, LDUMINA, LDUMINAL, LDUMINL

Atomic unsigned minimum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDUMINA` and `LDUMINAL` load from memory with acquire semantics.
- `LDUMINL` and `LDUMINAL` store to memory with release semantics.
- `LDUMIN` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is used by the alias STUMIN, STUMINL. See *Alias conditions* on page C6-1080 for details of when each alias is preferred.

ARMv8.1

31 30	29 28	27	26	25	24	23	22	21	20 … 16	15	14 … 12	11	10	9 … 5	4 … 0
1 x	1 1	1	0	0	0	A	R	1	Rs	0	1 1 1	0	0	Rn	Rt
size											opc				

32-bit LDUMIN variant

Applies when `size == 10 && A == 0 && R == 0`.

```
LDUMIN <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDUMINA variant

Applies when `size == 10 && A == 1 && R == 0`.

```
LDUMINA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDUMINAL variant

Applies when `size == 10 && A == 1 && R == 1`.

```
LDUMINAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDUMINL variant

Applies when `size == 10 && A == 0 && R == 1`.

```
LDUMINL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit LDUMIN variant

Applies when `size == 11 && A == 0 && R == 0`.

```
LDUMIN <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDUMINA variant

Applies when `size == 11 && A == 1 && R == 0`.

```
LDUMINA <Xs>, <Xt>, [<Xn|SP>]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

64-bit LDUMINAL variant

Applies when `size == 11 && A == 1 && R == 1`.

```
LDUMINAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDUMINL variant

Applies when `size == 11 && A == 0 && R == 1`.

```
LDUMINL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Alias conditions

Alias	is preferred when
STUMIN, STUMINL	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Wt>`	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
`<Xs>`	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_UMIN, value, ldacctype, stacctype);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.167 LDUR

Load Register (unscaled) calculates an address from a base register and an immediate offset, loads a 32-bit word or 64-bit doubleword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
1	x	1	1	1	0	0	0	0	1	0	imm9	0	0	Rn	Rt
size								opc							

32-bit variant

Applies when `size == 10`.

```
LDUR <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit variant

Applies when `size == 11`.

```
LDUR <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
boolean tag_checked = n != 31;
```

Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    address = X[n];

address = address + offset;

data = Mem[address, datasize DIV 8, AccType_NORMAL];
X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.168 LDURB

Load Register Byte (unscaled) calculates an address from a base register and an immediate offset, loads a byte from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 0	1 1 1	0	0 0	0 1	0	imm9	0 0	Rn	Rt
size				opc					

Unscaled offset variant

```
LDURB <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, 1, AccType_NORMAL];
X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.169 LDURH

Load Register Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a halfword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 1	1 1 1	0	0 0	0 1	0	imm9	0 0	Rn	Rt
size				opc					

Unscaled offset variant

```
LDURH <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, 2, AccType_NORMAL];
X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.170 LDURSB

Load Register Signed Byte (unscaled) calculates an address from a base register and an immediate offset, loads a signed byte from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 0	1 1 1	0	0 0	1 x	0	imm9	0 0	Rn	Rt
size				opc					

32-bit variant

Applies when `opc == 11`.

```
LDURSB <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit variant

Applies when `opc == 10`.

```
LDURSB <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 1, AccType_NORMAL] = data;

    when MemOp_LOAD
        data = Mem[address, 1, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.171 LDURSH

Load Register Signed Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a signed halfword from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 1	1 1 1	0	0 0	1 x	0	imm9	0 0	Rn	Rt
size				opc					

32-bit variant

Applies when `opc == 11`.

```
LDURSH <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit variant

Applies when `opc == 10`.

```
LDURSH <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 2, AccType_NORMAL] = data;

    when MemOp_LOAD
        data = Mem[address, 2, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.172 LDURSW

Load Register Signed Word (unscaled) calculates an address from a base register and an immediate offset, loads a signed word from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
1 0	1 1 1	0	0 0	1 0	0	imm9	0 0	Rn	Rt
size				opc					

Unscaled offset variant

```
LDURSW <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(32) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, 4, AccType_NORMAL];
X[t] = SignExtend(data, 64);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.173 LDXP

Load Exclusive Pair of Registers derives an address from a base register value, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and is single-copy atomic for each doubleword at doubleword granularity. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores* on page B2-167. For information about memory accesses see *Load/Store addressing modes* on page C1-189.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	10	9	5	4	0
1	sz	0	0	1	0	0	0	0	1	1	(1)	(1)	(1)	(1)	(1)	0	Rt2	Rn	Rt
									L			Rs				o0			

32-bit variant

Applies when `sz == 0`.

```
LDXP <Wt1>, <Wt2>, [<Xn|SP>{,#0}]
```

64-bit variant

Applies when `sz == 1`.

```
LDXP <Xt1>, <Xt2>, [<Xn|SP>{,#0}]
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);

integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2;
boolean tag_checked = n != 31;
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *LDXP* on page K1-7866.

Assembler symbols

`<Wt1>`	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
`<Wt2>`	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
`<Xt1>`	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt2>`	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    X[t] = bits(datasize) UNKNOWN;    // In this case t = t2
elsif elsize == 32 then
    // 32-bit load exclusive pair (atomic)
    data = Mem[address, dbytes, AccType_ATOMIC];
    if BigEndian() then
        X[t] = data<datasize-1:elsize>;
        X[t2] = data<elsize-1:0>;
    else
        X[t] = data<elsize-1:0>;
        X[t2] = data<datasize-1:elsize>;
else // elsize == 64
    // 64-bit load exclusive pair (not atomic),
    // but must be 128-bit aligned
    if address != Align(address, dbytes) then
        AArch64.Abort(address, AArch64.AlignmentFault(AccType_ATOMIC, FALSE, FALSE));
    X[t] = Mem[address, 8, AccType_ATOMIC];
    X[t2] = Mem[address+8, 8, AccType_ATOMIC];
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.174 LDXR

Load Exclusive Register derives an address from a base register value, loads a 32-bit word or a 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores* on page B2-167. For information about memory accesses see *Load/Store addressing modes* on page C1-189.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	10	9	5	4	0
1	x	0	0	1	0	0	0	0	1	0	(1)(1)(1)(1)(1)		0	(1)(1)(1)(1)(1)		Rn		Rt	
size									L		Rs		o0	Rt2					

32-bit variant

Applies when `size == 10`.

```
LDXR <Wt>, [<Xn|SP>{,#0}]
```

64-bit variant

Applies when `size == 11`.

```
LDXR <Xt>, [<Xn|SP>{,#0}]
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
boolean tag_checked = n != 31;
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
AArch64.SetExclusiveMonitors(address, dbytes);

data = Mem[address, dbytes, AccType_ATOMIC];
X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.175 LDXRB

Load Exclusive Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores* on page B2-167. For information about memory accesses see *Load/Store addressing modes* on page C1-189.

31 30	29 28	27 26	25 24	23	22	21	20 16	15	14 10	9 5	4 0
0 0	0 0	1 0	0 0	0	1	0	(1)(1)(1)(1)(1)	0	(1)(1)(1)(1)(1)	Rn	Rt
size					L		Rs	o0	Rt2		

No offset variant

```
LDXRB <Wt>, [<Xn|SP>{,#0}]
```

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 1);

data = Mem[address, 1, AccType_ATOMIC];
X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.176 LDXRH

Load Exclusive Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores* on page B2-167. For information about memory accesses see *Load/Store addressing modes* on page C1-189.

31 30	29 28 27 26 25 24	23	22	21	20 16	15	14 10	9 5	4 0
0 1	0 0 1 0 0 0	0	1	0	(1)(1)(1)(1)(1)	0	(1)(1)(1)(1)(1)	Rn	Rt
size			L		Rs	o0	Rt2		

No offset variant

```
LDXRH <Wt>, [<Xn|SP>{,#0}]
```

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 2);

data = Mem[address, 2, AccType_ATOMIC];
X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.177 LSL (register)

Logical Shift Left (register) shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is left-shifted.

This instruction is an alias of the LSLV instruction. This means that:

- The encodings in this description are named to match the encodings of LSLV.
- The description of LSLV gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
sf	0	0	1	1	0	1	0	1	1	0	Rm		0	0	1	0	0	0	Rn		Rd	
																	op2					

32-bit variant

Applies when `sf == 0`.

```
LSL <Wd>, <Wn>, <Wm>
```

is equivalent to

```
LSLV <Wd>, <Wn>, <Wm>
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

```
LSL <Xd>, <Xn>, <Xm>
```

is equivalent to

```
LSLV <Xd>, <Xn>, <Xm>
```

and is always the preferred disassembly.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Wm>`	Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

The description of LSLV gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.178 LSL (immediate)

Logical Shift Left (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This instruction is an alias of the UBFM instruction. This means that:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.

31	30 29	28	27 26 25 24	23	22 21	16	15 10	9 5	4 0
sf	1 0	1	0 0 1 1	0	N	immr	imms	Rn	Rd
	opc								

32-bit variant

Applies when `sf == 0 && N == 0 && imms != 011111`.

```
LSL <Wd>, <Wn>, #<shift>
```

is equivalent to

```
UBFM <Wd>, <Wn>, #(-<shift> MOD 32), #(31-<shift>)
```

and is the preferred disassembly when `imms + 1 == immr`.

64-bit variant

Applies when `sf == 1 && N == 1 && imms != 111111`.

```
LSL <Xd>, <Xn>, #<shift>
```

is equivalent to

```
UBFM <Xd>, <Xn>, #(-<shift> MOD 64), #(63-<shift>)
```

and is the preferred disassembly when `imms + 1 == immr`.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
`<shift>`	For the 32-bit variant: is the shift amount, in the range 0 to 31. For the 64-bit variant: is the shift amount, in the range 0 to 63.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.179 LSLV

Logical Shift Left Variable shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is left-shifted.

This instruction is used by the alias LSL (register). The alias is always the preferred disassembly.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
sf	0	0	1	1	0	1	0	1	1	0	Rm		0	0	1	0	0	0	Rn		Rd	
																	op2					

32-bit variant

Applies when `sf == 0`.

```
LSLV <Wd>, <Wn>, <Wm>
```

64-bit variant

Applies when `sf == 1`.

```
LSLV <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ShiftType shift_type = DecodeShift(op2);
```

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

```
bits(datasize) result;
bits(datasize) operand2 = X[m];

result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize);
X[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.180 LSR (register)

Logical Shift Right (register) shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is an alias of the LSRV instruction. This means that:

- The encodings in this description are named to match the encodings of LSRV.
- The description of LSRV gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
sf	0	0	1	1	0	1	0	1	1	0	Rm	0	0	1	0	0	1	Rn	Rd
																op2			

32-bit variant

Applies when `sf == 0`.

```
LSR <Wd>, <Wn>, <Wm>
```

is equivalent to

```
LSRV <Wd>, <Wn>, <Wm>
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

```
LSR <Xd>, <Xn>, <Xm>
```

is equivalent to

```
LSRV <Xd>, <Xn>, <Xm>
```

and is always the preferred disassembly.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Wm>`	Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

The description of LSRV gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.181 LSR (immediate)

Logical Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This instruction is an alias of the UBFM instruction. This means that:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.

31	30 29	28	27 26 25 24	23	22 21	16	15	10 9	5 4	0
sf	1 0	1	0 0 1 1	0	N	immr	x 1 1 1 1 1	Rn	Rd	
	opc						imms			

32-bit variant

Applies when `sf == 0 && N == 0 && imms == 011111`.

```
LSR <Wd>, <Wn>, #<shift>
```

is equivalent to

```
UBFM <Wd>, <Wn>, #<shift>, #31
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1 && N == 1 && imms == 111111`.

```
LSR <Xd>, <Xn>, #<shift>
```

is equivalent to

```
UBFM <Xd>, <Xn>, #<shift>, #63
```

and is always the preferred disassembly.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
`<shift>`	For the 32-bit variant: is the shift amount, in the range 0 to 31, encoded in the "immr" field. For the 64-bit variant: is the shift amount, in the range 0 to 63, encoded in the "immr" field.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.182 LSRV

Logical Shift Right Variable shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is used by the alias LSR (register). The alias is always the preferred disassembly.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
sf	0	0	1	1	0	1	0	1	1	0	Rm		0	0	1	0	0	1	Rn		Rd	
																	op2					

32-bit variant

Applies when `sf == 0`.

```
LSRV <Wd>, <Wn>, <Wm>
```

64-bit variant

Applies when `sf == 1`.

```
LSRV <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ShiftType shift_type = DecodeShift(op2);
```

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

```
bits(datasize) result;
bits(datasize) operand2 = X[m];

result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize);
X[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.183 MADD

Multiply-Add multiplies two register values, adds a third register value, and writes the result to the destination register.

This instruction is used by the alias MUL. See *Alias conditions* for details of when each alias is preferred.

31	30 29	28	27 26 25 24	23 22 21 20	16	15	14 10	9 5	4 0
sf	0 0	1	1 0 1 1	0 0 0	Rm	0	Ra	Rn	Rd
						o0			

32-bit variant

Applies when `sf == 0`.

```
MADD <Wd>, <Wn>, <Wm>, <Wa>
```

64-bit variant

Applies when `sf == 1`.

```
MADD <Xd>, <Xn>, <Xm>, <Xa>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
integer destsize = if sf == '1' then 64 else 32;
```

Alias conditions

Alias	is preferred when
MUL	`Ra == '11111'`

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
`<Wm>`	Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
`<Wa>`	Is the 32-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Xa> Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation

```
bits(destsize) operand1 = X[n];
bits(destsize) operand2 = X[m];
bits(destsize) operand3 = X[a];

integer result;

result = UInt(operand3) + (UInt(operand1) * UInt(operand2));

X[d] = result<destsize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.184 MNEG

Multiply-Negate multiplies two register values, negates the product, and writes the result to the destination register.

This instruction is an alias of the MSUB instruction. This means that:

- The encodings in this description are named to match the encodings of MSUB.
- The description of MSUB gives the operational pseudocode for this instruction.

31	30 29	28	27 26 25 24	23 22 21 20	16	15	14 10	9 5	4 0
sf	0 0	1	1 0 1 1	0 0 0	Rm	1	1 1 1 1 1	Rn	Rd
						o0	Ra		

32-bit variant

Applies when `sf == 0`.

```
MNEG <Wd>, <Wn>, <Wm>
```

is equivalent to

```
MSUB <Wd>, <Wn>, <Wm>, WZR
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

```
MNEG <Xd>, <Xn>, <Xm>
```

is equivalent to

```
MSUB <Xd>, <Xn>, <Xm>, XZR
```

and is always the preferred disassembly.

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of MSUB gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.185 MOV (to/from SP)

Move between register and stack pointer : `Rd = Rn`

This instruction is an alias of the ADD (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of ADD (immediate).
- The description of ADD (immediate) gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	10	9	5	4	0
sf	0	0	1	0	0	0	1	0	0	0 0 0 0 0 0 0 0 0 0 0 0		Rn		Rd	
	op	S							sh	imm12					

32-bit variant

Applies when `sf == 0`.

```
MOV <Wd|WSP>, <Wn|WSP>
```

is equivalent to

```
ADD <Wd|WSP>, <Wn|WSP>, #0
```

and is the preferred disassembly when `(Rd == '11111' || Rn == '11111')`.

64-bit variant

Applies when `sf == 1`.

```
MOV <Xd|SP>, <Xn|SP>
```

is equivalent to

```
ADD <Xd|SP>, <Xn|SP>, #0
```

and is the preferred disassembly when `(Rd == '11111' || Rn == '11111')`.

Assembler symbols

`<Wd|WSP>`	Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
`<Wn|WSP>`	Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
`<Xd|SP>`	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
`<Xn|SP>`	Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

Operation

The description of ADD (immediate) gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.186 MOV (inverted wide immediate)

Move (inverted wide immediate) moves an inverted 16-bit immediate value to a register.

This instruction is an alias of the MOVN instruction. This means that:

- The encodings in this description are named to match the encodings of MOVN.
- The description of MOVN gives the operational pseudocode for this instruction.

31	30 29	28 27 26 25 24	23 22	21 20	5 4	0
sf	0 0	1 0 0 1 0 1	hw	imm16		Rd
	opc					

32-bit variant

Applies when `sf == 0 && hw == 0x`.

```
MOV <Wd>, #<imm>
```

is equivalent to

```
MOVN <Wd>, #<imm16>, LSL #<shift>
```

and is the preferred disassembly when `! (IsZero(imm16) && hw != '00') && ! IsOnes(imm16)`.

64-bit variant

Applies when `sf == 1`.

```
MOV <Xd>, #<imm>
```

is equivalent to

```
MOVN <Xd>, #<imm16>, LSL #<shift>
```

and is the preferred disassembly when `! (IsZero(imm16) && hw != '00')`.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<imm>`	For the 32-bit variant: is a 32-bit immediate, the bitwise inverse of which can be encoded in "imm16:hw", but excluding 0xffff0000 and 0x0000ffff For the 64-bit variant: is a 64-bit immediate, the bitwise inverse of which can be encoded in "imm16:hw".
`<shift>`	For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16. For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

Operation

The description of MOVN gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.187 MOV (wide immediate)

Move (wide immediate) moves a 16-bit immediate value to a register.

This instruction is an alias of the MOVZ instruction. This means that:

- The encodings in this description are named to match the encodings of MOVZ.
- The description of MOVZ gives the operational pseudocode for this instruction.

31	30 29	28	27 26 25 24	23 22	21 20	5 4	0
sf	1 0	1	0 0 1 0 1	hw	imm16		Rd
	opc						

32-bit variant

Applies when `sf == 0 && hw == 0x`.

```
MOV <Wd>, #<imm>
```

is equivalent to

```
MOVZ <Wd>, #<imm16>, LSL #<shift>
```

and is the preferred disassembly when `! (IsZero(imm16) && hw != '00')`.

64-bit variant

Applies when `sf == 1`.

```
MOV <Xd>, #<imm>
```

is equivalent to

```
MOVZ <Xd>, #<imm16>, LSL #<shift>
```

and is the preferred disassembly when `! (IsZero(imm16) && hw != '00')`.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<imm>`	For the 32-bit variant: is a 32-bit immediate which can be encoded in "imm16:hw". For the 64-bit variant: is a 64-bit immediate which can be encoded in "imm16:hw".
`<shift>`	For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16. For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

Operation

The description of MOVZ gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.188 MOV (bitmask immediate)

Move (bitmask immediate) writes a bitmask immediate value to a register.

This instruction is an alias of the ORR (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of ORR (immediate).
- The description of ORR (immediate) gives the operational pseudocode for this instruction.

31	30 29	28	27 26 25 24	23	22 21	16	15 10	9 5	4 0
sf	0 1	1	0 0 1 0	0	N	immr	imms	1 1 1 1 1	Rd
	opc							Rn	

32-bit variant

Applies when `sf == 0 && N == 0`.

```
MOV <Wd|WSP>, #<imm>
```

is equivalent to

```
ORR <Wd|WSP>, WZR, #<imm>
```

and is the preferred disassembly when `! MoveWidePreferred(sf, N, imms, immr)`.

64-bit variant

Applies when `sf == 1`.

```
MOV <Xd|SP>, #<imm>
```

is equivalent to

```
ORR <Xd|SP>, XZR, #<imm>
```

and is the preferred disassembly when `! MoveWidePreferred(sf, N, imms, immr)`.

Assembler symbols

`<Wd|WSP>` Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

`<Xd|SP>` Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

`<imm>` For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr", but excluding values which could be encoded by MOVZ or MOVN.

For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr", but excluding values which could be encoded by MOVZ or MOVN.

Operation

The description of ORR (immediate) gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.189 MOV (register)

Move (register) copies the value in a source register to the destination register.

This instruction is an alias of the ORR (shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of ORR (shifted register).
- The description of ORR (shifted register) gives the operational pseudocode for this instruction.

31	30 29	28	27 26 25 24	23 22	21	20 16	15 10	9 5	4 0
sf	0 1	0	1 0 1 0	0 0	0	Rm	0 0 0 0 0 0	1 1 1 1 1	Rd
	opc			shift	N		imm6	Rn	

32-bit variant

Applies when `sf == 0`.

```
MOV <Wd>, <Wm>
```

is equivalent to

```
ORR <Wd>, WZR, <Wm>
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

```
MOV <Xd>, <Xm>
```

is equivalent to

```
ORR <Xd>, XZR, <Xm>
```

and is always the preferred disassembly.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wm>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xm>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

Operation

The description of ORR (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.190 MOVK

Move wide with keep moves an optionally-shifted 16-bit immediate value into a register, keeping other bits unchanged.

31	30 29	28 27 26 25 24	23 22	21 20	5 4	0
sf	1 1 (opc)	1 0 0 1 0 1	hw	imm16		Rd

32-bit variant

Applies when `sf == 0 && hw == 0x`.

```
MOVK <Wd>, #<imm>{, LSL #<shift>}
```

64-bit variant

Applies when `sf == 1`.

```
MOVK <Xd>, #<imm>{, LSL #<shift>}
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer datasize = if sf == '1' then 64 else 32;
integer pos;

if sf == '0' && hw<1> == '1' then UNDEFINED;
pos = UInt(hw:'0000');
```

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<imm>`	Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.
`<shift>`	For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16. For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

Operation

```
bits(datasize) result;

result = X[d];
result<pos+15:pos> = imm16;
X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.191 MOVN

Move wide with NOT moves the inverse of an optionally-shifted 16-bit immediate value to a register.

This instruction is used by the alias MOV (inverted wide immediate). See *Alias conditions* for details of when each alias is preferred.

31	30 29	28 27 26 25 24	23 22	21 20	5 4	0
sf	0 0 (opc)	1 0 0 1 0 1	hw	imm16		Rd

32-bit variant

Applies when `sf == 0 && hw == 0x`.

```
MOVN <Wd>, #<imm>{, LSL #<shift>}
```

64-bit variant

Applies when `sf == 1`.

```
MOVN <Xd>, #<imm>{, LSL #<shift>}
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer datasize = if sf == '1' then 64 else 32;
integer pos;

if sf == '0' && hw<1> == '1' then UNDEFINED;
pos = UInt(hw:'0000');
```

Alias conditions

Alias	of variant	is preferred when
MOV (inverted wide immediate)	64-bit	`! (IsZero(imm16) && hw != '00')`
MOV (inverted wide immediate)	32-bit	`! (IsZero(imm16) && hw != '00') && ! IsOnes(imm16)`

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<imm>`	Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.
`<shift>`	For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16. For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

Operation

```
bits(datasize) result;

result = Zeros();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
result<pos+15:pos> = imm16;
result = NOT(result);
X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.192 MOVZ

Move wide with zero moves an optionally-shifted 16-bit immediate value to a register.

This instruction is used by the alias MOV (wide immediate). See *Alias conditions* for details of when each alias is preferred.

31	30 29	28 27 26 25 24	23 22	21 20 … 5	4 … 0
sf	1 0 (opc)	1 0 0 1 0 1	hw	imm16	Rd

32-bit variant

Applies when `sf == 0 && hw == 0x`.

```
MOVZ <Wd>, #<imm>{, LSL #<shift>}
```

64-bit variant

Applies when `sf == 1`.

```
MOVZ <Xd>, #<imm>{, LSL #<shift>}
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer datasize = if sf == '1' then 64 else 32;
integer pos;

if sf == '0' && hw<1> == '1' then UNDEFINED;
pos = UInt(hw:'0000');
```

Alias conditions

Alias	is preferred when
MOV (wide immediate)	`! (IsZero(imm16) && hw != '00')`

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<imm>`	Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.
`<shift>`	For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16. For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

Operation

```
bits(datasize) result;

result = Zeros();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
result<pos+15:pos> = imm16;
X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.193 MRS

Move System Register allows the PE to read an AArch64 System register into a general-purpose register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18 16	15 12	11 8	7 5	4 0
1	1	0	1	0	1	0	1	0	0	1	1	o0	op1	CRn	CRm	op2	Rt
										L							

System variant

```
MRS <Xt>, (<systemreg>|S<op0>_<op1>_<Cn>_<Cm>_<op2>)
```

Decode for this encoding

```
AArch64.CheckSystemAccess('1':o0, op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);

integer sys_op0 = 2 + UInt(o0);
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
```

Assembler symbols

`<Xt>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.

`<systemreg>` Is a System register name, encoded in the "o0:op1:CRn:CRm:op2".

The System register names are defined in Chapter D13 *AArch64 System Register Descriptions*.

`<op0>` Is an unsigned immediate, encoded in the "o0" field. It can have the following values:

`2` when `o0 = 0`

`3` when `o0 = 1`

`<op1>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

`<Cn>` Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.

`<Cm>` Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

`<op2>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

Operation

```
X[t] = AArch64.SysRegRead(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.194 MSR (immediate)

Move immediate value to Special Register moves an immediate value to selected bits of the PSTATE. For more information, see PSTATE.

The bits that can be written by this instruction are:

- PSTATE.D, PSTATE.A, PSTATE.I, PSTATE.F, and PSTATE.SP.
- If ARMv8.0-SSBS is implemented, PSTATE.SSBS.
- If ARMv8.1-PAN is implemented, PSTATE.PAN.
- If ARMv8.2-UAO is implemented, PSTATE.UAO.
- If ARMv8.4-DIT is implemented, PSTATE.DIT.
- If ARMv8.5-MemTag is implemented, PSTATE.TCO.

31 30 29 28	27 26 25 24	23 22	21	20	19 18 16	15 14 13 12	11 8	7 5	4	3 2 1 0
1 1 0 1	0 1 0 1	0 0	0	0	op1	0 1 0 0	CRm	op2	1	1 1 1 1

System variant

```
MSR <pstatefield>, #<imm>
```

Decode for this encoding

```
if op1 == '000' && op2 == '000' then SEE "CFINV";
if op1 == '000' && op2 == '001' then SEE "XAFLAG";
if op1 == '000' && op2 == '010' then SEE "AXFLAG";

AArch64.CheckSystemAccess('00', op1, '0100', CRm, op2, '11111', '0');
bits(2) min_EL;
boolean need_secure;
(min_EL, need_secure) = AArch64.GetMinELSecurityState(op1);
if UInt(PSTATE.EL) < UInt(min_EL) || (need_secure && !IsSecure()) then
    UNDEFINED;

PSTATEField field;
case op1:op2 of
    when '000 011'
        if !HaveUAOExt() then
            UNDEFINED;
        field = PSTATEField_UAO;
    when '000 100'
        if !HavePANExt() then
            UNDEFINED;
        field = PSTATEField_PAN;
    when '000 101' field = PSTATEField_SP;
    when '011 010'
        if !HaveDITExt() then
            UNDEFINED;
        field = PSTATEField_DIT;
    when '011 100'
        if !HaveMTEExt() then
            UNDEFINED;
        field = PSTATEField_TCO;
    when '011 110' field = PSTATEField_DAIFSet;
    when '011 111' field = PSTATEField_DAIFClr;
    when '011 001'
        if !HaveSSBSExt() then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            UNDEFINED;
        field = PSTATEField_SSBS;
    otherwise UNDEFINED;

// Check that an AArch64 MSR/MRS access to the DAIF flags is permitted
if PSTATE.EL == EL0 && field IN {PSTATEField_DAIFSet, PSTATEField_DAIFClr} then
    if !ELUsingAArch32(EL1) && ((EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') || SCTLR_EL1.UMA == '0')
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
```

Assembler symbols

`<pstatefield>` Is a PSTATE field name, encoded in the "op1:op2" field. It can have the following values:

`SPSel` when `op1 = 000`, `op2 = 101`

`DAIFSet` when `op1 = 011`, `op2 = 110`

`DAIFClr` when `op1 = 011`, `op2 = 111`

When ARMv8.2-UAO is implemented, the following value is also valid:

`UAO` when `op1 = 000`, `op2 = 011`

When ARMv8.1-PAN is implemented, the following value is also valid:

`PAN` when `op1 = 000`, `op2 = 100`

When ARMv8.0-SSBS is implemented, the following value is also valid:

`SSBS` when `op1 = 011`, `op2 = 001`

When ARMv8.4-DIT is implemented, the following value is also valid:

`DIT` when `op1 = 011`, `op2 = 010`

When ARMv8.5-MemTag is implemented, the following value is also valid:

`TCO` when `op1 = 011`, `op2 = 100`

See *PSTATE* on page C4-276 when `op1 = 000`, `op2 = 00x`.

See *PSTATE* on page C4-276 when `op1 = 000`, `op2 = 010`.

The following encodings are reserved:

- `op1 = 000`, `op2 = 11x`.
- `op1 = 001`, `op2 = xxx`.
- `op1 = 010`, `op2 = xxx`.
- `op1 = 011`, `op2 = 000`.
- `op1 = 011`, `op2 = 011`.
- `op1 = 011`, `op2 = 101`.
- `op1 = 1xx`, `op2 = xxx`.

`<imm>` Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

Operation

```
case field of
    when PSTATEField_SSBS
        PSTATE.SSBS = CRm<0>;
    when PSTATEField_SP
        PSTATE.SP = CRm<0>;
    when PSTATEField_DAIFSet
        PSTATE.D = PSTATE.D OR CRm<3>;
        PSTATE.A = PSTATE.A OR CRm<2>;
        PSTATE.I = PSTATE.I OR CRm<1>;
        PSTATE.F = PSTATE.F OR CRm<0>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
when PSTATEField_DAIFClr
    PSTATE.D = PSTATE.D AND NOT(CRm<3>);
    PSTATE.A = PSTATE.A AND NOT(CRm<2>);
    PSTATE.I = PSTATE.I AND NOT(CRm<1>);
    PSTATE.F = PSTATE.F AND NOT(CRm<0>);
when PSTATEField_PAN
    PSTATE.PAN = CRm<0>;
when PSTATEField_UAO
    PSTATE.UAO = CRm<0>;
when PSTATEField_DIT
    PSTATE.DIT = CRm<0>;
when PSTATEField_TCO
    PSTATE.TCO = CRm<0>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.195 MSR (register)

Move general-purpose register to System Register allows the PE to write an AArch64 System register from a general-purpose register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18 … 16	15 … 12	11 … 8	7 … 5	4 … 0
1	1	0	1	0	1	0	1	0	0	0	1	o0	op1	CRn	CRm	op2	Rt
										L							

System variant

```
MSR (<systemreg>|S<op0>_<op1>_<Cn>_<Cm>_<op2>), <Xt>
```

Decode for this encoding

```
AArch64.CheckSystemAccess('1':o0, op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);

integer sys_op0 = 2 + UInt(o0);
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
```

Assembler symbols

`<systemreg>` Is a System register name, encoded in the "o0:op1:CRn:CRm:op2".

The System register names are defined in Chapter D13 *AArch64 System Register Descriptions*.

`<op0>` Is an unsigned immediate, encoded in the "o0" field. It can have the following values:

`2` when `o0 = 0`

`3` when `o0 = 1`

`<op1>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

`<Cn>` Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.

`<Cm>` Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

`<op2>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

`<Xt>` Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

Operation

```
AArch64.SysRegWrite(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.196 MSUB

Multiply-Subtract multiplies two register values, subtracts the product from a third register value, and writes the result to the destination register.

This instruction is used by the alias MNEG. See *Alias conditions* for details of when each alias is preferred.

31	30 29	28 27 26 25 24	23 22 21 20	16	15 14	10 9	5 4	0
sf	0 0	1 1 0 1 1	0 0 0	Rm	1	Ra	Rn	Rd
					o0			

32-bit variant

Applies when `sf == 0`.

```
MSUB <Wd>, <Wn>, <Wm>, <Wa>
```

64-bit variant

Applies when `sf == 1`.

```
MSUB <Xd>, <Xn>, <Xm>, <Xa>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
integer destsize = if sf == '1' then 64 else 32;
```

Alias conditions

Alias	is preferred when
MNEG	`Ra == '11111'`

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

`<Wa>` Is the 32-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Xa> Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

Operation

```
bits(destsize) operand1 = X[n];
bits(destsize) operand2 = X[m];
bits(destsize) operand3 = X[a];

integer result;

result = UInt(operand3) - (UInt(operand1) * UInt(operand2));
X[d] = result<destsize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.197 MUL

Multiply : `Rd = Rn * Rm`

This instruction is an alias of the MADD instruction. This means that:

- The encodings in this description are named to match the encodings of MADD.
- The description of MADD gives the operational pseudocode for this instruction.

31	30 29	28 27 26 25 24	23 22 21 20	16	15	14 10	9 5	4 0
sf	0 0	1 1 0 1 1	0 0 0	Rm	0	1 1 1 1 1	Rn	Rd
					o0	Ra		

32-bit variant

Applies when `sf == 0`.

```
MUL <Wd>, <Wn>, <Wm>
```

is equivalent to

```
MADD <Wd>, <Wn>, <Wm>, WZR
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

```
MUL <Xd>, <Xn>, <Xm>
```

is equivalent to

```
MADD <Xd>, <Xn>, <Xm>, XZR
```

and is always the preferred disassembly.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
`<Wm>`	Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of MADD gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.198 MVN

Bitwise NOT writes the bitwise inverse of a register value to the destination register.

This instruction is an alias of the ORN (shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of ORN (shifted register).
- The description of ORN (shifted register) gives the operational pseudocode for this instruction.

31	30 29	28 27 26 25 24	23 22	21	20 16	15 10	9 5	4 0
sf	0 1	0 1 0 1 0	shift	1	Rm	imm6	1 1 1 1 1	Rd
	opc			N			Rn	

32-bit variant

Applies when `sf == 0`.

```
MVN <Wd>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
ORN <Wd>, WZR, <Wm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

```
MVN <Xd>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
ORN <Xd>, XZR, <Xm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wm>` Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xm>` Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:

`LSL`	when `shift = 00`
`LSR`	when `shift = 01`
`ASR`	when `shift = 10`
`ROR`	when `shift = 11`

`<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

The description of ORN (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.199 NEG (shifted register)

Negate (shifted register) negates an optionally-shifted register value, and writes the result to the destination register.

This instruction is an alias of the SUB (shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of SUB (shifted register).
- The description of SUB (shifted register) gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23 22	21	20 16	15 10	9 5	4 0
sf	1	0	0	1	0	1	1	shift	0	Rm	imm6	1 1 1 1 1	Rd
	op	S										Rn	

32-bit variant

Applies when `sf == 0`.

```
NEG <Wd>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
SUB  <Wd>, WZR, <Wm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

```
NEG <Xd>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
SUB  <Xd>, XZR, <Xm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wm>` Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xm>` Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in the "shift" field. It can have the following values:

`LSL`	when `shift = 00`
`LSR`	when `shift = 01`
`ASR`	when `shift = 10`

The encoding `shift = 11` is reserved.

`<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

The description of SUB (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.200 NEGS

Negate, setting flags, negates an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is an alias of the SUBS (shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of SUBS (shifted register).
- The description of SUBS (shifted register) gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23 22	21	20 16	15 10	9 5	4 0
sf	1	1	0	1	0	1	1	shift	0	Rm	imm6	1 1 1 1 1	Rd
	op	S										Rn	

32-bit variant

Applies when `sf == 0`.

```
NEGS <Wd>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
SUBS <Wd>, WZR, <Wm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

```
NEGS <Xd>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
SUBS <Xd>, XZR, <Xm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wm>` Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xm>` Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in the "shift" field. It can have the following values:

`LSL`	when `shift = 00`
`LSR`	when `shift = 01`
`ASR`	when `shift = 10`

The encoding `shift = 11` is reserved.

`<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

The description of SUBS (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.201 NGC

Negate with Carry negates the sum of a register value and the value of NOT (Carry flag), and writes the result to the destination register.

This instruction is an alias of the SBC instruction. This means that:

- The encodings in this description are named to match the encodings of SBC.
- The description of SBC gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
sf	1	0	1	1	0	1	0	0	0	0	Rm		0	0	0	0	0	0	1	1	1	1	1	Rd
	op	S																		Rn		

32-bit variant

Applies when `sf == 0`.

```
NGC <Wd>, <Wm>
```

is equivalent to

```
SBC <Wd>, WZR, <Wm>
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

```
NGC <Xd>, <Xm>
```

is equivalent to

```
SBC <Xd>, XZR, <Xm>
```

and is always the preferred disassembly.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wm>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xm>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

Operation

The description of SBC gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.202 NGCS

Negate with Carry, setting flags, negates the sum of a register value and the value of NOT (Carry flag), and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is an alias of the SBCS instruction. This means that:

- The encodings in this description are named to match the encodings of SBCS.
- The description of SBCS gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
sf	1	1	1	1	0	1	0	0	0	0	Rm		0	0	0	0	0	0	1 1 1 1 1		Rd	
	op	S																	Rn			

32-bit variant

Applies when `sf == 0`.

```
NGCS <Wd>, <Wm>
```

is equivalent to

```
SBCS <Wd>, WZR, <Wm>
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

```
NGCS <Xd>, <Xm>
```

is equivalent to

```
SBCS <Xd>, XZR, <Xm>
```

and is always the preferred disassembly.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wm>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xm>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

Operation

The description of SBCS gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.203 NOP

No Operation does nothing, other than advance the value of the program counter by 4. This instruction can be used for instruction alignment purposes.

Note

The timing effects of including a `NOP` instruction in a program are not guaranteed. It can increase execution time, leave it unchanged, or even reduce it. Therefore, `NOP` instructions are not suitable for timing loops.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11			8	7			5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1
																				CRm				op2								

System variant

```
NOP
```

Decode for this encoding

```
// Empty.
```

Operation

```
// do nothing
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.204 ORN (shifted register)

Bitwise OR NOT (shifted register) performs a bitwise (inclusive) OR of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.

This instruction is used by the alias MVN. See *Alias conditions* for details of when each alias is preferred.

31	30 29	28 27 26 25	24 23 22	21	20 16	15 10	9 5	4 0
sf	0 1	0 1 0 1 0	shift	1	Rm	imm6	Rn	Rd
	opc			N				

32-bit variant

Applies when `sf == 0`.

```
ORN <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit variant

Applies when `sf == 1`.

```
ORN <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

Alias conditions

Alias	is preferred when
MVN	`Rn == '11111'`

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:

- `LSL` when `shift = 00`
- `LSR` when `shift = 01`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ASR when shift = 10

ROR when shift = 11

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

operand2 = NOT(operand2);

result = operand1 OR operand2;
X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.205 ORR (immediate)

Bitwise OR (immediate) performs a bitwise (inclusive) OR of a register value and an immediate register value, and writes the result to the destination register.

This instruction is used by the alias MOV (bitmask immediate). See *Alias conditions* for details of when each alias is preferred.

31	30 29	28 27 26 25 24	23	22 21 16	15 10	9 5	4 0
sf	0 1 (opc)	1 0 0 1 0 0	N	immr	imms	Rn	Rd

32-bit variant

Applies when `sf == 0 && N == 0`.

```
ORR <Wd|WSP>, <Wn>, #<imm>
```

64-bit variant

Applies when `sf == 1`.

```
ORR <Xd|SP>, <Xn>, #<imm>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;
if sf == '0' && N != '0' then UNDEFINED;
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);
```

Alias conditions

Alias	is preferred when
MOV (bitmask immediate)	`Rn == '11111' && ! MoveWidePreferred(sf, N, imms, immr)`

Assembler symbols

`<Wd|WSP>`	Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
`<Xd|SP>`	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
`<imm>`	For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr". For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];

result = operand1 OR imm;
if d == 31 then
    SP[] = result;
else
    X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.206 ORR (shifted register)

Bitwise OR (shifted register) performs a bitwise (inclusive) OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

This instruction is used by the alias MOV (register). See *Alias conditions* for details of when each alias is preferred.

31	30 29	28	27 26 25 24	23 22	21	20 16	15 10	9 5	4 0
sf	0 1	0	1 0 1 0	shift	0	Rm	imm6	Rn	Rd
	opc				N				

32-bit variant

Applies when `sf == 0`.

```
ORR <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit variant

Applies when `sf == 1`.

```
ORR <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

Alias conditions

Alias	is preferred when
MOV (register)	`shift == '00' && imm6 == '000000' && Rn == '11111'`

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:

`LSL` when `shift = 00`

`LSR` when `shift = 01`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`ASR` when `shift = 10`

`ROR` when `shift = 11`

`<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

result = operand1 OR operand2;
X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.207 PACDA, PACDZA

Pointer Authentication Code for Data address, using key A. This instruction computes and inserts a pointer authentication code for a data address, using a modifier and key A.

The address is in the general-purpose register that is specified by `<Xd>`.

The modifier is:

- In the general-purpose register or stack pointer that is specified by `<Xn|SP>` for `PACDA`.
- The value zero, for `PACDZA`.

ARMv8.3

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	0	Z	0	1	0	Rn		Rd	

PACDA variant

Applies when `Z == 0`.

```
PACDA <Xd>, <Xn|SP>
```

PACDZA variant

Applies when `Z == 1 && Rn == 11111`.

```
PACDZA <Xd>
```

Decode for all variants of this encoding

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UNDEFINED;

if Z == '0' then // PACDA
    if n == 31 then source_is_sp = TRUE;
else // PACDZA
    if n != 31 then UNDEFINED;
```

Assembler symbols

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

```
if source_is_sp then
    X[d] = AddPACDA(X[d], SP[]);
else
    X[d] = AddPACDA(X[d], X[n]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.208 PACDB, PACDZB

Pointer Authentication Code for Data address, using key B. This instruction computes and inserts a pointer authentication code for a data address, using a modifier and key B.

The address is in the general-purpose register that is specified by `<Xd>`.

The modifier is:

- In the general-purpose register or stack pointer that is specified by `<Xn|SP>` for `PACDB`.
- The value zero, for `PACDZB`.

ARMv8.3

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	0	Z	0	1	1	Rn		Rd	

PACDB variant

Applies when `Z == 0`.

```
PACDB <Xd>, <Xn|SP>
```

PACDZB variant

Applies when `Z == 1 && Rn == 11111`.

```
PACDZB <Xd>
```

Decode for all variants of this encoding

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UNDEFINED;

if Z == '0' then // PACDB
    if n == 31 then source_is_sp = TRUE;
else // PACDZB
    if n != 31 then UNDEFINED;
```

Assembler symbols

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

```
if source_is_sp then
    X[d] = AddPACDB(X[d], SP[]);
else
    X[d] = AddPACDB(X[d], X[n]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.209 PACGA

Pointer Authentication Code, using Generic key. This instruction computes the pointer authentication code for an address in the first source register, using a modifier in the second source register, and the Generic key. The computed pointer authentication code is returned in the upper 32 bits of the destination register.

ARMv8.3

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
1	0	0	1	1	0	1	0	1	1	0	Rm		0	0	1	1	0	0	Rn		Rd	

Integer variant

```
PACGA <Xd>, <Xn>, <Xm|SP>
```

Decode for this encoding

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if !HavePACExt() then
    UNDEFINED;

if m == 31 then source_is_sp = TRUE;
```

Assembler symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm|SP>	Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the "Rm" field.

Operation

```
if source_is_sp then
    X[d] = AddPACGA(X[n], SP[]);
else
    X[d] = AddPACGA(X[n], X[m]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.210 PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA

Pointer Authentication Code for Instruction address, using key A. This instruction computes and inserts a pointer authentication code for an instruction address, using a modifier and key A.

The address is:

- In the general-purpose register that is specified by `<Xd>` for `PACIA` and `PACIZA`.
- In X17, for `PACIA1716`.
- In X30, for `PACIASP` and `PACIAZ`.

The modifier is:

- In the general-purpose register or stack pointer that is specified by `<Xn|SP>` for `PACIA`.
- The value zero, for `PACIZA` and `PACIAZ`.
- In X16, for `PACIA1716`.
- In SP, for `PACIASP`.

Integer

ARMv8.3

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	0	Z	0	0	0	Rn		Rd	

PACIA variant

Applies when `Z == 0`.

```
PACIA <Xd>, <Xn|SP>
```

PACIZA variant

Applies when `Z == 1 && Rn == 11111`.

```
PACIZA <Xd>
```

Decode for all variants of this encoding

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UNDEFINED;

if Z == '0' then // PACIA
    if n == 31 then source_is_sp = TRUE;
else // PACIZA
    if n != 31 then UNDEFINED;
```

System

ARMv8.3

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 8	7 5	4	3 2 1 0
1 1 0 1	0 1 0 1	0 0 0 0	0 1 1	0 0 1 0	0 0 x 1	0 0 x	1	1 1 1 1
					CRm	op2		

PACIA1716 variant

Applies when `CRm == 0001 && op2 == 000`.

```
PACIA1716
```

PACIASP variant

Applies when `CRm == 0011 && op2 == 001`.

```
PACIASP
```

PACIAZ variant

Applies when `CRm == 0011 && op2 == 000`.

```
PACIAZ
```

Decode for all variants of this encoding

```
integer d;
integer n;
boolean source_is_sp = FALSE;

case CRm:op2 of
    when '0011 000'    // PACIAZ
        d = 30;
        n = 31;
    when '0011 001'    // PACIASP
        d = 30;
        source_is_sp = TRUE;
        if HaveBTIExt() then
            // Check for branch target compatibility between PSTATE.BTYPE
            // and implicit branch target of PACIASP instruction.
            SetBTypeCompatible(BTypeCompatible_PACIXSP());

    when '0001 000'    // PACIA1716
        d = 17;
        n = 16;
    when '0001 010' SEE "PACIB";
    when '0001 100' SEE "AUTIA";
    when '0001 110' SEE "AUTIB";
    when '0011 01x' SEE "PACIB";
    when '0011 10x' SEE "AUTIA";
    when '0011 11x' SEE "AUTIB";
    when '0000 111' SEE "XPACLRI";
    otherwise SEE "HINT";
```

Assembler symbols

`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if HavePACExt() then
    if source_is_sp then
        X[d] = AddPACIA(X[d], SP[]);
    else
        X[d] = AddPACIA(X[d], X[n]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.211 PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB

Pointer Authentication Code for Instruction address, using key B. This instruction computes and inserts a pointer authentication code for an instruction address, using a modifier and key B.

The address is:

- In the general-purpose register that is specified by `<Xd>` for `PACIB` and `PACIZB`.
- In X17, for `PACIB1716`.
- In X30, for `PACIBSP` and `PACIBZ`.

The modifier is:

- In the general-purpose register or stack pointer that is specified by `<Xn|SP>` for `PACIB`.
- The value zero, for `PACIZB` and `PACIBZ`.
- In X16, for `PACIB1716`.
- In SP, for `PACIBSP`.

Integer

ARMv8.3

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	0	Z	0	0	1	Rn		Rd	

PACIB variant

Applies when `Z == 0`.

```
PACIB <Xd>, <Xn|SP>
```

PACIZB variant

Applies when `Z == 1 && Rn == 11111`.

```
PACIZB <Xd>
```

Decode for all variants of this encoding

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UNDEFINED;

if Z == '0' then // PACIB
    if n == 31 then source_is_sp = TRUE;
else // PACIZB
    if n != 31 then UNDEFINED;
```

System

ARMv8.3

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11			8	7			5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	x	1	0	1	x		1	1	1	1	1
																				CRm				op2								

PACIB1716 variant

Applies when `CRm == 0001 && op2 == 010`.

```
PACIB1716
```

PACIBSP variant

Applies when `CRm == 0011 && op2 == 011`.

```
PACIBSP
```

PACIBZ variant

Applies when `CRm == 0011 && op2 == 010`.

```
PACIBZ
```

Decode for all variants of this encoding

```
integer d;
integer n;
boolean source_is_sp = FALSE;

case CRm:op2 of
    when '0011 010'    // PACIBZ
        d = 30;
        n = 31;
    when '0011 011'    // PACIBSP
        d = 30;
        source_is_sp = TRUE;
        if HaveBTIExt() then
            // Check for branch target compatibility between PSTATE.BTYPE
            // and implicit branch target of PACIBSP instruction.
            SetBTypeCompatible(BTypeCompatible_PACIXSP());
    when '0001 010'    // PACIB1716
        d = 17;
        n = 16;
    when '0001 000' SEE "PACIA";
    when '0001 100' SEE "AUTIA";
    when '0001 110' SEE "AUTIB";
    when '0011 00x' SEE "PACIA";
    when '0011 10x' SEE "AUTIA";
    when '0011 11x' SEE "AUTIB";
    when '0000 111' SEE "XPACLRI";
    otherwise SEE "HINT";
```

Assembler symbols

`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation for all encodings

```
if HavePACExt() then
    if source_is_sp then
        X[d] = AddPACIB(X[d], SP[]);
    else
        X[d] = AddPACIB(X[d], X[n]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.212 PRFM (immediate)

Prefetch Memory (immediate) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFM instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory* on page C3-220.

For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

31 30	29 28 27	26	25 24	23 22	21 ... 10	9 ... 5	4 ... 0
1 1	1 1 1	0	0 1	1 0	imm12	Rn	Rt
size				opc			

Unsigned offset variant

```
PRFM (<prfop>|#<imm5>), [<Xn|SP>{, #<pimm>}]
```

Decode for this encoding

```
bits(64) offset = LSL(ZeroExtend(imm12, 64), 3);
```

Assembler symbols

`<prfop>` Is the prefetch operation, defined as `<type><target><policy>`.

`<type>` is one of:

- `PLD` Prefetch for load, encoded in the "Rt<4:3>" field as `0b00`.
- `PLI` Preload instructions, encoded in the "Rt<4:3>" field as `0b01`.
- `PST` Prefetch for store, encoded in the "Rt<4:3>" field as `0b10`.

`<target>` is one of:

- `L1` Level 1 cache, encoded in the "Rt<2:1>" field as `0b00`.
- `L2` Level 2 cache, encoded in the "Rt<2:1>" field as `0b01`.
- `L3` Level 3 cache, encoded in the "Rt<2:1>" field as `0b10`.

`<policy>` is one of:

- `KEEP` Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as `0`.
- `STRM` Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as `1`.

For more information on these prefetch operations, see *Prefetch memory* on page C3-220.

For other encodings of the "Rt" field, use `<imm5>`.

`<imm5>` Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field.

This syntax is only for encodings that are not accessible using `<prfop>`.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<pimm>` Is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
```

Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(FALSE);

bits(64) address;

if n == 31 then
    address = SP[];
else
    address = X[n];

address = address + offset;

Prefetch(address, t<4:0>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.213 PRFM (literal)

Prefetch Memory (literal) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFM instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory* on page C3-220.

For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

31 30	29 28 27	26	25 24	23 … 5	4 … 0
1 1	0 1 1	0	0 0	imm19	Rt
opc					

Literal variant

```
PRFM (<prfop>|#<imm5>), <label>
```

Decode for this encoding

```
integer t = UInt(Rt);
bits(64) offset;

offset = SignExtend(imm19:'00', 64);
```

Assembler symbols

`<prfop>` Is the prefetch operation, defined as `<type><target><policy>`.

`<type>` is one of:

- `PLD` Prefetch for load, encoded in the "Rt<4:3>" field as `0b00`.
- `PLI` Preload instructions, encoded in the "Rt<4:3>" field as `0b01`.
- `PST` Prefetch for store, encoded in the "Rt<4:3>" field as `0b10`.

`<target>` is one of:

- `L1` Level 1 cache, encoded in the "Rt<2:1>" field as `0b00`.
- `L2` Level 2 cache, encoded in the "Rt<2:1>" field as `0b01`.
- `L3` Level 3 cache, encoded in the "Rt<2:1>" field as `0b10`.

`<policy>` is one of:

- `KEEP` Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as `0`.
- `STRM` Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as `1`.

For more information on these prefetch operations, see *Prefetch memory* on page C3-220.

For other encodings of the "Rt" field, use `<imm5>`.

`<imm5>` Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field.

This syntax is only for encodings that are not accessible using `<prfop>`.

`<label>` Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
bits(64) address = PC[] + offset;

if HaveMTEExt() then
    SetTagCheckedInstruction(FALSE);

Prefetch(address, t<4:0>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.214 PRFM (register)

Prefetch Memory (register) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFM instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory* on page C3-220.

For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

31 30	29 28 27	26	25 24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
1 1	1 1 1	0	0 0	1 0	1	Rm	option	S	1 0	Rn	Rt
size				opc							

Integer variant

```
PRFM (<prfop>|#<imm5>), [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for this encoding

```
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 3 else 0;
```

Assembler symbols

`<prfop>` Is the prefetch operation, defined as `<type><target><policy>`.

`<type>` is one of:

- `PLD` Prefetch for load, encoded in the "Rt<4:3>" field as `0b00`.
- `PLI` Preload instructions, encoded in the "Rt<4:3>" field as `0b01`.
- `PST` Prefetch for store, encoded in the "Rt<4:3>" field as `0b10`.

`<target>` is one of:

- `L1` Level 1 cache, encoded in the "Rt<2:1>" field as `0b00`.
- `L2` Level 2 cache, encoded in the "Rt<2:1>" field as `0b01`.
- `L3` Level 3 cache, encoded in the "Rt<2:1>" field as `0b10`.

`<policy>` is one of:

- `KEEP` Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as `0`.
- `STRM` Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as `1`.

For more information on these prefetch operations, see *Prefetch memory* on page C3-220.

For other encodings of the "Rt" field, use `<imm5>`.

`<imm5>` Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field.

This syntax is only for encodings that are not accessible using `<prfop>`.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<Wm>` When `option<0>` is set to `0`, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in the "option" field. It can have the following values:

UXTW when option = 010

LSL when option = 011

SXTW when option = 110

SXTX when option = 111

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

#0 when S = 0

#3 when S = 1

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
    SetTagCheckedInstruction(FALSE);

bits(64) address;

if n == 31 then
    address = SP[];
else
    address = X[n];

address = address + offset;

Prefetch(address, t<4:0>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.215 PRFUM

Prefetch Memory (unscaled offset) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFUM instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory* on page C3-220.

For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
1 1	1 1 1	0	0 0	1 0	0	imm9	0 0	Rn	Rt
size				opc					

Unscaled offset variant

```
PRFUM (<prfop>|#<imm5>), [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<prfop>` Is the prefetch operation, defined as `<type><target><policy>`.

`<type>` is one of:

- `PLD` Prefetch for load, encoded in the "Rt<4:3>" field as `0b00`.
- `PLI` Preload instructions, encoded in the "Rt<4:3>" field as `0b01`.
- `PST` Prefetch for store, encoded in the "Rt<4:3>" field as `0b10`.

`<target>` is one of:

- `L1` Level 1 cache, encoded in the "Rt<2:1>" field as `0b00`.
- `L2` Level 2 cache, encoded in the "Rt<2:1>" field as `0b01`.
- `L3` Level 3 cache, encoded in the "Rt<2:1>" field as `0b10`.

`<policy>` is one of:

- `KEEP` Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as `0`.
- `STRM` Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as `1`.

For more information on these prefetch operations, see *Prefetch memory* on page C3-220.

For other encodings of the "Rt" field, use `<imm5>`.

`<imm5>` Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field.

This syntax is only for encodings that are not accessible using `<prfop>`.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<simm>` Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
```

Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(FALSE);

bits(64) address;

if n == 31 then
    address = SP[];
else
    address = X[n];

address = address + offset;

Prefetch(address, t<4:0>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.216 PSB CSYNC

Profiling Synchronization Barrier. This instruction is a barrier that ensures that all existing profiling data for the current PE has been formatted, and profiling buffer addresses have been translated such that all writes to the profiling buffer have been initiated. A following `DSB` instruction completes when the writes to the profiling buffer have completed.

If the Statistical Profiling Extension is not implemented, this instruction executes as a `NOP`.

ARMv8.2

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 8	7 5	4	3 2 1 0
1 1 0 1	0 1 0 1	0 0 0 0	0 1 1 0	0 0 1 0	0 0 1 0	0 0 1	1	1 1 1 1
					CRm	op2		

System variant

```
PSB CSYNC
```

Decode for this encoding

```
if !HaveStatisticalProfiling() then EndOfInstruction();
```

Operation

```
ProfilingSynchronizationBarrier();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.217 PSSBB

Physical Speculative Store Bypass Barrier is a memory barrier which prevents speculative loads from bypassing earlier stores to the same physical address.

The semantics of the Physical Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the PSSBB, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order before the PSSBB.
- When a load to a location appears in program order before the PSSBB, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order after the PSSBB.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11			8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	1	0	1	0	0	1	0	0	1	1	1	1	1
																					CRm				opc						

System variant

```
PSSBB
```

Decode for this encoding

```
// No additional decoding required
```

Operation

```
SpeculativeStoreBypassBarrierToPA();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.218 RBIT

Reverse Bits reverses the bit order in a register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
sf	1	0	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	Rn		Rd	

32-bit variant

Applies when `sf == 0`.

```
RBIT <Wd>, <Wn>
```

64-bit variant

Applies when `sf == 1`.

```
RBIT <Xd>, <Xn>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize = if sf == '1' then 64 else 32;
```

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
bits(datasize) operand = X[n];
bits(datasize) result;

for i = 0 to datasize-1
    result<datasize-1-i> = operand<i>;

X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.219 RET

Return from subroutine branches unconditionally to an address in a register, with a hint that this is a subroutine return.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
1	1	0	1	0	1	1	0	0	1	0	1	1	1	1	1	0	0	0	0	0	0	Rn		0 0 0 0 0	
							Z		op											A	M			Rm	

Integer variant

```
RET {<Xn>}
```

Decode for this encoding

```
integer n = UInt(Rn);
```

Assembler symbols

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field. Defaults to X30 if absent.

Operation

```
bits(64) target = X[n];

BranchTo(target, BranchType_RET);
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.220 RETAA, RETAB

Return from subroutine, with pointer authentication. This instruction authenticates the address that is held in LR, using SP as the modifier and the specified key, branches to the authenticated address, with a hint that this instruction is a subroutine return.

Key A is used for `RETAA`, and key B is used for `RETAB`.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to LR.

ARMv8.3

31 30 29 28	27 26 25 24	23	22 21 20	19 18 17 16	15 14 13 12	11	10	9	5	4	0
1 1 0 1	0 1 1 0	0	1 0	1 1 1 1	0 0 0 0	1	M	1 1 1 1 1		1	1 1 1 1 1
		Z	op			A		Rn			Rm

RETAA variant

Applies when `M == 0`.

```
RETAA
```

RETAB variant

Applies when `M == 1`.

```
RETAB
```

Decode for all variants of this encoding

```
boolean use_key_a = (M == '0');

if !HavePACExt() then
    UNDEFINED;
```

Operation

```
bits(64) target = X[30];

bits(64) modifier = SP[];

if use_key_a then
    target = AuthIA(target, modifier, TRUE);
else
    target = AuthIB(target, modifier, TRUE);

BranchTo(target, BranchType_RET);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.221 REV

Reverse Bytes reverses the byte order in a register.

This instruction is used by the pseudo-instruction REV64. The pseudo-instruction is never the preferred disassembly.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
sf	1	0	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	1	x	Rn		Rd	
																				opc					

32-bit variant

Applies when `sf == 0 && opc == 10`.

```
REV <Wd>, <Wn>
```

64-bit variant

Applies when `sf == 1 && opc == 11`.

```
REV <Xd>, <Xn>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize = if sf == '1' then 64 else 32;

integer container_size;
case opc of
    when '00'
        Unreachable();
    when '01'
        container_size = 16;
    when '10'
        container_size = 32;
    when '11'
        if sf == '0' then UNDEFINED;
        container_size = 64;
```

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
bits(datasize) operand = X[n];
bits(datasize) result;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV 8;
integer index = 0;
integer rev_index;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
for c = 0 to containers-1
    rev_index = index + ((elements_per_container - 1) * 8);
    for e = 0 to elements_per_container-1
        result<rev_index+7:rev_index> = operand<index+7:index>;
        index = index + 8;
        rev_index = rev_index - 8;

X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.222 REV16

Reverse bytes in 16-bit halfwords reverses the byte order in each 16-bit halfword of a register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
sf	1	0	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	Rn	Rd
																				opc			

32-bit variant

Applies when `sf == 0`.

```
REV16 <Wd>, <Wn>
```

64-bit variant

Applies when `sf == 1`.

```
REV16 <Xd>, <Xn>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize = if sf == '1' then 64 else 32;

integer container_size;
case opc of
    when '00'
        Unreachable();
    when '01'
        container_size = 16;
    when '10'
        container_size = 32;
    when '11'
        if sf == '0' then UNDEFINED;
        container_size = 64;
```

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
bits(datasize) operand = X[n];
bits(datasize) result;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV 8;
integer index = 0;
integer rev_index;
for c = 0 to containers-1
    rev_index = index + ((elements_per_container - 1) * 8);
    for e = 0 to elements_per_container-1
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        result<rev_index+7:rev_index> = operand<index+7:index>;
        index = index + 8;
        rev_index = rev_index - 8;

X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.223 REV32

Reverse bytes in 32-bit words reverses the byte order in each 32-bit word of a register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	Rn	Rd
sf																				opc			

64-bit variant

```
REV32 <Xd>, <Xn>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize = if sf == '1' then 64 else 32;

integer container_size;
case opc of
    when '00'
        Unreachable();
    when '01'
        container_size = 16;
    when '10'
        container_size = 32;
    when '11'
        if sf == '0' then UNDEFINED;
        container_size = 64;
```

Assembler symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
bits(datasize) operand = X[n];
bits(datasize) result;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV 8;
integer index = 0;
integer rev_index;
for c = 0 to containers-1
    rev_index = index + ((elements_per_container - 1) * 8);
    for e = 0 to elements_per_container-1
        result<rev_index+7:rev_index> = operand<index+7:index>;
        index = index + 8;
        rev_index = rev_index - 8;

X[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.224 REV64

Reverse Bytes reverses the byte order in a 64-bit general-purpose register.

When assembling for Armv8.2, an assembler must support this pseudo-instruction. It is OPTIONAL whether an assembler supports this pseudo-instruction when assembling for an architecture earlier than Armv8.2.

This instruction is a pseudo-instruction of the REV instruction. This means that:

- The encodings in this description are named to match the encodings of REV.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of REV gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	1	1	Rn	Rd
sf																				opc			

64-bit variant

```
REV64 <Xd>, <Xn>
```

is equivalent to

```
REV  <Xd>, <Xn>
```

and is never the preferred disassembly.

Assembler symbols

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of REV gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.225 RMIF

Performs a rotation right of a value held in a general purpose register by an immediate value, and then inserts a selection of the bottom four bits of the result of the rotation into the PSTATE flags, under the control of a second immediate mask.

ARMv8.4

31	30	29	28	27	26	25	24	23	22	21	20	15	14	13	12	11	10	9	5	4	3	0
1	0	1	1	1	0	1	0	0	0	0	imm6	0	0	0	0	1	Rn			0	mask	

sf

Integer variant

```
RMIF <Xn>, #<shift>, #<mask>
```

Decode for this encoding

```
if !HaveFlagManipulateExt() then UNDEFINED;
integer lsb = UInt(imm6);
integer n = UInt(Rn);
```

Assembler symbols

`<Xn>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
`<shift>`	Is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,
`<mask>`	Is the flag bit mask, an immediate in the range 0 to 15, which selects the bits that are inserted into the NZCV condition flags, encoded in the "mask" field.

Operation

```
bits(4) tmp;
bits(64) tmpreg = X[n];
tmp = (tmpreg:tmpreg)<lsb+3:lsb>;
if mask<3> == '1' then PSTATE.N = tmp<3>;
if mask<2> == '1' then PSTATE.Z = tmp<2>;
if mask<1> == '1' then PSTATE.C = tmp<1>;
if mask<0> == '1' then PSTATE.V = tmp<0>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.226 ROR (immediate)

Rotate right (immediate) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

This instruction is an alias of the EXTR instruction. This means that:

- The encodings in this description are named to match the encodings of EXTR.
- The description of EXTR gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20–16	15–10	9–5	4–0
sf	0	0	1	0	0	1	1	1	N	0	Rm	imms	Rn	Rd

32-bit variant

Applies when `sf == 0 && N == 0 && imms == 0xxxxx`.

```
ROR <Wd>, <Ws>, #<shift>
```

is equivalent to

```
EXTR <Wd>, <Ws>, <Ws>, #<shift>
```

and is the preferred disassembly when `Rn == Rm`.

64-bit variant

Applies when `sf == 1 && N == 1`.

```
ROR <Xd>, <Xs>, #<shift>
```

is equivalent to

```
EXTR <Xd>, <Xs>, <Xs>, #<shift>
```

and is the preferred disassembly when `Rn == Rm`.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Ws>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xs>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
`<shift>`	For the 32-bit variant: is the amount by which to rotate, in the range 0 to 31, encoded in the "imms" field. For the 64-bit variant: is the amount by which to rotate, in the range 0 to 63, encoded in the "imms" field.

Operation

The description of EXTR gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.227 ROR (register)

Rotate Right (register) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is an alias of the RORV instruction. This means that:

- The encodings in this description are named to match the encodings of RORV.
- The description of RORV gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
sf	0	0	1	1	0	1	0	1	1	0	Rm	0	0	1	0	1	1	Rn	Rd
																op2			

32-bit variant

Applies when `sf == 0`.

```
ROR <Wd>, <Wn>, <Wm>
```

is equivalent to

```
RORV <Wd>, <Wn>, <Wm>
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

```
ROR <Xd>, <Xn>, <Xm>
```

is equivalent to

```
RORV <Xd>, <Xn>, <Xm>
```

and is always the preferred disassembly.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Wm>`	Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

The description of RORV gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.228 RORV

Rotate Right Variable provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is used by the alias ROR (register). The alias is always the preferred disassembly.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
sf	0	0	1	1	0	1	0	1	1	0	Rm		0	0	1	0	1	1	Rn		Rd	
																	op2					

32-bit variant

Applies when `sf == 0`.

```
RORV <Wd>, <Wn>, <Wm>
```

64-bit variant

Applies when `sf == 1`.

```
RORV <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ShiftType shift_type = DecodeShift(op2);
```

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

```
bits(datasize) result;
bits(datasize) operand2 = X[m];

result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize);
X[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.229 SB

Speculation Barrier is a barrier that controls speculation.

The semantics of the Speculation Barrier are that the execution, until the barrier completes, of any instruction that appears later in the program order than the barrier:

- Cannot be performed speculatively to the extent that such speculation can be observed through side-channels as a result of control flow speculation or data value speculation.
- Can be speculatively executed as a result of predicting that a potentially exception generating instruction has not generated an exception.

In particular, any instruction that appears later in the program order than the barrier cannot cause a speculative allocation into any caching structure where the allocation of that entry could be indicative of any data value present in memory or in the registers.

The SB instruction:

- Cannot be speculatively executed as a result of control flow speculation or data value speculation.
- Can be speculatively executed as a result of predicting that a potentially exception generating instruction has not generated an exception. The potentially exception generating instruction can complete once it is known not to be speculative, and all data values generated by instructions appearing in program order before the SB instruction have their predicted values confirmed.

When the prediction of the instruction stream is not informed by data taken from the register outputs of the speculative execution of instructions appearing in program order after an uncompleted SB instruction, the SB instruction has no effect on the use of prediction resources to predict the instruction stream that is being fetched.

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 8	7	6 5	4	3 2 1 0
1 1 0 1	0 1 0 1	0 0 0 0	0 1 1	0 0 1 1	(0) (0) (0) (0)	1	1 1	1	1 1 1 1
					CRm		opc		

System variant

```
SB
```

Decode for this encoding

```
if !HaveSBExt() then UNDEFINED;
```

Operation

```
SpeculationBarrier();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.230 SBC

Subtract with Carry subtracts a register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register.

This instruction is used by the alias NGC. See *Alias conditions* for details of when each alias is preferred.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
sf	1	0	1	1	0	1	0	0	0	0	Rm		0	0	0	0	0	0	Rn		Rd	
	op	S																				

32-bit variant

Applies when `sf == 0`.

```
SBC <Wd>, <Wn>, <Wm>
```

64-bit variant

Applies when `sf == 1`.

```
SBC <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
```

Alias conditions

Alias	is preferred when
NGC	`Rn == '11111'`

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Wm>`	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];

operand2 = NOT(operand2);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
(result, -) = AddWithCarry(operand1, operand2, PSTATE.C);

X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.231 SBCS

Subtract with Carry, setting flags, subtracts a register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias NGCS. See *Alias conditions* for details of when each alias is preferred.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
sf	1	1	1	1	0	1	0	0	0	0	Rm		0	0	0	0	0	0	Rn		Rd	
	op	S																				

32-bit variant

Applies when `sf == 0`.

```
SBCS <Wd>, <Wn>, <Wm>
```

64-bit variant

Applies when `sf == 1`.

```
SBCS <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
```

Alias conditions

Alias	is preferred when
NGCS	`Rn == '11111'`

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Wm>`	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(4) nzcv;

operand2 = NOT(operand2);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
(result, nzcv) = AddWithCarry(operand1, operand2, PSTATE.C);

PSTATE.<N,Z,C,V> = nzcv;

X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.232 SBFIZ

Signed Bitfield Insert in Zeros copies a bitfield of `<width>` bits from the least significant bits of the source register to bit position `<lsb>` of the destination register, setting the destination bits below the bitfield to zero, and the bits above the bitfield to a copy of the most significant bit of the bitfield.

This instruction is an alias of the SBFM instruction. This means that:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

31	30 29	28 27 26 25 24	23	22 21	16 15	10 9	5 4	0
sf	0 0	1 0 0 1 1 0	N	immr	imms	Rn	Rd	
	opc							

32-bit variant

Applies when `sf == 0 && N == 0`.

```
SBFIZ <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
SBFM <Wd>, <Wn>, #(-<lsb> MOD 32), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

64-bit variant

Applies when `sf == 1 && N == 1`.

```
SBFIZ <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
SBFM <Xd>, <Xn>, #(-<lsb> MOD 64), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
`<lsb>`	For the 32-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31. For the 64-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.
`<width>`	For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>. For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.233 SBFM

Signed Bitfield Move is usually accessed via one of its aliases, which are always preferred for disassembly.

If <imms> is greater than or equal to <immr>, this copies a bitfield of (<imms>-<immr>+1) bits starting from bit position <immr> in the source register to the least significant bits of the destination register.

If <imms> is less than <immr>, this copies a bitfield of (<imms>+1) bits from the least significant bits of the source register to bit position (regsize-<immr>) of the destination register, where regsize is the destination register size of 32 or 64 bits.

In both cases the destination bits below the bitfield are set to zero, and the bits above the bitfield are set to a copy of the most significant bit of the bitfield.

This instruction is used by the aliases ASR (immediate), SBFIZ, SBFX, SXTB, SXTH, and SXTW. See *Alias conditions* on page C6-1196 for details of when each alias is preferred.

31	30 29	28 27 26 25 24	23	22 21 16	15 10	9 5	4 0
sf	0 0	1 0 0 1 1 0	N	immr	imms	Rn	Rd
	opc						

32-bit variant

Applies when `sf == 0 && N == 0`.

```
SBFM <Wd>, <Wn>, #<immr>, #<imms>
```

64-bit variant

Applies when `sf == 1 && N == 1`.

```
SBFM <Xd>, <Xn>, #<immr>, #<imms>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;

integer R;
integer S;
bits(datasize) wmask;
bits(datasize) tmask;

if sf == '1' && N != '1' then UNDEFINED;
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then UNDEFINED;

R = UInt(immr);
S = UInt(imms);
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Alias conditions

Alias	of variant	is preferred when
ASR (immediate)	32-bit	`imms == '011111'`
ASR (immediate)	64-bit	`imms == '111111'`
SBFIZ	-	`UInt(imms) < UInt(immr)`
SBFX	-	`BFXPreferred(sf, opc<1>, imms, immr)`
SXTB	-	`immr == '000000' && imms == '000111'`
SXTH	-	`immr == '000000' && imms == '001111'`
SXTW	-	`immr == '000000' && imms == '011111'`

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

`<immr>` For the 32-bit variant: is the right rotate amount, in the range 0 to 31, encoded in the "immr" field.

For the 64-bit variant: is the right rotate amount, in the range 0 to 63, encoded in the "immr" field.

`<imms>` For the 32-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field.

For the 64-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the "imms" field.

Operation

```
bits(datasize) src = X[n];

// perform bitfield move on low bits
bits(datasize) bot = ROR(src, R) AND wmask;

// determine extension bits (sign, zero or dest register)
bits(datasize) top = Replicate(src<S>);

// combine extension bits and result bits
X[d] = (top AND NOT(tmask)) OR (bot AND tmask);
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.234 SBFX

Signed Bitfield Extract copies a bitfield of `<width>` bits starting from bit position `<lsb>` in the source register to the least significant bits of the destination register, and sets destination bits above the bitfield to a copy of the most significant bit of the bitfield.

This instruction is an alias of the SBFM instruction. This means that:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

31	30 29	28 27 26 25 24	23	22 21 16	15 10	9 5	4 0
sf	0 0	1 0 0 1 1 0	N	immr	imms	Rn	Rd
	opc						

32-bit variant

Applies when `sf == 0 && N == 0`.

```
SBFX <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
SBFM <Wd>, <Wn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `BFXPreferred(sf, opc<1>, imms, immr)`.

64-bit variant

Applies when `sf == 1 && N == 1`.

```
SBFX <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
SBFM <Xd>, <Xn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `BFXPreferred(sf, opc<1>, imms, immr)`.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
`<lsb>`	For the 32-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31. For the 64-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.
`<width>`	For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>. For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.235 SDIV

Signed Divide divides a signed integer register value by another signed integer register value, and writes the result to the destination register. The condition flags are not affected.

31	30	29	28 27 26 25	24 23 22 21	20 … 16	15 14 13 12	11	10 9 … 5	4 … 0
sf	0	0	1 1 0 1	0 1 1 0	Rm	0 0 0 0	1	Rn	Rd
							o1		

32-bit variant

Applies when `sf == 0`.

```
SDIV <Wd>, <Wn>, <Wm>
```

64-bit variant

Applies when `sf == 1`.

```
SDIV <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
```

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Wm>`	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
integer result;

if IsZero(operand2) then
    result = 0;
else
    result = RoundTowardsZero(Real(Int(operand1, FALSE)) / Real(Int(operand2, FALSE)));

X[d] = result<datasize-1:0>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.236 SETF8, SETF16

Set the PSTATE.NZV flags based on the value in the specified general-purpose register. `SETF8` treats the value as an 8 bit value, and `SETF16` treats the value as an 16 bit value.

The PSTATE.C flag is not affected by these instructions.

ARMv8.4

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	3	2	1	0
0	0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	sz	0	0	1	0	Rn		0	1	1	0	1

sf

SETF8 variant

Applies when `sz == 0`.

```
SETF8 <Wn>
```

SETF16 variant

Applies when `sz == 1`.

```
SETF16 <Wn>
```

Decode for all variants of this encoding

```
if !HaveFlagManipulateExt() then UNDEFINED;
integer msb = if sz == '1' then 15 else 7;
integer n = UInt(Rn);
```

Assembler symbols

`<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
bits(32) tmpreg = X[n];
PSTATE.N = tmpreg<msb>;
PSTATE.Z = if (tmpreg<msb:0> == Zeros(msb + 1)) then '1' else '0';
PSTATE.V = tmpreg<msb+1> EOR tmpreg<msb>;
//PSTATE.C unchanged;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.237 SEV

Send Event is a hint instruction. It causes an event to be signaled to all PEs in the multiprocessor system. For more information, see *Wait for Event mechanism and Send event* on page D1-2397.

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 8	7 5	4	3 2 1 0
1 1 0 1	0 1 0 1	0 0 0 0	0 0 1 1	0 0 1 0	0 0 0 0	1 0 0	1	1 1 1 1
					CRm	op2		

System variant

```
SEV
```

Decode for this encoding

```
// Empty.
```

Operation

```
SendEvent();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.238 SEVL

Send Event Local is a hint instruction that causes an event to be signaled locally without requiring the event to be signaled to other PEs in the multiprocessor system. It can prime a wait-loop which starts with a `WFE` instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11			8	7		5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	1	1	1	1	1
																				CRm				op2							

System variant

```
SEVL
```

Decode for this encoding

```
// Empty.
```

Operation

```
SendEventLocal();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.239 SMADDL

Signed Multiply-Add Long multiplies two 32-bit register values, adds a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias SMULL. See *Alias conditions* for details of when each alias is preferred.

31	30 29	28 27 26 25	24	23	22 21	20 … 16	15	14 … 10	9 … 5	4 … 0
1	0 0	1 1 0 1	1	0	0 1	Rm	0	Ra	Rn	Rd
				U			o0			

64-bit variant

```
SMADDL <Xd>, <Wn>, <Wm>, <Xa>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
```

Alias conditions

Alias	is preferred when
SMULL	`Ra == '11111'`

Assembler symbols

`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
`<Wm>`	Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
`<Xa>`	Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation

```
bits(32) operand1 = X[n];
bits(32) operand2 = X[m];
bits(64) operand3 = X[a];

integer result;

result = Int(operand3, FALSE) + (Int(operand1, FALSE) * Int(operand2, FALSE));

X[d] = result<63:0>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.240 SMC

Secure Monitor Call causes an exception to EL3.

`SMC` is available only for software executing at EL1 or higher. It is UNDEFINED in EL0.

If the values of HCR_EL2.TSC and SCR_EL3.SMD are both 0, execution of an `SMC` instruction at EL1 or higher generates a Secure Monitor Call exception, recording it in ESR_ELx, using the EC value `0x17`, that is taken to EL3.

If the value of HCR_EL2.TSC is 1 and EL2 is enabled in the current Security state, execution of an `SMC` instruction at EL1 generates an exception that is taken to EL2, regardless of the value of SCR_EL3.SMD. For more information, see *Traps to EL2 of EL1 execution of SMC instructions* on page D1-2385.

If the value of HCR_EL2.TSC is 0 and the value of SCR_EL3.SMD is 1, the SMC instruction is UNDEFINED.

31 30 29 28	27 26 25 24	23 22 21 20	5 4	3 2 1 0
1 1 0 1	0 1 0 0	0 0 0	imm16	0 0 0 1 1

System variant

```
SMC #<imm>
```

Decode for this encoding

```
// Empty.
```

Assembler symbols

`<imm>` Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

Operation

```
AArch64.CheckForSMCUndefOrTrap(imm16);

if SCR_EL3.SMD == '1' then
    // SMC disabled
    UNDEFINED;
else
    AArch64.CallSecureMonitor(imm16);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.241 SMNEGL

Signed Multiply-Negate Long multiplies two 32-bit register values, negates the product, and writes the result to the 64-bit destination register.

This instruction is an alias of the SMSUBL instruction. This means that:

- The encodings in this description are named to match the encodings of SMSUBL.
- The description of SMSUBL gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	10	9	5	4	0
1	0	0	1	1	0	1	1	0	0	1	Rm		1	1	1	1	1	Rn	Rd
								U					o0		Ra				

64-bit variant

```
SMNEGL <Xd>, <Wn>, <Wm>
```

is equivalent to

```
SMSUBL <Xd>, <Wn>, <Wm>, XZR
```

and is always the preferred disassembly.

Assembler symbols

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of SMSUBL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.242 SMSUBL

Signed Multiply-Subtract Long multiplies two 32-bit register values, subtracts the product from a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias SMNEGL. See *Alias conditions* for details of when each alias is preferred.

31	30 29	28 27 26 25 24	23	22 21	20 … 16	15	14 … 10	9 … 5	4 … 0
1	0 0	1 1 0 1 1	0	0 1	Rm	1	Ra	Rn	Rd
			U			o0			

64-bit variant

```
SMSUBL <Xd>, <Wn>, <Wm>, <Xa>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
```

Alias conditions

Alias	is preferred when
SMNEGL	`Ra == '11111'`

Assembler symbols

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

`<Xa>` Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

Operation

```
bits(32) operand1 = X[n];
bits(32) operand2 = X[m];
bits(64) operand3 = X[a];

integer result;

result = Int(operand3, FALSE) - (Int(operand1, FALSE) * Int(operand2, FALSE));
X[d] = result<63:0>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.243 SMULH

Signed Multiply High multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.

31	30 29	28 27 26 25 24	23	22 21 20	16	15	14 10	9 5	4 0
1	0 0	1 1 0 1 1	0	1 0	Rm	0	(1) (1) (1) (1) (1)	Rn	Rd
			U				Ra		

64-bit variant

```
SMULH <Xd>, <Xn>, <Xm>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

```
bits(64) operand1 = X[n];
bits(64) operand2 = X[m];

integer result;

result = Int(operand1, FALSE) * Int(operand2, FALSE);

X[d] = result<127:64>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.244 SMULL

Signed Multiply Long multiplies two 32-bit register values, and writes the result to the 64-bit destination register.

This instruction is an alias of the SMADDL instruction. This means that:

- The encodings in this description are named to match the encodings of SMADDL.
- The description of SMADDL gives the operational pseudocode for this instruction.

31	30 29	28	27 26 25 24	23	22 21	20 … 16	15	14 … 10	9 … 5	4 … 0
1	0 0	1	1 0 1 1	0	0 1	Rm	0	1 1 1 1 1	Rn	Rd
				U			o0	Ra		

64-bit variant

```
SMULL <Xd>, <Wn>, <Wm>
```

is equivalent to

```
SMADDL <Xd>, <Wn>, <Wm>, XZR
```

and is always the preferred disassembly.

Assembler symbols

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of SMADDL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.245 SSBB

Speculative Store Bypass Barrier is a memory barrier which prevents speculative loads from bypassing earlier stores to the same virtual address under certain conditions.

The semantics of the Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the SSBB, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - — The store is to the same location as the load.
  - — The store uses the same virtual address as the load.
  - — The store appears in program order before the SSBB.
- When a load to a location appears in program order before the SSBB, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - — The store is to the same location as the load.
  - — The store uses the same virtual address as the load.
  - — The store appears in program order after the SSBB.

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 8	7	6 5	4	3 2 1 0
1 1 0 1	0 1 0 1	0 0 0 0	0 0 1 1	0 0 1 1	0 0 0 0	1	0 0	1	1 1 1 1
					CRm		opc		

System variant

```
SSBB
```

Decode for this encoding

```
// No additional decoding required
```

Operation

```
SpeculativeStoreBypassBarrierToVA();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.246 ST2G

Store Allocation Tags stores an Allocation Tag to two Tag granules of memory. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

Post-index

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
1	1	0	1	1	0	0	1	1	0	1	imm9	0	1	Xn	Xt

Post-index variant

```
ST2G <Xt|SP>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;
```

Pre-index

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
1	1	0	1	1	0	0	1	1	0	1	imm9	1	1	Xn	Xt

Pre-index variant

```
ST2G <Xt|SP>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;
```

Signed offset

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
1	1	0	1	1	0	0	1	1	0	1	imm9	1	0	Xn	Xt

Signed offset variant

```
ST2G <Xt|SP>, [<Xn|SP>{, #<simm>}]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;
```

Assembler symbols

`<Xt|SP>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
`<simm>`	Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

Operation for all encodings

```
bits(64) address;
bits(64) data = if t == 31 then SP[] else X[t];
bits(4) tag = AArch64.AllocationTagFromAddress(data);

SetTagCheckedInstruction(FALSE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

AArch64.MemTag[address, AccType_NORMAL] = tag;
AArch64.MemTag[address+TAG_GRANULE, AccType_NORMAL] = tag;

if writeback then
    if postindex then
        address = address + offset;

    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.247 STADDB, STADDLB

Atomic add on byte in memory, without return, atomically loads an 8-bit byte from memory, adds the value held in a register to it, and stores the result back to memory.

- `STADDB` has no memory ordering semantics.
- `STADDLB` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is an alias of the LDADDB, LDADDAB, LDADDALB, LDADDLB instruction. This means that:

- The encodings in this description are named to match the encodings of LDADDB, LDADDAB, LDADDALB, LDADDLB.
- The description of LDADDB, LDADDAB, LDADDALB, LDADDLB gives the operational pseudocode for this instruction.

ARMv8.1

31 30	29 28 27	26 25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	0	R	1	Rs	0	0 0 0	0 0	Rn	1 1 1 1 1
size			A					opc			Rt

No memory ordering variant

Applies when `R == 0`.

```
STADDB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADDB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release variant

Applies when `R == 1`.

```
STADDLB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADDLB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDADDB, LDADDAB, LDADDALB, LDADDLB gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.248 STADDH, STADDLH

Atomic add on halfword in memory, without return, atomically loads a 16-bit halfword from memory, adds the value held in a register to it, and stores the result back to memory.

- `STADDH` has no memory ordering semantics.
- `STADDLH` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is an alias of the LDADDH, LDADDAH, LDADDALH, LDADDLH instruction. This means that:

- The encodings in this description are named to match the encodings of LDADDH, LDADDAH, LDADDALH, LDADDLH.
- The description of LDADDH, LDADDAH, LDADDALH, LDADDLH gives the operational pseudocode for this instruction.

ARMv8.1

31 30	29 28 27	26 25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	0	R	1 Rs	0	0 0 0	0 0	Rn	1 1 1 1 1
size			A					opc			Rt

No memory ordering variant

Applies when `R == 0`.

```
STADDH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADDH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release variant

Applies when `R == 1`.

```
STADDLH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADDLH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDADDH, LDADDAH, LDADDALH, LDADDLH gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.249 STADD, STADDL

Atomic add on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory.

- `STADD` has no memory ordering semantics.
- `STADDL` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is an alias of the LDADD, LDADDA, LDADDAL, LDADDL instruction. This means that:

- The encodings in this description are named to match the encodings of LDADD, LDADDA, LDADDAL, LDADDL.
- The description of LDADD, LDADDA, LDADDAL, LDADDL gives the operational pseudocode for this instruction.

ARMv8.1

31 30	29 28	27	26	25	24	23	22	21	20 16	15	14 12	11	10	9 5	4 0
1 x	1 1	1	0	0	0	0	R	1	Rs	0	0 0 0	0	0	Rn	1 1 1 1 1
size						A					opc				Rt

32-bit LDADD alias variant

Applies when `size == 10 && R == 0`.

```
STADD <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADD <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

32-bit LDADDL alias variant

Applies when `size == 10 && R == 1`.

```
STADDL <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADDL <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDADD alias variant

Applies when `size == 11 && R == 0`.

```
STADD <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDADD <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

64-bit LDADDL alias variant

Applies when `size == 11 && R == 1`.

```
STADDL <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDADDL <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xs>`	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDADD, LDADDA, LDADDAL, LDADDL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.250 STCLRB, STCLRLB

Atomic bit clear on byte in memory, without return, atomically loads an 8-bit byte from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- `STCLRB` has no memory ordering semantics.
- `STCLRLB` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is an alias of the LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB instruction. This means that:

- The encodings in this description are named to match the encodings of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB.
- The description of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB gives the operational pseudocode for this instruction.

ARMv8.1

31 30	29 28 27	26 25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	0	R	1 Rs	0	0 0 1	0 0	Rn	1 1 1 1 1
size			A					opc			Rt

No memory ordering variant

Applies when `R == 0`.

```
STCLRB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLRB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release variant

Applies when `R == 1`.

```
STCLRLB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLRLB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.251 STCLRH, STCLRLH

Atomic bit clear on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- `STCLRH` has no memory ordering semantics.
- `STCLRLH` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is an alias of the LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH instruction. This means that:

- The encodings in this description are named to match the encodings of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH.
- The description of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH gives the operational pseudocode for this instruction.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	0	R	1	Rs	0	0 0 1	0 0	Rn	1 1 1 1 1
size				A					opc			Rt

No memory ordering variant

Applies when `R == 0`.

```
STCLRH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLRH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release variant

Applies when `R == 1`.

```
STCLRLH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLRLH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.252 STCLR, STCLRL

Atomic bit clear on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- `STCLR` has no memory ordering semantics.
- `STCLRL` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is an alias of the LDCLR, LDCLRA, LDCLRAL, LDCLRL instruction. This means that:

- The encodings in this description are named to match the encodings of LDCLR, LDCLRA, LDCLRAL, LDCLRL.
- The description of LDCLR, LDCLRA, LDCLRAL, LDCLRL gives the operational pseudocode for this instruction.

ARMv8.1

31 30	29 28 27	26 25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
1 x	1 1 1	0 0 0	0	R	1	Rs	0	0 0 1	0 0	Rn	1 1 1 1 1
size			A					opc			Rt

32-bit LDCLR alias variant

Applies when `size == 10 && R == 0`.

```
STCLR <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLR <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

32-bit LDCLRL alias variant

Applies when `size == 10 && R == 1`.

```
STCLRL <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLRL <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDCLR alias variant

Applies when `size == 11 && R == 0`.

```
STCLR <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDCLR <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

64-bit LDCLRL alias variant

Applies when `size == 11 && R == 1`.

```
STCLRL <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDCLRL <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xs>`	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDCLR, LDCLRA, LDCLRAL, LDCLRL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.253 STEORB, STEORLB

Atomic exclusive OR on byte in memory, without return, atomically loads an 8-bit byte from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

- `STEORB` has no memory ordering semantics.
- `STEORLB` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is an alias of the LDEORB, LDEORAB, LDEORALB, LDEORLB instruction. This means that:

- The encodings in this description are named to match the encodings of LDEORB, LDEORAB, LDEORALB, LDEORLB.
- The description of LDEORB, LDEORAB, LDEORALB, LDEORLB gives the operational pseudocode for this instruction.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	0	R	1	Rs	0	0 1 0	0 0	Rn	1 1 1 1 1
size				A					opc			Rt

No memory ordering variant

Applies when `R == 0`.

```
STEORB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEORB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release variant

Applies when `R == 1`.

```
STEORLB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEORLB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDEORB, LDEORAB, LDEORALB, LDEORLB gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.254 STEORH, STEORLH

Atomic exclusive OR on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

- `STEORH` has no memory ordering semantics.
- `STEORLH` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is an alias of the LDEORH, LDEORAH, LDEORALH, LDEORLH instruction. This means that:

- The encodings in this description are named to match the encodings of LDEORH, LDEORAH, LDEORALH, LDEORLH.
- The description of LDEORH, LDEORAH, LDEORALH, LDEORLH gives the operational pseudocode for this instruction.

ARMv8.1

31 30	29 28 27	26 25 24	23	22	21	20 16	15	14 13 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	0	R	1 Rs	0	0 1 0	0 0	Rn	1 1 1 1 1
size			A					opc			Rt

No memory ordering variant

Applies when `R == 0`.

```
STEORH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEORH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release variant

Applies when `R == 1`.

```
STEORLH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEORLH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDEORH, LDEORAH, LDEORALH, LDEORLH gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.255 STEOR, STEORL

Atomic exclusive OR on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

- `STEOR` has no memory ordering semantics.
- `STEORL` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is an alias of the LDEOR, LDEORA, LDEORAL, LDEORL instruction. This means that:

- The encodings in this description are named to match the encodings of LDEOR, LDEORA, LDEORAL, LDEORL.
- The description of LDEOR, LDEORA, LDEORAL, LDEORL gives the operational pseudocode for this instruction.

ARMv8.1

31 30	29 28 27	26 25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
1 x	1 1 1	0 0 0	0	R	1	Rs	0	0 1 0	0 0	Rn	1 1 1 1 1
size			A				opc				Rt

32-bit LDEOR alias variant

Applies when `size == 10 && R == 0`.

```
STEOR <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEOR <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

32-bit LDEORL alias variant

Applies when `size == 10 && R == 1`.

```
STEORL <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEORL <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDEOR alias variant

Applies when `size == 11 && R == 0`.

```
STEOR <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDEOR <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

64-bit LDEORL alias variant

Applies when `size == 11 && R == 1`.

```
STEORL <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDEORL <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xs>`	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDEOR, LDEORA, LDEORAL, LDEORL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.256 STG

Store Allocation Tag stores an Allocation Tag to memory. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

Post-index

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
1	1	0	1	1	0	0	1	0	0	1	imm9	0	1	Xn	Xt

Post-index variant

```
STG <Xt|SP>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;
```

Pre-index

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
1	1	0	1	1	0	0	1	0	0	1	imm9	1	1	Xn	Xt

Pre-index variant

```
STG <Xt|SP>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;
```

Signed offset

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
1	1	0	1	1	0	0	1	0	0	1	imm9	1	0	Xn	Xt

Signed offset variant

```
STG <Xt|SP>, [<Xn|SP>{, #<simm>}]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;
```

Assembler symbols

`<Xt|SP>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
`<simm>`	Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

Operation for all encodings

```
bits(64) address;

SetTagCheckedInstruction(FALSE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

bits(64) data = if t == 31 then SP[] else X[t];
bits(4) tag = AArch64.AllocationTagFromAddress(data);
AArch64.MemTag[address, AccType_NORMAL] = tag;

if writeback then
    if postindex then
        address = address + offset;

    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.257 STGM

Store Tag Multiple writes a naturally aligned block of N Allocation Tags, where the size of N is identified in GMID_EL1.BS, and the Allocation Tag written to address A is taken from the source register at 4*A<7:4>+3:4*A<7:4>.

This instruction is UNDEFINED at EL0.

This instruction generates an Unchecked access.

If ID_AA64PFR1_EL1.MTE != 0b0010, this instruction is UNDEFINED.

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
1	1	0	1	1	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	Xn		Xt	

Integer variant

```
STGM <Xt>, [<Xn|SP>]
```

Decode for this encoding

```
integer t = UInt(Xt);
integer n = UInt(Xn);
```

Assembler symbols

`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

Operation

```
if PSTATE.EL == EL0 then
    UNDEFINED;

bits(64) data = X[t];
bits(64) address;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

integer size = 4 * (2 ^ (UInt(GMID_EL1.BS)));
address = Align(address, size);
integer count = size >> LOG2_TAG_GRANULE;
integer index = UInt(address<LOG2_TAG_GRANULE+3:LOG2_TAG_GRANULE>);

for i = 0 to count-1
    bits(4) tag = data<(index*4)+3:index*4>;
    AArch64.MemTag[address, AccType_NORMAL] = tag;
    address = address + TAG_GRANULE;
    index = index + 1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.258 STGP

Store Allocation Tag and Pair of registers stores an Allocation Tag and two 64-bit doublewords to memory, from two registers. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the base register.

This instruction generates an Unchecked access.

Post-index

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21	15	14	10	9	5	4	0
0	1	1	0	1	0	0	0	1	0	simm7	Xt2			Xn		Xt	

Post-index variant

```
STGP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

Decode for this encoding

```
integer n = UInt(Xn);
integer t = UInt(Xt);
integer t2 = UInt(Xt2);
bits(64) offset = LSL(SignExtend(simm7, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;
```

Pre-index

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21	15	14	10	9	5	4	0
0	1	1	0	1	0	0	1	1	0	simm7	Xt2			Xn		Xt	

Pre-index variant

```
STGP <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

Decode for this encoding

```
integer n = UInt(Xn);
integer t = UInt(Xt);
integer t2 = UInt(Xt2);
bits(64) offset = LSL(SignExtend(simm7, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;
```

Signed offset

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21	15	14	10	9	5	4	0
0	1	1	0	1	0	0	1	0	0	simm7	Xt2			Xn		Xt	

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Signed offset variant

```
STGP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
integer n = UInt(Xn);
integer t = UInt(Xt);
integer t2 = UInt(Xt2);
bits(64) offset = LSL(SignExtend(simm7, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;
```

Assembler symbols

<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Xt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Xt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
<imm>	For the post-index and pre-index variant: is the signed immediate offset, a multiple of 16 in the range -1024 to 1008, encoded in the "simm7" field. For the signed offset variant: is the optional signed immediate offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "simm7" field.

Operation for all encodings

```
bits(64) address;
bits(64) data1;
bits(64) data2;

SetTagCheckedInstruction(FALSE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data1 = X[t];
data2 = X[t2];

if !postindex then
    address = address + offset;

Mem[address, 8, AccType_NORMAL] = data1;
Mem[address+8, 8, AccType_NORMAL] = data2;

AArch64.MemTag[address, AccType_NORMAL] = AArch64.AllocationTagFromAddress(address);

if writeback then
    if postindex then
        address = address + offset;

    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.259 STLLRB

Store LORelease Register Byte stores a byte from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *LoadLOAcquire, StoreLORelease* on page B2-141. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

ARMv8.1

31 30	29 28	27 26	25 24	23	22	21	20 16	15	14 10	9 5	4 0
0 0	0 0	1 0	0 0	1	0	0	(1)(1)(1)(1)(1)	0	(1)(1)(1)(1)(1)	Rn	Rt
size					L		Rs	o0	Rt2		

No offset variant

```
STLLRB <Wt>, [<Xn|SP>{,#0}]
```

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = X[t];
Mem[address, 1, AccType_LIMITEDORDERED] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.260 STLLRH

Store LORelease Register Halfword stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *LoadLOAcquire, StoreLORelease* on page B2-141. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

ARMv8.1

31 30	29 28 27	26 25 24	23	22	21	20 16	15	14 10	9 5	4 0
0 1	0 0 1	0 0 0	1	0	0	(1)(1)(1)(1)(1)	0	(1)(1)(1)(1)(1)	Rn	Rt
size				L		Rs	o0	Rt2		

No offset variant

```
STLLRH <Wt>, [<Xn|SP>{,#0}]
```

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = X[t];
Mem[address, 2, AccType_LIMITEDORDERED] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.261 STLLR

Store LORelease Register stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The instruction also has memory ordering semantics as described in *LoadLOAcquire, StoreLORelease* on page B2-141. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	10	9	5	4	0
1	x	0	0	1	0	0	0	1	0	0	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	Rn	Rt
size									L		Rs		o0	Rt2					

32-bit variant

Applies when `size == 10`.

```
STLLR <Wt>, [<Xn|SP>{,#0}]
```

64-bit variant

Applies when `size == 11`.

```
STLLR <Xt>, [<Xn|SP>{,#0}]
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << UInt(size);
boolean tag_checked = n != 31;
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = X[t];
Mem[address, dbytes, AccType_LIMITEDORDERED] = data;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.262 STLR

Store-Release Register stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

31 30	29 28 27 26 25 24	23	22	21	20 16	15	14 10	9 5	4 0
1 x	0 0 1 0 0 0	1	0	0	(1)(1)(1)(1)(1)	1	(1)(1)(1)(1)(1)	Rn	Rt
size			L		Rs	o0	Rt2		

32-bit variant

Applies when `size == 10`.

```
STLR <Wt>, [<Xn|SP>{,#0}]
```

64-bit variant

Applies when `size == 11`.

```
STLR <Xt>, [<Xn|SP>{,#0}]
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << UInt(size);
boolean tag_checked = n != 31;
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = X[t];
Mem[address, dbytes, AccType_ORDERED] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.263 STLRB

Store-Release Register Byte stores a byte from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

31 30	29 28 27 26 25 24	23	22	21	20 16	15	14 10	9 5	4 0
0 0	0 0 1 0 0 0	1	0	0	(1)(1)(1)(1)(1)	1	(1)(1)(1)(1)(1)	Rn	Rt
size			L		Rs	o0	Rt2		

No offset variant

```
STLRB <Wt>, [<Xn|SP>{,#0}]
```

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = X[t];
Mem[address, 1, AccType_ORDERED] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.264 STLRH

Store-Release Register Halfword stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

31 30	29 28 27 26 25 24	23	22	21	20 16	15	14 10	9 5	4 0
0 1	0 0 1 0 0 0	1	0	0	(1)(1)(1)(1)(1)	1	(1)(1)(1)(1)(1)	Rn	Rt
size			L		Rs	o0	Rt2		

No offset variant

```
STLRH <Wt>, [<Xn|SP>{,#0}]
```

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = X[t];
Mem[address, 2, AccType_ORDERED] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.265 STLUR

Store-Release Register (unscaled) calculates an address from a base register value and an immediate offset, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140

For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

ARMv8.4

31 30	29 28 27 26 25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
1 x	0 1 1 0 0 1	0 0	0	imm9	0 0	Rn	Rt
size		opc					

32-bit variant

Applies when `size == 10`.

```
STLUR <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit variant

Applies when `size == 11`.

```
STLUR <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer datasize = 8 << scale;
boolean tag_checked = n != 31;
```

Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(datasize) data;

if n == 31 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = X[t];
Mem[address, datasize DIV 8, AccType_ORDERED] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.266 STLURB

Store-Release Register Byte (unscaled) calculates an address from a base register value and an immediate offset, and stores a byte to the calculated address, from a 32-bit register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140

For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

ARMv8.4

31 30	29 28 27 26 25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 0	0 1 1 0 0 1	0 0	0	imm9	0 0	Rn	Rt
size		opc					

Unscaled offset variant

```
STLURB <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = X[t];
Mem[address, 1, AccType_ORDERED] = data;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.267 STLURH

Store-Release Register Halfword (unscaled) calculates an address from a base register value and an immediate offset, and stores a halfword to the calculated address, from a 32-bit register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140

For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

ARMv8.4

31 30	29 28 27 26 25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 1	0 1 1 0 0 1	0 0	0	imm9	0 0	Rn	Rt
size		opc					

Unscaled offset variant

```
STLURH <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = X[t];
Mem[address, 2, AccType_ORDERED] = data;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.268 STLXP

Store-Release Exclusive Pair of registers stores two 32-bit words or two 64-bit doublewords to a memory location if the PE has exclusive access to the memory address, from two registers, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores* on page B2-167. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and, if the Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. The instruction also has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140. For information about memory accesses see *Load/Store addressing modes* on page C1-189.

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14 … 10	9 … 5	4 … 0
1	sz	0	0	1	0	0	0	0	0	1	Rs	1	Rt2	Rn	Rt
									L			o0			

32-bit variant

Applies when `sz == 0`.

```
STLXP <Ws>, <Wt1>, <Wt2>, [<Xn|SP>{,#0}]
```

64-bit variant

Applies when `sz == 1`.

```
STLXP <Ws>, <Xt1>, <Xt2>, [<Xn|SP>{,#0}]
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);    // ignored by load/store single register
integer s = UInt(Rs);    // ignored by all loads and store-release

integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2;
boolean tag_checked = n != 31;
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *STLXP* on page K1-7867.

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

- `0` If the operation updates memory.
- `1` If the operation fails to update memory.

`<Xt1>` Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

`<Xt2>` Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

`<Wt1>` Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

`<Wt2>` Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Ws>` is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch64.ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch64.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if s == t || (s == t2) then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_NONE    rt_unknown = FALSE;    // store original value
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_NONE    rn_unknown = FALSE;    // address is original base
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    bits(datasize DIV 2) el1 = X[t];
    bits(datasize DIV 2) el2 = X[t2];
    data = if BigEndian() then el1:el2 else el2:el1;
bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, dbytes) then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, AccType_ORDEREDATOMIC] = data;
    status = ExclusiveMonitorsStatus();
X[s] = ZeroExtend(status, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.269 STLXR

Store-Release Exclusive Register stores a 32-bit word or a 64-bit doubleword to memory if the PE has exclusive access to the memory address, from two registers, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores* on page B2-167. The memory access is atomic. The instruction also has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140. For information about memory accesses see *Load/Store addressing modes* on page C1-189.

31 30	29 28	27 26	25 24	23	22	21	20 16	15	14 10	9 5	4 0
1 x	0 0	1 0	0 0	0	0	0	Rs	1	(1)(1)(1)(1)(1)	Rn	Rt
size					L			o0	Rt2		

32-bit variant

Applies when `size == 10`.

```
STLXR <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

64-bit variant

Applies when `size == 11`.

```
STLXR <Ws>, <Xt>, [<Xn|SP>{,#0}]
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);    // ignored by all loads and store-release

integer elsize = 8 << UInt(size);
boolean tag_checked = n != 31;
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *STLXR* on page K1-7867.

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

- `0` If the operation updates memory.
- `1` If the operation fails to update memory.

`<Xt>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Ws>` is not updated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch64.ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch64.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if s == t then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_NONE    rt_unknown = FALSE;    // store original value
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_NONE    rn_unknown = FALSE;    // address is original base
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(elsize) UNKNOWN;
else
    data = X[t];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, dbytes) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, AccType_ORDEREDATOMIC] = data;
    status = ExclusiveMonitorsStatus();
X[s] = ZeroExtend(status, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.270 STLXRB

Store-Release Exclusive Register Byte stores a byte from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores* on page B2-167. The memory access is atomic. The instruction also has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140. For information about memory accesses see *Load/Store addressing modes* on page C1-189.

31 30	29 28	27 26	25 24	23	22	21	20 16	15	14 10	9 5	4 0
0 0	0 0	1 0	0 0	0	0	0	Rs	1	(1)(1)(1)(1)(1)	Rn	Rt
size					L			o0	Rt2		

No offset variant

```
STLXRB <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);    // ignored by all loads and store-release

boolean tag_checked = n != 31;
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *STLXRB* on page K1-7867.

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

- `0` If the operation updates memory.
- `1` If the operation fails to update memory.

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Ws>` is not updated.

If `AArch64.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(8) data;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExt() then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    SetTagCheckedInstruction(tag_checked);

if s == t then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_NONE    rt_unknown = FALSE;    // store original value
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_NONE    rn_unknown = FALSE;    // address is original base
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = X[t];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 1) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 1, AccType_ORDEREDATOMIC] = data;
    status = ExclusiveMonitorsStatus();
X[s] = ZeroExtend(status, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.271 STLXRH

Store-Release Exclusive Register Halfword stores a halfword from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores* on page B2-167. The memory access is atomic. The instruction also has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140. For information about memory accesses see *Load/Store addressing modes* on page C1-189.

31 30	29 28 27 26 25 24	23	22	21	20 16	15	14 10	9 5	4 0
0 1	0 0 1 0 0 0	0	0	0	Rs	1	(1) (1) (1) (1) (1)	Rn	Rt
size			L			o0	Rt2		

No offset variant

```
STLXRH <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);    // ignored by all loads and store-release

boolean tag_checked = n != 31;
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *STLXRH* on page K1-7868.

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

- `0` If the operation updates memory.
- `1` If the operation fails to update memory.

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Ws>` is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch64.ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch64.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
bits(64) address;
bits(16) data;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if s == t then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_NONE    rt_unknown = FALSE;    // store original value
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_NONE    rn_unknown = FALSE;    // address is original base
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 2) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 2, AccType_ORDEREDATOMIC] = data;
    status = ExclusiveMonitorsStatus();
X[s] = ZeroExtend(status, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.272 STNP

Store Pair of Registers, with non-temporal hint, calculates an address from a base register value and an immediate offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information about memory accesses, see *Load/Store addressing modes* on page C1-189. For information about Non-temporal pair instructions, see *Load/Store Non-temporal Pair* on page C3-212.

31 30	29 28 27	26	25 24 23	22	21 15	14 10	9 5	4 0
x 0	1 0 1	0	0 0 0	0	imm7	Rt2	Rn	Rt
opc				L				

32-bit variant

Applies when `opc == 00`.

```
STNP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

64-bit variant

Applies when `opc == 10`.

```
STNP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for all variants of this encoding

```
// Empty.
```

Assembler symbols

`<Wt1>` Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

`<Wt2>` Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

`<Xt1>` Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

`<Xt2>` Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<imm>` For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc<0> == '1' then UNDEFINED;
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = n != 31;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data1 = X[t];
data2 = X[t2];
Mem[address, dbytes, AccType_STREAM] = data1;
Mem[address+dbytes, dbytes, AccType_STREAM] = data2;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.273 STP

Store Pair of Registers calculates an address from a base register value and an immediate offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

Post-index

31 30	29 28 27	26	25 24 23	22	21 … 15	14 … 10	9 … 5	4 … 0
x 0	1 0 1	0	0 0 1	0	imm7	Rt2	Rn	Rt
opc				L				

32-bit variant

Applies when `opc == 00`.

```
STP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>
```

64-bit variant

Applies when `opc == 10`.

```
STP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
```

Pre-index

31 30	29 28 27	26	25 24 23	22	21 … 15	14 … 10	9 … 5	4 … 0
x 0	1 0 1	0	0 1 1	0	imm7	Rt2	Rn	Rt
opc				L				

32-bit variant

Applies when `opc == 00`.

```
STP <Wt1>, <Wt2>, [<Xn|SP>, #<imm>]!
```

64-bit variant

Applies when `opc == 10`.

```
STP <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
```

Signed offset

31 30	29 28 27	26	25 24 23	22	21 … 15	14 … 10	9 … 5	4 … 0
x 0	1 0 1	0	0 1 0	0	imm7	Rt2	Rn	Rt
opc				L				

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

32-bit variant

Applies when `opc == 00`.

```
STP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

64-bit variant

Applies when `opc == 10`.

```
STP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for all variants of this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *STP* on page K1-7866.

Assembler symbols

`<Wt1>` Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

`<Wt2>` Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

`<Xt1>` Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

`<Xt2>` Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<imm>` For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.

For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8.

For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if L:opc<0> == '01' || opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = wback || n != 31;
```

Operation for all encodings

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
boolean rt_unknown = FALSE;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // value stored is pre-writeback
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

if rt_unknown && t == n then
    data1 = bits(datasize) UNKNOWN;
else
    data1 = X[t];
if rt_unknown && t2 == n then
    data2 = bits(datasize) UNKNOWN;
else
    data2 = X[t2];
Mem[address, dbytes, AccType_NORMAL] = data1;
Mem[address+dbytes, dbytes, AccType_NORMAL] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.274 STR (immediate)

Store Register (immediate) stores a word or a doubleword from a register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

Post-index

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
1	x	1	1	1	0	0	0	0	0	0	imm9	0	1	Rn	Rt
size								opc							

32-bit variant

Applies when `size == 10`.

```
STR <Wt>, [<Xn|SP>], #<simm>
```

64-bit variant

Applies when `size == 11`.

```
STR <Xt>, [<Xn|SP>], #<simm>
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
1	x	1	1	1	0	0	0	0	0	0	imm9	1	1	Rn	Rt
size								opc							

32-bit variant

Applies when `size == 10`.

```
STR <Wt>, [<Xn|SP>, #<simm>]!
```

64-bit variant

Applies when `size == 11`.

```
STR <Xt>, [<Xn|SP>, #<simm>]!
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Unsigned offset

31 30	29 28 27	26	25 24	23 22	21 … 10	9 … 5	4 … 0
1 x	1 1 1	0	0 1	0 0	imm12	Rn	Rt
size				opc			

32-bit variant

Applies when `size == 10`.

```
STR <Wt>, [<Xn|SP>{, #<pimm>}]
```

64-bit variant

Applies when `size == 11`.

```
STR <Xt>, [<Xn|SP>{, #<pimm>}]
```

Decode for all variants of this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
`<pimm>`	For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4. For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer datasize = 8 << scale;
boolean tag_checked = wback || n != 31;
```

Operation for all encodings

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(datasize) data;

boolean rt_unknown = FALSE;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // value stored is original value
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    data = X[t];
Mem[address, datasize DIV 8, AccType_NORMAL] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.275 STR (register)

Store Register (register) calculates an address from a base register value and an offset register value, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

31 30	29 28 27	26 25 24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
1 x	1 1 1	0 0 0	0 0	1	Rm	option	S	1 0	Rn	Rt
size			opc							

32-bit variant

Applies when `size == 10`.

```
STR <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

64-bit variant

Applies when `size == 11`.

```
STR <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for all variants of this encoding

```
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xt>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<Wm>` When `option<0>` is set to `0`, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

`<Xm>` When `option<0>` is set to `1`, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

`<extend>` Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in the "option" field. It can have the following values:

- `UXTW` when `option = 010`
- `LSL` when `option = 011`
- `SXTW` when `option = 110`
- `SXTX` when `option = 111`

`<amount>` For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

- `#0` when `S = 0`
- `#2` when `S = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

`#0` when `S = 0`

`#3` when `S = 1`

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);

integer datasize = 8 << scale;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
    SetTagCheckedInstruction(TRUE);

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = X[t];
Mem[address, datasize DIV 8, AccType_NORMAL] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.276 STRB (immediate)

Store Register Byte (immediate) stores the least significant byte of a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

Post-index

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 0	1 1 1	0	0 0	0 0	0	imm9	0 1	Rn	Rt
size				opc					

Post-index variant

```
STRB <Wt>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 0	1 1 1	0	0 0	0 0	0	imm9	1 1	Rn	Rt
size				opc					

Pre-index variant

```
STRB <Wt>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31 30	29 28 27	26	25 24	23 22	21	20 … 10	9 … 5	4 … 0
0 0	1 1 1	0	0 1	0 0		imm12	Rn	Rt
size				opc				

Unsigned offset variant

```
STRB <Wt>, [<Xn|SP>{, #<pimm>}]
```

Decode for this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *STRB (immediate)* on page K1-7868.

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
`<pimm>`	Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = wback || n != 31;
```

Operation for all encodings

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;

boolean rt_unknown = FALSE;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = X[t];
Mem[address, 1, AccType_NORMAL] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.277 STRB (register)

Store Register Byte (register) calculates an address from a base register value and an offset register value, and stores a byte from a 32-bit register to the calculated address. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

31 30	29 28 27	26	25 24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	0 0	1	Rm	option	S	1 0	Rn	Rt
size				opc							

Extended register variant

Applies when `option != 011`.

```
STRB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

Shifted register variant

Applies when `option == 011`.

```
STRB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

Decode for all variants of this encoding

```
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
```

Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<Wm>` When `option<0>` is set to `0`, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

`<Xm>` When `option<0>` is set to `1`, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

`<extend>` Is the index extend specifier, encoded in the "option" field. It can have the following values:

- `UXTW` when `option = 010`
- `SXTW` when `option = 110`
- `SXTX` when `option = 111`

`<amount>` Is the index shift amount, it must be `#0`, encoded in "S" as `0` if omitted, or as `1` if present.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
bits(64) offset = ExtendReg(m, extend_type, 0);
if HaveMTEExt() then
    SetTagCheckedInstruction(TRUE);

bits(64) address;
bits(8) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = X[t];
Mem[address, 1, AccType_NORMAL] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.278 STRH (immediate)

Store Register Halfword (immediate) stores the least significant halfword of a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

Post-index

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 1	1 1 1	0	0 0	0 0	0	imm9	0 1	Rn	Rt
size				opc					

Post-index variant

```
STRH <Wt>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 1	1 1 1	0	0 0	0 0	0	imm9	1 1	Rn	Rt
size				opc					

Pre-index variant

```
STRH <Wt>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31 30	29 28 27	26	25 24	23 22	21 … 10	9 … 5	4 … 0
0 1	1 1 1	0	0 1	0 0	imm12	Rn	Rt
size				opc			

Unsigned offset variant

```
STRH <Wt>, [<Xn|SP>{, #<pimm>}]
```

Decode for this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *STRH (immediate)* on page K1-7869.

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
`<pimm>`	Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = wback || n != 31;
```

Operation for all encodings

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;

boolean rt_unknown = FALSE;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t];
Mem[address, 2, AccType_NORMAL] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.279 STRH (register)

Store Register Halfword (register) calculates an address from a base register value and an offset register value, and stores a halfword from a 32-bit register to the calculated address. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

31 30	29 28 27	26 25 24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	0 0	1	Rm	option	S	1 0	Rn	Rt
size				opc						

32-bit variant

```
STRH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for this encoding

```
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 1 else 0;
```

Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<Wm>` When `option<0>` is set to `0`, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

`<Xm>` When `option<0>` is set to `1`, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

`<extend>` Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in the "option" field. It can have the following values:

- `UXTW` when `option = 010`
- `LSL` when `option = 011`
- `SXTW` when `option = 110`
- `SXTX` when `option = 111`

`<amount>` Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

- `#0` when `S = 0`
- `#1` when `S = 1`

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
    SetTagCheckedInstruction(TRUE);

bits(64) address;
bits(16) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = X[t];
Mem[address, 2, AccType_NORMAL] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.280 STSETB, STSETLB

Atomic bit set on byte in memory, without return, atomically loads an 8-bit byte from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- `STSETB` has no memory ordering semantics.
- `STSETLB` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is an alias of the LDSETB, LDSETAB, LDSETALB, LDSETLB instruction. This means that:

- The encodings in this description are named to match the encodings of LDSETB, LDSETAB, LDSETALB, LDSETLB.
- The description of LDSETB, LDSETAB, LDSETALB, LDSETLB gives the operational pseudocode for this instruction.

ARMv8.1

31 30	29 28 27	26 25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0 0 0	0	R	1	Rs	0	0 1 1	0 0	Rn	1 1 1 1 1
size			A					opc			Rt

No memory ordering variant

Applies when `R == 0`.

```
STSETB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSETB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release variant

Applies when `R == 1`.

```
STSETLB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSETLB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSETB, LDSETAB, LDSETALB, LDSETLB gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.281 STSETH, STSETLH

Atomic bit set on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- `STSETH` has no memory ordering semantics.
- `STSETLH` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is an alias of the LDSETH, LDSETAH, LDSETALH, LDSETLH instruction. This means that:

- The encodings in this description are named to match the encodings of LDSETH, LDSETAH, LDSETALH, LDSETLH.
- The description of LDSETH, LDSETAH, LDSETALH, LDSETLH gives the operational pseudocode for this instruction.

ARMv8.1

31 30	29 28 27	26 25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	0	R	1 Rs	0	0 1 1	0 0	Rn	1 1 1 1 1
size			A					opc			Rt

No memory ordering variant

Applies when `R == 0`.

```
STSETH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSETH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release variant

Applies when `R == 1`.

```
STSETLH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSETLH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSETH, LDSETAH, LDSETALH, LDSETLH gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.282 STSET, STSETL

Atomic bit set on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- `STSET` has no memory ordering semantics.
- `STSETL` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is an alias of the LDSET, LDSETA, LDSETAL, LDSETL instruction. This means that:

- The encodings in this description are named to match the encodings of LDSET, LDSETA, LDSETAL, LDSETL.
- The description of LDSET, LDSETA, LDSETAL, LDSETL gives the operational pseudocode for this instruction.

ARMv8.1

31 30	29 28 27	26 25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
1 x	1 1 1	0 0 0	0	R	1	Rs	0	0 1 1	0 0	Rn	1 1 1 1 1
size			A					opc			Rt

32-bit LDSET alias variant

Applies when `size == 10 && R == 0`.

```
STSET <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSET <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

32-bit LDSETL alias variant

Applies when `size == 10 && R == 1`.

```
STSETL <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSETL <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDSET alias variant

Applies when `size == 11 && R == 0`.

```
STSET <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDSET <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

64-bit LDSETL alias variant

Applies when `size == 11 && R == 1`.

```
STSETL <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDSETL <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xs>`	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSET, LDSETA, LDSETAL, LDSETL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.283 STSMAXB, STSMAXLB

Atomic signed maximum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- `STSMAXB` has no memory ordering semantics.
- `STSMAXLB` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is an alias of the LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB instruction. This means that:

- The encodings in this description are named to match the encodings of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB.
- The description of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB gives the operational pseudocode for this instruction.

ARMv8.1

31 30	29 28 27	26 25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0 0 0	0	R	1	Rs	0	1 0 0	0 0	Rn	1 1 1 1 1
size			A				opc				Rt

No memory ordering variant

Applies when `R == 0`.

```
STSMAXB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release variant

Applies when `R == 1`.

```
STSMAXLB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXLB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

The description of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.284 STSMAXH, STSMAXLH

Atomic signed maximum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- `STSMAXH` has no memory ordering semantics.
- `STSMAXLH` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is an alias of the LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH instruction. This means that:

- The encodings in this description are named to match the encodings of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH.
- The description of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH gives the operational pseudocode for this instruction.

ARMv8.1

31 30	29 28 27	26 25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	0	R	1	Rs	0	1 0 0	0 0	Rn	1 1 1 1 1
size			A					opc			Rt

No memory ordering variant

Applies when `R == 0`.

```
STSMAXH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release variant

Applies when `R == 1`.

```
STSMAXLH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXLH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

The description of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.285 STSMAX, STSMAXL

Atomic signed maximum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- `STSMAX` has no memory ordering semantics.
- `STSMAXL` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is an alias of the LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL instruction. This means that:

- The encodings in this description are named to match the encodings of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL.
- The description of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL gives the operational pseudocode for this instruction.

ARMv8.1

31 30	29 28	27 26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
1 x	1 1 1	0	0 0	0	R	1	Rs	0	1 0 0	0 0	Rn	1 1 1 1 1
size				A					opc			Rt

32-bit LDSMAX alias variant

Applies when `size == 10 && R == 0`.

```
STSMAX <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAX <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

32-bit LDSMAXL alias variant

Applies when `size == 10 && R == 1`.

```
STSMAXL <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXL <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDSMAX alias variant

Applies when `size == 11 && R == 0`.

```
STSMAX <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDSMAX <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

64-bit LDSMAXL alias variant

Applies when `size == 11 && R == 1`.

```
STSMAXL <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXL <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xs>`	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.286 STSMINB, STSMINLB

Atomic signed minimum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- `STSMINB` has no memory ordering semantics.
- `STSMINLB` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is an alias of the LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB instruction. This means that:

- The encodings in this description are named to match the encodings of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB.
- The description of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB gives the operational pseudocode for this instruction.

ARMv8.1

31 30	29 28 27	26 25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	0	R	1	Rs	0	1 0 1	0 0	Rn	1 1 1 1 1
size			A					opc			Rt

No memory ordering variant

Applies when `R == 0`.

```
STSMINB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMINB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release variant

Applies when `R == 1`.

```
STSMINLB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMINLB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

The description of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.287 STSMINH, STSMINLH

Atomic signed minimum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- `STSMINH` has no memory ordering semantics.
- `STSMINLH` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is an alias of the LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH instruction. This means that:

- The encodings in this description are named to match the encodings of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH.
- The description of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH gives the operational pseudocode for this instruction.

ARMv8.1

31 30	29 28 27	26 25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	0	R	1	Rs	0	1 0 1	0 0	Rn	1 1 1 1 1
size			A					opc			Rt

No memory ordering variant

Applies when `R == 0`.

```
STSMINH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMINH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release variant

Applies when `R == 1`.

```
STSMINLH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMINLH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

The description of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.288 STSMIN, STSMINL

Atomic signed minimum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- `STSMIN` has no memory ordering semantics.
- `STSMINL` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is an alias of the LDSMIN, LDSMINA, LDSMINAL, LDSMINL instruction. This means that:

- The encodings in this description are named to match the encodings of LDSMIN, LDSMINA, LDSMINAL, LDSMINL.
- The description of LDSMIN, LDSMINA, LDSMINAL, LDSMINL gives the operational pseudocode for this instruction.

ARMv8.1

31 30	29 28	27	26	25	24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
1 x	1 1	1	0	0	0	0	R	1	Rs	0	1 0 1	0 0	Rn	1 1 1 1 1
size						A				opc				Rt

32-bit LDSMIN alias variant

Applies when `size == 10 && R == 0`.

```
STSMIN <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMIN <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

32-bit LDSMINL alias variant

Applies when `size == 10 && R == 1`.

```
STSMINL <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMINL <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDSMIN alias variant

Applies when `size == 11 && R == 0`.

```
STSMIN <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDSMIN <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

64-bit LDSMINL alias variant

Applies when `size == 11 && R == 1`.

```
STSMINL <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDSMINL <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xs>`	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSMIN, LDSMINA, LDSMINAL, LDSMINL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.289 STTR

Store Register (unprivileged) stores a word or doubleword from a register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
1 x	1 1 1	0	0 0	0 0	0	imm9	1 0	Rn	Rt
size				opc					

32-bit variant

Applies when `size == 10`.

```
STTR <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit variant

Applies when `size == 11`.

```
STTR <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer datasize = 8 << scale;
boolean tag_checked = n != 31;
```

Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = X[t];
Mem[address, datasize DIV 8, acctype] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.290 STTRB

Store Register Byte (unprivileged) stores a byte from a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 0	1 1 1	0	0 0	0 0	0	imm9	1 0	Rn	Rt
size				opc					

Unscaled offset variant

```
STTRB <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

boolean tag_checked = n != 31;
```

Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = X[t];
Mem[address, 1, acctype] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.291 STTRH

Store Register Halfword (unprivileged) stores a halfword from a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 1	1 1 1	0	0 0	0 0	0	imm9	1 0	Rn	Rt
size				opc					

Unscaled offset variant

```
STTRH <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

boolean tag_checked = n != 31;
```

Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = X[t];
Mem[address, 2, acctype] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.292 STUMAXB, STUMAXLB

Atomic unsigned maximum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- `STUMAXB` has no memory ordering semantics.
- `STUMAXLB` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is an alias of the LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB instruction. This means that:

- The encodings in this description are named to match the encodings of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB.
- The description of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB gives the operational pseudocode for this instruction.

ARMv8.1

31 30	29 28 27	26 25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	0	R	1	Rs	0	1 1 0	0 0	Rn	1 1 1 1 1
size			A					opc			Rt

No memory ordering variant

Applies when `R == 0`.

```
STUMAXB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release variant

Applies when `R == 1`.

```
STUMAXLB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXLB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

The description of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.293 STUMAXH, STUMAXLH

Atomic unsigned maximum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- `STUMAXH` has no memory ordering semantics.
- `STUMAXLH` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is an alias of the LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH instruction. This means that:

- The encodings in this description are named to match the encodings of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH.
- The description of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH gives the operational pseudocode for this instruction.

ARMv8.1

31 30	29 28 27	26 25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	0	R	1	Rs	0	1 1 0	0 0	Rn	1 1 1 1 1
size			A					opc			Rt

No memory ordering variant

Applies when `R == 0`.

```
STUMAXH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release variant

Applies when `R == 1`.

```
STUMAXLH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXLH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

The description of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.294 STUMAX, STUMAXL

Atomic unsigned maximum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- `STUMAX` has no memory ordering semantics.
- `STUMAXL` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is an alias of the LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL instruction. This means that:

- The encodings in this description are named to match the encodings of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL.
- The description of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL gives the operational pseudocode for this instruction.

ARMv8.1

31 30	29 28	27 26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
1 x	1 1	1	0 0	0	R	1	Rs	0	1 1 0	0 0	Rn	1 1 1 1 1
size				A					opc			Rt

32-bit LDUMAX alias variant

Applies when `size == 10 && R == 0`.

```
STUMAX <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAX <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

32-bit LDUMAXL alias variant

Applies when `size == 10 && R == 1`.

```
STUMAXL <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXL <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDUMAX alias variant

Applies when `size == 11 && R == 0`.

```
STUMAX <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDUMAX <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

64-bit LDUMAXL alias variant

Applies when `size == 11 && R == 1`.

```
STUMAXL <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXL <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xs>`	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.295 STUMINB, STUMINLB

Atomic unsigned minimum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- `STUMINB` has no memory ordering semantics.
- `STUMINLB` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is an alias of the LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB instruction. This means that:

- The encodings in this description are named to match the encodings of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB.
- The description of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB gives the operational pseudocode for this instruction.

ARMv8.1

31 30	29 28 27	26 25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	0	R	1	Rs	0	1 1 1	0 0	Rn	1 1 1 1 1
size			A					opc			Rt

No memory ordering variant

Applies when `R == 0`.

```
STUMINB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMINB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release variant

Applies when `R == 1`.

```
STUMINLB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMINLB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

The description of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.296 STUMINH, STUMINLH

Atomic unsigned minimum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- `STUMINH` has no memory ordering semantics.
- `STUMINLH` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is an alias of the LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH instruction. This means that:

- The encodings in this description are named to match the encodings of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH.
- The description of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH gives the operational pseudocode for this instruction.

ARMv8.1

31 30	29 28 27	26 25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	0	R	1	Rs	0	1 1 1	0 0	Rn	1 1 1 1 1
size			A					opc			Rt

No memory ordering variant

Applies when `R == 0`.

```
STUMINH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMINH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release variant

Applies when `R == 1`.

```
STUMINLH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMINLH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

The description of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.297 STUMIN, STUMINL

Atomic unsigned minimum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- `STUMIN` has no memory ordering semantics.
- `STUMINL` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

This instruction is an alias of the LDUMIN, LDUMINA, LDUMINAL, LDUMINL instruction. This means that:

- The encodings in this description are named to match the encodings of LDUMIN, LDUMINA, LDUMINAL, LDUMINL.
- The description of LDUMIN, LDUMINA, LDUMINAL, LDUMINL gives the operational pseudocode for this instruction.

ARMv8.1

31 30	29 28 27	26 25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
1 x	1 1 1	0 0 0	0	R	1	Rs	0	1 1 1	0 0	Rn	1 1 1 1 1
size			A					opc			Rt

32-bit LDUMIN alias variant

Applies when `size == 10 && R == 0`.

```
STUMIN <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMIN <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

32-bit LDUMINL alias variant

Applies when `size == 10 && R == 1`.

```
STUMINL <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMINL <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDUMIN alias variant

Applies when `size == 11 && R == 0`.

```
STUMIN <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDUMIN <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

64-bit LDUMINL alias variant

Applies when `size == 11 && R == 1`.

```
STUMINL <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDUMINL <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xs>`	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDUMIN, LDUMINA, LDUMINAL, LDUMINL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.298 STUR

Store Register (unscaled) calculates an address from a base register value and an immediate offset, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
1	x	1	1	1	0	0	0	0	0	0	imm9	0	0	Rn	Rt
size								opc							

32-bit variant

Applies when `size == 10`.

```
STUR <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit variant

Applies when `size == 11`.

```
STUR <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer datasize = 8 << scale;
boolean tag_checked = n != 31;
```

Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
address = address + offset;

data = X[t];
Mem[address, datasize DIV 8, AccType_NORMAL] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.299 STURB

Store Register Byte (unscaled) calculates an address from a base register value and an immediate offset, and stores a byte to the calculated address, from a 32-bit register. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 0	1 1 1	0	0 0	0 0	0	imm9	0 0	Rn	Rt
size				opc					

Unscaled offset variant

```
STURB <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = X[t];
Mem[address, 1, AccType_NORMAL] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.300 STURH

Store Register Halfword (unscaled) calculates an address from a base register value and an immediate offset, and stores a halfword to the calculated address, from a 32-bit register. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 1	1 1 1	0	0 0	0 0	0	imm9	0 0	Rn	Rt
size				opc					

Unscaled offset variant

```
STURH <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = X[t];
Mem[address, 2, AccType_NORMAL] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.301 STXP

Store Exclusive Pair of registers stores two 32-bit words or two 64-bit doublewords from two registers to a memory location if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores* on page B2-167. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and, if the Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. For information about memory accesses see *Load/Store addressing modes* on page C1-189.

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14 … 10	9 … 5	4 … 0
1	sz	0	0	1	0	0	0	0	0	1	Rs	0	Rt2	Rn	Rt
									L			o0			

32-bit variant

Applies when `sz == 0`.

```
STXP <Ws>, <Wt1>, <Wt2>, [<Xn|SP>{,#0}]
```

64-bit variant

Applies when `sz == 1`.

```
STXP <Ws>, <Xt1>, <Xt2>, [<Xn|SP>{,#0}]
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);    // ignored by load/store single register
integer s = UInt(Rs);    // ignored by all loads and store-release

integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2;
boolean tag_checked = n != 31;
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *STXP* on page K1-7869.

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

- `0` If the operation updates memory.
- `1` If the operation fails to update memory.

`<Xt1>` Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

`<Xt2>` Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

`<Wt1>` Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

`<Wt2>` Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if s == t || (s == t2) then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_NONE    rt_unknown = FALSE;    // store original value
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_NONE    rn_unknown = FALSE;    // address is original base
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    bits(datasize DIV 2) el1 = X[t];
    bits(datasize DIV 2) el2 = X[t2];
    data = if BigEndian() then el1:el2 else el2:el1;
bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, dbytes) then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, AccType_ATOMIC] = data;
    status = ExclusiveMonitorsStatus();
X[s] = ZeroExtend(status, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.302 STXR

Store Exclusive Register stores a 32-bit word or a 64-bit doubleword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores* on page B2-167. For information about memory accesses see *Load/Store addressing modes* on page C1-189.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	10	9	5	4	0
1	x	0	0	1	0	0	0	0	0	0	Rs		0	(1) (1) (1) (1) (1)		Rn		Rt	
size									L				o0	Rt2					

32-bit variant

Applies when `size == 10`.

```
STXR <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

64-bit variant

Applies when `size == 11`.

```
STXR <Ws>, <Xt>, [<Xn|SP>{,#0}]
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);    // ignored by all loads and store-release

integer elsize = 8 << UInt(size);
boolean tag_checked = n != 31;
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *STXR* on page K1-7869.

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

- `0` If the operation updates memory.
- `1` If the operation fails to update memory.

`<Xt>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Ws>` is not updated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch64.ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch64.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if s == t then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_NONE    rt_unknown = FALSE;    // store original value
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_NONE    rn_unknown = FALSE;    // address is original base
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(elsize) UNKNOWN;
else
    data = X[t];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, dbytes) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, AccType_ATOMIC] = data;
    status = ExclusiveMonitorsStatus();
X[s] = ZeroExtend(status, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.303 STXRB

Store Exclusive Register Byte stores a byte from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores* on page B2-167. The memory access is atomic.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

31 30	29 28 27 26 25 24	23	22	21	20 16	15	14 10	9 5	4 0
0 0	0 0 1 0 0 0	0	0	0	Rs	0	(1) (1) (1) (1) (1)	Rn	Rt
size			L			o0	Rt2		

No offset variant

```
STXRB <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);    // ignored by all loads and store-release

boolean tag_checked = n != 31;
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *STXRB* on page K1-7870.

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

- `0` If the operation updates memory.
- `1` If the operation fails to update memory.

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Ws>` is not updated.

If `AArch64.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(8) data;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExt() then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    SetTagCheckedInstruction(tag_checked);

if s == t then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_NONE    rt_unknown = FALSE;    // store original value
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_NONE    rn_unknown = FALSE;    // address is original base
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = X[t];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 1) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 1, AccType_ATOMIC] = data;
    status = ExclusiveMonitorsStatus();
X[s] = ZeroExtend(status, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.304 STXRH

Store Exclusive Register Halfword stores a halfword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores* on page B2-167. The memory access is atomic.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

31 30	29 28 27 26 25 24	23 22	21	20 16	15	14 10	9 5	4 0
0 1	0 0 1 0 0 0	0 0	0	Rs	0	(1) (1) (1) (1) (1)	Rn	Rt
size		L			o0	Rt2		

No offset variant

```
STXRH <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);    // ignored by all loads and store-release

boolean tag_checked = n != 31;
```

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

- `0` If the operation updates memory.
- `1` If the operation fails to update memory.

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Ws>` is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch64.ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch64.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(16) data;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if s == t then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_NONE    rt_unknown = FALSE;    // store original value
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_NONE    rn_unknown = FALSE;    // address is original base
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 2) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 2, AccType_ATOMIC] = data;
    status = ExclusiveMonitorsStatus();
X[s] = ZeroExtend(status, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.305 STZ2G

Store Allocation Tags, Zeroing stores an Allocation Tag to two Tag granules of memory, zeroing the associated data locations. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

Post-index

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21	20	12	11	10	9	5	4	0
1	1	0	1	1	0	0	1	1	1	1	imm9		0	1	Xn		Xt	

Post-index variant

```
STZ2G <Xt|SP>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;
```

Pre-index

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21	20	12	11	10	9	5	4	0
1	1	0	1	1	0	0	1	1	1	1	imm9		1	1	Xn		Xt	

Pre-index variant

```
STZ2G <Xt|SP>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;
```

Signed offset

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21	20	12	11	10	9	5	4	0
1	1	0	1	1	0	0	1	1	1	1	imm9		1	0	Xn		Xt	

Signed offset variant

```
STZ2G <Xt|SP>, [<Xn|SP>{, #<simm>}]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;
```

Assembler symbols

`<Xt|SP>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
`<simm>`	Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

Operation for all encodings

```
bits(64) address;
bits(64) data = if t == 31 then SP[] else X[t];
bits(4) tag = AArch64.AllocationTagFromAddress(data);

SetTagCheckedInstruction(FALSE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

Mem[address, TAG_GRANULE, AccType_NORMAL] = Zeros(8 * TAG_GRANULE);
Mem[address+TAG_GRANULE, TAG_GRANULE, AccType_NORMAL] = Zeros(8 * TAG_GRANULE);

AArch64.MemTag[address, AccType_NORMAL] = tag;
AArch64.MemTag[address+TAG_GRANULE, AccType_NORMAL] = tag;

if writeback then
    if postindex then
        address = address + offset;

    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.306 STZG

Store Allocation Tag, Zeroing stores an Allocation Tag to memory, zeroing the associated data location. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

Post-index

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21	20	12	11	10	9	5	4	0
1	1	0	1	1	0	0	1	0	1	1	imm9		0	1	Xn		Xt	

Post-index variant

```
STZG <Xt|SP>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;
```

Pre-index

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21	20	12	11	10	9	5	4	0
1	1	0	1	1	0	0	1	0	1	1	imm9		1	1	Xn		Xt	

Pre-index variant

```
STZG <Xt|SP>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;
```

Signed offset

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21	20	12	11	10	9	5	4	0
1	1	0	1	1	0	0	1	0	1	1	imm9		1	0	Xn		Xt	

Signed offset variant

```
STZG <Xt|SP>, [<Xn|SP>{, #<simm>}]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;
```

Assembler symbols

`<Xt|SP>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
`<simm>`	Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

Operation for all encodings

```
bits(64) address;

SetTagCheckedInstruction(FALSE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

Mem[address, TAG_GRANULE, AccType_NORMAL] = Zeros(TAG_GRANULE * 8);

bits(64) data = if t == 31 then SP[] else X[t];
bits(4) tag = AArch64.AllocationTagFromAddress(data);
AArch64.MemTag[address, AccType_NORMAL] = tag;

if writeback then
    if postindex then
        address = address + offset;

    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.307 STZGM

Store Tag and Zero Multiple writes a naturally aligned block of N Allocation Tags and stores zero to the associated data locations, where the size of N is identified in DCZID_EL0.BS, and the Allocation Tag written to address A is taken from the source register bits<3:0>.

This instruction is UNDEFINED at EL0.

This instruction generates an Unchecked access.

If ID_AA64PFR1_EL1.MTE != 0b0010, this instruction is UNDEFINED.

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
1	1	0	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	Xn		Xt	

Integer variant

```
STZGM <Xt>, [<Xn|SP>]
```

Decode for this encoding

```
integer t = UInt(Xt);
integer n = UInt(Xn);
```

Assembler symbols

<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

Operation

```
if PSTATE.EL == EL0 then
    UNDEFINED;

bits(64) data = X[t];
bits(4) tag = data<3:0>;
bits(64) address;
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

integer size = 4 * (2 ^ (UInt(DCZID_EL0.BS)));
address = Align(address, size);
integer count = size >> LOG2_TAG_GRANULE;

for i = 0 to count-1
    AArch64.MemTag[address, AccType_NORMAL] = tag;
    Mem[address, TAG_GRANULE, AccType_NORMAL] = Zeros(8 * TAG_GRANULE);
    address = address + TAG_GRANULE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.308 SUB (extended register)

Subtract (extended register) subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value, and writes the result to the destination register. The argument that is extended from the `<Rm>` register can be a byte, halfword, word, or doubleword.

31	30	29	28 27 26 25 24	23 22	21	20 16	15 13	12 10	9 5	4 0
sf	1	0	0 1 0 1 1	0 0	1	Rm	option	imm3	Rn	Rd
	op	S								

32-bit variant

Applies when `sf == 0`.

```
SUB <Wd|WSP>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

64-bit variant

Applies when `sf == 1`.

```
SUB <Xd|SP>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then UNDEFINED;
```

Assembler symbols

`<Wd|WSP>` Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

`<Wn|WSP>` Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<Xd|SP>` Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

`<Xn|SP>` Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

`<R>` Is a width specifier, encoded in the "option" field. It can have the following values:

`W`	when `option = 00x`
`W`	when `option = 010`
`X`	when `option = x11`
`W`	when `option = 10x`
`W`	when `option = 110`

`<m>` Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

UXTB when option = 000

UXTH when option = 001

LSL|UXTW when option = 010

UXTX when option = 011

SXTB when option = 100

SXTH when option = 101

SXTW when option = 110

SXTX when option = 111

If "Rd" or "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

UXTB when option = 000

UXTH when option = 001

UXTW when option = 010

LSL|UXTX when option = 011

SXTB when option = 100

SXTH when option = 101

SXTW when option = 110

SXTX when option = 111

If "Rd" or "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);

operand2 = NOT(operand2);
(result, -) = AddWithCarry(operand1, operand2, '1');

if d == 31 then
    SP[] = result;
else
    X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

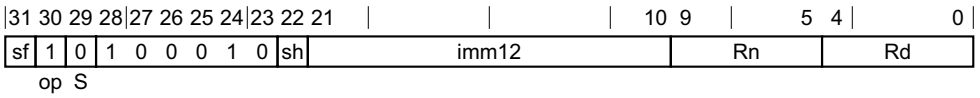
Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.309 SUB (immediate)

Subtract (immediate) subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register.



32-bit variant

Applies when `sf == 0`.

```
SUB <Wd|WSP>, <Wn|WSP>, #<imm>{, <shift>}
```

64-bit variant

Applies when `sf == 1`.

```
SUB <Xd|SP>, <Xn|SP>, #<imm>{, <shift>}
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;

case sh of
    when '0' imm = ZeroExtend(imm12, datasize);
    when '1' imm = ZeroExtend(imm12:Zeros(12), datasize);
```

Assembler symbols

`<Wd|WSP>` Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

`<Wn|WSP>` Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

`<Xd|SP>` Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

`<Xn|SP>` Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

`<imm>` Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.

`<shift>` Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "sh" field. It can have the following values:

`LSL #0`	when `sh = 0`
`LSL #12`	when `sh = 1`

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2;

operand2 = NOT(imm);
(result, -) = AddWithCarry(operand1, operand2, '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if d == 31 then
    SP[] = result;
else
    X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.310 SUB (shifted register)

Subtract (shifted register) subtracts an optionally-shifted register value from a register value, and writes the result to the destination register.

This instruction is used by the alias NEG (shifted register). See *Alias conditions* for details of when each alias is preferred.

31	30	29	28	27	26	25	24	23 22	21	20 16	15 10	9 5	4 0
sf	1	0	0	1	0	1	1	shift	0	Rm	imm6	Rn	Rd
	op	S											

32-bit variant

Applies when `sf == 0`.

```
SUB <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit variant

Applies when `sf == 1`.

```
SUB <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;

if shift == '11' then UNDEFINED;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

Alias conditions

Alias	is preferred when
NEG (shifted register)	`Rn == '11111'`

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Wm>`	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<shift>` Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in the "shift" field. It can have the following values:

`LSL` when `shift = 00`

`LSR` when `shift = 01`

`ASR` when `shift = 10`

The encoding `shift = 11` is reserved.

`<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

operand2 = NOT(operand2);
(result, -) = AddWithCarry(operand1, operand2, '1');

X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.311 SUBG

Subtract with Tag subtracts an immediate value scaled by the Tag granule from the address in the source register, modifies the Logical Address Tag of the address using an immediate value, and writes the result to the destination register. Tags specified in GCR_EL1.Exclude are excluded from the possible outputs when modifying the Logical Address Tag.

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21 … 16	15	14	13 … 10	9 … 5	4 … 0
1	1	0	1	0	0	0	1	1	0	uimm6	(0)	(0)	uimm4	Xn	Xd
											op3				

Integer variant

```
SUBG <Xd|SP>, <Xn|SP>, #<uimm6>, #<uimm4>
```

Decode for this encoding

```
integer d = UInt(Xd);
integer n = UInt(Xn);
bits(64) offset = LSL(ZeroExtend(uimm6, 64), LOG2_TAG_GRANULE);
```

Assembler symbols

<Xd|SP>	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Xd" field.
<Xn|SP>	Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Xn" field.
<uimm6>	Is an unsigned immediate, a multiple of 16 in the range 0 to 1008, encoded in the "uimm6" field.
<uimm4>	Is an unsigned immediate, in the range 0 to 15, encoded in the "uimm4" field.

Operation

```
bits(64) operand1 = if n == 31 then SP[] else X[n];
bits(4) start_tag = AArch64.AllocationTagFromAddress(operand1);
bits(16) exclude = GCR_EL1.Exclude;
bits(64) result;
bits(4) rtag;

if AArch64.AllocationTagAccessIsEnabled() then
    rtag = AArch64.ChooseNonExcludedTag(start_tag, uimm4, exclude);
else
    rtag = '0000';

(result, -) = AddWithCarry(operand1, NOT(offset), '1');

result = AArch64.AddressWithAllocationTag(result, rtag);

if d == 31 then
    SP[] = result;
else
    X[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.312 SUBP

Subtract Pointer subtracts the 56-bit address held in the second source register from the 56-bit address held in the first source register, sign-extends the result to 64-bits, and writes the result to the destination register.

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
1	0	0	1	1	0	1	0	1	1	0	Xm		0	0	0	0	0	0	Xn		Xd	

Integer variant

```
SUBP <Xd>, <Xn|SP>, <Xm|SP>
```

Decode for this encoding

```
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);
```

Assembler symbols

`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Xd" field.
`<Xn|SP>`	Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.
`<Xm|SP>`	Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the "Xm" field.

Operation

```
bits(64) operand1 = if n == 31 then SP[] else X[n];
bits(64) operand2 = if m == 31 then SP[] else X[m];
operand1 = SignExtend(operand1<55:0>, 64);
operand2 = SignExtend(operand2<55:0>, 64);

bits(64) result;

operand2 = NOT(operand2);
(result, -) = AddWithCarry(operand1, operand2, '1');

X[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.313 SUBPS

Subtract Pointer, setting Flags subtracts the 56-bit address held in the second source register from the 56-bit address held in the first source register, sign-extends the result to 64-bits, and writes the result to the destination register. It updates the condition flags based on the result of the subtraction.

This instruction is used by the alias CMPP. See *Alias conditions* for details of when each alias is preferred.

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
1	0	1	1	1	0	1	0	1	1	0	Xm		0	0	0	0	0	0	Xn		Xd	

Integer variant

```
SUBPS <Xd>, <Xn|SP>, <Xm|SP>
```

Decode for this encoding

```
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);
```

Alias conditions

Alias	is preferred when
CMPP	`S == '1' && Xd == '11111'`

Assembler symbols

`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Xd" field.
`<Xn|SP>`	Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.
`<Xm|SP>`	Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the "Xm" field.

Operation

```
bits(64) operand1 = if n == 31 then SP[] else X[n];
bits(64) operand2 = if m == 31 then SP[] else X[m];
operand1 = SignExtend(operand1<55:0>, 64);
operand2 = SignExtend(operand2<55:0>, 64);

bits(64) result;
bits(4) nzcv;

operand2 = NOT(operand2);
(result, nzcv) = AddWithCarry(operand1, operand2, '1');

PSTATE.<N,Z,C,V> = nzcv;
X[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.314 SUBS (extended register)

Subtract (extended register), setting flags, subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value, and writes the result to the destination register. The argument that is extended from the `<Rm>` register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result.

This instruction is used by the alias CMP (extended register). See Alias conditions for details of when each alias is preferred.

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15 … 13	12 … 10	9 … 5	4 … 0
sf	1 (op)	1 (S)	0	1	0	1	1	0	0	1	Rm	option	imm3	Rn	Rd

32-bit variant

Applies when `sf == 0`.

```
SUBS <Wd>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

64-bit variant

Applies when `sf == 1`.

```
SUBS <Xd>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then UNDEFINED;
```

Alias conditions

Alias	is preferred when
CMP (extended register)	`Rd == '11111'`

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn|WSP>` Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn|SP>` Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

`<R>` Is a width specifier, encoded in the "option" field. It can have the following values:

`W` when `option = 00x`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

W	when option = 010
X	when option = x11
W	when option = 10x
W	when option = 110

<m> Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

UXTB	when option = 000
UXTH	when option = 001
LSL|UXTW	when option = 010
UXTX	when option = 011
SXTB	when option = 100
SXTH	when option = 101
SXTW	when option = 110
SXTX	when option = 111

If "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

UXTB	when option = 000
UXTH	when option = 001
UXTW	when option = 010
LSL|UXTX	when option = 011
SXTB	when option = 100
SXTH	when option = 101
SXTW	when option = 110
SXTX	when option = 111

If "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);
bits(4) nzcv;

operand2 = NOT(operand2);
(result, nzcv) = AddWithCarry(operand1, operand2, '1');

PSTATE.<N,Z,C,V> = nzcv;

X[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.315 SUBS (immediate)

Subtract (immediate), setting flags, subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias CMP (immediate). See *Alias conditions* for details of when each alias is preferred.

31	30	29	28	27	26	25	24	23	22	21	10	9	5	4	0
sf	1	1	1	0	0	0	1	0	sh	imm12			Rn		Rd
	op	S													

32-bit variant

Applies when `sf == 0`.

```
SUBS <Wd>, <Wn|WSP>, #<imm>{, <shift>}
```

64-bit variant

Applies when `sf == 1`.

```
SUBS <Xd>, <Xn|SP>, #<imm>{, <shift>}
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;

case sh of
    when '0' imm = ZeroExtend(imm12, datasize);
    when '1' imm = ZeroExtend(imm12:Zeros(12), datasize);
```

Alias conditions

Alias	is preferred when
CMP (immediate)	`Rd == '11111'`

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn|WSP>`	Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn|SP>`	Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
`<imm>`	Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.

`<shift>`	Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "sh" field. It can have the following values:
	`LSL #0` when `sh = 0`
	`LSL #12` when `sh = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2;
bits(4) nzcv;

operand2 = NOT(imm);
(result, nzcv) = AddWithCarry(operand1, operand2, '1');

PSTATE.<N,Z,C,V> = nzcv;

X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.316 SUBS (shifted register)

Subtract (shifted register), setting flags, subtracts an optionally-shifted register value from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the aliases CMP (shifted register) and NEGS. See *Alias conditions* for details of when each alias is preferred.

31	30	29	28	27	26	25	24	23 22	21	20 16	15 10	9 5	4 0
sf	1	1	0	1	0	1	1	shift	0	Rm	imm6	Rn	Rd
	op	S											

32-bit variant

Applies when `sf == 0`.

```
SUBS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit variant

Applies when `sf == 1`.

```
SUBS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;

if shift == '11' then UNDEFINED;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

Alias conditions

Alias	is preferred when
CMP (shifted register)	`Rd == '11111'`
NEGS	`Rn == '11111'`

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Wm>`	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<shift>` Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in the "shift" field. It can have the following values:

`LSL` when `shift = 00`

`LSR` when `shift = 01`

`ASR` when `shift = 10`

The encoding `shift = 11` is reserved.

`<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
bits(4) nzcv;

operand2 = NOT(operand2);
(result, nzcv) = AddWithCarry(operand1, operand2, '1');

PSTATE.<N,Z,C,V> = nzcv;

X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.317 SVC

Supervisor Call causes an exception to be taken to EL1.

On executing an `SVC` instruction, the PE records the exception as a Supervisor Call exception in ESR_ELx, using the EC value `0x15`, and the value of the immediate argument.

31 30 29 28	27 26 25 24	23 22 21 20	5 4	3 2 1 0
1 1 0 1	0 1 0 0	0 0 0	imm16	0 0 0 0 1

System variant

```
SVC #<imm>
```

Decode for this encoding

```
// Empty.
```

Assembler symbols

`<imm>` Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

Operation

```
AArch64.CheckForSVCTrap(imm16);
AArch64.CallSupervisor(imm16);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.318 SWPB, SWPAB, SWPALB, SWPLB

Swap byte in memory atomically loads an 8-bit byte from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `SWPAB` and `SWPALB` load from memory with acquire semantics.
- `SWPLB` and `SWPALB` store to memory with release semantics.
- `SWPB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 13 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	A	R	1	Rs	1	0 0 0	0 0	Rn	Rt
size												

SWPAB variant

Applies when `A == 1 && R == 0`.

```
SWPAB <Ws>, <Wt>, [<Xn|SP>]
```

SWPALB variant

Applies when `A == 1 && R == 1`.

```
SWPALB <Ws>, <Wt>, [<Xn|SP>]
```

SWPB variant

Applies when `A == 0 && R == 0`.

```
SWPB <Ws>, <Wt>, [<Xn|SP>]
```

SWPLB variant

Applies when `A == 0 && R == 1`.

```
SWPLB <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;
bits(8) store_value;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

store_value = X[s];
data = MemAtomic(address, MemAtomicOp_SWP, store_value, ldacctype, stacctype);
X[t] = ZeroExtend(data, 32);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.319 SWPH, SWPAH, SWPALH, SWPLH

Swap halfword in memory atomically loads a 16-bit halfword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `SWPAH` and `SWPALH` load from memory with acquire semantics.
- `SWPLH` and `SWPALH` store to memory with release semantics.
- `SWPH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

ARMv8.1

31 30	29 28 27	26 25 24	23	22	21	20 16	15	14 13 12	11 10	9 5	4 0
0 1	1 1 1	0 0 0	A	R	1	Rs	1	0 0 0	0 0	Rn	Rt
size											

SWPAH variant

Applies when `A == 1 && R == 0`.

```
SWPAH <Ws>, <Wt>, [<Xn|SP>]
```

SWPALH variant

Applies when `A == 1 && R == 1`.

```
SWPALH <Ws>, <Wt>, [<Xn|SP>]
```

SWPH variant

Applies when `A == 0 && R == 0`.

```
SWPH <Ws>, <Wt>, [<Xn|SP>]
```

SWPLH variant

Applies when `A == 0 && R == 1`.

```
SWPLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;
bits(16) store_value;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

store_value = X[s];
data = MemAtomic(address, MemAtomicOp_SWP, store_value, ldacctype, stacctype);
X[t] = ZeroExtend(data, 32);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.320 SWP, SWPA, SWPAL, SWPL

Swap word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `SWPA` and `SWPAL` load from memory with acquire semantics.
- `SWPL` and `SWPAL` store to memory with release semantics.
- `SWP` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

For information about memory accesses see *Load/Store addressing modes* on page C1-189.

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
1	x	1	1	1	0	0	0	A	R	1	Rs	1	0	0	0	0	0	Rn	Rt
size																			

32-bit SWP variant

Applies when `size == 10 && A == 0 && R == 0`.

```
SWP <Ws>, <Wt>, [<Xn|SP>]
```

32-bit SWPA variant

Applies when `size == 10 && A == 1 && R == 0`.

```
SWPA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit SWPAL variant

Applies when `size == 10 && A == 1 && R == 1`.

```
SWPAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit SWPL variant

Applies when `size == 10 && A == 0 && R == 1`.

```
SWPL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit SWP variant

Applies when `size == 11 && A == 0 && R == 0`.

```
SWP <Xs>, <Xt>, [<Xn|SP>]
```

64-bit SWPA variant

Applies when `size == 11 && A == 1 && R == 0`.

```
SWPA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit SWPAL variant

Applies when `size == 11 && A == 1 && R == 1`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
SWPAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit SWPL variant

Applies when `size == 11 && A == 0 && R == 1`.

```
SWPL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
`<Wt>`	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
`<Xs>`	Is the 64-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) data;
bits(datasize) store_value;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

store_value = X[s];
data = MemAtomic(address, MemAtomicOp_SWP, store_value, ldacctype, stacctype);
X[t] = ZeroExtend(data, regsize);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.321 SXTB

Signed Extend Byte extracts an 8-bit value from a register, sign-extends it to the size of the register, and writes the result to the destination register.

This instruction is an alias of the SBFM instruction. This means that:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

31	30 29	28 27 26 25 24	23	22 21 16	15 10	9 5	4 0
sf	0 0	1 0 0 1 1 0	N	0 0 0 0 0 0	0 0 0 1 1 1	Rn	Rd
	opc			immr	imms		

32-bit variant

Applies when `sf == 0 && N == 0`.

```
SXTB <Wd>, <Wn>
```

is equivalent to

```
SBFM <Wd>, <Wn>, #0, #7
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1 && N == 1`.

```
SXTB <Xd>, <Wn>
```

is equivalent to

```
SBFM <Xd>, <Xn>, #0, #7
```

and is always the preferred disassembly.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
`<Wn>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.322 SXTH

Sign Extend Halfword extracts a 16-bit value, sign-extends it to the size of the register, and writes the result to the destination register.

This instruction is an alias of the SBFM instruction. This means that:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

31	30 29	28 27 26 25 24	23	22 21 16	15 10	9 5	4 0
sf	0 0	1 0 0 1 1 0	N	0 0 0 0 0 0	0 0 1 1 1 1	Rn	Rd
	opc			immr	imms		

32-bit variant

Applies when `sf == 0 && N == 0`.

```
SXTH <Wd>, <Wn>
```

is equivalent to

```
SBFM <Wd>, <Wn>, #0, #15
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1 && N == 1`.

```
SXTH <Xd>, <Wn>
```

is equivalent to

```
SBFM <Xd>, <Xn>, #0, #15
```

and is always the preferred disassembly.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
`<Wn>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.323 SXTW

Sign Extend Word sign-extends a word to the size of the register, and writes the result to the destination register.

This instruction is an alias of the SBFM instruction. This means that:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

31	30 29	28 27 26 25 24	23	22 21 16	15 10	9 5	4 0
1	0 0	1 0 0 1 1 0	1	0 0 0 0 0 0	0 1 1 1 1 1	Rn	Rd
sf	opc		N	immr	imms		

64-bit variant

```
SXTW <Xd>, <Wn>
```

is equivalent to

```
SBFM <Xd>, <Xn>, #0, #31
```

and is always the preferred disassembly.

Assembler symbols

`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
`<Wn>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.324 SYS

System instruction. For more information, see *op0==0b01, cache maintenance, TLB maintenance, and address translation instructions* on page C5-380 for the encodings of System instructions.

This instruction is used by the aliases AT, CFP, CPP, DC, DVP, IC, and TLBI. See *Alias conditions* for details of when each alias is preferred.

31 30 29 28	27 26 25 24	23 22 21 20	19 18 16	15 12	11 8	7 5	4 0
1 1 0 1	0 1 0 1	0 0 0 1	op1	CRn	CRm	op2	Rt
		L					

System variant

```
SYS #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>}
```

Decode for this encoding

```
AArch64.CheckSystemAccess('01', op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);

integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
```

Alias conditions

Alias	is preferred when
AT	`CRn == '0111' && CRm == '100x' && SysOp(op1,'0111',CRm,op2) == Sys_AT`
CFP	`op1 == '011' && CRn == '0111' && CRm == '0011' && op2 == '100'`
CPP	`op1 == '011' && CRn == '0111' && CRm == '0011' && op2 == '111'`
DC	`CRn == '0111' && SysOp(op1,'0111',CRm,op2) == Sys_DC`
DVP	`op1 == '011' && CRn == '0111' && CRm == '0011' && op2 == '101'`
IC	`CRn == '0111' && SysOp(op1,'0111',CRm,op2) == Sys_IC`
TLBI	`CRn == '1000' && SysOp(op1,'1000',CRm,op2) == Sys_TLBI`

Assembler symbols

`<op1>`	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
`<Cn>`	Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
`<Cm>`	Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
`<op2>`	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.
`<Xt>`	Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
AArch64.SysInstr(1, sys_op1, sys_crn, sys_crm, sys_op2, X[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.325 SYSL

System instruction with result. For more information, see *op0==0b01, cache maintenance, TLB maintenance, and address translation instructions* on page C5-380 for the encodings of System instructions.

31 30 29 28	27 26 25 24	23 22 21	20	19 18 16	15 12	11 8	7 5	4 0
1 1 0 1	0 1 0 1	0 0 1	0 1	op1	CRn	CRm	op2	Rt
			L					

System variant

```
SYSL <Xt>, #<op1>, <Cn>, <Cm>, #<op2>
```

Decode for this encoding

```
AArch64.CheckSystemAccess('01', op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);

integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
```

Assembler symbols

<Xt>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.
<op1>	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
<Cn>	Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
<Cm>	Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
<op2>	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

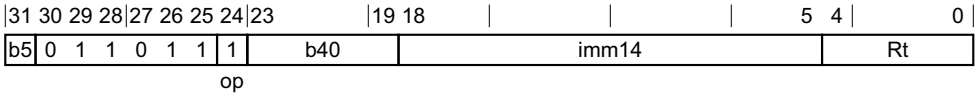
Operation

```
// No architecturally defined instructions here.
X[t] = AArch64.SysInstrWithResult(1, sys_op1, sys_crn, sys_crm, sys_op2);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.326 TBNZ

Test bit and Branch if Nonzero compares the value of a bit in a general-purpose register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is not equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.



14-bit signed PC-relative branch offset variant

```
TBNZ <R><t>, #<imm>, <label>
```

Decode for this encoding

```
integer t = UInt(Rt);

integer datasize = if b5 == '1' then 64 else 32;
integer bit_pos = UInt(b5:b40);
bits(64) offset = SignExtend(imm14:'00', 64);
```

Assembler symbols

`<R>` Is a width specifier, encoded in the "b5" field. It can have the following values:

`W` when `b5 = 0`

`X` when `b5 = 1`

In assembler source code an 'X' specifier is always permitted, but a 'W' specifier is only permitted when the bit number is less than 32.

`<t>` Is the number [0-30] of the general-purpose register to be tested or the name ZR (31), encoded in the "Rt" field.

`<imm>` Is the bit number to be tested, in the range 0 to 63, encoded in "b5:b40".

`<label>` Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-32KB, is encoded as "imm14" times 4.

Operation

```
bits(datasize) operand = X[t];

if operand<bit_pos> == op then
    BranchTo(PC[] + offset, BranchType_DIR);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.327 TBZ

Test bit and Branch if Zero compares the value of a test bit with zero, and conditionally branches to a label at a PC-relative offset if the comparison is equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.

31	30 29 28	27 26 25	24	23 19	18 5	4 0
b5	0 1 1	0 1 1	0	b40	imm14	Rt
			op			

14-bit signed PC-relative branch offset variant

```
TBZ <R><t>, #<imm>, <label>
```

Decode for this encoding

```
integer t = UInt(Rt);

integer datasize = if b5 == '1' then 64 else 32;
integer bit_pos = UInt(b5:b40);
bits(64) offset = SignExtend(imm14:'00', 64);
```

Assembler symbols

`<R>` Is a width specifier, encoded in the "b5" field. It can have the following values:

`W` when `b5 = 0`

`X` when `b5 = 1`

In assembler source code an 'X' specifier is always permitted, but a 'W' specifier is only permitted when the bit number is less than 32.

`<t>` Is the number [0-30] of the general-purpose register to be tested or the name ZR (31), encoded in the "Rt" field.

`<imm>` Is the bit number to be tested, in the range 0 to 63, encoded in "b5:b40".

`<label>` Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-32KB, is encoded as "imm14" times 4.

Operation

```
bits(datasize) operand = X[t];

if operand<bit_pos> == op then
    BranchTo(PC[] + offset, BranchType_DIR);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.328 TLBI

TLB Invalidate operation. For more information, see *op0==0b01, cache maintenance, TLB maintenance, and address translation instructions* on page C5-380.

This instruction is an alias of the SYS instruction. This means that:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode for this instruction.

31 30 29 28	27 26 25 24	23 22	21	20	19 18 16	15 12	11 8	7 5	4 0
1 1 0 1	0 1 0 1	0 0	0	0 1	op1	1 0 0 0	CRm	op2	Rt
			L			CRn			

System variant

```
TLBI <tlbi_op>{, <Xt>}
```

is equivalent to

```
SYS #<op1>, C8, <Cm>, #<op2>{, <Xt>}
```

and is the preferred disassembly when `SysOp(op1,'1000',CRm,op2) == Sys_TLBI`.

Assembler symbols

`<op1>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

`<Cm>` Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

`<op2>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

`<tlbi_op>` Is a TLBI instruction name, as listed for the TLBI system instruction group, encoded in the "op1:CRm:op2" field. It can have the following values:

`VMALLE1IS`	when `op1 = 000, CRm = 0011, op2 = 000`
`VAE1IS`	when `op1 = 000, CRm = 0011, op2 = 001`
`ASIDE1IS`	when `op1 = 000, CRm = 0011, op2 = 010`
`VAAE1IS`	when `op1 = 000, CRm = 0011, op2 = 011`
`VALE1IS`	when `op1 = 000, CRm = 0011, op2 = 101`
`VAALE1IS`	when `op1 = 000, CRm = 0011, op2 = 111`
`VMALLE1`	when `op1 = 000, CRm = 0111, op2 = 000`
`VAE1`	when `op1 = 000, CRm = 0111, op2 = 001`
`ASIDE1`	when `op1 = 000, CRm = 0111, op2 = 010`
`VAAE1`	when `op1 = 000, CRm = 0111, op2 = 011`
`VALE1`	when `op1 = 000, CRm = 0111, op2 = 101`
`VAALE1`	when `op1 = 000, CRm = 0111, op2 = 111`
`IPAS2E1IS`	when `op1 = 100, CRm = 0000, op2 = 001`
`IPAS2LE1IS`	when `op1 = 100, CRm = 0000, op2 = 101`
`ALLE2IS`	when `op1 = 100, CRm = 0011, op2 = 000`
`VAE2IS`	when `op1 = 100, CRm = 0011, op2 = 001`
`ALLE1IS`	when `op1 = 100, CRm = 0011, op2 = 100`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

VALE2IS when op1 = 100, CRm = 0011, op2 = 101

VMALLS12E1IS when op1 = 100, CRm = 0011, op2 = 110

IPAS2E1 when op1 = 100, CRm = 0100, op2 = 001

IPAS2LE1 when op1 = 100, CRm = 0100, op2 = 101

ALLE2 when op1 = 100, CRm = 0111, op2 = 000

VAE2 when op1 = 100, CRm = 0111, op2 = 001

ALLE1 when op1 = 100, CRm = 0111, op2 = 100

VALE2 when op1 = 100, CRm = 0111, op2 = 101

VMALLS12E1 when op1 = 100, CRm = 0111, op2 = 110

ALLE3IS when op1 = 110, CRm = 0011, op2 = 000

VAE3IS when op1 = 110, CRm = 0011, op2 = 001

VALE3IS when op1 = 110, CRm = 0011, op2 = 101

ALLE3 when op1 = 110, CRm = 0111, op2 = 000

VAE3 when op1 = 110, CRm = 0111, op2 = 001

VALE3 when op1 = 110, CRm = 0111, op2 = 101

When ARMv8.4-TLBI is implemented, the following values are also valid:

VMALLE1OS when op1 = 000, CRm = 0001, op2 = 000

VAE1OS when op1 = 000, CRm = 0001, op2 = 001

ASIDE1OS when op1 = 000, CRm = 0001, op2 = 010

VAAE1OS when op1 = 000, CRm = 0001, op2 = 011

VALE1OS when op1 = 000, CRm = 0001, op2 = 101

VAALE1OS when op1 = 000, CRm = 0001, op2 = 111

RVAE1IS when op1 = 000, CRm = 0010, op2 = 001

RVAAE1IS when op1 = 000, CRm = 0010, op2 = 011

RVALE1IS when op1 = 000, CRm = 0010, op2 = 101

RVAALE1IS when op1 = 000, CRm = 0010, op2 = 111

RVAE1OS when op1 = 000, CRm = 0101, op2 = 001

RVAAE1OS when op1 = 000, CRm = 0101, op2 = 011

RVALE1OS when op1 = 000, CRm = 0101, op2 = 101

RVAALE1OS when op1 = 000, CRm = 0101, op2 = 111

RVAE1 when op1 = 000, CRm = 0110, op2 = 001

RVAAE1 when op1 = 000, CRm = 0110, op2 = 011

RVALE1 when op1 = 000, CRm = 0110, op2 = 101

RVAALE1 when op1 = 000, CRm = 0110, op2 = 111

RIPAS2E1IS when op1 = 100, CRm = 0000, op2 = 010

RIPAS2LE1IS when op1 = 100, CRm = 0000, op2 = 110

ALLE2OS when op1 = 100, CRm = 0001, op2 = 000

VAE2OS when op1 = 100, CRm = 0001, op2 = 001

ALLE1OS when op1 = 100, CRm = 0001, op2 = 100

VALE2OS when op1 = 100, CRm = 0001, op2 = 101

VMALLS12E1OS when op1 = 100, CRm = 0001, op2 = 110

RVAE2IS when op1 = 100, CRm = 0010, op2 = 001

RVALE2IS when op1 = 100, CRm = 0010, op2 = 101

IPAS2E1OS when op1 = 100, CRm = 0100, op2 = 000

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

RIPAS2E1	when op1 = 100, CRm = 0100, op2 = 010
RIPAS2E1OS	when op1 = 100, CRm = 0100, op2 = 011
IPAS2LE1OS	when op1 = 100, CRm = 0100, op2 = 100
RIPAS2LE1	when op1 = 100, CRm = 0100, op2 = 110
RIPAS2LE1OS	when op1 = 100, CRm = 0100, op2 = 111
RVAE2OS	when op1 = 100, CRm = 0101, op2 = 001
RVALE2OS	when op1 = 100, CRm = 0101, op2 = 101
RVAE2	when op1 = 100, CRm = 0110, op2 = 001
RVALE2	when op1 = 100, CRm = 0110, op2 = 101
ALLE3OS	when op1 = 110, CRm = 0001, op2 = 000
VAE3OS	when op1 = 110, CRm = 0001, op2 = 001
VALE3OS	when op1 = 110, CRm = 0001, op2 = 101
RVAE3IS	when op1 = 110, CRm = 0010, op2 = 001
RVALE3IS	when op1 = 110, CRm = 0010, op2 = 101
RVAE3OS	when op1 = 110, CRm = 0101, op2 = 001
RVALE3OS	when op1 = 110, CRm = 0101, op2 = 101
RVAE3	when op1 = 110, CRm = 0110, op2 = 001
RVALE3	when op1 = 110, CRm = 0110, op2 = 101

<Xt> Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.329 TSB CSYNC

Trace Synchronization Barrier. This instruction is a barrier that synchronizes the trace operations of instructions.

If ARMv8.4-Trace is not implemented, this instruction executes as a `NOP`.

ARMv8.4

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11			8	7			5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	1	0	0	1	0		1	1	1	1	1
																				CRm				op2								

System variant

```
TSB CSYNC
```

Decode for this encoding

```
if !HaveSelfHostedTrace() then EndOfInstruction();
```

Operation

```
TraceSynchronizationBarrier();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.330 TST (immediate)

Test bits (immediate) , setting the condition flags and discarding the result : `Rn AND imm`

This instruction is an alias of the ANDS (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of ANDS (immediate).
- The description of ANDS (immediate) gives the operational pseudocode for this instruction.

31	30 29	28	27 26 25 24	23	22 21	16	15 10	9 5	4 0
sf	1 1	1	0 0 1 0	0	N	immr	imms	Rn	1 1 1 1 1
	opc								Rd

32-bit variant

Applies when `sf == 0 && N == 0`.

```
TST <Wn>, #<imm>
```

is equivalent to

```
ANDS WZR, <Wn>, #<imm>
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

```
TST <Xn>, #<imm>
```

is equivalent to

```
ANDS XZR, <Xn>, #<imm>
```

and is always the preferred disassembly.

Assembler symbols

`<Wn>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
`<Xn>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
`<imm>`	For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr". For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

Operation

The description of ANDS (immediate) gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.331 TST (shifted register)

Test (shifted register) performs a bitwise AND operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

This instruction is an alias of the ANDS (shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of ANDS (shifted register).
- The description of ANDS (shifted register) gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23 22	21	20 16	15 10	9 5	4 0
sf	1	1	0	1	0	1	0	shift	0	Rm	imm6	Rn	1 1 1 1 1
opc									N				Rd

32-bit variant

Applies when `sf == 0`.

```
TST <Wn>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
ANDS WZR, <Wn>, <Wm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

```
TST <Xn>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
ANDS XZR, <Xn>, <Xm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler symbols

`<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:

`LSL`	when `shift = 00`
`LSR`	when `shift = 01`
`ASR`	when `shift = 10`
`ROR`	when `shift = 11`

`<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

The description of ANDS (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.332 UBFIZ

Unsigned Bitfield Insert in Zeros copies a bitfield of `<width>` bits from the least significant bits of the source register to bit position `<lsb>` of the destination register, setting the destination bits above and below the bitfield to zero.

This instruction is an alias of the UBFM instruction. This means that:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.

31	30 29	28	27 26 25 24	23	22 21	16	15 10	9 5	4 0
sf	1 0	1	0 0 1 1	0	N	immr	imms	Rn	Rd
	opc								

32-bit variant

Applies when `sf == 0 && N == 0`.

```
UBFIZ <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
UBFM <Wd>, <Wn>, #(-<lsb> MOD 32), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

64-bit variant

Applies when `sf == 1 && N == 1`.

```
UBFIZ <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
UBFM <Xd>, <Xn>, #(-<lsb> MOD 64), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
`<lsb>`	For the 32-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31. For the 64-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.
`<width>`	For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>. For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.333 UBFM

Unsigned Bitfield Move is usually accessed via one of its aliases, which are always preferred for disassembly.

If `<imms>` is greater than or equal to `<immr>`, this copies a bitfield of (`<imms>`-`<immr>`+1) bits starting from bit position `<immr>` in the source register to the least significant bits of the destination register.

If `<imms>` is less than `<immr>`, this copies a bitfield of (`<imms>`+1) bits from the least significant bits of the source register to bit position (regsize-`<immr>`) of the destination register, where regsize is the destination register size of 32 or 64 bits.

In both cases the destination bits below and above the bitfield are set to zero.

This instruction is used by the aliases LSL (immediate), LSR (immediate), UBFIZ, UBFX, UXTB, and UXTH. See *Alias conditions* on page C6-1377 for details of when each alias is preferred.

31	30 29	28 27 26 25 24	23	22 21	16 15	10 9	5 4	0
sf	1 0	1 0 0 1 1 0	N	immr	imms	Rn	Rd	
	opc							

32-bit variant

Applies when `sf == 0 && N == 0`.

```
UBFM <Wd>, <Wn>, #<immr>, #<imms>
```

64-bit variant

Applies when `sf == 1 && N == 1`.

```
UBFM <Xd>, <Xn>, #<immr>, #<imms>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;

integer R;
bits(datasize) wmask;
bits(datasize) tmask;

if sf == '1' && N != '1' then UNDEFINED;
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then UNDEFINED;

R = UInt(immr);
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Alias conditions

Alias	of variant	is preferred when
LSL (immediate)	32-bit	`imms != '011111' && imms + 1 == immr`
LSL (immediate)	64-bit	`imms != '111111' && imms + 1 == immr`
LSR (immediate)	32-bit	`imms == '011111'`
LSR (immediate)	64-bit	`imms == '111111'`
UBFIZ	-	`UInt(imms) < UInt(immr)`
UBFX	-	`BFXPreferred(sf, opc<1>, imms, immr)`
UXTB	-	`immr == '000000' && imms == '000111'`
UXTH	-	`immr == '000000' && imms == '001111'`

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

`<immr>` For the 32-bit variant: is the right rotate amount, in the range 0 to 31, encoded in the "immr" field.

For the 64-bit variant: is the right rotate amount, in the range 0 to 63, encoded in the "immr" field.

`<imms>` For the 32-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field.

For the 64-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the "imms" field.

Operation

```
bits(datasize) src = X[n];

// perform bitfield move on low bits
bits(datasize) bot = ROR(src, R) AND wmask;

// combine extension bits and result bits
X[d] = bot AND tmask;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.334 UBFX

Unsigned Bitfield Extract copies a bitfield of `<width>` bits starting from bit position `<lsb>` in the source register to the least significant bits of the destination register, and sets destination bits above the bitfield to zero.

This instruction is an alias of the UBFM instruction. This means that:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.

31	30 29	28	27 26 25 24	23	22 21	16	15 10	9 5	4 0
sf	1 0	1	0 0 1 1	0	N	immr	imms	Rn	Rd
	opc								

32-bit variant

Applies when `sf == 0 && N == 0`.

```
UBFX <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
UBFM <Wd>, <Wn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `BFXPreferred(sf, opc<1>, imms, immr)`.

64-bit variant

Applies when `sf == 1 && N == 1`.

```
UBFX <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
UBFM <Xd>, <Xn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `BFXPreferred(sf, opc<1>, imms, immr)`.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
`<lsb>`	For the 32-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31. For the 64-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.
`<width>`	For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>. For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.335 UDF

Permanently Undefined generates an Undefined Instruction exception (ESR_ELx.EC = 0b000000). The encodings for UDF used in this section are defined as permanently UNDEFINED in the Armv8-A architecture.

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 … 0
0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	imm16

Integer variant

```
UDF #<imm>
```

Decode for this encoding

```
// The imm16 field is ignored by hardware.
UNDEFINED;
```

Assembler symbols

`<imm>` is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field. The PE ignores the value of this constant.

Operation

```
// No operation.
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.336 UDIV

Unsigned Divide divides an unsigned integer register value by another unsigned integer register value, and writes the result to the destination register. The condition flags are not affected.

31	30	29	28 27 26 25 24	23 22 21 20	16 15	14 13 12	11	10	9 5	4 0
sf	0	0	1 1 0 1 0	1 1 0	Rm	0 0 0	0 1	0	Rn	Rd
								o1		

32-bit variant

Applies when `sf == 0`.

```
UDIV <Wd>, <Wn>, <Wm>
```

64-bit variant

Applies when `sf == 1`.

```
UDIV <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
```

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Wm>`	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
integer result;

if IsZero(operand2) then
    result = 0;
else
    result = RoundTowardsZero(Real(Int(operand1, TRUE)) / Real(Int(operand2, TRUE)));

X[d] = result<datasize-1:0>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.337 UMADDL

Unsigned Multiply-Add Long multiplies two 32-bit register values, adds a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias UMULL. See *Alias conditions* for details of when each alias is preferred.

31	30 29	28 27 26 25	24	23	22 21	20 … 16	15	14 … 10	9 … 5	4 … 0
1	0 0	1 1 0 1	1	1	0 1	Rm	0	Ra	Rn	Rd
				U			o0			

64-bit variant

```
UMADDL <Xd>, <Wn>, <Wm>, <Xa>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
```

Alias conditions

Alias	is preferred when
UMULL	`Ra == '11111'`

Assembler symbols

`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
`<Wm>`	Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
`<Xa>`	Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation

```
bits(32) operand1 = X[n];
bits(32) operand2 = X[m];
bits(64) operand3 = X[a];

integer result;

result = Int(operand3, TRUE) + (Int(operand1, TRUE) * Int(operand2, TRUE));

X[d] = result<63:0>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.338 UMNEGL

Unsigned Multiply-Negate Long multiplies two 32-bit register values, negates the product, and writes the result to the 64-bit destination register.

This instruction is an alias of the UMSUBL instruction. This means that:

- The encodings in this description are named to match the encodings of UMSUBL.
- The description of UMSUBL gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	10	9	5	4	0
1	0	0	1	1	0	1	1	1	0	1	Rm		1	1	1	1	1	Rn	Rd
								U					o0		Ra				

64-bit variant

```
UMNEGL <Xd>, <Wn>, <Wm>
```

is equivalent to

```
UMSUBL <Xd>, <Wn>, <Wm>, XZR
```

and is always the preferred disassembly.

Assembler symbols

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of UMSUBL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.339 UMSUBL

Unsigned Multiply-Subtract Long multiplies two 32-bit register values, subtracts the product from a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias UMNEGL. See *Alias conditions* for details of when each alias is preferred.

31	30 29	28 27 26 25	24	23	22 21	20 … 16	15	14 … 10	9 … 5	4 … 0
1	0 0	1 1 0 1	1	1	0 1	Rm	1	Ra	Rn	Rd
				U			o0			

64-bit variant

```
UMSUBL <Xd>, <Wn>, <Wm>, <Xa>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
```

Alias conditions

Alias	is preferred when
UMNEGL	`Ra == '11111'`

Assembler symbols

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

`<Xa>` Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

Operation

```
bits(32) operand1 = X[n];
bits(32) operand2 = X[m];
bits(64) operand3 = X[a];

integer result;

result = Int(operand3, TRUE) - (Int(operand1, TRUE) * Int(operand2, TRUE));
X[d] = result<63:0>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.340 UMULH

Unsigned Multiply High multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.

31	30 29	28	27 26 25 24	23	22 21 20	16	15	14 10	9 5	4 0
1	0 0	1	1 0 1 1	1	1 0	Rm	0	(1) (1) (1) (1) (1)	Rn	Rd
				U				Ra		

64-bit variant

```
UMULH <Xd>, <Xn>, <Xm>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

```
bits(64) operand1 = X[n];
bits(64) operand2 = X[m];

integer result;

result = Int(operand1, TRUE) * Int(operand2, TRUE);

X[d] = result<127:64>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.341 UMULL

Unsigned Multiply Long multiplies two 32-bit register values, and writes the result to the 64-bit destination register.

This instruction is an alias of the UMADDL instruction. This means that:

- The encodings in this description are named to match the encodings of UMADDL.
- The description of UMADDL gives the operational pseudocode for this instruction.

31	30 29	28	27 26 25 24	23	22 21	20 … 16	15	14 … 10	9 … 5	4 … 0
1	0 0	1	1 0 1 1	1	0 1	Rm	0	1 1 1 1 1	Rn	Rd
				U			o0	Ra		

64-bit variant

```
UMULL <Xd>, <Wn>, <Wm>
```

is equivalent to

```
UMADDL <Xd>, <Wn>, <Wm>, XZR
```

and is always the preferred disassembly.

Assembler symbols

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of UMADDL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.342 UXTB

Unsigned Extend Byte extracts an 8-bit value from a register, zero-extends it to the size of the register, and writes the result to the destination register.

This instruction is an alias of the UBFM instruction. This means that:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.

31	30 29	28 27 26 25 24	23	22 21 16	15 10	9 5	4 0
0	1 0	1 0 0 1 1 0	0	0 0 0 0 0 0	0 0 0 1 1 1	Rn	Rd
sf	opc		N	immr	imms		

32-bit variant

```
UXTB <Wd>, <Wn>
```

is equivalent to

```
UBFM <Wd>, <Wn>, #0, #7
```

and is always the preferred disassembly.

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.343 UXTH

Unsigned Extend Halfword extracts a 16-bit value from a register, zero-extends it to the size of the register, and writes the result to the destination register.

This instruction is an alias of the UBFM instruction. This means that:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.

31	30 29	28 27 26 25 24	23	22 21 16	15 10	9 5	4 0
0	1 0	1 0 0 1 1 0	0	0 0 0 0 0 0	0 0 1 1 1 1	Rn	Rd
sf	opc		N	immr	imms		

32-bit variant

```
UXTH <Wd>, <Wn>
```

is equivalent to

```
UBFM <Wd>, <Wn>, #0, #15
```

and is always the preferred disassembly.

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.344 WFE

Wait For Event is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. Wakeup events include the event signaled as a result of executing the `SEV` instruction on any PE in the multiprocessor system. For more information, see *Wait for Event mechanism and Send event* on page D1-2397.

As described in *Wait for Event mechanism and Send event* on page D1-2397, the execution of a `WFE` instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level. See:

- *Traps to EL1 of EL0 execution of WFE and WFI instructions* on page D1-2377.
- *Traps to EL2 of EL0 and EL1 execution of WFE and WFI instructions* on page D1-2386.
- *Traps to EL3 of EL2, EL1, and EL0 execution of WFE and WFI instructions* on page D1-2393.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11			8	7		5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	1	0	1	1	1	1	1
																				CRm				op2							

System variant

```
WFE
```

Decode for this encoding

```
// Empty.
```

Operation

```
if IsEventRegisterSet() then
    ClearEventRegister();
else
    if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFxTrap(EL1, TRUE);
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFxTrap(EL2, TRUE);
    if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFxTrap(EL3, TRUE);
    WaitForEvent();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.345 WFI

Wait For Interrupt is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. For more information, see *Wait For Interrupt* on page D1-2400.

As described in *Wait For Interrupt* on page D1-2400, the execution of a `WFI` instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level. See:

- *Traps to EL1 of EL0 execution of WFE and WFI instructions* on page D1-2377.
- *Traps to EL2 of EL0 and EL1 execution of WFE and WFI instructions* on page D1-2386.
- *Traps to EL3 of EL2, EL1, and EL0 execution of WFE and WFI instructions* on page D1-2393.

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 8	7 5	4	3 2 1 0
1 1 0 1	0 1 0 1	0 0 0 0	0 1 1 0	0 0 1 0	0 0 0 0	0 1 1	1	1 1 1 1
					CRm	op2		

System variant

```
WFI
```

Decode for this encoding

```
// Empty.
```

Operation

```
if !InterruptPending() then
    if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFxTrap(EL1, FALSE);
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFxTrap(EL2, FALSE);
    if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFxTrap(EL3, FALSE);
    WaitForInterrupt();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.346 XAFLAG

Convert floating-point condition flags from external format to Arm format. This instruction converts the state of the PSTATE.{N,Z,C,V} flags from an alternative representation required by some software to a form representing the result of an Arm floating-point scalar compare instruction.

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11			8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	1	0	0	(0)	(0)	(0)	(0)	0	0	1	1	1	1	1	1
																				CRm											

System variant

```
XAFLAG
```

Decode for this encoding

```
if !HaveFlagFormatExt() then UNDEFINED;
```

Operation

```
bit N = NOT(PSTATE.C) AND NOT(PSTATE.Z);
bit Z = PSTATE.Z AND PSTATE.C;
bit C = PSTATE.C OR PSTATE.Z;
bit V = NOT(PSTATE.C) AND PSTATE.Z;

PSTATE.N = N;
PSTATE.Z = Z;
PSTATE.C = C;
PSTATE.V = V;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.347 XPACD, XPACI, XPACLRI

Strip Pointer Authentication Code. This instruction removes the pointer authentication code from an address. The address is in the specified general-purpose register for `XPACI` and `XPACD`, and is in LR for `XPACLRI`.

The `XPACD` instruction is used for data addresses, and `XPACI` and `XPACLRI` are used for instruction addresses.

Integer

ARMv8.3

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9			5	4		0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	1	0	0	0	D	1	1	1	1	1	Rd	
																							Rn					

XPACD variant

Applies when `D == 1`.

```
XPACD <Xd>
```

XPACI variant

Applies when `D == 0`.

```
XPACI <Xd>
```

Decode for all variants of this encoding

```
boolean data = (D == '1');
integer d = UInt(Rd);

if !HavePACExt() then
    UNDEFINED;
```

System

ARMv8.3

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	1	1	1	1	1	1	1	1

System variant

```
XPACLRI
```

Decode for this encoding

```
integer d = 30;
boolean data = FALSE;
```

Assembler symbols

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

Operation for all encodings

```
if HavePACExt() then
    X[d] = Strip(X[d], data);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C6.2.348 YIELD

YIELD is a hint instruction. Software with a multithreading capability can use a `YIELD` instruction to indicate to the PE that it is performing a task, for example a spin-lock, that could be swapped out to improve overall system performance. The PE can use this hint to suspend and resume multiple software threads if it supports the capability.

For more information about the recommended use of this instruction, see *The YIELD instruction* on page B1-112.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11			8	7		5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	1	1	1	1	1	1
																				CRm				op2							

System variant

```
YIELD
```

Decode for this encoding

```
// Empty.
```

Operation

```
Hint_Yield();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter C7
A64 Advanced SIMD and Floating-point Instruction Descriptions

This chapter describes the A64 Advanced SIMD and floating-point instructions.

It contains the following sections:

- *About the A64 SIMD and floating-point instructions* on page C7-1398.
- *Alphabetical list of A64 Advanced SIMD and floating-point instructions* on page C7-1400.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.1 About the A64 SIMD and floating-point instructions

Alphabetical list of A64 Advanced SIMD and floating-point instructions on page C7-1400 gives full descriptions of the A64 instructions that are in the following instruction groups:

- Loads and store instructions associated with the SIMD and floating-point registers.
- Data processing instructions with SIMD and floating-point registers.

A64 instruction set encoding on page C4-268 in the A64 Instruction Encodings chapter provides an overview of the instruction encodings as part of an instruction class within a functional group.

The rest of this section is a general description of the SIMD and floating-point instructions. It contains the following subsections:

- *Register size*.
- *Data types*.
- *Condition flags and related instructions* on page C7-1399.
- *General capabilities* on page C7-1399.

C7.1.1 Register size

A64 provides a comprehensive set of packed Single Instruction Multiple Data (SIMD) and scalar operations using data held in the 32 entry 128-bit wide SIMD and floating-point register file.

Each SIMD and floating-point register can be used to hold:

- A single scalar value of the floating-point or integer type.
- A 64-bit wide vector containing one or more elements.
- A 128-bit wide vector containing two or more elements.

Where the entire 128-bit wide register is not fully utilized, the vector or scalar quantity is held in the least significant bits of the register, with the most significant bits being cleared to zero on a write, see *Vector formats* on page A1-41.

The following instructions can insert data into individual elements within a SIMD and floating-pointer register without clearing the remaining bits to zero:

- Insert vector element from another vector element or general-purpose register, `INS`.
- Load structure into a single lane, for example `LD3`.
- All second-part narrowing operations, for example `SHRN2`.

C7.1.2 Data types

The A64 instruction set provides support for arithmetic, conversion, and bitwise operations on:

- Half-precision, single-precision, and double-precision floating-points.
- Signed and unsigned integers.
- Polynomials over {0, 1}.
- When ARMv8.3-CompNum is implemented, complex numbers.

For all AArch64 floating-point operations, including SIMD operations, the rounding mode and exception trap handling are controlled by the FPCR.

Note

- AArch32 Advanced SIMD operations always use Arm standard floating-point arithmetic, regardless of the rounding mode specified by the AArch64 FPCR or the AArch32 FPSCR.
- In AArch64 state, floating-point multiply-add operations are always performed as fused operations, but AArch32 state provides both fused and chained multiply-add instructions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In addition to operations that consume and produce values of the same width and type, the A64 instruction set supports SIMD and scalar operations that produce a wider or narrower vector result:

- Where a SIMD operation narrows a 128-bit vector to a 64-bit vector, the A64 instruction set provides a second-part operation, for example `SHRN2`, that can pack the result of a second operation into the upper part of the same destination register.
- Where a SIMD operation widens a 64-bit vector to a 128-bit vector, the A64 instruction set provides a second-part operation, for example `SMLAL2`, that can extract the source from the upper 64 bits of the source registers.

All SIMD operations that could produce side-effects that are not limited to the destination SIMD and floating-point register, for example a potential update of FPSR.Q or FPSR.IDC, have a dedicated scalar variant to support the use of SIMD with loops requiring specialised head or tail handling, or both.

C7.1.3 Condition flags and related instructions

The A64 instruction set provides support for flag setting and conditional operations on the SIMD and floating-point register file:

- Floating-point `FCSEL` and `FCCMP` instructions are equivalent to the integer `CSEL` and `CCMP` instructions.
- Floating-point `FCMP`, `FCMPE`, `FCCMP`, and `FCCMP` instructions set the PSTATE.{N, Z, C, V} flags based on the result of the floating-point comparison.
- Floating-point `FJCVTZS` instruction sets the PSTATE.Z flag if the result of the conversion, when converted back to a double-precision floating-point number, gives precisely the same value as the original. Other PSTATE flags are cleared by this instruction.
- Floating-point and integer instructions provide a means of producing either a scalar or a vector mask based on a comparison in a SIMD and floating-point register, for example `FCMEQ`.

Note

`FCMP` and `FCMPE` differ from the A32/T32 `VCMP` and `VCMPE` instructions, which use the dedicated FPSCR.NZCV field for the result. A64 instructions store the result of an `FCMP` or `FCMPE` operation in the PSTATE.{N, Z, C, V} field.

If ARMv8.5-CondM is implemented, base instructions XAFLAG and AXFLAG convert between the PSTATE condition flag format used by the FCMP instruction and an alternative format. See Table C6-1 on page C6-774.

C7.1.4 General capabilities

A64 SIMD and floating-point instructions provide the following capabilities:

- General arithmetic on vector and scalar floating-point and integer values.
- Dedicated polynomial multiply over {0, 1}.
- Vector and scalar fused multiply-addition of single-precision and double-precision floating-points, and for half-precision floating-points when ARMv8.2-FP16 is implemented.
- Load and store of single and pairs of SIMD and floating-point registers.
- Load and store of structures and individual lanes of between one and four SIMD and floating-point registers.
- Direct conversion between 64-bit integers and floating-point values, with explicit rounding.
- When ARMv8.2-BF16 is implemented, support for BFloat16 format, including dot product and matrix multiplication.
- When ARMv8.2-I8MM is implemented, integer matrix multiply-accumulate and mixed sign dot product.
- When ARMv8.3-JSconv is implemented, conversion from double precision floating-point values to 32-bit integers, with rounding to zero.
- Double-rounding free conversion between double-precision and half-precision floating-point values.
- Comprehensive SIMD with widening and narrowing support.
- Vector to scalar reduction returning the minimum or maximum value, or the sum.
- Floating-point to nearest integer in floating-point format.
- When ARMv8.3-CompNum is implemented, complex number arithmetic.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2 Alphabetical list of A64 Advanced SIMD and floating-point instructions

This section lists every section in the Advanced SIMD and floating-point categories of the A64 instruction set. For details of the format used, see *Structure of the A64 assembler language* on page C1-183.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.1 ABS

Absolute value (vector). This instruction calculates the absolute value of each vector element in the source SIMD&FP register, puts the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	size		1	0	0	0	0	0	1	0	1	1	1	0	Rn		Rd	
		U																							

Scalar variant

```
ABS <V><d>, <V><n>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean neg = (U == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	size		1	0	0	0	0	0	1	0	1	1	1	0	Rn		Rd	
		U																							

Vector variant

```
ABS <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

D when size = 11

The following encodings are reserved:

- size = 0x.
- size = 10.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

`<n>` Is the number of the SIMD&FP source register, encoded in the "Rn" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B`	when `size = 00`, `Q = 0`
`16B`	when `size = 00`, `Q = 1`
`4H`	when `size = 01`, `Q = 0`
`8H`	when `size = 01`, `Q = 1`
`2S`	when `size = 10`, `Q = 0`
`4S`	when `size = 10`, `Q = 1`
`2D`	when `size = 11`, `Q = 1`

The encoding `size = 11`, `Q = 0` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);
    Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.2 ADD (vector)

Add (vector). This instruction adds corresponding elements in the two source SIMD&FP registers, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31 30	29	28 27 26 25 24	23 22	21	20 16	15 14 13 12	11 10	9 5	4 0
0 1	0	1 1 1 1 0	size	1	Rm	1 0 0 0 0	1	Rn	Rd
	U								

Scalar variant

```
ADD <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (U == '1');
```

Vector

31	30	29	28 27 26 25 24	23 22	21	20 16	15 14 13 12	11 10	9 5	4 0
0	Q	0	0 1 1 1 0	size	1	Rm	1 0 0 0 0	1	Rn	Rd
		U								

Vector variant

```
ADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (U == '1');
```

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

D when size = 11

The following encodings are reserved:

- size = 0x.
- size = 10.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<m>` Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B`	when `size = 00`, `Q = 0`
`16B`	when `size = 00`, `Q = 1`
`4H`	when `size = 01`, `Q = 0`
`8H`	when `size = 01`, `Q = 1`
`2S`	when `size = 10`, `Q = 0`
`4S`	when `size = 10`, `Q = 1`
`2D`	when `size = 11`, `Q = 1`

The encoding `size = 11`, `Q = 0` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then
        Elem[result, e, esize] = element1 - element2;
    else
        Elem[result, e, esize] = element1 + element2;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.3 ADDHN, ADDHN2

Add returning High Narrow. This instruction adds each vector element in the first source SIMD&FP register to the corresponding vector element in the second source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register.

The results are truncated. For rounded results, see RADDHN, RADDHN2.

The `ADDHN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `ADDHN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	size	1	Rm	0	1	0	0	0	0	Rn	Rd
		U											o1					

Three registers, not all the same type variant

```
ADDHN{2} <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Tb>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B` when `size = 00`, `Q = 0`

`16B` when `size = 00`, `Q = 1`

`4H` when `size = 01`, `Q = 0`

`8H` when `size = 01`, `Q = 1`

`2S` when `size = 10`, `Q = 0`

`4S` when `size = 10`, `Q = 1`

The encoding `size = 11`, `Q = x` is reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

`8H` when `size = 00`

`4S` when `size = 01`

`2D` when `size = 10`

The encoding `size = 11` is reserved.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(2*datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
bits(2*esize) element1;
bits(2*esize) element2;
bits(2*esize) sum;

for e = 0 to elements-1
    element1 = Elem[operand1, e, 2*esize];
    element2 = Elem[operand2, e, 2*esize];
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    sum = sum + round_const;
    Elem[result, e, esize] = sum<2*esize-1:esize>;

Vpart[d, part] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.4 ADDP (scalar)

Add Pair of elements (scalar). This instruction adds two vector elements in the source SIMD&FP register and writes the scalar result into the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30	29	28 27 26 25 24	23 22	21 20 19 18 17	16 15 14 13 12	11 10	9 5	4 0
0 1	0	1 1 1 1 0	size	1 1 0 0 0	1 1 0 1 1	1 0	Rn	Rd

Advanced SIMD variant

```
ADDP <V><d>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;

integer esize = 8 << UInt(size);
integer datasize = esize * 2;
```

Assembler symbols

<V> Is the destination width specifier, encoded in the "size" field. It can have the following values:

D when size = 11

The following encodings are reserved:

- size = 0x.
- size = 10.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is the source arrangement specifier, encoded in the "size" field. It can have the following values:

2D when size = 11

The following encodings are reserved:

- size = 0x.
- size = 10.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_ADD, operand, esize);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.5 ADDP (vector)

Add Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, adds each pair of values together, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	size	1	Rm	1	0	1	1	1	1	Rn	Rd

Three registers of the same type variant

```
ADDP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B`	when `size = 00`, `Q = 0`
`16B`	when `size = 00`, `Q = 1`
`4H`	when `size = 01`, `Q = 0`
`8H`	when `size = 01`, `Q = 1`
`2S`	when `size = 10`, `Q = 0`
`4S`	when `size = 10`, `Q = 1`
`2D`	when `size = 11`, `Q = 1`

The encoding `size = 11`, `Q = 0` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
for e = 0 to elements-1
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
    Elem[result, e, esize] = element1 + element2;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.6 ADDV

Add across Vector. This instruction adds every vector element in the source SIMD&FP register together, and writes the scalar result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	size		1	1	0	0	0	1	1	0	1	1	1	0	Rn		Rd	

Advanced SIMD variant

```
ADDV <V><d>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler symbols

<V> Is the destination width specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10

The encoding size = 11 is reserved.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 4S when size = 10, Q = 1

The following encodings are reserved:

- size = 10, Q = 0.
- size = 11, Q = x.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_ADD, operand, esize);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.7 AESD

AES single round decryption.

31 30 29 28	27 26 25 24	23 22	21 20 19 18 17 16	15 14 13 12	11 10	9 … 5	4 … 0
0 1 0 0	1 1 1 0	0 0	1 0 1 0 0	0 0 1 0	1 1 0	Rn	Rd
				D			

Advanced SIMD variant

```
AESD <Vd>.16B, <Vn>.16B
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAESExt() then UNDEFINED;
```

Assembler symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.

<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) result;
result = operand1 EOR operand2;
result = AESInvSubBytes(AESInvShiftRows(result));
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.8 AESE

AES single round encryption.

31 30 29 28	27 26 25 24	23 22	21 20	19 18 17 16	15 14 13 12	11	10	9 … 5	4 … 0
0 1 0 0	1 1 1 0	0 0	1 0	1 0 0 0	0 0 1 0	0	1 0	Rn	Rd
						D			

Advanced SIMD variant

```
AESE <Vd>.16B, <Vn>.16B
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAESExt() then UNDEFINED;
```

Assembler symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.

<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) result;
result = operand1 EOR operand2;
result = AESSubBytes(AESShiftRows(result));

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.9 AESIMC

AES inverse mix columns.

31 30 29 28	27 26 25 24	23 22	21 20 19 18 17	16 15 14 13	12	11 10	9 … 5	4 … 0
0 1 0 0	1 1 1 0	0 0	1 0 1 0 0	0 0 1 1	1 (D)	1 0	Rn	Rd

Advanced SIMD variant

```
AESIMC <Vd>.16B, <Vn>.16B
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAESExt() then UNDEFINED;
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand = V[n];
bits(128) result;
result = AESInvMixColumns(operand);
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.10 AESMC

AES mix columns.

31 30 29 28	27 26 25 24	23 22	21 20 19 18 17	16 15 14 13	12	11 10	9 … 5	4 … 0
0 1 0 0	1 1 1 0	0 0	1 0 1 0 0	0 0 1 1	0	1 0	Rn	Rd
					D			

Advanced SIMD variant

```
AESMC <Vd>.16B, <Vn>.16B
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAESExt() then UNDEFINED;
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand = V[n];
bits(128) result;
result = AESMixColumns(operand);
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.11 AND (vector)

Bitwise AND (vector). This instruction performs a bitwise AND between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20–16	15	14	13	12	11	10	9–5	4–0
0	Q	0	0	1	1	1	0	0	0	1	Rm	0	0	0	1	1	1	Rn	Rd
								size											

Three registers of the same type variant

```
AND <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `8B` when `Q = 0`
- `16B` when `Q = 1`

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

result = operand1 AND operand2;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.12 BCAX

Bit Clear and Exclusive OR performs a bitwise AND of the 128-bit vector in a source SIMD&FP register and the complement of the vector in another source SIMD&FP register, then performs a bitwise exclusive OR of the resulting vector and the vector in a third source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction is implemented only when ARMv8.2-SHA is implemented.

ARMv8.2

31 30 29 28	27 26 25 24	23 22 21 20	16	15 14	10 9	5 4	0
1 1 0 0	1 1 1 0 0	0 1	Rm	0	Ra	Rn	Rd

Advanced SIMD variant

```
BCAX <Vd>.16B, <Vn>.16B, <Vm>.16B, <Va>.16B
```

Decode for this encoding

```
if !HaveSHA3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer a = UInt(Ra);
```

Assembler symbols

<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn>	Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
<Va>	Is the name of the third SIMD&FP source register, encoded in the "Ra" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Va = V[a];
V[d] = Vn EOR (Vm AND NOT(Va));
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.13 BFCVT

Floating-point convert from single-precision to BFloat16 format (scalar) converts the single-precision floating-point value in the 32-bit SIMD&FP source register to BFloat16 format and writes the result in the 16-bit SIMD&FP destination register.

Unlike the BFloat16 multiplication instructions, this instruction honors all the control bits in the FPCR that apply to single-precision arithmetic, including the rounding mode. This instruction can generate a floating-point exception that causes a cumulative exception bit in the FPSR to be set, or a synchronous exception to be taken, depending on the enable bits in the FPCR. ID_AA64ISAR1_EL1.BF16 indicates whether this instruction is supported.

ARMv8.6

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	0	0	1	1	1	1	0	0	1	1	0	0	0	1	1	0	1	0	0	0	0	Rn		Rd	

Single-precision to BFloat16 variant

```
BFCVT <Hd>, <Sn>
```

Decode for this encoding

```
if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer d = UInt(Rd);
```

Assembler symbols

`<Hd>`	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Sn>`	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(32) operand = V[n];
bits(16) result;

result = FPConvertBF(operand, FPCR);
V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.14 BFCVTN, BFCVTN2

Floating-point convert from single-precision to BFloat16 format (vector) reads each single-precision element in the SIMD&FP source vector, converts each value to BFloat16 format, and writes the results in the lower or upper half of the SIMD&FP destination vector. The result elements are half the width of the source elements.

The BFCVTN instruction writes the half-width results to the lower half of the destination vector and clears the upper half to zero, while the BFCVTN2 instruction writes the results to the upper half of the destination vector without affecting the other bits in the register.

Unlike the BFloat16 multiplication instructions, this instruction honors all of the control bits in the FPCR that apply to single-precision arithmetic, including the rounding mode. It can also generate a floating-point exception that causes cumulative exception bits in the FPSR to be set, or a synchronous exception to be taken, depending on the enable bits in the FPCR.

ARMv8.6

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	1	0	1	0	0	0	0	1	0	1	1	0	1	0	Rn		Rd	

Vector single-precision and BFloat16 variant

```
BFCVTN{2} <Vd>.<Ta>, <Vn>.4S
```

Decode for this encoding

```
if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer d = UInt(Rd);
integer part = UInt(Q);
integer elements = 64 DIV 16;
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`4H` when `Q = 0`

`8H` when `Q = 1`

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(128) operand = V[n];
bits(64) result;

for e = 0 to elements-1
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    Elem[result, e, 16] = FPConvertBF(Elem[operand, e, 32], FPCR);

Vpart[d, part] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.15 BFDOT (by element)

BFloat16 floating-point dot product (vector, by element). This instruction delimits the source vectors into pairs of 16-bit BF16 elements. Each pair of elements in the first source vector is multiplied by the specified pair of elements in the second source vector. The resulting single-precision products are then summed and added destructively to the single-precision element of the destination vector that aligns with the pair of BF16 values in the first source vector. The instruction ignores the FPCR and does not update the FPSR exception status.

The BF16 pair within the second source vector is specified using an immediate index. The index range is from 0 to 3 inclusive. ID_AA64ISAR1_EL1.BF16 indicates whether this instruction is supported.

ARMv8.6

31	30	29	28	27	26	25	24	23	22	21	20	19 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	1	0	1	L	M	Rm	1	1	1	1	H	0	Rn	Rd

Vector variant

```
BFDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.2H[<index>]
```

Decode for this encoding

```
if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer d = UInt(Rd);
integer i = UInt(H:L);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `2S` when `Q = 0`
- `4S` when `Q = 1`

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `4H` when `Q = 0`
- `8H` when `Q = 1`

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

`<index>` Is the immediate index of a pair of 16-bit elements in the range 0 to 3, encoded in the "H:L" fields.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(128) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;

for e = 0 to elements-1
    bits(16) elt1_a = Elem[operand1, 2*e+0, 16];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    bits(16) elt1_b = Elem[operand1, 2*e+1, 16];
    bits(16) elt2_a = Elem[operand2, 2*i+0, 16];
    bits(16) elt2_b = Elem[operand2, 2*i+1, 16];

    bits(32) sum = BFAdd(BFMul(elt1_a, elt2_a), BFMul(elt1_b, elt2_b));
    Elem[result, e, 32] = BFAdd(Elem[operand3, e, 32], sum);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.16 BFDOT (vector)

BFloat16 floating-point dot product (vector). This instruction delimits the source vectors into pairs of 16-bit BF16 elements. Within each pair, the elements in the first source vector are multiplied by the corresponding elements in the second source vector. The resulting single-precision products are then summed and added destructively to the single-precision element of the destination vector that aligns with the pair of BF16 values in the first source vector. The instruction ignores the FPCR and does not update the FPSR exception status.

ARMv8.6

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	0	1	0	Rm	1	1	1	1	1	1	Rn	Rd

Vector variant

```
BFDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `2S` when `Q = 0`
- `4S` when `Q = 1`

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `4H` when `Q = 0`
- `8H` when `Q = 1`

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;

for e = 0 to elements-1
    bits(16) elt1_a = Elem[operand1, 2*e+0, 16];
    bits(16) elt1_b = Elem[operand1, 2*e+1, 16];
    bits(16) elt2_a = Elem[operand2, 2*e+0, 16];
    bits(16) elt2_b = Elem[operand2, 2*e+1, 16];

    bits(32) sum = BFAdd(BFMul(elt1_a, elt2_a), BFMul(elt1_b, elt2_b));
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    Elem[result, e, 32] = BFAdd(Elem[operand3, e, 32], sum);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.17 BFMLALB, BFMLALT (by element)

BFloat16 floating-point widening multiply-add long (by element) widens the even-numbered (bottom) or odd-numbered (top) 16-bit elements in the first source vector, and the indexed element in the second source vector from Bfloat16 to single-precision format. The instruction then multiplies and adds these values to the overlapping single-precision elements of the destination vector.

This performs a fused multiply-add without intermediate rounding that honors all of the control bits in the FPCR that apply to single-precision arithmetic, including the rounding mode. It can also generate a floating-point exception that causes cumulative exception bits in the FPSR to be set, or a synchronous exception to be taken, depending on the enable bits in the FPCR. ID_AA64ISAR1_EL1.BF16 indicates whether this instruction is supported.

ARMv8.6

31	30	29	28	27	26	25	24	23	22	21	20	19 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	1	1	1	L	M	Rm	1	1	1	1	H	0	Rn	Rd

Vector variant

```
BFMLAL<bt> <Vd>.4S, <Vn>.8H, <Vm>.H[<index>]
```

Decode for this encoding

```
if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt('0':Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer elements = 128 DIV 32;
integer sel = UInt(Q);
```

Assembler symbols

`<bt>` Is the bottom or top element specifier, encoded in the "Q" field. It can have the following values:

- `B` when `Q = 0`
- `T` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

`<index>` Is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(128) operand1 = V[n];
bits(128) operand2 = V[m];
bits(128) operand3 = V[d];
bits(128) result;

bits(32) element2 = Elem[operand2, index, 16]:Zeros(16);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
for e = 0 to elements-1
    bits(32) element1 = Elem[operand1, 2*e+sel, 16]:Zeros(16);
    bits(32) addend = Elem[operand3, e, 32];
    Elem[result, e, 32] = FPMulAdd(addend, element1, element2, FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.18 BFMLALB, BFMLALT (vector)

BFloat16 floating-point widening multiply-add long (vector) widens the even-numbered (bottom) or odd-numbered (top) 16-bit elements in the first and second source vectors from Bfloat16 to single-precision format. The instruction then multiplies and adds these values to the overlapping single-precision elements of the destination vector.

This performs a fused multiply-add without intermediate rounding that honors all of the control bits in the FPCR that apply to single-precision arithmetic, including the rounding mode. It can also generate a floating-point exception that causes cumulative exception bits in the FPSR to be set, or a synchronous exception to be taken, depending on the enable bits in the FPCR. ID_AA64ISAR1_EL1.BF16 indicates whether these instruction is supported.

ARMv8.6

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	1	1	0	Rm	1	1	1	1	1	1	Rn	Rd

Vector variant

```
BFMLAL<bt> <Vd>.4S, <Vn>.8H, <Vm>.8H
```

Decode for this encoding

```
if !HaveBF16Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer elements = 128 DIV 32;
integer sel = UInt(Q);
```

Assembler symbols

`<bt>` Is the bottom or top element specifier, encoded in the "Q" field. It can have the following values:

- `B` when `Q = 0`
- `T` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(128) operand1 = V[n];
bits(128) operand2 = V[m];
bits(128) operand3 = V[d];
bits(128) result;

for e = 0 to elements-1
    bits(32) element1 = Elem[operand1, 2*e+sel, 16]:Zeros(16);
    bits(32) element2 = Elem[operand2, 2*e+sel, 16]:Zeros(16);
    bits(32) addend = Elem[operand3, e, 32];
    Elem[result, e, 32] = FPMulAdd(addend, element1, element2, FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.19 BFMMLA

BFloat16 floating-point matrix multiply-accumulate into 2x2 matrix. This instruction multiplies the 2x4 matrix of BF16 values held in the first 128-bit source vector by the 4x2 BF16 matrix in the second 128-bit source vector. The resulting 2x2 single-precision matrix product is then added destructively to the 2x2 single-precision matrix in the 128-bit destination vector. This is equivalent to performing a 4-way dot product per destination element. The instruction ignores the FPCR and does not update the FPSR exception status.

Note

Arm expects that the BFMMLA instruction will deliver a peak BF16 multiply throughput that is at least as high as can be achieved using two BFDOT instructions, with a goal that it should have significantly higher throughput.

ARMv8.6

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	1	0	1	1	1	0	0	1	0	Rm	1	1	1	0	1	1	Rn	Rd

Vector variant

```
BFMMLA <Vd>.4S, <Vn>.8H, <Vm>.8H
```

Decode for this encoding

```
if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
```

Assembler symbols

<Vd>	Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(128) op1 = V[n];
bits(128) op2 = V[m];
bits(128) acc = V[d];

V[d] = BFMatMulAdd(acc, op1, op2);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.20 BIC (vector, immediate)

Bitwise bit Clear (vector, immediate). This instruction reads each vector element from the destination SIMD&FP register, performs a bitwise AND between each result and the complement of an immediate constant, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	0	0	0	0	a	b	c	x	x	x	1	0	1	d	e	f	g	h	Rd				
		op														cmode															

16-bit variant

Applies when `cmode == 10x1`.

```
BIC <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

32-bit variant

Applies when `cmode == 0xx1`.

```
BIC <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

Decode for all variants of this encoding

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
    when '0xx01' operation = ImmediateOp_MVNI;
    when '0xx11' operation = ImmediateOp_BIC;
    when '10x01' operation = ImmediateOp_MVNI;
    when '10x11' operation = ImmediateOp_BIC;
    when '110x1' operation = ImmediateOp_MVNI;
    when '1110x' operation = ImmediateOp_MOVI;
    when '11111'
        // FMOV Dn,#imm is in main FP instruction set
        if Q == '0' then UNDEFINED;
        operation = ImmediateOp_MOVI;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP register, encoded in the "Rd" field.

`<T>` For the 16-bit variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `4H` when `Q = 0`
- `8H` when `Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the 32-bit variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`2S` when `Q = 0`

`4S` when `Q = 1`

`<imm8>` Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

`<amount>` For the 16-bit variant: is the shift amount encoded in the "cmode<1>" field. It can have the following values:

`0` when `cmode<1> = 0`

`8` when `cmode<1> = 1`

defaulting to 0 if LSL is omitted.

For the 32-bit variant: is the shift amount encoded in the "cmode<2:1>" field. It can have the following values:

`0` when `cmode<2:1> = 00`

`8` when `cmode<2:1> = 01`

`16` when `cmode<2:1> = 10`

`24` when `cmode<2:1> = 11`

defaulting to 0 if LSL is omitted.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
    when ImmediateOp_MOVI
        result = imm;
    when ImmediateOp_MVNI
        result = NOT(imm);
    when ImmediateOp_ORR
        operand = V[rd];
        result = operand OR imm;
    when ImmediateOp_BIC
        operand = V[rd];
        result = operand AND NOT(imm);

V[rd] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.21 BIC (vector, register)

Bitwise bit Clear (vector, register). This instruction performs a bitwise AND between the first source SIMD&FP register and the complement of the second source SIMD&FP register, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	size		1	Rm		0	0	0	1	1	1	Rn		Rd	

Three registers of the same type variant

```
BIC <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `8B` when `Q = 0`
- `16B` when `Q = 1`

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

operand2 = NOT(operand2);

result = operand1 AND operand2;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.22 BIF

Bitwise Insert if False. This instruction inserts each bit from the first source SIMD&FP register into the destination SIMD&FP register if the corresponding bit of the second source SIMD&FP register is 0, otherwise leaves the bit in the destination register unchanged.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	1	1	1	Rm	0	0	0	1	1	1	Rn	Rd
								opc2											

Three registers of the same type variant

```
BIF <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `8B` when `Q = 0`
- `16B` when `Q = 1`

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand3;
bits(datasize) operand4 = V[n];

operand1 = V[d];
operand3 = NOT(V[m]);

V[d] = operand1 EOR ((operand1 EOR operand4) AND operand3);
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.23 BIT

Bitwise Insert if True. This instruction inserts each bit from the first source SIMD&FP register into the SIMD&FP destination register if the corresponding bit of the second source SIMD&FP register is 1, otherwise leaves the bit in the destination register unchanged.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20–16	15	14	13	12	11	10	9–5	4–0
0	Q	1	0	1	1	1	0	1	0	1	Rm	0	0	0	1	1	1	Rn	Rd
								opc2											

Three registers of the same type variant

```
BIT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `8B` when `Q = 0`
- `16B` when `Q = 1`

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand3;
bits(datasize) operand4 = V[n];

operand1 = V[d];
operand3 = V[m];
V[d] = operand1 EOR ((operand1 EOR operand4) AND operand3);
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.24 BSL

Bitwise Select. This instruction sets each bit in the destination SIMD&FP register to the corresponding bit from the first source SIMD&FP register when the original destination bit was 1, otherwise from the second source SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	0	1	1	Rm	0	0	0	1	1	1	Rn	Rd
								opc2											

Three registers of the same type variant

```
BSL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `8B` when `Q = 0`
- `16B` when `Q = 1`

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand3;
bits(datasize) operand4 = V[n];

operand1 = V[m];
operand3 = V[d];
V[d] = operand1 EOR ((operand1 EOR operand4) AND operand3);
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.25 CLS (vector)

Count Leading Sign bits (vector). This instruction counts the number of consecutive bits following the most significant bit that are the same as the most significant bit in each vector element in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The count does not include the most significant bit itself.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	17	16	15	14	13	12	11	10	9 5	4 0
0	Q	0	0	1	1	1	0	size	1	0	0	0	0	0	0	1	0	0	1	0	Rn	Rd
		U																				

Vector variant

```
CLS <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CountOp countop = if U == '1' then CountOp_CLZ else CountOp_CLS;
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B	when size = 00, Q = 0
16B	when size = 00, Q = 1
4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

integer count;
for e = 0 to elements-1
    if countop == CountOp_CLS then
        count = CountLeadingSignBits(Elem[operand, e, esize]);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        count = CountLeadingZeroBits(Elem[operand, e, esize]);
    Elem[result, e, esize] = count<esize-1:0>;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.26 CLZ (vector)

Count Leading Zero bits (vector). This instruction counts the number of consecutive zeros, starting from the most significant bit, in each vector element in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size	1	0	0	0	0	0	0	1	0	0	1	0	Rn	Rd
		U																				

Vector variant

```
CLZ <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CountOp countop = if U == '1' then CountOp_CLZ else CountOp_CLS;
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B	when size = 00, Q = 0
16B	when size = 00, Q = 1
4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

integer count;
for e = 0 to elements-1
    if countop == CountOp_CLS then
        count = CountLeadingSignBits(Elem[operand, e, esize]);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        count = CountLeadingZeroBits(Elem[operand, e, esize]);
    Elem[result, e, esize] = count<esize-1:0>;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.27 CMEQ (register)

Compare bitwise Equal (vector). This instruction compares each vector element from the first source SIMD&FP register with the corresponding vector element from the second source SIMD&FP register, and if the comparison is equal sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	1	1	1	1	1	0	size	1	Rm	1	0	0	0	1	1	Rn	Rd
		U																

Scalar variant

```
CMEQ <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean and_test = (U == '0');
```

Vector

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size	1	Rm	1	0	0	0	1	1	Rn	Rd
		U																

Vector variant

```
CMEQ <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean and_test = (U == '0');
```

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

D when size = 11

The following encodings are reserved:

- size = 0x.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- `size = 10`.

`<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<m>` Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B`	when `size = 00`, `Q = 0`
`16B`	when `size = 00`, `Q = 1`
`4H`	when `size = 01`, `Q = 0`
`8H`	when `size = 01`, `Q = 1`
`2S`	when `size = 10`, `Q = 0`
`4S`	when `size = 10`, `Q = 1`
`2D`	when `size = 11`, `Q = 1`

The encoding `size = 11`, `Q = 0` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if and_test then
        test_passed = !IsZero(element1 AND element2);
    else
        test_passed = (element1 == element2);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.28 CMEQ (zero)

Compare bitwise Equal to zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the value is equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	size		1	0	0	0	0	0	1	0	0	1	1	0	Rn		Rd	
		U																	op						

Scalar variant

```
CMEQ <V><d>, <V><n>, #0
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	size		1	0	0	0	0	0	1	0	0	1	1	0	Rn		Rd	
		U																	op						

Vector variant

```
CMEQ <Vd>.<T>, <Vn>.<T>, #0
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

D when size = 11

The following encodings are reserved:

- size = 0x.
- size = 10.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean test_passed;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.29 CMGE (register)

Compare signed Greater than or Equal (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first signed integer value is greater than or equal to the second signed integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31 30	29	28 27 26 25 24	23 22	21	20 16	15 14 13 12	11	10	9 5	4 0
0 1	0	1 1 1 1 0	size	1	Rm	0 0 1 1	1	1	Rn	Rd
	U						eq			

Scalar variant

```
CMGE <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

Vector

31	30	29	28 27 26 25 24	23 22	21	20 16	15 14 13 12	11	10	9 5	4 0
0	Q	0	0 1 1 1 0	size	1	Rm	0 0 1 1	1	1	Rn	Rd
		U						eq			

Vector variant

```
CMGE <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

D when size = 11

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following encodings are reserved:

- `size = 0x`.
- `size = 10`.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B	when `size = 00`, `Q = 0`
16B	when `size = 00`, `Q = 1`
4H	when `size = 01`, `Q = 0`
8H	when `size = 01`, `Q = 1`
2S	when `size = 10`, `Q = 0`
4S	when `size = 10`, `Q = 1`
2D	when `size = 11`, `Q = 1`

The encoding `size = 11`, `Q = 0` is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.30 CMGE (zero)

Compare signed Greater than or Equal to zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is greater than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	size		1	0	0	0	0	0	1	0	0	0	1	0	Rn		Rd	
		U																	op						

Scalar variant

```
CMGE <V><d>, <V><n>, #0
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	size		1	0	0	0	0	0	1	0	0	0	1	0	Rn		Rd	
		U																	op						

Vector variant

```
CMGE <Vd>.<T>, <Vn>.<T>, #0
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

D when size = 11

The following encodings are reserved:

- size = 0x.
- size = 10.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean test_passed;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.31 CMGT (register)

Compare signed Greater than (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first signed integer value is greater than the second signed integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	size		1	Rm		0	0	1	1	0	1	Rn		Rd	
		U															eq					

Scalar variant

```
CMGT <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	size		1	Rm		0	0	1	1	0	1	Rn		Rd	
		U															eq					

Vector variant

```
CMGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

D when size = 11

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following encodings are reserved:

- `size = 0x`.
- `size = 10`.

`<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<m>` Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B`	when `size = 00`, `Q = 0`
`16B`	when `size = 00`, `Q = 1`
`4H`	when `size = 01`, `Q = 0`
`8H`	when `size = 01`, `Q = 1`
`2S`	when `size = 10`, `Q = 0`
`4S`	when `size = 10`, `Q = 1`
`2D`	when `size = 11`, `Q = 1`

The encoding `size = 11`, `Q = 0` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.32 CMGT (zero)

Compare signed Greater than zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is greater than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31 30	29	28 27 26 25 24	23 22	21 20 19 18 17 16	15 14 13 12	11 10	9 5	4 0
0 1	0	1 1 1 1 0	size	1 0 0 0 0	0 1 0 0	0 1 0	Rn	Rd
	U				op			

Scalar variant

```
CMGT <V><d>, <V><n>, #0
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector

31 30	29	28 27 26 25 24	23 22	21 20 19 18 17 16	15 14 13 12	11 10	9 5	4 0
0 Q	0	0 1 1 1 0	size	1 0 0 0 0	0 1 0 0	0 1 0	Rn	Rd
	U				op			

Vector variant

```
CMGT <Vd>.<T>, <Vn>.<T>, #0
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

D when size = 11

The following encodings are reserved:

- size = 0x.
- size = 10.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean test_passed;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.33 CMHI (register)

Compare unsigned Higher (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first unsigned integer value is greater than the second unsigned integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	1	1	1	1	1	0	size		1	Rm	0	0	1	1	0	1	Rn	Rd
		U														eq			

Scalar variant

```
CMHI <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size		1	Rm	0	0	1	1	0	1	Rn	Rd
		U														eq			

Vector variant

```
CMHI <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

D when size = 11

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following encodings are reserved:

- `size = 0x`.
- `size = 10`.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B	when `size = 00`, `Q = 0`
16B	when `size = 00`, `Q = 1`
4H	when `size = 01`, `Q = 0`
8H	when `size = 01`, `Q = 1`
2S	when `size = 10`, `Q = 0`
4S	when `size = 10`, `Q = 1`
2D	when `size = 11`, `Q = 1`

The encoding `size = 11`, `Q = 0` is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.34 CMHS (register)

Compare unsigned Higher or Same (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first unsigned integer value is greater than or equal to the second unsigned integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	size		1	Rm		0	0	1	1	1	1	Rn		Rd	
		U															eq					

Scalar variant

```
CMHS <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	size		1	Rm		0	0	1	1	1	1	Rn		Rd	
		U															eq					

Vector variant

```
CMHS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

D when size = 11

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following encodings are reserved:

- `size = 0x`.
- `size = 10`.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B	when `size = 00`, `Q = 0`
16B	when `size = 00`, `Q = 1`
4H	when `size = 01`, `Q = 0`
8H	when `size = 01`, `Q = 1`
2S	when `size = 10`, `Q = 0`
4S	when `size = 10`, `Q = 1`
2D	when `size = 11`, `Q = 1`

The encoding `size = 11`, `Q = 0` is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.35 CMLE (zero)

Compare signed Less than or Equal to zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is less than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23 22	21	20	19	18	17	16	15	14	13	12	11	10	9 5	4 0
0	1	1	1	1	1	1	0	size	1	0	0	0	0	0	1	0	0	1	1	0	Rn	Rd
		U																op				

Scalar variant

```
CMLE <V><d>, <V><n>, #0
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector

31	30	29	28	27	26	25	24	23 22	21	20	19	18	17	16	15	14	13	12	11	10	9 5	4 0
0	Q	1	0	1	1	1	0	size	1	0	0	0	0	0	1	0	0	1	1	0	Rn	Rd
		U																op				

Vector variant

```
CMLE <Vd>.<T>, <Vn>.<T>, #0
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

D when size = 11

The following encodings are reserved:

- size = 0x.
- size = 10.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean test_passed;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.36 CMLT (zero)

Compare signed Less than zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is less than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	size		1	0	0	0	0	0	1	0	1	0	1	0	Rn		Rd	

Scalar variant

```
CMLT <V><d>, <V><n>, #0
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison = CompareOp_LT;
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	size		1	0	0	0	0	0	1	0	1	0	1	0	Rn		Rd	

Vector variant

```
CMLT <Vd>.<T>, <Vn>.<T>, #0
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison = CompareOp_LT;
```

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

D when size = 11

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following encodings are reserved:

- `size = 0x`.
- `size = 10`.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B	when `size = 00`, `Q = 0`
16B	when `size = 00`, `Q = 1`
4H	when `size = 01`, `Q = 0`
8H	when `size = 01`, `Q = 1`
2S	when `size = 10`, `Q = 0`
4S	when `size = 10`, `Q = 1`
2D	when `size = 11`, `Q = 1`

The encoding `size = 11`, `Q = 0` is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean test_passed;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.37 CMTST

Compare bitwise Test bits nonzero (vector). This instruction reads each vector element in the first source SIMD&FP register, performs an AND with the corresponding vector element in the second source SIMD&FP register, and if the result is not zero, sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	0	1	1	1	1	0	size		1	Rm	1	0	0	0	1	1	Rn	Rd
		U																	

Scalar variant

```
CMTST <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean and_test = (U == '0');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	size		1	Rm	1	0	0	0	1	1	Rn	Rd
		U																	

Vector variant

```
CMTST <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean and_test = (U == '0');
```

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

D when `size = 11`

The following encodings are reserved:

- `size = 0x`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- `size = 10`.

`<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<m>` Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B`	when `size = 00`, `Q = 0`
`16B`	when `size = 00`, `Q = 1`
`4H`	when `size = 01`, `Q = 0`
`8H`	when `size = 01`, `Q = 1`
`2S`	when `size = 10`, `Q = 0`
`4S`	when `size = 10`, `Q = 1`
`2D`	when `size = 11`, `Q = 1`

The encoding `size = 11`, `Q = 0` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if and_test then
        test_passed = !IsZero(element1 AND element2);
    else
        test_passed = (element1 == element2);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.38 CNT

Population Count per byte. This instruction counts the number of bits that have a value of one in each vector element in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	17	16	15	14	13	12	11	10	9 5	4 0
0	Q	0	0	1	1	1	0	size	1	0	0	0	0	0	0	1	0	1	1	0	Rn	Rd

Vector variant

```
CNT <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '00' then UNDEFINED;
integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B` when `size = 00`, `Q = 0`

`16B` when `size = 00`, `Q = 1`

The following encodings are reserved:

- `size = 01`, `Q = x`.
- `size = 1x`, `Q = x`.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

integer count;
for e = 0 to elements-1
    count = BitCount(Elem[operand, e, esize]);
    Elem[result, e, esize] = count<esize-1:0>;
V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.39 DUP (element)

Duplicate vector element to vector or scalar. This instruction duplicates the vector element at the specified element index in the source SIMD&FP register into a scalar or each element in a vector, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (scalar). The alias is always the preferred disassembly.

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	0	0	0	imm5		0	0	0	0	0	1	Rn		Rd	

Scalar variant

```
DUP <V><d>, <Vn>.<T>[<index>]
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);
if size > 3 then UNDEFINED;

integer index = UInt(imm5<4:size+1>);
integer idxdsize = if imm5<4> == '1' then 128 else 64;

integer esize = 8 << size;
integer datasize = esize;
integer elements = 1;
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	0	0	imm5		0	0	0	0	0	1	Rn		Rd	

Vector variant

```
DUP <Vd>.<T>, <Vn>.<Ts>[<index>]
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);
if size > 3 then UNDEFINED;

integer index = UInt(imm5<4:size+1>);
integer idxdsize = if imm5<4> == '1' then 128 else 64;

if size == 3 && Q == '0' then UNDEFINED;
integer esize = 8 << size;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<T>` For the scalar variant: is the element width specifier, encoded in the "imm5" field. It can have the following values:

`B` when `imm5 = xxxx1`

`H` when `imm5 = xxx10`

`S` when `imm5 = xx100`

`D` when `imm5 = x1000`

The encoding `imm5 = x0000` is reserved.

For the vector variant: is an arrangement specifier, encoded in the "imm5:Q" field. It can have the following values:

`8B` when `imm5 = xxxx1`, `Q = 0`

`16B` when `imm5 = xxxx1`, `Q = 1`

`4H` when `imm5 = xxx10`, `Q = 0`

`8H` when `imm5 = xxx10`, `Q = 1`

`2S` when `imm5 = xx100`, `Q = 0`

`4S` when `imm5 = xx100`, `Q = 1`

`2D` when `imm5 = x1000`, `Q = 1`

The following encodings are reserved:

- `imm5 = x0000`, `Q = x`.
- `imm5 = x1000`, `Q = 0`.

`<Ts>` Is an element size specifier, encoded in the "imm5" field. It can have the following values:

`B` when `imm5 = xxxx1`

`H` when `imm5 = xxx10`

`S` when `imm5 = xx100`

`D` when `imm5 = x1000`

The encoding `imm5 = x0000` is reserved.

`<V>` Is the destination width specifier, encoded in the "imm5" field. It can have the following values:

`B` when `imm5 = xxxx1`

`H` when `imm5 = xxx10`

`S` when `imm5 = xx100`

`D` when `imm5 = x1000`

The encoding `imm5 = x0000` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<index>` Is the element index encoded in the "imm5" field. It can have the following values:

`imm5<4:1>` when `imm5 = xxxx1`

`imm5<4:2>` when `imm5 = xxx10`

`imm5<4:3>` when `imm5 = xx100`

`imm5<4>` when `imm5 = x1000`

The encoding `imm5 = x0000` is reserved.

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(idxdsize) operand = V[n];
bits(datasize) result;
bits(esize) element;

element = Elem[operand, index, esize];
for e = 0 to elements-1
    Elem[result, e, esize] = element;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.40 DUP (general)

Duplicate general-purpose register to vector. This instruction duplicates the contents of the source general-purpose register into a scalar or each element in a vector, and writes the result to the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28 27 26 25	24 23 22 21	20 16	15	14 13 12 11	10	9 5	4 0
0	Q	0	0 1 1 1	0 0 0 0	imm5	0	0 0 0 1	1	Rn	Rd

Advanced SIMD variant

```
DUP <Vd>.<T>, <R><n>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);
if size > 3 then UNDEFINED;

// imm5<4:size+1> is IGNORED

if size == 3 && Q == '0' then UNDEFINED;
integer esize = 8 << size;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "imm5:Q" field. It can have the following values:

8B when imm5 = xxxx1, Q = 0

16B when imm5 = xxxx1, Q = 1

4H when imm5 = xxx10, Q = 0

8H when imm5 = xxx10, Q = 1

2S when imm5 = xx100, Q = 0

4S when imm5 = xx100, Q = 1

2D when imm5 = x1000, Q = 1

The following encodings are reserved:

- imm5 = x0000, Q = x.
- imm5 = x1000, Q = 0.

<R> Is the width specifier for the general-purpose source register, encoded in the "imm5" field. It can have the following values:

W when imm5 = xxxx1

W when imm5 = xxx10

W when imm5 = xx100

X when imm5 = x1000

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The encoding `imm5 = x0000` is reserved.

Unspecified bits in "imm5" are ignored but should be set to zero by an assembler.

`<n>` Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(esize) element = X[n];
bits(datasize) result;

for e = 0 to elements-1
    Elem[result, e, esize] = element;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.41 EOR (vector)

Bitwise Exclusive OR (vector). This instruction performs a bitwise Exclusive OR operation between the two source SIMD&FP registers, and places the result in the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	0	0	1	Rm	0	0	0	1	1	1	Rn	Rd
								opc2											

Three registers of the same type variant

```
EOR <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `8B` when `Q = 0`
- `16B` when `Q = 1`

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand2;
bits(datasize) operand3;
bits(datasize) operand4 = V[n];

operand1 = V[m];
operand2 = Zeros();
operand3 = Ones();
V[d] = operand1 EOR ((operand2 EOR operand4) AND operand3);
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.42 EOR3

Three-way Exclusive OR performs a three-way exclusive OR of the values in the three source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

This instruction is implemented only when ARMv8.2-SHA is implemented.

ARMv8.2

31 30 29 28	27 26 25 24	23 22 21 20	16	15 14	10 9	5 4	0
1 1 0 0	1 1 1 0 0	0 0	Rm	0	Ra	Rn	Rd

Advanced SIMD variant

```
EOR3 <Vd>.16B, <Vn>.16B, <Vm>.16B, <Va>.16B
```

Decode for this encoding

```
if !HaveSHA3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer a = UInt(Ra);
```

Assembler symbols

<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn>	Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
<Va>	Is the name of the third SIMD&FP source register, encoded in the "Ra" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Va = V[a];
V[d] = Vn EOR Vm EOR Va;
```

Operational information

If PSTATE.DIT is 1:

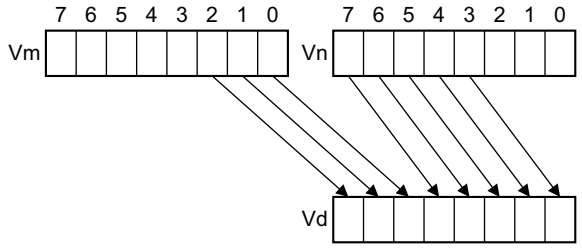
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.43 EXT

Extract vector from pair of vectors. This instruction extracts the lowest vector elements from the second source SIMD&FP register and the highest vector elements from the first source SIMD&FP register, concatenates the results into a vector, and writes the vector to the destination SIMD&FP register vector. The index value specifies the lowest vector element to extract from the first source register, and consecutive elements are extracted from the first, then second, source registers until the destination vector is filled.

The following figure shows the operation of EXT doubleword operation for Q = 0 and imm4<2:0> = 3.



Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20–16	15	14–11	10	9–5	4–0
0	Q	1	0	1	1	1	0	0	0	0	Rm	0	imm4	0	Rn	Rd

Advanced SIMD variant

```
EXT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<index>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if Q == '0' && imm4<3> == '1' then UNDEFINED;

integer datasize = if Q == '1' then 128 else 64;
integer position = UInt(imm4) << 3;
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`8B` when `Q = 0`

`16B` when `Q = 1`

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

`<index>` Is the lowest numbered byte element to be extracted, encoded in the "Q:imm4" field. It can have the following values:

`imm4<2:0>` when `Q = 0`, `imm4<3> = 0`

`imm4` when `Q = 1`, `imm4<3> = x`

The encoding `Q = 0`, `imm4<3> = 1` is reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) hi = V[m];
bits(datasize) lo = V[n];
bits(datasize*2) concat = hi:lo;

V[d] = concat<position+datasize-1:position>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.44 FABD

Floating-point Absolute Difference (vector). This instruction subtracts the floating-point values in the elements of the second source SIMD&FP register, from the corresponding floating-point values in the elements of the first source SIMD&FP register, places the absolute value of each result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	1	1	0	Rm		0	0	0	1	0	1	Rn		Rd	

Scalar half precision variant

```
FABD <Hd>, <Hn>, <Hm>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean abs = TRUE;
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	1	sz	1	Rm		1	1	0	1	0	1	Rn		Rd	

Scalar single-precision and double-precision variant

```
FABD <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean abs = TRUE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Vector half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	1	1	0	Rm		0	0	0	1	0	1	Rn		Rd	
		U																				

Vector half precision variant

```
FABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	1	sz	1	Rm		1	1	0	1	0	1	Rn		Rd	
		U																				

Vector single-precision and double-precision variant

```
FABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');
```

Assembler symbols

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

`<V>` Is a width specifier, encoded in the "sz" field. It can have the following values:

- `S` when `sz = 0`
- `D` when `sz = 1`

`<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<m>` Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the vector half precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`4H` when `Q = 0`

`8H` when `Q = 1`

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

`2S` when `sz = 0`, `Q = 0`

`4S` when `sz = 0`, `Q = 1`

`2D` when `sz = 1`, `Q = 1`

The encoding `sz = 1`, `Q = 0` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) diff;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    diff = FPSub(element1, element2, FPCR);
    Elem[result, e, esize] = if abs then FPAbs(diff) else diff;

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.45 FABS (vector)

Floating-point Absolute value (vector). This instruction calculates the absolute value of each vector element in the source SIMD&FP register, writes the result to a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	1	1	1	1	1	0	0	0	1	1	1	1	1	0	Rn		Rd	
		U																							

Half-precision variant

```
FABS <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	1	1	1	0	Rn		Rd	
		U																							

Single-precision and double-precision variant

```
FABS <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`4H` when `Q = 0`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`8H` when `Q = 1`

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

`2S` when `sz = 0`, `Q = 0`

`4S` when `sz = 0`, `Q = 1`

`2D` when `sz = 1`, `Q = 1`

The encoding `sz = 1`, `Q = 0` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    if neg then
        element = FPNeg(element);
    else
        element = FPAbs(element);
    Elem[result, e, esize] = element;

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.46 FABS (scalar)

Floating-point Absolute value (scalar). This instruction calculates the absolute value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	17	16	15	14	13	12	11	10	9 5	4 0
0	0	0	1	1	1	1	0	ftype	1	0	0	0	0	0	1	1	0	0	0	0	Rn	Rd
														opc								

Half-precision variant

Applies when `ftype == 11`.

```
FABS <Hd>, <Hn>
```

Single-precision variant

Applies when `ftype == 00`.

```
FABS <Sd>, <Sn>
```

Double-precision variant

Applies when `ftype == 01`.

```
FABS <Dd>, <Dn>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;
```

Assembler symbols

`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Dn>`	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
`<Hd>`	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Hn>`	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
`<Sd>`	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Sn>`	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPAbs(operand);
V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.47 FACGE

Floating-point Absolute Compare Greater than or Equal (vector). This instruction compares the absolute value of each floating-point value in the first source SIMD&FP register with the absolute value of the corresponding floating-point value in the second source SIMD&FP register and if the first value is greater than or equal to the second value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	0	1	0	Rm		0	0	1	0	1	1	Rn		Rd	
		U						E									ac					

Scalar half precision variant

```
FACGE <Hd>, <Hn>, <Hm>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	0	sz	1	Rm		1	1	1	0	1	1	Rn		Rd	
		U						E									ac					

Scalar single-precision and double-precision variant

```
FACGE <V><d>, <V><n>, <V><m>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

Vector half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	0	1	0	Rm		0	0	1	0	1	1	Rn		Rd	
		U						E									ac					

Vector half precision variant

```
FACGE <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	0	sz	1	Rm		1	1	1	0	1	1	Rn		Rd	
		U						E									ac					

Vector single-precision and double-precision variant

```
FACGE <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

Assembler symbols

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

`<V>` Is a width specifier, encoded in the "sz" field. It can have the following values:

- `S` when `sz = 0`
- `D` when `sz = 1`

`<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<m>` Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the vector half precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `4H` when `Q = 0`
- `8H` when `Q = 1`

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- `2S` when `sz = 0`, `Q = 0`
- `4S` when `sz = 0`, `Q = 1`
- `2D` when `sz = 1`, `Q = 1`

The encoding `sz = 1`, `Q = 0` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(esize) element1;
bits(esize) element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, FPCR);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.48 FACGT

Floating-point Absolute Compare Greater than (vector). This instruction compares the absolute value of each vector element in the first source SIMD&FP register with the absolute value of the corresponding vector element in the second source SIMD&FP register and if the first value is greater than the second value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	1	1	0	Rm		0	0	1	0	1	1	Rn		Rd	
		U						E									ac					

Scalar half precision variant

```
FACGT <Hd>, <Hn>, <Hm>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	1	sz	1	Rm		1	1	1	0	1	1	Rn		Rd	
		U						E									ac					

Scalar single-precision and double-precision variant

```
FACGT <V><d>, <V><n>, <V><m>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

Vector half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	1	1	0	Rm		0	0	1	0	1	1	Rn		Rd	
		U						E									ac					

Vector half precision variant

```
FACGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	1	sz	1	Rm		1	1	1	0	1	1	Rn		Rd	
		U						E									ac					

Vector single-precision and double-precision variant

```
FACGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

Assembler symbols

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

`<V>` Is a width specifier, encoded in the "sz" field. It can have the following values:

- `S` when `sz = 0`
- `D` when `sz = 1`

`<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<m>` Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the vector half precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `4H` when `Q = 0`
- `8H` when `Q = 1`

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- `2S` when `sz = 0`, `Q = 0`
- `4S` when `sz = 0`, `Q = 1`
- `2D` when `sz = 1`, `Q = 1`

The encoding `sz = 1`, `Q = 0` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(esize) element1;
bits(esize) element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, FPCR);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.49 FADD (vector)

Floating-point Add (vector). This instruction adds corresponding vector elements in the two source SIMD&FP registers, writes the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	1	0	Rm		0	0	0	1	0	1	Rn		Rd	
		U																				

Half-precision variant

```
FADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	sz	1	Rm		1	1	0	1	0	1	Rn		Rd	
		U																				

Single-precision and double-precision variant

```
FADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`4H` when `Q = 0`

`8H` when `Q = 1`

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

`2S` when `sz = 0`, `Q = 0`

`4S` when `sz = 0`, `Q = 1`

`2D` when `sz = 1`, `Q = 1`

The encoding `sz = 1`, `Q = 0` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPAdd(element1, element2, FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.50 FADD (scalar)

Floating-point Add (scalar). This instruction adds the floating-point values of the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	0	0	1	1	1	1	0	ftype	1	Rm	0	0	1	0	1	0	Rn	Rd
														op				

Half-precision variant

Applies when `ftype == 11`.

```
FADD <Hd>, <Hn>, <Hm>
```

Single-precision variant

Applies when `ftype == 00`.

```
FADD <Sd>, <Sn>, <Sm>
```

Double-precision variant

Applies when `ftype == 01`.

```
FADD <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;
```

Assembler symbols

`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Dn>`	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
`<Dm>`	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
`<Hd>`	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Hn>`	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Sn>` Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result = FPAdd(operand1, operand2, FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.51 FADDP (scalar)

Floating-point Add Pair of elements (scalar). This instruction adds two floating-point vector elements in the source SIMD&FP register and writes the scalar result into the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31 30	29	28 27 26 25	24 23	22	21 20 19 18 17	16 15 14 13	12 11 10	9 … 5	4 … 0
0 1	0	1 1 1 1	0 0	0	1 1 0 0 0	0 1 1 0	1 1 0	Rn	Rd
				sz					

Half-precision variant

```
FADDP <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 16;
integer datasize = 32;
```

Single-precision and double-precision

31 30	29	28 27 26 25	24 23	22	21 20 19 18 17	16 15 14 13	12 11 10	9 … 5	4 … 0
0 1	1	1 1 1 1	0 0	sz	1 1 0 0 0	0 1 1 0	1 1 0	Rn	Rd

Single-precision and double-precision variant

```
FADDP <V><d>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32;
integer datasize = 64;
```

Assembler symbols

<V> For the half-precision variant: is the destination width specifier, encoded in the "sz" field. It can have the following values:

H when sz = 0

The encoding sz = 1 is reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the single-precision and double-precision variant: is the destination width specifier, encoded in the "sz" field. It can have the following values:

`S` when `sz = 0`

`D` when `sz = 1`

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<T>` For the half-precision variant: is the source arrangement specifier, encoded in the "sz" field. It can have the following values:

`2H` when `sz = 0`

The encoding `sz = 1` is reserved.

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in the "sz" field. It can have the following values:

`2S` when `sz = 0`

`2D` when `sz = 1`

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FADD, operand, esize);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.52 FADDP (vector)

Floating-point Add Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, adds each pair of values together, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	0	1	0	Rm		0	0	0	1	0	1	Rn		Rd	
		U																				

Half-precision variant

```
FADDP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	0	sz	1	Rm		1	1	0	1	0	1	Rn		Rd	
		U																				

Single-precision and double-precision variant

```
FADDP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`4H` when `Q = 0`

`8H` when `Q = 1`

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

`2S` when `sz = 0`, `Q = 0`

`4S` when `sz = 0`, `Q = 1`

`2D` when `sz = 1`, `Q = 1`

The encoding `sz = 1`, `Q = 0` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPAdd(element1, element2, FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.53 FCADD

Floating-point Complex Add.

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following computation on the corresponding complex number element pairs from the two source registers:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 90 or 270 degrees.
- The rotated complex number is added to the complex number from the first source register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

ARMv8.3

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size	0	Rm	1	1	1	rot	0	1	Rn	Rd

Vector variant

```
FCADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<rotate>
```

Decode for this encoding

```
if !HaveFCADDExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '00' then UNDEFINED;
if Q == '0' && size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

`<rotate>` Is the rotation, encoded in the "rot" field. It can have the following values:

`90` when `rot = 0`

`270` when `rot = 1`

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element3;

for e = 0 to (elements DIV 2)-1
    case rot of
        when '0'
            element1 = FPNeg(Elem[operand2, e*2+1, esize]);
            element3 = Elem[operand2, e*2, esize];
        when '1'
            element1 = Elem[operand2, e*2+1, esize];
            element3 = FPNeg(Elem[operand2, e*2, esize]);
    Elem[result, e*2, esize] = FPAdd(Elem[operand1, e*2, esize], element1, FPCR);
    Elem[result, e*2+1, esize] = FPAdd(Elem[operand1, e*2+1, esize], element3, FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.54 FCCMP

Floating-point Conditional quiet Compare (scalar). This instruction compares the two SIMD&FP source register values and writes the result to the PSTATE.{N, Z, C, V} flags. If the condition does not pass then the PSTATE.{N, Z, C, V} flags are set to the flag bit specifier.

It raises an Invalid Operation exception only if either operand is a signaling NaN.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 16	15 12	11	10	9 5	4	3 0
0	0	0	1	1	1	1	0	ftype	1	Rm	cond	0	1	Rn	0	nzcv
															op	

Half-precision variant

Applies when `ftype == 11`.

```
FCCMP <Hn>, <Hm>, #<nzcv>, <cond>
```

Single-precision variant

Applies when `ftype == 00`.

```
FCCMP <Sn>, <Sm>, #<nzcv>, <cond>
```

Double-precision variant

Applies when `ftype == 01`.

```
FCCMP <Dn>, <Dm>, #<nzcv>, <cond>
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;

bits(4) flags = nzcv;
```

Assembler symbols

`<Dn>`	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
`<Dm>`	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
`<Hn>`	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

`<Sn>` Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

`<nzcv>` Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.

`<cond>` Is one of the standard conditions, encoded in the "cond" field in the standard way.

NaNs

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands are NaNs, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. This case results in the FPSCR flags being set to N=0, Z=0, C=1, and V=1.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;

operand2 = V[m];

if ConditionHolds(cond) then
    flags = FPCompare(operand1, operand2, FALSE, FPCR);
PSTATE.<N,Z,C,V> = flags;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.55 FCCMPE

Floating-point Conditional signaling Compare (scalar). This instruction compares the two SIMD&FP source register values and writes the result to the PSTATE.{N, Z, C, V} flags. If the condition does not pass then the PSTATE.{N, Z, C, V} flags are set to the flag bit specifier.

If either operand is any type of NaN, or if either operand is a signaling NaN, the instruction raises an Invalid Operation exception.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15 … 12	11	10	9 … 5	4	3 … 0
0	0	0	1	1	1	1	0	ftype	1	Rm	cond	0	1	Rn	1	nzcv
															op	

Half-precision variant

Applies when `ftype == 11`.

```
FCCMPE <Hn>, <Hm>, #<nzcv>, <cond>
```

Single-precision variant

Applies when `ftype == 00`.

```
FCCMPE <Sn>, <Sm>, #<nzcv>, <cond>
```

Double-precision variant

Applies when `ftype == 01`.

```
FCCMPE <Dn>, <Dm>, #<nzcv>, <cond>
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;

bits(4) flags = nzcv;
```

Assembler symbols

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Hn>` Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

`<Sn>` Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

`<nzcv>` Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.

`<cond>` Is one of the standard conditions, encoded in the "cond" field in the standard way.

NaNs

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands are NaNs, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. This case results in the FPSCR flags being set to N=0, Z=0, C=1, and V=1.

`FCCMPE` raises an Invalid Operation exception if either operand is any type of NaN, and is suitable for testing for <, <=, >, >=, and other predicates that raise an exception when the operands are unordered.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;

operand2 = V[m];

if ConditionHolds(cond) then
    flags = FPCompare(operand1, operand2, TRUE, FPCR);
PSTATE.<N,Z,C,V> = flags;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.56 FCMEQ (register)

Floating-point Compare Equal (vector). This instruction compares each floating-point value from the first source SIMD&FP register, with the corresponding floating-point value from the second source SIMD&FP register, and if the comparison is equal sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

ARMv8.2

31 30 29 28	27 26 25 24	23 22 21 20	16	15 14 13 12	11 10 9	5 4	0
0 1 0	1 1 1 1 0	0 1 0	Rm	0 0 1 0	0 1	Rn	Rd
U		E			ac		

Scalar half precision variant

```
FCMEQ <Hd>, <Hn>, <Hm>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

Scalar single-precision and double-precision

31 30 29 28	27 26 25 24	23 22 21 20	16	15 14 13 12	11 10 9	5 4	0
0 1 0	1 1 1 1 0	0 sz 1	Rm	1 1 1 0	0 1	Rn	Rd
U		E			ac		

Scalar single-precision and double-precision variant

```
FCMEQ <V><d>, <V><n>, <V><m>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

Vector half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	1	0	Rm		0	0	1	0	0	1	Rn		Rd	
		U						E									ac					

Vector half precision variant

```
FCMEQ <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	sz	1	Rm		1	1	1	0	0	1	Rn		Rd	
		U						E									ac					

Vector single-precision and double-precision variant

```
FCMEQ <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

Assembler symbols

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

`<V>` Is a width specifier, encoded in the "sz" field. It can have the following values:

- `S` when `sz = 0`
- `D` when `sz = 1`

`<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<m>` Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the vector half precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `4H` when `Q = 0`
- `8H` when `Q = 1`

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- `2S` when `sz = 0`, `Q = 0`
- `4S` when `sz = 0`, `Q = 1`
- `2D` when `sz = 1`, `Q = 1`

The encoding `sz = 1`, `Q = 0` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(esize) element1;
bits(esize) element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, FPCR);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.57 FCMEQ (zero)

Floating-point Compare Equal to zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	0	1	1	0	1	1	0	Rn		Rd	
		U																	op						

Scalar half precision variant

```
FCMEQ <Hd>, <Hn>, #0.0
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	0	1	1	0	Rn		Rd	
		U																	op						

Scalar single-precision and double-precision variant

```
FCMEQ <V><d>, <V><n>, #0.0
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	1	1	1	1	1	0	0	0	1	1	0	1	1	0	Rn		Rd	
		U																	op						

Vector half precision variant

```
FCMEQ <Vd>.<T>, <Vn>.<T>, #0.0
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	0	1	1	0	Rn		Rd	
		U																	op						

Vector single-precision and double-precision variant

```
FCMEQ <Vd>.<T>, <Vn>.<T>, #0.0
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Assembler symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0

D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

4H when Q = 0

8H when Q = 1

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0

4S when sz = 0, Q = 1

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.58 FCMGE (register)

Floating-point Compare Greater than or Equal (vector). This instruction reads each floating-point value in the first source SIMD&FP register and if the value is greater than or equal to the corresponding floating-point value in the second source SIMD&FP register sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	0	1	0	Rm		0	0	1	0	0	1	Rn		Rd	
		U						E									ac					

Scalar half precision variant

```
FCMGE <Hd>, <Hn>, <Hm>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	0	sz	1	Rm		1	1	1	0	0	1	Rn		Rd	
		U						E									ac					

Scalar single-precision and double-precision variant

```
FCMGE <V><d>, <V><n>, <V><m>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

Vector half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	0	1	0	Rm		0	0	1	0	0	1	Rn		Rd	
		U						E									ac					

Vector half precision variant

```
FCMGE <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	0	sz	1	Rm		1	1	1	0	0	1	Rn		Rd	
		U						E									ac					

Vector single-precision and double-precision variant

```
FCMGE <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

Assembler symbols

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

`<V>` Is a width specifier, encoded in the "sz" field. It can have the following values:

- `S` when `sz = 0`
- `D` when `sz = 1`

`<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<m>` Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the vector half precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `4H` when `Q = 0`
- `8H` when `Q = 1`

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- `2S` when `sz = 0`, `Q = 0`
- `4S` when `sz = 0`, `Q = 1`
- `2D` when `sz = 1`, `Q = 1`

The encoding `sz = 1`, `Q = 0` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(esize) element1;
bits(esize) element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, FPCR);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.59 FCMGE (zero)

Floating-point Compare Greater than or Equal to zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is greater than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	1	1	1	1	1	0	0	0	1	1	0	0	1	0	Rn		Rd	
		U																	op						

Scalar half precision variant

```
FCMGE <Hd>, <Hn>, #0.0
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	0	0	1	0	Rn		Rd	
		U																	op						

Scalar single-precision and double-precision variant

```
FCMGE <V><d>, <V><n>, #0.0
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	1	1	1	1	1	0	0	0	1	1	0	0	1	0	Rn		Rd	
		U																	op						

Vector half precision variant

```
FCMGE <Vd>.<T>, <Vn>.<T>, #0.0
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	0	0	1	0	Rn		Rd	
		U																	op						

Vector single-precision and double-precision variant

```
FCMGE <Vd>.<T>, <Vn>.<T>, #0.0
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Assembler symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0

D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

4H when Q = 0

8H when Q = 1

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0

4S when sz = 0, Q = 1

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.60 FCMGT (register)

Floating-point Compare Greater than (vector). This instruction reads each floating-point value in the first source SIMD&FP register and if the value is greater than the corresponding floating-point value in the second source SIMD&FP register sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	1	1	0	Rm		0	0	1	0	0	1	Rn		Rd	
		U						E									ac					

Scalar half precision variant

```
FCMGT <Hd>, <Hn>, <Hm>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	1	sz	1	Rm		1	1	1	0	0	1	Rn		Rd	
		U						E									ac					

Scalar single-precision and double-precision variant

```
FCMGT <V><d>, <V><n>, <V><m>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

Vector half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	1	1	0	Rm		0	0	1	0	0	1	Rn		Rd	
		U						E									ac					

Vector half precision variant

```
FCMGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	1	sz	1	Rm		1	1	1	0	0	1	Rn		Rd	
		U						E									ac					

Vector single-precision and double-precision variant

```
FCMGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

Assembler symbols

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

`<V>` Is a width specifier, encoded in the "sz" field. It can have the following values:

- `S` when `sz = 0`
- `D` when `sz = 1`

`<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<m>` Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the vector half precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `4H` when `Q = 0`
- `8H` when `Q = 1`

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- `2S` when `sz = 0`, `Q = 0`
- `4S` when `sz = 0`, `Q = 1`
- `2D` when `sz = 1`, `Q = 1`

The encoding `sz = 1`, `Q = 0` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(esize) element1;
bits(esize) element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, FPCR);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.61 FCMGT (zero)

Floating-point Compare Greater than zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is greater than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	0	1	1	0	0	1	0	Rn		Rd	
		U																	op						

Scalar half precision variant

```
FCMGT <Hd>, <Hn>, #0.0
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	0	0	1	0	Rn		Rd	
		U																	op						

Scalar single-precision and double-precision variant

```
FCMGT <V><d>, <V><n>, #0.0
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	1	1	1	1	1	0	0	0	1	1	0	0	1	0	Rn		Rd	
		U																	op						

Vector half precision variant

```
FCMGT <Vd>.<T>, <Vn>.<T>, #0.0
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	0	0	1	0	Rn		Rd	
		U																	op						

Vector single-precision and double-precision variant

```
FCMGT <Vd>.<T>, <Vn>.<T>, #0.0
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Assembler symbols

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

`<V>` Is a width specifier, encoded in the "sz" field. It can have the following values:

`S` when `sz = 0`

`D` when `sz = 1`

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

`<n>` Is the number of the SIMD&FP source register, encoded in the "Rn" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the vector half precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`4H` when `Q = 0`

`8H` when `Q = 1`

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

`2S` when `sz = 0`, `Q = 0`

`4S` when `sz = 0`, `Q = 1`

`2D` when `sz = 1`, `Q = 1`

The encoding `sz = 1`, `Q = 0` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.62 FCMLA (by element)

Floating-point Complex Multiply Accumulate (by element).

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following computation on complex numbers from the first source register and the destination register with the specified complex number from the second source register:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees.
- The two elements of the transformed complex number are multiplied by:
  - The real element of the complex number from the first source register, if the transformation was a rotation by 0 or 180 degrees.
  - The imaginary element of the complex number from the first source register, if the transformation was a rotation by 90 or 270 degrees.
- The complex number resulting from that multiplication is added to the complex number from the destination register.

The multiplication and addition operations are performed as a fused multiply-add, without any intermediate rounding.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

ARMv8.3

31	30	29	28	27	26	25	24	23 22	21	20	19 16	15	14 13	12	11	10	9 5	4 0
0	Q	1	0	1	1	1	1	size	L	M	Rm	0	rot	1	H	0	Rn	Rd

Encoding

Applies when `size == 01`.

```
FCMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>], #<rotate>
```

Encoding

Applies when `size == 10`.

```
FCMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>], #<rotate>
```

Decode for all variants of this encoding

```
if !HaveFCADDExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(M:Rm);
if size == '00' || size == '11' then UNDEFINED;
if size == '01' then index = UInt(H:L);
if size == '10' then index = UInt(H);
integer esize = 8 << UInt(size);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
if size == '10' && (L == '1' || Q == '0') then UNDEFINED;
if size == '01' && H == '1' && Q == '0' then UNDEFINED;
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`4H` when `size = 01`, `Q = 0`

`8H` when `size = 01`, `Q = 1`

`4S` when `size = 10`, `Q = 1`

The following encodings are reserved:

- `size = 00`, `Q = x`.
- `size = 10`, `Q = 0`.
- `size = 11`, `Q = x`.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

`<Ts>` Is an element size specifier, encoded in the "size" field. It can have the following values:

`H` when `size = 01`

`S` when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

`<index>` Is the element index, encoded in the "size:H:L" field. It can have the following values:

`H:L` when `size = 01`

`H` when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

`<rotate>` Is the rotation, encoded in the "rot" field. It can have the following values:

`0` when `rot = 00`

`90` when `rot = 01`

`180` when `rot = 10`

`270` when `rot = 11`

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;

for e = 0 to (elements DIV 2)-1
    case rot of
        when '00'
            element1 = Elem[operand2, index*2, esize];
            element2 = Elem[operand1, e*2, esize];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            element3 = Elem[operand2, index*2+1, esize];
            element4 = Elem[operand1, e*2, esize];
        when '01'
            element1 = FPNeg(Elem[operand2, index*2+1, esize]);
            element2 = Elem[operand1, e*2+1, esize];
            element3 = Elem[operand2, index*2, esize];
            element4 = Elem[operand1, e*2+1, esize];
        when '10'
            element1 = FPNeg(Elem[operand2, index*2, esize]);
            element2 = Elem[operand1, e*2, esize];
            element3 = FPNeg(Elem[operand2, index*2+1, esize]);
            element4 = Elem[operand1, e*2, esize];
        when '11'
            element1 = Elem[operand2, index*2+1, esize];
            element2 = Elem[operand1, e*2+1, esize];
            element3 = FPNeg(Elem[operand2, index*2, esize]);
            element4 = Elem[operand1, e*2+1, esize];

    Elem[result, e*2, esize] = FPMulAdd(Elem[operand3, e*2, esize], element2, element1, FPCR);
    Elem[result, e*2+1, esize] = FPMulAdd(Elem[operand3, e*2+1, esize], element4, element3, FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.63 FCMLA

Floating-point Complex Multiply Accumulate.

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following computation on the corresponding complex number element pairs from the two source registers and the destination register:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees.
- The two elements of the transformed complex number are multiplied by:
  - The real element of the complex number from the first source register, if the transformation was a rotation by 0 or 180 degrees.
  - The imaginary element of the complex number from the first source register, if the transformation was a rotation by 90 or 270 degrees.
- The complex number resulting from that multiplication is added to the complex number from the destination register.

The multiplication and addition operations are performed as a fused multiply-add, without any intermediate rounding.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

ARMv8.3

31	30	29	28	27	26	25	24	23 22	21	20 16	15	14	13	12	11 10	9 5	4 0
0	Q	1	0	1	1	1	0	size	0	Rm	1	1	0	rot	1	Rn	Rd

Vector variant

```
FCMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<rotate>
```

Decode for this encoding

```
if !HaveFCADDExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '00' then UNDEFINED;
if Q == '0' && size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1
2D	when size = 11, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = 0.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<rotate> Is the rotation, encoded in the "rot" field. It can have the following values:

0	when rot = 00
90	when rot = 01
180	when rot = 10
270	when rot = 11

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) element3;
bits(esize) element4;

for e = 0 to (elements DIV 2)-1
    case rot of
        when '00'
            element1 = Elem[operand2, e*2, esize];
            element2 = Elem[operand1, e*2, esize];
            element3 = Elem[operand2, e*2+1, esize];
            element4 = Elem[operand1, e*2, esize];
        when '01'
            element1 = FPNeg(Elem[operand2, e*2+1, esize]);
            element2 = Elem[operand1, e*2+1, esize];
            element3 = Elem[operand2, e*2, esize];
            element4 = Elem[operand1, e*2+1, esize];
        when '10'
            element1 = FPNeg(Elem[operand2, e*2, esize]);
            element2 = Elem[operand1, e*2, esize];
            element3 = FPNeg(Elem[operand2, e*2+1, esize]);
            element4 = Elem[operand1, e*2, esize];
        when '11'
            element1 = Elem[operand2, e*2+1, esize];
            element2 = Elem[operand1, e*2+1, esize];
            element3 = FPNeg(Elem[operand2, e*2, esize]);
            element4 = Elem[operand1, e*2+1, esize];

    Elem[result, e*2, esize] = FPMulAdd(Elem[operand3, e*2, esize], element2, element1, FPCR);
    Elem[result, e*2+1, esize] = FPMulAdd(Elem[operand3, e*2+1, esize], element4, element3, FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.64 FCMLE (zero)

Floating-point Compare Less than or Equal to zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is less than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	1	1	1	1	1	0	0	0	1	1	0	1	1	0	Rn		Rd	
		U																	op						

Scalar half precision variant

```
FCMLE <Hd>, <Hn>, #0.0
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	0	1	1	0	Rn		Rd	
		U																	op						

Scalar single-precision and double-precision variant

```
FCMLE <V><d>, <V><n>, #0.0
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	1	1	1	1	1	0	0	0	1	1	0	1	1	0	Rn		Rd	
		U																	op						

Vector half precision variant

```
FCMLE <Vd>.<T>, <Vn>.<T>, #0.0
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	0	1	1	0	Rn		Rd	
		U																	op						

Vector single-precision and double-precision variant

```
FCMLE <Vd>.<T>, <Vn>.<T>, #0.0
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Assembler symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0

D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

4H when Q = 0

8H when Q = 1

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0

4S when sz = 0, Q = 1

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.65 FCMLT (zero)

Floating-point Compare Less than zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is less than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	0	1	1	1	0	1	0	Rn		Rd	

Scalar half precision variant

```
FCMLT <Hd>, <Hn>, #0.0
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison = CompareOp_LT;
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	1	0	1	0	Rn		Rd	

Scalar single-precision and double-precision variant

```
FCMLT <V><d>, <V><n>, #0.0
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison = CompareOp_LT;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Vector half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	1	1	1	1	1	0	0	0	1	1	1	0	1	0	Rn		Rd	

Vector half precision variant

```
FCMLT <Vd>.<T>, <Vn>.<T>, #0.0
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison = CompareOp_LT;
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	1	0	1	0	Rn		Rd	

Vector single-precision and double-precision variant

```
FCMLT <Vd>.<T>, <Vn>.<T>, #0.0
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison = CompareOp_LT;
```

Assembler symbols

`<Hd>`	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Hn>`	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

`<V>`	Is a width specifier, encoded in the "sz" field. It can have the following values:
	`S` when `sz = 0`
	`D` when `sz = 1`

`<d>`	Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
`<n>`	Is the number of the SIMD&FP source register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the vector half precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`4H` when `Q = 0`

`8H` when `Q = 1`

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

`2S` when `sz = 0`, `Q = 0`

`4S` when `sz = 0`, `Q = 1`

`2D` when `sz = 1`, `Q = 1`

The encoding `sz = 1`, `Q = 0` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.66 FCMP

Floating-point quiet Compare (scalar). This instruction compares the two SIMD&FP source register values, or the first SIMD&FP source register value and zero. It writes the result to the PSTATE.{N, Z, C, V} flags.

It raises an Invalid Operation exception only if either operand is a signaling NaN.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 16	15	14	13	12	11	10	9 5	4	3	2	1	0
0	0	0	1	1	1	1	0	ftype	1	Rm	0	0	1	0	0	0	Rn	0	x	0	0	0
																		opc				

Half-precision variant

Applies when `ftype == 11 && opc == 00`.

```
FCMP <Hn>, <Hm>
```

Half-precision, zero variant

Applies when `ftype == 11 && Rm == (00000) && opc == 01`.

```
FCMP <Hn>, #0.0
```

Single-precision variant

Applies when `ftype == 00 && opc == 00`.

```
FCMP <Sn>, <Sm>
```

Single-precision, zero variant

Applies when `ftype == 00 && Rm == (00000) && opc == 01`.

```
FCMP <Sn>, #0.0
```

Double-precision variant

Applies when `ftype == 01 && opc == 00`.

```
FCMP <Dn>, <Dm>
```

Double-precision, zero variant

Applies when `ftype == 01 && Rm == (00000) && opc == 01`.

```
FCMP <Dn>, #0.0
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer m = UInt(Rm);    // ignored when opc<0> == '1'

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;

boolean signal_all_nans = (opc<1> == '1');
boolean cmp_with_zero = (opc<0> == '1');
```

Assembler symbols

`<Dn>` For the double-precision variant: is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

For the double-precision, zero variant: is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

`<Hn>` For the half-precision variant: is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

For the half-precision, zero variant: is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

`<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

`<Sn>` For the single-precision variant: is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

For the single-precision, zero variant: is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

`<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

NaNs

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands are NaNs, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. This case results in the FPSCR flags being set to N=0, Z=0, C=1, and V=1.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;

operand2 = if cmp_with_zero then FPZero('0') else V[m];

PSTATE.<N,Z,C,V> = FPCompare(operand1, operand2, signal_all_nans, FPCR);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.67 FCMPE

Floating-point signaling Compare (scalar). This instruction compares the two SIMD&FP source register values, or the first SIMD&FP source register value and zero. It writes the result to the PSTATE.{N, Z, C, V} flags.

If either operand is any type of NaN, or if either operand is a signaling NaN, the instruction raises an Invalid Operation exception.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4	3	2	1	0
0	0	0	1	1	1	1	0	ftype	1	Rm	0	0	1	0	0	0	Rn	1	x	0	0	0
																		opc				

Half-precision variant

Applies when `ftype == 11 && opc == 10`.

```
FCMPE <Hn>, <Hm>
```

Half-precision, zero variant

Applies when `ftype == 11 && Rm == (00000) && opc == 11`.

```
FCMPE <Hn>, #0.0
```

Single-precision variant

Applies when `ftype == 00 && opc == 10`.

```
FCMPE <Sn>, <Sm>
```

Single-precision, zero variant

Applies when `ftype == 00 && Rm == (00000) && opc == 11`.

```
FCMPE <Sn>, #0.0
```

Double-precision variant

Applies when `ftype == 01 && opc == 10`.

```
FCMPE <Dn>, <Dm>
```

Double-precision, zero variant

Applies when `ftype == 01 && Rm == (00000) && opc == 11`.

```
FCMPE <Dn>, #0.0
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer m = UInt(Rm);    // ignored when opc<0> == '1'

integer datasize;
case ftype of
    when '00' datasize = 32;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;

boolean signal_all_nans = (opc<1> == '1');
boolean cmp_with_zero = (opc<0> == '1');
```

Assembler symbols

`<Dn>` For the double-precision variant: is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

For the double-precision, zero variant: is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

`<Hn>` For the half-precision variant: is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

For the half-precision, zero variant: is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

`<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

`<Sn>` For the single-precision variant: is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

For the single-precision, zero variant: is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

`<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

NaNs

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands are NaNs, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. This case results in the FPSCR flags being set to N=0, Z=0, C=1, and V=1.

`FCMPE` raises an Invalid Operation exception if either operand is any type of NaN, and is suitable for testing for <, <=, >, >=, and other predicates that raise an exception when the operands are unordered.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;

operand2 = if cmp_with_zero then FPZero('0') else V[m];

PSTATE.<N,Z,C,V> = FPCompare(operand1, operand2, signal_all_nans, FPCR);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.68 FCSEL

Floating-point Conditional Select (scalar). This instruction allows the SIMD&FP destination register to take the value from either one or the other of two SIMD&FP source registers. If the condition passes, the first SIMD&FP source register value is taken, otherwise the second SIMD&FP source register value is taken.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15 … 12	11	10	9 … 5	4 … 0
0	0	0	1	1	1	1	0	ftype	1	Rm	cond	1	1	Rn	Rd

Half-precision variant

Applies when `ftype == 11`.

```
FCSEL <Hd>, <Hn>, <Hm>, <cond>
```

Single-precision variant

Applies when `ftype == 00`.

```
FCSEL <Sd>, <Sn>, <Sm>, <cond>
```

Double-precision variant

Applies when `ftype == 01`.

```
FCSEL <Dd>, <Dn>, <Dm>, <cond>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;
```

Assembler symbols

`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Dn>`	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
`<Dm>`	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
`<Hd>`	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Hn>`	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
`<Hm>`	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Sn>` Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

`<cond>` Is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;

result = if ConditionHolds(cond) then V[n] else V[m];

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.69 FCVT

Floating-point Convert precision (scalar). This instruction converts the floating-point value in the SIMD&FP source register to the precision for the destination register data type using the rounding mode that is determined by the FPCR and writes the result to the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	17	16	15	14	13	12	11	10	9 5	4 0
0	0	0	1	1	1	1	0	ftype	1	0	0	0	1	opc	1	0	0	0	0	0	Rn	Rd

Half-precision to single-precision variant

Applies when `ftype == 11 && opc == 00`.

```
FCVT <Sd>, <Hn>
```

Half-precision to double-precision variant

Applies when `ftype == 11 && opc == 01`.

```
FCVT <Dd>, <Hn>
```

Single-precision to half-precision variant

Applies when `ftype == 00 && opc == 11`.

```
FCVT <Hd>, <Sn>
```

Single-precision to double-precision variant

Applies when `ftype == 00 && opc == 01`.

```
FCVT <Dd>, <Sn>
```

Double-precision to half-precision variant

Applies when `ftype == 01 && opc == 11`.

```
FCVT <Hd>, <Dn>
```

Double-precision to single-precision variant

Applies when `ftype == 01 && opc == 00`.

```
FCVT <Sd>, <Dn>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer srcsize;
integer dstsize;

if ftype == opc then UNDEFINED;

case ftype of
    when '00' srcsize = 32;
    when '01' srcsize = 64;
    when '10' UNDEFINED;
    when '11' srcsize = 16;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
case opc of
    when '00' dstsize = 32;
    when '01' dstsize = 64;
    when '10' UNDEFINED;
    when '11' dstsize = 16;
```

Assembler symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(dstsize) result;
bits(srcsize) operand = V[n];

result = FPConvert(operand, FPCR);
V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.70 FCVTAS (vector)

Floating-point Convert to Signed integer, rounding to nearest with ties to Away (vector). This instruction converts each element in a vector from a floating-point value to a signed integer value using the Round to Nearest with Ties to Away rounding mode and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	0	1	0	Rn		Rd	
		U																							

Scalar half precision variant

```
FCVTAS <Hd>, <Hn>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	1	0	0	1	0	Rn		Rd	
		U																							

Scalar single-precision and double-precision variant

```
FCVTAS <V><d>, <V><n>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Vector half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	0	1	0	Rn	Rd
		U																					

Vector half precision variant

```
FCVTAS <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	1	0	0	1	0	Rn	Rd
		U																					

Vector single-precision and double-precision variant

```
FCVTAS <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

Assembler symbols

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

`<V>` Is a width specifier, encoded in the "sz" field. It can have the following values:

- `S` when `sz = 0`
- `D` when `sz = 1`

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<n>` Is the number of the SIMD&FP source register, encoded in the "Rn" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the vector half precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`4H` when `Q = 0`

`8H` when `Q = 1`

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

`2S` when `sz = 0`, `Q = 0`

`4S` when `sz = 0`, `Q = 1`

`2D` when `sz = 1`, `Q = 1`

The encoding `sz = 1`, `Q = 0` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.71 FCVTAS (scalar)

Floating-point Convert to Signed integer, rounding to nearest with ties to Away (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round to Nearest with Ties to Away rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	16	15	14	13	12	11	10	9	5	4	0
sf	0	0	1	1	1	1	0	ftype	1	0	0	1	0	0	0	0	0	0	0	Rn			Rd
										rmode		opcode											

Half-precision to 32-bit variant

Applies when `sf == 0 && ftype == 11`.

```
FCVTAS <Wd>, <Hn>
```

Half-precision to 64-bit variant

Applies when `sf == 1 && ftype == 11`.

```
FCVTAS <Xd>, <Hn>
```

Single-precision to 32-bit variant

Applies when `sf == 0 && ftype == 00`.

```
FCVTAS <Wd>, <Sn>
```

Single-precision to 64-bit variant

Applies when `sf == 1 && ftype == 00`.

```
FCVTAS <Xd>, <Sn>
```

Double-precision to 32-bit variant

Applies when `sf == 0 && ftype == 01`.

```
FCVTAS <Wd>, <Dn>
```

Double-precision to 64-bit variant

Applies when `sf == 1 && ftype == 01`.

```
FCVTAS <Xd>, <Dn>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

case ftype of
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
when '00'
    fltsize = 32;
when '01'
    fltsize = 64;
when '10'
    UNDEFINED;
when '11'
    if HaveFP16Ext() then
        fltsize = 16;
    else
        UNDEFINED;
```

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, 0, FALSE, FPCR, FPRounding_TIEAWAY);
X[d] = intval;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.72 FCVTAU (vector)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (vector). This instruction converts each element in a vector from a floating-point value to an unsigned integer value using the Round to Nearest with Ties to Away rounding mode and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	0	1	0	Rn		Rd	
		U																							

Scalar half precision variant

```
FCVTAU <Hd>, <Hn>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	1	0	0	1	0	Rn		Rd	
		U																							

Scalar single-precision and double-precision variant

```
FCVTAU <V><d>, <V><n>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Vector half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	0	1	0	Rn		Rd	
		U																							

Vector half precision variant

```
FCVTAU <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	1	0	0	1	0	Rn		Rd	
		U																							

Vector single-precision and double-precision variant

```
FCVTAU <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

Assembler symbols

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

`<V>` Is a width specifier, encoded in the "sz" field. It can have the following values:

- `S` when `sz = 0`
- `D` when `sz = 1`

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<n>` Is the number of the SIMD&FP source register, encoded in the "Rn" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the vector half precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`4H` when `Q = 0`

`8H` when `Q = 1`

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

`2S` when `sz = 0`, `Q = 0`

`4S` when `sz = 0`, `Q = 1`

`2D` when `sz = 1`, `Q = 1`

The encoding `sz = 1`, `Q = 0` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.73 FCVTAU (scalar)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round to Nearest with Ties to Away rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	16	15	14	13	12	11	10	9 5	4 0
sf	0	0	1	1	1	1	0	ftype	1	0	0	1	0	1	0	0	0	0	0	0	Rn	Rd
										rmode		opcode									

Half-precision to 32-bit variant

Applies when `sf == 0 && ftype == 11`.

```
FCVTAU <Wd>, <Hn>
```

Half-precision to 64-bit variant

Applies when `sf == 1 && ftype == 11`.

```
FCVTAU <Xd>, <Hn>
```

Single-precision to 32-bit variant

Applies when `sf == 0 && ftype == 00`.

```
FCVTAU <Wd>, <Sn>
```

Single-precision to 64-bit variant

Applies when `sf == 1 && ftype == 00`.

```
FCVTAU <Xd>, <Sn>
```

Double-precision to 32-bit variant

Applies when `sf == 0 && ftype == 01`.

```
FCVTAU <Wd>, <Dn>
```

Double-precision to 64-bit variant

Applies when `sf == 1 && ftype == 01`.

```
FCVTAU <Xd>, <Dn>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

case ftype of
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
when '00'
    fltsize = 32;
when '01'
    fltsize = 64;
when '10'
    UNDEFINED;
when '11'
    if HaveFP16Ext() then
        fltsize = 16;
    else
        UNDEFINED;
```

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, 0, TRUE, FPCR, FPRounding_TIEAWAY);
X[d] = intval;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.74 FCVTL, FCVTL2

Floating-point Convert to higher precision Long (vector). This instruction reads each element in a vector in the SIMD&FP source register, converts each value to double the precision of the source element using the rounding mode that is determined by the FPCR, and writes each result to the equivalent element of the vector in the SIMD&FP destination register.

Where the operation lengthens a 64-bit vector to a 128-bit vector, the `FCVTL2` variant operates on the elements in the top 64 bits of the source register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	sz	1	0	0	0	0	1	0	1	1	1	1	0	Rn		Rd	

Vector single-precision and double-precision variant

```
FCVTL{2} <Vd>.<Ta>, <Vn>.<Tb>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16 << UInt(sz);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- `[absent]` when `Q = 0`
- `[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "sz" field. It can have the following values:

- `4S` when `sz = 0`
- `2D` when `sz = 1`

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- `4H` when `sz = 0`, `Q = 0`
- `8H` when `sz = 0`, `Q = 1`
- `2S` when `sz = 1`, `Q = 0`
- `4S` when `sz = 1`, `Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part];
bits(2*datasize) result;

for e = 0 to elements-1
    Elem[result, e, 2*esize] = FPConvert(Elem[operand, e, esize], FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.75 FCVTMS (vector)

Floating-point Convert to Signed integer, rounding toward Minus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Scalar half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	0	1	1	1	1	0	0	1	1	0	1	1	1	0	Rn		Rd	
		U						o2											o1						

Scalar half precision variant

```
FCVTMS <Hd>, <Hn>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	1	1	1	0	Rn		Rd	
		U						o2											o1						

Scalar single-precision and double-precision variant

```
FCVTMS <V><d>, <V><n>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Vector half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	1	1	1	1	0	0	1	1	0	1	1	1	0	Rn		Rd	
		U						o2											o1						

Vector half precision variant

```
FCVTMS <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	1	1	1	0	Rn		Rd	
		U						o2											o1						

Vector single-precision and double-precision variant

```
FCVTMS <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler symbols

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

`<V>` Is a width specifier, encoded in the "sz" field. It can have the following values:

- `S` when `sz = 0`
- `D` when `sz = 1`

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<n>` Is the number of the SIMD&FP source register, encoded in the "Rn" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the vector half precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`4H` when `Q = 0`

`8H` when `Q = 1`

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

`2S` when `sz = 0`, `Q = 0`

`4S` when `sz = 0`, `Q = 1`

`2D` when `sz = 1`, `Q = 1`

The encoding `sz = 1`, `Q = 0` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.76 FCVTMS (scalar)

Floating-point Convert to Signed integer, rounding toward Minus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round towards Minus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19 18	16	15 14 13 12	11 10	9 5	4 0
sf	0	0	1	1	1	1	0	ftype	1	1 0	0 0	0	0 0 0 0	0 0	Rn	Rd
										rmode	opcode					

Half-precision to 32-bit variant

Applies when `sf == 0 && ftype == 11`.

```
FCVTMS <Wd>, <Hn>
```

Half-precision to 64-bit variant

Applies when `sf == 1 && ftype == 11`.

```
FCVTMS <Xd>, <Hn>
```

Single-precision to 32-bit variant

Applies when `sf == 0 && ftype == 00`.

```
FCVTMS <Wd>, <Sn>
```

Single-precision to 64-bit variant

Applies when `sf == 1 && ftype == 00`.

```
FCVTMS <Xd>, <Sn>
```

Double-precision to 32-bit variant

Applies when `sf == 0 && ftype == 01`.

```
FCVTMS <Wd>, <Dn>
```

Double-precision to 64-bit variant

Applies when `sf == 1 && ftype == 01`.

```
FCVTMS <Xd>, <Dn>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;

rounding = FPDecodeRounding(rmode);
```

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, 0, FALSE, FPCR, rounding);
X[d] = intval;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.77 FCVTMU (vector)

Floating-point Convert to Unsigned integer, rounding toward Minus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Scalar half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	0	1	1	1	1	0	0	1	1	0	1	1	1	0	Rn		Rd	
		U						o2											o1						

Scalar half precision variant

```
FCVTMU <Hd>, <Hn>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	1	1	1	0	Rn		Rd	
		U						o2											o1						

Scalar single-precision and double-precision variant

```
FCVTMU <V><d>, <V><n>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Vector half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	0	1	1	1	1	0	0	1	1	0	1	1	1	0	Rn		Rd	
		U						o2											o1						

Vector half precision variant

```
FCVTMU <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	1	1	1	0	Rn		Rd	
		U						o2											o1						

Vector single-precision and double-precision variant

```
FCVTMU <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler symbols

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

`<V>` Is a width specifier, encoded in the "sz" field. It can have the following values:

- `S` when `sz = 0`
- `D` when `sz = 1`

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<n>` Is the number of the SIMD&FP source register, encoded in the "Rn" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the vector half precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`4H` when `Q = 0`

`8H` when `Q = 1`

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

`2S` when `sz = 0`, `Q = 0`

`4S` when `sz = 0`, `Q = 1`

`2D` when `sz = 1`, `Q = 1`

The encoding `sz = 1`, `Q = 0` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.78 FCVTMU (scalar)

Floating-point Convert to Unsigned integer, rounding toward Minus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Minus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19 18	16	15 14 13 12	11 10 9	5	4	0
sf	0	0	1	1	1	1	0	ftype	1	1	0	0 0 1	0 0 0 0	0 0	Rn		Rd
										rmode		opcode					

Half-precision to 32-bit variant

Applies when `sf == 0 && ftype == 11`.

```
FCVTMU <Wd>, <Hn>
```

Half-precision to 64-bit variant

Applies when `sf == 1 && ftype == 11`.

```
FCVTMU <Xd>, <Hn>
```

Single-precision to 32-bit variant

Applies when `sf == 0 && ftype == 00`.

```
FCVTMU <Wd>, <Sn>
```

Single-precision to 64-bit variant

Applies when `sf == 1 && ftype == 00`.

```
FCVTMU <Xd>, <Sn>
```

Double-precision to 32-bit variant

Applies when `sf == 0 && ftype == 01`.

```
FCVTMU <Wd>, <Dn>
```

Double-precision to 64-bit variant

Applies when `sf == 1 && ftype == 01`.

```
FCVTMU <Xd>, <Dn>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;

rounding = FPDecodeRounding(rmode);
```

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, 0, TRUE, FPCR, rounding);
X[d] = intval;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.79 FCVTN, FCVTN2

Floating-point Convert to lower precision Narrow (vector). This instruction reads each vector element in the SIMD&FP source register, converts each result to half the precision of the source element, writes the final result to a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. The rounding mode is determined by the FPCR.

The `FCVTN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `FCVTN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	0	sz	1	0	0	0	0	1	0	1	1	0	1	0	Rn	Rd

Vector single-precision and double-precision variant

```
FCVTN{2} <Vd>.<Tb>, <Vn>.<Ta>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16 << UInt(sz);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- `[absent]` when `Q = 0`
- `[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Tb>` Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- `4H` when `sz = 0`, `Q = 0`
- `8H` when `sz = 0`, `Q = 1`
- `2S` when `sz = 1`, `Q = 0`
- `4S` when `sz = 1`, `Q = 1`

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<Ta>` Is an arrangement specifier, encoded in the "sz" field. It can have the following values:

- `4S` when `sz = 0`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

2D	when sz = 1

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;

for e = 0 to elements-1
    Elem[result, e, esize] = FPConvert(Elem[operand, e, 2*esize], FPCR);

Vpart[d, part] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.80 FCVTNS (vector)

Floating-point Convert to Signed integer, rounding to nearest with ties to even (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Scalar half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	0	1	1	1	1	0	0	1	1	0	1	0	1	0	Rn		Rd	
		U						o2											o1						

Scalar half precision variant

```
FCVTNS <Hd>, <Hn>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	1	0	1	0	Rn		Rd	
		U						o2											o1						

Scalar single-precision and double-precision variant

```
FCVTNS <V><d>, <V><n>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Vector half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	1	1	1	1	0	0	1	1	0	1	0	1	0	Rn		Rd	
		U						o2											o1						

Vector half precision variant

```
FCVTNS <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	1	0	1	0	Rn		Rd	
		U						o2											o1						

Vector single-precision and double-precision variant

```
FCVTNS <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler symbols

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

`<V>` Is a width specifier, encoded in the "sz" field. It can have the following values:

- `S` when `sz = 0`
- `D` when `sz = 1`

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<n>` Is the number of the SIMD&FP source register, encoded in the "Rn" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the vector half precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`4H` when `Q = 0`

`8H` when `Q = 1`

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

`2S` when `sz = 0`, `Q = 0`

`4S` when `sz = 0`, `Q = 1`

`2D` when `sz = 1`, `Q = 1`

The encoding `sz = 1`, `Q = 0` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.81 FCVTNS (scalar)

Floating-point Convert to Signed integer, rounding to nearest with ties to even (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round to Nearest rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19 18	16	15 14	13	12	11	10	9 5	4 0
sf	0	0	1	1	1	1	0	ftype	1	0	0	0	0	0	0	0	0	Rn	Rd
										rmode		opcode							

Half-precision to 32-bit variant

Applies when `sf == 0 && ftype == 11`.

```
FCVTNS <Wd>, <Hn>
```

Half-precision to 64-bit variant

Applies when `sf == 1 && ftype == 11`.

```
FCVTNS <Xd>, <Hn>
```

Single-precision to 32-bit variant

Applies when `sf == 0 && ftype == 00`.

```
FCVTNS <Wd>, <Sn>
```

Single-precision to 64-bit variant

Applies when `sf == 1 && ftype == 00`.

```
FCVTNS <Xd>, <Sn>
```

Double-precision to 32-bit variant

Applies when `sf == 0 && ftype == 01`.

```
FCVTNS <Wd>, <Dn>
```

Double-precision to 64-bit variant

Applies when `sf == 1 && ftype == 01`.

```
FCVTNS <Xd>, <Dn>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;

rounding = FPDecodeRounding(rmode);
```

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, 0, FALSE, FPCR, rounding);
X[d] = intval;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.82 FCVTNU (vector)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Scalar half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	0	1	1	1	1	0	0	1	1	0	1	0	1	0	Rn		Rd	
		U						o2											o1						

Scalar half precision variant

```
FCVTNU <Hd>, <Hn>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	1	0	1	0	Rn		Rd	
		U						o2											o1						

Scalar single-precision and double-precision variant

```
FCVTNU <V><d>, <V><n>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Vector half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	0	1	1	1	1	0	0	1	1	0	1	0	1	0	Rn		Rd	
		U						o2											o1						

Vector half precision variant

```
FCVTNU <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	1	0	1	0	Rn		Rd	
		U						o2											o1						

Vector single-precision and double-precision variant

```
FCVTNU <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler symbols

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

`<V>` Is a width specifier, encoded in the "sz" field. It can have the following values:

- `S` when `sz = 0`
- `D` when `sz = 1`

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<n>` Is the number of the SIMD&FP source register, encoded in the "Rn" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the vector half precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`4H` when `Q = 0`

`8H` when `Q = 1`

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

`2S` when `sz = 0`, `Q = 0`

`4S` when `sz = 0`, `Q = 1`

`2D` when `sz = 1`, `Q = 1`

The encoding `sz = 1`, `Q = 0` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.83 FCVTNU (scalar)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round to Nearest rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	16	15	14	13	12	11	10	9 5	4 0
sf	0	0	1	1	1	1	0	ftype	1	0	0	0	1	0	0	0	0	0	0	Rn	Rd
										rmode		opcode									

Half-precision to 32-bit variant

Applies when `sf == 0 && ftype == 11`.

```
FCVTNU <Wd>, <Hn>
```

Half-precision to 64-bit variant

Applies when `sf == 1 && ftype == 11`.

```
FCVTNU <Xd>, <Hn>
```

Single-precision to 32-bit variant

Applies when `sf == 0 && ftype == 00`.

```
FCVTNU <Wd>, <Sn>
```

Single-precision to 64-bit variant

Applies when `sf == 1 && ftype == 00`.

```
FCVTNU <Xd>, <Sn>
```

Double-precision to 32-bit variant

Applies when `sf == 0 && ftype == 01`.

```
FCVTNU <Wd>, <Dn>
```

Double-precision to 64-bit variant

Applies when `sf == 1 && ftype == 01`.

```
FCVTNU <Xd>, <Dn>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;

rounding = FPDecodeRounding(rmode);
```

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, 0, TRUE, FPCR, rounding);
X[d] = intval;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.84 FCVTPS (vector)

Floating-point Convert to Signed integer, rounding toward Plus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Scalar half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	0	1	0	Rn		Rd	
		U						o2											o1						

Scalar half precision variant

```
FCVTPS <Hd>, <Hn>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	1	0	1	0	Rn		Rd	
		U						o2											o1						

Scalar single-precision and double-precision variant

```
FCVTPS <V><d>, <V><n>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Vector half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	0	1	0	Rn		Rd	
		U						o2											o1						

Vector half precision variant

```
FCVTPS <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	1	0	1	0	Rn		Rd	
		U						o2											o1						

Vector single-precision and double-precision variant

```
FCVTPS <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler symbols

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

`<V>` Is a width specifier, encoded in the "sz" field. It can have the following values:

- `S` when `sz = 0`
- `D` when `sz = 1`

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<n>` Is the number of the SIMD&FP source register, encoded in the "Rn" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the vector half precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`4H` when `Q = 0`

`8H` when `Q = 1`

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

`2S` when `sz = 0`, `Q = 0`

`4S` when `sz = 0`, `Q = 1`

`2D` when `sz = 1`, `Q = 1`

The encoding `sz = 1`, `Q = 0` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.85 FCVTPS (scalar)

Floating-point Convert to Signed integer, rounding toward Plus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round towards Plus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19 18	16	15 14 13 12	11 10	9 5	4 0
sf	0	0	1	1	1	1	0	ftype	1	0 1	0 0	0	0 0 0 0	0 0	Rn	Rd
										rmode	opcode					

Half-precision to 32-bit variant

Applies when `sf == 0 && ftype == 11`.

```
FCVTPS <Wd>, <Hn>
```

Half-precision to 64-bit variant

Applies when `sf == 1 && ftype == 11`.

```
FCVTPS <Xd>, <Hn>
```

Single-precision to 32-bit variant

Applies when `sf == 0 && ftype == 00`.

```
FCVTPS <Wd>, <Sn>
```

Single-precision to 64-bit variant

Applies when `sf == 1 && ftype == 00`.

```
FCVTPS <Xd>, <Sn>
```

Double-precision to 32-bit variant

Applies when `sf == 0 && ftype == 01`.

```
FCVTPS <Wd>, <Dn>
```

Double-precision to 64-bit variant

Applies when `sf == 1 && ftype == 01`.

```
FCVTPS <Xd>, <Dn>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;

rounding = FPDecodeRounding(rmode);
```

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, 0, FALSE, FPCR, rounding);
X[d] = intval;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.86 FCVTPU (vector)

Floating-point Convert to Unsigned integer, rounding toward Plus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Scalar half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	0	1	0	Rn		Rd	
		U						o2											o1						

Scalar half precision variant

```
FCVTPU <Hd>, <Hn>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	1	0	1	0	Rn		Rd	
		U						o2											o1						

Scalar single-precision and double-precision variant

```
FCVTPU <V><d>, <V><n>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Vector half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	0	1	0	Rn		Rd	
		U						o2											o1						

Vector half precision variant

```
FCVTPU <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	1	0	1	0	Rn		Rd	
		U						o2											o1						

Vector single-precision and double-precision variant

```
FCVTPU <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler symbols

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

`<V>` Is a width specifier, encoded in the "sz" field. It can have the following values:

- `S` when `sz = 0`
- `D` when `sz = 1`

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<n>` Is the number of the SIMD&FP source register, encoded in the "Rn" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the vector half precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`4H` when `Q = 0`

`8H` when `Q = 1`

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

`2S` when `sz = 0`, `Q = 0`

`4S` when `sz = 0`, `Q = 1`

`2D` when `sz = 1`, `Q = 1`

The encoding `sz = 1`, `Q = 0` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.87 FCVTPU (scalar)

Floating-point Convert to Unsigned integer, rounding toward Plus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Plus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	16	15	14	13	12	11	10	9	5	4	0
sf	0	0	1	1	1	1	0	ftype	1	0	1	0	0	1	0	0	0	0	0	0	Rn		Rd
										rmode		opcode											

Half-precision to 32-bit variant

Applies when `sf == 0 && ftype == 11`.

```
FCVTPU <Wd>, <Hn>
```

Half-precision to 64-bit variant

Applies when `sf == 1 && ftype == 11`.

```
FCVTPU <Xd>, <Hn>
```

Single-precision to 32-bit variant

Applies when `sf == 0 && ftype == 00`.

```
FCVTPU <Wd>, <Sn>
```

Single-precision to 64-bit variant

Applies when `sf == 1 && ftype == 00`.

```
FCVTPU <Xd>, <Sn>
```

Double-precision to 32-bit variant

Applies when `sf == 0 && ftype == 01`.

```
FCVTPU <Wd>, <Dn>
```

Double-precision to 64-bit variant

Applies when `sf == 1 && ftype == 01`.

```
FCVTPU <Xd>, <Dn>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;

rounding = FPDecodeRounding(rmode);
```

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, 0, TRUE, FPCR, rounding);
X[d] = intval;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.88 FCVTXN, FCVTXN2

Floating-point Convert to lower precision Narrow, rounding to odd (vector). This instruction reads each vector element in the source SIMD&FP register, narrows each value to half the precision of the source element using the Round to Odd rounding mode, writes the result to a vector, and writes the vector to the destination SIMD&FP register.

Note

This instruction uses the Round to Odd rounding mode which is not defined by the IEEE 754-2008 standard. This rounding mode ensures that if the result of the conversion is inexact the least significant bit of the mantissa is forced to 1. This rounding mode enables a floating-point value to be converted to a lower precision format via an intermediate precision format while avoiding double rounding errors. For example, a 64-bit floating-point value can be converted to a correctly rounded 16-bit floating-point value by first using this instruction to produce a 32-bit value and then using another instruction with the wanted rounding mode to convert the 32-bit value to the final 16-bit floating-point value.

The `FCVTXN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `FCVTXN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	1	1	1	1	1	0	0	sz	1	0	0	0	0	1	0	1	1	0	1	0	Rn	Rd

Scalar variant

```
FCVTXN <Vb><d>, <Va><n>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz == '0' then UNDEFINED;
integer esize = 32;
integer datasize = esize;
integer elements = 1;
integer part = 0;
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	0	sz	1	0	0	0	0	1	0	1	1	0	1	0	Rn	Rd

Vector variant

```
FCVTXN{2} <Vd>.<Tb>, <Vn>.<Ta>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz == '0' then UNDEFINED;
integer esize = 32;
integer datasize = 64;
integer elements = 2;
integer part = UInt(Q);
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Tb>` Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

`2S` when `sz = 1`, `Q = 0`

`4S` when `sz = 1`, `Q = 1`

The encoding `sz = 0`, `Q = x` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<Ta>` Is an arrangement specifier, encoded in the "sz" field. It can have the following values:

`2D` when `sz = 1`

The encoding `sz = 0` is reserved.

`<Vb>` Is the destination width specifier, encoded in the "sz" field. It can have the following values:

`S` when `sz = 1`

The encoding `sz = 0` is reserved.

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

`<Va>` Is the source width specifier, encoded in the "sz" field. It can have the following values:

`D` when `sz = 1`

The encoding `sz = 0` is reserved.

`<n>` Is the number of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;

for e = 0 to elements-1
    Elem[result, e, esize] = FPConvert(Elem[operand, e, 2*esize], FPCR, FPRounding_ODD);

Vpart[d, part] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.89 FCVTZS (vector, fixed-point)

Floating-point Convert to Signed fixed-point, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from floating-point to fixed-point signed integer using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23	22	19	18	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	1	0	!=0000	immb			1	1	1	1	1	1	Rn		Rd	
		U							immh													

Scalar variant

```
FCVTZS <V><d>, <V><n>, #<fbits>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then UNDEFINED;
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;
```

Vector

31	30	29	28	27	26	25	24	23	22	19	18	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	1	0	!=0000	immb			1	1	1	1	1	1	Rn		Rd	
		U							immh													

Vector variant

```
FCVTZS <Vd>.<T>, <Vn>.<T>, #<fbits>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then UNDEFINED;
if immh<3>:Q == '10' then UNDEFINED;
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;
```

Assembler symbols

<V> Is a width specifier, encoded in the "immh" field. It can have the following values:

H when immh = 001x

S when immh = 01xx

D when immh = 1xxx

The encoding immh = 000x is reserved.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

2D when immh = 1xxx, Q = 1

See *Advanced SIMD modified immediate* on page C4-354 when immh = 0000, Q = x.

The following encodings are reserved:

- immh = 0001, Q = x.
- immh = 1xxx, Q = 0.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits> For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in the "immh:immb" field. It can have the following values:

(32-Uint(immh:immb)) when immh = 001x

(64-UInt(immh:immb)) when immh = 01xx

(128-UInt(immh:immb)) when immh = 1xxx

The encoding immh = 000x is reserved.

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in the "immh:immb" field. It can have the following values:

(32-Uint(immh:immb)) when immh = 001x

(64-UInt(immh:immb)) when immh = 01xx

(128-UInt(immh:immb)) when immh = 1xxx

See *Advanced SIMD modified immediate* on page C4-354 when immh = 0000.

The encoding immh = 0001 is reserved.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, fracbits, unsigned, FPCR, rounding);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.90 FCVTZS (vector, integer)

Floating-point Convert to Signed integer, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Scalar half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	1	1	0	Rn		Rd	
		U						o2											o1						

Scalar half precision variant

```
FCVTZS <Hd>, <Hn>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	1	1	1	0	Rn		Rd	
		U						o2											o1						

Scalar single-precision and double-precision variant

```
FCVTZS <V><d>, <V><n>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Vector half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	1	1	0	Rn		Rd	
		U						o2											o1						

Vector half precision variant

```
FCVTZS <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	1	1	1	0	Rn		Rd	
		U						o2											o1						

Vector single-precision and double-precision variant

```
FCVTZS <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler symbols

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

`<V>` Is a width specifier, encoded in the "sz" field. It can have the following values:

- `S` when `sz = 0`
- `D` when `sz = 1`

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<n>` Is the number of the SIMD&FP source register, encoded in the "Rn" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the vector half precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`4H` when `Q = 0`

`8H` when `Q = 1`

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

`2S` when `sz = 0`, `Q = 0`

`4S` when `sz = 0`, `Q = 1`

`2D` when `sz = 1`, `Q = 1`

The encoding `sz = 1`, `Q = 0` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.91 FCVTZS (scalar, fixed-point)

Floating-point Convert to Signed fixed-point, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit fixed-point signed integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	16	15 10	9 5	4 0
sf	0	0	1	1	1	1	0	ftype	0	1	1	0	0 0	scale	Rn	Rd
										rmode		opcode				

Half-precision to 32-bit variant

Applies when `sf == 0 && ftype == 11`.

```
FCVTZS <Wd>, <Hn>, #<fbits>
```

Half-precision to 64-bit variant

Applies when `sf == 1 && ftype == 11`.

```
FCVTZS <Xd>, <Hn>, #<fbits>
```

Single-precision to 32-bit variant

Applies when `sf == 0 && ftype == 00`.

```
FCVTZS <Wd>, <Sn>, #<fbits>
```

Single-precision to 64-bit variant

Applies when `sf == 1 && ftype == 00`.

```
FCVTZS <Xd>, <Sn>, #<fbits>
```

Double-precision to 32-bit variant

Applies when `sf == 0 && ftype == 01`.

```
FCVTZS <Wd>, <Dn>, #<fbits>
```

Double-precision to 64-bit variant

Applies when `sf == 1 && ftype == 01`.

```
FCVTZS <Xd>, <Dn>, #<fbits>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

case ftype of
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    when '00' fltsize = 32;
    when '01' fltsize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;

if sf == '0' && scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - UInt(scale);
```

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Sn>`	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
`<Hn>`	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
`<Dn>`	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
`<fbits>`	For the double-precision to 32-bit, half-precision to 32-bit and single-precision to 32-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 32, encoded as 64 minus "scale". For the double-precision to 64-bit, half-precision to 64-bit and single-precision to 64-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 64, encoded as 64 minus "scale".

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, fracbits, FALSE, FPCR, FPRounding_ZERO);
X[d] = intval;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.92 FCVTZS (scalar, integer)

Floating-point Convert to Signed integer, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	16	15	14	13	12	11	10	9 5	4 0
sf	0	0	1	1	1	1	0	ftype	1	1	1	0	0	0	0	0	0	0	0	Rn	Rd
										rmode		opcode									

Half-precision to 32-bit variant

Applies when `sf == 0 && ftype == 11`.

```
FCVTZS <Wd>, <Hn>
```

Half-precision to 64-bit variant

Applies when `sf == 1 && ftype == 11`.

```
FCVTZS <Xd>, <Hn>
```

Single-precision to 32-bit variant

Applies when `sf == 0 && ftype == 00`.

```
FCVTZS <Wd>, <Sn>
```

Single-precision to 64-bit variant

Applies when `sf == 1 && ftype == 00`.

```
FCVTZS <Xd>, <Sn>
```

Double-precision to 32-bit variant

Applies when `sf == 0 && ftype == 01`.

```
FCVTZS <Wd>, <Dn>
```

Double-precision to 64-bit variant

Applies when `sf == 1 && ftype == 01`.

```
FCVTZS <Xd>, <Dn>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;

rounding = FPDecodeRounding(rmode);
```

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, 0, FALSE, FPCR, rounding);
X[d] = intval;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.93 FCVTZU (vector, fixed-point)

Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from floating-point to fixed-point unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23	22	19	18	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	1	0	!=0000	immb			1	1	1	1	1	1	Rn		Rd	
		U							immh													

Scalar variant

```
FCVTZU <V><d>, <V><n>, #<fbits>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then UNDEFINED;
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;
```

Vector

31	30	29	28	27	26	25	24	23	22	19	18	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	1	0	!=0000	immb			1	1	1	1	1	1	Rn		Rd	
		U							immh													

Vector variant

```
FCVTZU <Vd>.<T>, <Vn>.<T>, #<fbits>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then UNDEFINED;
if immh<3>:Q == '10' then UNDEFINED;
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;
```

Assembler symbols

<V> Is a width specifier, encoded in the "immh" field. It can have the following values:

H when immh = 001x

S when immh = 01xx

D when immh = 1xxx

The encoding immh = 000x is reserved.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

2D when immh = 1xxx, Q = 1

See *Advanced SIMD modified immediate* on page C4-354 when immh = 0000, Q = x.

The following encodings are reserved:

- immh = 0001, Q = x.
- immh = 1xxx, Q = 0.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits> For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in the "immh:immb" field. It can have the following values:

(32-Uint(immh:immb)) when immh = 001x

(64-UInt(immh:immb)) when immh = 01xx

(128-UInt(immh:immb)) when immh = 1xxx

The encoding immh = 000x is reserved.

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in the "immh:immb" field. It can have the following values:

(32-Uint(immh:immb)) when immh = 001x

(64-UInt(immh:immb)) when immh = 01xx

(128-UInt(immh:immb)) when immh = 1xxx

See *Advanced SIMD modified immediate* on page C4-354 when immh = 0000.

The encoding immh = 0001 is reserved.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, fracbits, unsigned, FPCR, rounding);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.94 FCVTZU (vector, integer)

Floating-point Convert to Unsigned integer, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Scalar half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	1	1	0	Rn		Rd	
		U						o2											o1						

Scalar half precision variant

```
FCVTZU <Hd>, <Hn>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	1	1	1	0	Rn		Rd	
		U						o2											o1						

Scalar single-precision and double-precision variant

```
FCVTZU <V><d>, <V><n>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Vector half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	1	1	0	Rn		Rd	
		U						o2											o1						

Vector half precision variant

```
FCVTZU <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	1	1	1	0	Rn		Rd	
		U						o2											o1						

Vector single-precision and double-precision variant

```
FCVTZU <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler symbols

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

`<V>` Is a width specifier, encoded in the "sz" field. It can have the following values:

- `S` when `sz = 0`
- `D` when `sz = 1`

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<n>` Is the number of the SIMD&FP source register, encoded in the "Rn" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the vector half precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`4H` when `Q = 0`

`8H` when `Q = 1`

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

`2S` when `sz = 0`, `Q = 0`

`4S` when `sz = 0`, `Q = 1`

`2D` when `sz = 1`, `Q = 1`

The encoding `sz = 1`, `Q = 0` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.95 FCVTZU (scalar, fixed-point)

Floating-point Convert to Unsigned fixed-point, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit fixed-point unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 19	18 16	15 10	9 5	4 0
sf	0	0	1	1	1	1	0	ftype	0	1 1	0 0 1	scale	Rn	Rd
											rmode opcode			

Half-precision to 32-bit variant

Applies when `sf == 0 && ftype == 11`.

```
FCVTZU <Wd>, <Hn>, #<fbits>
```

Half-precision to 64-bit variant

Applies when `sf == 1 && ftype == 11`.

```
FCVTZU <Xd>, <Hn>, #<fbits>
```

Single-precision to 32-bit variant

Applies when `sf == 0 && ftype == 00`.

```
FCVTZU <Wd>, <Sn>, #<fbits>
```

Single-precision to 64-bit variant

Applies when `sf == 1 && ftype == 00`.

```
FCVTZU <Xd>, <Sn>, #<fbits>
```

Double-precision to 32-bit variant

Applies when `sf == 0 && ftype == 01`.

```
FCVTZU <Wd>, <Dn>, #<fbits>
```

Double-precision to 64-bit variant

Applies when `sf == 1 && ftype == 01`.

```
FCVTZU <Xd>, <Dn>, #<fbits>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

case ftype of
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    when '00' fltsize = 32;
    when '01' fltsize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;

if sf == '0' && scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - UInt(scale);
```

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<fbits>	For the double-precision to 32-bit, half-precision to 32-bit and single-precision to 32-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 32, encoded as 64 minus "scale". For the double-precision to 64-bit, half-precision to 64-bit and single-precision to 64-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 64, encoded as 64 minus "scale".

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, fracbits, TRUE, FPCR, FPRounding_ZERO);
X[d] = intval;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.96 FCVTZU (scalar, integer)

Floating-point Convert to Unsigned integer, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	16	15	14	13	12	11	10	9 5	4 0
sf	0	0	1	1	1	1	0	ftype	1	1	1	0	0	1	0	0	0	0	0	0	Rn	Rd

rmode opcode

Half-precision to 32-bit variant

Applies when `sf == 0 && ftype == 11`.

```
FCVTZU <Wd>, <Hn>
```

Half-precision to 64-bit variant

Applies when `sf == 1 && ftype == 11`.

```
FCVTZU <Xd>, <Hn>
```

Single-precision to 32-bit variant

Applies when `sf == 0 && ftype == 00`.

```
FCVTZU <Wd>, <Sn>
```

Single-precision to 64-bit variant

Applies when `sf == 1 && ftype == 00`.

```
FCVTZU <Xd>, <Sn>
```

Double-precision to 32-bit variant

Applies when `sf == 0 && ftype == 01`.

```
FCVTZU <Wd>, <Dn>
```

Double-precision to 64-bit variant

Applies when `sf == 1 && ftype == 01`.

```
FCVTZU <Xd>, <Dn>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;

rounding = FPDecodeRounding(rmode);
```

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, 0, TRUE, FPCR, rounding);
X[d] = intval;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.97 FDIV (vector)

Floating-point Divide (vector). This instruction divides the floating-point values in the elements in the first source SIMD&FP register, by the floating-point values in the corresponding elements in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	0	1	0	Rm		0	0	1	1	1	1	Rn		Rd	

Half-precision variant

```
FDIV <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	0	sz	1	Rm		1	1	1	1	1	1	Rn		Rd	

Single-precision and double-precision variant

```
FDIV <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

4H when Q = 0

8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0

4S when sz = 0, Q = 1

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPDiv(element1, element2, FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.98 FDIV (scalar)

Floating-point Divide (scalar). This instruction divides the floating-point value of the first source SIMD&FP register by the floating-point value of the second source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	0	0	1	1	1	1	0	ftype	1	Rm	0	0	0	1	1	0	Rn	Rd

Half-precision variant

Applies when `ftype == 11`.

```
FDIV <Hd>, <Hn>, <Hm>
```

Single-precision variant

Applies when `ftype == 00`.

```
FDIV <Sd>, <Sn>, <Sm>
```

Double-precision variant

Applies when `ftype == 01`.

```
FDIV <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;
```

Assembler symbols

`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Dn>`	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
`<Dm>`	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
`<Hd>`	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Hn>` Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Sn>` Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result = FPDiv(operand1, operand2, FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.99 FJCVTZS

Floating-point JavaScript Convert to Signed fixed-point, rounding toward Zero. This instruction converts the double-precision floating-point value in the SIMD&FP source register to a 32-bit signed integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register. If the result is too large to be accommodated as a signed 32-bit integer, then the result is the integer modulo 2^{32}, as held in a 32-bit signed integer.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

ARMv8.3

31	30	29	28	27	26	25	24	23	22	21	20	19	18	16	15	14	13	12	11	10	9	5	4	0
0	0	0	1	1	1	1	0	0	1	1	1	1	1	0	0	0	0	0	0	0	Rn			Rd
sf								ftype			rmode		opcode											

Double-precision to 32-bit variant

```
FJCVTZS <Wd>, <Dn>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HaveFJCVTZSExt() then UNDEFINED;
```

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) fltval;
bits(32) intval;

bit Z;
fltval = V[n];
(intval, Z) = FPToFixedJS(fltval, FPCR, TRUE);
PSTATE.<N,Z,C,V> = '0':Z:'00';
X[d] = intval;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.100 FMADD

Floating-point fused Multiply-Add (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, adds the product to the value of the third SIMD&FP source register, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14 … 10	9 … 5	4 … 0
0	0	0	1	1	1	1	1	ftype	0	Rm	0	Ra	Rn	Rd
									o1		o0			

Half-precision variant

Applies when `ftype == 11`.

```
FMADD <Hd>, <Hn>, <Hm>, <Ha>
```

Single-precision variant

Applies when `ftype == 00`.

```
FMADD <Sd>, <Sn>, <Sm>, <Sa>
```

Double-precision variant

Applies when `ftype == 01`.

```
FMADD <Dd>, <Dn>, <Dm>, <Da>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;
```

Assembler symbols

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Dm>` Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

`<Da>` Is the 64-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

`<Hm>` Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

`<Ha>` Is the 16-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Sn>` Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

`<Sm>` Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

`<Sa>` Is the 32-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operanda = V[a];
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result = FPMulAdd(operanda, operand1, operand2, FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.101 FMAX (vector)

Floating-point Maximum (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, places the larger of each of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	1	0	Rm		0	0	1	1	0	1	Rn		Rd	
		U						o1														

Half-precision variant

```
FMAX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	sz	1	Rm		1	1	1	1	0	1	Rn		Rd	
		U						o1														

Single-precision and double-precision variant

```
FMAX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`4H` when `Q = 0`

`8H` when `Q = 1`

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

`2S` when `sz = 0`, `Q = 0`

`4S` when `sz = 0`, `Q = 1`

`2D` when `sz = 1`, `Q = 1`

The encoding `sz = 1`, `Q = 0` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMin(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMax(element1, element2, FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.102 FMAX (scalar)

Floating-point Maximum (scalar). This instruction compares the two source SIMD&FP registers, and writes the larger of the two floating-point values to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	0	0	1	1	1	1	0	ftype	1	Rm	0	1	0	0	1	0	Rn	Rd
													op					

Half-precision variant

Applies when `ftype == 11`.

```
FMAX <Hd>, <Hn>, <Hm>
```

Single-precision variant

Applies when `ftype == 00`.

```
FMAX <Sd>, <Sn>, <Sm>
```

Double-precision variant

Applies when `ftype == 01`.

```
FMAX <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;
```

Assembler symbols

`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Dn>`	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
`<Dm>`	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
`<Hd>`	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Hn>`	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Sn>` Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result = FPMax(operand1, operand2, FPCR);
V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.103 FMAXNM (vector)

Floating-point Maximum Number (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, writes the larger of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result placed in the vector is the numerical value, otherwise the result is identical to FMAX (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	1	0	Rm		0	0	0	0	0	1	Rn		Rd	
		U						a														

Half-precision variant

```
FMAXNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	sz	1	Rm		1	1	0	0	0	1	Rn		Rd	
		U						o1														

Single-precision and double-precision variant

```
FMAXNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

4H when Q = 0

8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0

4S when sz = 0, Q = 1

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMinNum(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.104 FMAXNM (scalar)

Floating-point Maximum Number (scalar). This instruction compares the first and second source SIMD&FP register values, and writes the larger of the two floating-point values to the destination SIMD&FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result that is placed in the vector is the numerical value, otherwise the result is identical to FMAX (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	0	0	1	1	1	1	0	ftype	1	Rm	0	1	1	0	1	0	Rn	Rd
													op					

Half-precision variant

Applies when `ftype == 11`.

```
FMAXNM <Hd>, <Hn>, <Hm>
```

Single-precision variant

Applies when `ftype == 00`.

```
FMAXNM <Sd>, <Sn>, <Sm>
```

Double-precision variant

Applies when `ftype == 01`.

```
FMAXNM <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;
```

Assembler symbols

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Sn>` Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result = FPMaxNum(operand1, operand2, FPCR);
V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.105 FMAXNMP (scalar)

Floating-point Maximum Number of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the largest of the floating-point values as a scalar to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	0	0	1	1	0	0	0	0	1	1	0	0	1	0	Rn		Rd	
								o1	sz																

Half-precision variant

```
FMAXNMP <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = 32;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	0	sz	1	1	0	0	0	0	1	1	0	0	1	0	Rn		Rd	
								o1																	

Single-precision and double-precision variant

```
FMAXNMP <V><d>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32;
integer datasize = 64;
```

Assembler symbols

<V> For the half-precision variant: is the destination width specifier, encoded in the "sz" field. It can have the following values:

H when sz = 0

The encoding sz = 1 is reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the single-precision and double-precision variant: is the destination width specifier, encoded in the "sz" field. It can have the following values:

`S` when `sz = 0`

`D` when `sz = 1`

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<T>` For the half-precision variant: is the source arrangement specifier, encoded in the "sz" field. It can have the following values:

`2H` when `sz = 0`

The encoding `sz = 1` is reserved.

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in the "sz" field. It can have the following values:

`2S` when `sz = 0`

`2D` when `sz = 1`

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMAXNUM, operand, esize);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.106 FMAXNMP (vector)

Floating-point Maximum Number Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result is the numerical value, otherwise the result is identical to FMAX (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	0	1	0	Rm		0	0	0	0	0	1	Rn		Rd	
		U						a														

Half-precision variant

```
FMAXNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	0	sz	1	Rm		1	1	0	0	0	1	Rn		Rd	
		U						o1														

Single-precision and double-precision variant

```
FMAXNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- 4H when Q = 0
- 8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMinNum(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.107 FMAXNMV

Floating-point Maximum Number across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result of the comparison is the numerical value, otherwise the result is identical to FMAX (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	0	1	1	0	0	0	0	1	1	0	0	1	0	Rn		Rd	
								o1																	

Half-precision variant

```
FMAXNMV <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	0	sz	1	1	0	0	0	0	1	1	0	0	1	0	Rn		Rd	
								o1																	

Single-precision and double-precision variant

```
FMAXNMV <V><d>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q != '01' then UNDEFINED;    // .4S only

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

<V> For the half-precision variant: is the destination width specifier, H.

For the single-precision and double-precision variant: is the destination width specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0

The encoding sz = 1 is reserved.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

4H when Q = 0

8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "Q:sz" field. It can have the following values:

4S when Q = 1, sz = 0

The following encodings are reserved:

- Q = 0, sz = x.
- Q = 1, sz = 1.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMAXNUM, operand, esize);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.108 FMAXP (scalar)

Floating-point Maximum of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the largest of the floating-point values as a scalar to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	0	0	1	1	0	0	0	0	1	1	1	1	1	0	Rn		Rd	
								o1	sz																

Half-precision variant

```
FMAXP <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = 32;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	0	sz	1	1	0	0	0	0	1	1	1	1	1	0	Rn		Rd	
								o1																	

Single-precision and double-precision variant

```
FMAXP <V><d>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32;
integer datasize = 64;
```

Assembler symbols

<V> For the half-precision variant: is the destination width specifier, encoded in the "sz" field. It can have the following values:

H when sz = 0

The encoding sz = 1 is reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the single-precision and double-precision variant: is the destination width specifier, encoded in the "sz" field. It can have the following values:

`S` when `sz = 0`

`D` when `sz = 1`

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<T>` For the half-precision variant: is the source arrangement specifier, encoded in the "sz" field. It can have the following values:

`2H` when `sz = 0`

The encoding `sz = 1` is reserved.

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in the "sz" field. It can have the following values:

`2S` when `sz = 0`

`2D` when `sz = 1`

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMAX, operand, esize);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.109 FMAXP (vector)

Floating-point Maximum Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, writes the larger of each pair of values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	0	1	0	Rm		0	0	1	1	0	1	Rn		Rd	
		U						o1														

Half-precision variant

```
FMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	0	sz	1	Rm		1	1	1	1	0	1	Rn		Rd	
		U						o1														

Single-precision and double-precision variant

```
FMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `4H` when `Q = 0`
- `8H` when `Q = 1`

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- `2S` when `sz = 0`, `Q = 0`
- `4S` when `sz = 0`, `Q = 1`
- `2D` when `sz = 1`, `Q = 1`

The encoding `sz = 1`, `Q = 0` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMin(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMax(element1, element2, FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.110 FMAXV

Floating-point Maximum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	0	1	1	0	0	0	0	1	1	1	1	1	0	Rn		Rd	
								o1																	

Half-precision variant

```
FMAXV <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	0	sz	1	1	0	0	0	0	1	1	1	1	1	0	Rn		Rd	
								o1																	

Single-precision and double-precision variant

```
FMAXV <V><d>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q != '01' then UNDEFINED;

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler symbols

<V> For the half-precision variant: is the destination width specifier, H.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the single-precision and double-precision variant: is the destination width specifier, encoded in the "sz" field. It can have the following values:

`S` when `sz = 0`

The encoding `sz = 1` is reserved.

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<T>` For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`4H` when `Q = 0`

`8H` when `Q = 1`

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "Q:sz" field. It can have the following values:

`4S` when `Q = 1`, `sz = 0`

The following encodings are reserved:

- `Q = 0`, `sz = x`.
- `Q = 1`, `sz = 1`.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMAX, operand, esize);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.111 FMIN (vector)

Floating-point minimum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the smaller of each of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	1	1	0	Rm		0	0	1	1	0	1	Rn		Rd	
		U						o1														

Half-precision variant

```
FMIN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	1	sz	1	Rm		1	1	1	1	0	1	Rn		Rd	
		U						o1														

Single-precision and double-precision variant

```
FMIN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`4H` when `Q = 0`

`8H` when `Q = 1`

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

`2S` when `sz = 0`, `Q = 0`

`4S` when `sz = 0`, `Q = 1`

`2D` when `sz = 1`, `Q = 1`

The encoding `sz = 1`, `Q = 0` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMin(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMax(element1, element2, FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.112 FMIN (scalar)

Floating-point Minimum (scalar). This instruction compares the first and second source SIMD&FP register values, and writes the smaller of the two floating-point values to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	0	0	1	1	1	1	0	ftype	1	Rm	0	1	0	1	1	0	Rn	Rd
													op					

Half-precision variant

Applies when `ftype == 11`.

```
FMIN <Hd>, <Hn>, <Hm>
```

Single-precision variant

Applies when `ftype == 00`.

```
FMIN <Sd>, <Sn>, <Sm>
```

Double-precision variant

Applies when `ftype == 01`.

```
FMIN <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;
```

Assembler symbols

`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Dn>`	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
`<Dm>`	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
`<Hd>`	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Hn>`	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Sn>` Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result = FPMin(operand1, operand2, FPCR);
V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.113 FMINNM (vector)

Floating-point Minimum Number (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, writes the smaller of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result placed in the vector is the numerical value, otherwise the result is identical to FMIN (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	1	1	0	Rm		0	0	0	0	0	1	Rn		Rd	
		U						a														

Half-precision variant

```
FMINNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	1	sz	1	Rm		1	1	0	0	0	1	Rn		Rd	
		U						o1														

Single-precision and double-precision variant

```
FMINNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- 4H when Q = 0
- 8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMinNum(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.114 FMINNM (scalar)

Floating-point Minimum Number (scalar). This instruction compares the first and second source SIMD&FP register values, and writes the smaller of the two floating-point values to the destination SIMD&FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result that is placed in the vector is the numerical value, otherwise the result is identical to FMIN (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	0	0	1	1	1	1	0	ftype	1	Rm	0	1	1	1	1	0	Rn	Rd
											op							

Half-precision variant

Applies when `ftype == 11`.

```
FMINNM <Hd>, <Hn>, <Hm>
```

Single-precision variant

Applies when `ftype == 00`.

```
FMINNM <Sd>, <Sn>, <Sm>
```

Double-precision variant

Applies when `ftype == 01`.

```
FMINNM <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;
```

Assembler symbols

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result = FPMinNum(operand1, operand2, FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.115 FMINNMP (scalar)

Floating-point Minimum Number of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the smallest of the floating-point values as a scalar to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	1	0	1	1	0	0	0	0	1	1	0	0	1	0	Rn		Rd	
								o1	sz																

Half-precision variant

```
FMINNMP <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = 32;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	1	sz	1	1	0	0	0	0	1	1	0	0	1	0	Rn		Rd	
								o1																	

Single-precision and double-precision variant

```
FMINNMP <V><d>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32;
integer datasize = 64;
```

Assembler symbols

<V> For the half-precision variant: is the destination width specifier, encoded in the "sz" field. It can have the following values:

H when sz = 0

The encoding sz = 1 is reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the single-precision and double-precision variant: is the destination width specifier, encoded in the "sz" field. It can have the following values:

`S` when `sz = 0`

`D` when `sz = 1`

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<T>` For the half-precision variant: is the source arrangement specifier, encoded in the "sz" field. It can have the following values:

`2H` when `sz = 0`

The encoding `sz = 1` is reserved.

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in the "sz" field. It can have the following values:

`2S` when `sz = 0`

`2D` when `sz = 1`

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMINNUM, operand, esize);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.116 FMINNMP (vector)

Floating-point Minimum Number Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the smallest of each pair of floating-point values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result is the numerical value, otherwise the result is identical to FMIN (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	1	1	0	Rm		0	0	0	0	0	1	Rn		Rd	
		U						a														

Half-precision variant

```
FMINNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	1	sz	1	Rm		1	1	0	0	0	1	Rn		Rd	
		U						o1														

Single-precision and double-precision variant

```
FMINNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- 4H when Q = 0
- 8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMinNum(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.117 FMINNMV

Floating-point Minimum Number across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result of the comparison is the numerical value, otherwise the result is identical to FMIN (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	1	0	1	1	0	0	0	0	1	1	0	0	1	0	Rn		Rd	
								o1																	

Half-precision variant

```
FMINNMV <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	1	sz	1	1	0	0	0	0	1	1	0	0	1	0	Rn		Rd	
								o1																	

Single-precision and double-precision variant

```
FMINNMV <V><d>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q != '01' then UNDEFINED;    // .4S only

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

<V> For the half-precision variant: is the destination width specifier, H.

For the single-precision and double-precision variant: is the destination width specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0

The encoding sz = 1 is reserved.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

4H when Q = 0

8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "Q:sz" field. It can have the following values:

4S when Q = 1, sz = 0

The following encodings are reserved:

- Q = 0, sz = x.
- Q = 1, sz = 1.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMINNUM, operand, esize);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.118 FMINP (scalar)

Floating-point Minimum of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the smallest of the floating-point values as a scalar to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	1	0	1	1	0	0	0	0	1	1	1	1	1	0	Rn		Rd	
								o1	sz																

Half-precision variant

```
FMINP <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = 32;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	1	sz	1	1	0	0	0	0	1	1	1	1	1	0	Rn		Rd	
								o1																	

Single-precision and double-precision variant

```
FMINP <V><d>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32;
integer datasize = 64;
```

Assembler symbols

<V> For the half-precision variant: is the destination width specifier, encoded in the "sz" field. It can have the following values:

H when sz = 0

The encoding sz = 1 is reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the single-precision and double-precision variant: is the destination width specifier, encoded in the "sz" field. It can have the following values:

`S` when `sz = 0`

`D` when `sz = 1`

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<T>` For the half-precision variant: is the source arrangement specifier, encoded in the "sz" field. It can have the following values:

`2H` when `sz = 0`

The encoding `sz = 1` is reserved.

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in the "sz" field. It can have the following values:

`2S` when `sz = 0`

`2D` when `sz = 1`

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMIN, operand, esize);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.119 FMINP (vector)

Floating-point Minimum Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, writes the smaller of each pair of values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	1	1	0	Rm		0	0	1	1	0	1	Rn		Rd	
		U						o1														

Half-precision variant

```
FMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	1	sz	1	Rm		1	1	1	1	0	1	Rn		Rd	
		U						o1														

Single-precision and double-precision variant

```
FMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- 4H when Q = 0
- 8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMin(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMax(element1, element2, FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.120 FMINV

Floating-point Minimum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	1	0	1	1	0	0	0	0	1	1	1	1	1	0	Rn		Rd	
								o1																	

Half-precision variant

```
FMINV <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	1	sz	1	1	0	0	0	0	1	1	1	1	1	0	Rn		Rd	
								o1																	

Single-precision and double-precision variant

```
FMINV <V><d>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q != '01' then UNDEFINED;

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler symbols

<V> For the half-precision variant: is the destination width specifier, H.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the single-precision and double-precision variant: is the destination width specifier, encoded in the "sz" field. It can have the following values:

`S` when `sz = 0`

The encoding `sz = 1` is reserved.

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<T>` For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`4H` when `Q = 0`

`8H` when `Q = 1`

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "Q:sz" field. It can have the following values:

`4S` when `Q = 1`, `sz = 0`

The following encodings are reserved:

- `Q = 0`, `sz = x`.
- `Q = 1`, `sz = 1`.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMIN, operand, esize);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.121 FMLA (by element)

Floating-point fused Multiply-Add to accumulator (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the results in the vector elements of the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar, half-precision

ARMv8.2

31 30	29	28	27 26 25 24	23 22	21	20	19 16	15	14	13 12	11	10	9 5	4 0
0 1	0	1	1 1 1 1	0 0	L	M	Rm	0	0	0 1	H	0	Rn	Rd
									o2					

Scalar, half-precision variant

```
FMLA <Hd>, <Hn>, <Vm>.H[<index>]
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');
```

Scalar, single-precision and double-precision

31 30	29	28	27 26 25 24	23	22	21	20	19 16	15	14	13 12	11	10	9 5	4 0
0 1	0	1	1 1 1 1	1	sz	L	M	Rm	0	0	0 1	H	0	Rn	Rd
										o2					

Scalar, single-precision and double-precision variant

```
FMLA <V><d>, <V><n>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');
```

Vector, half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	1	0	0	L	M	Rm	0	0	0	1	H	0	Rn	Rd
														o2						

Vector, half-precision variant

```
FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');
```

Vector, single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	1	1	sz	L	M	Rm	0	0	0	1	H	0	Rn	Rd
														o2						

Vector, single-precision and double-precision variant

```
FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');
```

Assembler symbols

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

`<V>` Is a width specifier, encoded in the "sz" field. It can have the following values:

- `S` when `sz = 0`
- `D` when `sz = 1`

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the vector, half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `4H` when `Q = 0`
- `8H` when `Q = 1`

For the vector, single-precision and double-precision variant: is an arrangement specifier, encoded in the "Q:sz" field. It can have the following values:

- `2S` when `Q = 0`, `sz = 0`
- `4S` when `Q = 1`, `sz = 0`
- `2D` when `Q = 1`, `sz = 1`

The encoding `Q = 0`, `sz = 1` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

`<Ts>` Is an element size specifier, encoded in the "sz" field. It can have the following values:

- `S` when `sz = 0`
- `D` when `sz = 1`

`<index>` For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

For the single-precision and double-precision variant: is the element index, encoded in the "sz:L:H" field. It can have the following values:

- `H:L` when `sz = 0`, `L = x`
- `H` when `sz = 1`, `L = 0`

The encoding `sz = 1`, `L = 1` is reserved.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, FPCR);
V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.122 FMLA (vector)

Floating-point fused Multiply-Add to accumulator (vector). This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD&FP registers, adds the product to the corresponding vector element of the destination SIMD&FP register, and writes the result to the destination SIMD&FP register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	1	0	Rm		0	0	0	0	1	1	Rn		Rd	
								a														

Half-precision variant

```
FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (a == '1');
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	sz	1	Rm		1	1	0	0	1	1	Rn		Rd	
								op														

Single-precision and double-precision variant

```
FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (op == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`4H` when `Q = 0`

`8H` when `Q = 1`

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

`2S` when `sz = 0`, `Q = 0`

`4S` when `sz = 0`, `Q = 1`

`2D` when `sz = 1`, `Q = 1`

The encoding `sz = 1`, `Q = 0` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.123 FMLAL, FMLAL2 (by element)

Floating-point fused Multiply-Add Long to accumulator (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_AA64ISAR0_EL1.FHM indicates whether this instruction is supported.

FMLAL

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11	10	9–5	4–0
0	Q	0	0	1	1	1	1	1	0	L	M	Rm	0	0	0	0	H	0	Rn	Rd
									sz					S						

FMLAL variant

```
FMLAL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[<index>]
```

Decode for this encoding

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt('0':Rm);    // Vm can only be in bottom 16 registers.
if sz == '1' then UNDEFINED;
integer index = UInt(H:L:M);

integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (S == '1');
integer part = 0;
```

FMLAL2

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11	10	9–5	4–0
0	Q	1	0	1	1	1	1	1	0	L	M	Rm	1	0	0	0	H	0	Rn	Rd
									sz					S						

FMLAL2 variant

```
FMLAL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[<index>]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt('0':Rm);    // Vm can only be in bottom 16 registers.
if sz == '1' then UNDEFINED;
integer index = UInt(H:L:M);

integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (S == '1');
integer part = 1;
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- 2S when Q = 0
- 4S when Q = 1

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- 2H when Q = 0
- 4H when Q = 1

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<index> Is the element index, encoded in the "H:L:M" fields.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize DIV 2) operand1 = Vpart[n, part];
bits(128) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2 = Elem[operand2, index, esize DIV 2];

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize DIV 2];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR);
V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.124 FMLAL, FMLAL2 (vector)

Floating-point fused Multiply-Add Long to accumulator (vector). This instruction multiplies corresponding half-precision floating-point values in the vectors in the two source SIMD&FP registers, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_AA64ISAR0_EL1.FHM indicates whether this instruction is supported.

FMLAL

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	0	0	1	Rm	1	1	1	0	1	1	Rn	Rd
								S	sz										

FMLAL variant

```
FMLAL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (S == '1');
integer part = 0;
```

FMLAL2

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	0	0	1	Rm	1	1	0	0	1	1	Rn	Rd
								S	sz										

FMLAL2 variant

```
FMLAL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (S == '1');
integer part = 1;
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- 2S when Q = 0
- 4S when Q = 1

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- 2H when Q = 0
- 4H when Q = 1

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize DIV 2) operand1 = Vpart[n, part];
bits(datasize DIV 2) operand2 = Vpart[m, part];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize DIV 2];
    element2 = Elem[operand2, e, esize DIV 2];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR);
V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.125 FMLS (by element)

Floating-point fused Multiply-Subtract from accumulator (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and subtracts the results from the vector elements of the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar, half-precision

ARMv8.2

31 30	29	28	27 26 25 24	23 22	21	20	19 16	15	14	13 12	11	10	9 5	4 0
0 1	0	1	1 1 1 1	0 0	L	M	Rm	0	1	0 1	H	0	Rn	Rd
									o2					

Scalar, half-precision variant

```
FMLS <Hd>, <Hn>, <Vm>.H[<index>]
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');
```

Scalar, single-precision and double-precision

31 30	29	28	27 26 25 24	23	22	21	20	19 16	15	14	13 12	11	10	9 5	4 0
0 1	0	1	1 1 1 1	1	sz	L	M	Rm	0	1	0 1	H	0	Rn	Rd
										o2					

Scalar, single-precision and double-precision variant

```
FMLS <V><d>, <V><n>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');
```

Vector, half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	1	0	0	L	M	Rm	0	1	0	1	H	0	Rn	Rd
													o2							

Vector, half-precision variant

```
FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');
```

Vector, single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	1	1	sz	L	M	Rm	0	1	0	1	H	0	Rn	Rd
													o2							

Vector, single-precision and double-precision variant

```
FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');
```

Assembler symbols

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

`<V>` Is a width specifier, encoded in the "sz" field. It can have the following values:

- `S` when `sz = 0`
- `D` when `sz = 1`

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the vector, half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `4H` when `Q = 0`
- `8H` when `Q = 1`

For the vector, single-precision and double-precision variant: is an arrangement specifier, encoded in the "Q:sz" field. It can have the following values:

- `2S` when `Q = 0`, `sz = 0`
- `4S` when `Q = 1`, `sz = 0`
- `2D` when `Q = 1`, `sz = 1`

The encoding `Q = 0`, `sz = 1` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

`<Ts>` Is an element size specifier, encoded in the "sz" field. It can have the following values:

- `S` when `sz = 0`
- `D` when `sz = 1`

`<index>` For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

For the single-precision and double-precision variant: is the element index, encoded in the "sz:L:H" field. It can have the following values:

- `H:L` when `sz = 0`, `L = x`
- `H` when `sz = 1`, `L = 0`

The encoding `sz = 1`, `L = 1` is reserved.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, FPCR);
V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.126 FMLS (vector)

Floating-point fused Multiply-Subtract from accumulator (vector). This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD&FP registers, negates the product, adds the result to the corresponding vector element of the destination SIMD&FP register, and writes the result to the destination SIMD&FP register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	1	1	0	Rm		0	0	0	0	1	1	Rn		Rd	
								a														

Half-precision variant

```
FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (a == '1');
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	1	sz	1	Rm		1	1	0	0	1	1	Rn		Rd	
								op														

Single-precision and double-precision variant

```
FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (op == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`4H` when `Q = 0`

`8H` when `Q = 1`

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

`2S` when `sz = 0`, `Q = 0`

`4S` when `sz = 0`, `Q = 1`

`2D` when `sz = 1`, `Q = 1`

The encoding `sz = 1`, `Q = 0` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.127 FMLSL, FMLSL2 (by element)

Floating-point fused Multiply-Subtract Long from accumulator (by element). This instruction multiplies the negated vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_AA64ISAR0_EL1.FHM indicates whether this instruction is supported.

FMLSL

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11	10	9–5	4–0
0	Q	0	0	1	1	1	1	1	0	L	M	Rm	0	1	0	0	H	0	Rn	Rd
									sz					S						

FMLSL variant

```
FMLSL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[<index>]
```

Decode for this encoding

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt('0':Rm);    // Vm can only be in bottom 16 registers.
if sz == '1' then UNDEFINED;
integer index = UInt(H:L:M);

integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (S == '1');
integer part = 0;
```

FMLSL2

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11	10	9–5	4–0
0	Q	1	0	1	1	1	1	1	0	L	M	Rm	1	1	0	0	H	0	Rn	Rd
									sz					S						

FMLSL2 variant

```
FMLSL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[<index>]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt('0':Rm);    // Vm can only be in bottom 16 registers.
if sz == '1' then UNDEFINED;
integer index = UInt(H:L:M);

integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (S == '1');
integer part = 1;
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- 2S when Q = 0
- 4S when Q = 1

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- 2H when Q = 0
- 4H when Q = 1

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<index> Is the element index, encoded in the "H:L:M" fields.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize DIV 2) operand1 = Vpart[n, part];
bits(128) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2 = Elem[operand2, index, esize DIV 2];

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize DIV 2];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR);
V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.128 FMLSL, FMLSL2 (vector)

Floating-point fused Multiply-Subtract Long from accumulator (vector). This instruction negates the values in the vector of one SIMD&FP register, multiplies these with the corresponding values in another vector, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_AA64ISAR0_EL1.FHM indicates whether this instruction is supported.

FMLSL

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	1	0	1	Rm	1	1	1	0	1	1	Rn	Rd
								S	sz										

FMLSL variant

```
FMLSL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (S == '1');
integer part = 0;
```

FMLSL2

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	1	0	1	Rm	1	1	0	0	1	1	Rn	Rd
								S	sz										

FMLSL2 variant

```
FMLSL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (S == '1');
integer part = 1;
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- 2S when Q = 0
- 4S when Q = 1

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- 2H when Q = 0
- 4H when Q = 1

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize DIV 2) operand1 = Vpart[n, part];
bits(datasize DIV 2) operand2 = Vpart[m, part];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize DIV 2];
    element2 = Elem[operand2, e, esize DIV 2];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR);
V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.129 FMOV (vector, immediate)

Floating-point move immediate (vector). This instruction copies an immediate floating-point constant into every element of the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	0
0	Q	0	0	1	1	1	1	0	0	0	0	0	a	b	c	1	1	1	1	1	1	d	e	f	g	h	Rd	

Half-precision variant

```
FMOV <Vd>.<T>, #<imm>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;

imm8 = a:b:c:d:e:f:g:h;
imm16 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>, 2):imm8<5:0>:Zeros(6);

imm = Replicate(imm16, datasize DIV 16);
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	0
0	Q	op	0	1	1	1	1	0	0	0	0	0	a	b	c	1	1	1	1	0	1	d	e	f	g	h	Rd	
																cmode												

Single-precision variant

Applies when `op == 0`.

```
FMOV <Vd>.<T>, #<imm>
```

Double-precision variant

Applies when `Q == 1 && op == 1`.

```
FMOV <Vd>.2D, #<imm>
```

Decode for all variants of this encoding

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

if cmode:op == '11111' then
    // FMOV Dn,#imm is in main FP instruction set
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if Q == '0' then UNDEFINED;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `4H` when `Q = 0`
- `8H` when `Q = 1`

For the single-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `2S` when `Q = 0`
- `4S` when `Q = 1`

`<imm>` Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in "a:b:c:d:e:f:g:h". For details of the range of constants available and the encoding of `<imm>`, see *Modified immediate constants in A64 floating-point instructions* on page C2-198.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

V[rd] = imm;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.130 FMOV (register)

Floating-point Move register without conversion. This instruction copies the floating-point value in the SIMD&FP source register to the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 19 18 17	16	15	14 13 12 11 10	9 5	4 0
0	0	0	1	1	1	1	0	ftype	1	0 0 0 0	0	0	1 0 0 0 0	Rn	Rd
											opc				

Half-precision variant

Applies when `ftype == 11`.

```
FMOV <Hd>, <Hn>
```

Single-precision variant

Applies when `ftype == 00`.

```
FMOV <Sd>, <Sn>
```

Double-precision variant

Applies when `ftype == 01`.

```
FMOV <Dd>, <Dn>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;
```

Assembler symbols

`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Dn>`	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
`<Hd>`	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Hn>`	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
`<Sd>`	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Sn>`	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];

V[d] = operand;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.131 FMOV (general)

Floating-point Move to or from general-purpose register without conversion. This instruction transfers the contents of a SIMD&FP register to a general-purpose register, or the contents of a general-purpose register to a SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	16	15	14	13	12	11	10	9	5	4	0
sf	0	0	1	1	1	1	0	ftype	1	0	x	1	1	x	0	0	0	0	0	0	Rn		Rd
										rmode		opcode											

Half-precision to 32-bit variant

Applies when `sf == 0 && ftype == 11 && rmode == 00 && opcode == 110`.

```
FMOV <Wd>, <Hn>
```

Half-precision to 64-bit variant

Applies when `sf == 1 && ftype == 11 && rmode == 00 && opcode == 110`.

```
FMOV <Xd>, <Hn>
```

32-bit to half-precision variant

Applies when `sf == 0 && ftype == 11 && rmode == 00 && opcode == 111`.

```
FMOV <Hd>, <Wn>
```

32-bit to single-precision variant

Applies when `sf == 0 && ftype == 00 && rmode == 00 && opcode == 111`.

```
FMOV <Sd>, <Wn>
```

Single-precision to 32-bit variant

Applies when `sf == 0 && ftype == 00 && rmode == 00 && opcode == 110`.

```
FMOV <Wd>, <Sn>
```

64-bit to half-precision variant

Applies when `sf == 1 && ftype == 11 && rmode == 00 && opcode == 111`.

```
FMOV <Hd>, <Xn>
```

64-bit to double-precision variant

Applies when `sf == 1 && ftype == 01 && rmode == 00 && opcode == 111`.

```
FMOV <Dd>, <Xn>
```

64-bit to top half of 128-bit variant

Applies when `sf == 1 && ftype == 10 && rmode == 01 && opcode == 111`.

```
FMOV <Vd>.D[1], <Xn>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Double-precision to 64-bit variant

Applies when `sf == 1 && ftype == 01 && rmode == 00 && opcode == 110`.

```
FMOV <Xd>, <Dn>
```

Top half of 128-bit to 64-bit variant

Applies when `sf == 1 && ftype == 10 && rmode == 01 && opcode == 110`.

```
FMOV <Xd>, <Vn>.D[1]
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        if opcode<2:1>:rmode != '11 01' then UNDEFINED;
        fltsize = 128;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;

case opcode<2:1>:rmode of
    when '00 xx'    // FCVT[NPMZ][US]
        rounding = FPDecodeRounding(rmode);
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '01 00'    // [US]CVTF
        rounding = FPRoundingMode(FPCR);
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_ItoF;
    when '10 00'    // FCVTA[US]
        rounding = FPRounding_TIEAWAY;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '11 00'    // FMOV
        if fltsize != 16 && fltsize != intsize then UNDEFINED;
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 0;
    when '11 01'    // FMOV D[1]
        if intsize != 64 || fltsize != 128 then UNDEFINED;
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 1;
        fltsize = 64;    // size of D[1] is 64
    when '11 11'    // FJCVTZS
        if !HaveFJCVTZSExt() then UNDEFINED;
        rounding = FPRounding_ZERO;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI_JS;
    otherwise
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Hd>`	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Sd>`	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
`<Vd>`	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Sn>`	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Vn>`	Is the name of the SIMD&FP source register, encoded in the "Rn" field.
`<Hn>`	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
`<Dn>`	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
    when FPConvOp_CVT_FtoI
        fltval = V[n];
        intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
        X[d] = intval;
    when FPConvOp_CVT_ItoF
        intval = X[n];
        fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
        V[d] = fltval;
    when FPConvOp_MOV_FtoI
        fltval = Vpart[n, part];
        intval = ZeroExtend(fltval, intsize);
        X[d] = intval;
    when FPConvOp_MOV_ItoF
        intval = X[n];
        fltval = intval<fltsize-1:0>;
        Vpart[d, part] = fltval;
    when FPConvOp_CVT_FtoI_JS
        bit Z;
        fltval = V[n];
        (intval, Z) = FPToFixedJS(fltval, FPCR, TRUE);
        PSTATE.<N,Z,C,V> = '0':Z:'00';
        X[d] = intval;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.132 FMOV (scalar, immediate)

Floating-point move immediate (scalar). This instruction copies a floating-point immediate constant into the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 13	12	11	10	9	8	7	6	5	4	0
0	0	0	1	1	1	1	0	ftype	1	imm8	1	0	0	0	0	0	0	0	Rd	

Half-precision variant

Applies when `ftype == 11`.

```
FMOV <Hd>, #<imm>
```

Single-precision variant

Applies when `ftype == 00`.

```
FMOV <Sd>, #<imm>
```

Double-precision variant

Applies when `ftype == 01`.

```
FMOV <Dd>, #<imm>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;

bits(datasize) imm = VFPExpandImm(imm8);
```

Assembler symbols

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<imm>` Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in the "imm8" field. For details of the range of constants available and the encoding of `<imm>`, see *Modified immediate constants in A64 floating-point instructions* on page C2-198.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
CheckFPAdvSIMDEnabled64();

V[d] = imm;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.133 FMSUB

Floating-point Fused Multiply-Subtract (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, negates the product, adds that to the value of the third SIMD&FP source register, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14 … 10	9 … 5	4 … 0
0	0	0	1	1	1	1	1	ftype	0	Rm	1	Ra	Rn	Rd
									o1		o0			

Half-precision variant

Applies when `ftype == 11`.

```
FMSUB <Hd>, <Hn>, <Hm>, <Ha>
```

Single-precision variant

Applies when `ftype == 00`.

```
FMSUB <Sd>, <Sn>, <Sm>, <Sa>
```

Double-precision variant

Applies when `ftype == 01`.

```
FMSUB <Dd>, <Dn>, <Dm>, <Da>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;
```

Assembler symbols

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Dm>` Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

`<Da>` Is the 64-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

`<Hm>` Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

`<Ha>` Is the 16-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Sn>` Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

`<Sm>` Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

`<Sa>` Is the 32-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operanda = V[a];
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

operand1 = FPNeg(operand1);
result = FPMulAdd(operanda, operand1, operand2, FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.134 FMUL (by element)

Floating-point Multiply (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar, half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11	10	9–5	4–0
0	1	0	1	1	1	1	1	0	0	L	M	Rm	1	0	0	1	H	0	Rn	Rd
		U																		

Scalar, half-precision variant

```
FMUL <Hd>, <Hn>, <Vm>.H[<index>]
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');
```

Scalar, single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11	10	9–5	4–0
0	1	0	1	1	1	1	1	1	sz	L	M	Rm	1	0	0	1	H	0	Rn	Rd
		U																		

Scalar, single-precision and double-precision variant

```
FMUL <V><d>, <V><n>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UNDEFINED;

integer d = UInt(Rd);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');
```

Vector, half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11	10	9–5	4–0
0	Q	0	0	1	1	1	1	0	0	L	M	Rm	1	0	0	1	H	0	Rn	Rd
		U																		

Vector, half-precision variant

```
FMUL <Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');
```

Vector, single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11	10	9–5	4–0
0	Q	0	0	1	1	1	1	1	sz	L	M	Rm	1	0	0	1	H	0	Rn	Rd
		U																		

Vector, single-precision and double-precision variant

```
FMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');
```

Assembler symbols

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

`<V>` Is a width specifier, encoded in the "sz" field. It can have the following values:

- `S` when `sz = 0`
- `D` when `sz = 1`

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the vector, half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `4H` when `Q = 0`
- `8H` when `Q = 1`

For the vector, single-precision and double-precision variant: is an arrangement specifier, encoded in the "Q:sz" field. It can have the following values:

- `2S` when `Q = 0`, `sz = 0`
- `4S` when `Q = 1`, `sz = 0`
- `2D` when `Q = 1`, `sz = 1`

The encoding `Q = 0`, `sz = 1` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

`<Ts>` Is an element size specifier, encoded in the "sz" field. It can have the following values:

- `S` when `sz = 0`
- `D` when `sz = 1`

`<index>` For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

For the single-precision and double-precision variant: is the element index, encoded in the "sz:L:H" field. It can have the following values:

- `H:L` when `sz = 0`, `L = x`
- `H` when `sz = 1`, `L = 0`

The encoding `sz = 1`, `L = 1` is reserved.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(esize) element2 = Elem[operand2, index, esize];

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if mulx_op then
        Elem[result, e, esize] = FPMulX(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMul(element1, element2, FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.135 FMUL (vector)

Floating-point Multiply (vector). This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD&FP registers, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	0	1	0	Rm		0	0	0	1	1	1	Rn		Rd	

Half-precision variant

```
FMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	0	sz	1	Rm		1	1	0	1	1	1	Rn		Rd	

Single-precision and double-precision variant

```
FMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

4H when Q = 0

8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0

4S when sz = 0, Q = 1

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMul(element1, element2, FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.136 FMUL (scalar)

Floating-point Multiply (scalar). This instruction multiplies the floating-point values of the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 16	15	14 13 12	11 10	9 5	4 0
0	0	0	1	1	1	1	0	ftype	1	Rm	0	0 0 0	1 0	Rn	Rd
											op				

Half-precision variant

Applies when `ftype == 11`.

```
FMUL <Hd>, <Hn>, <Hm>
```

Single-precision variant

Applies when `ftype == 00`.

```
FMUL <Sd>, <Sn>, <Sm>
```

Double-precision variant

Applies when `ftype == 01`.

```
FMUL <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;
```

Assembler symbols

`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Dn>`	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
`<Dm>`	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
`<Hd>`	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Hn>`	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Sn>` Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result = FPMul(operand1, operand2, FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.137 FMULX (by element)

Floating-point Multiply extended (by element). This instruction multiplies the floating-point values in the vector elements in the first source SIMD&FP register by the specified floating-point value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If one value is zero and the other value is infinite, the result is 2.0. In this case, the result is negative if only one of the values is negative, otherwise the result is positive.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar, half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11	10	9–5	4–0
0	1	1	1	1	1	1	1	0	0	L	M	Rm	1	0	0	1	H	0	Rn	Rd
		U																		

Scalar, half-precision variant

```
FMULX <Hd>, <Hn>, <Vm>.H[<index>]
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');
```

Scalar, single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11	10	9–5	4–0
0	1	1	1	1	1	1	1	1	sz	L	M	Rm	1	0	0	1	H	0	Rn	Rd
		U																		

Scalar, single-precision and double-precision variant

```
FMULX <V><d>, <V><n>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');
```

Vector, half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11	10	9–5	4–0
0	Q	1	0	1	1	1	1	0	0	L	M	Rm	1	0	0	1	H	0	Rn	Rd
		U																		

Vector, half-precision variant

```
FMULX <Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');
```

Vector, single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11	10	9–5	4–0
0	Q	1	0	1	1	1	1	1	sz	L	M	Rm	1	0	0	1	H	0	Rn	Rd
		U																		

Vector, single-precision and double-precision variant

```
FMULX <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');
```

Assembler symbols

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

`<V>` Is a width specifier, encoded in the "sz" field. It can have the following values:

- `S` when `sz = 0`
- `D` when `sz = 1`

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the vector, half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `4H` when `Q = 0`
- `8H` when `Q = 1`

For the vector, single-precision and double-precision variant: is an arrangement specifier, encoded in the "Q:sz" field. It can have the following values:

- `2S` when `Q = 0`, `sz = 0`
- `4S` when `Q = 1`, `sz = 0`
- `2D` when `Q = 1`, `sz = 1`

The encoding `Q = 0`, `sz = 1` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

`<Ts>` Is an element size specifier, encoded in the "sz" field. It can have the following values:

- `S` when `sz = 0`
- `D` when `sz = 1`

`<index>` For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

For the single-precision and double-precision variant: is the element index, encoded in the "sz:L:H" field. It can have the following values:

- `H:L` when `sz = 0`, `L = x`
- `H` when `sz = 1`, `L = 0`

The encoding `sz = 1`, `L = 1` is reserved.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(datasize) result;
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if mulx_op then
        Elem[result, e, esize] = FPMulX(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMul(element1, element2, FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.138 FMULX

Floating-point Multiply extended. This instruction multiplies corresponding floating-point values in the vectors of the two source SIMD&FP registers, places the resulting floating-point values in a vector, and writes the vector to the destination SIMD&FP register.

If one value is zero and the other value is infinite, the result is 2.0. In this case, the result is negative if only one of the values is negative, otherwise the result is positive.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

ARMv8.2

31 30 29 28	27 26 25 24	23 22 21 20	16	15 14 13 12	11 10 9	5 4	0
0 1 0	1 1 1 1 0	0 1 0	Rm	0 0 0 1 1	1	Rn	Rd

Scalar half precision variant

```
FMULX <Hd>, <Hn>, <Hm>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
```

Scalar single-precision and double-precision

31 30 29 28	27 26 25 24	23 22 21 20	16	15 14 13 12	11 10 9	5 4	0
0 1 0	1 1 1 1 0	0 sz 1	Rm	1 1 0 1 1	1	Rn	Rd

Scalar single-precision and double-precision variant

```
FMULX <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```

Vector half precision

ARMv8.2

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	1	0	Rm		0	0	0	1	1	1	Rn		Rd	

Vector half precision variant

```
FMULX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	sz	1	Rm		1	1	0	1	1	1	Rn		Rd	

Vector single-precision and double-precision variant

```
FMULX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler symbols

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

`<V>` Is a width specifier, encoded in the "sz" field. It can have the following values:

- `S` when `sz = 0`
- `D` when `sz = 1`

`<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<m>` Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<T> For the vector half precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

4H when Q = 0

8H when Q = 1

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0

4S when sz = 0, Q = 1

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMulX(element1, element2, FPCR);
V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.139 FNEG (vector)

Floating-point Negate (vector). This instruction negates the value of each vector element in the source SIMD&FP register, writes the result to a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	1	1	1	1	1	0	0	0	1	1	1	1	1	0	Rn		Rd	
		U																							

Half-precision variant

```
FNEG <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	1	1	1	0	Rn		Rd	
		U																							

Single-precision and double-precision variant

```
FNEG <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`4H` when `Q = 0`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`8H` when `Q = 1`

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

`2S` when `sz = 0`, `Q = 0`

`4S` when `sz = 0`, `Q = 1`

`2D` when `sz = 1`, `Q = 1`

The encoding `sz = 1`, `Q = 0` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    if neg then
        element = FPNeg(element);
    else
        element = FPAbs(element);
    Elem[result, e, esize] = element;

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.140 FNEG (scalar)

Floating-point Negate (scalar). This instruction negates the value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 19 18 17	16	15	14 13 12 11 10	9 5	4 0
0	0	0	1	1	1	1	0	ftype	1	0 0 0 0	1	0	1 0 0 0 0	Rn	Rd
											opc				

Half-precision variant

Applies when `ftype == 11`.

```
FNEG <Hd>, <Hn>
```

Single-precision variant

Applies when `ftype == 00`.

```
FNEG <Sd>, <Sn>
```

Double-precision variant

Applies when `ftype == 01`.

```
FNEG <Dd>, <Dn>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;
```

Assembler symbols

`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Dn>`	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
`<Hd>`	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Hn>`	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
`<Sd>`	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Sn>`	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPNeg(operand);
V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.141 FNMADD

Floating-point Negated fused Multiply-Add (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, negates the product, subtracts the value of the third SIMD&FP source register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14 … 10	9 … 5	4 … 0
0	0	0	1	1	1	1	1	ftype	1	Rm	0	Ra	Rn	Rd
									o1		o0			

Half-precision variant

Applies when `ftype == 11`.

```
FNMADD <Hd>, <Hn>, <Hm>, <Ha>
```

Single-precision variant

Applies when `ftype == 00`.

```
FNMADD <Sd>, <Sn>, <Sm>, <Sa>
```

Double-precision variant

Applies when `ftype == 01`.

```
FNMADD <Dd>, <Dn>, <Dm>, <Da>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;
```

Assembler symbols

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Dm>` Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

`<Da>` Is the 64-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

`<Hm>` Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

`<Ha>` Is the 16-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Sn>` Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

`<Sm>` Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

`<Sa>` Is the 32-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operanda = V[a];
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

operanda = FPNeg(operanda);
operand1 = FPNeg(operand1);
result = FPMulAdd(operanda, operand1, operand2, FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.142 FNMSUB

Floating-point Negated fused Multiply-Subtract (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, subtracts the value of the third SIMD&FP source register, and writes the result to the destination SIMD&FP register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14 … 10	9 … 5	4 … 0
0	0	0	1	1	1	1	1	ftype	1	Rm	1	Ra	Rn	Rd
									o1		o0			

Half-precision variant

Applies when `ftype == 11`.

```
FNMSUB <Hd>, <Hn>, <Hm>, <Ha>
```

Single-precision variant

Applies when `ftype == 00`.

```
FNMSUB <Sd>, <Sn>, <Sm>, <Sa>
```

Double-precision variant

Applies when `ftype == 01`.

```
FNMSUB <Dd>, <Dn>, <Dm>, <Da>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;
```

Assembler symbols

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Dm>` Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

`<Da>` Is the 64-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

`<Hm>` Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

`<Ha>` Is the 16-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Sn>` Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

`<Sm>` Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

`<Sa>` Is the 32-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operanda = V[a];
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

operanda = FPNeg(operanda);
result = FPMulAdd(operanda, operand1, operand2, FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.143 FNMUL (scalar)

Floating-point Multiply-Negate (scalar). This instruction multiplies the floating-point values of the two source SIMD&FP registers, and writes the negation of the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	0	0	1	1	1	1	0	ftype	1	Rm	1	0	0	0	1	0	Rn	Rd
											op							

Half-precision variant

Applies when `ftype == 11`.

```
FNMUL <Hd>, <Hn>, <Hm>
```

Single-precision variant

Applies when `ftype == 00`.

```
FNMUL <Sd>, <Sn>, <Sm>
```

Double-precision variant

Applies when `ftype == 01`.

```
FNMUL <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;
```

Assembler symbols

`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Dn>`	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
`<Dm>`	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
`<Hd>`	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Hn>`	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Sn>` Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result = FPMul(operand1, operand2, FPCR);

result = FPNeg(result);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.144 FRECPE

Floating-point Reciprocal Estimate. This instruction finds an approximate reciprocal estimate for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn		Rd	

Scalar half precision variant

```
FRECPE <Hd>, <Hn>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	1	0	1	1	0	Rn		Rd	

Scalar single-precision and double-precision variant

```
FRECPE <V><d>, <V><n>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```

Vector half precision

ARMv8.2

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	1	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn		Rd	

Vector half precision variant

```
FRECPE <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	1	0	1	1	0	Rn		Rd	

Vector single-precision and double-precision variant

```
FRECPE <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler symbols

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

`<V>` Is a width specifier, encoded in the "sz" field. It can have the following values:

- `S` when `sz = 0`
- `D` when `sz = 1`

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

`<n>` Is the number of the SIMD&FP source register, encoded in the "Rn" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the vector half precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `4H` when `Q = 0`
- `8H` when `Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

`2S` when `sz = 0, Q = 0`

`4S` when `sz = 0, Q = 1`

`2D` when `sz = 1, Q = 1`

The encoding `sz = 1, Q = 0` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRecipEstimate(element, FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.145 FRECPS

Floating-point Reciprocal Step. This instruction multiplies the corresponding floating-point values in the vectors of the two source SIMD&FP registers, subtracts each of the products from 2.0, places the resulting floating-point values in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	0	1	0	Rm		0	0	1	1	1	1	Rn		Rd	

Scalar half precision variant

```
FRECPS <Hd>, <Hn>, <Hm>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	0	sz	1	Rm		1	1	1	1	1	1	Rn		Rd	

Scalar single-precision and double-precision variant

```
FRECPS <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```

Vector half precision

ARMv8.2

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	1	0	Rm		0	0	1	1	1	1	Rn		Rd	

Vector half precision variant

```
FRECPS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	sz	1	Rm		1	1	1	1	1	1	Rn		Rd	

Vector single-precision and double-precision variant

```
FRECPS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler symbols

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

`<V>` Is a width specifier, encoded in the "sz" field. It can have the following values:

- `S` when `sz = 0`
- `D` when `sz = 1`

`<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<m>` Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<T> For the vector half precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

4H when Q = 0

8H when Q = 1

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0

4S when sz = 0, Q = 1

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPRecipStepFused(element1, element2);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.146 FRECPX

Floating-point Reciprocal exponent (scalar). This instruction finds an approximate reciprocal exponent for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	1	1	1	1	1	1	0	Rn		Rd	

Half-precision variant

```
FRECPX <Hd>, <Hn>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	1	1	1	1	0	Rn		Rd	

Single-precision and double-precision variant

```
FRECPX <V><d>, <V><n>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```

Assembler symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0

D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRecpX(element, FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.147 FRINT32X (vector)

Floating-point Round to 32-bit Integer, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 32-bit integer size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	1	1	0	1	0	Rn		Rd	
		U																	op						

Vector single-precision and double-precision variant

```
FRINT32X <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer intsize = if op == '0' then 32 else 64;
FPRounding rounding = if U == '0' then FPRounding_ZERO else FPRoundingMode(FPCR);
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- `2S` when `sz = 0`, `Q = 0`
- `4S` when `sz = 0`, `Q = 1`
- `2D` when `sz = 1`, `Q = 1`

The encoding `sz = 1`, `Q = 0` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundIntN(element, FPCR, rounding, intsize);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.148 FRINT32X (scalar)

Floating-point Round to 32-bit Integer, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 32-bit integer size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns {for the corresponding result value} the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	0	0	1	1	1	1	0	0	x	1	0	1	0	0	0	1	1	0	0	0	0	Rn		Rd	
								ftype								op									

Single-precision variant

Applies when `ftype == 00`.

```
FRINT32X <Sd>, <Sn>
```

Double-precision variant

Applies when `ftype == 01`.

```
FRINT32X <Dd>, <Dn>
```

Decode for all variants of this encoding

```
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UNDEFINED;

FPRounding rounding = FPRoundingMode(FPCR);
```

Assembler symbols

`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Dn>`	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
`<Sd>`	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Sn>`	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundIntN(operand, FPCR, rounding, 32);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.149 FRINT32Z (vector)

Floating-point Round to 32-bit Integer toward Zero (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 32-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	1	1	0	1	0	Rn		Rd	
		U																	op						

Vector single-precision and double-precision variant

```
FRINT32Z <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer intsize = if op == '0' then 32 else 64;
FPRounding rounding = if U == '0' then FPRounding_ZERO else FPRoundingMode(FPCR);
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundIntN(element, FPCR, rounding, intsize);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.150 FRINT32Z (scalar)

Floating-point Round to 32-bit Integer toward Zero (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 32-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the {corresponding} input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns {for the corresponding result value} the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

ARMv8.5

31 30 29 28	27 26 25 24	23 22	21	20 19 18 17	16 15	14 13 12 11 10	9 5	4 0
0 0 0	1 1 1 1 0	0 x	1	0 1 0 0	0 0	1 0 0 0 0	Rn	Rd
		ftype			op			

Single-precision variant

Applies when `ftype == 00`.

```
FRINT32Z <Sd>, <Sn>
```

Double-precision variant

Applies when `ftype == 01`.

```
FRINT32Z <Dd>, <Dn>
```

Decode for all variants of this encoding

```
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UNDEFINED;
```

Assembler symbols

`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Dn>`	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
`<Sd>`	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Sn>`	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundIntN(operand, FPCR, FPRounding_ZERO, 32);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.151 FRINT64X (vector)

Floating-point Round to 64-bit Integer, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 64-bit integer size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	1	1	1	1	0	Rn		Rd	
		U																	op						

Vector single-precision and double-precision variant

```
FRINT64X <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer intsize = if op == '0' then 32 else 64;
FPRounding rounding = if U == '0' then FPRounding_ZERO else FPRoundingMode(FPCR);
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- `2S` when `sz = 0`, `Q = 0`
- `4S` when `sz = 0`, `Q = 1`
- `2D` when `sz = 1`, `Q = 1`

The encoding `sz = 1`, `Q = 0` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundIntN(element, FPCR, rounding, intsize);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.152 FRINT64X (scalar)

Floating-point Round to 64-bit Integer, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 64-bit integer size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns {for the corresponding result value} the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	0	0	1	1	1	1	0	0	x	1	0	1	0	0	1	1	1	0	0	0	0	Rn		Rd	
								ftype							op										

Single-precision variant

Applies when `ftype == 00`.

```
FRINT64X <Sd>, <Sn>
```

Double-precision variant

Applies when `ftype == 01`.

```
FRINT64X <Dd>, <Dn>
```

Decode for all variants of this encoding

```
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UNDEFINED;

FPRounding rounding = FPRoundingMode(FPCR);
```

Assembler symbols

`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Dn>`	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
`<Sd>`	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Sn>`	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundIntN(operand, FPCR, rounding, 64);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.153 FRINT64Z (vector)

Floating-point Round to 64-bit Integer toward Zero (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 64-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	1	1	1	1	0	Rn		Rd	
		U																	op						

Vector single-precision and double-precision variant

```
FRINT64Z <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer intsize = if op == '0' then 32 else 64;
FPRounding rounding = if U == '0' then FPRounding_ZERO else FPRoundingMode(FPCR);
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundIntN(element, FPCR, rounding, intsize);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.154 FRINT64Z (scalar)

Floating-point Round to 64-bit Integer toward Zero (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 64-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the {corresponding} input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns {for the corresponding result value} the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

ARMv8.5

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	0	0	1	1	1	1	0	0	x	1	0	1	0	0	1	0	1	0	0	0	0	Rn		Rd	
								ftype							op										

Single-precision variant

Applies when `ftype == 00`.

```
FRINT64Z <Sd>, <Sn>
```

Double-precision variant

Applies when `ftype == 01`.

```
FRINT64Z <Dd>, <Dn>
```

Decode for all variants of this encoding

```
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UNDEFINED;
```

Assembler symbols

`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Dn>`	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
`<Sd>`	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Sn>`	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundIntN(operand, FPCR, FPRounding_ZERO, 64);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.155 FRINTA (vector)

Floating-point Round to Integral, to nearest with ties to Away (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round to Nearest with Ties to Away rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	0	1	1	1	1	0	0	1	1	0	0	0	1	0	Rn		Rd	
		U						o2											o1						

Half-precision variant

```
FRINTA <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	0	0	1	0	Rn		Rd	
		U						o2											o1						

Single-precision and double-precision variant

```
FRINTA <Vd>.<T>, <Vn>.<T>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- 4H when Q = 0
- 8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.156 FRINTA (scalar)

Floating-point Round to Integral, to nearest with ties to Away (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round to Nearest with Ties to Away rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 19	18	17	15	14	13	12	11	10	9 5	4 0
0	0	0	1	1	1	1	0	ftype	1	0	0	1	1	0	0	1	0	0	Rn	Rd
												rmode								

Half-precision variant

Applies when `ftype == 11`.

```
FRINTA <Hd>, <Hn>
```

Single-precision variant

Applies when `ftype == 00`.

```
FRINTA <Sd>, <Sn>
```

Double-precision variant

Applies when `ftype == 01`.

```
FRINTA <Dd>, <Dn>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;
```

Assembler symbols

`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Dn>`	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
`<Hd>`	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Sn>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, FPRounding_TIEAWAY, FALSE);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.157 FRINTI (vector)

Floating-point Round to Integral, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	1	1	1	1	1	0	0	1	1	0	0	1	1	0	Rn		Rd	
		U						o2											o1						

Half-precision variant

```
FRINTI <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	0	1	1	0	Rn		Rd	
		U						o2											o1						

Single-precision and double-precision variant

```
FRINTI <Vd>.<T>, <Vn>.<T>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- 4H when Q = 0
- 8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.158 FRINTI (scalar)

Floating-point Round to Integral, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	17	15	14	13	12	11	10	9 5	4 0
0	0	0	1	1	1	1	0	ftype	1	0	0	1	1	1	1	0	0	0	0	Rn	Rd
													rmode								

Half-precision variant

Applies when `ftype == 11`.

```
FRINTI <Hd>, <Hn>
```

Single-precision variant

Applies when `ftype == 00`.

```
FRINTI <Sd>, <Sn>
```

Double-precision variant

Applies when `ftype == 01`.

```
FRINTI <Dd>, <Dn>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;

FPRounding rounding;
rounding = FPRoundingMode(FPCR);
```

Assembler symbols

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Dn>` Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Sn>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, FALSE);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.159 FRINTM (vector)

Floating-point Round to Integral, toward Minus infinity (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	1	1	1	1	0	0	1	1	0	0	1	1	0	Rn		Rd	
		U						o2											o1						

Half-precision variant

```
FRINTM <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	0	1	1	0	Rn		Rd	
		U						o2											o1						

Single-precision and double-precision variant

```
FRINTM <Vd>.<T>, <Vn>.<T>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- 4H when Q = 0
- 8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.160 FRINTM (scalar)

Floating-point Round to Integral, toward Minus infinity (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	17	15	14	13	12	11	10	9 5	4 0
0	0	0	1	1	1	1	0	ftype	1	0	0	1	0	1	0	1	0	0	0	Rn	Rd
													rmode								

Half-precision variant

Applies when `ftype == 11`.

```
FRINTM <Hd>, <Hn>
```

Single-precision variant

Applies when `ftype == 00`.

```
FRINTM <Sd>, <Sn>
```

Double-precision variant

Applies when `ftype == 01`.

```
FRINTM <Dd>, <Dn>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;

FPRounding rounding;
rounding = FPDecodeRounding('10');
```

Assembler symbols

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Dn>` Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Sn>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, FALSE);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.161 FRINTN (vector)

Floating-point Round to Integral, to nearest with ties to even (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	1	1	1	1	0	0	1	1	0	0	0	1	0	Rn		Rd	
		U						o2											o1						

Half-precision variant

```
FRINTN <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	0	0	1	0	Rn		Rd	
		U						o2											o1						

Single-precision and double-precision variant

```
FRINTN <Vd>.<T>, <Vn>.<T>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- 4H when Q = 0
- 8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.162 FRINTN (scalar)

Floating-point Round to Integral, to nearest with ties to even (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	17	15	14	13	12	11	10	9	5	4	0
0	0	0	1	1	1	1	0	ftype	1	0	0	1	0	0	0	1	0	0	0	0	Rn		Rd
													rmode										

Half-precision variant

Applies when `ftype == 11`.

```
FRINTN <Hd>, <Hn>
```

Single-precision variant

Applies when `ftype == 00`.

```
FRINTN <Sd>, <Sn>
```

Double-precision variant

Applies when `ftype == 01`.

```
FRINTN <Dd>, <Dn>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;

FPRounding rounding;
rounding = FPDecodeRounding('00');
```

Assembler symbols

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Dn>` Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Sn>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, FALSE);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.163 FRINTP (vector)

Floating-point Round to Integral, toward Plus infinity (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	1	1	1	1	1	0	0	1	1	0	0	0	1	0	Rn		Rd	
		U						o2											o1						

Half-precision variant

```
FRINTP <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	0	0	1	0	Rn		Rd	
		U						o2											o1						

Single-precision and double-precision variant

```
FRINTP <Vd>.<T>, <Vn>.<T>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- 4H when Q = 0
- 8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.164 FRINTP (scalar)

Floating-point Round to Integral, toward Plus infinity (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	17	15	14	13	12	11	10	9 5	4 0
0	0	0	1	1	1	1	0	ftype	1	0	0	1	0	0	1	1	0	0	0	0	Rn	Rd
													rmode								

Half-precision variant

Applies when `ftype == 11`.

```
FRINTP <Hd>, <Hn>
```

Single-precision variant

Applies when `ftype == 00`.

```
FRINTP <Sd>, <Sn>
```

Double-precision variant

Applies when `ftype == 01`.

```
FRINTP <Dd>, <Dn>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;

FPRounding rounding;
rounding = FPDecodeRounding('01');
```

Assembler symbols

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Dn>` Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Sn>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, FALSE);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.165 FRINTX (vector)

Floating-point Round to Integral exact, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

When a result value is not numerically equal to the corresponding input value, an Inexact exception is raised. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	0	1	1	1	1	0	0	1	1	0	0	1	1	0	Rn		Rd	
		U						o2											o1						

Half-precision variant

```
FRINTX <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	0	1	1	0	Rn		Rd	
		U						o2											o1						

Single-precision and double-precision variant

```
FRINTX <Vd>.<T>, <Vn>.<T>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- 4H when Q = 0
- 8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.166 FRINTX (scalar)

Floating-point Round to Integral exact, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

When the result value is not numerically equal to the input value, an Inexact exception is raised. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30 29 28	27 26 25 24	23 22	21	20 19	18 17	15 14 13 12	11 10	9 5	4 0
0 0 0	1 1 1 1 0	ftype	1	0 0 1	1 1 0	1 0 0 0	0	Rn	Rd
					rmode				

Half-precision variant

Applies when `ftype == 11`.

```
FRINTX <Hd>, <Hn>
```

Single-precision variant

Applies when `ftype == 00`.

```
FRINTX <Sd>, <Sn>
```

Double-precision variant

Applies when `ftype == 01`.

```
FRINTX <Dd>, <Dn>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;

FPRounding rounding;
rounding = FPRoundingMode(FPCR);
```

Assembler symbols

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Dn>` Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Sn>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, TRUE);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.167 FRINTZ (vector)

Floating-point Round to Integral, toward Zero (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	1	1	1	1	1	0	0	1	1	0	0	1	1	0	Rn		Rd	
		U						o2											o1						

Half-precision variant

```
FRINTZ <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	0	1	1	0	Rn		Rd	
		U						o2											o1						

Single-precision and double-precision variant

```
FRINTZ <Vd>.<T>, <Vn>.<T>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- 4H when Q = 0
- 8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.168 FRINTZ (scalar)

Floating-point Round to Integral, toward Zero (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	17	15	14	13	12	11	10	9 5	4 0
0	0	0	1	1	1	1	0	ftype	1	0	0	1	0	1	1	0	0	0	0	Rn	Rd
													rmode								

Half-precision variant

Applies when `ftype == 11`.

```
FRINTZ <Hd>, <Hn>
```

Single-precision variant

Applies when `ftype == 00`.

```
FRINTZ <Sd>, <Sn>
```

Double-precision variant

Applies when `ftype == 01`.

```
FRINTZ <Dd>, <Dn>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;

FPRounding rounding;
rounding = FPDecodeRounding('11');
```

Assembler symbols

`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Dn>`	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Sn>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, FALSE);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.169 FRSQRTE

Floating-point Reciprocal Square Root Estimate. This instruction calculates an approximate square root for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	1	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn		Rd	

Scalar half precision variant

```
FRSQRTE <Hd>, <Hn>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	1	0	1	1	0	Rn		Rd	

Scalar single-precision and double-precision variant

```
FRSQRTE <V><d>, <V><n>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```

Vector half precision

ARMv8.2

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	1	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn		Rd	

Vector half precision variant

```
FRSQRTE <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	1	0	1	1	0	Rn		Rd	

Vector single-precision and double-precision variant

```
FRSQRTE <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler symbols

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

`<V>` Is a width specifier, encoded in the "sz" field. It can have the following values:

- `S` when `sz = 0`
- `D` when `sz = 1`

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

`<n>` Is the number of the SIMD&FP source register, encoded in the "Rn" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the vector half precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `4H` when `Q = 0`
- `8H` when `Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

`2S` when `sz = 0, Q = 0`

`4S` when `sz = 0, Q = 1`

`2D` when `sz = 1, Q = 1`

The encoding `sz = 1, Q = 0` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRSqrtEstimate(element, FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.170 FRSQRTS

Floating-point Reciprocal Square Root Step. This instruction multiplies corresponding floating-point values in the vectors of the two source SIMD&FP registers, subtracts each of the products from 3.0, divides these results by 2.0, places the results into a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	1	1	0	Rm		0	0	1	1	1	1	Rn		Rd	

Scalar half precision variant

```
FRSQRTS <Hd>, <Hn>, <Hm>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	1	sz	1	Rm		1	1	1	1	1	1	Rn		Rd	

Scalar single-precision and double-precision variant

```
FRSQRTS <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```

Vector half precision

ARMv8.2

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	1	1	0	Rm		0	0	1	1	1	1	Rn		Rd	

Vector half precision variant

```
FRSQRTS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	1	sz	1	Rm		1	1	1	1	1	1	Rn		Rd	

Vector single-precision and double-precision variant

```
FRSQRTS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler symbols

`<Hd>`	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Hn>`	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
`<Hm>`	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

`<V>`	Is a width specifier, encoded in the "sz" field. It can have the following values:
	`S` when `sz = 0`
	`D` when `sz = 1`

`<d>`	Is the number of the SIMD&FP destination register, in the "Rd" field.
`<n>`	Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
`<m>`	Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
`<Vd>`	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<T> For the vector half precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

4H when Q = 0

8H when Q = 1

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0

4S when sz = 0, Q = 1

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPRSqrtStepFused(element1, element2);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.171 FSQRT (vector)

Floating-point Square Root (vector). This instruction calculates the square root for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	1	1	1	1	1	0	0	1	1	1	1	1	1	0	Rn		Rd	

Half-precision variant

```
FSQRT <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	1	1	1	1	0	Rn		Rd	

Single-precision and double-precision variant

```
FSQRT <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

4H when Q = 0

8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0

4S when sz = 0, Q = 1

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPSqrt(element, FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.172 FSQRT (scalar)

Floating-point Square Root (scalar). This instruction calculates the square root of the value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 19 18 17	16	15	14 13 12	11 10 9	5 4	0
0	0	0	1	1	1	1	0	ftype	1	0 0 0 0	1	1	1 0 0	0 0	Rn	Rd
											opc					

Half-precision variant

Applies when `ftype == 11`.

```
FSQRT <Hd>, <Hn>
```

Single-precision variant

Applies when `ftype == 00`.

```
FSQRT <Sd>, <Sn>
```

Double-precision variant

Applies when `ftype == 01`.

```
FSQRT <Dd>, <Dn>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;
```

Assembler symbols

`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Dn>`	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
`<Hd>`	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Hn>`	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
`<Sd>`	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPSqrt(operand, FPCR);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.173 FSUB (vector)

Floating-point Subtract (vector). This instruction subtracts the elements in the vector in the second source SIMD&FP register, from the corresponding elements in the vector in the first source SIMD&FP register, places each result into elements of a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	1	1	0	Rm		0	0	0	1	0	1	Rn		Rd	
		U																				

Half-precision variant

```
FSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	1	sz	1	Rm		1	1	0	1	0	1	Rn		Rd	
		U																				

Single-precision and double-precision variant

```
FSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`4H` when `Q = 0`

`8H` when `Q = 1`

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

`2S` when `sz = 0`, `Q = 0`

`4S` when `sz = 0`, `Q = 1`

`2D` when `sz = 1`, `Q = 1`

The encoding `sz = 1`, `Q = 0` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) diff;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    diff = FPSub(element1, element2, FPCR);
    Elem[result, e, esize] = if abs then FPAbs(diff) else diff;

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.174 FSUB (scalar)

Floating-point Subtract (scalar). This instruction subtracts the floating-point value of the second source SIMD&FP register from the floating-point value of the first source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	0	0	1	1	1	1	0	ftype	1	Rm	0	0	1	1	1	0	Rn	Rd
														op				

Half-precision variant

Applies when `ftype == 11`.

```
FSUB <Hd>, <Hn>, <Hm>
```

Single-precision variant

Applies when `ftype == 00`.

```
FSUB <Sd>, <Sn>, <Sm>
```

Double-precision variant

Applies when `ftype == 01`.

```
FSUB <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;
```

Assembler symbols

`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Dn>`	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
`<Dm>`	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
`<Hd>`	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Hn>` Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Sn>` Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result = FPSub(operand1, operand2, FPCR);
V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.175 INS (element)

Insert vector element from another vector element. This instruction copies the vector element of the source SIMD&FP register to the specified vector element of the destination SIMD&FP register.

This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (element). The alias is always the preferred disassembly.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	11	10	9	5	4	0
0	1	1	0	1	1	1	0	0	0	0	imm5		0	imm4		1	Rn			Rd

Advanced SIMD variant

```
INS <Vd>.<Ts>[<index1>], <Vn>.<Ts>[<index2>]
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);
if size > 3 then UNDEFINED;

integer dst_index = UInt(imm5<4:size+1>);
integer src_index = UInt(imm4<3:size>);
integer idxdsize = if imm4<3> == '1' then 128 else 64;
// imm4<size-1:0> is IGNORED

integer esize = 8 << size;
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ts>` Is an element size specifier, encoded in the "imm5" field. It can have the following values:

- `B` when `imm5 = xxxx1`
- `H` when `imm5 = xxx10`
- `S` when `imm5 = xx100`
- `D` when `imm5 = x1000`

The encoding `imm5 = x0000` is reserved.

`<index1>` Is the destination element index encoded in the "imm5" field. It can have the following values:

- `imm5<4:1>` when `imm5 = xxxx1`
- `imm5<4:2>` when `imm5 = xxx10`
- `imm5<4:3>` when `imm5 = xx100`
- `imm5<4>` when `imm5 = x1000`

The encoding `imm5 = x0000` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<index2>` Is the source element index encoded in the "imm5:imm4" field. It can have the following values:

`imm4<3:0>` when `imm5 = xxxx1`

`imm4<3:1>` when `imm5 = xxx10`

`imm4<3:2>` when `imm5 = xx100`

`imm4<3>` when `imm5 = x1000`

The encoding `imm5 = x0000` is reserved.

Unspecified bits in "imm4" are ignored but should be set to zero by an assembler.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(idxdsize) operand = V[n];
bits(128) result;

result = V[d];
Elem[result, dst_index, esize] = Elem[operand, src_index, esize];
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.176 INS (general)

Insert vector element from general-purpose register. This instruction copies the contents of the source general-purpose register to the specified vector element in the destination SIMD&FP register.

This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (from general). The alias is always the preferred disassembly.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	1	0	0	1	1	1	0	0	0	0	imm5		0	0	0	1	1	1	Rn		Rd	

Advanced SIMD variant

```
INS <Vd>.<Ts>[<index>], <R><n>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);

if size > 3 then UNDEFINED;
integer index = UInt(imm5<4:size+1>);

integer esize = 8 << size;
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ts>` Is an element size specifier, encoded in the "imm5" field. It can have the following values:

- `B` when `imm5 = xxxx1`
- `H` when `imm5 = xxx10`
- `S` when `imm5 = xx100`
- `D` when `imm5 = x1000`

The encoding `imm5 = x0000` is reserved.

`<index>` Is the element index encoded in the "imm5" field. It can have the following values:

- `imm5<4:1>` when `imm5 = xxxx1`
- `imm5<4:2>` when `imm5 = xxx10`
- `imm5<4:3>` when `imm5 = xx100`
- `imm5<4>` when `imm5 = x1000`

The encoding `imm5 = x0000` is reserved.

`<R>` Is the width specifier for the general-purpose source register, encoded in the "imm5" field. It can have the following values:

- `W` when `imm5 = xxxx1`
- `W` when `imm5 = xxx10`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`W` when `imm5 = xx100`

`X` when `imm5 = x1000`

The encoding `imm5 = x0000` is reserved.

`<n>` Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(esize) element = X[n];
bits(128) result;

result = V[d];
Elem[result, index, esize] = element;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.177 LD1 (multiple structures)

Load multiple single-element structures to one, two, three, or four registers. This instruction loads multiple single-element structures from memory and writes the result to one, two, three, or four SIMD&FP registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15			12	11	10	9	5	4	0
0	Q	0	0	1	1	0	0	0	1	0	0	0	0	0	0	x	x	1	x	size		Rn		Rt	
									L							opcode									

One register variant

Applies when `opcode == 0111`.

```
LD1 { <Vt>.<T> }, [<Xn|SP>]
```

Two registers variant

Applies when `opcode == 1010`.

```
LD1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

Three registers variant

Applies when `opcode == 0110`.

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

Four registers variant

Applies when `opcode == 0010`.

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20		16	15			12	11	10	9	5	4	0
0	Q	0	0	1	1	0	0	1	1	0		Rm		x	x	1	x	size		Rn		Rt	
									L							opcode							

One register, immediate offset variant

Applies when `Rm == 11111 && opcode == 0111`.

```
LD1 { <Vt>.<T> }, [<Xn|SP>], <imm>
```

One register, register offset variant

Applies when `Rm != 11111 && opcode == 0111`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
LD1 { <Vt>.<T> }, [<Xn|SP>], <Xm>
```

Two registers, immediate offset variant

Applies when `Rm == 11111 && opcode == 1010`.

```
LD1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Two registers, register offset variant

Applies when `Rm != 11111 && opcode == 1010`.

```
LD1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

Three registers, immediate offset variant

Applies when `Rm == 11111 && opcode == 0110`.

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Three registers, register offset variant

Applies when `Rm != 11111 && opcode == 0110`.

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

Four registers, immediate offset variant

Applies when `Rm == 11111 && opcode == 0010`.

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

Four registers, register offset variant

Applies when `Rm != 11111 && opcode == 0010`.

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler symbols

`<Vt>` Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B`	when `size = 00`, `Q = 0`
`16B`	when `size = 00`, `Q = 1`
`4H`	when `size = 01`, `Q = 0`
`8H`	when `size = 01`, `Q = 1`
`2S`	when `size = 10`, `Q = 0`
`4S`	when `size = 10`, `Q = 1`
`1D`	when `size = 11`, `Q = 0`
`2D`	when `size = 11`, `Q = 1`

`<Vt2>` Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

`<Vt3>` Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Vt4>` Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<imm>` For the one register, immediate offset variant: is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

`#8` when `Q = 0`

`#16` when `Q = 1`

For the two registers, immediate offset variant: is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

`#16` when `Q = 0`

`#32` when `Q = 1`

For the three registers, immediate offset variant: is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

`#24` when `Q = 0`

`#48` when `Q = 1`

For the four registers, immediate offset variant: is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

`#32` when `Q = 0`

`#64` when `Q = 1`

`<Xm>` Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared decode for all encodings

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;    // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
    otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.178 LD1 (single structure)

Load one single-element structure to one lane of one register. This instruction loads a single-element structure from memory and writes the result to the specified lane of the SIMD&FP register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	1	0	1	0	0	0	0	0	0	x	x	0	S	size			Rn		Rt
									L	R						opcode									

8-bit variant

Applies when `opcode == 000`.

```
LD1 { <Vt>.B }[<index>], [<Xn|SP>]
```

16-bit variant

Applies when `opcode == 010 && size == x0`.

```
LD1 { <Vt>.H }[<index>], [<Xn|SP>]
```

32-bit variant

Applies when `opcode == 100 && size == 00`.

```
LD1 { <Vt>.S }[<index>], [<Xn|SP>]
```

64-bit variant

Applies when `opcode == 100 && S == 0 && size == 01`.

```
LD1 { <Vt>.D }[<index>], [<Xn|SP>]
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	1	1	1	0	Rm		x	x	0	S	size			Rn		Rt
									L	R			opcode									

8-bit, immediate offset variant

Applies when `Rm == 11111 && opcode == 000`.

```
LD1 { <Vt>.B }[<index>], [<Xn|SP>], #1
```

8-bit, register offset variant

Applies when `Rm != 11111 && opcode == 000`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
LD1 { <Vt>.B }[<index>], [<Xn|SP>], <Xm>
```

16-bit, immediate offset variant

Applies when `Rm == 11111 && opcode == 010 && size == x0`.

```
LD1 { <Vt>.H }[<index>], [<Xn|SP>], #2
```

16-bit, register offset variant

Applies when `Rm != 11111 && opcode == 010 && size == x0`.

```
LD1 { <Vt>.H }[<index>], [<Xn|SP>], <Xm>
```

32-bit, immediate offset variant

Applies when `Rm == 11111 && opcode == 100 && size == 00`.

```
LD1 { <Vt>.S }[<index>], [<Xn|SP>], #4
```

32-bit, register offset variant

Applies when `Rm != 11111 && opcode == 100 && size == 00`.

```
LD1 { <Vt>.S }[<index>], [<Xn|SP>], <Xm>
```

64-bit, immediate offset variant

Applies when `Rm == 11111 && opcode == 100 && S == 0 && size == 01`.

```
LD1 { <Vt>.D }[<index>], [<Xn|SP>], #8
```

64-bit, register offset variant

Applies when `Rm != 11111 && opcode == 100 && S == 0 && size == 01`.

```
LD1 { <Vt>.D }[<index>], [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler symbols

`<Vt>`	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
`<index>`	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared decode for all encodings

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation for all encodings

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.179 LD1R

Load one single-element structure and Replicate to all lanes (of one register). This instruction loads a single-element structure from memory and replicates the structure to all the lanes of the SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	size		Rn		Rt	
									L	R						opcode			S						

No offset variant

```
LD1R { <Vt>.<T> }, [<Xn|SP>]
```

Decode for this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	1	1	1	0	Rm		1	1	0	0	size		Rn		Rt	
									L	R			opcode			S						

Immediate offset variant

Applies when `Rm == 11111`.

```
LD1R { <Vt>.<T> }, [<Xn|SP>], <imm>
```

Register offset variant

Applies when `Rm != 11111`.

```
LD1R { <Vt>.<T> }, [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler symbols

`<Vt>` Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `8B` when `size = 00`, `Q = 0`
- `16B` when `size = 00`, `Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1
1D	when size = 11, Q = 0
2D	when size = 11, Q = 1

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in the "size" field. It can have the following values:

#1	when size = 00
#2	when size = 01
#4	when size = 10
#8	when size = 11

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared decode for all encodings

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation for all encodings

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.180 LD2 (multiple structures)

Load multiple 2-element structures to two registers. This instruction loads multiple 2-element structures from memory and writes the result to the two SIMD&FP registers, with de-interleaving.

For an example of de-interleaving, see `LD3 (multiple structures)`.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

No offset

31	30	29 28	27 26 25 24	23	22 21 20	19 18 17 16	15 12	11 10	9 5	4 0
0	Q	0 0	1 1 0 0	0	1	0 0 0 0	0 0	1 0	size	Rn	Rt
					L		opcode			

No offset variant

```
LD2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

Decode for this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index

31	30	29 28	27 26 25 24	23	22	21 20	16	15 12	11 10	9 5	4 0
0	Q	0 0	1 1 0 0	1	1	0	Rm	1 0 0 0	size	Rn	Rt
					L			opcode			

Immediate offset variant

Applies when `Rm == 11111`.

```
LD2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Register offset variant

Applies when `Rm != 11111`.

```
LD2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler symbols

`<Vt>` Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B` when `size = 00`, `Q = 0`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

16B	when size = 00, Q = 1
4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1
2D	when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

#16	when Q = 0
#32	when Q = 1

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared decode for all encodings

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;    // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
    otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.181 LD2 (single structure)

Load single 2-element structure to one lane of two registers. This instruction loads a 2-element structure from memory and writes the result to the corresponding elements of the two SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	1	0	1	1	0	0	0	0	0	x	x	0	S	size			Rn		Rt
									L	R						opcode									

8-bit variant

Applies when `opcode == 000`.

```
LD2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>]
```

16-bit variant

Applies when `opcode == 010 && size == x0`.

```
LD2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>]
```

32-bit variant

Applies when `opcode == 100 && size == 00`.

```
LD2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>]
```

64-bit variant

Applies when `opcode == 100 && S == 0 && size == 01`.

```
LD2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>]
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	1	1	1	1	Rm		x	x	0	S	size			Rn		Rt
									L	R			opcode									

8-bit, immediate offset variant

Applies when `Rm == 11111 && opcode == 000`.

```
LD2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], #2
```

8-bit, register offset variant

Applies when `Rm != 11111 && opcode == 000`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
LD2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], <Xm>
```

16-bit, immediate offset variant

Applies when `Rm == 11111 && opcode == 010 && size == x0`.

```
LD2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], #4
```

16-bit, register offset variant

Applies when `Rm != 11111 && opcode == 010 && size == x0`.

```
LD2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], <Xm>
```

32-bit, immediate offset variant

Applies when `Rm == 11111 && opcode == 100 && size == 00`.

```
LD2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], #8
```

32-bit, register offset variant

Applies when `Rm != 11111 && opcode == 100 && size == 00`.

```
LD2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], <Xm>
```

64-bit, immediate offset variant

Applies when `Rm == 11111 && opcode == 100 && S == 0 && size == 01`.

```
LD2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], #16
```

64-bit, register offset variant

Applies when `Rm != 11111 && opcode == 100 && S == 0 && size == 01`.

```
LD2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler symbols

`<Vt>`	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
`<Vt2>`	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
`<index>`	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Shared decode for all encodings

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation for all encodings

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.182 LD2R

Load single 2-element structure and Replicate to all lanes of two registers. This instruction loads a 2-element structure from memory and replicates the structure to all the lanes of the two SIMD&FP registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	1	0	1	1	0	0	0	0	0	1	1	0	0	size		Rn		Rt	
									L	R						opcode			S						

No offset variant

```
LD2R { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

Decode for this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	1	1	1	1	Rm		1	1	0	0	size		Rn		Rt	
									L	R			opcode			S						

Immediate offset variant

Applies when `Rm == 11111`.

```
LD2R { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Register offset variant

Applies when `Rm != 11111`.

```
LD2R { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler symbols

`<Vt>` Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `8B` when `size = 00`, `Q = 0`
- `16B` when `size = 00`, `Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1
1D	when size = 11, Q = 0
2D	when size = 11, Q = 1

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in the "size" field. It can have the following values:

#2	when size = 00
#4	when size = 01
#8	when size = 10
#16	when size = 11

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared decode for all encodings

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation for all encodings

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Operational information

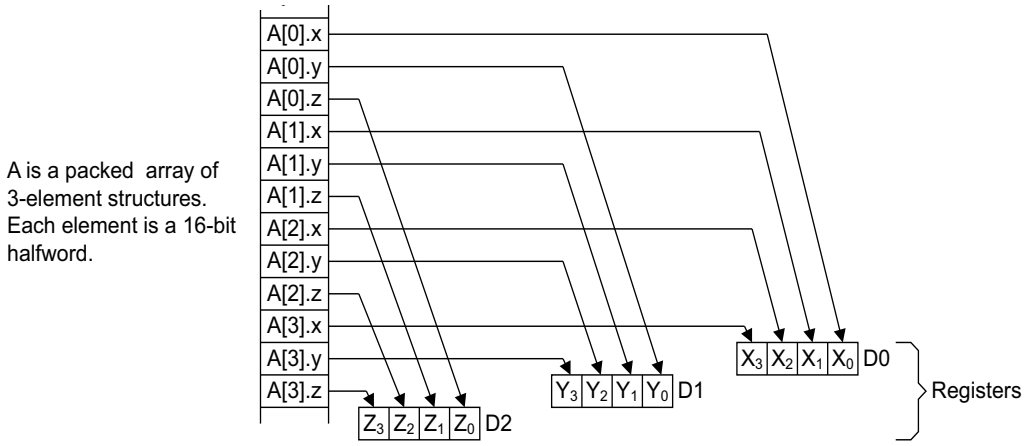
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.183 LD3 (multiple structures)

Load multiple 3-element structures to three registers. This instruction loads multiple 3-element structures from memory and writes the result to the three SIMD&FP registers, with de-interleaving.

The following figure shows the operation of de-interleaving of a LD3.16 (multiple 3-element structures) instruction:.



Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15			12	11	10	9	5	4	0
0	Q	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	size			Rn		Rt
									L							opcode									

No offset variant

```
LD3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

Decode for this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20		16	15			12	11	10	9	5	4	0
0	Q	0	0	1	1	0	0	1	1	0	Rm			0	1	0	0	size			Rn		Rt
									L							opcode							

Immediate offset variant

Applies when `Rm == 11111`.

```
LD3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Register offset variant

Applies when `Rm != 11111`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
LD3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler symbols

`<Vt>` Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B`	when `size = 00`, `Q = 0`
`16B`	when `size = 00`, `Q = 1`
`4H`	when `size = 01`, `Q = 0`
`8H`	when `size = 01`, `Q = 1`
`2S`	when `size = 10`, `Q = 0`
`4S`	when `size = 10`, `Q = 1`
`2D`	when `size = 11`, `Q = 1`

The encoding `size = 11`, `Q = 0` is reserved.

`<Vt2>` Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

`<Vt3>` Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<imm>` Is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

`#24`	when `Q = 0`
`#48`	when `Q = 1`

`<Xm>` Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared decode for all encodings

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;    // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
    otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.184 LD3 (single structure)

Load single 3-element structure to one lane of three registers). This instruction loads a 3-element structure from memory and writes the result to the corresponding elements of the three SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	1	0	1	0	0	0	0	0	0	x	x	1	S	size		Rn		Rt
									L	R						opcode								

8-bit variant

Applies when `opcode == 001`.

```
LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>]
```

16-bit variant

Applies when `opcode == 011 && size == x0`.

```
LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>]
```

32-bit variant

Applies when `opcode == 101 && size == 00`.

```
LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>]
```

64-bit variant

Applies when `opcode == 101 && S == 0 && size == 01`.

```
LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>]
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	16	15	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	1	1	1	0		Rm	x	x	1	S	size		Rn		Rt
									L	R			opcode								

8-bit, immediate offset variant

Applies when `Rm == 11111 && opcode == 001`.

```
LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], #3
```

8-bit, register offset variant

Applies when `Rm != 11111 && opcode == 001`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], <Xm>
```

16-bit, immediate offset variant

Applies when `Rm == 11111 && opcode == 011 && size == x0`.

```
LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], #6
```

16-bit, register offset variant

Applies when `Rm != 11111 && opcode == 011 && size == x0`.

```
LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], <Xm>
```

32-bit, immediate offset variant

Applies when `Rm == 11111 && opcode == 101 && size == 00`.

```
LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], #12
```

32-bit, register offset variant

Applies when `Rm != 11111 && opcode == 101 && size == 00`.

```
LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], <Xm>
```

64-bit, immediate offset variant

Applies when `Rm == 11111 && opcode == 101 && S == 0 && size == 01`.

```
LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], #24
```

64-bit, register offset variant

Applies when `Rm != 11111 && opcode == 101 && S == 0 && size == 01`.

```
LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler symbols

`<Vt>`	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
`<Vt2>`	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
`<Vt3>`	Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
`<index>`	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Shared decode for all encodings

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation for all encodings

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.185 LD3R

Load single 3-element structure and Replicate to all lanes of three registers. This instruction loads a 3-element structure from memory and replicates the structure to all the lanes of the three SIMD&FP registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	1	0	1	0	0	0	0	0	0	1	1	1	0	size		Rn		Rt	
									L	R						opcode			S						

No offset variant

```
LD3R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

Decode for this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	1	1	1	0	Rm		1	1	1	0	size		Rn		Rt	
									L	R			opcode			S						

Immediate offset variant

Applies when `Rm == 11111`.

```
LD3R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Register offset variant

Applies when `Rm != 11111`.

```
LD3R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler symbols

`<Vt>` Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `8B` when `size = 00`, `Q = 0`
- `16B` when `size = 00`, `Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1
1D	when size = 11, Q = 0
2D	when size = 11, Q = 1

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in the "size" field. It can have the following values:

#3	when size = 00
#6	when size = 01
#12	when size = 10
#24	when size = 11

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared decode for all encodings

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation for all encodings

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.186 LD4 (multiple structures)

Load multiple 4-element structures to four registers. This instruction loads multiple 4-element structures from memory and writes the result to the four SIMD&FP registers, with de-interleaving.

For an example of de-interleaving, see `LD3 (multiple structures)`.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	size	Rn	Rt	
									L							opcode							

No offset variant

```
LD4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

Decode for this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	16	15	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	0	1	1	0	Rm		0	0	0	0	size	Rn	Rt	
									L				opcode							

Immediate offset variant

Applies when `Rm == 11111`.

```
LD4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

Register offset variant

Applies when `Rm != 11111`.

```
LD4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler symbols

`<Vt>` Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B` when `size = 00`, `Q = 0`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

16B	when size = 00, Q = 1
4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1
2D	when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

#32	when Q = 0
#64	when Q = 1

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared decode for all encodings

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;    // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
    otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.187 LD4 (single structure)

Load single 4-element structure to one lane of four registers. This instruction loads a 4-element structure from memory and writes the result to the corresponding elements of the four SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	1	0	1	1	0	0	0	0	0	x	x	1	S	size			Rn		Rt
									L	R						opcode									

8-bit variant

Applies when `opcode == 001`.

```
LD4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>]
```

16-bit variant

Applies when `opcode == 011 && size == x0`.

```
LD4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>]
```

32-bit variant

Applies when `opcode == 101 && size == 00`.

```
LD4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>]
```

64-bit variant

Applies when `opcode == 101 && S == 0 && size == 01`.

```
LD4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>]
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	1	1	1	1	Rm		x	x	1	S	size			Rn		Rt
									L	R			opcode									

8-bit, immediate offset variant

Applies when `Rm == 11111 && opcode == 001`.

```
LD4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], #4
```

8-bit, register offset variant

Applies when `Rm != 11111 && opcode == 001`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
LD4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], <Xm>
```

16-bit, immediate offset variant

Applies when `Rm == 11111 && opcode == 011 && size == x0`.

```
LD4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], #8
```

16-bit, register offset variant

Applies when `Rm != 11111 && opcode == 011 && size == x0`.

```
LD4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], <Xm>
```

32-bit, immediate offset variant

Applies when `Rm == 11111 && opcode == 101 && size == 00`.

```
LD4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], #16
```

32-bit, register offset variant

Applies when `Rm != 11111 && opcode == 101 && size == 00`.

```
LD4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], <Xm>
```

64-bit, immediate offset variant

Applies when `Rm == 11111 && opcode == 101 && S == 0 && size == 01`.

```
LD4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], #32
```

64-bit, register offset variant

Applies when `Rm != 11111 && opcode == 101 && S == 0 && size == 01`.

```
LD4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler symbols

`<Vt>`	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
`<Vt2>`	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
`<Vt3>`	Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
`<Vt4>`	Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
`<index>`	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Shared decode for all encodings

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation for all encodings

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.188 LD4R

Load single 4-element structure and Replicate to all lanes of four registers. This instruction loads a 4-element structure from memory and replicates the structure to all the lanes of the four SIMD&FP registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	1	0	1	1	0	0	0	0	0	1	1	1	0	size		Rn		Rt	
									L	R						opcode			S						

No offset variant

```
LD4R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

Decode for this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	1	1	1	1	Rm		1	1	1	0	size		Rn		Rt	
									L	R			opcode			S						

Immediate offset variant

Applies when `Rm == 11111`.

```
LD4R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

Register offset variant

Applies when `Rm != 11111`.

```
LD4R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler symbols

`<Vt>` Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `8B` when `size = 00`, `Q = 0`
- `16B` when `size = 00`, `Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1
1D	when size = 11, Q = 0
2D	when size = 11, Q = 1

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in the "size" field. It can have the following values:

#4	when size = 00
#8	when size = 01
#16	when size = 10
#32	when size = 11

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared decode for all encodings

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation for all encodings

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.189 LDNP (SIMD&FP)

Load Pair of SIMD&FP registers, with Non-temporal hint. This instruction loads a pair of SIMD&FP registers from memory, issuing a hint to the memory system that the access is non-temporal. The address that is used for the load is calculated from a base register value and an optional immediate offset.

For information about non-temporal pair instructions, see *Load/Store SIMD and Floating-point Non-temporal pair* on page C3-218.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30	29 28	27	26	25 24	23	22 21	15 14	10 9	5 4	0
opc	1 0	1	1	0 0	0	1	imm7	Rt2	Rn	Rt
						L				

32-bit variant

Applies when `opc == 00`.

```
LDNP <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

64-bit variant

Applies when `opc == 01`.

```
LDNP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

128-bit variant

Applies when `opc == 10`.

```
LDNP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

Decode for all variants of this encoding

```
// Empty.
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *LDNP (SIMD&FP)* on page K1-7863.

Assembler symbols

`<Dt1>`	Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
`<Dt2>`	Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
`<Qt1>`	Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
`<Qt2>`	Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
`<St1>`	Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
`<St2>`	Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<imm>`	For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

For the 128-bit variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = n != 31;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data1 = Mem[address, dbytes, AccType_VECSTREAM];
data2 = Mem[address+dbytes, dbytes, AccType_VECSTREAM];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
V[t] = data1;
V[t2] = data2;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.190 LDP (SIMD&FP)

Load Pair of SIMD&FP registers. This instruction loads a pair of SIMD&FP registers from memory. The address that is used for the load is calculated from a base register value and an optional immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Post-index

31 30 29 28	27 26	25 24	23	22 21	15 14	10 9	5 4	0
opc	1 0 1	1	0 0 1	1 (L)	imm7	Rt2	Rn	Rt

32-bit variant

Applies when `opc == 00`.

```
LDP <St1>, <St2>, [<Xn|SP>], #<imm>
```

64-bit variant

Applies when `opc == 01`.

```
LDP <Dt1>, <Dt2>, [<Xn|SP>], #<imm>
```

128-bit variant

Applies when `opc == 10`.

```
LDP <Qt1>, <Qt2>, [<Xn|SP>], #<imm>
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
```

Pre-index

31 30 29 28	27 26	25 24	23	22 21	15 14	10 9	5 4	0
opc	1 0 1	1	0 1 1	1 (L)	imm7	Rt2	Rn	Rt

32-bit variant

Applies when `opc == 00`.

```
LDP <St1>, <St2>, [<Xn|SP>, #<imm>]!
```

64-bit variant

Applies when `opc == 01`.

```
LDP <Dt1>, <Dt2>, [<Xn|SP>, #<imm>]!
```

128-bit variant

Applies when `opc == 10`.

```
LDP <Qt1>, <Qt2>, [<Xn|SP>, #<imm>]!
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
```

Signed offset

31 30	29 28 27	26	25 24 23	22	21 … 15	14 … 10	9 … 5	4 … 0
opc	1 0 1	1	0 1 0	1 (L)	imm7	Rt2	Rn	Rt

32-bit variant

Applies when `opc == 00`.

```
LDP <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

64-bit variant

Applies when `opc == 01`.

```
LDP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

128-bit variant

Applies when `opc == 10`.

```
LDP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

Decode for all variants of this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *LDP (SIMD&FP)* on page K1-7863, and particularly *LDNP (SIMD&FP)* on page K1-7863.

Assembler symbols

`<Dt1>`	Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
`<Dt2>`	Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
`<Qt1>`	Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
`<Qt2>`	Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
`<St1>`	Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
`<St2>`	Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<imm>`	For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4. For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4. For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

For the 128-bit post-index and 128-bit pre-index variant: is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the "imm7" field as <imm>/16.

For the 128-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = wback || n != 31;
```

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

data1 = Mem[address, dbytes, AccType_VEC];
data2 = Mem[address+dbytes, dbytes, AccType_VEC];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
V[t] = data1;
V[t2] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.191 LDR (immediate, SIMD&FP)

Load SIMD&FP Register (immediate offset). This instruction loads an element from memory, and writes the result as a scalar to the SIMD&FP register. The address that is used for the load is calculated from a base register value, a signed immediate offset, and an optional offset that is a multiple of the element size.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Post-index

31 30	29 28	27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
size	1 1	1	1	0 0	x 1	0	imm9	0 1	Rn	Rt
					opc					

8-bit variant

Applies when `size == 00 && opc == 01`.

```
LDR <Bt>, [<Xn|SP>], #<simm>
```

16-bit variant

Applies when `size == 01 && opc == 01`.

```
LDR <Ht>, [<Xn|SP>], #<simm>
```

32-bit variant

Applies when `size == 10 && opc == 01`.

```
LDR <St>, [<Xn|SP>], #<simm>
```

64-bit variant

Applies when `size == 11 && opc == 01`.

```
LDR <Dt>, [<Xn|SP>], #<simm>
```

128-bit variant

Applies when `size == 00 && opc == 11`.

```
LDR <Qt>, [<Xn|SP>], #<simm>
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31 30	29 28	27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
size	1 1	1	1	0 0	x 1	0	imm9	1 1	Rn	Rt
					opc					

8-bit variant

Applies when `size == 00 && opc == 01`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
LDR <Bt>, [<Xn|SP>, #<simm>]!
```

16-bit variant

Applies when `size == 01 && opc == 01`.

```
LDR <Ht>, [<Xn|SP>, #<simm>]!
```

32-bit variant

Applies when `size == 10 && opc == 01`.

```
LDR <St>, [<Xn|SP>, #<simm>]!
```

64-bit variant

Applies when `size == 11 && opc == 01`.

```
LDR <Dt>, [<Xn|SP>, #<simm>]!
```

128-bit variant

Applies when `size == 00 && opc == 11`.

```
LDR <Qt>, [<Xn|SP>, #<simm>]!
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31 30	29 28	27	26	25 24	23 22	21 ... 10	9 ... 5	4 ... 0
size	1 1	1	1	0 1	x 1	imm12	Rn	Rt
					opc			

8-bit variant

Applies when `size == 00 && opc == 01`.

```
LDR <Bt>, [<Xn|SP>{, #<pimm>}]
```

16-bit variant

Applies when `size == 01 && opc == 01`.

```
LDR <Ht>, [<Xn|SP>{, #<pimm>}]
```

32-bit variant

Applies when `size == 10 && opc == 01`.

```
LDR <St>, [<Xn|SP>{, #<pimm>}]
```

64-bit variant

Applies when `size == 11 && opc == 01`.

```
LDR <Dt>, [<Xn|SP>{, #<pimm>}]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

128-bit variant

Applies when `size == 00 && opc == 11`.

```
LDR <Qt>, [<Xn|SP>{, #<pimm>}]
```

Decode for all variants of this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

Assembler symbols

`<Bt>`	Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
`<Dt>`	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
`<Ht>`	Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
`<Qt>`	Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
`<St>`	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
`<pimm>`	For the 8-bit variant: is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field. For the 16-bit variant: is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2. For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4. For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8. For the 128-bit variant: is the optional positive immediate byte offset, a multiple of 16 in the range 0 to 65520, defaulting to 0 and encoded in the "imm12" field as <pimm>/16.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);
```

Operation for all encodings

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if !postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, AccType_VEC] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, AccType_VEC];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.192 LDR (literal, SIMD&FP)

Load SIMD&FP Register (PC-relative literal). This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from the PC value and an immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30	29 28 27	26	25 24	23 … 5	4 … 0
opc	0 1 1	1	0 0	imm19	Rt

32-bit variant

Applies when `opc == 00`.

```
LDR <St>, <label>
```

64-bit variant

Applies when `opc == 01`.

```
LDR <Dt>, <label>
```

128-bit variant

Applies when `opc == 10`.

```
LDR <Qt>, <label>
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer size;
bits(64) offset;

case opc of
    when '00'
        size = 4;
    when '01'
        size = 8;
    when '10'
        size = 16;
    when '11'
        UNDEFINED;

offset = SignExtend(imm19:'00', 64);
```

Assembler symbols

`<Dt>`	Is the 64-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field.
`<Qt>`	Is the 128-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field.
`<St>`	Is the 32-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field.
`<label>`	Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
bits(64) address = PC[] + offset;
bits(size*8) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(FALSE);

CheckFPAdvSIMDEnabled64();

data = Mem[address, size, AccType_VEC];
V[t] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.193 LDR (register, SIMD&FP)

Load SIMD&FP Register (register offset). This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from a base register value and an offset register value. The offset can be optionally shifted and extended.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30	29 28	27	26	25 24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
size	1 1	1	1	0 0	x 1 (opc)	1	Rm	option	S	1 0	Rn	Rt

8-fsreg,LDR-8-fsreg variant

Applies when `size == 00 && opc == 01 && option != 011`.

```
LDR <Bt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

8-fsreg,LDR-8-fsreg variant

Applies when `size == 00 && opc == 01 && option == 011`.

```
LDR <Bt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

16-fsreg,LDR-16-fsreg variant

Applies when `size == 01 && opc == 01`.

```
LDR <Ht>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

32-fsreg,LDR-32-fsreg variant

Applies when `size == 10 && opc == 01`.

```
LDR <St>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

64-fsreg,LDR-64-fsreg variant

Applies when `size == 11 && opc == 01`.

```
LDR <Dt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

128-fsreg,LDR-128-fsreg variant

Applies when `size == 00 && opc == 11`.

```
LDR <Qt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for all variants of this encoding

```
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler symbols

`<Bt>`	Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
`<Dt>`	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Ht>` Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

`<Qt>` Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

`<St>` Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<Wm>` When `option<0>` is set to `0`, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

`<Xm>` When `option<0>` is set to `1`, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

`<extend>` For the 8-bit variant: is the index extend specifier, encoded in the "option" field. It can have the following values:

`UXTW` when `option = 010`

`SXTW` when `option = 110`

`SXTX` when `option = 111`

For the 128-bit, 16-bit, 32-bit and 64-bit variant: is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in the "option" field. It can have the following values:

`UXTW` when `option = 010`

`LSL` when `option = 011`

`SXTW` when `option = 110`

`SXTX` when `option = 111`

`<amount>` For the 8-bit variant: is the index shift amount, it must be `#0`, encoded in "S" as `0` if omitted, or as `1` if present.

For the 16-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

`#0` when `S = 0`

`#1` when `S = 1`

For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

`#0` when `S = 0`

`#2` when `S = 1`

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

`#0` when `S = 0`

`#3` when `S = 1`

For the 128-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

`#0` when `S = 0`

`#4` when `S = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, AccType_VEC] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, AccType_VEC];
        V[t] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.194 LDUR (SIMD&FP)

Load SIMD&FP Register (unscaled offset). This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from a base register value and an optional immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30	29 28	27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
size	1 1	1	1	0 0	x 1 (opc)	0	imm9	0 0	Rn	Rt

8-bit variant

Applies when `size == 00 && opc == 01`.

```
LDUR <Bt>, [<Xn|SP>{, #<simm>}]
```

16-bit variant

Applies when `size == 01 && opc == 01`.

```
LDUR <Ht>, [<Xn|SP>{, #<simm>}]
```

32-bit variant

Applies when `size == 10 && opc == 01`.

```
LDUR <St>, [<Xn|SP>{, #<simm>}]
```

64-bit variant

Applies when `size == 11 && opc == 01`.

```
LDUR <Dt>, [<Xn|SP>{, #<simm>}]
```

128-bit variant

Applies when `size == 00 && opc == 11`.

```
LDUR <Qt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Bt>`	Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
`<Dt>`	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
`<Ht>`	Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
`<Qt>`	Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
`<St>`	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);
```

Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, AccType_VEC] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, AccType_VEC];
        V[t] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.195 MLA (by element)

Multiply-Add to accumulator (vector, by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19 16	15	14	13	12	11	10	9 5	4 0
0	Q	1	0	1	1	1	1	size	L	M	Rm	0	0	0	0	H	0	Rn	Rd
													o2						

Vector variant

```
MLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `4H` when `size = 01`, `Q = 0`
- `8H` when `size = 01`, `Q = 1`
- `2S` when `size = 10`, `Q = 0`
- `4S` when `size = 10`, `Q = 1`

The following encodings are reserved:

- `size = 00`, `Q = x`.
- `size = 11`, `Q = x`.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:

- `0:Rm` when `size = 01`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`M:Rm` when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in the "size" field. It can have the following values:

`H` when `size = 01`

`S` when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

<index> Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

`H:L:M` when `size = 01`

`H:L` when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1*element2)<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.196 MLA (vector)

Multiply-Add to accumulator (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, and accumulates the results with the vector elements of the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	size	1	Rm	1	0	0	1	0	1	Rn	Rd
		U																

Three registers of the same type variant

```
MLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (U == '1');
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B	when `size = 00`, `Q = 0`
16B	when `size = 00`, `Q = 1`
4H	when `size = 01`, `Q = 0`
8H	when `size = 01`, `Q = 1`
2S	when `size = 10`, `Q = 0`
4S	when `size = 10`, `Q = 1`

The encoding `size = 11`, `Q = x` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    product = (UInt(element1)*UInt(element2))<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.197 MLS (by element)

Multiply-Subtract from accumulator (vector, by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and subtracts the results from the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19 16	15	14	13	12	11	10	9 5	4 0
0	Q	1	0	1	1	1	1	size	L	M	Rm	0	1	0	0	H	0	Rn	Rd
													o2						

Vector variant

```
MLS <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `4H` when `size = 01`, `Q = 0`
- `8H` when `size = 01`, `Q = 1`
- `2S` when `size = 10`, `Q = 0`
- `4S` when `size = 10`, `Q = 1`

The following encodings are reserved:

- `size = 00`, `Q = x`.
- `size = 11`, `Q = x`.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:

- `0:Rm` when `size = 01`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`M:Rm` when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in the "size" field. It can have the following values:

`H` when `size = 01`

`S` when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

<index> Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

`H:L:M` when `size = 01`

`H:L` when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1*element2)<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.198 MLS (vector)

Multiply-Subtract from accumulator (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size	1	Rm	1	0	0	1	0	1	Rn	Rd
		U																

Three registers of the same type variant

```
MLS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (U == '1');
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B	when `size = 00`, `Q = 0`
16B	when `size = 00`, `Q = 1`
4H	when `size = 01`, `Q = 0`
8H	when `size = 01`, `Q = 1`
2S	when `size = 10`, `Q = 0`
4S	when `size = 10`, `Q = 1`

The encoding `size = 11`, `Q = x` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    product = (UInt(element1)*UInt(element2))<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.199 MOV (scalar)

Move vector element to scalar. This instruction duplicates the specified vector element in the SIMD&FP source register into a scalar, and writes the result to the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is an alias of the DUP (element) instruction. This means that:

- The encodings in this description are named to match the encodings of DUP (element).
- The description of DUP (element) gives the operational pseudocode for this instruction.

31 30 29	28	27 26 25 24	23 22 21 20	16	15	14 13 12	11	10 9	5 4	0
0 1 0	1	1 1 1 0	0 0 0 0	imm5	0	0 0 0 0	1	Rn	Rd	

Scalar variant

```
MOV <V><d>, <Vn>.<T>[<index>]
```

is equivalent to

```
DUP  <V><d>, <Vn>.<T>[<index>]
```

and is always the preferred disassembly.

Assembler symbols

`<V>` Is the destination width specifier, encoded in the "imm5" field. It can have the following values:

- `B` when `imm5 = xxxx1`
- `H` when `imm5 = xxx10`
- `S` when `imm5 = xx100`
- `D` when `imm5 = x1000`

The encoding `imm5 = x0000` is reserved.

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<T>` Is the element width specifier, encoded in the "imm5" field. It can have the following values:

- `B` when `imm5 = xxxx1`
- `H` when `imm5 = xxx10`
- `S` when `imm5 = xx100`
- `D` when `imm5 = x1000`

The encoding `imm5 = x0000` is reserved.

`<index>` Is the element index encoded in the "imm5" field. It can have the following values:

- `imm5<4:1>` when `imm5 = xxxx1`
- `imm5<4:2>` when `imm5 = xxx10`
- `imm5<4:3>` when `imm5 = xx100`
- `imm5<4>` when `imm5 = x1000`

The encoding `imm5 = x0000` is reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

The description of DUP (element) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.200 MOV (element)

Move vector element to another vector element. This instruction copies the vector element of the source SIMD&FP register to the specified vector element of the destination SIMD&FP register.

This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is an alias of the INS (element) instruction. This means that:

- The encodings in this description are named to match the encodings of INS (element).
- The description of INS (element) gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	11	10	9	5	4	0
0	1	1	0	1	1	1	0	0	0	0	imm5		0	imm4		1	Rn		Rd	

Advanced SIMD variant

```
MOV <Vd>.<Ts>[<index1>], <Vn>.<Ts>[<index2>]
```

is equivalent to

```
INS  <Vd>.<Ts>[<index1>], <Vn>.<Ts>[<index2>]
```

and is always the preferred disassembly.

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ts>` Is an element size specifier, encoded in the "imm5" field. It can have the following values:

- `B` when `imm5 = xxxx1`
- `H` when `imm5 = xxx10`
- `S` when `imm5 = xx100`
- `D` when `imm5 = x1000`

The encoding `imm5 = x0000` is reserved.

`<index1>` Is the destination element index encoded in the "imm5" field. It can have the following values:

- `imm5<4:1>` when `imm5 = xxxx1`
- `imm5<4:2>` when `imm5 = xxx10`
- `imm5<4:3>` when `imm5 = xx100`
- `imm5<4>` when `imm5 = x1000`

The encoding `imm5 = x0000` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<index2>` Is the source element index encoded in the "imm5:imm4" field. It can have the following values:

- `imm4<3:0>` when `imm5 = xxxx1`
- `imm4<3:1>` when `imm5 = xxx10`
- `imm4<3:2>` when `imm5 = xx100`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`imm4<3>` when `imm5 = x1000`

The encoding `imm5 = x0000` is reserved.

Unspecified bits in "imm4" are ignored but should be set to zero by an assembler.

Operation

The description of INS (element) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.201 MOV (from general)

Move general-purpose register to a vector element. This instruction copies the contents of the source general-purpose register to the specified vector element in the destination SIMD&FP register.

This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is an alias of the INS (general) instruction. This means that:

- The encodings in this description are named to match the encodings of INS (general).
- The description of INS (general) gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	1	0	0	1	1	1	0	0	0	0	imm5		0	0	0	1	1	1	Rn		Rd	

Advanced SIMD variant

```
MOV <Vd>.<Ts>[<index>], <R><n>
```

is equivalent to

```
INS  <Vd>.<Ts>[<index>], <R><n>
```

and is always the preferred disassembly.

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ts>` Is an element size specifier, encoded in the "imm5" field. It can have the following values:

- `B` when `imm5 = xxxx1`
- `H` when `imm5 = xxx10`
- `S` when `imm5 = xx100`
- `D` when `imm5 = x1000`

The encoding `imm5 = x0000` is reserved.

`<index>` Is the element index encoded in the "imm5" field. It can have the following values:

- `imm5<4:1>` when `imm5 = xxxx1`
- `imm5<4:2>` when `imm5 = xxx10`
- `imm5<4:3>` when `imm5 = xx100`
- `imm5<4>` when `imm5 = x1000`

The encoding `imm5 = x0000` is reserved.

`<R>` Is the width specifier for the general-purpose source register, encoded in the "imm5" field. It can have the following values:

- `W` when `imm5 = xxxx1`
- `W` when `imm5 = xxx10`
- `W` when `imm5 = xx100`
- `X` when `imm5 = x1000`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The encoding `imm5 = x0000` is reserved.

`<n>` Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the "Rn" field.

Operation

The description of INS (general) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.202 MOV (vector)

Move vector. This instruction copies the vector in the source SIMD&FP register into the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is an alias of the ORR (vector, register) instruction. This means that:

- The encodings in this description are named to match the encodings of ORR (vector, register).
- The description of ORR (vector, register) gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20–16	15	14	13	12	11	10	9–5	4–0
0	Q	0	0	1	1	1	0	1	0	1	Rm	0	0	0	1	1	1	Rn	Rd
								size											

Three registers of the same type variant

```
MOV <Vd>.<T>, <Vn>.<T>
```

is equivalent to

```
ORR  <Vd>.<T>, <Vn>.<T>, <Vn>.<T>
```

and is the preferred disassembly when `Rm == Rn`.

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`8B`	when `Q = 0`
`16B`	when `Q = 1`

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Operation

The description of ORR (vector, register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.203 MOV (to general)

Move vector element to general-purpose register. This instruction reads the unsigned integer from the source SIMD&FP register, zero-extends it to form a 32-bit or 64-bit value, and writes the result to the destination general-purpose register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is an alias of the UMOV instruction. This means that:

- The encodings in this description are named to match the encodings of UMOV.
- The description of UMOV gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	0	0	x	x x x 0 0	0	0	1	1	1	1	Rn		Rd	
												imm5										

32-bit variant

Applies when `Q == 0 && imm5 == xx100`.

```
MOV <Wd>, <Vn>.S[<index>]
```

is equivalent to

```
UMOV <Wd>, <Vn>.S[<index>]
```

and is always the preferred disassembly.

64-reg,UMOV-64-reg variant

Applies when `Q == 1 && imm5 == x1000`.

```
MOV <Xd>, <Vn>.D[<index>]
```

is equivalent to

```
UMOV <Xd>, <Vn>.D[<index>]
```

and is always the preferred disassembly.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Vn>`	Is the name of the SIMD&FP source register, encoded in the "Rn" field.
`<index>`	For the 32-bit variant: is the element index encoded in "imm5<4:3>". For the 64-reg,UMOV-64-reg variant: is the element index encoded in "imm5<4>".

Operation

The description of UMOV gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.204 MOVI

Move Immediate (vector). This instruction places an immediate constant into every vector element of the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	12	11	10	9	8	7	6	5	4	0
0	Q	op	0	1	1	1	1	0	0	0	0	0	a	b	c	cmode		0	1	d	e	f	g	h	Rd	

8-bit variant

Applies when `op == 0 && cmode == 1110`.

```
MOVI <Vd>.<T>, #<imm8>{, LSL #0}
```

16-bit shifted immediate variant

Applies when `op == 0 && cmode == 10x0`.

```
MOVI <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

32-bit shifted immediate variant

Applies when `op == 0 && cmode == 0xx0`.

```
MOVI <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

32-bit shifting ones variant

Applies when `op == 0 && cmode == 110x`.

```
MOVI <Vd>.<T>, #<imm8>, MSL #<amount>
```

64-bit scalar variant

Applies when `Q == 0 && op == 1 && cmode == 1110`.

```
MOVI <Dd>, #<imm>
```

64-bit vector variant

Applies when `Q == 1 && op == 1 && cmode == 1110`.

```
MOVI <Vd>.2D, #<imm>
```

Decode for all variants of this encoding

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
    when '0xx00' operation = ImmediateOp_MOVI;
    when '0xx01' operation = ImmediateOp_MVNI;
    when '0xx10' operation = ImmediateOp_ORR;
    when '0xx11' operation = ImmediateOp_BIC;
    when '10x00' operation = ImmediateOp_MOVI;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    when '10x01' operation = ImmediateOp_MVNI;
    when '10x10' operation = ImmediateOp_ORR;
    when '10x11' operation = ImmediateOp_BIC;
    when '110x0' operation = ImmediateOp_MOVI;
    when '110x1' operation = ImmediateOp_MVNI;
    when '1110x' operation = ImmediateOp_MOVI;
    when '11110' operation = ImmediateOp_MOVI;
    when '11111'
        // FMOV Dn,#imm is in main FP instruction set
        if Q == '0' then UNDEFINED;
        operation = ImmediateOp_MOVI;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

Assembler symbols

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<imm>` Is a 64-bit immediate 'aaaaaaaabbbbbbbbcccccccddddddddeeeeeeeeffffffffggggggggghhhhhhhh', encoded in "a:b:c:d:e:f:g:h".

`<T>` For the 8-bit variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`8B` when `Q = 0`

`16B` when `Q = 1`

For the 16-bit variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`4H` when `Q = 0`

`8H` when `Q = 1`

For the 32-bit variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`2S` when `Q = 0`

`4S` when `Q = 1`

`<imm8>` Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

`<amount>` For the 16-bit shifted immediate variant: is the shift amount encoded in the "cmode<1>" field. It can have the following values:

`0` when `cmode<1> = 0`

`8` when `cmode<1> = 1`

defaulting to 0 if LSL is omitted.

For the 32-bit shifted immediate variant: is the shift amount encoded in the "cmode<2:1>" field. It can have the following values:

`0` when `cmode<2:1> = 00`

`8` when `cmode<2:1> = 01`

`16` when `cmode<2:1> = 10`

`24` when `cmode<2:1> = 11`

defaulting to 0 if LSL is omitted.

For the 32-bit shifting ones variant: is the shift amount encoded in the "cmode<0>" field. It can have the following values:

`8` when `cmode<0> = 0`

`16` when `cmode<0> = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
    when ImmediateOp_MOVI
        result = imm;
    when ImmediateOp_MVNI
        result = NOT(imm);
    when ImmediateOp_ORR
        operand = V[rd];
        result = operand OR imm;
    when ImmediateOp_BIC
        operand = V[rd];
        result = operand AND NOT(imm);

V[rd] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.205 MUL (by element)

Multiply (vector, by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19 16	15	14	13	12	11	10	9 5	4 0
0	Q	0	0	1	1	1	1	size	L	M	Rm	1	0	0	0	H	0	Rn	Rd

Vector variant

```
MUL <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `4H` when `size = 01`, `Q = 0`
- `8H` when `size = 01`, `Q = 1`
- `2S` when `size = 10`, `Q = 0`
- `4S` when `size = 10`, `Q = 1`

The following encodings are reserved:

- `size = 00`, `Q = x`.
- `size = 11`, `Q = x`.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:

- `0:Rm` when `size = 01`
- `M:Rm` when `size = 10`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in the "size" field. It can have the following values:

`H` when `size = 01`

`S` when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

<index> Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

`H:L:M` when `size = 01`

`H:L` when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1*element2)<esize-1:0>;
    Elem[result, e, esize] = product;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.206 MUL (vector)

Multiply (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 16	15	14	13	12	11	10	9 5	4 0
0	Q	0	0	1	1	1	0	size	1	Rm	1	0	0	1	1	1	Rn	Rd
		U																

Three registers of the same type variant

```
MUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if U == '1' && size != '00' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean poly = (U == '1');
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B	when size = 00, Q = 0
16B	when size = 00, Q = 1
4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;

for e = 0 to elements-1
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if poly then
        product = PolynomialMult(element1, element2)<esize-1:0>;
    else
        product = (UInt(element1)*UInt(element2))<esize-1:0>;
    Elem[result, e, esize] = product;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.207 MVN

Bitwise NOT (vector). This instruction reads each vector element from the source SIMD&FP register, places the inverse of each value into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is an alias of the NOT instruction. This means that:

- The encodings in this description are named to match the encodings of NOT.
- The description of NOT gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	0	0	1	0	0	0	0	0	0	1	0	1	1	0	Rn	Rd

Vector variant

```
MVN <Vd>.<T>, <Vn>.<T>
```

is equivalent to

```
NOT  <Vd>.<T>, <Vn>.<T>
```

and is always the preferred disassembly.

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`8B`	when `Q = 0`
`16B`	when `Q = 1`

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

The description of NOT gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.208 MVNI

Move inverted Immediate (vector). This instruction places the inverse of an immediate constant into every vector element of the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15 – 12	11	10	9	8	7	6	5	4	0 – 4
0	Q	1	0	1	1	1	1	0	0	0	0	0	a	b	c	cmode	0	1	d	e	f	g	h	Rd	
		op																							

16-bit shifted immediate variant

Applies when `cmode == 10x0`.

```
MVNI <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

32-bit shifted immediate variant

Applies when `cmode == 0xx0`.

```
MVNI <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

32-bit shifting ones variant

Applies when `cmode == 110x`.

```
MVNI <Vd>.<T>, #<imm8>, MSL #<amount>
```

Decode for all variants of this encoding

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
    when '0xx01' operation = ImmediateOp_MVNI;
    when '0xx11' operation = ImmediateOp_BIC;
    when '10x01' operation = ImmediateOp_MVNI;
    when '10x11' operation = ImmediateOp_BIC;
    when '110x1' operation = ImmediateOp_MVNI;
    when '1110x' operation = ImmediateOp_MOVI;
    when '11111'
        // FMOV Dn,#imm is in main FP instruction set
        if Q == '0' then UNDEFINED;
        operation = ImmediateOp_MOVI;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the 16-bit variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`4H` when `Q = 0`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

8H when Q = 1

For the 32-bit variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

2S when Q = 0

4S when Q = 1

<imm8> Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

<amount> For the 16-bit shifted immediate variant: is the shift amount encoded in the "cmode<1>" field. It can have the following values:

0 when cmode<1> = 0

8 when cmode<1> = 1

defaulting to 0 if LSL is omitted.

For the 32-bit shifted immediate variant: is the shift amount encoded in the "cmode<2:1>" field. It can have the following values:

0 when cmode<2:1> = 00

8 when cmode<2:1> = 01

16 when cmode<2:1> = 10

24 when cmode<2:1> = 11

defaulting to 0 if LSL is omitted.

For the 32-bit shifting ones variant: is the shift amount encoded in the "cmode<0>" field. It can have the following values:

8 when cmode<0> = 0

16 when cmode<0> = 1

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
    when ImmediateOp_MOVI
        result = imm;
    when ImmediateOp_MVNI
        result = NOT(imm);
    when ImmediateOp_ORR
        operand = V[rd];
        result = operand OR imm;
    when ImmediateOp_BIC
        operand = V[rd];
        result = operand AND NOT(imm);

V[rd] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.209 NEG (vector)

Negate (vector). This instruction reads each vector element from the source SIMD&FP register, negates each value, puts the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23 22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	1	1	1	1	1	0	size	1	0	0	0	0	0	1	0	1	1	1	0	Rn	Rd
		U																				

Scalar variant

```
NEG <V><d>, <V><n>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean neg = (U == '1');
```

Vector

31	30	29	28	27	26	25	24	23 22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size	1	0	0	0	0	0	1	0	1	1	1	0	Rn	Rd
		U																				

Vector variant

```
NEG <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

D when size = 11

The following encodings are reserved:

- size = 0x.
- size = 10.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

`<n>` Is the number of the SIMD&FP source register, encoded in the "Rn" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B`	when `size = 00`, `Q = 0`
`16B`	when `size = 00`, `Q = 1`
`4H`	when `size = 01`, `Q = 0`
`8H`	when `size = 01`, `Q = 1`
`2S`	when `size = 10`, `Q = 0`
`4S`	when `size = 10`, `Q = 1`
`2D`	when `size = 11`, `Q = 1`

The encoding `size = 11`, `Q = 0` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);
    Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.210 NOT

Bitwise NOT (vector). This instruction reads each vector element from the source SIMD&FP register, places the inverse of each value into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MVN. The alias is always the preferred disassembly.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	0	0	1	0	0	0	0	0	0	1	0	1	1	0	Rn		Rd	

Vector variant

```
NOT <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `8B` when `Q = 0`
- `16B` when `Q = 1`

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = NOT(element);

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.211 ORN (vector)

Bitwise inclusive OR NOT (vector). This instruction performs a bitwise OR NOT between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	1	1	1	Rm	0	0	0	1	1	1	Rn	Rd
								size											

Three registers of the same type variant

```
ORN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `8B` when `Q = 0`
- `16B` when `Q = 1`

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

operand2 = NOT(operand2);

result = operand1 OR operand2;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.212 ORR (vector, immediate)

Bitwise inclusive OR (vector, immediate). This instruction reads each vector element from the destination SIMD&FP register, performs a bitwise OR between each result and an immediate constant, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	0	0	0	0	a	b	c	x	x	x	1	0	1	d	e	f	g	h	Rd				
		op														cmode															

16-bit variant

Applies when `cmode == 10x1`.

```
ORR <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

32-bit variant

Applies when `cmode == 0xx1`.

```
ORR <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

Decode for all variants of this encoding

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
    when '0xx00' operation = ImmediateOp_MOVI;
    when '0xx10' operation = ImmediateOp_ORR;
    when '10x00' operation = ImmediateOp_MOVI;
    when '10x10' operation = ImmediateOp_ORR;
    when '110x0' operation = ImmediateOp_MOVI;
    when '1110x' operation = ImmediateOp_MOVI;
    when '11110' operation = ImmediateOp_MOVI;
imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP register, encoded in the "Rd" field.

`<T>` For the 16-bit variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `4H` when `Q = 0`
- `8H` when `Q = 1`

For the 32-bit variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `2S` when `Q = 0`
- `4S` when `Q = 1`

`<imm8>` Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<amount>` For the 16-bit variant: is the shift amount encoded in the "cmode<1>" field. It can have the following values:

`0` when `cmode<1> = 0`

`8` when `cmode<1> = 1`

defaulting to 0 if LSL is omitted.

For the 32-bit variant: is the shift amount encoded in the "cmode<2:1>" field. It can have the following values:

`0` when `cmode<2:1> = 00`

`8` when `cmode<2:1> = 01`

`16` when `cmode<2:1> = 10`

`24` when `cmode<2:1> = 11`

defaulting to 0 if LSL is omitted.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
    when ImmediateOp_MOVI
        result = imm;
    when ImmediateOp_MVNI
        result = NOT(imm);
    when ImmediateOp_ORR
        operand = V[rd];
        result = operand OR imm;
    when ImmediateOp_BIC
        operand = V[rd];
        result = operand AND NOT(imm);

V[rd] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.213 ORR (vector, register)

Bitwise inclusive OR (vector, register). This instruction performs a bitwise OR between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (vector). See *Alias conditions* for details of when each alias is preferred.

31	30	29	28	27	26	25	24	23	22	21	20–16	15	14	13	12	11	10	9–5	4–0
0	Q	0	0	1	1	1	0	1	0	1	Rm	0	0	0	1	1	1	Rn	Rd
								size											

Three registers of the same type variant

```
ORR <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
```

Alias conditions

Alias	is preferred when
MOV (vector)	`Rm == Rn`

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `8B` when `Q = 0`
- `16B` when `Q = 1`

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

result = operand1 OR operand2;

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.214 PMUL

Polynomial Multiply. This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

For information about multiplying polynomials see *Polynomial arithmetic over {0, 1}* on page A1-50.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size	1	Rm	1	0	0	1	1	1	Rn	Rd
		U																

Three registers of the same type variant

```
PMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if U == '1' && size != '00' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean poly = (U == '1');
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B` when `size = 00`, `Q = 0`

`16B` when `size = 00`, `Q = 1`

The following encodings are reserved:

- `size = 01`, `Q = x`.
- `size = 1x`, `Q = x`.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    element2 = Elem[operand2, e, esize];
    if poly then
        product = PolynomialMult(element1, element2)<esize-1:0>;
    else
        product = (UInt(element1)*UInt(element2))<esize-1:0>;
    Elem[result, e, esize] = product;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.215 PMULL, PMULL2

Polynomial Multiply Long. This instruction multiplies corresponding elements in the lower or upper half of the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

For information about multiplying polynomials see *Polynomial arithmetic over {0, 1}* on page A1-50.

The `PMULL` instruction extracts each source vector from the lower half of each source register, while the `PMULL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 16	15	14	13	12	11	10	9 5	4 0
0	Q	0	0	1	1	1	0	size	1	Rm	1	1	1	0	0	0	Rn	Rd

Three registers, not all the same type variant

```
PMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '01' || size == '10' then UNDEFINED;
if size == '11' && !HaveBit128PMULLExt() then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

`8H` when `size = 00`

`1Q` when `size = 11`

The following encodings are reserved:

- `size = 01`.
- `size = 10`.

The '1Q' arrangement is only allocated in an implementation that includes the Cryptographic Extension, and is otherwise RESERVED.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

1D when size = 11, Q = 0

2D when size = 11, Q = 1

The following encodings are reserved:

- size = 01, Q = x.
- size = 10, Q = x.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, 2*esize] = PolynomialMult(element1, element2);

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.216 RADDHN, RADDHN2

Rounding Add returning High Narrow. This instruction adds each vector element in the first source SIMD&FP register to the corresponding vector element in the second source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register.

The results are rounded. For truncated results, see ADDHN, ADDHN2.

The `RADDHN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `RADDHN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 16	15	14	13	12	11	10	9 5	4 0
0	Q	1	0	1	1	1	0	size	1	Rm	0	1	0	0	0	0	Rn	Rd
		U											o1					

Three registers, not all the same type variant

```
RADDHN{2} <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Tb>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B` when `size = 00`, `Q = 0`

`16B` when `size = 00`, `Q = 1`

`4H` when `size = 01`, `Q = 0`

`8H` when `size = 01`, `Q = 1`

`2S` when `size = 10`, `Q = 0`

`4S` when `size = 10`, `Q = 1`

The encoding `size = 11`, `Q = x` is reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

`8H` when `size = 00`

`4S` when `size = 01`

`2D` when `size = 10`

The encoding `size = 11` is reserved.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(2*datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
bits(2*esize) element1;
bits(2*esize) element2;
bits(2*esize) sum;

for e = 0 to elements-1
    element1 = Elem[operand1, e, 2*esize];
    element2 = Elem[operand2, e, 2*esize];
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    sum = sum + round_const;
    Elem[result, e, esize] = sum<2*esize-1:esize>;

Vpart[d, part] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.217 RAX1

Rotate and Exclusive OR rotates each 64-bit element of the 128-bit vector in a source SIMD&FP register left by 1, performs a bitwise exclusive OR of the resulting 128-bit vector and the vector in another source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction is implemented only when ARMv8.2-SHA is implemented.

ARMv8.2

31 30 29 28	27 26 25 24	23 22 21 20	16	15	14	13 12	11 10	9 5	4 0
1 1 0 0	1 1 1 0	0 1 1	Rm	1	0	0 0	1 1	Rn	Rd

Advanced SIMD variant

```
RAX1 <Vd>.2D, <Vn>.2D, <Vm>.2D
```

Decode for this encoding

```
if !HaveSHA3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler symbols

`<Vd>`	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Vn>`	Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
`<Vm>`	Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
V[d] = Vn EOR (ROL(Vm<127:64>, 1):ROL(Vm<63:0>, 1));
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.218 RBIT (vector)

Reverse Bit order (vector). This instruction reads each vector element from the source SIMD&FP register, reverses the bits of the element, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	0	1	1	0	0	0	0	0	0	0	1	0	1	1	0	Rn	Rd	

Vector variant

```
RBIT <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- 8B when Q = 0
- 16B when Q = 1

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;
bits(esize) rev;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    for i = 0 to esize-1
        rev<esize-1-i> = element<i>;
    Elem[result, e, esize] = rev;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.219 REV16 (vector)

Reverse elements in 16-bit halfwords (vector). This instruction reverses the order of 8-bit elements in each halfword of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	17	16	15	14	13	12	11	10	9 5	4 0
0	Q	0	0	1	1	1	0	size	1	0	0	0	0	0	0	0	0	1	1	0	Rn	Rd
		U																o0				

Vector variant

```
REV16 <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

// size=esize:   B(0),  H(1),  S(1), D(S)
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;

// op=REVx: 64(0), 32(1), 16(2)
bits(2) op = o0:U;

// => op+size:
//    64+B = 0, 64+H = 1, 64+S = 2, 64+D = X
//    32+B = 1, 32+H = 2, 32+S = X, 32+D = X
//    16+B = 2, 16+H = X, 16+S = X, 16+D = X
//     8+B = X,  8+H = X,  8+S = X,  8+D = X
// => 3-(op+size) (index bits in group)
//    64/B = 3, 64+H = 2, 64+S = 1, 64+D = X
//    32+B = 2, 32+H = 1, 32+S = X, 32+D = X
//    16+B = 1, 16+H = X, 16+S = X, 16+D = X
//     8+B = X,  8+H = X,  8+S = X,  8+D = X

// index bits within group: 1, 2, 3
if UInt(op) + UInt(size) >= 3 then UNDEFINED;

integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when `size = 00`, `Q = 0`

16B when `size = 00`, `Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following encodings are reserved:

- `size = 01, Q = x`.
- `size = 1x, Q = x`.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element = 0;
integer rev_element;
for c = 0 to containers-1
    rev_element = element + elements_per_container - 1;
    for e = 0 to elements_per_container-1
        Elem[result, rev_element, esize] = Elem[operand, element, esize];
        element = element + 1;
        rev_element = rev_element - 1;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.220 REV32 (vector)

Reverse elements in 32-bit words (vector). This instruction reverses the order of 8-bit or 16-bit elements in each word of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	17	16	15	14	13	12	11	10	9 5	4 0
0	Q	1	0	1	1	1	0	size	1	0	0	0	0	0	0	0	0	0	1	0	Rn	Rd
		U																o0				

Vector variant

```
REV32 <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

// size=esize:   B(0),  H(1),  S(1), D(S)
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;

// op=REVx: 64(0), 32(1), 16(2)
bits(2) op = o0:U;

// => op+size:
//    64+B = 0, 64+H = 1, 64+S = 2, 64+D = X
//    32+B = 1, 32+H = 2, 32+S = X, 32+D = X
//    16+B = 2, 16+H = X, 16+S = X, 16+D = X
//     8+B = X,  8+H = X,  8+S = X,  8+D = X
// => 3-(op+size) (index bits in group)
//    64/B = 3, 64+H = 2, 64+S = 1, 64+D = X
//    32+B = 2, 32+H = 1, 32+S = X, 32+D = X
//    16+B = 1, 16+H = X, 16+S = X, 16+D = X
//     8+B = X,  8+H = X,  8+S = X,  8+D = X

// index bits within group: 1, 2, 3
if UInt(op) + UInt(size) >= 3 then UNDEFINED;

integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`8H` when `size = 01`, `Q = 1`

The encoding `size = 1x`, `Q = x` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element = 0;
integer rev_element;
for c = 0 to containers-1
    rev_element = element + elements_per_container - 1;
    for e = 0 to elements_per_container-1
        Elem[result, rev_element, esize] = Elem[operand, element, esize];
        element = element + 1;
        rev_element = rev_element - 1;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.221 REV64

Reverse elements in 64-bit doublewords (vector). This instruction reverses the order of 8-bit, 16-bit, or 32-bit elements in each doubleword of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	17	16	15	14	13	12	11	10	9 5	4 0
0	Q	0	0	1	1	1	0	size	1	0	0	0	0	0	0	0	0	0	1	0	Rn	Rd
		U																o0				

Vector variant

```
REV64 <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

// size=esize:   B(0),  H(1),  S(1), D(S)
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;

// op=REVx: 64(0), 32(1), 16(2)
bits(2) op = o0:U;

// => op+size:
//    64+B = 0, 64+H = 1, 64+S = 2, 64+D = X
//    32+B = 1, 32+H = 2, 32+S = X, 32+D = X
//    16+B = 2, 16+H = X, 16+S = X, 16+D = X
//     8+B = X,  8+H = X,  8+S = X,  8+D = X
// => 3-(op+size) (index bits in group)
//    64/B = 3, 64+H = 2, 64+S = 1, 64+D = X
//    32+B = 2, 32+H = 1, 32+S = X, 32+D = X
//    16+B = 1, 16+H = X, 16+S = X, 16+D = X
//     8+B = X,  8+H = X,  8+S = X,  8+D = X

// index bits within group: 1, 2, 3
if UInt(op) + UInt(size) >= 3 then UNDEFINED;

integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element = 0;
integer rev_element;
for c = 0 to containers-1
    rev_element = element + elements_per_container - 1;
    for e = 0 to elements_per_container-1
        Elem[result, rev_element, esize] = Elem[operand, element, esize];
        element = element + 1;
        rev_element = rev_element - 1;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.222 RSHRN, RSHRN2

Rounding Shift Right Narrow (immediate). This instruction reads each unsigned integer value from the vector in the source SIMD&FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. The results are rounded. For truncated results, see SHRN, SHRN2.

The RSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	19	18	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	1	0	!=0000	immb			1	0	0	0	1	1	Rn		Rd	
									immh								op					

Vector variant

```
RSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

[absent] when Q = 0

[present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B when immh = 0001, Q = 0

16B when immh = 0001, Q = 1

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

See *Advanced SIMD modified immediate* on page C4-354 when immh = 0000, Q = x.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The encoding `immh = 1xxx`, `Q = x` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<Ta>` Is an arrangement specifier, encoded in the "immh" field. It can have the following values:

`8H` when `immh = 0001`

`4S` when `immh = 001x`

`2D` when `immh = 01xx`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`.

The encoding `immh = 1xxx` is reserved.

`<shift>` Is the right shift amount, in the range 1 to the destination element width in bits, encoded in the "immh:immb" field. It can have the following values:

`(16-UInt(immh:immb))` when `immh = 0001`

`(32-UInt(immh:immb))` when `immh = 001x`

`(64-UInt(immh:immb))` when `immh = 01xx`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`.

The encoding `immh = 1xxx` is reserved.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

for e = 0 to elements-1
    element = (UInt(Elem[operand, e, 2*esize]) + round_const) >> shift;
    Elem[result, e, esize] = element<esize-1:0>;

Vpart[d, part] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.223 RSUBHN, RSUBHN2

Rounding Subtract returning High Narrow. This instruction subtracts each vector element of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register.

The results are rounded. For truncated results, see SUBHN, SUBHN2.

The `RSUBHN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `RSUBHN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size	1	Rm	0	1	1	0	0	0	Rn	Rd
		U											o1					

Three registers, not all the same type variant

```
RSUBHN{2} <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- `[absent]` when `Q = 0`
- `[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Tb>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `8B` when `size = 00, Q = 0`
- `16B` when `size = 00, Q = 1`
- `4H` when `size = 01, Q = 0`
- `8H` when `size = 01, Q = 1`
- `2S` when `size = 10, Q = 0`
- `4S` when `size = 10, Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The encoding `size = 11`, `Q = x` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

`8H` when `size = 00`

`4S` when `size = 01`

`2D` when `size = 10`

The encoding `size = 11` is reserved.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(2*datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
bits(2*esize) element1;
bits(2*esize) element2;
bits(2*esize) sum;

for e = 0 to elements-1
    element1 = Elem[operand1, e, 2*esize];
    element2 = Elem[operand2, e, 2*esize];
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    sum = sum + round_const;
    Elem[result, e, esize] = sum<2*esize-1:esize>;

Vpart[d, part] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.224 SABA

Signed Absolute difference and Accumulate. This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the elements of the vector of the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	size	1	Rm	0	1	1	1	1	1	Rn	Rd
		U													ac			

Three registers of the same type variant

```
SABA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B	when size = 00, Q = 0
16B	when size = 00, Q = 1
4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.225 SABAL, SABAL2

Signed Absolute difference and Accumulate Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The `SABAL` instruction extracts each source vector from the lower half of each source register, while the `SABAL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	size	1	Rm	0	1	0	1	0	0	Rn	Rd
		U											op					

Three registers, not all the same type variant

```
SABAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- `[absent]` when `Q = 0`
- `[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- `8H` when `size = 00`
- `4S` when `size = 01`
- `2D` when `size = 10`

The encoding `size = 11` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `8B` when `size = 00`, `Q = 0`
- `16B` when `size = 00`, `Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.226 SABD

Signed Absolute Difference. This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, places the absolute values of the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	size	1	Rm	0	1	1	1	0	1	Rn	Rd
		U													ac			

Three registers of the same type variant

```
SABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B	when size = 00, Q = 0
16B	when size = 00, Q = 1
4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.227 SABDL, SABDL2

Signed Absolute Difference Long. This instruction subtracts the vector elements of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, places the absolute value of the results into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The `SABDL` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SABDL2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	size	1	Rm	0	1	1	1	0	0	Rn	Rd
		U											op					

Three registers, not all the same type variant

```
SABDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

`8H` when `size = 00`

`4S` when `size = 01`

`2D` when `size = 10`

The encoding `size = 11` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B` when `size = 00`, `Q = 0`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

16B	when size = 00, Q = 1
4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.228 SADALP

Signed Add and Accumulate Long Pairwise. This instruction adds pairs of adjacent signed integer values from the vector in the source SIMD&FP register and accumulates the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	size	1	0	0	0	0	0	0	1	1	0	1	0	Rn	Rd
		U														op						

Vector variant

```
SADALP <Vd>.<Ta>, <Vn>.<Tb>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2 * esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `4H` when `size = 00`, `Q = 0`
- `8H` when `size = 00`, `Q = 1`
- `2S` when `size = 01`, `Q = 0`
- `4S` when `size = 01`, `Q = 1`
- `1D` when `size = 10`, `Q = 0`
- `2D` when `size = 10`, `Q = 1`

The encoding `size = 11`, `Q = x` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `8B` when `size = 00`, `Q = 0`
- `16B` when `size = 00`, `Q = 1`
- `4H` when `size = 01`, `Q = 0`
- `8H` when `size = 01`, `Q = 1`
- `2S` when `size = 10`, `Q = 0`
- `4S` when `size = 10`, `Q = 1`

The encoding `size = 11`, `Q = x` is reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

if acc then result = V[d];
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1+op2)<2*esize-1:0>;
    if acc then
        Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;
    else
        Elem[result, e, 2*esize] = sum;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.229 SADDL, SADDL2

Signed Add Long (vector). This instruction adds each vector element in the lower or upper half of the first source SIMD&FP register to the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The `SADDL` instruction extracts each source vector from the lower half of each source register, while the `SADDL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	size	1	Rm	0	0	0	0	0	0	Rn	Rd
		U											o1					

Three registers, not all the same type variant

```
SADDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- `[absent]` when `Q = 0`
- `[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- `8H` when `size = 00`
- `4S` when `size = 01`
- `2D` when `size = 10`

The encoding `size = 11` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `8B` when `size = 00`, `Q = 0`
- `16B` when `size = 00`, `Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.230 SADDLP

Signed Add Long Pairwise. This instruction adds pairs of adjacent signed integer values from the vector in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	size	1	0	0	0	0	0	0	0	1	0	1	0	Rn	Rd
		U														op						

Vector variant

```
SADDLP <Vd>.<Ta>, <Vn>.<Tb>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2 * esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `4H` when `size = 00`, `Q = 0`
- `8H` when `size = 00`, `Q = 1`
- `2S` when `size = 01`, `Q = 0`
- `4S` when `size = 01`, `Q = 1`
- `1D` when `size = 10`, `Q = 0`
- `2D` when `size = 10`, `Q = 1`

The encoding `size = 11`, `Q = x` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `8B` when `size = 00`, `Q = 0`
- `16B` when `size = 00`, `Q = 1`
- `4H` when `size = 01`, `Q = 0`
- `8H` when `size = 01`, `Q = 1`
- `2S` when `size = 10`, `Q = 0`
- `4S` when `size = 10`, `Q = 1`

The encoding `size = 11`, `Q = x` is reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

if acc then result = V[d];
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1+op2)<2*esize-1:0>;
    if acc then
        Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;
    else
        Elem[result, e, 2*esize] = sum;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.231 SADDLV

Signed Add Long across Vector. This instruction adds every vector element in the source SIMD&FP register together, and writes the scalar result to the destination SIMD&FP register. The destination scalar is twice as long as the source vector elements. All the values in this instruction are signed integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	size		1	1	0	0	0	0	0	0	1	1	1	0	Rn		Rd	
		U																							

Advanced SIMD variant

```
SADDLV <V><d>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler symbols

<V> Is the destination width specifier, encoded in the "size" field. It can have the following values:

- H when `size = 00`
- S when `size = 01`
- D when `size = 10`

The encoding `size = 11` is reserved.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when `size = 00, Q = 0`
- 16B when `size = 00, Q = 1`
- 4H when `size = 01, Q = 0`
- 8H when `size = 01, Q = 1`
- 4S when `size = 10, Q = 1`

The following encodings are reserved:

- `size = 10, Q = 0`.
- `size = 11, Q = x`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer sum;

sum = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    sum = sum + Int(Elem[operand, e, esize], unsigned);

V[d] = sum<2*esize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.232 SADDW, SADDW2

Signed Add Wide. This instruction adds vector elements of the first source SIMD&FP register to the corresponding vector elements in the lower or upper half of the second source SIMD&FP register, places the results in a vector, and writes the vector to the SIMD&FP destination register.

The `SADDW` instruction extracts the second source vector from the lower half of the second source register, while the `SADDW2` instruction extracts the second source vector from the upper half of the second source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	size	1	Rm	0	0	0	1	0	0	Rn	Rd
		U											o1					

Three registers, not all the same type variant

```
SADDW{2} <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- `[absent]` when `Q = 0`
- `[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- `8H` when `size = 00`
- `4S` when `size = 01`
- `2D` when `size = 10`

The encoding `size = 11` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

`<Tb>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `8B` when `size = 00`, `Q = 0`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

16B	when size = 00, Q = 1
4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, 2*esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.233 SCVTF (vector, fixed-point)

Signed fixed-point Convert to Floating-point (vector). This instruction converts each element in a vector from fixed-point to floating-point using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23	22	19	18	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	1	0	!=0000	immb			1	1	1	0	0	1	Rn		Rd	
		U							immh													

Scalar variant

```
SCVTF <V><d>, <V><n>, #<fbits>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then UNDEFINED;
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR);
```

Vector

31	30	29	28	27	26	25	24	23	22	19	18	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	1	0	!=0000	immb			1	1	1	0	0	1	Rn		Rd	
		U							immh													

Vector variant

```
SCVTF <Vd>.<T>, <Vn>.<T>, #<fbits>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then UNDEFINED;
if immh<3>:Q == '10' then UNDEFINED;
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR);
```

Assembler symbols

<V> Is a width specifier, encoded in the "immh" field. It can have the following values:

H when immh = 001x

S when immh = 01xx

D when immh = 1xxx

The encoding immh = 000x is reserved.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

2D when immh = 1xxx, Q = 1

See *Advanced SIMD modified immediate* on page C4-354 when immh = 0000, Q = x.

The following encodings are reserved:

- immh = 0001, Q = x.
- immh = 1xxx, Q = 0.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits> For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in the "immh:immb" field. It can have the following values:

(32-Uint(immh:immb)) when immh = 001x

(64-UInt(immh:immb)) when immh = 01xx

(128-UInt(immh:immb)) when immh = 1xxx

The encoding immh = 000x is reserved.

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in the "immh:immb" field. It can have the following values:

(32-Uint(immh:immb)) when immh = 001x

(64-UInt(immh:immb)) when immh = 01xx

(128-UInt(immh:immb)) when immh = 1xxx

See *Advanced SIMD modified immediate* on page C4-354 when immh = 0000.

The encoding immh = 0001 is reserved.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, fracbits, unsigned, FPCR, rounding);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.234 SCVTF (vector, integer)

Signed integer Convert to Floating-point (vector). This instruction converts each element in a vector from signed integer to floating-point using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Scalar half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn		Rd	
		U																							

Scalar half precision variant

```
SCVTF <Hd>, <Hn>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	1	0	1	1	0	Rn		Rd	
		U																							

Scalar single-precision and double-precision variant

```
SCVTF <V><d>, <V><n>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Vector half precision

ARMv8.2

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn		Rd	
		U																							

Vector half precision variant

```
SCVTF <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	1	0	1	1	0	Rn		Rd	
		U																							

Vector single-precision and double-precision variant

```
SCVTF <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler symbols

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

`<V>` Is a width specifier, encoded in the "sz" field. It can have the following values:

- `S` when `sz = 0`
- `D` when `sz = 1`

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

`<n>` Is the number of the SIMD&FP source register, encoded in the "Rn" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the vector half precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `4H` when `Q = 0`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`8H` when `Q = 1`

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

`2S` when `sz = 0`, `Q = 0`

`4S` when `sz = 0`, `Q = 1`

`2D` when `sz = 1`, `Q = 1`

The encoding `sz = 1`, `Q = 0` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
FPRounding rounding = FPRoundingMode(FPCR);
bits(esize) element;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.235 SCVTF (scalar, fixed-point)

Signed fixed-point Convert to Floating-point (scalar). This instruction converts the signed value in the 32-bit or 64-bit general-purpose source register to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	16	15 10	9 5	4 0
sf	0	0	1	1	1	1	0	ftype	0	0	0	1	0	scale	Rn	Rd
										rmode		opcode				

32-bit to half-precision variant

Applies when `sf == 0 && ftype == 11`.

```
SCVTF <Hd>, <Wn>, #<fbits>
```

32-bit to single-precision variant

Applies when `sf == 0 && ftype == 00`.

```
SCVTF <Sd>, <Wn>, #<fbits>
```

32-bit to double-precision variant

Applies when `sf == 0 && ftype == 01`.

```
SCVTF <Dd>, <Wn>, #<fbits>
```

64-bit to half-precision variant

Applies when `sf == 1 && ftype == 11`.

```
SCVTF <Hd>, <Xn>, #<fbits>
```

64-bit to single-precision variant

Applies when `sf == 1 && ftype == 00`.

```
SCVTF <Sd>, <Xn>, #<fbits>
```

64-bit to double-precision variant

Applies when `sf == 1 && ftype == 01`.

```
SCVTF <Dd>, <Xn>, #<fbits>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
case ftype of
    when '00' fltsize = 32;
    when '01' fltsize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;

if sf == '0' && scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - UInt(scale);

rounding = FPRoundingMode(FPCR);
```

Assembler symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<fbits>	For the 32-bit to double-precision, 32-bit to half-precision and 32-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 32, encoded as 64 minus "scale". For the 64-bit to double-precision, 64-bit to half-precision and 64-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 64, encoded as 64 minus "scale".

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

intval = X[n];
fltval = FixedToFP(intval, fracbits, FALSE, FPCR, rounding);
V[d] = fltval;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.236 SCVTF (scalar, integer)

Signed integer Convert to Floating-point (scalar). This instruction converts the signed integer value in the general-purpose source register to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	16	15	14	13	12	11	10	9 5	4 0
sf	0	0	1	1	1	1	0	ftype	1	0	0	0	1	0	0	0	0	0	0	Rn	Rd
										rmode		opcode									

32-bit to half-precision variant

Applies when `sf == 0 && ftype == 11`.

```
SCVTF <Hd>, <Wn>
```

32-bit to single-precision variant

Applies when `sf == 0 && ftype == 00`.

```
SCVTF <Sd>, <Wn>
```

32-bit to double-precision variant

Applies when `sf == 0 && ftype == 01`.

```
SCVTF <Dd>, <Wn>
```

64-bit to half-precision variant

Applies when `sf == 1 && ftype == 11`.

```
SCVTF <Hd>, <Xn>
```

64-bit to single-precision variant

Applies when `sf == 1 && ftype == 00`.

```
SCVTF <Sd>, <Xn>
```

64-bit to double-precision variant

Applies when `sf == 1 && ftype == 01`.

```
SCVTF <Dd>, <Xn>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;

rounding = FPRoundingMode(FPCR);
```

Assembler symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

intval = X[n];
fltval = FixedToFP(intval, 0, FALSE, FPCR, rounding);
V[d] = fltval;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.237 SDOT (by element)

Dot Product signed arithmetic (vector, by element). This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_AA64ISAR0_EL1.DP indicates whether this instruction is supported.

ARMv8.2

31	30	29	28	27	26	25	24	23 22	21	20	19 16	15	14	13	12	11	10	9 5	4 0
0	Q	0	0	1	1	1	1	size	L	M	Rm	1	1	1	0	H	0	Rn	Rd
		U																	

Vector variant

```
SDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.4B[<index>]
```

Decode for this encoding

```
if !HaveDOTPExt() then UNDEFINED;
if size != '10' then UNDEFINED;
boolean signed = (U == '0');

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer index = UInt(H:L);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `2S` when `Q = 0`
- `4S` when `Q = 1`

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `8B` when `Q = 0`
- `16B` when `Q = 1`

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

`<index>` Is the element index, encoded in the "H:L" fields.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(128) operand2 = V[m];
bits(datasize) result = V[d];
for e = 0 to elements-1
    integer res = 0;
    integer element1, element2;
    for i = 0 to 3
        if signed then
            element1 = SInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4*index+i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4*index+i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.238 SDOT (vector)

Dot Product signed arithmetic (vector). This instruction performs the dot product of the four signed 8-bit elements in each 32-bit element of the first source register with the four signed 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_AA64ISAR0_EL1.DP indicates whether this instruction is supported.

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	size		0	Rm	1	0	0	1	0	1	Rn	Rd
		U																	

Vector variant

```
SDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !HaveDOTPExt() then UNDEFINED;
if size != '10' then UNDEFINED;
boolean signed = (U == '0');
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `2S` when `Q = 0`
- `4S` when `Q = 1`

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `8B` when `Q = 0`
- `16B` when `Q = 1`

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

result = V[d];
for e = 0 to elements-1
    integer res = 0;
    integer element1, element2;
    for i = 0 to 3
        if signed then
            element1 = SInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4*e+i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4*e+i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.239 SHA1C

SHA1 hash update (choose).

31 30 29 28	27 26 25 24	23 22	21	20 … 16	15	14 13 12	11 10	9 … 5	4 … 0
0 1 0 1	1 1 1 0	0 0	0	Rm	0	0 0 0	0 0	Rn	Rd

Advanced SIMD variant

```
SHA1C <Qd>, <Sn>, <Vm>.4S
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED;
```

Assembler symbols

`<Qd>`	Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
`<Sn>`	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
`<Vm>`	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) X = V[d];
bits(32) Y = V[n];    // Note: 32 not 128 bits wide
bits(128) W = V[m];
bits(32) t;

for e = 0 to 3
    t = SHAchoose(X<63:32>, X<95:64>, X<127:96>);
    Y = Y + ROL(X<31:0>, 5) + t + Elem[W, e, 32];
    X<63:32> = ROL(X<63:32>, 30);
    <Y, X> = ROL(Y:X, 32);
V[d] = X;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.240 SHA1H

SHA1 fixed rotate.

31 30 29 28	27 26 25 24	23 22	21 20 19 18 17	16 15 14 13 12	11 10	9 5	4 0
0 1 0 1	1 1 1 0	0 0	1 0 1 0 0	0 0 0 0 0	1 0	Rn	Rd

Advanced SIMD variant

```
SHA1H <Sd>, <Sn>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveSHA1Ext() then UNDEFINED;
```

Assembler symbols

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(32) operand = V[n];    // read element [0] only,  [1-3] zeroed
V[d] = ROL(operand, 30);
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.241 SHA1M

SHA1 hash update (majority).

31 30 29 28	27 26 25 24	23 22	21	20 … 16	15	14 13 12	11 10	9 … 5	4 … 0
0 1 0 1	1 1 1 0	0 0	0	Rm	0	0 1 0	0 0	Rn	Rd

Advanced SIMD variant

```
SHA1M <Qd>, <Sn>, <Vm>.4S
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED;
```

Assembler symbols

`<Qd>`	Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
`<Sn>`	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
`<Vm>`	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) X = V[d];
bits(32) Y = V[n];    // Note: 32 not 128 bits wide
bits(128) W = V[m];
bits(32) t;

for e = 0 to 3
    t = SHAmajority(X<63:32>, X<95:64>, X<127:96>);
    Y = Y + ROL(X<31:0>, 5) + t + Elem[W, e, 32];
    X<63:32> = ROL(X<63:32>, 30);
    <Y, X> = ROL(Y:X, 32);
V[d] = X;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.242 SHA1P

SHA1 hash update (parity).

31 30 29 28	27 26 25 24	23 22	21	20 … 16	15	14 13 12	11 10	9 … 5	4 … 0
0 1 0 1	1 1 1 0	0 0	0	Rm	0	0 0 1	0 0	Rn	Rd

Advanced SIMD variant

```
SHA1P <Qd>, <Sn>, <Vm>.4S
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED;
```

Assembler symbols

`<Qd>`	Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
`<Sn>`	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
`<Vm>`	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) X = V[d];
bits(32) Y = V[n];    // Note: 32 not 128 bits wide
bits(128) W = V[m];
bits(32) t;

for e = 0 to 3
    t = SHAparity(X<63:32>, X<95:64>, X<127:96>);
    Y = Y + ROL(X<31:0>, 5) + t + Elem[W, e, 32];
    X<63:32> = ROL(X<63:32>, 30);
    <Y, X> = ROL(Y:X, 32);
V[d] = X;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.243 SHA1SU0

SHA1 schedule update 0.

31 30 29 28	27 26 25 24	23 22	21	20 … 16	15	14 13 12	11 10	9 … 5	4 … 0
0 1 0 1	1 1 1 0	0 0	0	Rm	0	0 1 1	0 0	Rn	Rd

Advanced SIMD variant

```
SHA1SU0 <Vd>.4S, <Vn>.4S, <Vm>.4S
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED;
```

Assembler symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.

<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) operand3 = V[m];
bits(128) result;

result = operand2<63:0>:operand1<127:64>;
result = result EOR operand1 EOR operand3;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.244 SHA1SU1

SHA1 schedule update 1.

31 30 29 28	27 26 25 24	23 22	21 20 19 18 17	16 15 14 13 12	11 10	9 5	4 0
0 1 0 1	1 1 1 0	0 0	1 0 1 0 0	0 0 0 0 1	1 0	Rn	Rd

Advanced SIMD variant

```
SHA1SU1 <Vd>.4S, <Vn>.4S
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveSHA1Ext() then UNDEFINED;
```

Assembler symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.

<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) result;
bits(128) T = operand1 EOR LSR(operand2, 32);
result<31:0> = ROL(T<31:0>, 1);
result<63:32> = ROL(T<63:32>, 1);
result<95:64> = ROL(T<95:64>, 1);
result<127:96> = ROL(T<127:96>, 1) EOR ROL(T<31:0>, 2);
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.245 SHA256H2

SHA256 hash update (part 2).

31 30 29 28	27 26 25 24	23 22 21 20	16	15	14 13 12	11 10	9 5	4 0
0 1 0 1	1 1 1 0	0 0 0	Rm	0	1 0 1	0 0	Rn	Rd
					P			

Advanced SIMD variant

```
SHA256H2 <Qd>, <Qn>, <Vm>.4S
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA256Ext() then UNDEFINED;
```

Assembler symbols

`<Qd>` Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.

`<Qn>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) result;
result = SHA256hash(V[n], V[d], V[m], FALSE);
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.246 SHA256H

SHA256 hash update (part 1).

31 30 29 28	27 26 25 24	23 22 21 20	16	15	14 13 12	11 10	9 5	4 0
0 1 0 1	1 1 1 0	0 0 0	Rm	0	1 0 0	0 0	Rn	Rd
					P			

Advanced SIMD variant

```
SHA256H <Qd>, <Qn>, <Vm>.4S
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA256Ext() then UNDEFINED;
```

Assembler symbols

`<Qd>`	Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
`<Qn>`	Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
`<Vm>`	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) result;
result = SHA256hash(V[d], V[n], V[m], TRUE);
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.247 SHA256SU0

SHA256 schedule update 0.

31 30 29 28	27 26 25 24	23 22	21 20	19 18 17 16	15 14 13 12	11 10	9 … 5	4 … 0
0 1 0 1	1 1 1 0	0 0	1 0	1 0 0	0 0 0 1 0	1 0	Rn	Rd

Advanced SIMD variant

```
SHA256SU0 <Vd>.4S, <Vn>.4S
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveSHA256Ext() then UNDEFINED;
```

Assembler symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.

<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) result;
bits(128) T = operand2<31:0>:operand1<127:32>;
bits(32) elt;

for e = 0 to 3
    elt = Elem[T, e, 32];
    elt = ROR(elt, 7) EOR ROR(elt, 18) EOR LSR(elt, 3);
    Elem[result, e, 32] = elt + Elem[operand1, e, 32];
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.248 SHA256SU1

SHA256 schedule update 1.

31 30 29 28	27 26 25 24	23 22	21	20 … 16	15	14 13 12	11 10	9 … 5	4 … 0
0 1 0 1	1 1 1 0	0 0	0	Rm	0	1 1 0	0 0	Rn	Rd

Advanced SIMD variant

```
SHA256SU1 <Vd>.4S, <Vn>.4S, <Vm>.4S
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA256Ext() then UNDEFINED;
```

Assembler symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) operand3 = V[m];
bits(128) result;
bits(128) T0 = operand3<31:0>:operand2<127:32>;
bits(64) T1;
bits(32) elt;

T1 = operand3<127:64>;
for e = 0 to 1
    elt = Elem[T1, e, 32];
    elt = ROR(elt, 17) EOR ROR(elt, 19) EOR LSR(elt, 10);
    elt = elt + Elem[operand1, e, 32] + Elem[T0, e, 32];
    Elem[result, e, 32] = elt;

T1 = result<63:0>;
for e = 2 to 3
    elt = Elem[T1, e-2, 32];
    elt = ROR(elt, 17) EOR ROR(elt, 19) EOR LSR(elt, 10);
    elt = elt + Elem[operand1, e, 32] + Elem[T0, e, 32];
    Elem[result, e, 32] = elt;

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.249 SHA512H

SHA512 Hash update part 1 takes the values from the three 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the sigma1 and chi functions of two iterations of the SHA512 computation. It returns this value to the destination SIMD&FP register.

This instruction is implemented only when ARMv8.2-SHA is implemented.

ARMv8.2

31 30 29 28	27 26 25 24	23 22 21 20	16	15	14	13 12	11 10	9 5	4 0
1 1 0 0	1 1 1 0	0 1 1	Rm	1	0	0 0	0 0	Rn	Rd

Advanced SIMD variant

```
SHA512H <Qd>, <Qn>, <Vm>.2D
```

Decode for this encoding

```
if !HaveSHA512Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler symbols

<Qd>	Is the 128-bit name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Qn>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vtmp;
bits(64) MSigma1;
bits(64) tmp;
bits(128) X = V[n];
bits(128) Y = V[m];
bits(128) W = V[d];

MSigma1 = ROR(Y<127:64>, 14) EOR ROR(Y<127:64>, 18) EOR ROR(Y<127:64>, 41);
Vtmp<127:64> = (Y<127:64> AND X<63:0>) EOR (NOT(Y<127:64>) AND X<127:64>);
Vtmp<127:64> = (Vtmp<127:64> + MSigma1 + W<127:64>);
tmp = Vtmp<127:64> + Y<63:0>;
MSigma1 = ROR(tmp, 14) EOR ROR(tmp, 18) EOR ROR(tmp, 41);
Vtmp<63:0> = (tmp AND Y<127:64>) EOR (NOT(tmp) AND X<63:0>);
Vtmp<63:0> = (Vtmp<63:0> + MSigma1 + W<63:0>);
V[d] = Vtmp;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.250 SHA512H2

SHA512 Hash update part 2 takes the values from the three 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the sigma0 and majority functions of two iterations of the SHA512 computation. It returns this value to the destination SIMD&FP register.

This instruction is implemented only when ARMv8.2-SHA is implemented.

ARMv8.2

31 30 29 28	27 26 25 24	23 22 21 20	16	15	14	13 12	11 10	9 5	4 0
1 1 0 0	1 1 1 0	0 1 1	Rm	1	0	0 0	0 1	Rn	Rd

Advanced SIMD variant

```
SHA512H2 <Qd>, <Qn>, <Vm>.2D
```

Decode for this encoding

```
if !HaveSHA512Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler symbols

`<Qd>`	Is the 128-bit name of the SIMD&FP source and destination register, encoded in the "Rd" field.
`<Qn>`	Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
`<Vm>`	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vtmp;
bits(64) NSigma0;
bits(128) X = V[n];
bits(128) Y = V[m];
bits(128) W = V[d];

NSigma0 = ROR(Y<63:0>, 28) EOR ROR(Y<63:0>, 34) EOR ROR(Y<63:0>, 39);
Vtmp<127:64> = (X<63:0> AND Y<127:64>) EOR (X<63:0> AND Y<63:0>) EOR (Y<127:64> AND Y<63:0>);
Vtmp<127:64> = (Vtmp<127:64> + NSigma0 + W<127:64>);
NSigma0 = ROR(Vtmp<127:64>, 28) EOR ROR(Vtmp<127:64>, 34) EOR ROR(Vtmp<127:64>, 39);
Vtmp<63:0> = (Vtmp<127:64> AND Y<63:0>) EOR (Vtmp<127:64> AND Y<127:64>) EOR (Y<127:64> AND Y<63:0>);
Vtmp<63:0> = (Vtmp<63:0> + NSigma0 + W<63:0>);

V[d] = Vtmp;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.251 SHA512SU0

SHA512 Schedule Update 0 takes the values from the two 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the gamma0 functions of two iterations of the SHA512 schedule update that are performed after the first 16 iterations within a block. It returns this value to the destination SIMD&FP register.

This instruction is implemented only when ARMv8.2-SHA is implemented.

ARMv8.2

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10	9 5	4 0
1 1 0 0	1 1 1 0	1 1 0 0	0 0 0 0	0 1 0 0	0 0	Rn	Rd

Advanced SIMD variant

```
SHA512SU0 <Vd>.2D, <Vn>.2D
```

Decode for this encoding

```
if !HaveSHA512Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
```

Assembler symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.

<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(64) sig0;
bits(128) Vtmp;
bits(128) X = V[n];
bits(128) W = V[d];
sig0 = ROR(W<127:64>, 1) EOR ROR(W<127:64>, 8) EOR ('0000000':W<127:71>);
Vtmp<63:0> = W<63:0> + sig0;
sig0 = ROR(X<63:0>, 1) EOR ROR(X<63:0>, 8) EOR ('0000000':X<63:7>);
Vtmp<127:64> = W<127:64> + sig0;
V[d] = Vtmp;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.252 SHA512SU1

SHA512 Schedule Update 1 takes the values from the three source SIMD&FP registers and produces a 128-bit output value that combines the gamma1 functions of two iterations of the SHA512 schedule update that are performed after the first 16 iterations within a block. It returns this value to the destination SIMD&FP register.

This instruction is implemented only when ARMv8.2-SHA is implemented.

ARMv8.2

31 30 29 28	27 26 25 24	23 22 21 20	16	15	14	13 12	11 10	9 5	4 0
1 1 0 0	1 1 1 0	0 1 1	Rm	1	0	0 0	1 0	Rn	Rd

Advanced SIMD variant

```
SHA512SU1 <Vd>.2D, <Vn>.2D, <Vm>.2D
```

Decode for this encoding

```
if !HaveSHA512Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler symbols

`<Vd>`	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
`<Vn>`	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
`<Vm>`	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(64) sig1;
bits(128) Vtmp;
bits(128) X = V[n];
bits(128) Y = V[m];
bits(128) W = V[d];

sig1 = ROR(X<127:64>, 19) EOR ROR(X<127:64>, 61) EOR ('000000':X<127:70>);
Vtmp<127:64> = W<127:64> + sig1 + Y<127:64>;
sig1 = ROR(X<63:0>, 19) EOR ROR(X<63:0>, 61) EOR ('000000':X<63:6>);
Vtmp<63:0> = W<63:0> + sig1 + Y<63:0>;
V[d] = Vtmp;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.253 SHADD

Signed Halving Add. This instruction adds corresponding signed integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see SRHADD.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 16	15	14	13	12	11	10	9 5	4 0
0	Q	0	0	1	1	1	0	size	1	Rm	0	0	0	0	0	1	Rn	Rd
		U																

Three registers of the same type variant

```
SHADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B`	when `size = 00`, `Q = 0`
`16B`	when `size = 00`, `Q = 1`
`4H`	when `size = 01`, `Q = 0`
`8H`	when `size = 01`, `Q = 1`
`2S`	when `size = 10`, `Q = 0`
`4S`	when `size = 10`, `Q = 1`

The encoding `size = 11`, `Q = x` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer sum;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = element1 + element2;
    Elem[result, e, esize] = sum<esize:1>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.254 SHL

Shift Left (immediate). This instruction reads each value from a vector, left shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31 30 29 28	27 26 25 24	23 22	19 18 16	15 14 13 12	11	10 9 5 4	0
0 1 0	1 1 1 1 1 0	!=0000	immb	0 1 0 1 0	1	Rn	Rd
		immh					

Scalar variant

```
SHL <V><d>, <V><n>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;
```

Vector

31 30 29 28	27 26 25 24	23 22	19 18 16	15 14 13 12	11	10 9 5 4	0
0 Q 0	0 1 1 1 1 0	!=0000	immb	0 1 0 1 0	1	Rn	Rd
		immh					

Vector variant

```
SHL <Vd>.<T>, <Vn>.<T>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;
```

Assembler symbols

<V> Is a width specifier, encoded in the "immh" field. It can have the following values:

D when immh = 1xxx

The encoding immh = 0xxx is reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

`8B` when `immh = 0001`, `Q = 0`

`16B` when `immh = 0001`, `Q = 1`

`4H` when `immh = 001x`, `Q = 0`

`8H` when `immh = 001x`, `Q = 1`

`2S` when `immh = 01xx`, `Q = 0`

`4S` when `immh = 01xx`, `Q = 1`

`2D` when `immh = 1xxx`, `Q = 1`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`, `Q = x`.

The encoding `immh = 1xxx`, `Q = 0` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<shift>` For the scalar variant: is the left shift amount, in the range 0 to 63, encoded in the "immh:immb" field. It can have the following values:

`(UInt(immh:immb)-64)` when `immh = 1xxx`

The encoding `immh = 0xxx` is reserved.

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in the "immh:immb" field. It can have the following values:

`(UInt(immh:immb)-8)` when `immh = 0001`

`(UInt(immh:immb)-16)` when `immh = 001x`

`(UInt(immh:immb)-32)` when `immh = 01xx`

`(UInt(immh:immb)-64)` when `immh = 1xxx`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

for e = 0 to elements-1
    Elem[result, e, esize] = LSL(Elem[operand, e, esize], shift);

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.255 SHLL, SHLL2

Shift Left Long (by element size). This instruction reads each vector element in the lower or upper half of the source SIMD&FP register, left shifts each result by the element size, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The `SHLL` instruction extracts vector elements from the lower half of the source register, while the `SHLL2` instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	17	16	15	14	13	12	11	10	9 5	4 0
0	Q	1	0	1	1	1	0	size	1	0	0	0	0	1	0	0	1	1	1	0	Rn	Rd

Vector variant

```
SHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = esize;
boolean unsigned = FALSE;    // Or TRUE without change of functionality
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- `[absent]` when `Q = 0`
- `[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- `8H` when `size = 00`
- `4S` when `size = 01`
- `2D` when `size = 10`

The encoding `size = 11` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `8B` when `size = 00`, `Q = 0`
- `16B` when `size = 00`, `Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<shift> Is the left shift amount, which must be equal to the source element width in bits, encoded in the "size" field. It can have the following values:

8 when size = 00

16 when size = 01

32 when size = 10

The encoding size = 11 is reserved.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part];
bits(2*datasize) result;
integer element;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], unsigned) << shift;
    Elem[result, e, 2*esize] = element<2*esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.256 SHRN, SHRN2

Shift Right Narrow (immediate). This instruction reads each unsigned integer value from the source SIMD&FP register, right shifts each result by an immediate value, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. The results are truncated. For rounded results, see RSHRN, RSHRN2.

The `RSHRN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `RSHRN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	19	18	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	1	0	!=0000	immb			1	0	0	0	0	1	Rn		Rd	
									immh								op					

Vector variant

```
SHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Tb>` Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

`8B` when `immh = 0001`, `Q = 0`

`16B` when `immh = 0001`, `Q = 1`

`4H` when `immh = 001x`, `Q = 0`

`8H` when `immh = 001x`, `Q = 1`

`2S` when `immh = 01xx`, `Q = 0`

`4S` when `immh = 01xx`, `Q = 1`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`, `Q = x`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The encoding `immh = 1xxx`, `Q = x` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<Ta>` Is an arrangement specifier, encoded in the "immh" field. It can have the following values:

`8H` when `immh = 0001`

`4S` when `immh = 001x`

`2D` when `immh = 01xx`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`.

The encoding `immh = 1xxx` is reserved.

`<shift>` Is the right shift amount, in the range 1 to the destination element width in bits, encoded in the "immh:immb" field. It can have the following values:

`(16-UInt(immh:immb))` when `immh = 0001`

`(32-UInt(immh:immb))` when `immh = 001x`

`(64-UInt(immh:immb))` when `immh = 01xx`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`.

The encoding `immh = 1xxx` is reserved.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

for e = 0 to elements-1
    element = (UInt(Elem[operand, e, 2*esize]) + round_const) >> shift;
    Elem[result, e, esize] = element<esize-1:0>;

Vpart[d, part] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.257 SHSUB

Signed Halving Subtract. This instruction subtracts the elements in the vector in the second source SIMD&FP register from the corresponding elements in the vector in the first source SIMD&FP register, shifts each result right one bit, places each result into elements of a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	size	1	Rm	0	0	1	0	0	1	Rn	Rd
		U																

Three registers of the same type variant

```
SHSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B`	when `size = 00`, `Q = 0`
`16B`	when `size = 00`, `Q = 1`
`4H`	when `size = 01`, `Q = 0`
`8H`	when `size = 01`, `Q = 1`
`2S`	when `size = 10`, `Q = 0`
`4S`	when `size = 10`, `Q = 1`

The encoding `size = 11`, `Q = x` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer diff;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    element2 = Int(Elem[operand2, e, esize], unsigned);
    diff = element1 - element2;
    Elem[result, e, esize] = diff<esize:1>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

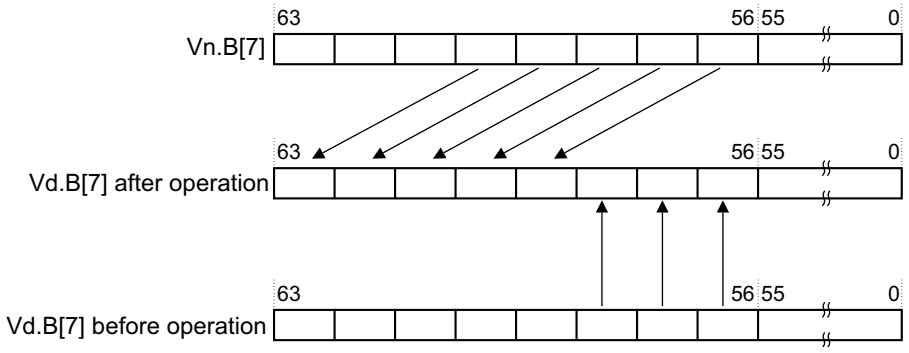
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.258 SLI

Shift Left and Insert (immediate). This instruction reads each vector element in the source SIMD&FP register, left shifts each vector element by an immediate value, and inserts the result into the corresponding vector element in the destination SIMD&FP register such that the new zero bits created by the shift are not inserted but retain their existing value. Bits shifted out of the left of each vector element in the source register are lost.

The following figure shows the operation of shift left by 3 for an 8-bit vector element.



Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23	22	19	18	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	1	0	!=0000	immb			0	1	0	1	0	1	Rn		Rd	
									immh													

Scalar variant

```
SLI <V><d>, <V><n>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;
```

Vector

31	30	29	28	27	26	25	24	23	22	19	18	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	1	0	!=0000	immb			0	1	0	1	0	1	Rn		Rd	
									immh													

Vector variant

```
SLI <Vd>.<T>, <Vn>.<T>, #<shift>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;
```

Assembler symbols

`<V>` Is a width specifier, encoded in the "immh" field. It can have the following values:

`D` when `immh = 1xxx`

The encoding `immh = 0xxx` is reserved.

`<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

`8B` when `immh = 0001`, `Q = 0`

`16B` when `immh = 0001`, `Q = 1`

`4H` when `immh = 001x`, `Q = 0`

`8H` when `immh = 001x`, `Q = 1`

`2S` when `immh = 01xx`, `Q = 0`

`4S` when `immh = 01xx`, `Q = 1`

`2D` when `immh = 1xxx`, `Q = 1`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`, `Q = x`.

The encoding `immh = 1xxx`, `Q = 0` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<shift>` For the scalar variant: is the left shift amount, in the range 0 to 63, encoded in the "immh:immb" field. It can have the following values:

`(UInt(immh:immb)-64)` when `immh = 1xxx`

The encoding `immh = 0xxx` is reserved.

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in the "immh:immb" field. It can have the following values:

`(UInt(immh:immb)-8)` when `immh = 0001`

`(UInt(immh:immb)-16)` when `immh = 001x`

`(UInt(immh:immb)-32)` when `immh = 01xx`

`(UInt(immh:immb)-64)` when `immh = 1xxx`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2 = V[d];
bits(datasize) result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(esize) mask = LSL(Ones(esize), shift);
bits(esize) shifted;

for e = 0 to elements-1
    shifted = LSL(Elem[operand, e, esize], shift);
    Elem[result, e, esize] = (Elem[operand2, e, esize] AND NOT(mask)) OR shifted;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.259 SM3PARTW1

SM3PARTW1 takes three 128-bit vectors from the three source SIMD&FP registers and returns a 128-bit result in the destination SIMD&FP register. The result is obtained by a three-way exclusive OR of the elements within the input vectors with some fixed rotations, see the Operation pseudocode for more information.

This instruction is implemented only when ARMv8.2-SM is implemented.

ARMv8.2

31 30 29 28	27 26 25 24	23 22 21 20	16	15	14	13 12	11 10	9 5	4 0
1 1 0 0	1 1 1 0	0 1 1	Rm	1	1	0 0	0 0	Rn	Rd

Advanced SIMD variant

```
SM3PARTW1 <Vd>.4S, <Vn>.4S, <Vm>.4S
```

Decode for this encoding

```
if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(128) result;

result<95:0> = (Vd EOR Vn)<95:0> EOR (ROL(Vm<127:96>, 15):ROL(Vm<95:64>, 15):ROL(Vm<63:32>, 15));

for i = 0 to 3
    if i == 3 then
        result<127:96> = (Vd EOR Vn)<127:96> EOR (ROL(result<31:0>, 15));
    result<(32*i)+31:(32*i)> = result<(32*i)+31:(32*i)> EOR ROL(result<(32*i)+31:(32*i)>, 15) EOR
ROL(result<(32*i)+31:(32*i)>, 23);
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.260 SM3PARTW2

SM3PARTW2 takes three 128-bit vectors from three source SIMD&FP registers and returns a 128-bit result in the destination SIMD&FP register. The result is obtained by a three-way exclusive OR of the elements within the input vectors with some fixed rotations, see the Operation pseudocode for more information.

This instruction is implemented only when ARMv8.2-SM is implemented.

ARMv8.2

31 30 29 28	27 26 25 24	23 22 21 20	16	15	14	13 12	11 10	9 5	4 0
1 1 0 0	1 1 1 0	0 1 1	Rm	1	1	0 0	0 1	Rn	Rd

Advanced SIMD variant

```
SM3PARTW2 <Vd>.4S, <Vn>.4S, <Vm>.4S
```

Decode for this encoding

```
if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler symbols

`<Vd>`	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
`<Vn>`	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
`<Vm>`	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(128) result;
bits(128) tmp;
bits(32) tmp2;
tmp<127:0> = Vn EOR (ROL(Vm<127:96>, 7):ROL(Vm<95:64>, 7):ROL(Vm<63:32>, 7):ROL(Vm<31:0>, 7));
result<127:0> = Vd<127:0> EOR tmp<127:0>;
tmp2 = ROL(tmp<31:0>, 15);
tmp2 = tmp2 EOR ROL(tmp2, 15) EOR ROL(tmp2, 23);
result<127:96> = result<127:96> EOR tmp2;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.261 SM3SS1

SM3SS1 rotates the top 32 bits of the 128-bit vector in the first source SIMD&FP register by 12, and adds that 32-bit value to the two other 32-bit values held in the top 32 bits of each of the 128-bit vectors in the second and third source SIMD&FP registers, rotating this result left by 7 and writing the final result into the top 32 bits of the vector in the destination SIMD&FP register, with the bottom 96 bits of the vector being written to 0.

This instruction is implemented only when ARMv8.2-SM is implemented.

ARMv8.2

31 30 29 28	27 26 25 24	23 22 21 20	16	15 14	10 9	5 4	0
1 1 0 0	1 1 1 0 0	1 0	Rm	0	Ra	Rn	Rd

Advanced SIMD variant

```
SM3SS1 <Vd>.4S, <Vn>.4S, <Vm>.4S, <Va>.4S
```

Decode for this encoding

```
if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer a = UInt(Ra);
```

Assembler symbols

<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn>	Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
<Va>	Is the name of the third SIMD&FP source register, encoded in the "Ra" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(128) Va = V[a];
Vd<127:96> = ROL((ROL(Vn<127:96>, 12) + Vm<127:96> + Va<127:96>), 7);
Vd<95:0> = Zeros();
V[d] = Vd;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.262 SM3TT1A

SM3TT1A takes three 128-bit vectors from three source SIMD&FP registers and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a three-way exclusive OR of the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the three-way exclusive OR.
- The result of the exclusive OR of the top 32-bit element of the second source vector, Vn, with a rotation left by 12 of the top 32-bit element of the first source vector.
- A 32-bit element indexed out of the third source vector, Vm.

The result of this addition is returned as the top element of the result. The other elements of the result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 9.

This instruction is implemented only when ARMv8.2-SM is implemented.

ARMv8.2

31 30 29 28	27 26 25 24	23 22 21 20	16	15 14	13 12	11 10	9 5	4 0
1 1 0 0	1 1 1 0	0 1 0	Rm	1 0	imm2	0 0	Rn	Rd

Advanced SIMD variant

```
SM3TT1A <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]
```

Decode for this encoding

```
if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer i = UInt(imm2);
```

Assembler symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.
<imm2>	Is a 32-bit element indexed out of <Vm>, encoded in "imm2".

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(32) WjPrime;
bits(128) result;
bits(32) TT1;
bits(32) SS2;

WjPrime = Elem[Vm, i, 32];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
SS2 = Vn<127:96> EOR ROL(Vd<127:96>, 12);
TT1 = Vd<63:32> EOR (Vd<127:96> EOR Vd<95:64>);
TT1 = (TT1+Vd<31:0>+SS2+WjPrime)<31:0>;
result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>, 9);
result<95:64> = Vd<127:96>;
result<127:96> = TT1;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.263 SM3TT1B

SM3TT1B takes three 128-bit vectors from three source SIMD&FP registers and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a 32-bit majority function between the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the 32-bit majority function.
- The result of the exclusive OR of the top 32-bit element of the second source vector, Vn, with a rotation left by 12 of the top 32-bit element of the first source vector.
- A 32-bit element indexed out of the third source vector, Vm.

The result of this addition is returned as the top element of the result. The other elements of the result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 9.

This instruction is implemented only when ARMv8.2-SM is implemented.

ARMv8.2

31 30 29 28	27 26 25 24	23 22 21 20	16	15 14	13 12	11 10	9 5	4 0
1 1 0 0	1 1 1 0	0 1 0	Rm	1 0	imm2	0 1	Rn	Rd

Advanced SIMD variant

```
SM3TT1B <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]
```

Decode for this encoding

```
if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer i = UInt(imm2);
```

Assembler symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.
<imm2>	Is a 32-bit element indexed out of <Vm>, encoded in "imm2".

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(32) WjPrime;
bits(128) result;
bits(32) TT1;
bits(32) SS2;

WjPrime = Elem[Vm, i, 32];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
SS2 = Vn<127:96> EOR ROL(Vd<127:96>, 12);
TT1 = (Vd<127:96> AND Vd<63:32>) OR (Vd<127:96> AND Vd<95:64>) OR (Vd<63:32> AND Vd<95:64>);
TT1 = (TT1+Vd<31:0>+SS2+WjPrime)<31:0>;
result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>, 9);
result<95:64> = Vd<127:96>;
result<127:96> = TT1;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.264 SM3TT2A

SM3TT2A takes three 128-bit vectors from three source SIMD&FP register and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a three-way exclusive OR of the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the three-way exclusive OR.
- The 32-bit element held in the top 32 bits of the second source vector, Vn.
- A 32-bit element indexed out of the third source vector, Vm.

A three-way exclusive OR is performed of the result of this addition, the result of the addition rotated left by 9, and the result of the addition rotated left by 17. The result of this exclusive OR is returned as the top element of the returned result. The other elements of this result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 19.

This instruction is implemented only when ARMv8.2-SM is implemented.

ARMv8.2

31 30 29 28	27 26 25 24	23 22 21 20	16	15 14	13 12	11 10	9 5	4 0
1 1 0 0	1 1 1 0	0 1 0	Rm	1 0	imm2	1 0	Rn	Rd

Advanced SIMD variant

```
SM3TT2A <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]
```

Decode for this encoding

```
if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer i = UInt(imm2);
```

Assembler symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.
<imm2>	Is a 32-bit element indexed out of <Vm>, encoded in "imm2".

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(32) Wj;
bits(128) result;
bits(32) TT2;

Wj = Elem[Vm, i, 32];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
TT2 = Vd<63:32> EOR (Vd<127:96> EOR Vd<95:64>);
TT2 = (TT2+Vd<31:0>+Vn<127:96>+Wj)<31:0>;

result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>, 19);
result<95:64> = Vd<127:96>;
result<127:96> = TT2 EOR ROL(TT2, 9) EOR ROL(TT2, 17);
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.265 SM3TT2B

SM3TT2B takes three 128-bit vectors from three source SIMD&FP registers, and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a 32-bit majority function between the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the 32-bit majority function.
- The 32-bit element held in the top 32 bits of the second source vector, Vn.
- A 32-bit element indexed out of the third source vector, Vm.

A three-way exclusive OR is performed of the result of this addition, the result of the addition rotated left by 9, and the result of the addition rotated left by 17. The result of this exclusive OR is returned as the top element of the returned result. The other elements of this result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 19.

This instruction is implemented only when ARMv8.2-SM is implemented.

ARMv8.2

31 30 29 28	27 26 25 24	23 22 21 20	16	15 14	13 12	11 10	9 5	4 0
1 1 0 0	1 1 1 0	0 1 0	Rm	1 0	imm2	1 1	Rn	Rd

Advanced SIMD variant

```
SM3TT2B <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]
```

Decode for this encoding

```
if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer i = UInt(imm2);
```

Assembler symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.
<imm2>	Is a 32-bit element indexed out of <Vm>, encoded in "imm2".

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(32) Wj;
bits(128) result;
bits(32) TT2;

Wj = Elem[Vm, i, 32];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
TT2 = (Vd<127:96> AND Vd<95:64>) OR (NOT(Vd<127:96>) AND Vd<63:32>);
TT2 = (TT2+Vd<31:0>+Vn<127:96>+Wj)<31:0>;

result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>, 19);
result<95:64> = Vd<127:96>;
result<127:96> = TT2 EOR ROL(TT2, 9) EOR ROL(TT2, 17);
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.266 SM4E

SM4 Encode takes input data as a 128-bit vector from the first source SIMD&FP register, and four iterations of the round key held as the elements of the 128-bit vector in the second source SIMD&FP register. It encrypts the data by four rounds, in accordance with the SM4 standard, returning the 128-bit result to the destination SIMD&FP register.

This instruction is implemented only when ARMv8.2-SM is implemented.

ARMv8.2

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10	9 5	4 0
1 1 0 0	1 1 1 0	1 1 0 0	0 0 0 0	1 0 0 0	0 1	Rn	Rd

Advanced SIMD variant

```
SM4E <Vd>.4S, <Vn>.4S
```

Decode for this encoding

```
if !HaveSM4Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
```

Assembler symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.

<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vn = V[n];
bits(32) intval;
bits(8) sboxout;
bits(128) roundresult;
bits(32) roundkey;

roundresult = V[d];
for index = 0 to 3
    roundkey = Elem[Vn, index, 32];

    intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR roundkey;

    for i = 0 to 3
        Elem[intval, i, 8] = Sbox(Elem[intval, i, 8]);

    intval = intval EOR ROL(intval, 2) EOR ROL(intval, 10) EOR ROL(intval, 18) EOR ROL(intval, 24);
    intval = intval EOR roundresult<31:0>;

    roundresult<31:0> = roundresult<63:32>;
    roundresult<63:32> = roundresult<95:64>;
    roundresult<95:64> = roundresult<127:96>;
    roundresult<127:96> = intval;
V[d] = roundresult;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.267 SM4EKEY

SM4 Key takes an input as a 128-bit vector from the first source SIMD&FP register and a 128-bit constant from the second SIMD&FP register. It derives four iterations of the output key, in accordance with the SM4 standard, returning the 128-bit result to the destination SIMD&FP register.

This instruction is implemented only when ARMv8.2-SM is implemented.

ARMv8.2

31 30 29 28	27 26 25 24	23 22 21 20	16	15	14	13 12	11 10	9 5	4 0
1 1 0 0	1 1 1 0	0 1 1	Rm	1	1	0 0	1 0	Rn	Rd

Advanced SIMD variant

```
SM4EKEY <Vd>.4S, <Vn>.4S, <Vm>.4S
```

Decode for this encoding

```
if !HaveSM4Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler symbols

<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn>	Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(32) intval;
bits(8) sboxout;
bits(128) result;
bits(32) const;
bits(128) roundresult;

roundresult = V[n];
for index = 0 to 3
    const = Elem[Vm, index, 32];

    intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR const;

    for i = 0 to 3
        Elem[intval, i, 8] = Sbox(Elem[intval, i, 8]);

    intval = intval EOR ROL(intval, 13) EOR ROL(intval, 23);
    intval = intval EOR roundresult<31:0>;

    roundresult<31:0> = roundresult<63:32>;
    roundresult<63:32> = roundresult<95:64>;
    roundresult<95:64> = roundresult<127:96>;
    roundresult<127:96> = intval;
V[d] = roundresult;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.268 SMAX

Signed Maximum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the larger of each pair of signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	size	1	Rm	0	1	1	0	0	1	Rn	Rd
		U													o1			

Three registers of the same type variant

```
SMAX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B	when size = 00, Q = 0
16B	when size = 00, Q = 1
4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.269 SMAXP

Signed Maximum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	size	1	Rm	1	0	1	0	0	1	Rn	Rd
		U													o1			

Three registers of the same type variant

```
SMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B`	when `size = 00`, `Q = 0`
`16B`	when `size = 00`, `Q = 1`
`4H`	when `size = 01`, `Q = 0`
`8H`	when `size = 01`, `Q = 1`
`2S`	when `size = 10`, `Q = 0`
`4S`	when `size = 10`, `Q = 1`

The encoding `size = 11`, `Q = x` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[concat, 2*e, esize], unsigned);
    element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.270 SMAXV

Signed Maximum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are signed integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	size	1	1	0	0	0	0	1	0	1	0	1	0	Rn	Rd
		U												op								

Advanced SIMD variant

```
SMAXV <V><d>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean min = (op == '1');
```

Assembler symbols

<V> Is the destination width specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10

The encoding size = 11 is reserved.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 4S when size = 10, Q = 1

The following encodings are reserved:

- size = 10, Q = 0.
- size = 11, Q = x.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer maxmin;
integer element;

maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);

V[d] = maxmin<esize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.271 SMIN

Signed Minimum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the smaller of each of the two signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	size	1	Rm	0	1	1	0	1	1	Rn	Rd
		U													o1			

Three registers of the same type variant

```
SMIN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B	when size = 00, Q = 0
16B	when size = 00, Q = 1
4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.272 SMINP

Signed Minimum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the smallest of each pair of signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	size		1	Rm		1	0	1	0	1	1	Rn		Rd	
		U															o1					

Three registers of the same type variant

```
SMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B`	when `size = 00`, `Q = 0`
`16B`	when `size = 00`, `Q = 1`
`4H`	when `size = 01`, `Q = 0`
`8H`	when `size = 01`, `Q = 1`
`2S`	when `size = 10`, `Q = 0`
`4S`	when `size = 10`, `Q = 1`

The encoding `size = 11`, `Q = x` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[concat, 2*e, esize], unsigned);
    element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.273 SMINV

Signed Minimum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are signed integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	size	1	1	0	0	0	1	1	0	1	0	1	0	Rn	Rd
		U												op								

Advanced SIMD variant

```
SMINV <V><d>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean min = (op == '1');
```

Assembler symbols

<V> Is the destination width specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10

The encoding size = 11 is reserved.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 4S when size = 10, Q = 1

The following encodings are reserved:

- size = 10, Q = 0.
- size = 11, Q = x.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer maxmin;
integer element;

maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);

V[d] = maxmin<esize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.274 SMLAL, SMLAL2 (by element)

Signed Multiply-Add Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element in the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are signed integer values.

The `SMLAL` instruction extracts vector elements from the lower half of the first source register, while the `SMLAL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19 16	15	14	13	12	11	10	9 5	4 0
0	Q	0	0	1	1	1	1	size	L	M	Rm	0	0	1	0	H	0	Rn	Rd
		U											o2						

Vector variant

```
SMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

`4S` when `size = 01`

`2D` when `size = 10`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`4H` when `size = 01`, `Q = 0`

`8H` when `size = 01`, `Q = 1`

`2S` when `size = 10`, `Q = 0`

`4S` when `size = 10`, `Q = 1`

The following encodings are reserved:

- `size = 00`, `Q = x`.
- `size = 11`, `Q = x`.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:

`0:Rm` when `size = 01`

`M:Rm` when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

Restricted to V0-V15 when element size <Ts> is H.

`<Ts>` Is an element size specifier, encoded in the "size" field. It can have the following values:

`H` when `size = 01`

`S` when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

`<index>` Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

`H:L:M` when `size = 01`

`H:L` when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.275 SMLAL, SMLAL2 (vector)

Signed Multiply-Add Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `SMLAL` instruction extracts each source vector from the lower half of each source register, while the `SMLAL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	size	1	Rm	1	0	0	0	0	0	Rn	Rd
		U											o1					

Three registers, not all the same type variant

```
SMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- `[absent]` when `Q = 0`
- `[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- `8H` when `size = 00`
- `4S` when `size = 01`
- `2D` when `size = 10`

The encoding `size = 11` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `8B` when `size = 00`, `Q = 0`
- `16B` when `size = 00`, `Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.276 SMLSL, SMLSL2 (by element)

Signed Multiply-Subtract Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `SMLSL` instruction extracts vector elements from the lower half of the first source register, while the `SMLSL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19 16	15	14	13	12	11	10	9 5	4 0
0	Q	0	0	1	1	1	1	size	L	M	Rm	0	1	1	0	H	0	Rn	Rd
		U											o2						

Vector variant

```
SMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

`4S` when `size = 01`

`2D` when `size = 10`

The following encodings are reserved:

- `size = 00`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- size = 11.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:

0:Rm when size = 01

M:Rm when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<index> Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

H:L:M when size = 01

H:L when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.277 SMLSL, SMLSL2 (vector)

Signed Multiply-Subtract Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `SMLSL` instruction extracts each source vector from the lower half of each source register, while the `SMLSL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	size	1	Rm	1	0	1	0	0	0	Rn	Rd
		U											o1					

Three registers, not all the same type variant

```
SMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- `[absent]` when `Q = 0`
- `[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- `8H` when `size = 00`
- `4S` when `size = 01`
- `2D` when `size = 10`

The encoding `size = 11` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `8B` when `size = 00`, `Q = 0`
- `16B` when `size = 00`, `Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

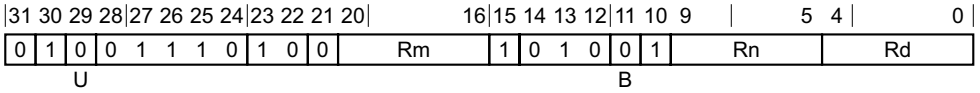
Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.278 SMMLA (vector)

Signed 8-bit integer matrix multiply-accumulate. This instruction multiplies the 2x8 matrix of signed 8-bit integer values in the first source vector by the 8x2 matrix of signed 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2 to Armv8.5, this is an OPTIONAL instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. ID_AA64ISAR1_EL1.I8MM indicates whether this instruction is supported.

ARMv8.6



Vector variant

```
SMMLA <Vd>.4S, <Vn>.16B, <Vm>.16B
```

Decode for this encoding

```
if !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
```

Assembler symbols

<Vd>	Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(128) operand1 = V[n];
bits(128) operand2 = V[m];
bits(128) addend = V[d];

V[d] = MatMulAdd(addend, operand1, operand2, FALSE, FALSE);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.279 SMOV

Signed Move vector element to general-purpose register. This instruction reads the signed integer from the source SIMD&FP register, sign-extends it to form a 32-bit or 64-bit value, and writes the result to destination general-purpose register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	0	0	imm5		0	0	1	0	1	1	Rn		Rd	

32-bit variant

Applies when `Q == 0`.

```
SMOV <Wd>, <Vn>.<Ts>[<index>]
```

64-reg,SMOV-64-reg variant

Applies when `Q == 1`.

```
SMOV <Xd>, <Vn>.<Ts>[<index>]
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size;
case Q:imm5 of
    when 'xxxxx1' size = 0;    // SMOV [WX]d, Vn.B
    when 'xxxx10' size = 1;    // SMOV [WX]d, Vn.H
    when '1xx100' size = 2;    // SMOV Xd, Vn.S
    otherwise UNDEFINED;

integer idxdsize = if imm5<4> == '1' then 128 else 64;
integer index = UInt(imm5<4:size+1>);
integer esize = 8 << size;
integer datasize = if Q == '1' then 64 else 32;
```

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<Ts>` For the 32-bit variant: is an element size specifier, encoded in the "imm5" field. It can have the following values:

- `B` when `imm5 = xxxx1`
- `H` when `imm5 = xxx10`

The encoding `imm5 = xxx00` is reserved.

For the 64-reg,SMOV-64-reg variant: is an element size specifier, encoded in the "imm5" field. It can have the following values:

- `B` when `imm5 = xxxx1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H when imm5 = xxx10

S when imm5 = xx100

The encoding imm5 = xx000 is reserved.

<index> For the 32-bit variant: is the element index encoded in the "imm5" field. It can have the following values:

imm5<4:1> when imm5 = xxxx1

imm5<4:2> when imm5 = xxx10

The encoding imm5 = xxx00 is reserved.

For the 64-reg,SMOV-64-reg variant: is the element index encoded in the "imm5" field. It can have the following values:

imm5<4:1> when imm5 = xxxx1

imm5<4:2> when imm5 = xxx10

imm5<4:3> when imm5 = xx100

The encoding imm5 = xx000 is reserved.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(idxdsize) operand = V[n];

X[d] = SignExtend(Elem[operand, index, esize], datasize);
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.280 SMULL, SMULL2 (by element)

Signed Multiply Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `SMULL` instruction extracts vector elements from the lower half of the first source register, while the `SMULL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19 16	15	14	13	12	11	10	9 5	4 0
0	Q	0	0	1	1	1	1	size	L	M	Rm	1	0	1	0	H	0	Rn	Rd
		U																	

Vector variant

```
SMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

`4S` when `size = 01`

`2D` when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:

0:Rm when size = 01

M:Rm when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<index> Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

H:L:M when size = 01

H:L when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = product;

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.281 SMULL, SMULL2 (vector)

Signed Multiply Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

The destination vector elements are twice as long as the elements that are multiplied.

The `SMULL` instruction extracts each source vector from the lower half of each source register, while the `SMULL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	size	1	Rm	1	1	0	0	0	0	Rn	Rd
		U																

Three registers, not all the same type variant

```
SMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- `[absent]` when `Q = 0`
- `[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- `8H` when `size = 00`
- `4S` when `size = 01`
- `2D` when `size = 10`

The encoding `size = 11` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `8B` when `size = 00`, `Q = 0`
- `16B` when `size = 00`, `Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    Elem[result, e, 2*esize] = (element1*element2)<2*esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.282 SQABS

Signed saturating Absolute value. This instruction reads each vector element from the source SIMD&FP register, puts the absolute value of the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31 30	29	28 27 26 25 24	23 22	21 20 19 18 17 16	15 14 13 12	11 10	9 5	4 0
0 1	0	1 1 1 1 0	size	1 0 0 0 0	0 0 1 1 1	1 0	Rn	Rd
	U							

Scalar variant

```
SQABS <V><d>, <V><n>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean neg = (U == '1');
```

Vector

31 30	29	28 27 26 25 24	23 22	21 20 19 18 17 16	15 14 13 12	11 10	9 5	4 0
0 Q	0	0 1 1 1 0	size	1 0 0 0 0	0 0 1 1 1	1 0	Rn	Rd
	U							

Vector variant

```
SQABS <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D when size = 11

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);
    (Elem[result, e, esize], sat) = SignedSatQ(element, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.283 SQADD

Signed saturating Add. This instruction adds the values of corresponding elements of the two source SIMD&FP registers, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	0	1	1	1	1	0	size	1	Rm	0	0	0	0	1	1	Rn	Rd
		U																

Scalar variant

```
SQADD <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Vector

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	size	1	Rm	0	0	0	0	1	1	Rn	Rd
		U																

Vector variant

```
SQADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D when size = 11

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer sum;
boolean sat;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = element1 + element2;
    (Elem[result, e, esize], sat) = SatQ(sum, esize, unsigned);
    if sat then FPSR.QC = '1';

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.284 SQDMLAL, SQDMLAL2 (by element)

Signed saturating Doubling Multiply-Add Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, and accumulates the final results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQDMLAL` instruction extracts vector elements from the lower half of the first source register, while the `SQDMLAL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31 30	29	28	27 26 25 24	23 22	21	20	19 16	15	14	13 12	11	10	9 5	4 0
0 1	0	1	1 1 1 1	size	L	M	Rm	0	0	1 1	H	0	Rn	Rd
									o2					

Scalar variant

```
SQDMLAL <Va><d>, <Vb><n>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o2 == '1');
```

Vector

31 30	29	28	27 26 25 24	23 22	21	20	19 16	15	14	13 12	11	10	9 5	4 0
0	Q	0	0 1 1 1	size	L	M	Rm	0	0	1 1	H	0	Rn	Rd
									o2					

Vector variant

```
SQDMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

`4S` when `size = 01`

`2D` when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`4H` when `size = 01`, `Q = 0`

`8H` when `size = 01`, `Q = 1`

`2S` when `size = 10`, `Q = 0`

`4S` when `size = 10`, `Q = 1`

The following encodings are reserved:

- `size = 00`, `Q = x`.
- `size = 11`, `Q = x`.

`<Va>` Is the destination width specifier, encoded in the "size" field. It can have the following values:

`S` when `size = 01`

`D` when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:

0:Rm when size = 01

M:Rm when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<index> Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

H:L:M when size = 01

H:L when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2 * esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.285 SQDMLAL, SQDMLAL2 (vector)

Signed saturating Doubling Multiply-Add Long. This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, doubles the results, and accumulates the final results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQDMLAL` instruction extracts each source vector from the lower half of each source register, while the `SQDMLAL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31 30	29	28	27 26 25 24	23 22	21	20 16	15 14	13	12	11 10	9 5	4 0
0 1	0	1	1 1 1 0	size	1	Rm	1 0	0	1	0 0	Rn	Rd
								o1				

Scalar variant

```
SQDMLAL <Va><d>, <Vb><n>, <Vb><m>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o1 == '1');
```

Vector

31 30	29	28	27 26 25 24	23 22	21	20 16	15 14	13	12	11 10	9 5	4 0
0 Q	0	0	1 1 1 0	size	1	Rm	1 0	0	1	0 0	Rn	Rd
								o1				

Vector variant

```
SQDMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

`4S` when `size = 01`

`2D` when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`4H` when `size = 01`, `Q = 0`

`8H` when `size = 01`, `Q = 1`

`2S` when `size = 10`, `Q = 0`

`4S` when `size = 10`, `Q = 1`

The following encodings are reserved:

- `size = 00`, `Q = x`.
- `size = 11`, `Q = x`.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

`<Va>` Is the destination width specifier, encoded in the "size" field. It can have the following values:

`S` when `size = 01`

`D` when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

`<Vb>` Is the source width specifier, encoded in the "size" field. It can have the following values:

`H` when `size = 01`

`S` when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<m>` Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2 * esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.286 SQDMLSL, SQDMLSL2 (by element)

Signed saturating Doubling Multiply-Subtract Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, and subtracts the final results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQDMLSL` instruction extracts vector elements from the lower half of the first source register, while the `SQDMLSL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31 30 29 28	27 26 25 24	23 22	21	20	19 16	15	14	13 12	11	10	9 5	4 0
0 1 0	1 1 1 1 1	size	L	M	Rm	0	1	1 1	H	0	Rn	Rd
							o2					

Scalar variant

```
SQDMLSL <Va><d>, <Vb><n>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o2 == '1');
```

Vector

31 30 29 28	27 26 25 24	23 22	21	20	19 16	15	14	13 12	11	10	9 5	4 0
0 Q 0	0 1 1 1 1	size	L	M	Rm	0	1	1 1	H	0	Rn	Rd
							o2					

Vector variant

```
SQDMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

`4S` when `size = 01`

`2D` when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`4H` when `size = 01`, `Q = 0`

`8H` when `size = 01`, `Q = 1`

`2S` when `size = 10`, `Q = 0`

`4S` when `size = 10`, `Q = 1`

The following encodings are reserved:

- `size = 00`, `Q = x`.
- `size = 11`, `Q = x`.

`<Va>` Is the destination width specifier, encoded in the "size" field. It can have the following values:

`S` when `size = 01`

`D` when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:

0:Rm when size = 01

M:Rm when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<index> Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

H:L:M when size = 01

H:L when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
        (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2 * esize);
        if sat1 || sat2 then FPSR.QC = '1';

    V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.287 SQDMLSL, SQDMLSL2 (vector)

Signed saturating Doubling Multiply-Subtract Long. This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, doubles the results, and subtracts the final results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQDMLSL` instruction extracts each source vector from the lower half of each source register, while the `SQDMLSL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31 30 29 28	27 26 25 24	23 22	21	20 … 16	15 14	13	12	11 10	9 … 5	4 … 0
0 1 0	1 1 1 1 0	size	1	Rm	1 0	1	1	0 0	Rn	Rd
						o1				

Scalar variant

```
SQDMLSL <Va><d>, <Vb><n>, <Vb><m>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o1 == '1');
```

Vector

31	30	29 28	27 26 25 24	23 22	21	20 … 16	15 14	13	12	11 10	9 … 5	4 … 0
0	Q	0	0 1 1 1 0	size	1	Rm	1 0	1	1	0 0	Rn	Rd
								o1				

Vector variant

```
SQDMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

`4S` when `size = 01`

`2D` when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`4H` when `size = 01`, `Q = 0`

`8H` when `size = 01`, `Q = 1`

`2S` when `size = 10`, `Q = 0`

`4S` when `size = 10`, `Q = 1`

The following encodings are reserved:

- `size = 00`, `Q = x`.
- `size = 11`, `Q = x`.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

`<Va>` Is the destination width specifier, encoded in the "size" field. It can have the following values:

`S` when `size = 01`

`D` when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

`<Vb>` Is the source width specifier, encoded in the "size" field. It can have the following values:

`H` when `size = 01`

`S` when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<m>` Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2 * esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.288 SQDMULH (by element)

Signed saturating Doubling Multiply returning High half (by element). This instruction multiplies each vector element in the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see SQRDMULH (by element).

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31 30 29 28	27 26 25 24	23 22	21	20	19 16	15 14 13 12	11	10	9 5	4 0
0 1 0	1 1 1 1 1	size	L	M	Rm	1 1 0 0	H	0	Rn	Rd
						op				

Scalar variant

```
SQDMULH <V><d>, <V><n>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean round = (op == '1');
```

Vector

31 30 29 28	27 26 25 24	23 22	21	20	19 16	15 14 13 12	11	10	9 5	4 0
0 Q 0	0 1 1 1 1	size	L	M	Rm	1 1 0 0	H	0	Rn	Rd
						op				

Vector variant

```
SQDMULH <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean round = (op == '1');
```

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:

0:Rm when size = 01

M:Rm when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<index> Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

H:L:M when size = 01

H:L when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    product = (2 * element1 * element2) + round_const;
    // The following only saturates if element1 and element2 equal -(2^(esize-1))
    (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.289 SQDMULH (vector)

Signed saturating Doubling Multiply returning High half. This instruction multiplies the values of corresponding elements of the two source SIMD&FP registers, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see SQRDMULH (vector).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	0	1	1	1	1	0	size	1	Rm	1	0	1	1	0	1	Rn	Rd
		U																

Scalar variant

```
SQDMULH <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = (U == '1');
```

Vector

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	size	1	Rm	1	0	1	1	0	1	Rn	Rd
		U																

Vector variant

```
SQDMULH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = (U == '1');
```

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

H when size = 01

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    product = (2 * element1 * element2) + round_const;
    (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.290 SQDMULL, SQDMULL2 (by element)

Signed saturating Doubling Multiply Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQDMULL` instruction extracts the first source vector from the lower half of the first source register, while the `SQDMULL2` instruction extracts the first source vector from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31 30	29	28	27 26 25 24	23 22	21	20	19 16	15 14 13 12	11	10	9 5	4 0
0 1	0	1	1 1 1 1	size	L	M	Rm	1 0 1 1	H	0	Rn	Rd

Scalar variant

```
SQDMULL <Va><d>, <Vb><n>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;
```

Vector

31	30	29	28	27 26 25 24	23 22	21	20	19 16	15 14 13 12	11	10	9 5	4 0
0	Q	0	0	1 1 1 1	size	L	M	Rm	1 0 1 1	H	0	Rn	Rd

Vector variant

```
SQDMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

[absent] when Q = 0

[present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "size" field. It can have the following values:

4S when size = 01

2D when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Va> Is the destination width specifier, encoded in the "size" field. It can have the following values:

S when size = 01

D when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:

`0:Rm` when `size = 01`

`M:Rm` when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

Restricted to V0-V15 when element size <Ts> is H.

`<Ts>` Is an element size specifier, encoded in the "size" field. It can have the following values:

`H` when `size = 01`

`S` when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

`<index>` Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

`H:L:M` when `size = 01`

`H:L` when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    (product, sat) = SignedSatQ(2 * element1 * element2, 2 * esize);
    Elem[result, e, 2*esize] = product;
    if sat then FPSR.QC = '1';

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.291 SQDMULL, SQDMULL2 (vector)

Signed saturating Doubling Multiply Long. This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQDMULL` instruction extracts each source vector from the lower half of each source register, while the `SQDMULL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31 30	29	28 27 26 25	24	23 22	21	20 16	15 14 13 12	11 10	9 5	4 0
0 1	0	1 1 1 1	0	size	1	Rm	1 1 0 1	0 0	Rn	Rd

Scalar variant

```
SQDMULL <Va><d>, <Vb><n>, <Vb><m>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;
```

Vector

31	30	29	28 27 26 25	24	23 22	21	20 16	15 14 13 12	11 10	9 5	4 0
0	Q	0	0 1 1 1	0	size	1	Rm	1 1 0 1	0 0	Rn	Rd

Vector variant

```
SQDMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

`4S` when `size = 01`

`2D` when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`4H` when `size = 01`, `Q = 0`

`8H` when `size = 01`, `Q = 1`

`2S` when `size = 10`, `Q = 0`

`4S` when `size = 10`, `Q = 1`

The following encodings are reserved:

- `size = 00`, `Q = x`.
- `size = 11`, `Q = x`.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

`<Va>` Is the destination width specifier, encoded in the "size" field. It can have the following values:

`S` when `size = 01`

`D` when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

`<Vb>` Is the source width specifier, encoded in the "size" field. It can have the following values:

`H` when `size = 01`

`S` when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<m>` Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat) = SignedSatQ(2 * element1 * element2, 2 * esize);
    Elem[result, e, 2*esize] = product;
    if sat then FPSR.QC = '1';

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.292 SQNEG

Signed saturating Negate. This instruction reads each vector element from the source SIMD&FP register, negates each value, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31 30	29	28 27 26 25 24	23 22	21 20 19 18 17 16	15 14 13 12	11 10	9 5	4 0
0 1	1	1 1 1 1 0	size	1 0 0 0 0	0 0 1 1 1	1 0	Rn	Rd
	U							

Scalar variant

```
SQNEG <V><d>, <V><n>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean neg = (U == '1');
```

Vector

31 30	29	28 27 26 25 24	23 22	21 20 19 18 17 16	15 14 13 12	11 10	9 5	4 0
0 Q	1	0 1 1 1 0	size	1 0 0 0 0	0 0 1 1 1	1 0	Rn	Rd
	U							

Vector variant

```
SQNEG <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D when size = 11

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);
    (Elem[result, e, esize], sat) = SignedSatQ(element, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.293 SQRDMLAH (by element)

Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (by element). This instruction multiplies the vector elements of the first source SIMD&FP register with the value of a vector element of the second source SIMD&FP register without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

ARMv8.1

31	30	29	28	27	26	25	24	23 22	21	20	19 16	15	14	13	12	11	10	9 5	4 0
0	1	1	1	1	1	1	1	size	L	M	Rm	1	1	0	1	H	0	Rn	Rd
														S					

Scalar variant

```
SQRDMLAH <V><d>, <V><n>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
if !HaveQRDMLAHExt() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Vector

ARMv8.1

31	30	29	28	27	26	25	24	23 22	21	20	19 16	15	14	13	12	11	10	9 5	4 0
0	Q	1	0	1	1	1	1	size	L	M	Rm	1	1	0	1	H	0	Rn	Rd
														S					

Vector variant

```
SQRDMLAH <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
if !HaveQRDMLAHExt() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:

0:Rm when size = 01

M:Rm when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Restricted to V0-V15 when element size <Ts> is H.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Ts> Is an element size specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<index> Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

H:L:M when size = 01

H:L when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    if sub_op then
        accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
    else
        accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
    (Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.294 SQRDMLAH (vector)

Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (vector). This instruction multiplies the vector elements of the first source SIMD&FP register with the corresponding vector elements of the second source SIMD&FP register without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

ARMv8.1

31 30	29	28	27 26 25 24	23 22	21	20 16	15	14 13 12	11	10	9 5	4 0
0 1	1	1	1 1 1 0	size	0	Rm	1	0 0 0	0	1	Rn	Rd
									S			

Scalar variant

```
SQRDMLAH <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
if !HaveQRDMLAHExt() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Vector

ARMv8.1

31 30	29	28	27 26 25 24	23 22	21	20 16	15	14 13 12	11	10	9 5	4 0
0 Q	1	0	1 1 1 0	size	0	Rm	1	0 0 0	0	1	Rn	Rd
									S			

Vector variant

```
SQRDMLAH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !HaveQRDMLAHExt() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer elements = datasize DIV esize;
boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    if sub_op then
        accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
    else
        accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
    (Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.295 SQRDMLSH (by element)

Signed Saturating Rounding Doubling Multiply Subtract returning High Half (by element). This instruction multiplies the vector elements of the first source SIMD&FP register with the value of a vector element of the second source SIMD&FP register without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

ARMv8.1

31 30 29 28	27 26 25 24	23 22 21 20	19 16	15 14 13 12	11 10 9	5 4	0
0 1 1	1 1 1 1	size L M	Rm	1 1 1 1 (S)	H 0	Rn	Rd

Scalar variant

```
SQRDMLSH <V><d>, <V><n>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
if !HaveQRDMLAHExt() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Vector

ARMv8.1

31 30 29 28	27 26 25 24	23 22 21 20	19 16	15 14 13 12	11 10 9	5 4	0
0 Q 1	0 1 1 1	size L M	Rm	1 1 1 1 (S)	H 0	Rn	Rd

Vector variant

```
SQRDMLSH <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
if !HaveQRDMLAHExt() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:

0:Rm when size = 01

M:Rm when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Restricted to V0-V15 when element size <Ts> is H.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Ts> Is an element size specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<index> Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

H:L:M when size = 01

H:L when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    if sub_op then
        accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
    else
        accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
    (Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.296 SQRDMLSH (vector)

Signed Saturating Rounding Doubling Multiply Subtract returning High Half (vector). This instruction multiplies the vector elements of the first source SIMD&FP register with the corresponding vector elements of the second source SIMD&FP register without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

ARMv8.1

31 30	29	28	27 26 25 24	23 22	21	20 16	15	14 13 12	11	10	9 5	4 0
0 1	1	1	1 1 1 0	size	0	Rm	1	0 0 0	1	1	Rn	Rd
									S			

Scalar variant

```
SQRDMLSH <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
if !HaveQRDMLAHExt() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Vector

ARMv8.1

31 30	29	28	27 26 25 24	23 22	21	20 16	15	14 13 12	11	10	9 5	4 0
0 Q	1	0	1 1 1 0	size	0	Rm	1	0 0 0	1	1	Rn	Rd
									S			

Vector variant

```
SQRDMLSH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !HaveQRDMLAHExt() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer elements = datasize DIV esize;
boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    if sub_op then
        accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
    else
        accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
    (Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.297 SQRDMULH (by element)

Signed saturating Rounding Doubling Multiply returning High half (by element). This instruction multiplies each vector element in the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see SQDMULH (by element).

If any of the results overflows, they are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23 22	21	20	19 16	15	14	13	12	11	10	9 5	4 0
0	1	0	1	1	1	1	1	size	L	M	Rm	1	1	0	1	H	0	Rn	Rd
															op				

Scalar variant

```
SQRDMULH <V><d>, <V><n>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean round = (op == '1');
```

Vector

31	30	29	28	27	26	25	24	23 22	21	20	19 16	15	14	13	12	11	10	9 5	4 0
0	Q	0	0	1	1	1	1	size	L	M	Rm	1	1	0	1	H	0	Rn	Rd
															op				

Vector variant

```
SQRDMULH <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean round = (op == '1');
```

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:

0:Rm when size = 01

M:Rm when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<index> Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

H:L:M when size = 01

H:L when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    product = (2 * element1 * element2) + round_const;
    // The following only saturates if element1 and element2 equal -(2^(esize-1))
    (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.298 SQRDMULH (vector)

Signed saturating Rounding Doubling Multiply returning High half. This instruction multiplies the values of corresponding elements of the two source SIMD&FP registers, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see SQDMULH (vector).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	1	1	1	1	1	0	size	1	Rm	1	0	1	1	0	1	Rn	Rd
		U																

Scalar variant

```
SQRDMULH <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = (U == '1');
```

Vector

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size	1	Rm	1	0	1	1	0	1	Rn	Rd
		U																

Vector variant

```
SQRDMULH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = (U == '1');
```

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

H when size = 01

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    product = (2 * element1 * element2) + round_const;
    (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.299 SQRSHL

Signed saturating Rounding Shift Left (register). This instruction takes each vector element in the first source SIMD&FP register, shifts it by a value from the least significant byte of the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded. For truncated results, see SQSHL (register).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23 22	21	20 16	15	14	13	12	11	10	9 5	4 0
0	1	0	1	1	1	1	0	size	1	Rm	0	1	0	1	1	1	Rn	Rd
		U												R	S			

Scalar variant

```
SQRSHL <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;
```

Vector

31	30	29	28	27	26	25	24	23 22	21	20 16	15	14	13	12	11	10	9 5	4 0
0	Q	0	0	1	1	1	0	size	1	Rm	0	1	0	1	1	1	Rn	Rd
		U												R	S			

Vector variant

```
SQRSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1
- 2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1);    // 0 for left shift, 2^(n-1) for right shift
    element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.300 SQRSHRN, SQRSHRN2

Signed saturating Rounded Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are signed integer values. The destination vector elements are half as long as the source vector elements. The results are rounded. For truncated results, see SQSHRN, SQSHRN2.

The SQRSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQRSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31 30	29	28 27 26 25 24	23 22	19 18 16	15 14 13 12	11	10	9 5	4 0
0 1	0	1 1 1 1 1 0	!=0000	immb	1 0 0 1	1	1	Rn	Rd
	U		immh			op			

Scalar variant

```
SQRSHRN <Vb><d>, <Va><n>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

Vector

31 30	29	28 27 26 25 24	23 22	19 18 16	15 14 13 12	11	10	9 5	4 0
0 Q	0	0 1 1 1 1 0	!=0000	immb	1 0 0 1	1	1	Rn	Rd
	U		immh			op			

Vector variant

```
SQRSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3> == '1' then UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Tb>` Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

`8B` when `immh = 0001`, `Q = 0`

`16B` when `immh = 0001`, `Q = 1`

`4H` when `immh = 001x`, `Q = 0`

`8H` when `immh = 001x`, `Q = 1`

`2S` when `immh = 01xx`, `Q = 0`

`4S` when `immh = 01xx`, `Q = 1`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`, `Q = x`.

The encoding `immh = 1xxx`, `Q = x` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<Ta>` Is an arrangement specifier, encoded in the "immh" field. It can have the following values:

`8H` when `immh = 0001`

`4S` when `immh = 001x`

`2D` when `immh = 01xx`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`.

The encoding `immh = 1xxx` is reserved.

`<Vb>` Is the destination width specifier, encoded in the "immh" field. It can have the following values:

`B` when `immh = 0001`

`H` when `immh = 001x`

`S` when `immh = 01xx`

The following encodings are reserved:

- `immh = 0000`.
- `immh = 1xxx`.

`<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.

`<Va>` Is the source width specifier, encoded in the "immh" field. It can have the following values:

`H` when `immh = 0001`

`S` when `immh = 001x`

`D` when `immh = 01xx`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following encodings are reserved:

- `immh = 0000`.
- `immh = 1xxx`.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<shift>` For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in the "immh:immb" field. It can have the following values:

`(16-UInt(immh:immb))` when `immh = 0001`

`(32-UInt(immh:immb))` when `immh = 001x`

`(64-UInt(immh:immb))` when `immh = 01xx`

The following encodings are reserved:

- `immh = 0000`.
- `immh = 1xxx`.

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in the "immh:immb" field. It can have the following values:

`(16-UInt(immh:immb))` when `immh = 0001`

`(32-UInt(immh:immb))` when `immh = 001x`

`(64-UInt(immh:immb))` when `immh = 01xx`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`.

The encoding `immh = 1xxx` is reserved.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;

for e = 0 to elements-1
    element = (Int(Elem[operand, e, 2*esize], unsigned) + round_const) >> shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';

Vpart[d, part] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.301 SQRSHRUN, SQRSHRUN2

Signed saturating Rounded Shift Right Unsigned Narrow (immediate). This instruction reads each signed integer value in the vector of the source SIMD&FP register, right shifts each value by an immediate value, saturates the result to an unsigned integer value that is half the original width, places the final result into a vector, and writes the vector to the destination SIMD&FP register. The results are rounded. For truncated results, see SQSHRUN, SQSHRUN2.

The `SQRSHRUN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SQRSHRUN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31 30 29 28	27 26 25 24	23 22	19 18 16	15 14 13 12	11	10 9	5 4	0
0 1 1	1 1 1 1 1 0	!=0000	immb	1 0 0 0	1	1	Rn	Rd
		immh			op			

Scalar variant

```
SQRSHRUN <Vb><d>, <Va><n>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

Vector

31 30 29 28	27 26 25 24	23 22	19 18 16	15 14 13 12	11	10 9	5 4	0
0 Q 1	0 1 1 1 1 0	!=0000	immb	1 0 0 0	1	1	Rn	Rd
		immh			op			

Vector variant

```
SQRSHRUN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

[absent] when Q = 0

[present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B when immh = 0001, Q = 0

16B when immh = 0001, Q = 1

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

See *Advanced SIMD modified immediate* on page C4-354 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = x is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in the "immh" field. It can have the following values:

8H when immh = 0001

4S when immh = 001x

2D when immh = 01xx

See *Advanced SIMD modified immediate* on page C4-354 when immh = 0000.

The encoding immh = 1xxx is reserved.

<Vb> Is the destination width specifier, encoded in the "immh" field. It can have the following values:

B when immh = 0001

H when immh = 001x

S when immh = 01xx

The following encodings are reserved:

- immh = 0000.
- immh = 1xxx.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in the "immh" field. It can have the following values:

H when immh = 0001

S when immh = 001x

D when immh = 01xx

The following encodings are reserved:

- immh = 0000.
- immh = 1xxx.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<shift>` For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in the "immh:immb" field. It can have the following values:

`(16-UInt(immh:immb))` when `immh = 0001`

`(32-UInt(immh:immb))` when `immh = 001x`

`(64-UInt(immh:immb))` when `immh = 01xx`

The following encodings are reserved:

- `immh = 0000`.
- `immh = 1xxx`.

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in the "immh:immb" field. It can have the following values:

`(16-UInt(immh:immb))` when `immh = 0001`

`(32-UInt(immh:immb))` when `immh = 001x`

`(64-UInt(immh:immb))` when `immh = 01xx`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`.

The encoding `immh = 1xxx` is reserved.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;

for e = 0 to elements-1
    element = (SInt(Elem[operand, e, 2*esize]) + round_const) >> shift;
    (Elem[result, e, esize], sat) = UnsignedSatQ(element, esize);
    if sat then FPSR.QC = '1';

Vpart[d, part] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.302 SQSHL (immediate)

Signed saturating Shift Left (immediate). This instruction reads each vector element in the source SIMD&FP register, shifts each result by an immediate value, places the final result in a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see UQRSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31 30	29	28	27 26 25 24	23 22	19 18 16	15 14 13	12	11 10	9 5	4 0
0 1	0	1	1 1 1 1	1 0	!=0000	immb	0 1 1	1	0 1	Rn	Rd
	U				immh			op			

Scalar variant

```
SQSHL <V><d>, <V><n>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
    when '00' UNDEFINED;
    when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
    when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
    when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
```

Vector

31	30	29	28	27 26 25 24	23 22	19 18 16	15 14 13	12	11 10	9 5	4 0
0	Q	0	0	1 1 1 1	1 0	!=0000	immb	0 1 1	1	0 1	Rn	Rd
		U				immh			op			

Vector variant

```
SQSHL <Vd>.<T>, <Vn>.<T>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
    when '00' UNDEFINED;
    when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
    when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
    when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
```

Assembler symbols

`<V>` Is a width specifier, encoded in the "immh" field. It can have the following values:

`B`	when `immh = 0001`
`H`	when `immh = 001x`
`S`	when `immh = 01xx`
`D`	when `immh = 1xxx`

The encoding `immh = 0000` is reserved.

`<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

`8B`	when `immh = 0001`, `Q = 0`
`16B`	when `immh = 0001`, `Q = 1`
`4H`	when `immh = 001x`, `Q = 0`
`8H`	when `immh = 001x`, `Q = 1`
`2S`	when `immh = 01xx`, `Q = 0`
`4S`	when `immh = 01xx`, `Q = 1`
`2D`	when `immh = 1xxx`, `Q = 1`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`, `Q = x`.

The encoding `immh = 1xxx`, `Q = 0` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<shift>` For the scalar variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in the "immh:immb" field. It can have the following values:

`(UInt(immh:immb)-8)`	when `immh = 0001`
`(UInt(immh:immb)-16)`	when `immh = 001x`
`(UInt(immh:immb)-32)`	when `immh = 01xx`
`(UInt(immh:immb)-64)`	when `immh = 1xxx`

The encoding `immh = 0000` is reserved.

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in the "immh:immb" field. It can have the following values:

`(UInt(immh:immb)-8)`	when `immh = 0001`
`(UInt(immh:immb)-16)`	when `immh = 001x`
`(UInt(immh:immb)-32)`	when `immh = 01xx`
`(UInt(immh:immb)-64)`	when `immh = 1xxx`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], src_unsigned) << shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, dst_unsigned);
    if sat then FPSR.QC = '1';

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.303 SQSHL (register)

Signed saturating Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts each element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated. For rounded results, see SQRSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31 30	29	28 27 26 25 24	23 22	21	20 16	15 14 13	12	11	10	9 5	4 0
0 1	0	1 1 1 1 0	size	1	Rm	0 1 0	0	1	1	Rn	Rd
	U						R	S			

Scalar variant

```
SQSHL <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;
```

Vector

31 30	29	28 27 26 25 24	23 22	21	20 16	15 14 13	12	11	10	9 5	4 0
0 Q	0	0 1 1 1 0	size	1	Rm	0 1 0	0	1	1	Rn	Rd
	U						R	S			

Vector variant

```
SQSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1
- 2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1);    // 0 for left shift, 2^(n-1) for right shift
    element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.304 SQSHLU

Signed saturating Shift Left Unsigned (immediate). This instruction reads each signed integer value in the vector of the source SIMD&FP register, shifts each value by an immediate value, saturates the shifted result to an unsigned integer value, places the result in a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see UQRSHL.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23	22	19	18	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	1	0	!=0000			immb	0	1	1	0	0	1	Rn			Rd
		U							immh							op						

Scalar variant

```
SQSHLU <V><d>, <V><n>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
    when '00' UNDEFINED;
    when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
    when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
    when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
```

Vector

31	30	29	28	27	26	25	24	23	22	19	18	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	1	0	!=0000			immb	0	1	1	0	0	1	Rn			Rd
		U							immh							op						

Vector variant

```
SQSHLU <Vd>.<T>, <Vn>.<T>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
    when '00' UNDEFINED;
    when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
    when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
    when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
```

Assembler symbols

`<V>` Is a width specifier, encoded in the "immh" field. It can have the following values:

`B`	when `immh = 0001`
`H`	when `immh = 001x`
`S`	when `immh = 01xx`
`D`	when `immh = 1xxx`

The encoding `immh = 0000` is reserved.

`<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

`8B`	when `immh = 0001`, `Q = 0`
`16B`	when `immh = 0001`, `Q = 1`
`4H`	when `immh = 001x`, `Q = 0`
`8H`	when `immh = 001x`, `Q = 1`
`2S`	when `immh = 01xx`, `Q = 0`
`4S`	when `immh = 01xx`, `Q = 1`
`2D`	when `immh = 1xxx`, `Q = 1`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`, `Q = x`.

The encoding `immh = 1xxx`, `Q = 0` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<shift>` For the scalar variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in the "immh:immb" field. It can have the following values:

`(UInt(immh:immb)-8)`	when `immh = 0001`
`(UInt(immh:immb)-16)`	when `immh = 001x`
`(UInt(immh:immb)-32)`	when `immh = 01xx`
`(UInt(immh:immb)-64)`	when `immh = 1xxx`

The encoding `immh = 0000` is reserved.

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in the "immh:immb" field. It can have the following values:

`(UInt(immh:immb)-8)`	when `immh = 0001`
`(UInt(immh:immb)-16)`	when `immh = 001x`
`(UInt(immh:immb)-32)`	when `immh = 01xx`
`(UInt(immh:immb)-64)`	when `immh = 1xxx`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], src_unsigned) << shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, dst_unsigned);
    if sat then FPSR.QC = '1';

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.305 SQSHRN, SQSHRN2

Signed saturating Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts and truncates each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are signed integer values. The destination vector elements are half as long as the source vector elements. For rounded results, see SQRSHRN, SQRSHRN2.

The `SQSHRN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SQSHRN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31 30 29 28	27 26 25 24	23 22	19 18 16	15 14 13 12	11 10	9 5	4 0
0 1 0	1 1 1 1 1 0	!=0000	immb	1 0 0 1	0 1	Rn	Rd
U		immh			op		

Scalar variant

```
SQSHRN <Vb><d>, <Va><n>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

Vector

31 30 29 28	27 26 25 24	23 22	19 18 16	15 14 13 12	11 10	9 5	4 0
0 Q 0	0 1 1 1 1 0	!=0000	immb	1 0 0 1	0 1	Rn	Rd
U		immh			op		

Vector variant

```
SQSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Tb>` Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

`8B` when `immh = 0001`, `Q = 0`

`16B` when `immh = 0001`, `Q = 1`

`4H` when `immh = 001x`, `Q = 0`

`8H` when `immh = 001x`, `Q = 1`

`2S` when `immh = 01xx`, `Q = 0`

`4S` when `immh = 01xx`, `Q = 1`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`, `Q = x`.

The encoding `immh = 1xxx`, `Q = x` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<Ta>` Is an arrangement specifier, encoded in the "immh" field. It can have the following values:

`8H` when `immh = 0001`

`4S` when `immh = 001x`

`2D` when `immh = 01xx`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`.

The encoding `immh = 1xxx` is reserved.

`<Vb>` Is the destination width specifier, encoded in the "immh" field. It can have the following values:

`B` when `immh = 0001`

`H` when `immh = 001x`

`S` when `immh = 01xx`

The following encodings are reserved:

- `immh = 0000`.
- `immh = 1xxx`.

`<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.

`<Va>` Is the source width specifier, encoded in the "immh" field. It can have the following values:

`H` when `immh = 0001`

`S` when `immh = 001x`

`D` when `immh = 01xx`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following encodings are reserved:

- `immh = 0000`.
- `immh = 1xxx`.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<shift>` For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in the "immh:immb" field. It can have the following values:

`(16-UInt(immh:immb))` when `immh = 0001`

`(32-UInt(immh:immb))` when `immh = 001x`

`(64-UInt(immh:immb))` when `immh = 01xx`

The following encodings are reserved:

- `immh = 0000`.
- `immh = 1xxx`.

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in the "immh:immb" field. It can have the following values:

`(16-UInt(immh:immb))` when `immh = 0001`

`(32-UInt(immh:immb))` when `immh = 001x`

`(64-UInt(immh:immb))` when `immh = 01xx`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`.

The encoding `immh = 1xxx` is reserved.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;

for e = 0 to elements-1
    element = (Int(Elem[operand, e, 2*esize], unsigned) + round_const) >> shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';

Vpart[d, part] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.306 SQSHRUN, SQSHRUN2

Signed saturating Shift Right Unsigned Narrow (immediate). This instruction reads each signed integer value in the vector of the source SIMD&FP register, right shifts each value by an immediate value, saturates the result to an unsigned integer value that is half the original width, places the final result into a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see SQRSHRUN, SQRSHRUN2.

The `SQSHRUN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SQSHRUN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31 30	29	28	27 26 25 24	23 22	19 18 16	15 14 13 12	11	10	9 5	4 0
0 1	1	1	1 1 1 1 1 0	!=0000	immb	1 0 0 0	0	1	Rn	Rd
				immh			op			

Scalar variant

```
SQSHRUN <Vb><d>, <Va><n>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

Vector

31 30	29	28	27 26 25 24	23 22	19 18 16	15 14 13 12	11	10	9 5	4 0
0 Q	1	0	1 1 1 1 0	!=0000	immb	1 0 0 0	0	1	Rn	Rd
				immh			op			

Vector variant

```
SQSHRUN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

[absent] when Q = 0

[present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B when immh = 0001, Q = 0

16B when immh = 0001, Q = 1

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

See *Advanced SIMD modified immediate* on page C4-354 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = x is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in the "immh" field. It can have the following values:

8H when immh = 0001

4S when immh = 001x

2D when immh = 01xx

See *Advanced SIMD modified immediate* on page C4-354 when immh = 0000.

The encoding immh = 1xxx is reserved.

<Vb> Is the destination width specifier, encoded in the "immh" field. It can have the following values:

B when immh = 0001

H when immh = 001x

S when immh = 01xx

The following encodings are reserved:

- immh = 0000.
- immh = 1xxx.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in the "immh" field. It can have the following values:

H when immh = 0001

S when immh = 001x

D when immh = 01xx

The following encodings are reserved:

- immh = 0000.
- immh = 1xxx.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<shift>` For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in the "immh:immb" field. It can have the following values:

`(16-UInt(immh:immb))` when `immh = 0001`

`(32-UInt(immh:immb))` when `immh = 001x`

`(64-UInt(immh:immb))` when `immh = 01xx`

The following encodings are reserved:

- `immh = 0000`.
- `immh = 1xxx`.

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in the "immh:immb" field. It can have the following values:

`(16-UInt(immh:immb))` when `immh = 0001`

`(32-UInt(immh:immb))` when `immh = 001x`

`(64-UInt(immh:immb))` when `immh = 01xx`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`.

The encoding `immh = 1xxx` is reserved.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;

for e = 0 to elements-1
    element = (SInt(Elem[operand, e, 2*esize]) + round_const) >> shift;
    (Elem[result, e, esize], sat) = UnsignedSatQ(element, esize);
    if sat then FPSR.QC = '1';

Vpart[d, part] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.307 SQSUB

Signed saturating Subtract. This instruction subtracts the element values of the second source SIMD&FP register from the corresponding element values of the first source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	0	1	1	1	1	0	size	1	Rm	0	0	1	0	1	1	Rn	Rd
		U																

Scalar variant

```
SQSUB <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Vector

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	size	1	Rm	0	0	1	0	1	1	Rn	Rd
		U																

Vector variant

```
SQSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D when size = 11

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer diff;
boolean sat;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    diff = element1 - element2;
    (Elem[result, e, esize], sat) = SatQ(diff, esize, unsigned);
    if sat then FPSR.QC = '1';

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.308 SQXTN, SQXTN2

Signed saturating extract Narrow. This instruction reads each vector element from the source SIMD&FP register, saturates the value to half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQXTN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SQXTN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	size		1	0	0	0	0	1	0	1	0	0	1	0	Rn		Rd	
		U																							

Scalar variant

```
SQXTN <Vb><d>, <Va><n>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer part = 0;
integer elements = 1;

boolean unsigned = (U == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	size		1	0	0	0	0	1	0	1	0	0	1	0	Rn		Rd	
		U																							

Vector variant

```
SQXTN{2} <Vd>.<Tb>, <Vn>.<Ta>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

[absent] when Q = 0

[present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in the "size" field. It can have the following values:

8H when size = 00

4S when size = 01

2D when size = 10

The encoding size = 11 is reserved.

<Vb> Is the destination width specifier, encoded in the "size" field. It can have the following values:

B when size = 00

H when size = 01

S when size = 10

The encoding size = 11 is reserved.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va> Is the source width specifier, encoded in the "size" field. It can have the following values:

H when size = 00

S when size = 01

D when size = 10

The encoding size = 11 is reserved.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;
bits(2*esize) element;
boolean sat;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    (Elem[result, e, esize], sat) = SatQ(Int(element, unsigned), esize, unsigned);
    if sat then FPSR.QC = '1';

Vpart[d, part] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.309 SQXTUN, SQXTUN2

Signed saturating extract Unsigned Narrow. This instruction reads each signed integer value in the vector of the source SIMD&FP register, saturates the value to an unsigned integer value that is half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQXTUN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SQXTUN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31 30	29	28 27 26 25 24	23 22	21 20 19 18 17 16	15 14 13 12	11 10	9 5	4 0
0 1	1	1 1 1 1 0	size	1 0 0 0 0	1 0 0 1 0	1 0	Rn	Rd

Scalar variant

```
SQXTUN <Vb><d>, <Va><n>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer part = 0;
integer elements = 1;
```

Vector

31 30	29	28 27 26 25 24	23 22	21 20 19 18 17 16	15 14 13 12	11 10	9 5	4 0
0 Q	1	0 1 1 1 0	size	1 0 0 0 0	1 0 0 1 0	1 0	Rn	Rd

Vector variant

```
SQXTUN{2} <Vd>.<Tb>, <Vn>.<Ta>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Tb>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B` when `size = 00`, `Q = 0`

`16B` when `size = 00`, `Q = 1`

`4H` when `size = 01`, `Q = 0`

`8H` when `size = 01`, `Q = 1`

`2S` when `size = 10`, `Q = 0`

`4S` when `size = 10`, `Q = 1`

The encoding `size = 11`, `Q = x` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

`8H` when `size = 00`

`4S` when `size = 01`

`2D` when `size = 10`

The encoding `size = 11` is reserved.

`<Vb>` Is the destination width specifier, encoded in the "size" field. It can have the following values:

`B` when `size = 00`

`H` when `size = 01`

`S` when `size = 10`

The encoding `size = 11` is reserved.

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

`<Va>` Is the source width specifier, encoded in the "size" field. It can have the following values:

`H` when `size = 00`

`S` when `size = 01`

`D` when `size = 10`

The encoding `size = 11` is reserved.

`<n>` Is the number of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;
bits(2*esize) element;
boolean sat;

for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    (Elem[result, e, esize], sat) = UnsignedSatQ(SInt(element), esize);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if sat then FPSR.QC = '1';

Vpart[d, part] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.310 SRHADD

Signed Rounding Halving Add. This instruction adds corresponding signed integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see SHADD.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 16	15	14	13	12	11	10	9 5	4 0
0	Q	0	0	1	1	1	0	size	1	Rm	0	0	0	1	0	1	Rn	Rd
		U																

Three registers of the same type variant

```
SRHADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

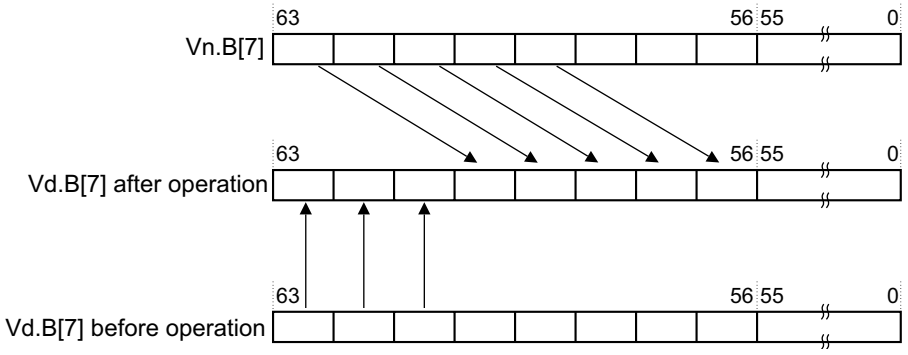
```
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    Elem[result, e, esize] = (element1+element2+1)<esize:1>;

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.311 SRI

Shift Right and Insert (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each vector element by an immediate value, and inserts the result into the corresponding vector element in the destination SIMD&FP register such that the new zero bits created by the shift are not inserted but retain their existing value. Bits shifted out of the right of each vector element of the source register are lost.

The following figure shows the operation of shift right by 3 for an 8-bit vector element.



Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31 30	29	28 27 26 25 24	23 22	19 18 16	15 14 13 12	11	10 9 5	4 0
0 1	1	1 1 1 1 1 0	!=0000 immh	immb	0 1 0 0 0	1	Rn	Rd

Scalar variant

```
SRI <V><d>, <V><n>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
```

Vector

31 30	29	28 27 26 25 24	23 22	19 18 16	15 14 13 12	11	10 9 5	4 0
0 Q	1	0 1 1 1 1 0	!=0000 immh	immb	0 1 0 0 0	1	Rn	Rd

Vector variant

```
SRI <Vd>.<T>, <Vn>.<T>, #<shift>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
```

Assembler symbols

`<V>` Is a width specifier, encoded in the "immh" field. It can have the following values:

`D` when `immh = 1xxx`

The encoding `immh = 0xxx` is reserved.

`<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

`8B` when `immh = 0001`, `Q = 0`

`16B` when `immh = 0001`, `Q = 1`

`4H` when `immh = 001x`, `Q = 0`

`8H` when `immh = 001x`, `Q = 1`

`2S` when `immh = 01xx`, `Q = 0`

`4S` when `immh = 01xx`, `Q = 1`

`2D` when `immh = 1xxx`, `Q = 1`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`, `Q = x`.

The encoding `immh = 1xxx`, `Q = 0` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<shift>` For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in the "immh:immb" field. It can have the following values:

`(128-UInt(immh:immb))` when `immh = 1xxx`

The encoding `immh = 0xxx` is reserved.

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in the "immh:immb" field. It can have the following values:

`(16-UInt(immh:immb))` when `immh = 0001`

`(32-UInt(immh:immb))` when `immh = 001x`

`(64-UInt(immh:immb))` when `immh = 01xx`

`(128-UInt(immh:immb))` when `immh = 1xxx`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2 = V[d];
bits(datasize) result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(esize) mask = LSR(Ones(esize), shift);
bits(esize) shifted;

for e = 0 to elements-1
    shifted = LSR(Elem[operand, e, esize], shift);
    Elem[result, e, esize] = (Elem[operand2, e, esize] AND NOT(mask)) OR shifted;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.312 SRSHL

Signed Rounding Shift Left (register). This instruction takes each signed integer value in the vector of the first source SIMD&FP register, shifts it by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift. For a truncating shift, see SSHL.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23 22	21	20 16	15	14	13	12	11	10	9 5	4 0
0	1	0	1	1	1	1	0	size	1	Rm	0	1	0	1	0	1	Rn	Rd
		U												R	S			

Scalar variant

```
SRSHL <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;
```

Vector

31	30	29	28	27	26	25	24	23 22	21	20 16	15	14	13	12	11	10	9 5	4 0
0	Q	0	0	1	1	1	0	size	1	Rm	0	1	0	1	0	1	Rn	Rd
		U												R	S			

Vector variant

```
SRSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

D when size = 11

The following encodings are reserved:

- size = 0x.
- size = 10.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1);    // 0 for left shift, 2^(n-1) for right shift
    element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.313 SRSHR

Signed Rounding Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, places the final result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are rounded. For truncated results, see SSHR.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23	22	19	18	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	1	0	!=0000	immb			0	0	1	0	0	1	Rn	Rd		
		U							immh						o1	o0						

Scalar variant

```
SRSHR <V><d>, <V><n>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	19	18	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	1	0	!=0000	immb			0	0	1	0	0	1	Rn	Rd		
		U							immh						o1	o0						

Vector variant

```
SRSHR <Vd>.<T>, <Vn>.<T>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

<V> Is a width specifier, encoded in the "immh" field. It can have the following values:

D when immh = 1xxx

The encoding immh = 0xxx is reserved.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B when immh = 0001, Q = 0

16B when immh = 0001, Q = 1

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

2D when immh = 1xxx, Q = 1

See *Advanced SIMD modified immediate* on page C4-354 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in the "immh:immb" field. It can have the following values:

(128-UInt(immh:immb)) when immh = 1xxx

The encoding immh = 0xxx is reserved.

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in the "immh:immb" field. It can have the following values:

(16-UInt(immh:immb)) when immh = 0001

(32-UInt(immh:immb)) when immh = 001x

(64-UInt(immh:immb)) when immh = 01xx

(128-UInt(immh:immb)) when immh = 1xxx

See *Advanced SIMD modified immediate* on page C4-354 when immh = 0000.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.314 SRSRA

Signed Rounding Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are rounded. For truncated results, see SSRA.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23	22	19	18	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	1	0	!=0000	immb			0	0	1	1	0	1	Rn		Rd	
		U							immh						o1	o0						

Scalar variant

```
SRSRA <V><d>, <V><n>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	19	18	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	1	0	!=0000	immb			0	0	1	1	0	1	Rn		Rd	
		U							immh						o1	o0						

Vector variant

```
SRSRA <Vd>.<T>, <Vn>.<T>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

<V> Is a width specifier, encoded in the "immh" field. It can have the following values:

D when immh = 1xxx

The encoding immh = 0xxx is reserved.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B when immh = 0001, Q = 0

16B when immh = 0001, Q = 1

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

2D when immh = 1xxx, Q = 1

See *Advanced SIMD modified immediate* on page C4-354 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in the "immh:immb" field. It can have the following values:

(128-UInt(immh:immb)) when immh = 1xxx

The encoding immh = 0xxx is reserved.

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in the "immh:immb" field. It can have the following values:

(16-UInt(immh:immb)) when immh = 0001

(32-UInt(immh:immb)) when immh = 001x

(64-UInt(immh:immb)) when immh = 01xx

(128-UInt(immh:immb)) when immh = 1xxx

See *Advanced SIMD modified immediate* on page C4-354 when immh = 0000.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.315 SSHL

Signed Shift Left (register). This instruction takes each signed integer value in the vector of the first source SIMD&FP register, shifts each value by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift. For a rounding shift, see SRSHL.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23 22	21	20 16	15	14	13	12	11	10	9 5	4 0
0	1	0	1	1	1	1	0	size	1	Rm	0	1	0	0	0	1	Rn	Rd
		U												R	S			

Scalar variant

```
SSHL <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;
```

Vector

31	30	29	28	27	26	25	24	23 22	21	20 16	15	14	13	12	11	10	9 5	4 0
0	Q	0	0	1	1	1	0	size	1	Rm	0	1	0	0	0	1	Rn	Rd
		U												R	S			

Vector variant

```
SSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

D when size = 11

The following encodings are reserved:

- size = 0x.
- size = 10.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1);    // 0 for left shift, 2^(n-1) for right shift
    element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.316 SSHLL, SSHLL2

Signed Shift Left Long (immediate). This instruction reads each vector element from the source SIMD&FP register, left shifts each vector element by the specified shift amount, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The `SSHLL` instruction extracts vector elements from the lower half of the source register, while the `SSHLL2` instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias SXTL, SXTL2. See *Alias conditions* for details of when each alias is preferred.

31	30	29	28	27	26	25	24	23	22	19	18	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	1	0	!=0000	immb			1	0	1	0	0	1	Rn		Rd	
		U							immh													

Vector variant

```
SSHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;
boolean unsigned = (U == '1');
```

Alias conditions

Alias	is preferred when
SXTL, SXTL2	`immb == '000' && BitCount(immh) == 1`

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "immh" field. It can have the following values:

`8H` when `immh = 0001`

`4S` when `immh = 001x`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`2D` when `immh = 01xx`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`.

The encoding `immh = 1xxx` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

`8B` when `immh = 0001`, `Q = 0`

`16B` when `immh = 0001`, `Q = 1`

`4H` when `immh = 001x`, `Q = 0`

`8H` when `immh = 001x`, `Q = 1`

`2S` when `immh = 01xx`, `Q = 0`

`4S` when `immh = 01xx`, `Q = 1`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`, `Q = x`.

The encoding `immh = 1xxx`, `Q = x` is reserved.

`<shift>` Is the left shift amount, in the range 0 to the source element width in bits minus 1, encoded in the "immh:immb" field. It can have the following values:

`(UInt(immh:immb)-8)` when `immh = 0001`

`(UInt(immh:immb)-16)` when `immh = 001x`

`(UInt(immh:immb)-32)` when `immh = 01xx`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`.

The encoding `immh = 1xxx` is reserved.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part];
bits(datasize*2) result;
integer element;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], unsigned) << shift;
    Elem[result, e, 2*esize] = element<2*esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.317 SSHR

Signed Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, places the final result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are truncated. For rounded results, see SRSHR.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23	22	19	18	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	1	0	!=0000	immb			0	0	0	0	0	1	Rn	Rd		
		U							immh						o1	o0						

Scalar variant

```
SSHR <V><d>, <V><n>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	19	18	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	1	0	!=0000	immb			0	0	0	0	0	1	Rn	Rd		
		U							immh						o1	o0						

Vector variant

```
SSHR <Vd>.<T>, <Vn>.<T>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

<V> Is a width specifier, encoded in the "immh" field. It can have the following values:

D when immh = 1xxx

The encoding immh = 0xxx is reserved.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B when immh = 0001, Q = 0

16B when immh = 0001, Q = 1

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

2D when immh = 1xxx, Q = 1

See *Advanced SIMD modified immediate* on page C4-354 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in the "immh:immb" field. It can have the following values:

(128-UInt(immh:immb)) when immh = 1xxx

The encoding immh = 0xxx is reserved.

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in the "immh:immb" field. It can have the following values:

(16-UInt(immh:immb)) when immh = 0001

(32-UInt(immh:immb)) when immh = 001x

(64-UInt(immh:immb)) when immh = 01xx

(128-UInt(immh:immb)) when immh = 1xxx

See *Advanced SIMD modified immediate* on page C4-354 when immh = 0000.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.318 SSRA

Signed Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are truncated. For rounded results, see SRSRA.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23	22	19	18	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	1	0	!=0000	immb			0	0	0	1	0	1	Rn	Rd		
		U							immh						o1	o0						

Scalar variant

```
SSRA <V><d>, <V><n>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	19	18	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	1	0	!=0000	immb			0	0	0	1	0	1	Rn	Rd		
		U							immh						o1	o0						

Vector variant

```
SSRA <Vd>.<T>, <Vn>.<T>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

<V> Is a width specifier, encoded in the "immh" field. It can have the following values:

D when immh = 1xxx

The encoding immh = 0xxx is reserved.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B when immh = 0001, Q = 0

16B when immh = 0001, Q = 1

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

2D when immh = 1xxx, Q = 1

See *Advanced SIMD modified immediate* on page C4-354 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in the "immh:immb" field. It can have the following values:

(128-UInt(immh:immb)) when immh = 1xxx

The encoding immh = 0xxx is reserved.

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in the "immh:immb" field. It can have the following values:

(16-UInt(immh:immb)) when immh = 0001

(32-UInt(immh:immb)) when immh = 001x

(64-UInt(immh:immb)) when immh = 01xx

(128-UInt(immh:immb)) when immh = 1xxx

See *Advanced SIMD modified immediate* on page C4-354 when immh = 0000.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.319 SSUBL, SSUBL2

Signed Subtract Long. This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. The destination vector elements are twice as long as the source vector elements.

The `SSUBL` instruction extracts each source vector from the lower half of each source register, while the `SSUBL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	size	1	Rm	0	0	1	0	0	0	Rn	Rd
		U											o1					

Three registers, not all the same type variant

```
SSUBL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- `[absent]` when `Q = 0`
- `[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- `8H` when `size = 00`
- `4S` when `size = 01`
- `2D` when `size = 10`

The encoding `size = 11` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `8B` when `size = 00`, `Q = 0`
- `16B` when `size = 00`, `Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.320 SSUBW, SSUBW2

Signed Subtract Wide. This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. All the values in this instruction are signed integer values.

The `SSUBW` instruction extracts the second source vector from the lower half of the second source register, while the `SSUBW2` instruction extracts the second source vector from the upper half of the second source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	size	1	Rm	0	0	1	1	0	0	Rn	Rd
		U											o1					

Three registers, not all the same type variant

```
SSUBW{2} <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- `[absent]` when `Q = 0`
- `[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- `8H` when `size = 00`
- `4S` when `size = 01`
- `2D` when `size = 10`

The encoding `size = 11` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B	when `size = 00`, `Q = 0`
16B	when `size = 00`, `Q = 1`
4H	when `size = 01`, `Q = 0`
8H	when `size = 01`, `Q = 1`
2S	when `size = 10`, `Q = 0`
4S	when `size = 10`, `Q = 1`

The encoding `size = 11`, `Q = x` is reserved.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, 2*esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.321 ST1 (multiple structures)

Store multiple single-element structures from one, two, three, or four registers. This instruction stores elements to memory from one, two, three, or four SIMD&FP registers, without interleaving. Every element of each register is stored.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15			12	11	10	9	5	4	0
0	Q	0	0	1	1	0	0	0	0	0	0	0	0	0	0	x	x	1	x	size		Rn		Rt	
									L							opcode									

One register variant

Applies when `opcode == 0111`.

```
ST1 { <Vt>.<T> }, [<Xn|SP>]
```

Two registers variant

Applies when `opcode == 1010`.

```
ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

Three registers variant

Applies when `opcode == 0110`.

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

Four registers variant

Applies when `opcode == 0010`.

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20		16	15			12	11	10	9	5	4	0
0	Q	0	0	1	1	0	0	1	0	0		Rm		x	x	1	x	size		Rn		Rt	
									L					opcode									

One register, immediate offset variant

Applies when `Rm == 11111 && opcode == 0111`.

```
ST1 { <Vt>.<T> }, [<Xn|SP>], <imm>
```

One register, register offset variant

Applies when `Rm != 11111 && opcode == 0111`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
ST1 { <Vt>.<T> }, [<Xn|SP>], <Xm>
```

Two registers, immediate offset variant

Applies when `Rm == 11111 && opcode == 1010`.

```
ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Two registers, register offset variant

Applies when `Rm != 11111 && opcode == 1010`.

```
ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

Three registers, immediate offset variant

Applies when `Rm == 11111 && opcode == 0110`.

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Three registers, register offset variant

Applies when `Rm != 11111 && opcode == 0110`.

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

Four registers, immediate offset variant

Applies when `Rm == 11111 && opcode == 0010`.

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

Four registers, register offset variant

Applies when `Rm != 11111 && opcode == 0010`.

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler symbols

`<Vt>` Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B`	when `size = 00`, `Q = 0`
`16B`	when `size = 00`, `Q = 1`
`4H`	when `size = 01`, `Q = 0`
`8H`	when `size = 01`, `Q = 1`
`2S`	when `size = 10`, `Q = 0`
`4S`	when `size = 10`, `Q = 1`
`1D`	when `size = 11`, `Q = 0`
`2D`	when `size = 11`, `Q = 1`

`<Vt2>` Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

`<Vt3>` Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Vt4>` Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<imm>` For the one register, immediate offset variant: is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

`#8` when `Q = 0`

`#16` when `Q = 1`

For the two registers, immediate offset variant: is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

`#16` when `Q = 0`

`#32` when `Q = 1`

For the three registers, immediate offset variant: is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

`#24` when `Q = 0`

`#48` when `Q = 1`

For the four registers, immediate offset variant: is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

`#32` when `Q = 0`

`#64` when `Q = 1`

`<Xm>` Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared decode for all encodings

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;    // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
    otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.322 ST1 (single structure)

Store a single-element structure from one lane of one register. This instruction stores the specified element of a SIMD&FP register to memory.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	1	0	0	0	0	0	0	0	0	x	x	0	S	size			Rn		Rt
									L	R						opcode									

8-bit variant

Applies when `opcode == 000`.

```
ST1 { <Vt>.B }[<index>], [<Xn|SP>]
```

16-bit variant

Applies when `opcode == 010 && size == x0`.

```
ST1 { <Vt>.H }[<index>], [<Xn|SP>]
```

32-bit variant

Applies when `opcode == 100 && size == 00`.

```
ST1 { <Vt>.S }[<index>], [<Xn|SP>]
```

64-bit variant

Applies when `opcode == 100 && S == 0 && size == 01`.

```
ST1 { <Vt>.D }[<index>], [<Xn|SP>]
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	1	1	0	0	Rm		x	x	0	S	size			Rn		Rt
									L	R			opcode									

8-bit, immediate offset variant

Applies when `Rm == 11111 && opcode == 000`.

```
ST1 { <Vt>.B }[<index>], [<Xn|SP>], #1
```

8-bit, register offset variant

Applies when `Rm != 11111 && opcode == 000`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
ST1 { <Vt>.B }[<index>], [<Xn|SP>], <Xm>
```

16-bit, immediate offset variant

Applies when `Rm == 11111 && opcode == 010 && size == x0`.

```
ST1 { <Vt>.H }[<index>], [<Xn|SP>], #2
```

16-bit, register offset variant

Applies when `Rm != 11111 && opcode == 010 && size == x0`.

```
ST1 { <Vt>.H }[<index>], [<Xn|SP>], <Xm>
```

32-bit, immediate offset variant

Applies when `Rm == 11111 && opcode == 100 && size == 00`.

```
ST1 { <Vt>.S }[<index>], [<Xn|SP>], #4
```

32-bit, register offset variant

Applies when `Rm != 11111 && opcode == 100 && size == 00`.

```
ST1 { <Vt>.S }[<index>], [<Xn|SP>], <Xm>
```

64-bit, immediate offset variant

Applies when `Rm == 11111 && opcode == 100 && S == 0 && size == 01`.

```
ST1 { <Vt>.D }[<index>], [<Xn|SP>], #8
```

64-bit, register offset variant

Applies when `Rm != 11111 && opcode == 100 && S == 0 && size == 01`.

```
ST1 { <Vt>.D }[<index>], [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler symbols

`<Vt>`	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
`<index>`	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared decode for all encodings

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation for all encodings

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.323 ST2 (multiple structures)

Store multiple 2-element structures from two registers. This instruction stores multiple 2-element structures from two SIMD&FP registers to memory, with interleaving. Every element of each register is stored.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	0	0	L	0	0	0	0	0	0	1	0	0	0	size			Rn		Rt

opcode

No offset variant

```
ST2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

Decode for this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	0	1	L	0	Rm					1	0	0	0	size			Rn		Rt

opcode

Immediate offset variant

Applies when `Rm == 11111`.

```
ST2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Register offset variant

Applies when `Rm != 11111`.

```
ST2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler symbols

`<Vt>` Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `8B` when `size = 00`, `Q = 0`
- `16B` when `size = 00`, `Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1
2D	when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

#16	when Q = 0
#32	when Q = 1

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared decode for all encodings

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;    // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
    otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.324 ST2 (single structure)

Store single 2-element structure from one lane of two registers. This instruction stores a 2-element structure to memory from corresponding elements of two SIMD&FP registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	1	0	0	1	0	0	0	0	0	x	x	0	S	size			Rn		Rt
									L	R						opcode									

8-bit variant

Applies when `opcode == 000`.

```
ST2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>]
```

16-bit variant

Applies when `opcode == 010 && size == x0`.

```
ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>]
```

32-bit variant

Applies when `opcode == 100 && size == 00`.

```
ST2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>]
```

64-bit variant

Applies when `opcode == 100 && S == 0 && size == 01`.

```
ST2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>]
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	1	1	0	1	Rm		x	x	0	S	size			Rn		Rt
									L	R			opcode									

8-bit, immediate offset variant

Applies when `Rm == 11111 && opcode == 000`.

```
ST2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], #2
```

8-bit, register offset variant

Applies when `Rm != 11111 && opcode == 000`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
ST2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], <Xm>
```

16-bit, immediate offset variant

Applies when `Rm == 11111 && opcode == 010 && size == x0`.

```
ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], #4
```

16-bit, register offset variant

Applies when `Rm != 11111 && opcode == 010 && size == x0`.

```
ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], <Xm>
```

32-bit, immediate offset variant

Applies when `Rm == 11111 && opcode == 100 && size == 00`.

```
ST2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], #8
```

32-bit, register offset variant

Applies when `Rm != 11111 && opcode == 100 && size == 00`.

```
ST2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], <Xm>
```

64-bit, immediate offset variant

Applies when `Rm == 11111 && opcode == 100 && S == 0 && size == 01`.

```
ST2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], #16
```

64-bit, register offset variant

Applies when `Rm != 11111 && opcode == 100 && S == 0 && size == 01`.

```
ST2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler symbols

`<Vt>`	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
`<Vt2>`	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
`<index>`	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Shared decode for all encodings

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation for all encodings

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.325 ST3 (multiple structures)

Store multiple 3-element structures from three registers. This instruction stores multiple 3-element structures to memory from three SIMD&FP registers, with interleaving. Every element of each register is stored.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	0	0	L	0	0	0	0	0	0	0	1	0	0	size			Rn		Rt

opcode

No offset variant

```
ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

Decode for this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	0	1	L	0	Rm		0	1	0	0	size			Rn		Rt

opcode

Immediate offset variant

Applies when `Rm == 11111`.

```
ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Register offset variant

Applies when `Rm != 11111`.

```
ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler symbols

`<Vt>` Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `8B` when `size = 00`, `Q = 0`
- `16B` when `size = 00`, `Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1
2D	when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

#24	when Q = 0
#48	when Q = 1

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared decode for all encodings

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;    // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
    otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.326 ST3 (single structure)

Store single 3-element structure from one lane of three registers. This instruction stores a 3-element structure to memory from corresponding elements of three SIMD&FP registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	1	0	0	0	0	0	0	0	0	x	x	1	S	size			Rn		Rt
									L	R						opcode									

8-bit variant

Applies when `opcode == 001`.

```
ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>]
```

16-bit variant

Applies when `opcode == 011 && size == x0`.

```
ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>]
```

32-bit variant

Applies when `opcode == 101 && size == 00`.

```
ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>]
```

64-bit variant

Applies when `opcode == 101 && S == 0 && size == 01`.

```
ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>]
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	1	1	0	0	Rm		x	x	1	S	size			Rn		Rt
									L	R			opcode									

8-bit, immediate offset variant

Applies when `Rm == 11111 && opcode == 001`.

```
ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], #3
```

8-bit, register offset variant

Applies when `Rm != 11111 && opcode == 001`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], <Xm>
```

16-bit, immediate offset variant

Applies when `Rm == 11111 && opcode == 011 && size == x0`.

```
ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], #6
```

16-bit, register offset variant

Applies when `Rm != 11111 && opcode == 011 && size == x0`.

```
ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], <Xm>
```

32-bit, immediate offset variant

Applies when `Rm == 11111 && opcode == 101 && size == 00`.

```
ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], #12
```

32-bit, register offset variant

Applies when `Rm != 11111 && opcode == 101 && size == 00`.

```
ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], <Xm>
```

64-bit, immediate offset variant

Applies when `Rm == 11111 && opcode == 101 && S == 0 && size == 01`.

```
ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], #24
```

64-bit, register offset variant

Applies when `Rm != 11111 && opcode == 101 && S == 0 && size == 01`.

```
ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler symbols

`<Vt>`	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
`<Vt2>`	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
`<Vt3>`	Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
`<index>`	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Shared decode for all encodings

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation for all encodings

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.327 ST4 (multiple structures)

Store multiple 4-element structures from four registers. This instruction stores multiple 4-element structures to memory from four SIMD&FP registers, with interleaving. Every element of each register is stored.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	size		Rn		Rt	
									L							opcode									

No offset variant

```
ST4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

Decode for this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	0	1	0	0	Rm		0	0	0	0	size		Rn		Rt	
									L				opcode									

Immediate offset variant

Applies when `Rm == 11111`.

```
ST4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

Register offset variant

Applies when `Rm != 11111`.

```
ST4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler symbols

`<Vt>` Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `8B` when `size = 00`, `Q = 0`
- `16B` when `size = 00`, `Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1
2D	when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

#32	when Q = 0
#64	when Q = 1

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared decode for all encodings

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;    // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
    otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.328 ST4 (single structure)

Store single 4-element structure from one lane of four registers. This instruction stores a 4-element structure to memory from corresponding elements of four SIMD&FP registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	1	0	0	1	0	0	0	0	0	x	x	1	S	size		Rn		Rt	
									L	R						opcode									

8-bit variant

Applies when `opcode == 001`.

```
ST4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>]
```

16-bit variant

Applies when `opcode == 011 && size == x0`.

```
ST4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>]
```

32-bit variant

Applies when `opcode == 101 && size == 00`.

```
ST4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>]
```

64-bit variant

Applies when `opcode == 101 && S == 0 && size == 01`.

```
ST4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>]
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	1	1	0	1	Rm		x	x	1	S	size		Rn		Rt	
									L	R			opcode									

8-bit, immediate offset variant

Applies when `Rm == 11111 && opcode == 001`.

```
ST4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], #4
```

8-bit, register offset variant

Applies when `Rm != 11111 && opcode == 001`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
ST4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], <Xm>
```

16-bit, immediate offset variant

Applies when `Rm == 11111 && opcode == 011 && size == x0`.

```
ST4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], #8
```

16-bit, register offset variant

Applies when `Rm != 11111 && opcode == 011 && size == x0`.

```
ST4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], <Xm>
```

32-bit, immediate offset variant

Applies when `Rm == 11111 && opcode == 101 && size == 00`.

```
ST4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], #16
```

32-bit, register offset variant

Applies when `Rm != 11111 && opcode == 101 && size == 00`.

```
ST4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], <Xm>
```

64-bit, immediate offset variant

Applies when `Rm == 11111 && opcode == 101 && S == 0 && size == 01`.

```
ST4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], #32
```

64-bit, register offset variant

Applies when `Rm != 11111 && opcode == 101 && S == 0 && size == 01`.

```
ST4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler symbols

`<Vt>`	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
`<Vt2>`	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
`<Vt3>`	Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
`<Vt4>`	Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
`<index>`	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Shared decode for all encodings

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation for all encodings

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.329 STNP (SIMD&FP)

Store Pair of SIMD&FP registers, with Non-temporal hint. This instruction stores a pair of SIMD&FP registers to memory, issuing a hint to the memory system that the access is non-temporal. The address used for the store is calculated from an address from a base register value and an immediate offset. For information about non-temporal pair instructions, see *Load/Store SIMD and Floating-point Non-temporal pair* on page C3-218.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30	29 28 27	26	25 24 23	22	21 15	14 10	9 5	4 0
opc	1 0 1	1	0 0 0	0	imm7	Rt2	Rn	Rt
				L				

32-bit variant

Applies when `opc == 00`.

```
STNP <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

64-bit variant

Applies when `opc == 01`.

```
STNP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

128-bit variant

Applies when `opc == 10`.

```
STNP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

Decode for all variants of this encoding

```
// Empty.
```

Assembler symbols

`<Dt1>`	Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
`<Dt2>`	Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
`<Qt1>`	Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
`<Qt2>`	Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
`<St1>`	Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
`<St2>`	Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<imm>`	For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4. For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8. For the 128-bit variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = n != 31;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data1 = V[t];
data2 = V[t2];
Mem[address, dbytes, AccType_VECSTREAM] = data1;
Mem[address+dbytes, dbytes, AccType_VECSTREAM] = data2;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.330 STP (SIMD&FP)

Store Pair of SIMD&FP registers. This instruction stores a pair of SIMD&FP registers to memory. The address used for the store is calculated from a base register value and an immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Post-index

31 30	29 28	27	26	25 24	23	22	21	15 14	10 9	5 4	0
opc	1 0	1	1	0 0	1	0	imm7	Rt2	Rn	Rt	
						L					

32-bit variant

Applies when `opc == 00`.

```
STP <St1>, <St2>, [<Xn|SP>], #<imm>
```

64-bit variant

Applies when `opc == 01`.

```
STP <Dt1>, <Dt2>, [<Xn|SP>], #<imm>
```

128-bit variant

Applies when `opc == 10`.

```
STP <Qt1>, <Qt2>, [<Xn|SP>], #<imm>
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
```

Pre-index

31 30	29 28	27	26	25 24	23	22	21	15 14	10 9	5 4	0
opc	1 0	1	1	0 1	1	0	imm7	Rt2	Rn	Rt	
						L					

32-bit variant

Applies when `opc == 00`.

```
STP <St1>, <St2>, [<Xn|SP>, #<imm>]!
```

64-bit variant

Applies when `opc == 01`.

```
STP <Dt1>, <Dt2>, [<Xn|SP>, #<imm>]!
```

128-bit variant

Applies when `opc == 10`.

```
STP <Qt1>, <Qt2>, [<Xn|SP>, #<imm>]!
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
```

Signed offset

31 30	29 28 27	26	25 24 23	22	21 … 15	14 … 10	9 … 5	4 … 0
opc	1 0 1	1	0 1 0	0 (L)	imm7	Rt2	Rn	Rt

32-bit variant

Applies when `opc == 00`.

```
STP <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

64-bit variant

Applies when `opc == 01`.

```
STP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

128-bit variant

Applies when `opc == 10`.

```
STP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

Decode for all variants of this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
```

Assembler symbols

`<Dt1>`	Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
`<Dt2>`	Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
`<Qt1>`	Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
`<Qt2>`	Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
`<St1>`	Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
`<St2>`	Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<imm>`	For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4. For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4. For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8. For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8. For the 128-bit post-index and 128-bit pre-index variant: is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the "imm7" field as <imm>/16.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the 128-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = wback || n != 31;
```

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

data1 = V[t];
data2 = V[t2];
Mem[address, dbytes, AccType_VEC] = data1;
Mem[address+dbytes, dbytes, AccType_VEC] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.331 STR (immediate, SIMD&FP)

Store SIMD&FP register (immediate offset). This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Post-index

31 30	29 28	27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
size	1 1	1	1	0 0	x 0 (opc)	0	imm9	0 1	Rn	Rt

8-bit variant

Applies when `size == 00 && opc == 00`.

```
STR <Bt>, [<Xn|SP>], #<simm>
```

16-bit variant

Applies when `size == 01 && opc == 00`.

```
STR <Ht>, [<Xn|SP>], #<simm>
```

32-bit variant

Applies when `size == 10 && opc == 00`.

```
STR <St>, [<Xn|SP>], #<simm>
```

64-bit variant

Applies when `size == 11 && opc == 00`.

```
STR <Dt>, [<Xn|SP>], #<simm>
```

128-bit variant

Applies when `size == 00 && opc == 10`.

```
STR <Qt>, [<Xn|SP>], #<simm>
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31 30	29 28	27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
size	1 1	1	1	0 0	x 0 (opc)	0	imm9	1 1	Rn	Rt

8-bit variant

Applies when `size == 00 && opc == 00`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
STR <Bt>, [<Xn|SP>, #<simm>]!
```

16-bit variant

Applies when `size == 01 && opc == 00`.

```
STR <Ht>, [<Xn|SP>, #<simm>]!
```

32-bit variant

Applies when `size == 10 && opc == 00`.

```
STR <St>, [<Xn|SP>, #<simm>]!
```

64-bit variant

Applies when `size == 11 && opc == 00`.

```
STR <Dt>, [<Xn|SP>, #<simm>]!
```

128-bit variant

Applies when `size == 00 && opc == 10`.

```
STR <Qt>, [<Xn|SP>, #<simm>]!
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31 30	29 28	27	26	25 24	23 22	21 ... 10	9 ... 5	4 ... 0
size	1 1	1	1	0 1	x 0 (opc)	imm12	Rn	Rt

8-bit variant

Applies when `size == 00 && opc == 00`.

```
STR <Bt>, [<Xn|SP>{, #<pimm>}]
```

16-bit variant

Applies when `size == 01 && opc == 00`.

```
STR <Ht>, [<Xn|SP>{, #<pimm>}]
```

32-bit variant

Applies when `size == 10 && opc == 00`.

```
STR <St>, [<Xn|SP>{, #<pimm>}]
```

64-bit variant

Applies when `size == 11 && opc == 00`.

```
STR <Dt>, [<Xn|SP>{, #<pimm>}]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

128-bit variant

Applies when `size == 00 && opc == 10`.

```
STR <Qt>, [<Xn|SP>{, #<pimm>}]
```

Decode for all variants of this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

Assembler symbols

`<Bt>`	Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
`<Dt>`	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
`<Ht>`	Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
`<Qt>`	Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
`<St>`	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
`<pimm>`	For the 8-bit variant: is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field. For the 16-bit variant: is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2. For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4. For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8. For the 128-bit variant: is the optional positive immediate byte offset, a multiple of 16 in the range 0 to 65520, defaulting to 0 and encoded in the "imm12" field as <pimm>/16.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);
```

Operation for all encodings

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if !postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, AccType_VEC] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, AccType_VEC];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.332 STR (register, SIMD&FP)

Store SIMD&FP register (register offset). This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an offset register value. The offset can be optionally shifted and extended.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30	29 28	27	26	25 24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
size	1 1	1	1	0 0	x 0 (opc)	1	Rm	option	S	1 0	Rn	Rt

8-fsreg,STR-8-fsreg variant

Applies when `size == 00 && opc == 00 && option != 011`.

```
STR <Bt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

8-fsreg,STR-8-fsreg variant

Applies when `size == 00 && opc == 00 && option == 011`.

```
STR <Bt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

16-fsreg,STR-16-fsreg variant

Applies when `size == 01 && opc == 00`.

```
STR <Ht>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

32-fsreg,STR-32-fsreg variant

Applies when `size == 10 && opc == 00`.

```
STR <St>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

64-fsreg,STR-64-fsreg variant

Applies when `size == 11 && opc == 00`.

```
STR <Dt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

128-fsreg,STR-128-fsreg variant

Applies when `size == 00 && opc == 10`.

```
STR <Qt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for all variants of this encoding

```
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler symbols

`<Bt>` Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

`<Dt>` Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Ht>` Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

`<Qt>` Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

`<St>` Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<Wm>` When `option<0>` is set to `0`, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

`<Xm>` When `option<0>` is set to `1`, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

`<extend>` For the 8-bit variant: is the index extend specifier, encoded in the "option" field. It can have the following values:

`UXTW` when `option = 010`

`SXTW` when `option = 110`

`SXTX` when `option = 111`

For the 128-bit, 16-bit, 32-bit and 64-bit variant: is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in the "option" field. It can have the following values:

`UXTW` when `option = 010`

`LSL` when `option = 011`

`SXTW` when `option = 110`

`SXTX` when `option = 111`

`<amount>` For the 8-bit variant: is the index shift amount, it must be `#0`, encoded in "S" as `0` if omitted, or as `1` if present.

For the 16-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

`#0` when `S = 0`

`#1` when `S = 1`

For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

`#0` when `S = 0`

`#2` when `S = 1`

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

`#0` when `S = 0`

`#3` when `S = 1`

For the 128-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

`#0` when `S = 0`

`#4` when `S = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, AccType_VEC] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, AccType_VEC];
        V[t] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.333 STUR (SIMD&FP)

Store SIMD&FP register (unscaled offset). This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an optional immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30	29 28	27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
size	1 1	1	1	0 0	x 0	0	imm9	0 0	Rn	Rt
					opc					

8-bit variant

Applies when `size == 00 && opc == 00`.

```
STUR <Bt>, [<Xn|SP>{, #<simm>}]
```

16-bit variant

Applies when `size == 01 && opc == 00`.

```
STUR <Ht>, [<Xn|SP>{, #<simm>}]
```

32-bit variant

Applies when `size == 10 && opc == 00`.

```
STUR <St>, [<Xn|SP>{, #<simm>}]
```

64-bit variant

Applies when `size == 11 && opc == 00`.

```
STUR <Dt>, [<Xn|SP>{, #<simm>}]
```

128-bit variant

Applies when `size == 00 && opc == 10`.

```
STUR <Qt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Bt>`	Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
`<Dt>`	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
`<Ht>`	Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
`<Qt>`	Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
`<St>`	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<simm>` Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);
```

Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, AccType_VEC] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, AccType_VEC];
        V[t] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.334 SUB (vector)

Subtract (vector). This instruction subtracts each vector element in the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23	22	21	20–16	15	14	13	12	11	10	9–5	4–0
0	1	1	1	1	1	1	0	size		1	Rm	1	0	0	0	0	1	Rn	Rd
		U																	

Scalar variant

```
SUB <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (U == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20–16	15	14	13	12	11	10	9–5	4–0
0	Q	1	0	1	1	1	0	size		1	Rm	1	0	0	0	0	1	Rn	Rd
		U																	

Vector variant

```
SUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (U == '1');
```

Assembler symbols

`<V>` Is a width specifier, encoded in the "size" field. It can have the following values:

`D` when `size = 11`

The following encodings are reserved:

- `size = 0x`.
- `size = 10`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<m>` Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B`	when `size = 00`, `Q = 0`
`16B`	when `size = 00`, `Q = 1`
`4H`	when `size = 01`, `Q = 0`
`8H`	when `size = 01`, `Q = 1`
`2S`	when `size = 10`, `Q = 0`
`4S`	when `size = 10`, `Q = 1`
`2D`	when `size = 11`, `Q = 1`

The encoding `size = 11`, `Q = 0` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then
        Elem[result, e, esize] = element1 - element2;
    else
        Elem[result, e, esize] = element1 + element2;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.335 SUBHN, SUBHN2

Subtract returning High Narrow. This instruction subtracts each vector element in the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are signed integer values.

The results are truncated. For rounded results, see RSUBHN, RSUBHN2.

The `SUBHN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SUBHN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	size	1	Rm	0	1	1	0	0	0	Rn	Rd
		U											o1					

Three registers, not all the same type variant

```
SUBHN{2} <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- `[absent]` when `Q = 0`
- `[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Tb>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `8B` when `size = 00, Q = 0`
- `16B` when `size = 00, Q = 1`
- `4H` when `size = 01, Q = 0`
- `8H` when `size = 01, Q = 1`
- `2S` when `size = 10, Q = 0`
- `4S` when `size = 10, Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The encoding `size = 11`, `Q = x` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

`8H` when `size = 00`

`4S` when `size = 01`

`2D` when `size = 10`

The encoding `size = 11` is reserved.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(2*datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
bits(2*esize) element1;
bits(2*esize) element2;
bits(2*esize) sum;

for e = 0 to elements-1
    element1 = Elem[operand1, e, 2*esize];
    element2 = Elem[operand2, e, 2*esize];
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    sum = sum + round_const;
    Elem[result, e, esize] = sum<2*esize-1:esize>;

Vpart[d, part] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.336 SUDOT (by element)

Dot product index form with signed and unsigned integers. This instruction performs the dot product of the four signed 8-bit integer values in each 32-bit element of the first source register with the four unsigned 8-bit integer values in an indexed 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination vector.

From Armv8.2 to Armv8.5, this is an OPTIONAL instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. ID_AA64ISAR1_EL1.I8MM indicates whether this instruction is supported.

ARMv8.6

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11	10	9–5	4–0
0	Q	0	0	1	1	1	1	0	0	L	M	Rm	1	1	1	1	H	0	Rn	Rd
								US												

Vector variant

```
SUDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.4B[<index>]
```

Decode for this encoding

```
if !HaveInt8MatMulExt() then UNDEFINED;
boolean op1_unsigned = (US == '1');
boolean op2_unsigned = (US == '0');
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer d = UInt(Rd);
integer i = UInt(H:L);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `2S` when `Q = 0`
- `4S` when `Q = 1`

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `8B` when `Q = 0`
- `16B` when `Q = 1`

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

`<index>` Is the immediate index of a quadtuplet of four 8-bit elements in the range 0 to 3, encoded in the "H:L" fields.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(128) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
for e = 0 to elements-1
    bits(32) res = Elem[operand3, e, 32];
    for b = 0 to 3
        integer element1 = Int(Elem[operand1, 4*e+b, 8], op1_unsigned);
        integer element2 = Int(Elem[operand2, 4*i+b, 8], op2_unsigned);
        res = res + element1 * element2;
    Elem[result, e, 32] = res;
V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.337 SUQADD

Signed saturating Accumulate of Unsigned value. This instruction adds the unsigned integer values of the vector elements in the source SIMD&FP register to corresponding signed integer values of the vector elements in the destination SIMD&FP register, and writes the resulting signed integer values to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31 30	29	28 27 26 25 24	23 22	21 20 19 18 17 16	15 14 13 12	11 10	9 5	4 0
0 1	0	1 1 1 1 0	size	1 0 0 0 0	0 0 0 1 1	1 0	Rn	Rd
	U							

Scalar variant

```
SUQADD <V><d>, <V><n>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean unsigned = (U == '1');
```

Vector

31 30	29	28 27 26 25 24	23 22	21 20 19 18 17 16	15 14 13 12	11 10	9 5	4 0
0 Q	0	0 1 1 1 0	size	1 0 0 0 0	0 0 0 1 1	1 0	Rn	Rd
	U							

Vector variant

```
SUQADD <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

B when size = 00

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

	H	when size = 01
	S	when size = 10
	D	when size = 11

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B	when size = 00, Q = 0
16B	when size = 00, Q = 1
4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1
2D	when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(datasize) operand2 = V[d];
integer op1;
integer op2;
boolean sat;

for e = 0 to elements-1
    op1 = Int(Elem[operand, e, esize], !unsigned);
    op2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], sat) = SatQ(op1 + op2, esize, unsigned);
    if sat then FPSR.QC = '1';
V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.338 SXTL, SXTL2

Signed extend Long. This instruction duplicates each vector element in the lower or upper half of the source SIMD&FP register into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The `SXTL` instruction extracts the source vector from the lower half of the source register, while the `SXTL2` instruction extracts the source vector from the upper half of the source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is an alias of the SSHLL, SSHLL2 instruction. This means that:

- The encodings in this description are named to match the encodings of SSHLL, SSHLL2.
- The description of SSHLL, SSHLL2 gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	19	18	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	1	0	!=0000	0	0	0	1	0	1	0	0	1	Rn		Rd	
		U							immh		immb											

Vector variant

```
SXTL{2} <Vd>.<Ta>, <Vn>.<Tb>
```

is equivalent to

```
SSHLL{2}  <Vd>.<Ta>, <Vn>.<Tb>, #0
```

and is the preferred disassembly when `BitCount(immh) == 1`.

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "immh" field. It can have the following values:

`8H` when `immh = 0001`

`4S` when `immh = 001x`

`2D` when `immh = 01xx`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`.

The encoding `immh = 1xxx` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

`8B` when `immh = 0001`, `Q = 0`

`16B` when `immh = 0001`, `Q = 1`

`4H` when `immh = 001x`, `Q = 0`

`8H` when `immh = 001x`, `Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`2S` when `immh = 01xx`, `Q = 0`

`4S` when `immh = 01xx`, `Q = 1`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`, `Q = x`.

The encoding `immh = 1xxx`, `Q = x` is reserved.

Operation

The description of SSHLL, SSHLL2 gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.339 TBL

Table vector Lookup. This instruction reads each value from the vector elements in the index source SIMD&FP register, uses each result as an index to perform a lookup in a table of bytes that is described by one to four source table SIMD&FP registers, places the lookup result in a vector, and writes the vector to the destination SIMD&FP register. If an index is out of range for the table, the result for that lookup is 0. If more than one source register is used to describe the table, the first source register describes the lowest bytes of the table.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20–16	15	14–13	12	11	10	9–5	4–0
0	Q	0	0	1	1	1	0	0	0	0	Rm	0	len	0	0	0	Rn	Rd
														op				

Two register table variant

Applies when `len == 01`.

```
TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B }, <Vm>.<Ta>
```

Three register table variant

Applies when `len == 10`.

```
TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B }, <Vm>.<Ta>
```

Four register table variant

Applies when `len == 11`.

```
TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B, <Vn+3>.16B }, <Vm>.<Ta>
```

Single register table variant

Applies when `len == 00`.

```
TBL <Vd>.<Ta>, { <Vn>.16B }, <Vm>.<Ta>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
integer regs = UInt(len) + 1;
boolean is_tbl = (op == '0');
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `8B` when `Q = 0`
- `16B` when `Q = 1`

`<Vn>` For the four register table, three register table and two register table variant: is the name of the first SIMD&FP table register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the single register table variant: is the name of the SIMD&FP table register, encoded in the "Rn" field.

<Vn+1> Is the name of the second SIMD&FP table register, encoded as "Rn" plus 1 modulo 32.

<Vn+2> Is the name of the third SIMD&FP table register, encoded as "Rn" plus 2 modulo 32.

<Vn+3> Is the name of the fourth SIMD&FP table register, encoded as "Rn" plus 3 modulo 32.

<Vm> Is the name of the SIMD&FP index register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) indices = V[m];
bits(128*regs) table = Zeros();
bits(datasize) result;
integer index;

// Create table from registers
for i = 0 to regs-1
    table<128*i+127:128*i> = V[n];
    n = (n + 1) MOD 32;

result = if is_tbl then Zeros() else V[d];
for i = 0 to elements-1
    index = UInt(Elem[indices, i, 8]);
    if index < 16 * regs then
        Elem[result, i, 8] = Elem[table, index, 8];

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.340 TBX

Table vector lookup extension. This instruction reads each value from the vector elements in the index source SIMD&FP register, uses each result as an index to perform a lookup in a table of bytes that is described by one to four source table SIMD&FP registers, places the lookup result in a vector, and writes the vector to the destination SIMD&FP register. If an index is out of range for the table, the existing value in the vector element of the destination register is left unchanged. If more than one source register is used to describe the table, the first source register describes the lowest bytes of the table.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14 13 12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	0	0	0	Rm	0	len	1	0	0	Rn	Rd
														op				

Two register table variant

Applies when `len == 01`.

```
TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B }, <Vm>.<Ta>
```

Three register table variant

Applies when `len == 10`.

```
TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B }, <Vm>.<Ta>
```

Four register table variant

Applies when `len == 11`.

```
TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B, <Vn+3>.16B }, <Vm>.<Ta>
```

Single register table variant

Applies when `len == 00`.

```
TBX <Vd>.<Ta>, { <Vn>.16B }, <Vm>.<Ta>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
integer regs = UInt(len) + 1;
boolean is_tbl = (op == '0');
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `8B` when `Q = 0`
- `16B` when `Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Vn>` For the four register table, three register table and two register table variant: is the name of the first SIMD&FP table register, encoded in the "Rn" field.

For the single register table variant: is the name of the SIMD&FP table register, encoded in the "Rn" field.

`<Vn+1>` Is the name of the second SIMD&FP table register, encoded as "Rn" plus 1 modulo 32.

`<Vn+2>` Is the name of the third SIMD&FP table register, encoded as "Rn" plus 2 modulo 32.

`<Vn+3>` Is the name of the fourth SIMD&FP table register, encoded as "Rn" plus 3 modulo 32.

`<Vm>` Is the name of the SIMD&FP index register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) indices = V[m];
bits(128*regs) table = Zeros();
bits(datasize) result;
integer index;

// Create table from registers
for i = 0 to regs-1
    table<128*i+127:128*i> = V[n];
    n = (n + 1) MOD 32;

result = if is_tbl then Zeros() else V[d];
for i = 0 to elements-1
    index = UInt(Elem[indices, i, 8]);
    if index < 16 * regs then
        Elem[result, i, 8] = Elem[table, index, 8];

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

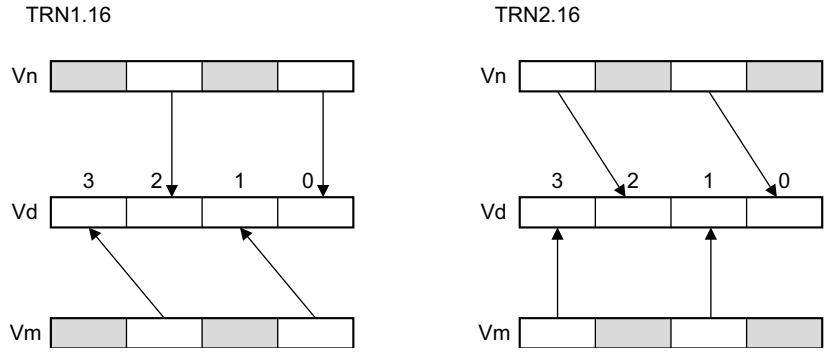
C7.2.341 TRN1

Transpose vectors (primary). This instruction reads corresponding even-numbered vector elements from the two source SIMD&FP registers, starting at zero, places each result into consecutive elements of a vector, and writes the vector to the destination SIMD&FP register. Vector elements from the first source register are placed into even-numbered elements of the destination vector, starting at zero, while vector elements from the second source register are placed into odd-numbered elements of the destination vector.

Note

By using this instruction with `TRN2`, a 2 x 2 matrix can be transposed.

The following figure shows the operation of TRN1 and TRN2 halfword operations where Q = 0.



Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	… 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	size		0		Rm	0	0	1	0	1	0	Rn	Rd
													op							

Advanced SIMD variant

```
TRN1 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
integer pairs = elements DIV 2;
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `8B` when `size = 00`, `Q = 0`
- `16B` when `size = 00`, `Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1
2D	when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, 2*p+part, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, 2*p+part, esize];

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

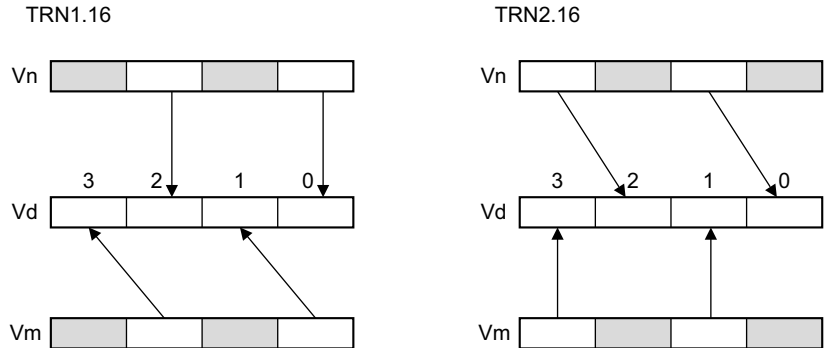
C7.2.342 TRN2

Transpose vectors (secondary). This instruction reads corresponding odd-numbered vector elements from the two source SIMD&FP registers, places each result into consecutive elements of a vector, and writes the vector to the destination SIMD&FP register. Vector elements from the first source register are placed into even-numbered elements of the destination vector, starting at zero, while vector elements from the second source register are placed into odd-numbered elements of the destination vector.

Note

By using this instruction with `TRN1`, a 2 x 2 matrix can be transposed.

The following figure shows the operation of TRN1 and TRN2 halfword operations where Q = 0.



Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	size		0	Rm		0	1	1	0	1	0	Rn		Rd	
													op									

Advanced SIMD variant

```
TRN2 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
integer pairs = elements DIV 2;
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `8B` when `size = 00`, `Q = 0`
- `16B` when `size = 00`, `Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1
2D	when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, 2*p+part, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, 2*p+part, esize];

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.343 UABA

Unsigned Absolute difference and Accumulate. This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the elements of the vector of the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size	1	Rm	0	1	1	1	1	1	Rn	Rd
		U													ac			

Three registers of the same type variant

```
UABA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B	when `size = 00`, `Q = 0`
16B	when `size = 00`, `Q = 1`
4H	when `size = 01`, `Q = 0`
8H	when `size = 01`, `Q = 1`
2S	when `size = 10`, `Q = 0`
4S	when `size = 10`, `Q = 1`

The encoding `size = 11`, `Q = x` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.344 UABAL, UABAL2

Unsigned Absolute difference and Accumulate Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The `UABAL` instruction extracts each source vector from the lower half of each source register, while the `UABAL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size		1	Rm	0	1	0	1	0	0	Rn	Rd
		U												op					

Three registers, not all the same type variant

```
UABAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

`8H` when `size = 00`

`4S` when `size = 01`

`2D` when `size = 10`

The encoding `size = 11` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B` when `size = 00`, `Q = 0`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

16B	when size = 00, Q = 1
4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.345 UABD

Unsigned Absolute Difference (vector). This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, places the absolute values of the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size	1	Rm	0	1	1	1	0	1	Rn	Rd
		U													ac			

Three registers of the same type variant

```
UABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B	when size = 00, Q = 0
16B	when size = 00, Q = 1
4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.346 UABDL, UABDL2

Unsigned Absolute Difference Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, places the absolute value of the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The `UABDL` instruction extracts each source vector from the lower half of each source register, while the `UABDL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size	1	Rm	0	1	1	1	0	0	Rn	Rd
		U											op					

Three registers, not all the same type variant

```
UABDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

`8H` when `size = 00`

`4S` when `size = 01`

`2D` when `size = 10`

The encoding `size = 11` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B` when `size = 00`, `Q = 0`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

16B	when size = 00, Q = 1
4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.347 UADALP

Unsigned Add and Accumulate Long Pairwise. This instruction adds pairs of adjacent unsigned integer values from the vector in the source SIMD&FP register and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size	1	0	0	0	0	0	0	1	1	0	1	0	Rn	Rd
		U														op						

Vector variant

```
UADALP <Vd>.<Ta>, <Vn>.<Tb>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2 * esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `4H` when `size = 00`, `Q = 0`
- `8H` when `size = 00`, `Q = 1`
- `2S` when `size = 01`, `Q = 0`
- `4S` when `size = 01`, `Q = 1`
- `1D` when `size = 10`, `Q = 0`
- `2D` when `size = 10`, `Q = 1`

The encoding `size = 11`, `Q = x` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `8B` when `size = 00`, `Q = 0`
- `16B` when `size = 00`, `Q = 1`
- `4H` when `size = 01`, `Q = 0`
- `8H` when `size = 01`, `Q = 1`
- `2S` when `size = 10`, `Q = 0`
- `4S` when `size = 10`, `Q = 1`

The encoding `size = 11`, `Q = x` is reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

if acc then result = V[d];
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1+op2)<2*esize-1:0>;
    if acc then
        Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;
    else
        Elem[result, e, 2*esize] = sum;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.348 UADDL, UADDL2

Unsigned Add Long (vector). This instruction adds each vector element in the lower or upper half of the first source SIMD&FP register to the corresponding vector element of the second source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The `UADDL` instruction extracts each source vector from the lower half of each source register, while the `UADDL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size	1	Rm	0	0	0	0	0	0	Rn	Rd
		U											o1					

Three registers, not all the same type variant

```
UADDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- `[absent]` when `Q = 0`
- `[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- `8H` when `size = 00`
- `4S` when `size = 01`
- `2D` when `size = 10`

The encoding `size = 11` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `8B` when `size = 00`, `Q = 0`
- `16B` when `size = 00`, `Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.349 UADDLP

Unsigned Add Long Pairwise. This instruction adds pairs of adjacent unsigned integer values from the vector in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size	1	0	0	0	0	0	0	0	1	0	1	0	Rn	Rd
		U														op						

Vector variant

```
UADDLP <Vd>.<Ta>, <Vn>.<Tb>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2 * esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `4H` when `size = 00`, `Q = 0`
- `8H` when `size = 00`, `Q = 1`
- `2S` when `size = 01`, `Q = 0`
- `4S` when `size = 01`, `Q = 1`
- `1D` when `size = 10`, `Q = 0`
- `2D` when `size = 10`, `Q = 1`

The encoding `size = 11`, `Q = x` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `8B` when `size = 00`, `Q = 0`
- `16B` when `size = 00`, `Q = 1`
- `4H` when `size = 01`, `Q = 0`
- `8H` when `size = 01`, `Q = 1`
- `2S` when `size = 10`, `Q = 0`
- `4S` when `size = 10`, `Q = 1`

The encoding `size = 11`, `Q = x` is reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

if acc then result = V[d];
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1+op2)<2*esize-1:0>;
    if acc then
        Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;
    else
        Elem[result, e, 2*esize] = sum;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.350 UADDLV

Unsigned sum Long across Vector. This instruction adds every vector element in the source SIMD&FP register together, and writes the scalar result to the destination SIMD&FP register. The destination scalar is twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	size		1	1	0	0	0	0	0	0	1	1	1	0	Rn		Rd	
		U																							

Advanced SIMD variant

```
UADDLV <V><d>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler symbols

<V> Is the destination width specifier, encoded in the "size" field. It can have the following values:

- H when size = 00
- S when size = 01
- D when size = 10

The encoding size = 11 is reserved.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 4S when size = 10, Q = 1

The following encodings are reserved:

- size = 10, Q = 0.
- size = 11, Q = x.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer sum;

sum = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    sum = sum + Int(Elem[operand, e, esize], unsigned);

V[d] = sum<2*esize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.351 UADDW, UADDW2

Unsigned Add Wide. This instruction adds the vector elements of the first source SIMD&FP register to the corresponding vector elements in the lower or upper half of the second source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. The vector elements of the destination register and the first source register are twice as long as the vector elements of the second source register. All the values in this instruction are unsigned integer values.

The `UADDW` instruction extracts vector elements from the lower half of the second source register, while the `UADDW2` instruction extracts vector elements from the upper half of the second source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size	1	Rm	0	0	0	1	0	0	Rn	Rd
		U											o1					

Three registers, not all the same type variant

```
UADDW{2} <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- `[absent]` when `Q = 0`
- `[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- `8H` when `size = 00`
- `4S` when `size = 01`
- `2D` when `size = 10`

The encoding `size = 11` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B	when size = 00, Q = 0
16B	when size = 00, Q = 1
4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, 2*esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.352 UCVTF (vector, fixed-point)

Unsigned fixed-point Convert to Floating-point (vector). This instruction converts each element in a vector from fixed-point to floating-point using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23 22	19 18 16	15	14	13	12	11	10	9 5	4 0
0	1	1	1	1	1	1	1	0	!=0000	immb	1	1	1	0	0	1	Rn	Rd
		U							immh								

Scalar variant

```
UCVTF <V><d>, <V><n>, #<fbits>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then UNDEFINED;
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR);
```

Vector

31	30	29	28	27	26	25	24	23 22	19 18 16	15	14	13	12	11	10	9 5	4 0
0	Q	1	0	1	1	1	1	0	!=0000	immb	1	1	1	0	0	1	Rn	Rd
		U							immh								

Vector variant

```
UCVTF <Vd>.<T>, <Vn>.<T>, #<fbits>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then UNDEFINED;
if immh<3>:Q == '10' then UNDEFINED;
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR);
```

Assembler symbols

<V> Is a width specifier, encoded in the "immh" field. It can have the following values:

H when immh = 001x

S when immh = 01xx

D when immh = 1xxx

The encoding immh = 000x is reserved.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

2D when immh = 1xxx, Q = 1

See *Advanced SIMD modified immediate* on page C4-354 when immh = 0000, Q = x.

The following encodings are reserved:

- immh = 0001, Q = x.
- immh = 1xxx, Q = 0.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits> For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in the "immh:immb" field. It can have the following values:

(32-Uint(immh:immb)) when immh = 001x

(64-UInt(immh:immb)) when immh = 01xx

(128-UInt(immh:immb)) when immh = 1xxx

The encoding immh = 000x is reserved.

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in the "immh:immb" field. It can have the following values:

(32-Uint(immh:immb)) when immh = 001x

(64-UInt(immh:immb)) when immh = 01xx

(128-UInt(immh:immb)) when immh = 1xxx

See *Advanced SIMD modified immediate* on page C4-354 when immh = 0000.

The encoding immh = 0001 is reserved.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, fracbits, unsigned, FPCR, rounding);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.353 UCVTF (vector, integer)

Unsigned integer Convert to Floating-point (vector). This instruction converts each element in a vector from an unsigned integer value to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Scalar half precision

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn		Rd	
		U																							

Scalar half precision variant

```
UCVTF <Hd>, <Hn>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	1	0	1	1	0	Rn		Rd	
		U																							

Scalar single-precision and double-precision variant

```
UCVTF <V><d>, <V><n>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Vector half precision

ARMv8.2

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn		Rd	
		U																							

Vector half precision variant

```
UCVTF <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	1	0	1	1	0	Rn		Rd	
		U																							

Vector single-precision and double-precision variant

```
UCVTF <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler symbols

`<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

`<V>` Is a width specifier, encoded in the "sz" field. It can have the following values:

- `S` when `sz = 0`
- `D` when `sz = 1`

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

`<n>` Is the number of the SIMD&FP source register, encoded in the "Rn" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` For the vector half precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `4H` when `Q = 0`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`8H` when `Q = 1`

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

`2S` when `sz = 0`, `Q = 0`

`4S` when `sz = 0`, `Q = 1`

`2D` when `sz = 1`, `Q = 1`

The encoding `sz = 1`, `Q = 0` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
FPRounding rounding = FPRoundingMode(FPCR);
bits(esize) element;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.354 UCVTF (scalar, fixed-point)

Unsigned fixed-point Convert to Floating-point (scalar). This instruction converts the unsigned value in the 32-bit or 64-bit general-purpose source register to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	16	15 10	9 5	4 0
sf	0	0	1	1	1	1	0	ftype	0	0	0	1	1	scale	Rn	Rd
										rmode		opcode				

32-bit to half-precision variant

Applies when `sf == 0 && ftype == 11`.

```
UCVTF <Hd>, <Wn>, #<fbits>
```

32-bit to single-precision variant

Applies when `sf == 0 && ftype == 00`.

```
UCVTF <Sd>, <Wn>, #<fbits>
```

32-bit to double-precision variant

Applies when `sf == 0 && ftype == 01`.

```
UCVTF <Dd>, <Wn>, #<fbits>
```

64-bit to half-precision variant

Applies when `sf == 1 && ftype == 11`.

```
UCVTF <Hd>, <Xn>, #<fbits>
```

64-bit to single-precision variant

Applies when `sf == 1 && ftype == 00`.

```
UCVTF <Sd>, <Xn>, #<fbits>
```

64-bit to double-precision variant

Applies when `sf == 1 && ftype == 01`.

```
UCVTF <Dd>, <Xn>, #<fbits>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
case ftype of
    when '00' fltsize = 32;
    when '01' fltsize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;

if sf == '0' && scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - UInt(scale);

rounding = FPRoundingMode(FPCR);
```

Assembler symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<fbits>	For the 32-bit to double-precision, 32-bit to half-precision and 32-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 32, encoded as 64 minus "scale". For the 64-bit to double-precision, 64-bit to half-precision and 64-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 64, encoded as 64 minus "scale".

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

intval = X[n];
fltval = FixedToFP(intval, fracbits, TRUE, FPCR, rounding);
V[d] = fltval;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.355 UCVTF (scalar, integer)

Unsigned integer Convert to Floating-point (scalar). This instruction converts the unsigned integer value in the general-purpose source register to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page D1-2360.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	16	15	14	13	12	11	10	9 5	4 0
sf	0	0	1	1	1	1	0	ftype	1	0	0	0	1	1	0	0	0	0	0	0	Rn	Rd
										rmode		opcode									

32-bit to half-precision variant

Applies when `sf == 0 && ftype == 11`.

```
UCVTF <Hd>, <Wn>
```

32-bit to single-precision variant

Applies when `sf == 0 && ftype == 00`.

```
UCVTF <Sd>, <Wn>
```

32-bit to double-precision variant

Applies when `sf == 0 && ftype == 01`.

```
UCVTF <Dd>, <Wn>
```

64-bit to half-precision variant

Applies when `sf == 1 && ftype == 11`.

```
UCVTF <Hd>, <Xn>
```

64-bit to single-precision variant

Applies when `sf == 1 && ftype == 00`.

```
UCVTF <Sd>, <Xn>
```

64-bit to double-precision variant

Applies when `sf == 1 && ftype == 01`.

```
UCVTF <Dd>, <Xn>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;

rounding = FPRoundingMode(FPCR);
```

Assembler symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

intval = X[n];
fltval = FixedToFP(intval, 0, TRUE, FPCR, rounding);
V[d] = fltval;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.356 UDOT (by element)

Dot Product unsigned arithmetic (vector, by element). This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_AA64ISAR0_EL1.DP indicates whether this instruction is supported.

ARMv8.2

31	30	29	28	27	26	25	24	23 22	21	20	19 16	15	14	13	12	11	10	9 5	4 0
0	Q	1	0	1	1	1	1	size	L	M	Rm	1	1	1	0	H	0	Rn	Rd
		U																	

Vector variant

```
UDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.4B[<index>]
```

Decode for this encoding

```
if !HaveDOTPExt() then UNDEFINED;
if size != '10' then UNDEFINED;
boolean signed = (U == '0');

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer index = UInt(H:L);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `2S` when `Q = 0`
- `4S` when `Q = 1`

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `8B` when `Q = 0`
- `16B` when `Q = 1`

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

`<index>` Is the element index, encoded in the "H:L" fields.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(128) operand2 = V[m];
bits(datasize) result = V[d];
for e = 0 to elements-1
    integer res = 0;
    integer element1, element2;
    for i = 0 to 3
        if signed then
            element1 = SInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4*index+i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4*index+i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.357 UDOT (vector)

Dot Product unsigned arithmetic (vector). This instruction performs the dot product of the four unsigned 8-bit elements in each 32-bit element of the first source register with the four unsigned 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_AA64ISAR0_EL1.DP indicates whether this instruction is supported.

ARMv8.2

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size	0	Rm	1	0	0	1	0	1	Rn	Rd
		U																

Vector variant

```
UDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !HaveDOTPExt() then UNDEFINED;
if size != '10' then UNDEFINED;
boolean signed = (U == '0');
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `2S` when `Q = 0`
- `4S` when `Q = 1`

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `8B` when `Q = 0`
- `16B` when `Q = 1`

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

result = V[d];
for e = 0 to elements-1
    integer res = 0;
    integer element1, element2;
    for i = 0 to 3
        if signed then
            element1 = SInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4*e+i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4*e+i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.358 UHADD

Unsigned Halving Add. This instruction adds corresponding unsigned integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see URHADD.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size	1	Rm	0	0	0	0	0	1	Rn	Rd
		U																

Three registers of the same type variant

```
UHADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B`	when `size = 00`, `Q = 0`
`16B`	when `size = 00`, `Q = 1`
`4H`	when `size = 01`, `Q = 0`
`8H`	when `size = 01`, `Q = 1`
`2S`	when `size = 10`, `Q = 0`
`4S`	when `size = 10`, `Q = 1`

The encoding `size = 11`, `Q = x` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer sum;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = element1 + element2;
    Elem[result, e, esize] = sum<esize:1>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.359 UHSUB

Unsigned Halving Subtract. This instruction subtracts the vector elements in the second source SIMD&FP register from the corresponding vector elements in the first source SIMD&FP register, shifts each result right one bit, places each result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size	1	Rm	0	0	1	0	0	1	Rn	Rd
		U																

Three registers of the same type variant

```
UHSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B`	when `size = 00`, `Q = 0`
`16B`	when `size = 00`, `Q = 1`
`4H`	when `size = 01`, `Q = 0`
`8H`	when `size = 01`, `Q = 1`
`2S`	when `size = 10`, `Q = 0`
`4S`	when `size = 10`, `Q = 1`

The encoding `size = 11`, `Q = x` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer diff;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    element2 = Int(Elem[operand2, e, esize], unsigned);
    diff = element1 - element2;
    Elem[result, e, esize] = diff<esize:1>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.360 UMAX

Unsigned Maximum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the larger of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size	1	Rm	0	1	1	0	0	1	Rn	Rd
		U													o1			

Three registers of the same type variant

```
UMAX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B	when size = 00, Q = 0
16B	when size = 00, Q = 1
4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.361 UMAXP

Unsigned Maximum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size	1	Rm	1	0	1	0	0	1	Rn	Rd
		U													o1			

Three registers of the same type variant

```
UMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B`	when `size = 00`, `Q = 0`
`16B`	when `size = 00`, `Q = 1`
`4H`	when `size = 01`, `Q = 0`
`8H`	when `size = 01`, `Q = 1`
`2S`	when `size = 10`, `Q = 0`
`4S`	when `size = 10`, `Q = 1`

The encoding `size = 11`, `Q = x` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[concat, 2*e, esize], unsigned);
    element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.362 UMAXV

Unsigned Maximum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size	1	1	0	0	0	0	1	0	1	0	1	0	Rn	Rd
		U												op								

Advanced SIMD variant

```
UMAXV <V><d>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean min = (op == '1');
```

Assembler symbols

<V> Is the destination width specifier, encoded in the "size" field. It can have the following values:

- B when `size = 00`
- H when `size = 01`
- S when `size = 10`

The encoding `size = 11` is reserved.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when `size = 00`, `Q = 0`
- 16B when `size = 00`, `Q = 1`
- 4H when `size = 01`, `Q = 0`
- 8H when `size = 01`, `Q = 1`
- 4S when `size = 10`, `Q = 1`

The following encodings are reserved:

- `size = 10`, `Q = 0`.
- `size = 11`, `Q = x`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer maxmin;
integer element;

maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);

V[d] = maxmin<esize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.363 UMIN

Unsigned Minimum (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, places the smaller of each of the two unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size	1	Rm	0	1	1	0	1	1	Rn	Rd
		U													o1			

Three registers of the same type variant

```
UMIN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B	when `size = 00`, `Q = 0`
16B	when `size = 00`, `Q = 1`
4H	when `size = 01`, `Q = 0`
8H	when `size = 01`, `Q = 1`
2S	when `size = 10`, `Q = 0`
4S	when `size = 10`, `Q = 1`

The encoding `size = 11`, `Q = x` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.364 UMINP

Unsigned Minimum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the smallest of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size	1	Rm	1	0	1	0	1	1	Rn	Rd
		U													o1			

Three registers of the same type variant

```
UMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B	when `size = 00`, `Q = 0`
16B	when `size = 00`, `Q = 1`
4H	when `size = 01`, `Q = 0`
8H	when `size = 01`, `Q = 1`
2S	when `size = 10`, `Q = 0`
4S	when `size = 10`, `Q = 1`

The encoding `size = 11`, `Q = x` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[concat, 2*e, esize], unsigned);
    element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.365 UMINV

Unsigned Minimum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size	1	1	0	0	0	1	1	0	1	0	1	0	Rn	Rd
		U												op								

Advanced SIMD variant

```
UMINV <V><d>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean min = (op == '1');
```

Assembler symbols

`<V>` Is the destination width specifier, encoded in the "size" field. It can have the following values:

- `B` when `size = 00`
- `H` when `size = 01`
- `S` when `size = 10`

The encoding `size = 11` is reserved.

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `8B` when `size = 00, Q = 0`
- `16B` when `size = 00, Q = 1`
- `4H` when `size = 01, Q = 0`
- `8H` when `size = 01, Q = 1`
- `4S` when `size = 10, Q = 1`

The following encodings are reserved:

- `size = 10, Q = 0`.
- `size = 11, Q = x`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer maxmin;
integer element;

maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);

V[d] = maxmin<esize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.366 UMLAL, UMLAL2 (by element)

Unsigned Multiply-Add Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `UMLAL` instruction extracts vector elements from the lower half of the first source register, while the `UMLAL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19 16	15	14	13	12	11	10	9 5	4 0
0	Q	1	0	1	1	1	1	size	L	M	Rm	0	0	1	0	H	0	Rn	Rd
		U											o2						

Vector variant

```
UMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

`4S` when `size = 01`

`2D` when `size = 10`

The following encodings are reserved:

- `size = 00`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- size = 11.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:

0:Rm when size = 01

M:Rm when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<index> Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

H:L:M when size = 01

H:L when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.367 UMLAL, UMLAL2 (vector)

Unsigned Multiply-Add Long (vector). This instruction multiplies the vector elements in the lower or upper half of the first source SIMD&FP register by the corresponding vector elements of the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `UMLAL` instruction extracts vector elements from the lower half of the first source register, while the `UMLAL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size	1	Rm	1	0	0	0	0	0	Rn	Rd
		U											o1					

Three registers, not all the same type variant

```
UMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- `[absent]` when `Q = 0`
- `[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- `8H` when `size = 00`
- `4S` when `size = 01`
- `2D` when `size = 10`

The encoding `size = 11` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `8B` when `size = 00`, `Q = 0`
- `16B` when `size = 00`, `Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.368 UMLSL, UMLSL2 (by element)

Unsigned Multiply-Subtract Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `UMLSL` instruction extracts vector elements from the lower half of the first source register, while the `UMLSL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19 16	15	14	13	12	11	10	9 5	4 0
0	Q	1	0	1	1	1	1	size	L	M	Rm	0	1	1	0	H	0	Rn	Rd
		U											o2						

Vector variant

```
UMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

`4S` when `size = 01`

`2D` when `size = 10`

The following encodings are reserved:

- `size = 00`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- size = 11.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:

0:Rm when size = 01

M:Rm when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<index> Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

H:L:M when size = 01

H:L when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.369 UMLSL, UMLSL2 (vector)

Unsigned Multiply-Subtract Long (vector). This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are unsigned integer values.

The `UMLSL` instruction extracts each source vector from the lower half of each source register, while the `UMLSL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size	1	Rm	1	0	1	0	0	0	Rn	Rd
		U											o1					

Three registers, not all the same type variant

```
UMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- `[absent]` when `Q = 0`
- `[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- `8H` when `size = 00`
- `4S` when `size = 01`
- `2D` when `size = 10`

The encoding `size = 11` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `8B` when `size = 00`, `Q = 0`
- `16B` when `size = 00`, `Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.370 UMMLA (vector)

Unsigned 8-bit integer matrix multiply-accumulate. This instruction multiplies the 2x8 matrix of unsigned 8-bit integer values in the first source vector by the 8x2 matrix of unsigned 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2 to Armv8.5, this is an OPTIONAL instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. ID_AA64ISAR1_EL1.I8MM indicates whether this instruction is supported.

ARMv8.6

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	1	0	1	1	1	0	1	0	0	Rm	1	0	1	0	0	1	Rn	Rd
		U														B			

Vector variant

```
UMMLA <Vd>.4S, <Vn>.16B, <Vm>.16B
```

Decode for this encoding

```
if !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
```

Assembler symbols

`<Vd>`	Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
`<Vn>`	Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
`<Vm>`	Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(128) operand1 = V[n];
bits(128) operand2 = V[m];
bits(128) addend = V[d];

V[d] = MatMulAdd(addend, operand1, operand2, TRUE, TRUE);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.371 UMOV

Unsigned Move vector element to general-purpose register. This instruction reads the unsigned integer from the source SIMD&FP register, zero-extends it to form a 32-bit or 64-bit value, and writes the result to the destination general-purpose register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (to general). See *Alias conditions* for details of when each alias is preferred.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	0	0	imm5		0	0	1	1	1	1	Rn		Rd	

32-bit variant

Applies when `Q == 0`.

```
UMOV <Wd>, <Vn>.<Ts>[<index>]
```

64-reg,UMOV-64-reg variant

Applies when `Q == 1 && imm5 == x1000`.

```
UMOV <Xd>, <Vn>.<Ts>[<index>]
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size;
case Q:imm5 of
    when '0xxxx1' size = 0;    // UMOV Wd, Vn.B
    when '0xxx10' size = 1;    // UMOV Wd, Vn.H
    when '0xx100' size = 2;    // UMOV Wd, Vn.S
    when '1x1000' size = 3;    // UMOV Xd, Vn.D
    otherwise UNDEFINED;

integer idxdsize = if imm5<4> == '1' then 128 else 64;
integer index = UInt(imm5<4:size+1>);
integer esize = 8 << size;
integer datasize = if Q == '1' then 64 else 32;
```

Alias conditions

Alias	is preferred when
MOV (to general)	`imm5 == 'x1000'`
MOV (to general)	`imm5 == 'xx100'`

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<Ts>` For the 32-bit variant: is an element size specifier, encoded in the "imm5" field. It can have the following values:

`B` when `imm5 = xxxx1`

`H` when `imm5 = xxx10`

`S` when `imm5 = xx100`

The encoding `imm5 = xx000` is reserved.

For the 64-reg,UMOV-64-reg variant: is an element size specifier, encoded in the "imm5" field. It can have the following values:

`D` when `imm5 = x1000`

The following encodings are reserved:

- `imm5 = x0000`.
- `imm5 = xxxx1`.
- `imm5 = xxx10`.
- `imm5 = xx100`.

`<index>` For the 32-bit variant: is the element index encoded in the "imm5" field. It can have the following values:

`imm5<4:1>` when `imm5 = xxxx1`

`imm5<4:2>` when `imm5 = xxx10`

`imm5<4:3>` when `imm5 = xx100`

The encoding `imm5 = xx000` is reserved.

For the 64-reg,UMOV-64-reg variant: is the element index encoded in "imm5<4>".

Operation

```
CheckFPAdvSIMDEnabled64();
bits(idxdsize) operand = V[n];

X[d] = ZeroExtend(Elem[operand, index, esize], datasize);
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.372 UMULL, UMULL2 (by element)

Unsigned Multiply Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `UMULL` instruction extracts vector elements from the lower half of the first source register, while the `UMULL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19 16	15	14	13	12	11	10	9 5	4 0
0	Q	1	0	1	1	1	1	size	L	M	Rm	1	0	1	0	H	0	Rn	Rd
		U																	

Vector variant

```
UMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

`4S` when `size = 01`

`2D` when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:

0:Rm when size = 01

M:Rm when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<index> Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

H:L:M when size = 01

H:L when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = product;

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.373 UMULL, UMULL2 (vector)

Unsigned Multiply long (vector). This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, places the result in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are unsigned integer values.

The `UMULL` instruction extracts each source vector from the lower half of each source register, while the `UMULL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size	1	Rm	1	1	0	0	0	0	Rn	Rd
		U																

Three registers, not all the same type variant

```
UMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- `[absent]` when `Q = 0`
- `[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- `8H` when `size = 00`
- `4S` when `size = 01`
- `2D` when `size = 10`

The encoding `size = 11` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `8B` when `size = 00`, `Q = 0`
- `16B` when `size = 00`, `Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    Elem[result, e, 2*esize] = (element1*element2)<2*esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.374 UQADD

Unsigned saturating Add. This instruction adds the values of corresponding elements of the two source SIMD&FP registers, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23 22	21	20 16	15	14	13	12	11	10	9 5	4 0
0	1	1	1	1	1	1	0	size	1	Rm	0	0	0	0	1	1	Rn	Rd
		U																

Scalar variant

```
UQADD <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Vector

31	30	29	28	27	26	25	24	23 22	21	20 16	15	14	13	12	11	10	9 5	4 0
0	Q	1	0	1	1	1	0	size	1	Rm	0	0	0	0	1	1	Rn	Rd
		U																

Vector variant

```
UQADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D when size = 11

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer sum;
boolean sat;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = element1 + element2;
    (Elem[result, e, esize], sat) = SatQ(sum, esize, unsigned);
    if sat then FPSR.QC = '1';

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.375 UQRSHL

Unsigned saturating Rounding Shift Left (register). This instruction takes each vector element of the first source SIMD&FP register, shifts the vector element by a value from the least significant byte of the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded. For truncated results, see UQSHL (immediate).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23 22	21	20 16	15	14	13	12	11	10	9 5	4 0
0	1	1	1	1	1	1	0	size	1	Rm	0	1	0	1	1	1	Rn	Rd
		U												R	S			

Scalar variant

```
UQRSHL <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;
```

Vector

31	30	29	28	27	26	25	24	23 22	21	20 16	15	14	13	12	11	10	9 5	4 0
0	Q	1	0	1	1	1	0	size	1	Rm	0	1	0	1	1	1	Rn	Rd
		U												R	S			

Vector variant

```
UQRSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

- B when `size = 00`
- H when `size = 01`
- S when `size = 10`
- D when `size = 11`

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when `size = 00`, `Q = 0`
- 16B when `size = 00`, `Q = 1`
- 4H when `size = 01`, `Q = 0`
- 8H when `size = 01`, `Q = 1`
- 2S when `size = 10`, `Q = 0`
- 4S when `size = 10`, `Q = 1`
- 2D when `size = 11`, `Q = 1`

The encoding `size = 11`, `Q = 0` is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1);    // 0 for left shift, 2^(n-1) for right shift
    element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.376 UQRSHRN, UQRSHRN2

Unsigned saturating Rounded Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see UQSHRN, UQSHRN2.

The `UQRSHRN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `UQRSHRN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31 30	29	28	27 26 25 24	23 22	19 18 16	15 14 13 12	11	10	9 5	4 0
0 1	1	1	1 1 1 1	1 0	!=0000	immb	1 0 0 1	1	1	Rn	Rd
	U				immh			op		

Scalar variant

```
UQRSHRN <Vb><d>, <Va><n>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

Vector

31 30	29	28	27 26 25 24	23 22	19 18 16	15 14 13 12	11	10	9 5	4 0
0 Q	1	0	1 1 1 1	1 0	!=0000	immb	1 0 0 1	1	1	Rn	Rd
	U				immh			op		

Vector variant

```
UQRSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Tb>` Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

`8B` when `immh = 0001`, `Q = 0`

`16B` when `immh = 0001`, `Q = 1`

`4H` when `immh = 001x`, `Q = 0`

`8H` when `immh = 001x`, `Q = 1`

`2S` when `immh = 01xx`, `Q = 0`

`4S` when `immh = 01xx`, `Q = 1`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`, `Q = x`.

The encoding `immh = 1xxx`, `Q = x` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<Ta>` Is an arrangement specifier, encoded in the "immh" field. It can have the following values:

`8H` when `immh = 0001`

`4S` when `immh = 001x`

`2D` when `immh = 01xx`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`.

The encoding `immh = 1xxx` is reserved.

`<Vb>` Is the destination width specifier, encoded in the "immh" field. It can have the following values:

`B` when `immh = 0001`

`H` when `immh = 001x`

`S` when `immh = 01xx`

The following encodings are reserved:

- `immh = 0000`.
- `immh = 1xxx`.

`<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.

`<Va>` Is the source width specifier, encoded in the "immh" field. It can have the following values:

`H` when `immh = 0001`

`S` when `immh = 001x`

`D` when `immh = 01xx`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following encodings are reserved:

- `immh = 0000`.
- `immh = 1xxx`.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<shift>` For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in the "immh:immb" field. It can have the following values:

`(16-UInt(immh:immb))` when `immh = 0001`

`(32-UInt(immh:immb))` when `immh = 001x`

`(64-UInt(immh:immb))` when `immh = 01xx`

The following encodings are reserved:

- `immh = 0000`.
- `immh = 1xxx`.

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in the "immh:immb" field. It can have the following values:

`(16-UInt(immh:immb))` when `immh = 0001`

`(32-UInt(immh:immb))` when `immh = 001x`

`(64-UInt(immh:immb))` when `immh = 01xx`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`.

The encoding `immh = 1xxx` is reserved.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;

for e = 0 to elements-1
    element = (Int(Elem[operand, e, 2*esize], unsigned) + round_const) >> shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';

Vpart[d, part] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.377 UQSHL (immediate)

Unsigned saturating Shift Left (immediate). This instruction takes each vector element in the source SIMD&FP register, shifts it by an immediate value, places the results in a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see UQRSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31 30 29 28	27 26 25 24	23 22	19 18 16	15 14 13 12	11	10	9 5	4 0
0 1 1	1 1 1 1 1 0	!=0000	immb	0 1 1 1	0	1	Rn	Rd
U		immh		op				

Scalar variant

```
UQSHL <V><d>, <V><n>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
    when '00' UNDEFINED;
    when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
    when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
    when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
```

Vector

31 30 29 28	27 26 25 24	23 22	19 18 16	15 14 13 12	11	10	9 5	4 0
0 Q 1	0 1 1 1 1 0	!=0000	immb	0 1 1 1	0	1	Rn	Rd
U		immh		op				

Vector variant

```
UQSHL <Vd>.<T>, <Vn>.<T>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
    when '00' UNDEFINED;
    when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
    when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
    when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
```

Assembler symbols

`<V>` Is a width specifier, encoded in the "immh" field. It can have the following values:

`B`	when `immh = 0001`
`H`	when `immh = 001x`
`S`	when `immh = 01xx`
`D`	when `immh = 1xxx`

The encoding `immh = 0000` is reserved.

`<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

`8B`	when `immh = 0001`, `Q = 0`
`16B`	when `immh = 0001`, `Q = 1`
`4H`	when `immh = 001x`, `Q = 0`
`8H`	when `immh = 001x`, `Q = 1`
`2S`	when `immh = 01xx`, `Q = 0`
`4S`	when `immh = 01xx`, `Q = 1`
`2D`	when `immh = 1xxx`, `Q = 1`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`, `Q = x`.

The encoding `immh = 1xxx`, `Q = 0` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<shift>` For the scalar variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in the "immh:immb" field. It can have the following values:

`(UInt(immh:immb)-8)`	when `immh = 0001`
`(UInt(immh:immb)-16)`	when `immh = 001x`
`(UInt(immh:immb)-32)`	when `immh = 01xx`
`(UInt(immh:immb)-64)`	when `immh = 1xxx`

The encoding `immh = 0000` is reserved.

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in the "immh:immb" field. It can have the following values:

`(UInt(immh:immb)-8)`	when `immh = 0001`
`(UInt(immh:immb)-16)`	when `immh = 001x`
`(UInt(immh:immb)-32)`	when `immh = 01xx`
`(UInt(immh:immb)-64)`	when `immh = 1xxx`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], src_unsigned) << shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, dst_unsigned);
    if sat then FPSR.QC = '1';

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.378 UQSHL (register)

Unsigned saturating Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts the element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated. For rounded results, see UQRSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23 22	21	20 16	15	14	13	12	11	10	9 5	4 0
0	1	1	1	1	1	1	0	size	1	Rm	0	1	0	0	1	1	Rn	Rd
		U												R	S			

Scalar variant

```
UQSHL <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;
```

Vector

31	30	29	28	27	26	25	24	23 22	21	20 16	15	14	13	12	11	10	9 5	4 0
0	Q	1	0	1	1	1	0	size	1	Rm	0	1	0	0	1	1	Rn	Rd
		U												R	S			

Vector variant

```
UQSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

B	when size = 00
H	when size = 01
S	when size = 10
D	when size = 11

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B	when size = 00, Q = 0
16B	when size = 00, Q = 1
4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1
2D	when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1);    // 0 for left shift, 2^(n-1) for right shift
    element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.379 UQSHRN, UQSHRN2

Unsigned saturating Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see UQRSHRN, UQRSHRN2.

The `UQSHRN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `UQSHRN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31 30	29	28	27 26 25 24	23 22	19 18 16	15 14 13 12	11	10 9	5 4	0
0 1	1	1	1 1 1 1 1 0	!=0000	immb	1 0 0 1	0	1	Rn	Rd
	U			immh			op			

Scalar variant

```
UQSHRN <Vb><d>, <Va><n>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

Vector

31 30	29	28	27 26 25 24	23 22	19 18 16	15 14 13 12	11	10 9	5 4	0
0 Q	1	0	1 1 1 1 0	!=0000	immb	1 0 0 1	0	1	Rn	Rd
	U			immh			op			

Vector variant

```
UQSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3> == '1' then UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Tb>` Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

`8B` when `immh = 0001`, `Q = 0`

`16B` when `immh = 0001`, `Q = 1`

`4H` when `immh = 001x`, `Q = 0`

`8H` when `immh = 001x`, `Q = 1`

`2S` when `immh = 01xx`, `Q = 0`

`4S` when `immh = 01xx`, `Q = 1`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`, `Q = x`.

The encoding `immh = 1xxx`, `Q = x` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<Ta>` Is an arrangement specifier, encoded in the "immh" field. It can have the following values:

`8H` when `immh = 0001`

`4S` when `immh = 001x`

`2D` when `immh = 01xx`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`.

The encoding `immh = 1xxx` is reserved.

`<Vb>` Is the destination width specifier, encoded in the "immh" field. It can have the following values:

`B` when `immh = 0001`

`H` when `immh = 001x`

`S` when `immh = 01xx`

The following encodings are reserved:

- `immh = 0000`.
- `immh = 1xxx`.

`<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.

`<Va>` Is the source width specifier, encoded in the "immh" field. It can have the following values:

`H` when `immh = 0001`

`S` when `immh = 001x`

`D` when `immh = 01xx`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following encodings are reserved:

- `immh = 0000`.
- `immh = 1xxx`.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<shift>` For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in the "immh:immb" field. It can have the following values:

`(16-UInt(immh:immb))` when `immh = 0001`

`(32-UInt(immh:immb))` when `immh = 001x`

`(64-UInt(immh:immb))` when `immh = 01xx`

The following encodings are reserved:

- `immh = 0000`.
- `immh = 1xxx`.

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in the "immh:immb" field. It can have the following values:

`(16-UInt(immh:immb))` when `immh = 0001`

`(32-UInt(immh:immb))` when `immh = 001x`

`(64-UInt(immh:immb))` when `immh = 01xx`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`.

The encoding `immh = 1xxx` is reserved.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;

for e = 0 to elements-1
    element = (Int(Elem[operand, e, 2*esize], unsigned) + round_const) >> shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';

Vpart[d, part] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.380 UQSUB

Unsigned saturating Subtract. This instruction subtracts the element values of the second source SIMD&FP register from the corresponding element values of the first source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23 22	21	20 16	15	14	13	12	11	10	9 5	4 0
0	1	1	1	1	1	1	0	size	1	Rm	0	0	1	0	1	1	Rn	Rd
		U																

Scalar variant

```
UQSUB <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Vector

31	30	29	28	27	26	25	24	23 22	21	20 16	15	14	13	12	11	10	9 5	4 0
0	Q	1	0	1	1	1	0	size	1	Rm	0	0	1	0	1	1	Rn	Rd
		U																

Vector variant

```
UQSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D when `size = 11`

`<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<m>` Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B	when `size = 00`, `Q = 0`
16B	when `size = 00`, `Q = 1`
4H	when `size = 01`, `Q = 0`
8H	when `size = 01`, `Q = 1`
2S	when `size = 10`, `Q = 0`
4S	when `size = 10`, `Q = 1`
2D	when `size = 11`, `Q = 1`

The encoding `size = 11`, `Q = 0` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer diff;
boolean sat;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    diff = element1 - element2;
    (Elem[result, e, esize], sat) = SatQ(diff, esize, unsigned);
    if sat then FPSR.QC = '1';

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.381 UQXTN, UQXTN2

Unsigned saturating extract Narrow. This instruction reads each vector element from the source SIMD&FP register, saturates each value to half the original width, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `UQXTN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `UQXTN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31 30	29	28 27 26 25 24	23 22	21 20 19 18 17 16	15 14 13 12	11 10	9 5	4 0
0 1	1	1 1 1 1 0	size	1 0 0 0 0	1 0 1 0 0	1 0	Rn	Rd
	U							

Scalar variant

```
UQXTN <Vb><d>, <Va><n>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer part = 0;
integer elements = 1;

boolean unsigned = (U == '1');
```

Vector

31 30	29	28 27 26 25 24	23 22	21 20 19 18 17 16	15 14 13 12	11 10	9 5	4 0
0 Q	1	0 1 1 1 0	size	1 0 0 0 0	1 0 1 0 0	1 0	Rn	Rd
	U							

Vector variant

```
UQXTN{2} <Vd>.<Tb>, <Vn>.<Ta>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Tb>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B` when `size = 00`, `Q = 0`

`16B` when `size = 00`, `Q = 1`

`4H` when `size = 01`, `Q = 0`

`8H` when `size = 01`, `Q = 1`

`2S` when `size = 10`, `Q = 0`

`4S` when `size = 10`, `Q = 1`

The encoding `size = 11`, `Q = x` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

`8H` when `size = 00`

`4S` when `size = 01`

`2D` when `size = 10`

The encoding `size = 11` is reserved.

`<Vb>` Is the destination width specifier, encoded in the "size" field. It can have the following values:

`B` when `size = 00`

`H` when `size = 01`

`S` when `size = 10`

The encoding `size = 11` is reserved.

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

`<Va>` Is the source width specifier, encoded in the "size" field. It can have the following values:

`H` when `size = 00`

`S` when `size = 01`

`D` when `size = 10`

The encoding `size = 11` is reserved.

`<n>` Is the number of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;
bits(2*esize) element;
boolean sat;

for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    (Elem[result, e, esize], sat) = SatQ(Int(element, unsigned), esize, unsigned);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        if sat then FPSR.QC = '1';

    Vpart[d, part] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.382 URECPE

Unsigned Reciprocal Estimate. This instruction reads each vector element from the source SIMD&FP register, calculates an approximate inverse for the unsigned integer value, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	1	0	0	1	0	Rn		Rd	

Vector variant

```
URECPE <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- `2S` when `sz = 0`, `Q = 0`
- `4S` when `sz = 0`, `Q = 1`

The encoding `sz = 1`, `Q = x` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(32) element;

for e = 0 to elements-1
    element = Elem[operand, e, 32];
    Elem[result, e, 32] = UnsignedRecipEstimate(element);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.383 URHADD

Unsigned Rounding Halving Add. This instruction adds corresponding unsigned integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see UHADD.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size	1	Rm	0	0	0	1	0	1	Rn	Rd
		U																

Three registers of the same type variant

```
URHADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B`	when `size = 00`, `Q = 0`
`16B`	when `size = 00`, `Q = 1`
`4H`	when `size = 01`, `Q = 0`
`8H`	when `size = 01`, `Q = 1`
`2S`	when `size = 10`, `Q = 0`
`4S`	when `size = 10`, `Q = 1`

The encoding `size = 11`, `Q = x` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    Elem[result, e, esize] = (element1+element2+1)<esize:1>;

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.384 URSHL

Unsigned Rounding Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts the vector element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31 30	29	28 27 26 25	24	23 22	21	20 16	15 14 13	12	11	10	9 5	4 0
0 1	1	1 1 1 1	0	size	1	Rm	0 1 0	1	0	1	Rn	Rd
	U							R	S			

Scalar variant

```
URSHL <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;
```

Vector

31 30	29	28 27 26 25	24	23 22	21	20 16	15 14 13	12	11	10	9 5	4 0
0 Q	1	0 1 1 1	0	size	1	Rm	0 1 0	1	0	1	Rn	Rd
	U							R	S			

Vector variant

```
URSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

D when size = 11

The following encodings are reserved:

- size = 0x.
- size = 10.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1);    // 0 for left shift, 2^(n-1) for right shift
    element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.385 URSHR

Unsigned Rounding Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see USHR.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23	22	19	18	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	1	0	!=0000		immb		0	0	1	0	0	1	Rn		Rd	
		U							immh						o1	o0						

Scalar variant

```
URSHR <V><d>, <V><n>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	19	18	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	1	0	!=0000		immb		0	0	1	0	0	1	Rn		Rd	
		U							immh						o1	o0						

Vector variant

```
URSHR <Vd>.<T>, <Vn>.<T>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

<V> Is a width specifier, encoded in the "immh" field. It can have the following values:

D when immh = 1xxx

The encoding immh = 0xxx is reserved.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B when immh = 0001, Q = 0

16B when immh = 0001, Q = 1

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

2D when immh = 1xxx, Q = 1

See *Advanced SIMD modified immediate* on page C4-354 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in the "immh:immb" field. It can have the following values:

(128-UInt(immh:immb)) when immh = 1xxx

The encoding immh = 0xxx is reserved.

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in the "immh:immb" field. It can have the following values:

(16-UInt(immh:immb)) when immh = 0001

(32-UInt(immh:immb)) when immh = 001x

(64-UInt(immh:immb)) when immh = 01xx

(128-UInt(immh:immb)) when immh = 1xxx

See *Advanced SIMD modified immediate* on page C4-354 when immh = 0000.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.386 URSQRTE

Unsigned Reciprocal Square Root Estimate. This instruction reads each vector element from the source SIMD&FP register, calculates an approximate inverse square root for each value, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	1	0	0	1	0	Rn		Rd	

Vector variant

```
URSQRTE <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- `2S` when `sz = 0`, `Q = 0`
- `4S` when `sz = 0`, `Q = 1`

The encoding `sz = 1`, `Q = x` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(32) element;

for e = 0 to elements-1
    element = Elem[operand, e, 32];
    Elem[result, e, 32] = UnsignedRSqrtEstimate(element);

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.387 URSRA

Unsigned Rounding Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see USRA.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31 30	29	28 27 26 25 24	23 22	19 18 16	15 14	13	12	11	10	9 5	4 0
0 1	1	1 1 1 1 1 0	!=0000	immb	0 0	1	1	0	1	Rn	Rd
	U		immh			o1	o0				

Scalar variant

```
URSRA <V><d>, <V><n>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector

31 30	29	28 27 26 25 24	23 22	19 18 16	15 14	13	12	11	10	9 5	4 0
0 Q	1	0 1 1 1 1 0	!=0000	immb	0 0	1	1	0	1	Rn	Rd
	U		immh			o1	o0				

Vector variant

```
URSRA <Vd>.<T>, <Vn>.<T>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

<V> Is a width specifier, encoded in the "immh" field. It can have the following values:

D when immh = 1xxx

The encoding immh = 0xxx is reserved.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B when immh = 0001, Q = 0

16B when immh = 0001, Q = 1

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

2D when immh = 1xxx, Q = 1

See *Advanced SIMD modified immediate* on page C4-354 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in the "immh:immb" field. It can have the following values:

(128-UInt(immh:immb)) when immh = 1xxx

The encoding immh = 0xxx is reserved.

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in the "immh:immb" field. It can have the following values:

(16-UInt(immh:immb)) when immh = 0001

(32-UInt(immh:immb)) when immh = 001x

(64-UInt(immh:immb)) when immh = 01xx

(128-UInt(immh:immb)) when immh = 1xxx

See *Advanced SIMD modified immediate* on page C4-354 when immh = 0000.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.388 USDOT (vector)

Dot Product vector form with unsigned and signed integers. This instruction performs the dot product of the four unsigned 8-bit integer values in each 32-bit element of the first source register with the four signed 8-bit integer values in the corresponding 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2 to Armv8.5, this is an OPTIONAL instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. ID_AA64ISAR1_EL1.I8MM indicates whether this instruction is supported.

ARMv8.6

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	1	0	0	Rm	1	0	0	1	1	1	Rn	Rd

Vector variant

```
USDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `2S` when `Q = 0`
- `4S` when `Q = 1`

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `8B` when `Q = 0`
- `16B` when `Q = 1`

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;

for e = 0 to elements-1
    bits(32) res = Elem[operand3, e, 32];
    for b = 0 to 3
        integer element1 = UInt(Elem[operand1, 4*e+b, 8]);
        integer element2 = SInt(Elem[operand2, 4*e+b, 8]);
        res = res + element1 * element2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        Elem[result, e, 32] = res;

    V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.389 USDOT (by element)

Dot Product index form with unsigned and signed integers. This instruction performs the dot product of the four unsigned 8-bit integer values in each 32-bit element of the first source register with the four signed 8-bit integer values in an indexed 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2 to Armv8.5, this is an OPTIONAL instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. ID_AA64ISAR1_EL1.I8MM indicates whether this instruction is supported.

ARMv8.6

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11	10	9–5	4–0
0	Q	0	0	1	1	1	1	1	0	L	M	Rm	1	1	1	1	H	0	Rn	Rd
								US												

Vector variant

```
USDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.4B[<index>]
```

Decode for this encoding

```
if !HaveInt8MatMulExt() then UNDEFINED;
boolean op1_unsigned = (US == '1');
boolean op2_unsigned = (US == '0');
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer d = UInt(Rd);
integer i = UInt(H:L);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `2S` when `Q = 0`
- `4S` when `Q = 1`

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `8B` when `Q = 0`
- `16B` when `Q = 1`

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

`<index>` Is the immediate index of a quadtuplet of four 8-bit elements in the range 0 to 3, encoded in the "H:L" fields.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(128) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
for e = 0 to elements-1
    bits(32) res = Elem[operand3, e, 32];
    for b = 0 to 3
        integer element1 = Int(Elem[operand1, 4*e+b, 8], op1_unsigned);
        integer element2 = Int(Elem[operand2, 4*i+b, 8], op2_unsigned);
        res = res + element1 * element2;
    Elem[result, e, 32] = res;
V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.390 USHL

Unsigned Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts each element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift. For a rounding shift, see URSHL.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31 30	29	28 27 26 25	24	23 22	21	20 16	15 14 13	12	11	10	9 5	4 0
0 1	1	1 1 1 1	0	size	1	Rm	0 1 0	0	0	1	Rn	Rd
	U							R	S			

Scalar variant

```
USHL <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;
```

Vector

31 30	29	28 27 26 25	24	23 22	21	20 16	15 14 13	12	11	10	9 5	4 0
0 Q	1	0 1 1 1	0	size	1	Rm	0 1 0	0	0	1	Rn	Rd
	U							R	S			

Vector variant

```
USHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

D when size = 11

The following encodings are reserved:

- size = 0x.
- size = 10.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1);    // 0 for left shift, 2^(n-1) for right shift
    element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.391 USHLL, USHLL2

Unsigned Shift Left Long (immediate). This instruction reads each vector element in the lower or upper half of the source SIMD&FP register, shifts the unsigned integer value left by the specified number of bits, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The `USHLL` instruction extracts vector elements from the lower half of the source register, while the `USHLL2` instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias UXTL, UXTL2. See *Alias conditions* for details of when each alias is preferred.

31	30	29	28	27	26	25	24	23	22	19	18	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	1	0	!=0000	immb			1	0	1	0	0	1	Rn		Rd	
		U							immh													

Vector variant

```
USHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;
boolean unsigned = (U == '1');
```

Alias conditions

Alias	is preferred when
UXTL, UXTL2	`immb == '000' && BitCount(immh) == 1`

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "immh" field. It can have the following values:

`8H` when `immh = 0001`

`4S` when `immh = 001x`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

2D when immh = 01xx

See *Advanced SIMD modified immediate* on page C4-354 when immh = 0000.

The encoding immh = 1xxx is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B when immh = 0001, Q = 0

16B when immh = 0001, Q = 1

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

See *Advanced SIMD modified immediate* on page C4-354 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = x is reserved.

<shift> Is the left shift amount, in the range 0 to the source element width in bits minus 1, encoded in the "immh:immb" field. It can have the following values:

(UInt(immh:immb)-8) when immh = 0001

(UInt(immh:immb)-16) when immh = 001x

(UInt(immh:immb)-32) when immh = 01xx

See *Advanced SIMD modified immediate* on page C4-354 when immh = 0000.

The encoding immh = 1xxx is reserved.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part];
bits(datasize*2) result;
integer element;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], unsigned) << shift;
    Elem[result, e, 2*esize] = element<2*esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.392 USHR

Unsigned Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see URSHR.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31 30	29	28 27 26 25 24	23 22	19 18 16	15 14	13	12	11	10	9 5	4 0
0 1	1	1 1 1 1 1 0	!=0000	immb	0 0	0	0	0	1	Rn	Rd
	U		immh			o1	o0				

Scalar variant

```
USHR <V><d>, <V><n>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector

31 30	29	28 27 26 25 24	23 22	19 18 16	15 14	13	12	11	10	9 5	4 0
0 Q	1	0 1 1 1 1 0	!=0000	immb	0 0	0	0	0	1	Rn	Rd
	U		immh			o1	o0				

Vector variant

```
USHR <Vd>.<T>, <Vn>.<T>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

<V> Is a width specifier, encoded in the "immh" field. It can have the following values:

D when immh = 1xxx

The encoding immh = 0xxx is reserved.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B when immh = 0001, Q = 0

16B when immh = 0001, Q = 1

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

2D when immh = 1xxx, Q = 1

See *Advanced SIMD modified immediate* on page C4-354 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in the "immh:immb" field. It can have the following values:

(128-UInt(immh:immb)) when immh = 1xxx

The encoding immh = 0xxx is reserved.

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in the "immh:immb" field. It can have the following values:

(16-UInt(immh:immb)) when immh = 0001

(32-UInt(immh:immb)) when immh = 001x

(64-UInt(immh:immb)) when immh = 01xx

(128-UInt(immh:immb)) when immh = 1xxx

See *Advanced SIMD modified immediate* on page C4-354 when immh = 0000.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.393 USMMLA (vector)

Unsigned and signed 8-bit integer matrix multiply-accumulate. This instruction multiplies the 2x8 matrix of unsigned 8-bit integer values in the first source vector by the 8x2 matrix of signed 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2 to Armv8.5, this is an OPTIONAL instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. ID_AA64ISAR1_EL1.I8MM indicates whether this instruction is supported.

ARMv8.6

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	0	0	1	1	1	0	1	0	0	Rm	1	0	1	0	1	1	Rn	Rd
		U														B			

Vector variant

```
USMMLA <Vd>.4S, <Vn>.16B, <Vm>.16B
```

Decode for this encoding

```
if !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
```

Assembler symbols

`<Vd>`	Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
`<Vn>`	Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
`<Vm>`	Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(128) operand1 = V[n];
bits(128) operand2 = V[m];
bits(128) addend = V[d];

V[d] = MatMulAdd(addend, operand1, operand2, TRUE, FALSE);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.394 USQADD

Unsigned saturating Accumulate of Signed value. This instruction adds the signed integer values of the vector elements in the source SIMD&FP register to corresponding unsigned integer values of the vector elements in the destination SIMD&FP register, and accumulates the resulting unsigned integer values with the vector elements of the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23 22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	1	1	1	1	1	0	size	1	0	0	0	0	0	0	0	1	1	1	0	Rn	Rd
		U																				

Scalar variant

```
USQADD <V><d>, <V><n>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean unsigned = (U == '1');
```

Vector

31	30	29	28	27	26	25	24	23 22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size	1	0	0	0	0	0	0	0	1	1	1	0	Rn	Rd
		U																				

Vector variant

```
USQADD <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

B when size = 00

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H	when size = 01
S	when size = 10
D	when size = 11

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B	when size = 00, Q = 0
16B	when size = 00, Q = 1
4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1
2D	when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(datasize) operand2 = V[d];
integer op1;
integer op2;
boolean sat;

for e = 0 to elements-1
    op1 = Int(Elem[operand, e, esize], !unsigned);
    op2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], sat) = SatQ(op1 + op2, esize, unsigned);
    if sat then FPSR.QC = '1';
V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.395 USRA

Unsigned Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see URSRA.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

31	30	29	28	27	26	25	24	23	22	19	18	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	1	0	!=0000		immb		0	0	0	1	0	1	Rn		Rd	
		U							immh						o1	o0						

Scalar variant

```
USRA <V><d>, <V><n>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	19	18	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	1	0	!=0000		immb		0	0	0	1	0	1	Rn		Rd	
		U							immh						o1	o0						

Vector variant

```
USRA <Vd>.<T>, <Vn>.<T>, #<shift>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

<V> Is a width specifier, encoded in the "immh" field. It can have the following values:

D when immh = 1xxx

The encoding immh = 0xxx is reserved.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B when immh = 0001, Q = 0

16B when immh = 0001, Q = 1

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

2D when immh = 1xxx, Q = 1

See *Advanced SIMD modified immediate* on page C4-354 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in the "immh:immb" field. It can have the following values:

(128-UInt(immh:immb)) when immh = 1xxx

The encoding immh = 0xxx is reserved.

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in the "immh:immb" field. It can have the following values:

(16-UInt(immh:immb)) when immh = 0001

(32-UInt(immh:immb)) when immh = 001x

(64-UInt(immh:immb)) when immh = 01xx

(128-UInt(immh:immb)) when immh = 1xxx

See *Advanced SIMD modified immediate* on page C4-354 when immh = 0000.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.396 USUBL, USUBL2

Unsigned Subtract Long. This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The destination vector elements are twice as long as the source vector elements.

The `USUBL` instruction extracts each source vector from the lower half of each source register, while the `USUBL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size	1	Rm	0	0	1	0	0	0	Rn	Rd
		U											o1					

Three registers, not all the same type variant

```
USUBL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- `[absent]` when `Q = 0`
- `[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- `8H` when `size = 00`
- `4S` when `size = 01`
- `2D` when `size = 10`

The encoding `size = 11` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `8B` when `size = 00`, `Q = 0`
- `16B` when `size = 00`, `Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.397 USUBW, USUBW2

Unsigned Subtract Wide. This instruction subtracts each vector element of the second source SIMD&FP register from the corresponding vector element in the lower or upper half of the first source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. All the values in this instruction are signed integer values.

The vector elements of the destination register and the first source register are twice as long as the vector elements of the second source register.

The `USUBW` instruction extracts vector elements from the lower half of the first source register, while the `USUBW2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size	1	Rm	0	0	1	1	0	0	Rn	Rd
		U											o1					

Three registers, not all the same type variant

```
USUBW{2} <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- `[absent]` when `Q = 0`
- `[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- `8H` when `size = 00`
- `4S` when `size = 01`
- `2D` when `size = 10`

The encoding `size = 11` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B	when size = 00, Q = 0
16B	when size = 00, Q = 1
4H	when size = 01, Q = 0
8H	when size = 01, Q = 1
2S	when size = 10, Q = 0
4S	when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, 2*esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.398 UXTL, UXTL2

Unsigned extend Long. This instruction copies each vector element from the lower or upper half of the source SIMD&FP register into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The `UXTL` instruction extracts vector elements from the lower half of the source register, while the `UXTL2` instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is an alias of the USHLL, USHLL2 instruction. This means that:

- The encodings in this description are named to match the encodings of USHLL, USHLL2.
- The description of USHLL, USHLL2 gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23 22	19 18 16	15	14	13	12	11	10	9 5	4 0
0	Q	1	0	1	1	1	1	!=0000	0 0 0	1	0	1	0	0	1	Rn	Rd
		U						immh	immb								

Vector variant

```
UXTL{2} <Vd>.<Ta>, <Vn>.<Tb>
```

is equivalent to

```
USHLL{2}  <Vd>.<Ta>, <Vn>.<Tb>, #0
```

and is the preferred disassembly when `BitCount(immh) == 1`.

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "immh" field. It can have the following values:

`8H` when `immh = 0001`

`4S` when `immh = 001x`

`2D` when `immh = 01xx`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`.

The encoding `immh = 1xxx` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

`8B` when `immh = 0001`, `Q = 0`

`16B` when `immh = 0001`, `Q = 1`

`4H` when `immh = 001x`, `Q = 0`

`8H` when `immh = 001x`, `Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`2S` when `immh = 01xx`, `Q = 0`

`4S` when `immh = 01xx`, `Q = 1`

See *Advanced SIMD modified immediate* on page C4-354 when `immh = 0000`, `Q = x`.

The encoding `immh = 1xxx`, `Q = x` is reserved.

Operation

The description of USHLL, USHLL2 gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

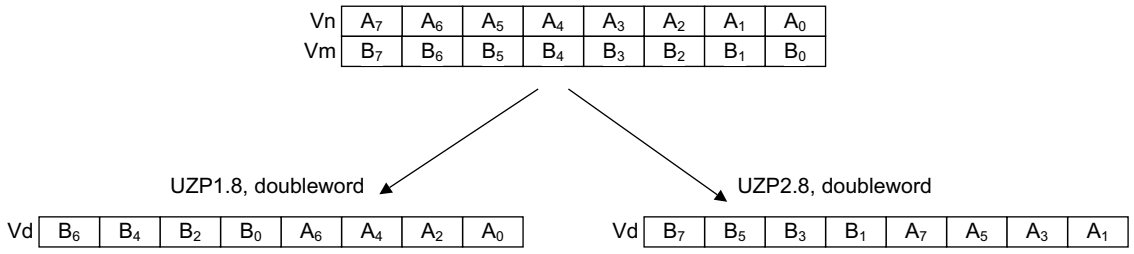
C7.2.399 UZP1

Unzip vectors (primary). This instruction reads corresponding even-numbered vector elements from the two source SIMD&FP registers, starting at zero, places the result from the first source register into consecutive elements in the lower half of a vector, and the result from the second source register into consecutive elements in the upper half of a vector, and writes the vector to the destination SIMD&FP register.

Note

This instruction can be used with `UZP2` to de-interleave two vectors.

The following figure shows the operation of UZP1 and UZP2 with the arrangement specifier 8B.



Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	size		0	Rm		0	0	0	1	1	0	Rn		Rd	
													op									

Advanced SIMD variant

```
UZP1 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B`	when `size = 00`, `Q = 0`
`16B`	when `size = 00`, `Q = 1`
`4H`	when `size = 01`, `Q = 0`
`8H`	when `size = 01`, `Q = 1`
`2S`	when `size = 10`, `Q = 0`
`4S`	when `size = 10`, `Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`2D` when `size = 11`, `Q = 1`

The encoding `size = 11`, `Q = 0` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operandl = V[n];
bits(datasize) operandh = V[m];
bits(datasize) result;

bits(datasize*2) zipped = operandh:operandl;
for e = 0 to elements-1
    Elem[result, e, esize] = Elem[zipped, 2*e+part, esize];

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

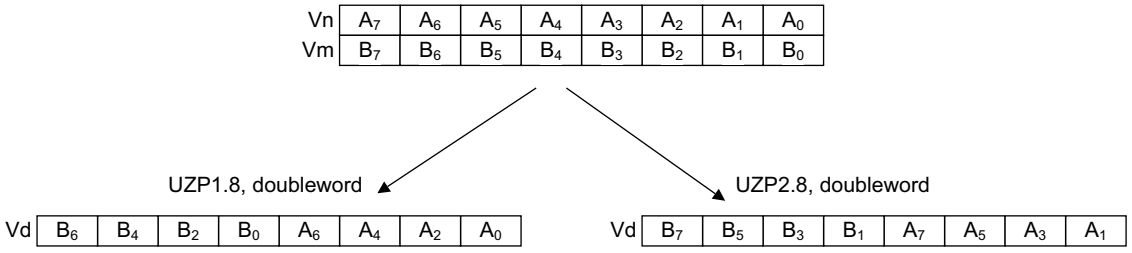
C7.2.400 UZP2

Unzip vectors (secondary). This instruction reads corresponding odd-numbered vector elements from the two source SIMD&FP registers, places the result from the first source register into consecutive elements in the lower half of a vector, and the result from the second source register into consecutive elements in the upper half of a vector, and writes the vector to the destination SIMD&FP register.

Note

This instruction can be used with `UZP1` to de-interleave two vectors.

The following figure shows the operation of UZP1 and UZP2 with the arrangement specifier 8B.



Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20–16	15	14	13	12	11	10	9–5	4–0
0	Q	0	0	1	1	1	0	size		0	Rm	0	1	0	1	1	0	Rn	Rd
												op							

Advanced SIMD variant

```
UZP2 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B`	when `size = 00`, `Q = 0`
`16B`	when `size = 00`, `Q = 1`
`4H`	when `size = 01`, `Q = 0`
`8H`	when `size = 01`, `Q = 1`
`2S`	when `size = 10`, `Q = 0`
`4S`	when `size = 10`, `Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`2D` when `size = 11`, `Q = 1`

The encoding `size = 11`, `Q = 0` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operandl = V[n];
bits(datasize) operandh = V[m];
bits(datasize) result;

bits(datasize*2) zipped = operandh:operandl;
for e = 0 to elements-1
    Elem[result, e, esize] = Elem[zipped, 2*e+part, esize];

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.401 XAR

Exclusive OR and Rotate performs a bitwise exclusive OR of the 128-bit vectors in the two source SIMD&FP registers, rotates each 64-bit element of the resulting 128-bit vector right by the value specified by a 6-bit immediate value, and writes the result to the destination SIMD&FP register.

This instruction is implemented only when ARMv8.2-SHA is implemented.

ARMv8.2

31 30 29 28	27 26 25 24	23 22 21 20	16	15 10	9 5	4 0
1 1 0 0	1 1 1 0	1 0 0	Rm	imm6	Rn	Rd

Advanced SIMD variant

```
XAR <Vd>.2D, <Vn>.2D, <Vm>.2D, #<imm6>
```

Decode for this encoding

```
if !HaveSHA3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler symbols

`<Vd>`	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Vn>`	Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
`<Vm>`	Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
`<imm6>`	Is a rotation right, encoded in "imm6".

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) tmp;
tmp = Vn EOR Vm;
V[d] = ROR(tmp<127:64>, UInt(imm6)):ROR(tmp<63:0>, UInt(imm6));
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C7.2.402 XTN, XTN2

Extract Narrow. This instruction reads each vector element from the source SIMD&FP register, narrows each value to half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements.

The `XTN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `XTN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	17	16	15	14	13	12	11	10	9 5	4 0
0	Q	0	0	1	1	1	0	size	1	0	0	0	0	1	0	0	1	0	1	0	Rn	Rd

Vector variant

```
XTN{2} <Vd>.<Tb>, <Vn>.<Ta>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Tb>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B` when `size = 00`, `Q = 0`

`16B` when `size = 00`, `Q = 1`

`4H` when `size = 01`, `Q = 0`

`8H` when `size = 01`, `Q = 1`

`2S` when `size = 10`, `Q = 0`

`4S` when `size = 10`, `Q = 1`

The encoding `size = 11`, `Q = x` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

`8H` when `size = 00`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

4S when `size = 01`

2D when `size = 10`

The encoding `size = 11` is reserved.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;
bits(2*esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    Elem[result, e, esize] = element<esize-1:0>;
Vpart[d, part] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

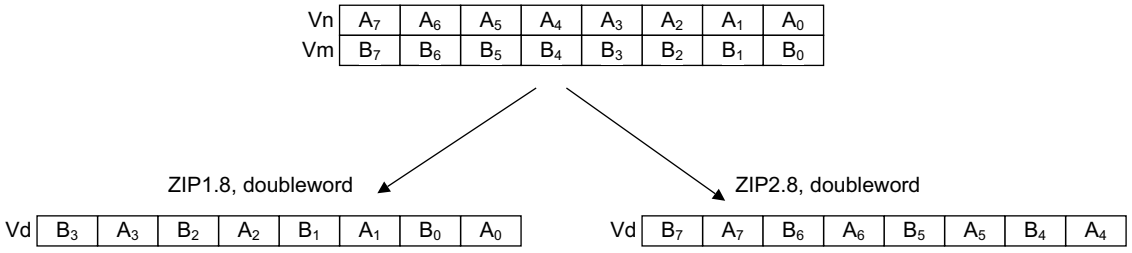
C7.2.403 ZIP1

Zip vectors (primary). This instruction reads adjacent vector elements from the lower half of two source SIMD&FP registers as pairs, interleaves the pairs and places them into a vector, and writes the vector to the destination SIMD&FP register. The first pair from the first source register is placed into the two lowest vector elements, with subsequent pairs taken alternately from each source register.

Note

This instruction can be used with `ZIP2` to interleave two vectors.

The following figure shows the operation of ZIP1 and ZIP2 with the arrangement specifier 8B.



Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29 28	27 26 25 24	23 22	21	20 16	15	14	13 12	11 10	9 5	4 0
0	Q	0 0	1 1 1 0	size	0	Rm	0	0	1 1	1 0	Rn	Rd
							op					

Advanced SIMD variant

```
ZIP1 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
integer pairs = elements DIV 2;
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B`	when `size = 00`, `Q = 0`
`16B`	when `size = 00`, `Q = 1`
`4H`	when `size = 01`, `Q = 0`
`8H`	when `size = 01`, `Q = 1`
`2S`	when `size = 10`, `Q = 0`
`4S`	when `size = 10`, `Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`2D` when `size = 11`, `Q = 1`

The encoding `size = 11`, `Q = 0` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer base = part * pairs;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, base+p, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, base+p, esize];

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

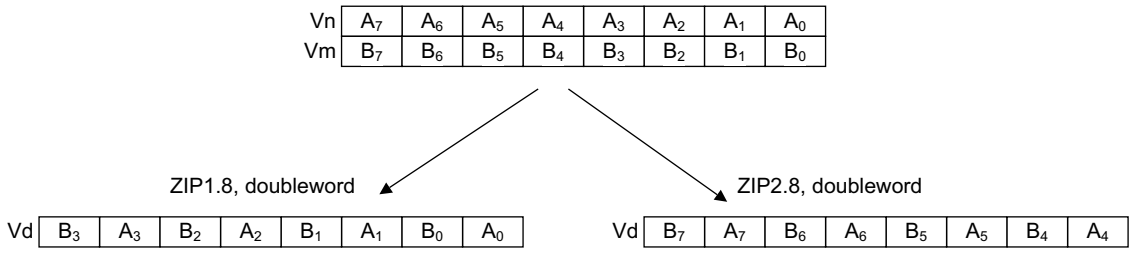
C7.2.404 ZIP2

Zip vectors (secondary). This instruction reads adjacent vector elements from the upper half of two source SIMD&FP registers as pairs, interleaves the pairs and places them into a vector, and writes the vector to the destination SIMD&FP register. The first pair from the first source register is placed into the two lowest vector elements, with subsequent pairs taken alternately from each source register.

Note

This instruction can be used with `ZIP1` to interleave two vectors.

The following figure shows the operation of ZIP1 and ZIP2 with the arrangement specifier 8B.



Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29 28	27 26 25 24	23 22	21	20 16	15	14	13 12	11 10	9 5	4 0
0	Q	0 0	1 1 1 0	size	0	Rm	0	1	1 1	1 0	Rn	Rd
							op					

Advanced SIMD variant

```
ZIP2 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
integer pairs = elements DIV 2;
```

Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B`	when `size = 00`, `Q = 0`
`16B`	when `size = 00`, `Q = 1`
`4H`	when `size = 01`, `Q = 0`
`8H`	when `size = 01`, `Q = 1`
`2S`	when `size = 10`, `Q = 0`
`4S`	when `size = 10`, `Q = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`2D` when `size = 11`, `Q = 1`

The encoding `size = 11`, `Q = 0` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer base = part * pairs;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, base+p, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, base+p, esize];

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Part D

The AArch64 System Level Architecture

Chapter D1
The AArch64 System Level Programmers' Model

This chapter describes the AArch64 system level programmers' model. It contains the following sections:

- *Exception levels* on page D1-2314.
- *Exception terminology* on page D1-2315.
- *Execution state* on page D1-2317.
- *Security state* on page D1-2318.
- *Virtualization* on page D1-2320.
- *Registers for instruction processing and exception handling* on page D1-2323.
- *Process state, PSTATE* on page D1-2330.
- *Program counter and stack pointer alignment* on page D1-2333.
- *Reset* on page D1-2335.
- *Exception entry* on page D1-2339.
- *Exception return* on page D1-2350.
- *Synchronous exception types, routing and priorities* on page D1-2354.
- *Asynchronous exception types, routing, masking and priorities* on page D1-2363.
- *Configurable instruction enables and disables, and trap controls* on page D1-2373.
- *System calls* on page D1-2396.
- *Mechanisms for entering a low-power state* on page D1-2397.
- *Self-hosted debug* on page D1-2403.
- *Event monitors* on page D1-2405.
- *Interprocessing* on page D1-2406.
- *The effect of implementation choices on the programmers' model* on page D1-2419.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D1.1 Exception levels

The Armv8-A architecture defines a set of Exception levels, EL0 to EL3, where:

- If ELn is the Exception level, increased values of *n* indicate increased software execution privilege.
- Execution at EL0 is called *unprivileged execution*.
- EL2 provides support for virtualization.
- EL3 provides support for switching between two Security states, Secure state and Non-secure state.

An implementation might not include all of the Exception levels. All implementations must include EL0 and EL1. EL2 and EL3 are optional.

Note

A PE is not required to implement a contiguous set of Exception levels. For example, it is permissible for an implementation to include only EL0, EL1, and EL3.

The effect of implementation choices on the programmers' model on page D1-2419 shows some example implementations.

When executing in AArch64 state, execution can move between Exception levels only on taking an exception or on returning from an exception:

- On taking an exception, the Exception level can only increase or remain the same.
- On returning from an exception, the Exception level can only decrease or remain the same.

The Exception level that execution changes to or remains in on taking an exception is called the *target Exception level* of the exception.

Each exception type has a target Exception level that is either:

- Implicit in the nature of the exception.
- Defined by configuration bits in the System registers.

An exception cannot target EL0.

Exception levels exist within a particular Security state. *The Armv8-A security model* on page D1-2318 describes this. When executing at an Exception level, the PE can access both of the following:

- The resources that are available for the combination of the current Exception level and the current Security state.
- The resources that are available at all lower Exception levels, provided that those resources are available to the current Security state.

This means that if the implementation includes EL3, then when execution is at EL3, the PE can access all resources available at all Exception levels, for both Security states.

Each Exception level other than EL0 has its own translation regime and associated control registers. For information on the translation regimes, see Chapter D5 *The AArch64 Virtual Memory System Architecture*.

D1.1.1 Typical Exception level usage model

The architecture does not specify what software uses which Exception level. Such choices are outside the scope of the architecture. However, the following is a common usage model for the Exception levels:

EL0 Applications.

EL1 OS kernel and associated functions that are typically described as *privileged*.

EL2 Hypervisor.

EL3 Secure monitor.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D1.2 Exception terminology

The following subsections define the terms used when describing exceptions:

- *Terminology for taking an exception*.
- *Terminology for returning from an exception*.
- *Exception levels*.
- *Definition of a precise exception*.
- *Definitions of synchronous and asynchronous exceptions* on page D1-2316.

D1.2.1 Terminology for taking an exception

An exception is *generated* when the PE first responds to an exceptional condition. The PE state at this time is the state the exception is *taken from*. The PE state immediately after taking the exception is the state the exception is *taken to*.

D1.2.2 Terminology for returning from an exception

To return from an exception, the PE must execute an exception return instruction. The PE state when an exception return instruction is committed for execution is the state the exception *returns from*. The PE state immediately after the execution of that instruction is the state the exception *returns to*.

D1.2.3 Exception levels

An Exception level, EL*n*, with a larger value of *n* than another Exception level, is described as being a *higher* Exception level than the other Exception level. For example, EL3 is a higher Exception level than EL1.

An Exception level with a smaller value of *n* than another Exception level is described as being a *lower* Exception level than the other Exception level. For example, EL0 is a lower Exception level than EL1.

An Exception level is described as:

- *Using AArch64* when execution in that Exception level is in the AArch64 Execution state.
- *Using AArch32* when execution in that Exception level is in the AArch32 Execution state.

D1.2.4 Definition of a precise exception

An exception is described as *precise* when the exception handler receives the PE state and memory system state that is consistent with the PE having executed all of the instructions up to but not including the point in the instruction stream where the exception was taken, and none afterwards.

Other than the *SError interrupt*, all exceptions taken to AArch64 state are required to be precise. For each occurrence of an SError interrupt, whether the interrupt is precise or imprecise is IMPLEMENTATION DEFINED.

Where a synchronous exception that is taken to AArch64 state is generated as part of an instruction that performs more than one single-copy atomic memory access, the definition of precise permits that the values in registers or memory affected by the instructions can be UNKNOWN, provided that:

- The accesses affecting those registers or memory locations do not, themselves, generate exceptions.
- The registers are not involved in the calculation of the memory address used by the instruction.

Also, for synchronous Data Aborts and Watchpoints from load or store instructions executed in AArch64 state:

- If the load or store instruction specifies writeback of a new base address, the base address is restored to the original value on taking the exception.
- If the instruction was a load to either the base address register or the offset register, that register is restored to the original value. Any other destination registers become UNKNOWN.
- If the instruction was a load that does not load the base address register or the offset register, then the destination registers become UNKNOWN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Examples of instructions that perform more than one single-copy atomic memory access are the AArch32 `LDM` and `STM` instructions and the AArch64 `LDP` and `STP` instructions.

Note

For the definition of a single-copy atomic access, see *Properties of single-copy atomic accesses* on page B2-120.

D1.2.5 Definitions of synchronous and asynchronous exceptions

An exception is described as *synchronous* if all of the following apply:

- The exception is generated as a result of direct execution or attempted execution of an instruction.
- The return address presented to the exception handler is guaranteed to indicate the instruction that caused the exception.
- The exception is precise.

For more information about synchronous exceptions, see *Synchronous exception types, routing and priorities* on page D1-2354.

An exception is described as *asynchronous* if any of the following apply:

- The exception is not generated as a result of direct execution or attempted execution of the instruction stream.
- The return address presented to the exception handler is not guaranteed to indicate the instruction that caused the exception.
- The exception is imprecise.

For more information about asynchronous exceptions, see *Asynchronous exception types, routing, masking and priorities* on page D1-2363.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D1.3 Execution state

The Execution states are:

AArch64 The 64-bit Execution state.

AArch32 The 32-bit Execution state. Operation in this state is compatible with Armv7-A operation.

Execution state on page A1-37 gives more information about them.

Exception levels *use* Execution states. For example, EL0, EL1 and EL2 might all be using AArch32, under EL3 using AArch64.

This means that:

- Different software layers, such as an application, an operating system kernel, and a hypervisor, executing at different Exception levels, can execute in different Execution states.
- The PE can change Execution states only either:
  - At reset.
  - On a change of Exception level.

Note

- *Typical Exception level usage model* on page D1-2314 shows which Exception levels different software layers might typically use.
- *The effect of implementation choices on the programmers' model* on page D1-2419 gives information on supported configurations of Exception levels and Execution states.

The interaction between the AArch64 and AArch32 Execution states is called *interprocessing*. *Interprocessing* on page D1-2406 describes this.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D1.4 Security state

The Armv8-A architecture provides two Security states, each with an associated physical memory address space, as follows:

Secure state When in this state, the PE can access both the Secure physical address space and the Non-secure physical address space.

Non-secure state When in this state, the PE:
- Can access only the Non-secure physical address space.
- Cannot access the Secure system control resources.

For information on how virtual addresses translate onto Secure physical and Non-secure addresses, see *About the Virtual Memory System Architecture (VMSA)* on page D5-2532.

D1.4.1 The Armv8-A security model

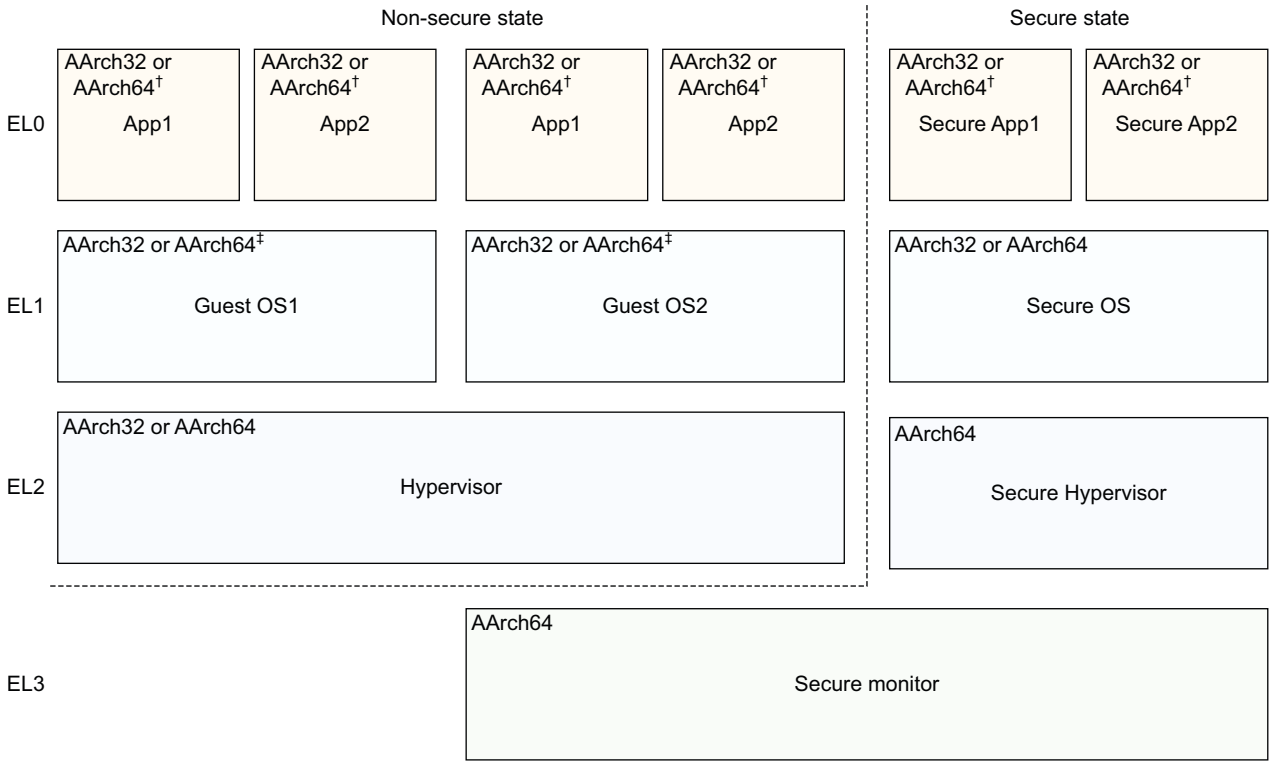
The principles of the Armv8-A security model are:

- If the implementation includes EL3, then it has two Security states, Secure and Non-secure, and:
  - EL3 exists only in Secure state.
  - A change from Non-secure state to Secure state can only occur on taking an exception to EL3.
  - A change from Secure state to Non-secure state can only occur on an exception return from EL3.
  - If ARMv8.4-SecEL2 is not implemented, EL2 exists only in Non-secure state.
  - If ARMv8.4-SecEL2 is implemented, EL2 can exist in Secure state. It is enabled when the value of SCR_EL3.EEL2 is 1.
- If the implementation does not include EL3, it has one Security state, that is:
  - IMPLEMENTATION DEFINED, if the implementation does not include EL2 or if ARMv8.4-SecEL2 is implemented.
  - Non-secure state, if the implementation includes EL2 and ARMv8.4-SecEL2 is not implemented.

Security model when EL3 is using AArch64 state

Figure D1-1 on page D1-2319 shows the security model when EL3 is using AArch64 state. The figure shows how instances of EL0 and EL1 are present in both Security states. It also shows the expected software usage of the different Exception levels.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta



† AArch64 permitted only if EL1 is using AArch64
‡ AArch64 permitted only if EL2 is using AArch64

Figure D1-1 Armv8-A security model when EL3 is using AArch64

For an overview of the Security model when EL3 is using AArch32, see Figure G1-1 on page G1-5680.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D1.5 Virtualization

The support for virtualization described in this section applies only to an implementation that includes EL2.

When enabled in the current Security state, EL2 provides a set of features that support virtualizing an Armv8-A implementation. The basic model of a virtualized system involves:

- A hypervisor, running in EL2, that is responsible for switching between *virtual machines*. A virtual machine comprises EL1 and EL0.
- A number of Guest operating systems. A Guest OS runs on a virtual machine in EL1.
- For each Guest operating system, applications, that run on the virtual machine of that Guest OS, usually in EL0.

Note

In some systems, a Guest OS is unaware that it is running on a virtual machine, and is unaware of any other Guest OS. In other systems, a hypervisor makes the Guest OS aware of these facts. The Armv8-A architecture supports both of these models.

The hypervisor assigns a *virtual machine identifier* (VMID) to each virtual machine.

EL2 supports Guest OS management and provides controls to:

- Provide virtual values for the contents of a small number of identification registers. A read of one of these registers by a Guest OS or the applications for a Guest OS returns the virtual value.
- *Trap* various operations, including memory management operations and accesses to many other registers. A trapped operation generates an exception that is taken to EL2. See *Configurable instruction enables and disables, and trap controls* on page D1-2373.
- Route interrupts to the appropriate one of:
  - The current Guest OS.
  - A Guest OS that is not currently running.
  - The hypervisor.

Armv8.1 introduces the Virtualization Host Extensions (VHE) that provide enhanced support for Type 2 hypervisors. For more information, see *Virtualization Host Extensions* on page D5-2638.

In an implementation that includes EL2:

- The implementation provides an independent translation regime for memory accesses from EL2, the EL2 translation regime. An implementation that includes ARMv8.1-VHE also supports an alternative EL2&0 translation regime.

  Note

  An implementation that includes ARMv8.1-VHE can be configured so that the EL2&0 translation regime is used both for accesses from EL2 and for accesses from EL0.

- For the EL1&0 translation regime, address translation occurs in two stages:
  - Stage 1 maps the *virtual address* (VA) to an *intermediate physical address* (IPA). This is managed at EL1, usually by a Guest OS. The Guest OS believes that the IPA is the *physical address* (PA).
  - Stage 2 maps the IPA to the PA. This is managed at EL2. The Guest OS might be completely unaware of this stage.
- When ARMv8.3-NV is implemented, a Guest Hypervisor can be run at EL1. For more information on how this affects address translation, see *Nested virtualization* on page D5-2644.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- When ARMv8.4-NV is implemented, then accesses of EL1 and EL2 registers that would be trapped are instead transformed into memory accesses. For more information, see *Enhanced support for nested virtualization* on page D5-2646.

For more information on the translation regimes, see Chapter D5 *The AArch64 Virtual Memory System Architecture*.

D1.5.1 The effect of implementing EL2 on the Exception model

An implementation that includes EL2 implements the following exceptions:

- *HVC* on page C6-933.
- Traps to EL2. *EL2 configurable controls* on page D1-2378, describes these.
- All of the virtual interrupts:
  - Virtual SError.
  - Virtual IRQ.
  - Virtual FIQ.

All virtual interrupts are always taken to EL1, and can only be taken from EL1 or EL0.

Each of the virtual interrupts can be independently enabled using controls at EL2.

Each of the virtual interrupts has a corresponding physical interrupt. See *Virtual interrupts*.

When a virtual interrupt is enabled, its corresponding physical exception is taken to EL2, unless EL3 has configured that physical exception to be taken to EL3.

For more information, see *Asynchronous exception types, routing, masking and priorities* on page D1-2363.

An implementation that includes EL2 also:

- Provides controls that can be used to route some synchronous exceptions. For more information, see:
  - *Routing exceptions from EL0 to EL2* on page D1-2354.
  - *Routing debug exceptions* on page D2-2429.
- Provides mechanisms to trap PE operations to EL2. See *EL2 configurable controls* on page D1-2378.

  When an operation is trapped to EL2, the hypervisor typically either:
  - Emulates the required operation. The application running in the Guest OS is unaware of the trap.
  - Returns an error to the Guest OS.

Virtual interrupts

The virtual interrupts have names that correspond to the physical interrupts, as shown in Table D1-1.

Table D1-1 The virtual interrupt

Physical interrupt	Corresponding virtual interrupt
SError	Virtual SError
IRQ	Virtual IRQ
FIQ	Virtual FIQ

Software executing in EL2 can use virtual interrupts to signal physical interrupts to EL1 and EL0. Example D1-1 on page D1-2322 shows a usage model for virtual interrupts.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Example D1-1 Virtual interrupt usage model

A virtual interrupt usage model is as follows:

1. Software executing at EL2 routes a physical interrupt to EL2.
2. When a physical interrupt of that type occurs, the exception handler executing in EL2 determines whether the interrupt can be handled in EL2 or requires routing to a Guest OS in EL1. If the interrupt requires routing to a Guest OS:
   - If the Guest OS is currently running, the hypervisor uses the appropriate virtual interrupt type to signal the physical interrupt to the Guest OS.
   - If the Guest OS is not currently running, the physical interrupt is marked as pending for the guest OS. When the hypervisor next switches to the virtual machine that is running that Guest OS, the hypervisor uses the appropriate virtual interrupt type to signal the physical interrupt to the Guest OS.

A hypervisor can prevent EL1 and EL0 from distinguishing a virtual interrupt from a physical interrupt.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D1.6 Registers for instruction processing and exception handling

In the Arm architecture, registers fall into two main categories:

- Registers that provide system control or status reporting. These are described in Chapter D13 *AArch64 System Register Descriptions*.
- Registers that are used in instruction processing, for example to accumulate a result, and in handling exceptions. This section introduces these registers, for execution in AArch64 state.

This section contains the following subsections:

- *The general purpose registers, R0-R30*.
- *The stack pointer registers*.
- *The SIMD and floating-point registers, V0-V31* on page D1-2324.
- *Saved Program Status Registers (SPSRs)* on page D1-2324.
- *Exception Link Registers (ELRs)* on page D1-2328.

D1.6.1 The general purpose registers, R0-R30

The general purpose register bank is used when processing instructions in the base instruction set. It comprises 31 general purpose registers, R0-R30.

These registers can be accessed as 31 64-bit registers, X0-X30, or 31 32-bit registers, W0-W30. See *Register size* on page C6-772.

For information on the format of these registers, see *Registers in AArch64 state* on page B1-107.

D1.6.2 The stack pointer registers

In AArch64 state, in addition to the general purpose registers, a dedicated stack pointer register is implemented for each implemented Exception level. The stack pointer registers are:

- SP_EL0 and SP_EL1.
- If the implementation includes EL2, SP_EL2.
- If the implementation includes EL3, SP_EL3.

Note

The four stack pointer register names define an architecture state requirement for four registers. For information on how to access these registers, and access restrictions, see *Special-purpose registers* on page C5-389.

For information on stack pointer alignment restrictions, see *SP alignment checking* on page D1-2333.

Stack pointer register selection

When executing at EL0, the PE uses the EL0 stack pointer, SP_EL0.

When executing at any other Exception level, the PE can be configured to use either SP_EL0 or the stack pointer for that Exception level, SP_ELx.

By default, taking an exception selects the stack pointer for the target Exception level, SP_ELx. For example, taking an exception to EL1 selects SP_EL1. Software executing at the target Exception level can then choose to change the stack pointer to SP_EL0 by updating PSTATE.SP.

This applies even if taking the exception does not change the Exception level. For example, if the PE is executing at EL1 and the PE is using the SP_EL0 stack pointer, then on taking an exception that targets EL1, the stack pointer changes to SP_EL1.

The selected stack pointer can be indicated by a suffix to the Exception level:

t Indicates use of the SP_EL0 stack pointer.

h Indicates use of the SP_ELx stack pointer.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

The t and h suffixes are based on the terminology of *thread* and *handler*.

Table D1-2 shows the set of stack pointer options.

Table D1-2 AArch64 stack pointer options

Exception level (EL)	Stack pointer (SP) options
EL0	SP_EL0t
EL1	SP_EL1t, SP_EL1h
EL2	SP_EL2t, SP_EL2h
EL3	SP_EL3t, SP_EL3h

D1.6.3 The SIMD and floating-point registers, V0-V31

The SIMD and floating-point instructions share a common bank of registers for floating-point, vector, and other SIMD-related scalar operations.

The SIMD and floating-point register bank comprises 32 quadword (128-bit) registers, V0-V31.

These registers can be accessed as:

- 32 doubleword (64-bit) registers, D0-D31.
- 32 word (32-bit) registers, S0-S31.
- 32 halfword (16-bit) registers, H0-H31.
- 32 byte (8-bit) registers, B0-B31.

For information on the format of these registers, see *Registers in AArch64 state* on page B1-107.

D1.6.4 Saved Program Status Registers (SPSRs)

The *Saved Program Status Registers* (SPSRs) are used to save PE state on taking exceptions.

In AArch64 state, there is an SPSR at each Exception level exceptions can be taken to, as follows:

- SPSR_EL1, for exceptions taken to EL1 using AArch64.
- If EL2 is implemented, SPSR_EL2, for exceptions taken to EL2 using AArch64.
- If EL3 is implemented, SPSR_EL3, for exceptions taken to EL3 using AArch64.

Note

Exceptions cannot be taken to EL0.

When the PE takes an exception, the PE state is saved from PSTATE in the SPSR at the Exception level the exception is taken to. For example, if the PE takes an exception to EL1, the PE state is saved in SPSR_EL1. For more information on PSTATE, see *Process state, PSTATE* on page D1-2330.

Saving the PE state means the exception handler can:

- On return from the exception, restore the PE state to the state stored in the SPSR at the Exception level the exception is returning from. For example, on returning from EL1, the PE state is restored to the state stored in SPSR_EL1.
- Examine the value that PSTATE had when the exception was taken, for example to determine the Execution state and Exception level in which the instruction that caused an exception was executed.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

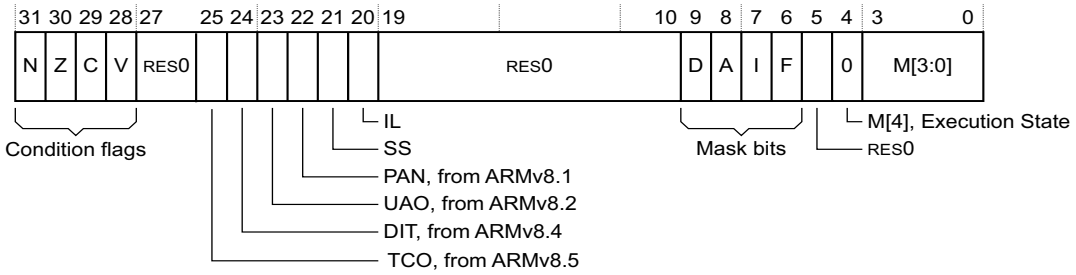
- All PSTATE fields are saved, including those which have no direct read and write access, and those that are meaningful only in AArch32 state.
- Those PSTATE fields that are meaningful only in AArch32 state are saved when an exception is taken from AArch32 state to AArch64 state.

The SPSRs are UNKNOWN on reset.

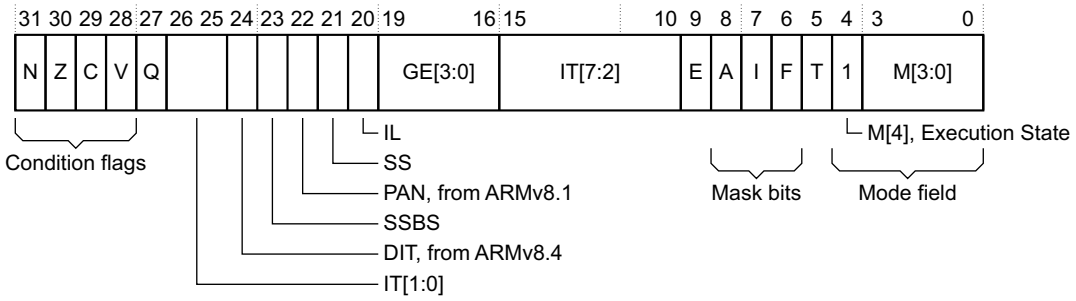
SPSR format for exceptions taken to AArch64 state

Exceptions can be taken to AArch64 state from AArch64 state or AArch32 state:

- For an exception taken to AArch64 state from AArch64 state, the SPSR bit assignments are:



- For an exception taken to AArch64 state from AArch32 state, the SPSR bit assignments are:



The following list describes the bit assignments:

N, Z, C, V, bits[31:28]

Shows the values of the PSTATE.{N, Z, C, V} Condition flags immediately before the exception was taken.

Bits[27:24], for exceptions taken from AArch64 state

Reserved, RES0.

Q, bit[27], for exceptions taken from AArch32 state

Shows the value of PSTATE.Q immediately before the exception was taken.

IT[1:0], bits[26:25], for exceptions taken from AArch32 state

See *IT[7:2]* on page D1-2326.

TCO, bit [25], for exceptions taken from AArch64 state, when ARMv8.5-MemTag is implemented

Shows the value of PSTATE.TCO immediately before the exception was taken.

DIT, bit[24], from Armv8.4

Shows the value of PSTATE.DIT immediately before the exception was taken.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

UAO, bit[23], from Armv8.2, for exceptions taken from AArch64 state

Shows the value of PSTATE.UAO immediately before the exception was taken.

This bit is RES0 when ARMv8.2-UAO is not implemented.

SSBS, bit[23], for exceptions taken from AArch32 state

Shows the value of PSTATE.SSBS immediately before the exception was taken.

This bit is RES0 when ARMv8.0-SSBS is not implemented.

PAN, bit[22], from Armv8.1

Shows the value of PSTATE.PAN immediately before the exception was taken.

This bit is RES0 when ARMv8.1-PAN is not implemented.

SS, bit[21] The Software Step bit.

SPSR_ELx.SS is used by a debugger to initiate a Software Step exception. The SS bit also indicates which software step state machine state the PE was in. See *Software Step exceptions* on page D2-2472.

IL, bit[20] Illegal Execution state bit. Shows the value of PSTATE.IL immediately before the exception was taken. See *Illegal return events from AArch64 state* on page D1-2351.

Bits[19:13], for exceptions taken from AArch64 state

Reserved, RES0.

SSBS, bit[12], for exceptions taken from AArch64 state

Shows the value of PSTATE.SSBS immediately before the exception was taken.

This bit is RES0 when ARMv8.0-SSBS is not implemented.

BTYPE, bits[11:10], from Armv8.5, for exceptions taken from AArch64 state

Shows the value of PSTATE.BTYPE immediately before the exception was taken.

This bit is RES0 when ARMv8.5-BTI is not implemented.

Bits[12:10], for exceptions taken from AArch32 state

Reserved, RES0.

GE[3:0], bits [19:16], for exceptions taken from AArch32 state

Shows the value of PSTATE.GE immediately before the exception was taken.

IT[7:2], bits [15:10], for exceptions taken from AArch32 state

With *IT[1:0]* on page D1-2325, shows the value of PSTATE.IT before the exception was taken.

D, bit[9], for exceptions taken from AArch64 state

The debug exception mask bit. Shows the value of PSTATE.D immediately before the exception was taken. See *The PSTATE debug mask bit, D* on page D1-2403.

E, bit[9], for exceptions taken from AArch32 state

Endianness of data accesses. Shows the value of PSTATE.E immediately before the exception was taken.

A, I, F, bits[8:6]

Shows the values of the PSTATE.{A, I, F} exception mask bits immediately before the exception was taken:

A, bit[8] SError interrupt mask bit.

I, bit[7] IRQ mask bit.

F, bit[6] FIQ mask bit.

See *Asynchronous exception masking* on page D1-2367.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bit[5], for exceptions taken from AArch64 state

Reserved, RES0.

T, bit[5], for exceptions taken from AArch32 state

Shows the value of PSTATE.T immediately before the exception was taken.

M[4:0], bits[4:0], for exceptions taken from AArch64 state

Note

The name of this field is inherited from Armv7, where the M field specified the *PE mode*.

M[4] The value of this is 0. M[4] encodes the value of PSTATE.nRW, that indicates the Execution state from which the exception was taken.

M[3:0] Encodes the Exception level and the stack pointer register selection, as shown in Table D1-3.

Table D1-3 M[3:0] encodings, for exceptions taken from AArch64 state

M[3:0][a]	Exception level and stack pointer
`0b1101`	EL3h
`0b1100`	EL3t
`0b1001`	EL2h
`0b1000`	EL2t
`0b0101`	EL1h
`0b0100`	EL1t
`0b0000`	EL0t

a. All M[3:0] encodings not shown in the table are reserved.

The M[3:0] encoding comprises:

M[3:2] Encodes the Exception level, 0-3.

M[1] Reserved, RES0. If set to 1 at the time of an exception return, then that exception return is treated as an Illegal Execution state exception return.

M[0] Selects the SP:

- **0** SP_EL0. Indicated by a t suffix on the Exception level.
- **1** SP_ELx, where *x* is the value of M[3:2]. Indicated by an h suffix on the Exception level.

See *Stack pointer register selection* on page D1-2323.

M[4:0], bits[4:0], for exceptions taken from AArch32 state

The Mode field. Specifies the AArch32 PE mode.

M[4] The value of this is 1. M[4] encodes the value of PSTATE.nRW, that indicates the Execution state from which the exception was taken.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

M[3:0] Encodes the AArch32 PE mode that the PE was in immediately before the exception was taken, as shown in Table D1-4.

Table D1-4 M[3:0] encodings, for exceptions taken from AArch32 state

M[3:0]	AArch32 PE mode
0b0000	User
0b0001	FIQ
0b0010	IRQ
0b0011	Supervisor
0b0111	Abort
0b1010	Hyp
0b1011	Undefined
0b1111	System

Note

In this description of SPSR:

- For exceptions taken from AArch64 state, PSTATE links to the AArch64 system level description.
- For exceptions taken from AArch32 state, PSTATE links to the AArch32 system level description.

SPSR bits:

- That are defined as RES0 on an exception taken from AArch32 state are ignored on any exception return to AArch32 state.
- That are defined as RES0 on an exception taken from AArch64 state are ignored on any exception return to AArch64 state.

Pseudocode description of SPSR operations

The `SPSR[]` pseudocode function accesses the current SPSR, and is common to AArch32 and AArch64 operations.

The `SetPSTATEFromPSR()` pseudocode function updates PSTATE from an SPSR.

D1.6.5 Exception Link Registers (ELRs)

Exception Link Registers hold preferred exception return addresses.

Whenever the PE takes an exception, the preferred return address is saved in the ELR at the Exception level the exception is taken to. For example, whenever the PE takes an exception to EL1, the preferred return address is saved in ELR_EL1.

On an exception return, the PC is restored to the address stored in the ELR. For example, on returning from EL1, the PC is restored to the address stored in ELR_EL1.

AArch64 state provides an ELR for each Exception level exceptions can be taken to. The ELRs that AArch64 state provides are:

- ELR_EL1, for exceptions taken to EL1.
- If EL2 is implemented, ELR_EL2, for exceptions taken to EL2.
- If EL3 is implemented, ELR_EL3, for exceptions taken to EL3.

On taking an exception from AArch32 state to AArch64 state, bits[63:32] of the ELR are set to zero.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The preferred return address depends on the nature of the exception. For more information, see *Preferred exception return address* on page D1-2341.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D1.7 Process state, PSTATE

In the Armv8-A architecture, Process state or PSTATE is an abstraction of process state information. All of the instruction sets provide instructions that operate on elements of PSTATE.

PSTATE includes all of the following:

- Fields that are meaningful only in AArch32 state.
- Fields that are meaningful only in AArch64 state.
- Fields that are meaningful in both Execution states.

PSTATE is defined in pseudocode as the `PSTATE` structure, of type `ProcState`. `ProcState` is defined in Chapter J1 *Armv8 Pseudocode*.

The PSTATE fields that are meaningful in AArch64 state are:

The Condition flags

N Negative Condition flag.

Z Zero Condition flag.

C Carry Condition flag.

V Overflow Condition flag.

Process state, PSTATE on page B1-108 gives more information about these flags.

The Execution state controls

SS Software Step bit, see *Software Step exceptions* on page D2-2472. On a reset or taking an exception to AArch64 state, this bit is set to 0.

IL Illegal Execution state bit, see *The Illegal Execution state exception* on page D1-2353. On a reset or taking an exception to AArch64 state, this bit is set to 0.

nRW Current Execution state, see *Execution state* on page D1-2317. This bit is 0 when the current Execution state is AArch64. This bit is set to 0:

- On reset into an Exception level that is using AArch64.
- On taking an exception to an Exception level that is using AArch64.

EL Current Exception level, see *Exception levels* on page D1-2314. On a reset to AArch64 state, this field holds the encoding for the highest implemented Exception level.

Note

The Arm architecture requires that a PE resets into the highest implemented Exception level.

SP Stack pointer register selection bit, see *Stack pointer register selection* on page D1-2323. On a reset or taking an exception to AArch64 state, this bit is set to 1, meaning that SP_ELx is selected.

The exception mask bits

D Debug exception mask bit, see *The PSTATE debug mask bit, D* on page D1-2403. On a reset or taking an exception to AArch64 state, this bit is set to 1.

A, I, F Asynchronous exception mask bits:

A SError interrupt mask bit.

I IRQ interrupt mask bit.

F FIQ interrupt mask bit.

See *Asynchronous exception types, routing, masking and priorities* on page D1-2363. On a reset or taking an exception to AArch64 state, each of these bits is set to 1.

Access control bits

PAN *Privileged Access Never* (PAN) state bit. For more information, see *About PSTATE.PAN* on page D5-2607.

This bit is implemented only when ARMv8.1-PAN is implemented.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

UAO *User Access Override* (UAO) bit. For more information, see *About PSTATE.UAO* on page D5-2608.

This bit is implemented only when ARMv8.2-UAO is implemented.

TCO *Tag Check Override* (TCO) bit. For more information see Chapter D6 *Armv8.5 Memory Tagging Extension*.

This bit is implemented only when ARMv8.5-MemTag is implemented.

BTYPE Branch target identification bit. For more information, see *About PSTATE.BTYPE* on page D5-2608.

This bit is implemented only when ARMv8.5-BTI is implemented.

Timing control bits

DIT *Data Independent Timing* (DIT) bit. For more information, see *About PSTATE.DIT* on page B1-113.

This bit is implemented only when ARMv8.4-DIT is implemented.

On a reset to AArch64 state, this bit is set to 0.

Speculation control bits

SSBS *Speculative Store Bypass Safe* (SSBS) bit. For more information, see *Speculative Store Bypass Safe (SSBS)* on page B2-133.

This bit is implemented only when ARMv8.0-SSBS is implemented.

On reset, this bit is set to an IMPLEMENTATION DEFINED value.

D1.7.1 Accessing PSTATE fields

In AArch64 state, PSTATE fields can be accessed using Special-purpose registers that can be directly read using the MRS instruction, and directly written using the MSR (register) instructions. Table D1-5 shows the Special-purpose registers that access the PSTATE fields that hold AArch64 state, when the PE is in AArch64 state. All other PSTATE fields do not have direct read and write access.

Table D1-5 Accessing PSTATE fields using MRS and MSR (register)

Special-purpose register	PSTATE fields
NZCV	N, Z, C, V
DAIF	D, A, I, F
CurrentEL	EL
SPSel	SP
PAN	PAN
UAO	UAO
DIT	DIT
SSBS	SSBS

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Software can also use the MSR (immediate) instruction to directly write to PSTATE.{D, A, I, F, SP, PAN, UAO, SSBS, TCO}. Table D1-6 shows the MSR (immediate) operands that can directly write to these PSTATE fields when the PE is in AArch64 state.

Table D1-6 Accessing PSTATE.{D, A, I, F, SP} using MSR (immediate)

Operand	PSTATE fields	Notes
DAIFSet	D, A, I, F	Directly sets any of the PSTATE.{D,A, I, F} bits to 1
DAIFClr	D, A, I, F	Directly clears any of the PSTATE.{D, A, I, F} bits to 0
SPSel	SP	Directly sets PSTATE.SP to either 1 or 0
PAN	PAN	Directly sets PSTATE.PAN to either 1 or 0
UAO	UAO	Directly sets PSTATE.UAO to either 1 or 0
DIT	DIT	Directly sets PSTATE.DIT to either 1 or 0
SSBS	SSBS	Directly sets PSTATE.SSBS to either 1 or 0
TCO	TCO[a]	Directly sets PSTATE.TCO to either 1 or 0.

a. PSTATE.TCO can also be accessed by an `MSR Xt` instruction.

PSTATE.{N, Z, C, V} can be accessed at EL0. Access to PSTATE.{D, A, I, F} at EL0 using AArch64 depends on SCTLR_EL1.UMA, see *Traps to EL1 of EL0 accesses to the PSTATE.{D, A, I, F} interrupt masks* on page D1-2377. All other PSTATE access instructions can be executed at EL1 or higher and are UNDEFINED at EL0.

Writes to the PSTATE fields have side-effects on various aspects of the PE operation. All of these side-effects are guaranteed:

- Not to be visible to earlier instructions in the execution stream.
- To be visible to later instructions in the execution stream.

D1.7.2 The Saved Program Status Registers (SPSRs)

On taking an exception, PSTATE is preserved in the SPSR of the Exception level the exception is taken to. The SPSRs are described in *Saved Program Status Registers (SPSRs)* on page D1-2324.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D1.8 Program counter and stack pointer alignment

This section contains the following:

- *PC alignment checking*.
- *SP alignment checking*.

D1.8.1 PC alignment checking

PC alignment checking generates a PC alignment fault exception associated with the instruction fetch if, in AArch64 state, there is an attempt to architecturally execute an instruction that was fetched with a misaligned PC. A misaligned PC is when bits[1:0] of the PC are not `0b00`.

Note

As with Instruction Aborts, speculative fetching of an instruction does not generate an exception. An exception occurs only on an attempt to architecturally execute the instruction.

If an exception is generated as a result of an instruction fetch at EL0, it is taken to EL1. If an exception occurs when HCR_EL2.TGE bit is 1 and EL2 is enabled in the current Security state, it is taken to EL2. If an exception is generated as a result of an instruction fetch at any other Exception level, the Exception level is unchanged.

A PC misalignment sets the EC field in the *Exception Syndrome Register* (ESR) to `0x22`, for the ESR associated with the target Exception level.

When the exception is taken to an Exception level using AArch64, the associated Exception Link Register holds the entire PC in its misaligned form, as does the FAR_ELx for the Exception level that the exception is taken to.

Exception return and PC alignment on page D1-2351 gives more information on PC alignment checking associated with exception returns.

Note

A misalignment of the PC is a common indication of a serious error, for example software corruption of an address.

The pseudocode function `AArch64.CheckPCAlignment()` performs PC alignment checking in AArch64 state. When necessary it calls `AArch64.PCAlignmentFault()` to generate an exception.

D1.8.2 SP alignment checking

A *misaligned stack pointer* is where bits[3:0] of the stack pointer are not `0b0000`, when the stack pointer is used as the base address of the calculation, regardless of any offset applied by the instruction.

The PE can be configured so that if a load or store instruction uses a misaligned stack pointer, the PE generates an SP alignment fault exception on the attempt to execute the instruction. In this configuration, `CheckSPAlignment()` performs the stack pointer check, and calls `AArch64.SPAlignmentFault()` if a misaligned stack pointer is found.

Note

- As with Data Aborts, a speculative data access to memory using the stack pointer does not generate the exception. The exception occurs only on an attempt to architecturally execute the instruction.
- Prefetch memory abort instructions do not cause synchronous exceptions. See *Prefetch memory* on page C3-220.

Stack pointer alignment checking is only performed in AArch64 state, and can be enabled for each Exception level as follows:

- SCTLR_EL1.{SA0, SA} controls EL0 and EL1, respectively.
- SCTLR_EL2.SA controls EL2.
- SCTLR_EL3.SA controls EL3.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If an exception is generated as a result of a load or store at EL0, it is taken as an exception to EL1. If an exception occurs when the HCR_EL2.TGE bit is set and EL2 is enabled in the current Security state, it is taken to EL2. If an exception is generated as a result of a load or store at any other Exception level, the Exception level is unchanged.

A stack pointer misalignment sets the EC field to `0x26`, in the ESR associated with the target Exception level. If memory alignment checking and stack pointer alignment checking are enabled, then an SP alignment fault has priority in setting the value of the EC field, in the ESR associated with the target Exception level.

The pseudocode function `CheckSPAlignment()` performs the stack pointer alignment check. When necessary it calls `AArch64.SPAlignmentFault()` to generate an exception.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D1.9 Reset

The Armv8-A architecture supports the following resets:

Cold reset Resets all of the logic on which the PE executes, including the integrated debug functionality.

In some contexts, this logic is described as belonging to the *Cold reset domain*.

Warm reset Resets some of the logic on which the PE executes. However, some state is purposefully unchanged by a Warm reset.

In some contexts, this logic is described as belonging to the *Warm reset domain*.

All logic on the which the PE executes that is reset by a Warm reset is also reset by a Cold reset.

Note

The Armv8-A architecture also supports an *external debug reset*. See *External debug register resets* on page H8-7070.

If an RMR_ELx register is implemented:

- A Warm reset permits debugging across a reset of the PE logic.
- Writing 1 to RMR_ELx.RR requests a Warm reset.

The mechanisms, other than RMR_ELx.RR, to assert these resets are IMPLEMENTATION DEFINED. It is IMPLEMENTATION DEFINED whether:

- It is possible to independently assert an External Debug reset and a Cold reset.
- It is possible to assert a Warm reset, as opposed to asserting a Cold reset, other than by the use of RMR_ELx.RR.

Note

Arm recommends that:

- If separate Core and Debug power domains are implemented, as described in *Reset and debug* on page H6-7041, then a Cold reset can be asserted independently of External Debug reset.
- A Warm reset can be asserted to permit debugging across a reset of the PE logic.

This means that an implementation can define other resets according to the requirements the implementation or system must fulfil. These other resets are outside the scope of the Armv8-A architecture. However, they can be mapped onto the resets described here.

In the description that follows, the term *reset* is used in contexts where there is no difference between the effect of a Cold reset and the effect of a Warm reset.

On a reset, the PE enters the highest implemented Exception level.

If the highest implemented Exception level can use either Execution state, then:

- The implementation must include a *Reset Management Register* (RMR). Only one RMR is implemented. The RMR implemented is the RMR is associated with the highest Exception level.
- On a Cold reset, the Execution state entered is determined by a configuration input signal.
- On a Warm reset, the Execution state entered is determined by RMR_ELx.AA64.

If the highest implemented Exception level is configured to use AArch64 state, then on reset:

- The stack pointer for the highest implemented Exception level, SP_ELx, is selected.
- Execution starts at an IMPLEMENTATION DEFINED address, anywhere in the physical address range. The RVBAR associated with the highest implemented Exception level, RVBAR_EL1, RVBAR_EL2, or RVBAR_EL3, holds this address.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The remainder of this section contains the following:

- *PE state on reset to AArch64 state*.
- *Code sequence to use RMR_ELx.RR to request a Warm reset* on page D1-2338.

For more information about reset see:

- *Behavior of caches at reset* on page D4-2505.
- *TLB behavior at reset* on page D5-2666.
- *Reset and debug* on page H6-7041.

D1.9.1 PE state on reset to AArch64 state

Note

See the *ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0 and version 4.0* for the reset requirements for GIC System registers.

Immediately after a reset, much of the PE state is UNKNOWN. However, some of the PE state is defined. If the PE resets to AArch64 state using either a Cold or a Warm reset, the PE state that is defined is as follows:

- Each of the PSTATE.{D, A, I, F} interrupt masks is set to 1.
- The Software step control bit, PSTATE.SS, is set to 0.
- The IL process state bit, PSTATE.IL, is set to 0.
- All general-purpose, and SIMD and floating-point registers are UNKNOWN.
- The ELR and SPSR for each Exception level are UNKNOWN.
- The stack pointer register for each Exception level is UNKNOWN.
- The global exclusive monitor and local exclusive monitor for the PE are UNKNOWN.
- Unless explicitly defined in this subsection, each System register at each Exception level is in an architecturally UNKNOWN state.
- The TLBs and caches are in an IMPLEMENTATION DEFINED state. This means that the TLBs, the caches, or both, might require invalidation using IMPLEMENTATION DEFINED invalidation sequences before the memory management system is enabled or Normal memory accesses are permitted to be Cacheable.

  Note

  — On reset, System register Cacheability control fields force all Normal memory accesses to be treated as Non-cacheable. This applies only for the translation regime used by the Exception level and Security state entered on reset. For information about these controls see *Enabling and disabling the caching of memory accesses* on page D4-2503.

  — The implementation might include IMPLEMENTATION DEFINED resets. If it does, each of these resets might treat the cache and TLB state differently. The Armv8-A architecture permits this.

  — Different IMPLEMENTATION DEFINED invalidation sequences might be required for different IMPLEMENTATION DEFINED resets.

  — In some implementations, the IMPLEMENTATION DEFINED invalidation sequence might be a NOP.

- In the SCTLR_ELx for the highest implemented Exception level:
  — Each of the {M, C, I} bits is set to 0
  — The EE bit is set to an IMPLEMENTATION DEFINED value, typically defined by a configuration input.
- If an RMR is implemented, RMR_ELx.RR is set to 0. EL*x* in this context is the highest implemented Exception level.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The enables for the counter event stream are set to 0. This means that the following bits are set to 0:
  - CNTKCTL_EL1.EVNTEN.
  - If the implementation includes EL2, CNTHCTL_EL2.EVNTEN.
- PMCR_EL0.E is set to 0.

  Note

  This means the Performance Monitors cannot assert interrupts at reset.

- OSDLR_EL1.DLK bit is set to 0.
- Each of MDCCINT_EL1.{TX, RX} is set to 0.
- EDPRCR.CWRR is set to 0.
- EDPRSR.SR is set to 1.
- If the implementation includes EL3, then each of MDCR_EL3.{EPMAD, EDAD, SPME} is set to 0.
- If the implementation includes EL2, then MDCR_EL2.HPMN is set to the value of PMCR_EL0.N.
- EDESR.OSUC is set to 0.
- If ARMv8.3-DoPD is not implemented,EDESR.SS is set to the value of EDECR.SS.
- If ARMv8.3-DoPD is implemented, EDESR.RC is set to the value of CTIDEVCTL.RCE. Otherwise EDESR.RC is set to the value of EDECR.RCE.

  Note

  On an External debug reset, EDECR.{SS, RCE} are set to 0. If ARMv8.3-DoPD is implemented, CTIDEVCTL.{OSUCE, RCE} are set to 0.

Additionally, for a Cold reset into AArch64 state:

- If an RMR is implemented, RMR_ELx.AA64 is set to 1. EL*x* in this context is the highest implemented Exception level.
- Each of MDCCSR_EL0.{TXfull, RXfull} is set to 0.
- If ARMv8.3-DoPD is not implemented, DBGPRCR_EL1.CORENPRDRQ is set to the value of EDPRCR.COREPURQ.

  Note

  An External Debug reset sets EDPRCR.COREPURQ to 0, see *External debug register resets* on page H8-7070. If an External Debug reset and a Cold reset coincide, both DBGPRCR_EL1.CORENPRDRQ and EDPRCR.COREPURQ are reset to 0.

  If ARMv8.3-DoPD is implemented, DBGPRCR_EL1.CORENPRDRQ is set to 1 if the powerup request is implemented and asserted, otherwise is set to 0.

- The debug CLAIM bits are reset to 0.

  Note

  These are the bits that are set to 1 by writing to DBGCLAIMSET_EL1.CLAIM, and cleared to 0 by writing to DBGCLAIMCLR_EL1.CLAIM.

- Each of EDSCR.{RXO, TXU, INTdis, TDA, MA, HDE, ERR, RXfull, TXfull} is set to 0.
- Each of EDECCR.{NSE, SE} is set to 0.
- If ARMv8.3-DoPD is implemented, EDECR.SS is set to 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- OSLSR_EL1.OSLK is set to 1.
- In the EDPRSR:
  - The SPMAD, SDAD fields are set to 0.
  - The SPD field is set to 1.
- Each field of AMCNTENCLR0_EL0, AMCNTENCLR1_EL0, AMCNTENSET0_EL0, and AMCNTENSET1_EL0 is set to 0.
- Each of the implemented architected activity monitor counters AMEVCNTR0<n>_EL0 and each of the implemented auxiliary activity monitor counters AMEVCNTR1<n>_EL0 are set to 0.

For more information about resets in AArch64 System registers, see Chapter D13 *AArch64 System Register Descriptions*.

D1.9.2 Code sequence to use RMR_ELx.RR to request a Warm reset

The following assembler sequence uses RMR_ELx.RR to request a Warm reset:

```
; in addition, interrupts and debug requests for this PE should be disabled
; in the system before running this sequence to ensure the WFI suspends execution
    MOV Wy, #3        ; for AArch64, #2 for AArch32; y is any register
    DSB               ; ensure all stores etc are complete
    MSR RMR_ELx, Wy   ; request the reset
    ISB               ; synchronise change to the RMR
Loop
    WFI               ; enter a quiescent state
    B Loop
```

D1.9.3 Pseudocode description of reset

The `AArch64.TakeReset()` pseudocode function performs a reset into AArch64 state.

`AArch64.TakeReset()` calls the functions `AArch64.ResetGeneralRegisters()`, `AArch64.ResetSIMDFPRegisters()`, `AArch64.ResetSpecialRegisters()`, `AArch64.ResetSystemRegisters()`, and `ResetExternalDebugRegisters()`.

`AArch64.ResetSystemRegisters()` resets all System registers to their reset state as defined in the register descriptions in *PE state on reset to AArch64 state* on page D1-2336 and Chapter D13 *AArch64 System Register Descriptions*.

Note

The `AArch64.ResetSystemRegisters()` function only resets the System registers.

`ResetExternalDebugRegisters()` resets all external debug registers to their reset state as defined in the register descriptions in Chapter H9 *External Debug Register Descriptions*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D1.10 Exception entry

Exceptions are targeted at particular Exception levels. The Exception level that an exception targets is either programmed by software, or is determined by the nature of the exception.

Under no circumstances do exceptions cause execution to move to a lower Exception level.

If an asynchronous exception targets a lower Exception level, the exception is not taken and remains pending. See *Asynchronous exception routing* on page D1-2364 and *Asynchronous exception masking* on page D1-2367.

Note

The construction of the architecture means that usually, it is impossible for an exception to target a lower Exception level.

The Security state can only change on taking an exception if taken from Non-secure state to EL3.

Note

Taking an exception to EL3 from any Exception level has no effect on the value of the SCR_EL3.NS bit.

On taking an exception to AArch64 state:

- The PE state is saved in the SPSR_ELx at the target Exception level. See *Saved Program Status Registers (SPSRs)* on page D1-2324.
- The preferred return address is saved in the ELR_ELx at the target Exception level. See *Exception Link Registers (ELRs)* on page D1-2328.
- All of PSTATE.{D, A, I, F} are set to 1. See *Process state, PSTATE* on page D1-2330.
- PSTATE.SSBS is set to the value of SCTLR_ELx.DSSBS.
- If ARMv8.2-UAO is implemented, PSTATE.UAO is set to 0. See *Process state, PSTATE* on page D1-2330.
- If the exception is a synchronous exception or an SError interrupt, information characterizing the reason for the exception is saved in the ESR_ELx at the target Exception level. See *Use of the ESR_EL1, ESR_EL2, and ESR_EL3* on page D1-2342.
- If ARMv8.5-MemTag is implemented:
  - PSTATE.TCO is set to 1. See *Process state, PSTATE* on page D1-2330.
  - A synchronous exception due to a Tag Check Fault is reported to as a Data Abort with a Data Fault Status Code of Synchronous Tag Check Fault and the faulting virtual address is reported in FAR_ELx. For more information see *PE handling of Tag Check Fault* on page D6-2696.
- If ARMv8.5-BTI is implemented, on taking an asynchronous exception from AArch64 to AArch64, PSTATE.BTYPE is copied to SPSR_ELx.BTYPE and then set to 0.
- If ARMv8.5-BTI is implemented, on taking certain types of synchronous exception from AArch64 to AArch64, PSTATE.BTYPE is copied to SPSR_ELx.BTYPE and then set to 0. These types of synchronous exceptions are:
  - Software Step exception.
  - PC alignment fault exception.
  - Instruction Abort exception.
  - Breakpoint exceptions or Address Matching Vector Catch exception.
  - Illegal Execution state exception.
  - Software Breakpoint exception.
  - Branch Target exception.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

On taking any other synchronous exception from AArch64 to AArch64, it is CONSTRAINED UNPREDICTABLE whether:

— SPSR_ELx.BTYPE is set to the value of PSTATE.BTYPE.

— SPSR_ELx.BTYPE is set to 0.

PSTATE.BTYPE is then set to 0.

- The stack pointer register selected is the dedicated stack pointer register for the target Exception level. See *The stack pointer registers* on page D1-2323.
- For a physical SError interrupt exception, the pending state of the physical SError is cleared when any of:

  — The SError interrupt is edge-triggered.

  — ARMv8.4-DFE is implemented.

  — If The Reliability, Availability, and Serviceability Extension (RAS Extension) is implemented, and on taking the SError interrupt, the syndrome recorded in ESR_ELx indicates an SError other than IMPLEMENTATION DEFINED or uncategorized SError interrupt syndrome.

  Otherwise, it is IMPLEMENTATION DEFINED whether the pending state of the physical SError is cleared.

  This IMPLEMENTATION DEFINED behavior might vary according to the nature of the SError interrupt.
- For a virtual SError interrupt exception, the pending state of the virtual SError, held in the HCR_EL2.VSE bit, is cleared to zero. See *Virtual interrupts* on page D1-2369.
- If ARMv8.2-IESB is implemented, when the Effective value of the SCTLR_ELx.IESB bit at the target Exception level is 1, the PE inserts an error synchronization event. See *ARM® Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for the ARMv8-A architecture profile*.
- Execution moves to the target Exception level, and starts at the address defined by the exception vector. Which exception vector is used is also an indicator of whether the exception came from a lower Exception level or the current Exception level. See *Exception vectors* on page D1-2341.
- If an Instruction Abort exception, Data Abort exception, PC alignment fault exception, or a Watchpoint exception is taken to an Exception level using AArch64, the faulting virtual address is saved in FAR_ELx. .For more information, see *Validity of FAR_ELx* on page D1-2348.
- If an Instruction Abort exception, or Data Abort exception is taken to EL2 and the fault is one connected with stage 2 translation, the faulting IPA is saved in HPFAR_EL2. For more information, see *Validity of HPFAR_EL2* on page D1-2349.

If ARMv8.5-CSEH is implemented and SCTLR_ELx.EIS is 0, though exception entry is not a context synchronization event, the indirect writes to ESR_ELx, FAR_ELx, SPSR_ELx, ELR_ELx, and HPFAR_EL2 due to exception entry are synchronized so that a direct read of the register after exception entry sees the indirectly written value caused by the exception entry.

Note

On exception entry, the memory transactions, including instruction fetches, from an exception level always use the translation resources associated with that translation regime.

The remainder of this section contains the following:

- *Preferred exception return address* on page D1-2341.
- *Exception vectors* on page D1-2341.
- *Pseudocode description of exception entry to AArch64 state* on page D1-2342.
- *Exception classes and the ESR_ELx syndrome registers* on page D1-2342.
- *Summary of register updates on faults taken to an Exception level that is using AArch64* on page D1-2348.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D1.10.1 Preferred exception return address

For an exception taken to an Exception level using AArch64, the Exception Link Register for that Exception level, ELR_ELx, holds the preferred exception return address. The preferred exception return address depends on the nature of the exception, as follows:

- For asynchronous exceptions, it is the address of the instruction following the instruction boundary at which the interrupt occurs. Therefore, it is the address of the first instruction that did not execute, or did not complete execution, as a result of taking the interrupt.
- For synchronous exceptions other than system calls, it is the address of the instruction that generates the exception.
- For exception generating instructions, it is the address of the instruction that follows the exception generating instruction.

Note

If an exception generating instruction is trapped, disabled, or is UNDEFINED because the Exception level has insufficient privilege to execute the instruction, the preferred exception return address is the address of the exception generating instruction.

When an exception is taken from an Exception level using AArch32 to an Exception level using AArch64, the top 32 bits of the modified ELR_ELx are 0.

D1.10.2 Exception vectors

When the PE takes an exception to an Exception level that is using AArch64, execution is forced to an address that is the *exception vector* for the exception. The exception vector exists in a *vector table* at the Exception level the exception is taken to.

A vector table occupies a number of word-aligned addresses in memory, starting at the *vector base address*.

Each Exception level has an associated *Vector Base Address Register* (VBAR), that defines the exception base address for the table at that Exception level.

For exceptions taken to AArch64 state, the vector table provides the following information:

- Whether the exception is one of the following:
  - Synchronous exception.
  - SError.
  - IRQ.
  - FIQ.
- Information about the Exception level that the exception came from, combined with information about the stack pointer in use, and the state of the register file.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-7 shows this.

Table D1-7 Vector offsets from vector table base address

Exception taken from	Offset for exception type			
	Synchronous	IRQ or vIRQ	FIQ or vFIQ	SError or vSError
Current Exception level with SP_EL0.	0x000[a]	0x080	0x100	0x180
Current Exception level with SP_ELx, x>0.	0x200[a]	0x280	0x300	0x380
Lower Exception level, where the implemented level immediately lower than the target level is using AArch64.[b]	0x400[a]	0x480	0x500	0x580
Lower Exception level, where the implemented level immediately lower than the target level is using AArch32.[b]	0x600[a]	0x680	0x700	0x780

a. When ARMv8.4-DFE is implemented, SCR_EL3.EASE is set to 1, and the exception is a synchronous External abort taken to EL3, the exception is routed to the offset in the SError or vSError column.

b. For exceptions taken to EL3, if EL2 is implemented, the level immediately lower than the target level is EL2 if the exception was taken from Non-secure state, but EL1 if the exception was taken from Secure EL1 or EL0.

Reset is treated as a special vector for the highest implemented Exception level. This special vector uses an IMPLEMENTATION DEFINED address that is typically set either by a hardwired configuration of the PE or by configuration input signals. The RVBAR_ELx register contains this reset vector address, where *x* is the number of the highest implemented Exception level.

D1.10.3 Pseudocode description of exception entry to AArch64 state

The `AArch64.TakeException()` pseudocode function describes the behavior when the PE takes an exception to an Exception level that is using AArch64. The `AArch64.ExceptionClass()` function determines the EC (Exception class) and IL (Instruction length) values required to report the exception, and `AArch64.ReportException()` reports the exception.

The pseudocode functions `AArch64.TakeException()`, `AArch64.ExceptionClass()`, and `AArch64.ReportException()` are described in Chapter J1 *Armv8 Pseudocode*.

D1.10.4 Exception classes and the ESR_ELx syndrome registers

If the exception is a synchronous exception or an SError interrupt, information characterizing the reason for the exception is saved in the ESR_ELx at the Exception level the exception is taken to. The information saved is determined at the time the exception is taken, and is not changed as a result of the explicit synchronization that takes place at the start of taking the exception. See *Synchronization requirements for AArch64 System registers* on page D13-2869. The following sections give more information:

- *Use of the ESR_EL1, ESR_EL2, and ESR_EL3*.
- *The EC used to report an exception routed to EL2 because HCR_EL2.TGE is 1* on page D1-2347.

Use of the ESR_EL1, ESR_EL2, and ESR_EL3

An ESR_ELx holds the syndrome information for an exception that is taken to AArch64 state.

Note

This use of a syndrome is also the reporting model used for exceptions taken to Hyp mode when they are taken to EL2 using AArch32.

Figure D1-2 on page D1-2343 shows the general format of the ESR_ELx registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Figure D1-2 Overall format of the ESR_ELx registers

The ESR_ELx fields are:

EC, bits[31:26] The Exception class field, that indicates the cause of the exception.

IL, bit[25] The Instruction length bit, for synchronous exceptions, that indicates whether a trapped instruction was a 16-bit or a 32-bit instruction.

ISS, bits[24:0] The Instruction specific syndrome field. Architecturally, this field can be defined independently for each defined Exception class. However, in practice, some ISS encodings are used for more than one Exception class.

ESR_EL1, Exception Syndrome Register (EL1) on page D13-2968, *ESR_EL2, Exception Syndrome Register (EL2)* on page D13-3008 and *ESR_EL3, Exception Syndrome Register (EL3)* on page D13-3048 describe the registers in full, including:

- Listing the valid EC field values.
- Describing the ISS for each Exception class.
- Giving a full description of the use of the IL field.

Table D1-8 shows the encoding of the ESR_ELx.EC field, the Exception class field. For each EC value, the table references a subsection of the ESR_ELx register definition that describes the ISS format, with links to descriptions of possible causes of the exception, for example the configuration required to enable a trap.

Table D1-8 ESR_ELx.EC field encoding

EC	Exception class	From, state		To, Exception level			ISS encoding description
		AArch32[a]	AArch64	EL1	EL2	EL3	
`000000`	Unknown reason	Yes	Yes	Yes	Yes	Yes	*ISS encoding for exceptions with an unknown reason* on page D13-2972
`000001`	Trapped `WFI` or `WFE` instruction execution[b]	Yes	Yes	Yes	Yes	Yes	*ISS encoding for an exception from a WFI or WFE instruction* on page D13-2973
`000011`	Trapped `MCR` or `MRC` access with (`coproc==0b1111`)[b] that is not reported using EC `0b000000`	Yes	No	Yes	Yes	Yes[c]	*ISS encoding for an exception from an MCR or MRC access* on page D13-2975
`000100`	Trapped `MCRR` or `MRRC` access with (`coproc==0b1111`)[b] that is not reported using EC `0b000000`	Yes	No	Yes	Yes	Yes[d]	*ISS encoding for an exception from an MCRR or MRRC access* on page D13-2978
`000101`	Trapped `MCR` or `MRC` access with (`coproc==0b1110`)[b]	Yes	No	Yes	Yes	Yes	*ISS encoding for an exception from an MCR or MRC access* on page D13-2975
`000110`	Trapped `LDC` or `STC` access[b]	Yes	No	Yes	Yes	Yes	*ISS encoding for an exception from an LDC or STC instruction* on page D13-2980

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-8 ESR_ELx.EC field encoding (continued)

EC	Exception class	From, state		To, Exception level			ISS encoding description
		AArch32[a]	AArch64	EL1	EL2	EL3	
000111	Access to SVE, Advanced SIMD or floating-point functionality trapped by CPACR_EL1.FPEN or CPTR_ELx.TFP control[e]	Yes	Yes	Yes	Yes	Yes	*ISS encoding for an exception from an access to SVE, Advanced SIMD or floating-point functionality, resulting from CPACR_EL1.FPEN, CPTR_EL2.FPEN or CPTR_ELx.TFP* on page D13-2982
001000	Trapped VMRS access, from ID group traps, that is not reported using EC 0b000111[f]	Yes	No	No	Yes	No	*ISS encoding for an exception from an MCR or MRC access* on page D13-2975
001001	Trapped access to an ARMv8.3-PAuth instruction	No	Yes	No	Yes	Yes	*ISS encoding for an exception from a Pointer Authentication instruction when HCR_EL2.API == 0 || SCR_EL3.API == 0* on page D13-3004
001100	Trapped MRRC access with (coproc==0b1110)[b]	Yes	No	Yes	Yes	Yes	*ISS encoding for an exception from an MCRR or MRRC access* on page D13-2978
001110	Illegal Execution state	Yes	Yes	Yes	Yes	Yes	*ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault* on page D13-2984
010001	SVC instruction execution in AArch32 state	Yes	No	Yes	Yes[g]	No	*ISS encoding for an exception from HVC or SVC instruction execution* on page D13-2984
010010	HVC instruction execution in AArch32 state, when HVC is not disabled	Yes	No	No	Yes	No	
010011	SMC instruction execution in AArch32 state, when SMC is not disabled	Yes	No	No	Yes[h]	Yes	*ISS encoding for an exception from SMC instruction execution in AArch32 state* on page D13-2985
010101	SVC instruction execution in AArch64 state	No	Yes	Yes	Yes	Yes	*ISS encoding for an exception from HVC or SVC instruction execution* on page D13-2984
010110	HVC instruction execution in AArch64 state, when HVC is not disabled	No	Yes	No	Yes	Yes	
010111	SMC instruction execution in AArch64 state, when SMC is not disabled	No	Yes	No	Yes[h]	Yes	*ISS encoding for an exception from SMC instruction execution in AArch64 state* on page D13-2986
011000	Trapped MSR, MRS, or System instruction execution, that is not reported using EC 0x00, 0x01, or 0x07 When ARMv8.4-IDST is implemented, trapped ID registers	No	Yes	Yes	Yes	Yes	*ISS encoding for an exception from MSR, MRS, or System instruction execution in AArch64 state* on page D13-2986

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-8 ESR_ELx.EC field encoding (continued)

EC	Exception class	From, state		To, Exception level			ISS encoding description
		AArch32[a]	AArch64	EL1	EL2	EL3	
011001	Trapped access to SVE functionality, that is not reported using EC 0b000000[i]	No	Yes	Yes	Yes	Yes	*ISS encoding for an exception from an access to SVE, Advanced SIMD or floating-point functionality, resulting from CPACR_EL1.FPEN, CPTR_EL2.FPEN or CPTR_ELx.TFP* on page D13-2982
011010	Trapped ERET, ERETAA or ERETAB instruction execution[j]	No	Yes	No	Yes	No	*ISS encoding for an exception from ERET, ERETAA or ERETAB instruction* on page D13-3003
011100	Exception from a pointer authentication instruction authentication failure	No	Yes	Yes	Yes	Yes	*ISS encoding for an exception from a Pointer Authentication instruction authentication failure* on page D13-3004
011111	IMPLEMENTATION DEFINED exception taken to EL3	Yes	Yes	No	No	Yes	*ISS encoding for an IMPLEMENTATION DEFINED exception to EL3* on page D13-2989
100000	Instruction Abort from a lower Exception level[k]	Yes	Yes	Yes	Yes	Yes	*ISS encoding for an exception from an Instruction Abort* on page D13-2990
100001	Instruction Abort taken without a change in Exception level[k]	Yes	Yes	Yes	Yes	Yes	
100010	PC alignment fault	Yes	Yes	Yes	Yes	Yes	*ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault* on page D13-2984
100100	Data Abort from a lower Exception level, excluding Data Aborts taken to EL2 as a result of accesses generated associated with VNCR_EL2 as part of nested virtualization support[l]	Yes	Yes	Yes	Yes	Yes	*ISS encoding for an exception from a Data Abort* on page D13-3072
100101	Data Abort taken without a change in Exception level, or Data Aborts taken to EL2 as a result of accesses generated associated with VNCR_EL2 as part of nested virtualization support[l]	Yes	Yes	Yes	Yes	Yes	
100110	SP alignment fault	No	Yes	Yes	Yes	Yes	*ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault* on page D13-2984

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-8 ESR_ELx.EC field encoding (continued)

EC	Exception class	From, state		To, Exception level			ISS encoding description
		AArch32[a]	AArch64	EL1	EL2	EL3	
101000	Trapped floating-point exception taken from AArch32 state	Yes	No	Yes	Yes	No	*ISS encoding for an exception from a trapped floating-point exception* on page D13-2997
101100	Trapped floating-point exception taken from AArch64 state	No	Yes	Yes	Yes	Yes	
101111	SError interrupt	Yes	Yes	Yes	Yes	Yes	*ISS encoding for an SError interrupt* on page D13-2999
110000	Breakpoint exception from a lower Exception level	Yes	Yes	Yes	Yes[m]	No	*ISS encoding for an exception from a Breakpoint or Vector Catch debug exception* on page D13-3000
110001	Breakpoint exception taken without a change in Exception level	Yes	Yes	Yes	Yes[m]	No	
110010	Software Step exception from a lower Exception level	Yes	Yes	Yes	Yes[m]	No	*ISS encoding for an exception from a Software Step exception* on page D13-3001
110011	Software Step exception taken without a change in Exception level	Yes	Yes	Yes	Yes[m]	No	
110100	Watchpoint exception from a lower Exception level, excluding watchpoint exceptions taken to EL2 as a result of accesses generated associated with VNCR_EL2 as part of nested virtualization support	Yes	Yes	Yes	Yes[m]	No	*ISS encoding for an exception from a Watchpoint exception* on page D13-3001
110101	Watchpoint exception taken without a change in Exception level, or Watchpoint exceptions taken to EL2 as a result of accesses generated associated with the VNCR_EL2 as part of nested virtualization support	Yes	Yes	Yes	Yes[m]	No	
111000	BKPT instruction execution in AArch32 state	Yes	No	Yes	Yes[m]	No	*ISS encoding for an exception from a Breakpoint or Vector Catch debug exception* on page D13-3000
111010	Vector Catch exception from AArch32 state	Yes	No	No	Yes[m]	No	
111100	BRK instruction execution in AArch64 state	No	Yes	Yes	Yes[m]	Yes[n]	

a. See also *Reporting AArch32 synchronous exceptions taken to an Exception level using AArch64* on page D1-2347.
b. Exceptions caused by configurable traps, enables, or disables.
c. See *Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32* on page D1-2391.
d. Only for MCRR or MRRC accesses to the PMCCNTR_EL0 or PMCCNTR.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

e. Excludes exceptions that are generated because the value of HCR_EL2.TGE is 1, see *The EC used to report an exception routed to EL2 because HCR_EL2.TGE is 1*.
f. Applies only to traps of accesses to MVFR0, MVFR1, MVFR2, or FPSID. Includes traps of `VMRS` accesses. Because the MVFR*n* registers are read-only and a `VMSR` access to the FPSID is ignored and not trapped, there are no `MCR` or `VMSR` accesses that can be trapped with this EC value.
g. Only as a result of HCR_EL2.TGE.
h. Only as a result of HCR_EL2.TSC.
i. Only if The Scalable Vector Extension (SVE) is implemented. Otherwise the EC value is reserved.
j. Only if ARMv8.3-NV is implemented and HCR_EL2.NV is 1.
k. Used for MMU faults generated by instruction accesses, and for synchronous External aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.
l. Used for MMU faults generated by data accesses, Alignment faults other than SP alignment faults and PC alignment faults, and for synchronous External aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.
m. Only as a result of HCR_EL2.TGE ==1 or MDCR_EL2.TDE ==1.
n. Only if the `BRK` instruction is executed in EL3. This is the only debug exception that can be taken to EL3 when EL3 is using AArch64.

Reserved EC values

For EC values not shown in Table D1-8 on page D1-2343:

- Unused EC values in the range `0b000000-0b101100` (`0x00-0x2C`) are reserved by Arm for future use for synchronous exceptions.
- Unused EC values in the range `0b101101-0b111111` (`0x2D-0x3F`) are reserved by Arm for future use, and might be used for synchronous or asynchronous exceptions.

The EC used to report an exception routed to EL2 because HCR_EL2.TGE is 1

When an exception is taken from EL0 to EL2 because the value of HCR_EL2.TGE is 1, the exception is reported in ESR_EL2. The EC value and corresponding ISS encoding used to report the exception in ESR_EL2 depend on how an exception of the same class would be reported in ESR_EL1 when the value of HCR_EL2.{TGE, RW} is {0, 1}:

- If the exception would have been reported in ESR_EL1 using the EC value `0x07` then it is reported in ESR_EL2 using the EC value `0x00` and corresponding ISS encoding.
- Otherwise, the exception is reported in ESR_EL2 using the EC value and ISS encoding that would have been used to report the exception ESR_EL1.

Reporting AArch32 synchronous exceptions taken to an Exception level using AArch64

Although possible exception causes are generally similar for AArch32 state and AArch64 state, AArch32 state has additional exception taxonomy that is not present in AArch64 state. The following sections described named AArch32 exceptions that can, in some contexts, be taken to an Exception level that is using AArch64:

- *Undefined Instruction exception* on page G1-5738.
- *Supervisor Call (SVC) exception* on page G1-5742.
- *Secure Monitor Call (SMC) exception* on page G1-5743.
- *Hypervisor Call (HVC) exception* on page G1-5744.
- *Prefetch Abort exception* on page G1-5745.
- *Data Abort exception* on page G1-5749.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When EL2 is using AArch64 and the value of HCR_EL2.TGE is 1, these exceptions are routed to EL2, and reported in the ESR_EL2. Table D1-9 shows how they are reported.

Table D1-9 Syndrome reporting in ESR_EL2 of HCR_EL2 routing of exceptions

AArch32 exception	Pseudocode	EC value used to report exception in ESR_ELx
Undefined Instruction	`AArch32.UndefinedFault()`	`0x00`, Exception for an unknown reason
Supervisor Call	`AArch32.CallSupervisor()`	`0x11`, Exception from `SVC` instruction executed in AArch32 state
Secure Monitor Call	See `SMC` on page F5-4694[a]	`0x13`, Exception from `SMC` instruction executed in AArch32 state
Hypervisor Call	`AArch32.CallHypervisor()`	`0x12`, Exception from `HVC` instruction executed in AArch32 state
Prefetch Abort	`AArch32.Abort()`	`0x20`, Exception from an Instruction abort at a lower Exception level
Data Abort	`AArch32.Abort()`	`0x24`, Exception from a Data abort at a lower Exception level

a. The pseudocode in *Operation for all encodings* on page F5-4695 identifies when the execution of an `SMC` instruction in AArch32 state generates an exception that is taken to EL3 using AArch64.

D1.10.5 Summary of register updates on faults taken to an Exception level that is using AArch64

For all exceptions taken to an Exception level using AArch64 that are not listed in *Validity of FAR_ELx*, the FAR_ELx for the Exception level the exception is taken to is UNKNOWN.

For all exceptions taken to EL2 using AArch64 that are not listed in *Validity of HPFAR_EL2* on page D1-2349, the HPFAR_EL2 is UNKNOWN.

The following sections give more information:

- *Validity of FAR_ELx*.
- *Validity of HPFAR_EL2* on page D1-2349.

Validity of FAR_ELx

The faulting virtual address is saved in FAR_ELx for the Exception level the exception is taken to if an exception is one of:

- An Instruction Abort exception.
- A Data Abort exception.
- A PC alignment fault exception.
- A Watchpoint exception.

The architecture permits that the FAR_ELx is UNKNOWN for synchronous External aborts other than synchronous External aborts on translation table walks. In this case, the ISS.FnV bit returned in ESR_ELx indicates whether FAR_ELx is valid.

If an exception is taken from an Exception level using AArch32 into an Exception level using AArch64, and that exception writes the FAR_ELx at the Exception level the exception is taken to, the most significant 32 bits of FAR_ELx are all zero, unless both of the following apply, in which case the most significant 32 bits of FAR_ELx are `0x00000001`:

- The faulting address was generated by a load or store that sequentially incremented from address `0xFFFFFFFF`. Such a load or store instruction is CONSTRAINED UNPREDICTABLE, see *Out of range VA* on page K1-7840.
- The implementation treats such incrementing as setting bit[32] of the virtual address to 1.

The FAR_ELx for an Exception level is made UNKNOWN as a result of an exception return from that Exception level.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Validity of HPFAR_EL2

The faulting IPA is saved in HPFAR_EL2 if the exception is an Instruction Abort or Data Abort taken to EL2 and the fault is one of:

- A Translation or Access Flag fault on a stage 2 translation.
- A stage 2 Address Size fault.
- A fault on the stage 2 translation of an address accessed in a stage 1 translation table walk.

HPFAR_EL2 is made UNKNOWN as a result of an exception return from EL2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D1.11 Exception return

In the Armv8-A architecture, an exception return is always to the same Exception level or a lower Exception level. An exception return is used for:

- A return to a previously executing thread.
- Entry to a new execution thread. For example:
  - The initialization of a hypervisor by a Secure monitor.
  - The initialization of an operating system by a hypervisor.
  - Application entry from an operating system or hypervisor.

If ARMv8.5-CSEH is not implemented, or if ARMv8.5-CSEH is implemented and the SCTLR_ELx.EOS field is set, exception return from ELx is a context synchronization event.

An exception return requires the simultaneous restoration of the PC and PSTATE to values that are consistent with the desired state of execution on returning from the exception. The indirect write of the PSTATE information and the PC is synchronized even if the return is not a context synchronization event.

In AArch64 state, an ERET instruction causes an exception return, see *ERET* on page C6-923.

If ARMv8.2-IESB is implemented, when the SCTLR_ELx.IESB bit at the Exception level the exception is returning from is 1, the PE inserts an error synchronization event before the ERET instruction. See *ARM® Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for the ARMv8-A architecture profile*.

On executing an ERET instruction at ELx:

- The PC is restored with the value held in ELR_ELx.
- PSTATE is restored by using the contents of SPSR_ELx.

ELR_ELx and SPSR_ELx are the ELR_ELx and SPSR_ELx at the Exception level the exception is returning from. The exception return makes this ELR_ELx and SPSR_ELx UNKNOWN.

See *Address tagging in AArch64 state* on page D5-2534 for details of how tagged addresses are handled in an Exception return from an Exception level using AArch64 to an Exception level using AArch64.

Note

When returning from an Exception level using AArch64 to an Exception level using AArch32, the top 32 bits of the ELR_ELx are ignored.

An ERET instruction also:

- Sets the Event Register for the PE executing the ERET instruction. See *Mechanisms for entering a low-power state* on page D1-2397.
- Resets the local Exclusives monitor for the PE executing the ERET instruction. This removes the risk of errors that might be caused when a path to an exception return fails to include a CLREX instruction.

  Note

  This behavior prevents self-hosted debug from software stepping through a Load-Exclusive/Store-Exclusive pair. However, when self-hosted debug is using software step, it is highly probable that the Exclusives monitor state would be lost anyway, for other reasons. *Stepping code that uses Exclusives monitors* on page D2-2483 describes this.

  It is IMPLEMENTATION DEFINED whether the resetting of the local Exclusives monitor also resets the global Exclusives monitor.

The ERET instruction is UNDEFINED in EL0.

When returning from an Exception level using AArch64 to an Exception level using AArch32, the AArch32 context is restored. The Armv8-A architecture defines the relationship between AArch64 state and AArch32 state, for:

- General-purpose registers.
- Special-purpose registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- System registers.

In an implementation that includes EL3, the Security state can only change on returning from an exception if the return is from EL3 to a lower Exception level.

The following sections give more information:

- *Exception return and PC alignment*.
- *Illegal return events from AArch64 state*.
- *Legal returns that set PSTATE.IL to 1* on page D1-2353.
- *The Illegal Execution state exception* on page D1-2353.
- *Pseudocode description of exception return* on page D1-2353.

D1.11.1 Exception return and PC alignment

When SPSR_ELx.M[4] == 0, indicating an Exception return to AArch64 state, the value of ELR_ELx is transferred to the PC. If this value is misaligned, subsequent execution results in a PC alignment fault exception.

When SPSR_ELx.M[4] == 1, indicating an Exception return to AArch32 state, the value of ELR_ELx is transferred to the PC except that, for a legal exception return:

- If SPSR_ELx.T is 0, ELR_ELx[1:0] are treated as being 0 for restoring the PC.
- If SPSR_ELx.T is 1, ELR_ELx[0] is treated as being 0 for restoring the PC.

This means that a PC alignment fault exception cannot occur following a legal exception return from AArch64 state to AArch32 state. However, where the Exception return with SPSR_ELx.M[4] == 1 is an illegal exception return then it is IMPLEMENTATION DEFINED whether a misaligned value in ELR_ELx is aligned when it is restored to the PC.

Note

In an implementation that forces the alignment of the PC value restored from SPSR_ELx on an illegal exception return with SPSR_ELx.M[4] == 1, if SPSR_ELx.T == 1 the restored PC value might give rise to a PC alignment fault exception, because the PE remains in AArch64 state and only ELR_ELx[0] is treated as being 0 for restoring the PC.

For more information about the illegal exception return cases see *Illegal return events from AArch64 state*.

D1.11.2 Illegal return events from AArch64 state

In this section:

Return
In AArch64 state, refers to any of:
- Execution of an `ERET` instruction.
- Execution of a `DRPS` instruction in Debug state.
- Exit from Debug state.

Saved process state value
In AArch64 state, refers to any of:
- The value held in the SPSR_ELx for an `ERET` instruction.
- The value held in the SPSR_ELx for a `DRPS` instruction executed in Debug state.
- The value held in the DSPSR_EL0 for a Debug state exit.

Link address
In AArch64 state, refers to any of:
- The address held in ELR_ELx for an ERET instruction.
- The address held in DLR_EL0 for a Debug state exit.

Configured from reset
Indicates the state determined on powerup or reset by a configuration input signal, or by another IMPLEMENTATION DEFINED mechanism.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The Armv8 architecture has a generic mechanism for handling returns to a mode or state that is illegal. In AArch64 state, this can occur as the result of any of the following situations:

- A return where the Exception level being returned to is higher than the current Exception level.
- A return where the Exception level being returned to is not implemented. For example a return to EL2 when EL2 is not implemented.
- A return to EL2 when EL3 is implemented and the value of the SCR_EL3.NS bit is 0 if ARMv8.4-SecEL2 is not implemented.
- A return to EL1 when EL2 is implemented and the value of the HCR_EL2.TGE bit is 1.
- A return where the value of the saved process state M[4] bit is 0, indicating a return to AArch64 state, and one of the following is true:
  - The M[1] bit is 1.
  - The M[3:0] bits are `0b0001`.
  - The Exception level being returned to is using AArch32 state, as programmed by the SCR_EL3.RW or HCR_EL2.RW bits, or as configured from reset.
- A return where the value of the saved process state M[4] bit is 1, indicating a return to AArch32 state, and one of the following is true:
  - The M field value is not a valid AArch32 state PE mode. Table D1-4 on page D1-2328 shows the valid M[3:0] values for AArch32 state PE modes. This includes the case where M[3:0] is `0b0000`, indicating User mode, and EL0 does not support AArch32 state.
  - The Exception level being returned to is using AArch64 state as determined by the SCR_EL3.RW or HCR_EL2.RW field or the configuration from reset. This includes the case where the Exception level being returned to does not support AArch32 state.

  Note

  This means that, in an implementation that supports only AArch64 state, any attempt to return to AArch32 state is an illegal exception return.

- A Debug state exit from EL0 using AArch64 state, to EL0 using AArch32 state.

In these cases:

- PSTATE.IL is set to 1, to indicate an illegal return.
- PSTATE.{EL, nRW, SP} are unchanged. This means the Exception level, Execution state, and stack pointer selection do not change as a result of the return.
- The following PSTATE bits are restored from the saved process state value:
  - The N, Z, C, V Condition flags.
  - The D, A, I, F exception mask bits.
- If the illegal return is an illegal exception return, the PSTATE.SS bit is handled as normal for a return. That is, the SS bit is handled in the same way as an exception return that is not an illegal exception return. See *Software Step exceptions* on page D2-2472.

  In all these cases the PSTATE.SS bit is handled as it would be for a normal return, as described in *Entering the active-not-pending state* on page D2-2474 and *Exiting Debug state* on page H2-6963. DRPS never sets the SS bit. This is indicated in *Entering the active-not-pending state* on page D2-2474.
- If the illegal return is not a `DRPS` instruction executed in Debug state, the PC is restored from the link address. However, if the value of the M[4] bit of the saved process state is 1, indicating a return to AArch32 state, then:
  - Bits[31:2] of the PC are restored from the link address.
  - Bits[63:32, 1:0] of the PC are UNKNOWN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When the value of the PSTATE.IL bit is 1, any attempt to execute any instruction results in an Illegal Execution state exception. See *The Illegal Execution state exception*.

All aspects of the illegal return, other than the effects described in this section, occur as they do for a legal return.

D1.11.3 Legal returns that set PSTATE.IL to 1

In this section, *return*, *saved process state value*, and *link address* have the same meaning as defined in *Illegal return events from AArch64 state* on page D1-2351.

If the value of the IL bit in the saved process state is 1, then it is copied to PSTATE by a return, meaning that PSTATE.IL is set to 1. In this case, if the return is not an illegal return, and targets AArch32 state, then the PSTATE.{IT, T} bits are either:

- Set to 0.
- Copied from the saved process state value.

The choice between these two options is determined by an implementation, and might vary dynamically within the implementation. Correspondingly software must regard the value as being an UNKNOWN choice between the two values.

The PSTATE.{IT, T} bits are only valid in AArch32 state, see *Process state, PSTATE* on page G1-5695.

When the PSTATE.IL bit is 1, any attempt to execute any instruction results in an Illegal Execution state exception. See *The Illegal Execution state exception*.

D1.11.4 The Illegal Execution state exception

When the value of the PSTATE.IL bit is 1, any attempt to execute any instruction results in an Illegal Execution state exception. In AArch64 state, the PSTATE.IL bit can be to set to 1 by any of:

- An illegal return, as described in *Illegal return events from AArch64 state* on page D1-2351.
- A legal return that sets PSTATE.IL to 1, as described in *Legal returns that set PSTATE.IL to 1*.

If an Illegal Execution state exception is generated at EL0, it is taken to EL1. If the exception occurs when EL2 is implemented and enabled in the current Security state, and HCR_EL2.TGE == 1, then it is taken to EL2. If an Illegal Execution state exception is generated at any other Exception level, the Exception level is unchanged.

An Illegal Execution state exception sets ESR_ELx.EC for the target Exception level to the value of `0x0E`.

On taking any exception to an Exception level that is using AArch64 state:

1. The value of the PSTATE.IL bit is copied into the SPSR_ELx.IL bit for the Exception level to which the exception is taken.
2. The PSTATE.IL bit is cleared to 0.

Note

This means that it is not possible for software to observe the value of PSTATE.IL.

For the priority of this exception class, see *Synchronous exception prioritization for exceptions taken to AArch64 state* on page D1-2355.

D1.11.5 Pseudocode description of exception return

The `AArch64.ExceptionReturn()` pseudocode function transfers the return address to the PC, and restores PSTATE to its saved value by calling `SetPSTATEFromPSR()`.

The `IllegalExceptionReturn()` function checks for an Illegal Execution state exception.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D1.12 Synchronous exception types, routing and priorities

Synchronous exceptions are:

- Any exception generated by attempting to execute an instruction that is UNDEFINED, including:
  - Attempts to execute instructions at an inappropriate Exception level.
  - Attempts to execute instructions when they are disabled.
  - Attempts to execute instruction bit patterns that have not been allocated.
- Illegal Execution state exceptions. These are caused by attempts to execute an instruction when the value of PSTATE.IL is 1, see *Illegal return events from AArch64 state* on page D1-2351.
- Exceptions caused by the use of a misaligned SP.
- Exceptions caused by attempting to execute an instruction with a misaligned PC.
- Exceptions caused by the exception-generating instructions `SVC`, `HVC`, or `SMC`.
- Traps on attempts to execute instructions that the System registers define as instructions that are trapped to a higher Exception level. See *Configurable instruction enables and disables, and trap controls* on page D1-2373.
- Instruction Aborts generated by the memory address translation system that are associated with attempts to execute instructions from areas of memory that generate faults.
- Data Aborts generated by the memory address translation system that are associated with attempts to read or write memory that generate faults.
- Data Aborts caused by a misaligned address.
- Data Aborts caused by a Tag Check Fault if ARMv8.5-MemTag is implemented, for more information see Chapter D6 *Armv8.5 Memory Tagging Extension*.
- All of the debug exceptions:
  - Breakpoint Instruction exceptions.
  - Breakpoint exceptions.
  - Watchpoint exceptions.
  - Vector Catch exceptions.
  - Software Step exceptions.
- In an implementation that supports the trapping of floating-point exceptions, exceptions caused by trapped IEEE floating-point exceptions, see *Floating-point exceptions and exception traps* on page D1-2360.
- In some implementations, External aborts. External aborts are failed memory accesses, and include accesses to those parts of the memory system that occur during the address translation. The Armv8 architecture permits, but does not require, implementations to treat such exceptions synchronously. See *External aborts* on page D4-2524.

This remainder of this section contains the following:

- *Routing exceptions from EL0 to EL2*.
- *Routing debug exceptions to EL2* on page D1-2355.
- *Routing synchronous External aborts* on page D1-2355
- *Synchronous exception prioritization for exceptions taken to AArch64 state* on page D1-2355.
- *Effect of Data Aborts and Watchpoints* on page D1-2359.
- *Floating-point exceptions and exception traps* on page D1-2360.

D1.12.1 Routing exceptions from EL0 to EL2

When EL2 is enabled in the current Security state and the value of HCR_EL2.TGE is 1, any exception taken from EL0 that would otherwise be taken to EL1 is, instead, routed to EL2. This means that an application can execute at EL0 without using any functionality at EL1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

- When EL2 is using AArch64 state, the HCR_EL2.TGE control applies regardless of whether EL0 is using AArch32 state or AArch64 state.
- Implementations typically use the following Exception level and software hierarchy:

  EL2 Hypervisor.

  EL1 Operating system.

  EL0 Application.

  In such an implementation, setting HCR_EL2.TGE to 1 means that an application can run at EL0 under the direct control of a hypervisor executing at EL2, with no operating system involvement.

D1.12.2 Routing debug exceptions to EL2

When EL2 is enabled in the current Security state and the value of MDCR_EL2.TDE is 1, debug exceptions are routed to EL2. For more information see *Routing debug exceptions* on page D2-2429.

When the value of MDCR_EL2.TDE is 1, each of the MDCR_EL2.{TDRA, TDOSA, TDA} bits is treated as 1 for all purposes other than direct reads of the MDCR_EL2.

D1.12.3 Routing synchronous External aborts

When the value of SCR_EL3.EA is 1, synchronous external aborts are taken to EL3.

When the RAS Extension is implemented, EL2 is enabled in the current Security state, and the value of HCR_EL2.TEA is 1, synchronous external aborts from EL0 and EL1 that are not routed to EL3 are routed to EL2.

D1.12.4 Synchronous exception prioritization for exceptions taken to AArch64 state

In principle, any single instruction can generate a number of different synchronous exceptions, between the fetching of the instruction, its decode, and eventual execution. For exceptions taken to an Exception level that is using AArch64, these are prioritized as follows, where 1 is the highest priority.

Note

The priority numbering in this list correlates with the equivalent AArch32 state list in *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-5707 and the list in *Debug state entry and debug event prioritization* on page H2-6929.

1-3 These priority numbers represent debug events.

4 Software Step exceptions. See *Software Step exceptions* on page D2-2472.

5 This priority number represents debug events.

6 PC alignment fault exceptions. See *PC alignment checking* on page D1-2333.

7 Instruction Abort exceptions. See *AArch64 state prioritization of synchronous aborts from a single stage of address translation* on page D5-2659.

8 Breakpoint exceptions or Address Matching Vector Catch exceptions. See:

- *Breakpoint exceptions* on page D2-2439.
- *Vector Catch exceptions* on page D2-2471.

Vector Catch exceptions are only taken from AArch32 state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

An Exception Trapping Vector Catch exception is generated on exception entry for an exception that has been prioritized as described in *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-5707. This means that it is outside the scope of the description of this section.

9 Illegal Execution state exceptions. See *Illegal return events from AArch64 state* on page D1-2351.

10 Software Breakpoint exceptions caused by the execution of a Breakpoint instruction:

- For exceptions taken from AArch64 state, `BRK`.
- For exceptions taken from AArch32 state, `BKPT`.

11 Branch Target exceptions generated for an instruction that lies within a guarded page if PSTATE.BTYPE field is not 0b00 and if the instruction is not any of:

- A BTI instruction that is compatible with the PSTATE.BTYPE field.
- A `PACIASP` or `PACIBSP` instruction, and the PSTATE.BTYPE is consistent with implicit BTI behavior of these instructions.
- A Breakpoint Instruction exception.
- A Halt Instruction debug event.

A Branch Target exception is taken to:

- EL1 when executing at EL0 and HCR_EL2.E2H==0.
- EL2 when executing at EL0 and HCR_EL2.E2H==1.
- ELx when executing at ELx.

For more information on Branch Target exception see *ISS encoding for an exception from Branch Target Identification instruction* on page D13-3003.

12 Exceptions taken from EL1 to EL2 because of one of the following configuration settings:

- HSTR_EL2.T*n*.
- HCR_EL2.TIDCP.
- If ARMv8.3-NV is implemented, HCR_EL2.NV or HCR_EL2.NV1.

Note

If ARMv8.4-NV is implemented and HCR_EL2.{NV, NV1, NV2} are set such that register accesses to EL1 are transformed into memory accesses, then HCR_EL2{NV, NV1} do not generate exceptions to EL2.

13 Exceptions that occur as a result of attempting to execute an instruction that is UNDEFINED for one or more of the following reasons:

- Attempting to execute an unallocated instruction encoding, including an encoding for an instruction that is not implemented in the PE implementation.
- Attempting to execute an instruction that is defined never to be accessible at the current Exception level regardless of any enables or traps.
- Debug state execution of an instruction encoding that is unallocated in Debug state.
- Non-debug state execution of an instruction encoding that is unallocated in Non-debug state.
- Execution of an `HVC` instruction, when `HVC` instructions are disabled by SCR_EL3.HCE or HCR_EL2.HCD.
- Execution of an `MSR` or `MRS` instruction to SP_EL0 when the value of SPSel is 0.
- Execution of an `HLT` instruction when `HLT` instructions are disabled by EDSCR.HDE or halting is prohibited.
- In Debug state:
  - Execution of a `DCPS1` instruction in Non-secure EL0 when HCR_EL2.TGE is 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— Execution of a DCPS2 instruction in EL1 or EL0 when EL2 is disabled in the current Security state or is not implemented.

— Execution of a DCPS3 instruction when EDSCR.SDD is 1 or when EL3 is not implemented.

— When the value of EDSCR.SDD is 1, execution in EL2, EL1, or EL0 of an instruction that is configured by EL3 control registers to trap to EL3. It is IMPLEMENTATION DEFINED whether this type of exception is prioritized at this level or has the priority of the original trap exception.

- When executing in AArch32 state, execution of an instruction that is UNDEFINED as a result of any of:

  — Being in an IT block when SCTLR_EL1.ITD is 1.

  — Executing a SETEND instruction executed SCTLR_EL1.SED.

  — Executing a CP15DMB, CP15DSB, or CP15ISB barrier instruction when SCTLR_EL1.CP15BEN is 0.

  Note

  These are the controls for exceptions taken to AArch64 state. For exceptions taken to AArch32 state the equivalent controls are SCTLR.{ITD, SED, CP15BEN}, with additional controls HSCTLR.{ITD, SED, CP15BEN}.

  See *Disabling or enabling EL0 use of AArch32 deprecated functionality* on page D1-2377

- When executing in AArch32 state, execution of an instruction that is UNDEFINED because at least one of FPCR.{Stride, Len} is nonzero, when programming these bits to nonzero values is supported. See *Floating-point exceptions and exception traps* on page G1-5776.

  Note

  — This case applies only when EL0 is using AArch32 and EL1 is using AArch64. The exception generated by the attempted execution at EL0 of the UNDEFINED instruction is taken to EL1 using AArch64.

  — When EL1 is using AArch32, the corresponding controls are FPSCR.{Stride, Len}, and any exception generated by the attempted execution at EL0 or EL1 of an instruction that is UNDEFINED because of a nonzero {Stride, Len} value is taken to EL1 using AArch32.

14 Exceptions taken to EL1, or taken to EL2 because the value of HCR_EL2.TGE is 1, that are generated because of configurable access to instructions, and that are not covered by any of priorities 4-13.

Note

When EL2 is using AArch32, the equivalent control for routing exceptions to EL2 is HCR.TGE.

15 Exceptions taken from EL0 to EL2 because of one of the following configuration settings:

- HSTR_EL2.T*n*.
- HCR_EL2.TIDCP.

Note

These are the controls for exceptions taken to AArch64 state. For exceptions taken to AArch32 state the equivalent controls are HSTR.T*n* and HCR.TIDCP.

16 Exceptions taken to EL2 because of configuration settings in CPTR_EL2.

Note

These are the controls for exceptions taken to AArch64 state. For exceptions taken to AArch32 state, the equivalent controls are in HCPTR.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

17 Exceptions taken to EL2 because of one of the following configuration settings:

- Any setting in HCR_EL2 other than the {TIDCP, NV} fields.
- Any setting in CNTHCTL_EL2.
- Any setting in MDCR_EL2.
- Any of the fine-grained traps in HAFGRTR_EL2, HDFGRTR_EL2, HDFGWTR_EL2, HFGITR_EL2, HFGRTR_EL2, HFGWTR_EL2.

Note

These are the controls for exceptions taken to AArch64 state. For exceptions taken to AArch32 state, equivalent controls are:

- Settings in HCR, other than the TIDCP bit.

  For exceptions taken to AArch32 state there is no control equivalent to HCR_EL2.NV.
- Any setting in CNTHCTL or HDCR.
- If EL1 is using AArch64 state, any of the fine-grained traps in HAFGRTR_EL2, HDFGRTR_EL2, HDFGWTR_EL2, HFGITR_EL2, HFGRTR_EL2, HFGWTR_EL2.

18 Exceptions taken to EL2 because of configurable access to instructions, and that are not covered by any of priorities 4-17.

19 Exceptions caused by the `SMC` instruction being UNDEFINED because the value of SCR_EL3.SMD is 1.

20 Exceptions caused by the execution of an Exception generating instruction not covered by priority 10:

- For exceptions taken from AArch64 state, *Branches, Exception generating, and System instructions* on page C3-202 defines these and the priority 10 instructions.
- When executing in AArch32 state, the exception-generating instructions are `SVC`, `HVC`, and `SMC`.

21 Exceptions taken to EL3 because of configuration settings in the CPTR_EL3.

Note

When in Debug state and the value of EDSCR.SDD is 1, instructions executed at EL2, EL1 or EL0 that are configured by EL3 control registers to trap to EL3 are treated as UNDEFINED and generate an exception taken to EL2 or EL1. It is IMPLEMENTATION DEFINED whether these exceptions are prioritized as an UNDEFINED instruction or have the priority of the original trap exception.

22 Exceptions taken to EL3 from Secure EL1 using AArch32, because of execution of the instructions listed in *Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32* on page D1-2391.

23 Exceptions taken to EL3 from EL0, EL1, or EL2 because of configuration settings in the MDCR_EL3.

Note

When in Debug state and the value of EDSCR.SDD is 1, instructions executed at EL2, EL1 or EL0 that are configured by EL3 control registers to trap to EL3 are treated as UNDEFINED and generate an exception taken to EL2 or EL1. It is IMPLEMENTATION DEFINED whether these exceptions are prioritized as an UNDEFINED instruction or have the priority of the original trap exception.

24 Exceptions taken to EL3 because of configurable access to instructions, and that are not covered by any of priorities 4-23.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

When in Debug state and the value of EDSCR.SDD is 1, instructions executed at EL2, EL1 or EL0 that are configured by EL3 control registers to trap to EL3 are treated as UNDEFINED and generate an exception taken to EL2 or EL1. It is IMPLEMENTATION DEFINED whether these exceptions are prioritized as an UNDEFINED instruction or have the priority of the original trap exception.

25 If ARMv8.3-FPAC is implemented, exceptions generated from a pointer authentication instruction authorization failure. See *Faulting on pointer authentication* on page D5-2539.

26 Trapped floating-point exceptions, if supported. See *Floating-point exceptions and exception traps* on page D1-2360.

27 This priority number represents debug events.

28 SP alignment faults. See *SP alignment checking* on page D1-2333.

29 Data Abort exceptions other than a Data Abort exception generated by a synchronous External abort that was not generated by a translation table walk or the update of a page table entry. That is, any Data Abort exception that is not covered by item 31. See *AArch64 state prioritization of synchronous aborts from a single stage of address translation* on page D5-2659. It is IMPLEMENTATION DEFINED whether synchronous External aborts are prioritized here or as item 31.

30 Watchpoint exceptions. See *Watchpoint exceptions* on page D2-2457.

31 Data Abort exception:

- Generated by a synchronous External abort that was not generated by a translation table walk or the update of a page table entry, see *External aborts* on page D4-2524.
- If ARMv8.5-MemTag is implemented and enabled, Tag Check Faults are prioritized as Data Abort. For more information see *PE handling of Tag Check Fault* on page D6-2696.
- It is IMPLEMENTATION DEFINED whether synchronous External aborts are prioritized here or as item 29.

For items 29-31, if an instruction results in more than one single-copy atomic memory access, the prioritization between synchronous exceptions generated on each of those different memory accesses is not defined by the architecture.

Note

Exceptions generated by a translation table walk are reported and prioritized as either an Instruction Abort exception, priority 7 in this list, or a Data Abort exception, priority 29 in this list. See also *AArch64 state prioritization of synchronous aborts from a single stage of address translation* on page D5-2659.

D1.12.5 Effect of Data Aborts and Watchpoints

If an instruction that stores to memory generates a Data Abort or Watchpoint, the value of each memory location that instruction stores to is either:

- Unchanged for any location for which one of the following applies:
  - An Alignment fault is generated.
  - An MMU fault is generated.
  - A Watchpoint exception or Watchpoint debug event is generated.
  - An External abort is generated, if that External abort is taken synchronously.

    Note

    If an External abort is taken asynchronously, using the SError interrupt, it is outside the scope of the architecture to define the effect of the store on the memory location, because it depends on the system-specific nature of the External abort. However, in general, Arm recommends that such memory locations are not updated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- UNKNOWN for any location for which no exception and no debug event is generated.

For External aborts and Watchpoint exceptions, the size of a memory location is defined as being the size for which a memory access is single-copy atomic.

Note

For the definition of a single-copy atomic access, see *Properties of single-copy atomic accesses* on page B2-120.

An External abort might signal a data corruption to the PE. For example a memory location might have been corrupted. The error that caused the External abort might have been propagated. The RAS Extension provides mechanisms for software to determine the extent of the corruption and contain propagation of the error. For more information, see the *ARM® Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for the ARMv8-A architecture profile*.

For Data Aborts from load or store instructions executed in AArch64 state, if the:

Data Abort is taken synchronously

- If the load or store instruction specifies writeback of a new base address, the base address is restored to the original value on taking the exception.
- If the instruction was a load to either the base address register or the offset register, that register is restored to the original value. Any other destination registers become UNKNOWN.
- If the instruction was a load that does not load the base address register or the offset register, then the destination registers become UNKNOWN.

Data Abort is taken asynchronously, using the SError interrupt

If the instruction was a load, the destination registers of the load take an UNKNOWN value if the SError interrupt is taken at a point in the instruction stream after the load.

D1.12.6 Floating-point exceptions and exception traps

Execution of a floating-point instruction, or an Advanced SIMD instruction that performs floating-point operations, can generate an exceptional condition, called a *floating-point exception*.

Note

In AArch64 state, a floating-point instruction performs only a single floating-point operation. However, an Advanced SIMD instruction that operates on floating-point values can perform multiple floating-point operations. Therefore, this section describes the handling of a floating-point exception on an *operation*, rather than on an *instruction*.

The Armv8-A architecture supports synchronous exception generation in the event of any or all of the following floating-point exceptions:

- Input Denormal.
- Inexact.
- Underflow.
- Overflow.
- Divide by Zero.
- Invalid Operation.

Whether an implementation includes synchronous exception generation for these floating-point exceptions is IMPLEMENTATION DEFINED:

- For an implementation that does provide this capability, FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} are the control bits that enable synchronous exception generation for each of the different floating-point exceptions.
- For an implementation that does not provide this capability, the FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} bits are RAZ/WI.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

- An Input Denormal floating-point exception is generated when a single-precision or double-precision floating-point value is flushed-to-zero because the value of FPCR.FZ is 1. However, no Input Denormal exception is generated when a half-precision floating-point value is flushed-to-zero because the value of FPCR.FZ16 is 1.
- The Armv8-A architecture does not support asynchronous reporting of floating-point exceptions.

When generating synchronous exceptions for one or more floating-point exceptions is enabled, the synchronous exceptions generated by the floating-point exception traps are taken to the lowest Exception level that can handle such an exception, while adhering to the rule that an exception can never be taken to a lower Exception level.

The exception is reported in the ESR_ELx for the Exception level to which it is taken.

In an implementation that includes synchronous exception generation for floating-point exceptions in AArch64 state:

- The registers that are presented to the exception handler are consistent with the state of the PE immediately before the instruction that caused the exception. An implementation is permitted not to restore the cumulative exception bits in the event of such an exception. For more information see *Combinations of floating-point exceptions*.
- When the execution of separate operations in separate SIMD elements causes multiple floating-point exceptions, the ESR_ELx reports one exception associated with one element that the instruction uses. The architecture does not specify which element is reported, however the element that is reported is identified in the ESR_ELx.

The `AArch64.FPTrappedException()` and `FPProcessException()` pseudocode functions describe the handling of trapped floating-point exceptions generated in AArch64 state.

Combinations of floating-point exceptions

Many pseudocode functions perform *floating-point operations*, including `FixedToFP()`, `FPAdd()`, `FPCompare()`, `FPCompareEQ()`, `FPCompareGE()`, `FPCompareGT()`, `FPDiv()`, `FPMax()`, `FPMin()`, `FPMul()`, `FPMulAdd()`, `FPRecipEstimate()`, `FPRecipStepFused()`, `FPRSqrtEstimate()`, `FPRSqrtStepFused()`, `FPSqrt()`, `FPSub()`, and `FPToFixed()`. All of these operations can generate floating-point exceptions.

Note

`FPAbs()` and `FPNeg()` are not classified as *floating-point operations* because:

- They cannot generate floating-point exceptions.
- The floating-point operation behavior described in the following sections does not apply to them:
  - *Flush-to-zero* on page A1-55.
  - *NaN handling and the Default NaN* on page A1-56.

More than one floating-point exception can occur on the same operation. The only combinations of floating-point exceptions that can occur are:

- Overflow with Inexact.
- Underflow with Inexact.
- Input Denormal with other floating-point exceptions.

The priority order of these floating-point exceptions is that the Inexact exception is treated as lowest priority, and the Input Denormal exception is treated as highest priority.

Some floating-point instructions specify more than one floating-point operation, as indicated by the pseudocode descriptions of the instruction. In such cases, a floating-point exception on one operation is treated as higher priority than a floating-point exception on another operation if the occurrence of the second floating-point exception depends on the result of the first operation. Otherwise, it is CONSTRAINED UNPREDICTABLE which floating-point exception is treated as higher priority, where the exception prioritized might differ between different instances of the same two floating-point exceptions being generated on the same operation during execution of the instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When none of the floating-point exceptions caused by an operation is trapped, any floating-point exception that occurs causes the associated cumulative bit in the FPSR to be set to 1.

When a floating-point exception is trapped:

- It is IMPLEMENTATION DEFINED whether the FPSR is restored when the trapped floating-point exception is taken. If the FPSR is not restored then, then it is CONSTRAINED UNPREDICTABLE which untrapped floating-point exception, if any, are indicated by the corresponding FPSR cumulative exception bits having the value 1.
- In the ESR_ELx for the Exception level to which the trapped exception is taken, the value of the floating-point exception trapped bit for the highest priority trapped floating-point exception must be 1.

  In this ESR_ELx:

  — The value of the floating-point exception trapped bit for any other untrapped floating-point exception generated by the same operation must be 0. This applies to both higher priority and lower priority untrapped floating point exceptions.

  — The value of the floating-point exception trapped bit for any lower priority trapped floating-point exception generated by the same operation might be 1, but the architecture does not require this.

For trapped floating-point exceptions from Advanced SIMD instructions, the architecture does not define the floating-point exception prioritization between different elements of the instruction. The architectural requirements for floating-point exception prioritization apply only to multiple floating-point exceptions generated on the same element of an Advanced SIMD operation.

Note

An implementation might provide information about a lower priority or untrapped floating-point exceptions in an IMPLEMENTATION DEFINED way, for example using an IMPLEMENTATION DEFINED register.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D1.13 Asynchronous exception types, routing, masking and priorities

In the Armv8-A architecture, asynchronous exceptions that are taken to AArch64 state are also known as *interrupts*.

There are two types of interrupts:

Physical interrupts Are signals sent to the PE from outside the PE. They are:
- SError. System Error.
- IRQ.
- FIQ.

Virtual interrupts Are interrupts that software executing at EL2 can enable and make pending. A virtual interrupt is taken from EL0 or EL1 to EL1.

Virtual interrupts have names that correspond to the physical interrupts:
- vSError.
- vIRQ.
- vFIQ.

Note

- For information about how virtual interrupts might be used see *Virtual interrupt usage model* on page D1-2322.
- The SError interrupt replaces the Armv7 asynchronous abort. The new name better describes the nature of the exception, and means that, in AArch64 state, it is categorized as a unique exception class, with EC encoding `0x2F`.

An External abort generated by the memory system might be taken asynchronously using the SError interrupt. These SError interrupts always behave as edge-triggered interrupts. An implementation might include other sources of SError interrupt. It is IMPLEMENTATION DEFINED whether these other sources are edge-triggered or level-sensitive. See also *External aborts* on page D4-2524.

Each physical interrupt type can be assigned a target Exception level of EL1, EL2 or EL3, as shown in *Asynchronous exception routing* on page D1-2364.

When an interrupt occurs:

- On taking an SError or a vSError interrupt to an Exception level using AArch64, the Exception Syndrome register for that Exception level is updated to describe an SError interrupt.

  When the RAS Extension is implemented, the exception syndrome for the vSError interrupt is taken from the values in the VSESR_EL2 register. See *Exception classes and the ESR_ELx syndrome registers* on page D1-2342, and the *ARM® Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for the ARMv8-A architecture profile*.

- On taking an IRQ, vIRQ, FIQ or vFIQ interrupt to an Exception level using AArch64, the Exception Syndrome register for that Exception level is not updated.

The remainder of this section contains the following:

- *Asynchronous exception routing* on page D1-2364.
- *Asynchronous exception masking* on page D1-2367.
- *Virtual interrupts* on page D1-2369.
- *Prioritization and recognition of interrupts* on page D1-2371.
- *Taking an interrupt or other exception during a multi-access load or store* on page D1-2372.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D1.13.1 Asynchronous exception routing

The following tables show the routing of physical interrupts when the highest implemented Exception level is using AArch64:

- For implementations that include both EL2 and EL3, see Table D1-10.
- For implementations that include EL3 but not EL2, see Table D1-11 on page D1-2365.
- For implementations that include EL2 but not EL3, see Table D1-12 on page D1-2366.

When the highest implemented Exception level is using AArch32, see Table G1-19 on page G1-5736.

In the tables:

SCR This is the *Effective value* of a field in SCR.

FIQ IRQ EA The *Effective value* of the field that handles the asynchronous exception type in SCR, if the highest EL is using AArch32, or SCR_EL3, if the highest EL is using AArch64.

HCR This is the *Effective value* of a field in HCR, if EL2 is using AArch32 or HCR_EL2 if EL2 is using AArch64.

When the value of the TGE is 1, the virtual exceptions are disabled.

When the *Effective value* of HCR.{E2H, TGE} is:

{0, 1} The *Effective value* of each of the HCR.{AMO, IMO, FMO} fields is 1.

{1, 1} The *Effective value* of each of the HCR.{AMO, IMO, FMO} fields is 0.

FMO IMO AMO The *Effective value* of the mask override field for the asynchronous exception type in HCR, if EL2 is using AArch32 or HCR_EL2 if EL2 is using AArch64.

EL2 The exception is taken to EL2 using AArch64.

EL3 The exception is taken to EL3 using AArch64.

C The interrupt is not taken and remains pending, regardless of the PSTATE.{A, I, F} interrupt masks.

FIQ IRQ Abt The exception is taken to the FIQ mode, the IRQ mode or the Abort mode according to the type of asynchronous exception.

Hyp The exception is taken to AArch32 Hyp mode.

Mon The exception is taken to AArch32 Monitor mode.

n/a Not applicable. The field does not exist in the register in this configuration or the Exception level is not accessible in this configuration.

Table D1-10 Routing when both EL3 and EL2 are implemented

SCR				HCR				Target when taken from EL0	Target when taken from EL1	Target when taken from EL2	Target when taken from EL3
NS	EEL2[a]	EA IRQ FIQ	RW	TGE	AMO IMO FMO	E2H	RW				
0	0	0	0	x	x	x	x	FIQ IRQ Abt	FIQ IRQ Abt	n/a	C
			1	x	x	x	x	EL1	EL1	n/a	C
		1	x	x	x	x	x	EL3	EL3	n/a	EL3
	1	0	x	0	0	0	0	FIQ IRQ Abt	FIQ IRQ Abt	C	C

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-10 Routing when both EL3 and EL2 are implemented (continued)

SCR				HCR				Target when taken from EL0	Target when taken from EL1	Target when taken from EL2	Target when taken from EL3
NS	EEL2[a]	EA IRQ FIQ	RW	TGE	AMO IMO FMO	E2H	RW				
							1	EL1	EL1	C	C
						1	x	EL1	EL1	C	C
					1	x	x	EL2	EL2	EL2	C
				1	x	x	x	EL2	n/a	EL2	C
		1	x	0	x	x	x	EL3	EL3	EL3	EL3
				1	x	x	x	EL3	n/a	EL3	EL3
1	x	0	0	0	0	n/a	n/a	FIQ IRQ Abt	FIQ IRQ Abt	Hyp	C
					1	n/a	n/a	Hyp	Hyp	Hyp	C
				1	x	n/a	n/a	Hyp	n/a	Hyp	C
			1	0	0	0	0	FIQ IRQ Abt	FIQ IRQ Abt	C	C
							1	EL1	EL1	C	C
						1	x	EL1	EL1	C	C
					1	x	x	EL2	EL2	EL2	C
				1	x	x	x	EL2	n/a	EL2	C
		1	x	0	x	x	x	EL3	EL3	EL3	EL3
				1	x	x	x	EL3	n/a	EL3	EL3

a. When the implementation does not include ARMv8.4-SecEL2, the SCR_EL3.EEL2 field is not implemented and the Effective value of EEL2 is 0.

Table D1-11 Routing when EL3 is implemented and EL2 is not implemented

SCR_EL3	Target Exception level when executing at		
EA IRQ FIQ	EL0	EL1	EL3
0	EL1	EL1	C
1	EL3	EL3	EL3

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-12 Routing when EL3 is not implemented and EL2 is implemented

HCR_EL2		Target Exception level when executing at		
TGE	**AMO IMO FMO**	**EL0**	**EL1**	**EL2**
0	0	EL1	EL1	C
	1	EL2	EL2	EL2
1	x	EL2	n/a	EL2

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D1.13.2 Asynchronous exception masking

When an interrupt is masked, it means that it cannot be taken. Instead, it remains pending.

When executing in AArch64 state, interrupts are masked implicitly when the target Exception level of the interrupt is lower than the current Exception level.

In addition, interrupts can be masked when the target Exception level is the current Exception level. The controls for this are:

SError PSTATE.A
IRQ PSTATE.I
FIQ PSTATE.F

When the target Exception level is higher than the current Exception level:

- If the target Exception level is EL3, the interrupt cannot be masked by the PSTATE.{A, I, F} bits.
- If the target Exception level is EL2, and either HCR_EL2.E2H is 0 or HCR_EL2.TGE is 0, the interrupt cannot be masked by the PSTATE.{A, I, F} bits.
- If the target Exception level is EL2, HCR_EL2.E2H is 1, and HCR_EL2.TGE is 1, the interrupt can be masked by the PSTATE.{A, I, F} bits.
- If the target Exception level is EL1, the interrupt can be masked by the PSTATE.{A, I, F} bits.

Note

- The ability to execute in EL0 with interrupts to EL1 masked is required by some user level driver code.
- The PSTATE.{A, I, F} bits can mask both physical interrupts and virtual interrupts.
- The Armv8-A architecture does not support *Non-maskable FIQ* (NMFI) operations. This means that it does not provide a configuration option to override the masking of FIQs by PSTATE.F.

On taking any exception to an Exception level using AArch64, all of PSTATE.{A, I, F} are set to 1, masking all interrupts that target that Exception level.

The following tables show the masking of physical interrupts when the highest implemented Exception level is using AArch64:

- For implementations that include both EL2 and EL3, see Table D1-13 on page D1-2368.
- For implementations that include EL3 but not EL2, see Table D1-14 on page D1-2369.
- For implementations that include EL2 but not EL3, see Table D1-15 on page D1-2369.

For the masking of interrupts when the highest implemented Exception level is using AArch32, see Table G1-20 on page G1-5737.

For the masking of virtual interrupts, see *Virtual interrupts* on page D1-2369.

In the tables:

SCR This is the *Effective value* of a field in SCR.

FIQ IRQ EA The *Effective value* of the field that handles the asynchronous exception type in SCR, if the highest EL is using AArch32, or SCR_EL3, if the highest EL is using AArch64.

HCR This is the *Effective value* of a field in HCR.

When the value of HCR.TGE is 1, the virtual exceptions are disabled.

When the *Effective value* of HCR.{E2H, TGE} is:

{0, 1} The *Effective value* of each of the HCR.{AMO, IMO, FMO} fields is 1.

{1, 1} The *Effective value* of each of the HCR.{AMO, IMO, FMO} fields is 0.

FMO IMO AMO The *Effective value* of the mask override field for the asynchronous exception type in HCR, if EL2 is using AArch32 or HCR_EL2 if EL2 is using AArch64.

A When the interrupt is asserted it is taken regardless of the value of the PSTATE.{A, I, F} interrupt masks.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

B When the interrupt is asserted it is subject to the corresponding Process state mask. If the value of the mask is 1 then the interrupt is not taken. If the value of the mask is 0 the interrupt is taken.

A/B When ARMv8.4-DFE is implemented, the interrupt is an SError interrupt, and SCR_EL3.NMEA is 1, then the interrupt behaves as A. Otherwise, the interrupt behaves as B.

C When the interrupt is asserted it is not taken, regardless of the value of the PSTATE.{A, I, F} interrupt masks.

n/a Not applicable. The PE cannot be executing at this Exception level for the specified state of HCR and SCR_EL3.

Table D1-13 Physical interrupt target and masking when both EL3 and EL2 are implemented

SCR				HCR			Effect of the interrupt mask when executing at:			
NS	EEL2[a]	EA IRQ FIQ	RW	TGE	E2H[b]	AMO IMO FMO	EL0	EL1	EL2	EL3
0	0	0	x	x	x	x	B	B	n/a	C
		1	x	x	x	x	A	A	n/a	A/B
	1	0	x	0	x	0	B	B	C	C
						1	A	A	B	C
				1	0	x	A	n/a	B	C
					1	x	B	n/a	B	C
		1	x	0	x	x	A	A	A	A/B
				1	x	x	A	n/a	A	A/B
1	x	0	0	0	n/a	0	B	B	B	C
						1	A	A	B	C
				1	n/a	x	A	n/a	B	C
			1	0	x	0	B	B	C	C
						1	A	A	B	C
				1	0	x	A	n/a	B	C
					1	x	B	n/a	B	C
		1	x	0	x	x	A	A	A	A/B
				1	x	x	A	n/a	A	A/B

a. When the implementation does not include ARMv8.4-SecEL2, the SCR_EL3.EEL2 field is not implemented and the Effective value of EEL2 is 0.

b. When the implementation does not include ARMv8.1-VHE, the HCR_EL2.E2H field is not implemented and the Effective value of E2H is 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-14 Physical interrupt target and masking when EL3 is implemented and EL2 is not implemented

SCR_EL3		Target Exception level	Effect of the interrupt mask when executing at:		
NS	EA IRQ FIQ		EL0	EL1	EL3
x	0	EL1	B	B	C
	1	EL3	A	A	A/B

Table D1-15 Physical interrupt target and masking when EL3 is not implemented and EL2 is implemented

HCR_EL2			Target Exception level	Effect of the interrupt mask when executing at:		
E2H[a]	TGE	AMO IMO FMO		EL0	EL1	EL2
x	0	0	EL1	B	B	C
		1	EL2	A	A	B
0	1	x	EL2	A	n/a	B
1	1	x	EL2	B	n/a	B

a. If the implementation does not include ARMv8.1-VHE, the HCR.E2H field is not implemented and behavior is as if the value of E2H is 0.

D1.13.3 Virtual interrupts

When the value of HCR_EL2.TGE is 0, setting an HCR_EL2.{FMO, IMO, AMO} routing control bit to 1 enables the corresponding virtual interrupt. When the value of HCR_EL2.TGE is 1 all virtual interrupts are disabled.

When execution is at EL2 or EL3, all types of virtual interrupt are always masked. If EL2 is not enabled in the current Security state, all types of virtual interrupts are always masked.

Virtual interrupts can only be taken from EL0 or EL1 to EL1. When a virtual interrupt type is enabled, that type of interrupt can be generated by:

- Software setting the corresponding virtual interrupt pending bit, HCR_EL2.{VSE, VI, VF}, to 1.
- For a vIRQ or a vFIQ, by an IMPLEMENTATION DEFINED mechanism. This might be a signal from an interrupt controller. See, for example, the *ARM Generic Interrupt Controller Architecture Specification*.

Note

For a usage model for virtual interrupts, see *Virtual interrupt usage model* on page D1-2322.

When a virtual interrupt is disabled:

- It cannot be taken.
- It cannot be seen in the ISR_EL1.

Each virtual interrupt type can be masked when execution is in EL1 or EL0, by using the same Process State mask bits that mask the physical interrupts, PSTATE.{A, I, F}.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-16 summarizes the bits that enable virtual interrupts and the bits that cause virtual interrupts to be pending.

Table D1-16 HCR_EL2 interrupt control bits

Virtual interrupt type	Enable control[a]	Cause a virtual interrupt to be pending
vSError	HCR_EL2.AMO	HCR_EL2.VSE
vIRQ	HCR_EL2.IMO	HCR_EL2.VI
vFIQ	HCR_EL2.FMO	HCR_EL2.VF

a. Applies only when the value of HCR_EL2.TGE is 0, otherwise the virtual interrupts are disabled.

On taking a vIRQ or a vFIQ interrupt, the corresponding virtual interrupt pending bit in the HCR_EL2 retains its state.

On taking a vSError interrupt, HCR_EL2.VSE is cleared to 0.

Note

This means that if the virtual interrupt pending bits are used, the vIRQ or vFIQ exception handler must cause software executing at EL2 or EL3 to set their corresponding virtual interrupt pending bits to 0.

Taking a vSError interrupt to an Exception level using AArch64 updates ESR_EL1 with the encoding for an SError interrupt. For the encoding, see *Exception classes and the ESR_ELx syndrome registers* on page D1-2342. When the RAS Extension is implemented, the exception syndrome for the vSError interrupt is taken from the values in the VSESR_EL2 register, see the *ARM® Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for the ARMv8-A architecture profile*. Taking a vIRQ or a vFIQ interrupt to an Exception level using AArch64 does not update the ESR_EL1.

The following table shows the masking of virtual interrupts when the highest implemented Exception level is using AArch64. In the table:

B When the interrupt is asserted it is subject to the corresponding Process state mask. If the value of the mask is 1 then the interrupt is not taken. If the value of the mask is 0 the interrupt is taken.

C When the interrupt is asserted it is not taken, regardless of the value of the Process state mask.

n/a Not applicable. The PE cannot be executing at this Exception level for the specified state of HCR and SCR_EL3.

HCR In Table D1-17, including in the table footnote:

- When EL2 is using AArch64 HCR refers to the AArch64 register HCR_EL2.
- When EL2 is using AArch32 HCR refers to the AArch32 register HCR.

When the value of HCR.TGE is 1, the virtual exceptions are disabled.

When the *Effective value* of HCR.{E2H, TGE} is:

{0, 1} The *Effective value* of each of the HCR.{AMO, IMO, FMO} fields is 1.

{1, 1} The *Effective value* of each of the HCR.{AMO, IMO, FMO} fields is 0.

Table D1-17 Virtual interrupt masking

SCR_EL3			HCR			Effect of the interrupt mask when executing at:			
EEL2	NS	EA IRQ FIQ	E2H[a]	TGE	AMO IMO FMO	EL0	EL1	EL2	EL3
0	0	x	x	x	x	C	C	n/a	C

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-17 Virtual interrupt masking (continued)

SCR_EL3			HCR			Effect of the interrupt mask when executing at:			
EEL2	NS	EA IRQ FIQ	E2H[a]	TGE	AMO IMO FMO	EL0	EL1	EL2	EL3
1	0	x	x	0	0	C	C	C	C
					1	B	B	C	C
				1	x	C	n/a	C	C
x	1	x	x	0	0	C	C	C	C
					1	B	B	C	C
				1	x	C	n/a	C	C

a. If EL2 is using AArch32 or the implementation does not include ARMv8.1-VHE, the HCR.E2H field is not implemented and behavior is as if the value of E2H is 0.

D1.13.4 Prioritization and recognition of interrupts

The prioritization of interrupts, including virtual interrupts, is IMPLEMENTATION DEFINED.

Note

As indicated at the start of *Asynchronous exception types, routing, masking and priorities* on page D1-2363, in AArch64 state all possible asynchronous exceptions are defined as *interrupts*.

Any interrupt that is pending before a *Context synchronization event* in the following list, is taken before the first instruction after the context synchronizing event, provided that the pending interrupt is not masked:

- Execution of an `ISB` instruction.
- Exception entry, if ARMv8.5-CSEH is not implemented, or if ARMv8.5-CSEH is implemented and the appropriate SCTLR_ELx.EIS bit is set.
- Exception return, if ARMv8.5-CSEH is not implemented or if ARMv8.5-CSEH is implemented and the appropriate SCTLR_ELx.EOS bit is set.
- Exit from Debug state.

Note

- If the first instruction after the context synchronizing event generates a synchronous exception, then the architecture does not define whether the PE takes the interrupt or the synchronous exception first.
- The ISR_EL1 identifies any pending interrupts.
- Interrupts are masked when the PE is in Debug state, and therefore this list of context synchronizing events does not include the `DCPS` and `DRPS` instructions.

An error synchronization event defines additional requirements for taking an SError interrupt, see the *ARM® Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for the ARMv8-A architecture profile*.

In the absence of a specific requirement to take an interrupt, the architecture only requires that unmasked pending interrupts are taken in finite time.

If an unmasked interrupt was pending but is changed to not pending before it is taken, then the architecture permits the interrupt to be taken, but does not require this to happen. If the interrupt is taken then it must be taken before the first *Context synchronization event* after the interrupt was changed to not pending.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D1.13.5 Taking an interrupt or other exception during a multi-access load or store

In AArch64 state, interrupts can be taken during a sequence of memory accesses caused by a single load or store instruction. This is true regardless of the memory type being accessed.

If an interrupt, or another exception, is taken from AArch64 during the execution of an instruction that performs a sequence of memory accesses, rather than a single single-copy atomic access, then:

- For a load, any register being loaded by the instruction other than ones used in the generation of the address by the instruction, can contain an UNKNOWN value. Registers used in the generation of the address are restored to their initial value.
- For a store, any data location being stored to by the instruction can contain an UNKNOWN value.
- For either a load or a store, if the instruction specifies writeback of the base address, then that register is restored to its initial value.

Note

- This interrupt behavior is in contrast to behavior in AArch32 state, when interrupts cannot be taken during a sequence of memory accesses caused by a single load or store instruction.
- In both Execution states, synchronous data abort exceptions can be taken during the execution of an instruction that performs a sequence of memory accesses.
- Software must avoid using multiple-register load and store instructions for accesses to Device memory, particularly to Device memory with the non-Gathering attribute, because an exception taken during the load or store can result in repeated accesses.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D1.14 Configurable instruction enables and disables, and trap controls

This section describes the controls provided by AArch64 state for enabling, disabling, and trapping particular instructions. Each control is categorized as an *instruction enable*, an *instruction disable*, or a *trap control*:

Instruction enables and instruction disables

Enable or disable the use of one or more particular instructions at a particular Exception level and Security state.

When an instruction is disabled as a result of an instruction enable or disable, it is UNDEFINED.

Trap controls A trap control determines whether one or more particular instructions, whenever executed at a particular Exception level, are *trapped*.

A trapped instruction generates a *Trap exception*.

For trap controls provided by:

EL1 Trap exceptions are taken to EL1, unless routed from EL0 to EL2 because HCR_EL2.TGE is 1 as described in *Routing exceptions from EL0 to EL2* on page D1-2354.

For descriptions of these controls see *EL1 configurable controls* on page D1-2374.

EL2 Trap exceptions are taken to EL2.

For descriptions of these controls see *EL2 configurable controls* on page D1-2378.

EL3 Trap exceptions are taken to EL3.

For descriptions of these controls see *EL3 configurable controls* on page D1-2389.

Note

The definitions of *traps* and *enables and disables* overlap, and the classification of some controls is historical. In AArch64 state, the most significant characteristic of an exception report is the ESR_ELx.EC value with which it is reported. Describing a register control field as an instruction enable, an instruction disable, or a trap control, gives no indication of how an exception that is generated as a consequence of the value of that field is handled or reported.

An exception generated as a result of an instruction enable or disable, or a trap control, is only taken if both of the following apply:

- The instruction generating the exception does not also generate a higher priority exception. *Synchronous exception prioritization for exceptions taken to AArch64 state* on page D1-2355 defines the prioritization of different exceptions on the same instruction.
- The instruction is not UNPREDICTABLE or CONSTRAINED UNPREDICTABLE in the PE state it is executed in. UNPREDICTABLE and CONSTRAINED UNPREDICTABLE instructions can generate exceptions as a result of these controls, but the architecture does not require them to do so.

Exceptions generated as a result of these controls are synchronous exceptions.

Exceptions are reported in the ESR_ELx, with an EC value that indicates the Exception class. and:

- Many cases, including all traps, are reported with a non-zero EC value and an associated syndrome.
- Some cases where an instruction is UNDEFINED are reported with an EC value `0x00`, the value for an exception for an unknown or uncategorized reason, and in these cases no syndrome is provided. *ISS encoding for exceptions with an unknown reason* on page D13-2972 identifies the cases that are reported with EC value `0x00`.

Table D1-8 on page D1-2343 lists the EC values that are used for exceptions that result from traps, enables, and disables.

Note

- A particular control might have a mnemonic that suggests it is different type of control to the control type it is categorized as. For example, SCTLR_EL1.DZE is a trap control even though DZE means DC ZVA Enable.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- In addition to the controls described in this section, a *routing control*, HCR_EL2.TGE, can be used to route exceptions from EL0 to EL2. See *Routing exceptions from EL0 to EL2* on page D1-2354.
- An implementation might provide additional controls, in IMPLEMENTATION DEFINED registers, to provide control of trapping of IMPLEMENTATION DEFINED features.

This section is organized as follows:

- *Traps on instructions*.
- *EL1 configurable controls*.
- *EL2 configurable controls* on page D1-2378.
- *EL3 configurable controls* on page D1-2389.

D1.14.1 Traps on instructions

When an instruction is disabled or trapped, the exception is taken before execution of the instruction. This means the preferred exception return of the exception is the instruction that is trapped.

If a conditional instruction is trapped, in AArch32 state, the Armv8-A architecture permits, but does not require the trap to apply to conditional AArch32 instructions that fail their Condition code check.

If the instruction is a register access instruction:

- No access is made before the exception is taken.
- Side-effects that are normally associated with the access do not occur before the exception is taken.

D1.14.2 EL1 configurable controls

These controls are in _EL0 and _EL1 System registers. The resulting exceptions might be taken from either Execution state. SPSR_EL1.M[4] indicates which Execution state the exception was taken from.

If HCR_EL2.TGE is 1 and EL2 is enabled in the current Security state, these Trap exceptions are routed to EL2 instead of EL1, see *Routing exceptions from EL0 to EL2* on page D1-2354.

Table D1-18 shows the_EL0 and _EL1 System registers that contain these controls.

Table D1-18 _EL1 registers that contain instruction enables and disables, and trap controls

Register name	Register description
AMUSERENR_EL0	Activity Monitors User Enable Register
CPACR_EL1	Architectural Feature Access Control Register
MDSCR_EL1	Monitor System Debug Control Register
PMUSERENR_EL0	Performance Monitors User Enable Register
SCTLR_EL1	System Control Register (EL1)
TCR_EL1	Translation Control Register (EL1)

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-19 summarizes the controls.

Table D1-19 Instruction enables and disables, and trap controls, provided by EL1

Control	Control type[a]	Description
AMUSERENR_EL0.EN	T	*Traps to EL1 of EL0 accesses to Activity Monitors registers*
CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN}	T	*Traps to EL1 of EL0 accesses to the Generic Timer registers* on page D1-2376
CPACR_EL1.TTA	T	*Traps to EL1 of EL0 and EL1 System register accesses to the trace registers* on page D1-2376
CPACR_EL1.FPEN	T	*Traps to EL1 of EL0 and EL1 accesses to SIMD and floating-point functionality* on page D1-2376
MDSCR_EL1.TDCC	T	*Traps to EL1 of EL0 accesses to the Debug Communications Channel (DCC) registers* on page D1-2376
PMUSERENR_EL0.{ER, CR, SW, EN}	T	*Traps to EL1 of EL0 accesses to Performance Monitors registers* on page D1-2376
SCTLR_EL1.{EnDA, EnDB, EnIA, EnIB}	E	*Enabling use of the Pointer authentication instructions, EL1&0 translation regime* on page D1-2377
SCTLR_EL1.UCI	T	*Traps to EL1 of EL0 execution of cache maintenance instructions* on page D1-2377
SCTLR_EL1.{nTWE, nTWI}	T	*Traps to EL1 of EL0 execution of WFE and WFI instructions* on page D1-2377
SCTLR_EL1.UCT	T	*Traps to EL1 of EL0 accesses to the CTR_EL0* on page D1-2377
SCTLR_EL1.DZE	T	*Traps to EL1 of EL0 execution of DC ZVA instructions* on page D1-2377
SCTLR_EL1.UMA	T	*Traps to EL1 of EL0 accesses to the PSTATE.{D, A, I, F} interrupt masks* on page D1-2377
SCTLR_EL1.{SED, ITD} SCTLR_EL1.CP15BEN	D E	*Disabling or enabling EL0 use of AArch32 deprecated functionality* on page D1-2377
TCR_EL1.{TBID0, TBID1}	D	*Disabling Address tagging for instruction accesses, EL1&0 translation regime* on page D1-2378

a. See Table D1-20.

Table D1-20 Control types, for exceptions taken to EL1

Abbreviation	Type	See
D	Disable	*Instruction enables and instruction disables* on page D1-2373
E	Enable	*Instruction enables and instruction disables* on page D1-2373
T	Trap	*Trap controls* on page D1-2373

Traps to EL1 of EL0 accesses to Activity Monitors registers

AMUSERENR_EL0.EN traps EL0 accesses to the Activity Monitors registers to EL1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Traps to EL1 of EL0 accesses to the Generic Timer registers

CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN} trap EL0 accesses to the Generic Timer registers to EL1, as follows:

- CNTKCTL_EL1.EL0PTEN traps EL0 accesses to the physical timer registers.
- CNTKCTL_EL1.EL0VTEN traps EL0 accesses to the virtual timer registers.
- CNTKCTL_EL1.EL0PCTEN traps EL0 accesses to the frequency register and physical counter register.
- CNTKCTL_EL1.EL0VCTEN traps EL0 accesses to the frequency register and virtual counter register.

Accesses to the frequency register, CNTFRQ_EL0 or CNTFRQ, are only trapped if CNTKCTL_EL1.EL0PCTEN and CNTKCTL_EL1.EL0VCTEN are both 0.

Traps to EL1 of EL0 and EL1 System register accesses to the trace registers

CPACR_EL1.TTA traps EL0 and EL1 System register accesses to the trace registers to EL1.

Note

- The ETMv4 architecture does not permit EL0 to access the trace registers. If the Armv8-A architecture is implemented with an ETMv4 implementation, EL0 accesses to the trace registers are UNDEFINED, and the resulting exception is higher priority than a CPACR_EL1.TTA Trap exception.
- The Armv8-A architecture does not provide traps on trace register accesses through the optional Memory-mapped interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, no side-effects occur before the exception is taken, see *Traps on instructions* on page D1-2374.

Traps to EL1 of EL0 and EL1 accesses to SIMD and floating-point functionality

When EL1 is using AArch64, CPACR_EL1.FPEN traps EL0 and EL1 accesses of the following registers to EL1:

- FPCR, FPSR, and any of the SIMD and floating-point registers V0-V31, including their views as D0-D31 registers or S0-S31 registers. See *The SIMD and floating-point registers, V0-V31* on page D1-2324.
- FPSCR, and any of the SIMD and floating-point registers Q0-Q15, including their views as D0-D31 registers or S0-S31 registers. See *Advanced SIMD and floating-point System registers* on page G1-5774.

The value of CPACR_EL1.FPEN determines whether the trap applies to accesses from both EL0 and EL1 using AArch64, or only to accesses from EL0 accesses from both Execution states.

Traps to EL1 of EL0 accesses to the Debug Communications Channel (DCC) registers

MDSCR_EL1.TDCC traps EL0 accesses to the DCC registers to EL1.

Traps of AArch32 accesses to DBGDTRRXint and DBGDTRTXint are ignored in Debug state.

Traps of AArch64 accesses to DBGDTR_EL0, DBGDTRRX_EL0, and DBGDTRTX_EL0 are ignored in Debug state.

Traps to EL1 of EL0 accesses to Performance Monitors registers

PMUSERENR_EL0.{ER, CR, SW, EN} trap EL0 accesses to the Performance Monitors registers to EL1.

For those Performance Monitors registers that more than one PMUSERENR_EL0.{ER, CR, SW, EN} control applies to, accesses are only trapped if all controls that apply are set to 0.

The accesses that these trap controls trap might be reads, writes, or both.

PMUSERENR_EL0.EN traps EL0 access only if the corresponding EL1 accesses is permitted. For example, the PMSWINC_EL0 and PMSWINC registers are WO at EL1, and therefore are not trapped.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Enabling use of the Pointer authentication instructions, EL1&0 translation regime

This control is implemented when ARMv8.3-PAuth is implemented.

Each of the SCTLR_EL1.{EnDA, EnDB, EnIA, EnIB} fields enables the pointer authentication functionality for the corresponding Pointer authentication instructions for the EL1&0 translation regime. For more information see *System register control of pointer authentication* on page D5-2539.

Note

These controls cause the pointer authentication instructions to execute as `NOP`s. They never cause an exception to be generated.

Traps to EL1 of EL0 execution of cache maintenance instructions

SCTLR_EL1.UCI traps EL0 execution using AArch64 of cache maintenance instructions to EL1.

Traps to EL1 of EL0 execution of `WFE` and `WFI` instructions

SCTLR_EL1.{nTWE, nTWI} trap EL0 execution of `WFE` and `WFI` instructions to EL1 if the instruction would otherwise have caused the PE to enter a low-power state.

Note

Since a `WFE` or `WFI` can complete at any time, even without a Wakeup event, the traps on `WFE` of `WFI` are not guaranteed to be taken, even if the `WFE` or `WFI` is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

For more information about these instructions, and when they can cause the PE to enter a low-power state, see:

- *Wait for Event mechanism and Send event* on page D1-2397.
- *Wait For Interrupt* on page D1-2400.

Traps to EL1 of EL0 accesses to the CTR_EL0

SCTLR_EL1.UCT traps EL0 accesses using AArch64 to the CTR_EL0 to EL1.

Traps to EL1 of EL0 execution of `DC ZVA` instructions

SCTLR_EL1.DZE traps EL0 execution of `DC ZVA` instructions to EL1. If the trap is enabled, reading the DCZID_EL0 returns a value that indicates that `DC ZVA` instructions are not implemented.

Traps to EL1 of EL0 accesses to the PSTATE.{D, A, I, F} interrupt masks

SCTLR_EL1.UMA traps EL0 execution of `MSR` and `MRS` instructions that access the PSTATE.{D, A, I, F} masks to EL1. If HCR_EL2.TGE is 1 and EL2 is enabled in the current Security state, these Trap exceptions are routed to EL2.

Disabling or enabling EL0 use of AArch32 deprecated functionality

Table D1-21 on page D1-2378 shows the deprecated AArch32 functionality that might have disable controls in the SCTLR_EL1:

- The SED control is always implemented.
- Whether each of the ITD, CP15BEN controls is implemented is IMPLEMENTATION DEFINED. If a control is not implemented then the associated functionality cannot be disabled.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

These SCTLR_EL1 controls apply only to execution at EL0 using AArch32. When an instruction is disabled by one of these controls, it is UNDEFINED at EL0 using AArch32. Table D1-21 shows how the exceptions are reported in ESR_EL1:

Table D1-21 EL1 controls for disabling and enabling EL0 use of AArch32 deprecated functionality

Deprecated AArch32 functionality	Instruction enable or disable in the SCTLR_EL1	Disabled instructions	Syndrome reporting in ESR_EL1[a]
`SETEND` instructions	SED[b]	`SETEND` instructions	Exception for an unknown reason, using EC value `0x00`
Some uses of `IT` instructions	ITD[c]	See the SCTLR_EL1.IT description	
Accesses to the CP15DMB, CP15DSB, and CP15ISB barrier instructions	CP15BEN[d]	`MCR` accesses to the CP15DMB, CP15DSB, and CP15ISB instructions	

a. If HCR_EL2.TGE is 1 and EL2 is enabled in the current Security state, the exception is routed to EL2 and reported in ESR_EL2 using the EC value shown in the table.

b. `SETEND` instruction disable. `SETEND` instructions are disabled when the value of this field is 1.

c. `IT` instruction disable. If this control is implemented, some uses of `IT` instructions are disabled when the value of this field is 1.

d. System register (`coproc==0b1111`) memory barrier enable. If this control is implemented, the specified register accesses are disabled when the value of CP15BEN is 0.

Note

- The uses of the `IT` instruction, and use of the CP15DMB, CP15DSB, and CP15ISB barrier instructions, are deprecated for performance reasons.
- The SCTLR provides similar controls that apply when EL1 is using AArch32, and the HSCTLR provides similar controls that apply when EL2 is using AArch32.

Disabling Address tagging for instruction accesses, EL1&0 translation regime

This control is implemented when ARMv8.3-PAuth is implemented.

When a TCR_EL1.{TBI0, TBI1} field enables the use of address tagging for the EL1&0 translation regime, the corresponding TCR_EL1.{TBID0, TBID1} field determines whether address tagging is used for both data and instruction addresses, or only for data addresses. For more information see *Address tagging in AArch64 state* on page D5-2534.

Note

These controls determine the scope of address tagging. They never cause an exception to be generated.

D1.14.3 EL2 configurable controls

These controls are in _EL2 System registers. The resulting exceptions might be taken from either Execution state. SPSR_EL2.M[4] indicates which Execution state the exception was taken from.

These controls are ignored in Secure state.

ARMv8.6-FGT provides fine-grained traps for architected system registers. Where implementations have IMPLEMENTATION DEFINED registers accessible from EL1 or EL0, Arm recommends that EL2 accessible fine-grained traps are provided for these registers using a control register held in IMPLEMENTATION DEFINED space.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-22 shows the _EL2 System registers that contain these controls.

Table D1-22 _EL2 registers that contain instruction disables and trap controls

Register name	Register description
CPTR_EL2	Architectural Feature Trap Register, EL2
HAFGRTR_EL2	Hypervisor Activity Monitors Fine-Grained Read Trap Register
HCR_EL2	Hypervisor Configuration Register
HDFGRTR_EL2	Hypervisor Debug Fine-Grained Read Trap Register
HDFGWTR_EL2	Hypervisor Debug Fine-Grained Write Trap Register
HFGITR_EL2	Hypervisor Fine-Grained Instruction Trap Register
HFGRTR_EL2	Hypervisor Fine-Grained Read Trap Register
HFGWTR_EL2	Hypervisor Fine-Grained Write Trap Register
HSTR_EL2	Hypervisor System Trap Register
MDCR_EL2	Monitor Debug Configuration Register, EL2
SCTLR_EL2	System Control Register, EL2
TCR_EL2	Translation Control Register, EL2

Table D1-23 summarizes the controls.

Note

For completeness, Table D1-23 includes the routing control described in *Routing exceptions from EL0 to EL2* on page D1-2354.

Table D1-23 Instruction disables and trap controls provided by EL2

Control	Control type[a]	Description
CNTHCTL_EL2.{EL1PCEN, EL1PCTEN}	T	*Traps to EL2 of EL0 and EL1 accesses to the Generic Timer registers* on page D1-2381
CPTR_EL2.TCPAC	T	*Trapping to EL2 of EL1 accesses to the CPACR_EL1 or CPACR* on page D1-2381
CPTR_EL2.TAM	T	*Traps to EL2 of EL1 and EL0 accesses to Activity Monitors registers* on page D1-2382
CPTR_EL2.TTA	T	*Traps to EL2 of System register accesses to the trace registers* on page D1-2382
CPTR_EL2.TFP	T	*General trapping to EL2 of accesses to the SIMD and floating-point registers* on page D1-2382
HAFGRTR_EL2	T	*Fine-grained traps to EL2 of EL0 and EL1 read accesses to Activity Monitors registers* on page D1-2382
HCR_EL2.FIEN	T	*Traps to EL2 of EL1 accesses to the RAS error record registers* on page D1-2382

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-23 Instruction disables and trap controls provided by EL2 (continued)

Control	Control type[a]	Description
HCR_EL2.AT	T	*Trap to EL2 of EL1 accesses to AT S1E* instructions* on page D1-2382
HCR_EL2.{NV, NV1}	T	*Traps to EL2 for Nested virtualization* on page D1-2383
HCR_EL2.API	T	*Trap to EL2 of EL0 accesses to Pointer authentication instructions* on page D1-2383
HCR_EL2.APK	T	*Trap to EL2 of EL1 accesses to Pointer authentication key registers* on page D1-2383
HCR_EL2.TERR	T	*Traps to EL2 of EL1 accesses to the RAS error record registers* on page D1-2382
HCR_EL2.{TRVM, TVM}	T	*Traps to EL2 of EL1 accesses to virtual memory control registers* on page D1-2384
HCR_EL2.HCD	D	*Disabling Non-secure state execution of HVC instructions* on page D1-2384
HCR_EL2.TDZ	T	*Traps to EL2 of EL0 and EL1 execution of DC ZVA instructions* on page D1-2384
HCR_EL2.TGE	R	*Routing exceptions from EL0 to EL2* on page D1-2354
HCR_EL2.TTLB	T	*Traps to EL2 of EL1 execution of TLB maintenance instructions* on page D1-2384
HCR_EL2.{TSW, TPC, TPU}	T	*Traps to EL2 of EL0 and EL1 execution of cache maintenance instructions* on page D1-2384
HCR_EL2.TACR	T	*Traps to EL2 of EL1 accesses to the Auxiliary Control Register* on page D1-2385
HCR_EL2.TIDCP	T	*Traps to EL2 of EL0 and EL1 accesses to lockdown, DMA, and TCM operations* on page D1-2385
HCR_EL2.TSC	T	*Traps to EL2 of EL1 execution of SMC instructions* on page D1-2385
HCR_EL2.{TID0, TID1, TID2, TID3}	T	*Traps to EL2 of EL0 and EL1 accesses to the ID registers* on page D1-2386
HCR_EL2.{TWI, TWE}	T	*Traps to EL2 of EL0 and EL1 execution of WFE and WFI instructions* on page D1-2386
HDFGRTR_EL2 HDFGWTR_EL2	T	*Fine-grained traps to EL2 of EL0 and EL1 accesses to the debug, trace, and PMU registers* on page D1-2387
HDFGRTR_EL2 HDFGWTR_EL2	T	*Fine-grained traps to EL2 of EL0 and EL1 accesses to System registers* on page D1-2387
HFGITR_EL2	T	*Fine-grained Traps to EL2 of EL0 and EL1 accesses to instructions* on page D1-2387
HSTR_EL2.{T0-T3, T5-T13, T15}	T	*General trapping to EL2 of EL0 and EL1 accesses to System registers, from AArch32 state only* on page D1-2387
MDCR_EL2.TDCC	T	*Traps to EL2 of EL0 and EL1 accesses to the Debug Communications Channel registers* on page D1-2389
MDCR_EL2.TTRF	T	*Traps to EL2 of System register accesses to the trace filter control registers* on page D1-2388

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-23 Instruction disables and trap controls provided by EL2 (continued)

Control	Control type[a]	Description
MDCR_EL2.{TDRA, TDOSA, TDA}	T	*Traps to EL2 of EL0 and EL1 System register accesses to debug registers* on page D1-2388
MDCR_EL2.{TPM, TPMCR}	T	*Traps to EL2 of EL0 and EL1 accesses to Performance Monitors registers* on page D1-2389
SCTLR_EL2.{EnDA, EnDB, EnIA, EnIB}	E	*Enabling use of the Pointer authentication instructions, EL2 translation regime* on page D1-2389
TCR_EL2.TBID0 or TCR_EL2.{TBID0, TBID1}	D	*Disabling Address tagging for instruction accesses, EL2 translation regime* on page D1-2389

a. See Table D1-24.

Table D1-24 Control types, for exceptions taken to EL1

Abbreviation	Type	See
D	Disable	*Instruction enables and instruction disables* on page D1-2373
E	Enable	*Instruction enables and instruction disables* on page D1-2373
R	Routing control	*Routing exceptions from EL0 to EL2* on page D1-2354
T	Trap	*Trap controls* on page D1-2373

Also see the following for more general information about traps to EL2:

- *Traps on instructions* on page D1-2374.
- For traps from an Exception level using AArch32:
  - *Instructions that fail their Condition code check* on page G1-5789.
  - *Trapping to EL2 of instructions that are UNPREDICTABLE* on page G1-5789.

Traps to EL2 of EL0 and EL1 accesses to the Generic Timer registers

CNTHCTL_EL2.{EL1PCEN, EL1PCTEN} trap EL0 and EL1 accesses to the Generic Timer registers to EL2 if enabled for the current Security state, as follows:

- CNTHCTL_EL2.EL1PCEN traps EL0 and EL1 accesses to the physical timer registers.
- CNTHCTL_EL2.EL1PCTEN traps EL0 and EL1 accesses to the physical counter register.

Trapping to EL2 of EL1 accesses to the CPACR_EL1 or CPACR

CPTR_EL2.TCPAC traps EL1 accesses to the CPACR_EL1 or CPACR to EL2:

Note

- The CPACR_EL1 or CPACR is not accessible at EL0.
- In Armv7 and earlier versions of the Arm architecture, one function of the CPACR is as an ID register that identifies what coprocessor or conceptual coprocessor functionality is implemented. Legacy software might use this identification mechanism, and a hypervisor can use this trap to emulate this mechanism. For more information about this coprocessor model see *Background to the System register interface* on page G1-5770.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Traps to EL2 of EL1 and EL0 accesses to Activity Monitors registers

CPTR_EL2.TAM traps EL1 and EL0 accesses to the Activity Monitor registers to EL2.

Traps to EL2 of System register accesses to the trace registers

CPTR_EL2.TTA traps System register accesses to the trace registers to EL2.

Note

- The ETMv4 architecture does not permit EL0 to access the trace registers. If the Armv8-A architecture is implemented with an ETMv4 implementation, EL0 accesses to the trace registers are UNDEFINED, and any resulting exception is higher priority than a CPTR_EL2.TTA Trap exception.
- EL2 does not provide traps on trace register accesses through the Memory-mapped interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, no side-effects occur before the exception is taken, see *Traps on instructions* on page D1-2374.

General trapping to EL2 of accesses to the SIMD and floating-point registers

CPTR_EL2.TFP traps accesses to the following SIMD and floating-point registers to EL2:

- FPCR, FPSR, FPEXC32_EL2, and any of the SIMD and floating-point registers V0-V31, including their views as D0-D31 registers or S0-S31 registers. See *The SIMD and floating-point registers, V0-V31* on page D1-2324
- FPSID, MVFR0, MVFR1, MVFR2, FPSCR, FPEXC, and any of the SIMD and floating-point registers Q0-Q15, including their views as D0-D31 registers or S0-S31 registers. See *Advanced SIMD and floating-point System registers* on page G1-5774. Permitted `VMSR` accesses to the FPSID are ignored, but for the purposes of this trap the architecture defines a `VMSR` access to the FPSID from EL1 or higher as an access to a SIMD and floating-point register.

Fine-grained traps to EL2 of EL0 and EL1 read accesses to Activity Monitors registers

The fields in HAFGRTR_EL2 trap read accesses to individual or pairs of Activity Monitors registers to EL2 if enabled in the current Security state. The values of the register are treated as 0 for all purposes other than direct reads of the register when HCR_EL2.{E2H, TGE} is {1,1}.

EL1 accesses are trapped from AArch64. When EL1 is using AArch64 and the functionality is accessible from EL0, EL0 accesses are trapped from AArch64 or AArch32.

If the Activity monitors extension is not implemented, HAFGRTR_EL2 is not implemented. If an Activity monitor auxiliary counter is not implemented, the corresponding field in HAGFRTR_EL2 is RES0.

Traps to EL2 of EL1 accesses to the RAS error record registers

HCR_EL2.TERR traps EL1 accesses to the RAS ER* registers to EL2 if enabled in the current Security state.

HCR_EL2.FIEN traps EL1 accesses to the RAS ER* registers to EL2 if enabled in the current Security state.

Trap to EL2 of EL1 accesses to `AT S1E*` instructions

This control is implemented when ARMv8.3-NV is implemented.

HCR_EL2.AT traps to EL2, if enabled in the current Security state, from EL1 accesses to some Address translation instructions. Because nested virtualization is supported only in AArch64 state this control only traps from AArch64 state.

For more information see *Effect of HCR_EL2.{NV, NV1}* on page D5-2644.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Traps to EL2 for Nested virtualization

These controls are implemented when ARMv8.3-NV is implemented.

Note

When ARMv8.4-NV is implemented and HCR_EL2.NV2 is 1, the redirection of register accesses to memory accesses has priority over the trapping of register accesses by HCR_EL2.{NV, NV1}, see *Enhanced support for nested virtualization* on page D5-2646.

HCR_EL2.NV traps the following to EL2, if enabled in the current Security state, from EL1:

- Some System register, System instruction, and Special-purpose register accesses that are UNDEFINED at EL1 when ARMv8.3-NV is not implemented.

  Only accesses that are not UNDEFINED at EL2 are trapped.

  Note

  This means that, for a register that is RO at EL2, and UNDEFINED at Non-secure EL1 when ARMv8.3-NV is not implemented, when ARMv8.3-NV is implemented and this trap is enabled:

  — Read accesses to the register from EL1 are trapped to EL2.

  — Write accesses to the register from EL1 remain UNDEFINED.

- The execution of some instructions that are UNDEFINED at EL1 when ARMv8.3-NV is not implemented.

Because nested virtualization is supported only in AArch64 state this control only traps from AArch64 state.

Note

In addition, when the value of HCR_EL2.NV is 1, a read of CurrentEL returns the value `0b10` for bits[3:2].

HCR_EL2.NV1 traps to EL2, if enabled in the current Security state, from EL1 accesses to some System registers and Special-purpose registers. Because nested virtualization is supported only in AArch64 state this control only traps from AArch64 state.

For more information see *Effect of HCR_EL2.{NV, NV1}* on page D5-2644.

Trap to EL2 of EL0 accesses to Pointer authentication instructions

This control is implemented when ARMv8.3-PAuth is implemented.

HCR_EL2.API traps, to EL2 if enabled in the current Security state, accesses to any of the Pointer authentication instructions for which pointer authentication is enabled, for instructions executed either:

- At EL1.
- If the *Effective value* of HCR_EL2.{TGE, E2H} is not {1, 1}, at EL0.

Because pointer authentication is supported only in AArch64 state, this control only traps from AArch64 state.

For more information, including the description of when pointer authentication is enabled for an instruction, see *System register control of pointer authentication* on page D5-2539.

Trap to EL2 of EL1 accesses to Pointer authentication key registers

This control is implemented when ARMv8.3-PAuth is implemented.

HCR_EL2.APK traps, to EL2 if enabled in the current Security state, accesses to the Pointer authentication key registers from EL1 to EL2. Because pointer authentication is supported only in AArch64 state this control only traps from AArch64 state.

For more information see *System register control of pointer authentication* on page D5-2539.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Traps to EL2 of EL1 accesses to virtual memory control registers

HCR_EL2.{TRVM, TVM} trap EL1 accesses to the virtual memory control registers to EL2, if enabled in the current Security state.

Note

EL2 provides a second stage of address translation, that a hypervisor can use to remap the address map defined by a Guest OS. In addition, a hypervisor can trap attempts by a Guest OS to write to the registers that control the Non-secure memory system. A hypervisor might use this trap as part of its virtualization of memory management.

Disabling Non-secure state execution of HVC instructions

HCR_EL2.HCD disables execution of HVC instructions at EL2 and EL1, and any resulting exception is taken from the current Exception level to the current Exception level.

Note

HVC instructions are always UNDEFINED at EL0.

HCR_EL2.HCD is only implemented if EL3 is not implemented. Otherwise, it is RES0.

Traps to EL2 of EL0 and EL1 execution of DC ZVA instructions

HCR_EL2.TDZ traps EL0 and EL1 execution of DC ZVA instructions to EL2 if enabled in the current Security state, and reading the DCZID_EL0 returns a value that indicates that DC ZVA instructions are not implemented.

Traps to EL2 of EL1 execution of TLB maintenance instructions

In the Armv8-A architecture, the System instruction encoding space includes TLB maintenance instructions.

HCR_EL2.TTLB traps EL1 execution of TLB maintenance instructions to EL2 if enabled in the current Security state:

Note

These instructions are always UNDEFINED at EL0.

For more information about these instructions, see:

- *TLB maintenance instructions* on page D5-2671, for the AArch64 state instructions.
- *The scope of TLB maintenance instructions* on page G5-6004, for the AArch32 state instructions.

Traps to EL2 of EL0 and EL1 execution of cache maintenance instructions

HCR_EL2.{TSW, TPC, TPU} trap cache maintenance instructions to EL2, if enabled in the current Security state. Execution is trapped from EL1, or from EL0 if permitted by SCTLR_EL1.UCI.

HCR_EL2.TSW traps data or unified cache maintenance by set/way instructions.

Note

These instructions are always UNDEFINED at EL0.

HCR_EL2.TPC traps data or unified cache maintenance to point of coherency instructions.

Note

DC IVAC is always UNDEFINED at EL0 using AArch64.

DCIMVAC, DCCIMVAC, and DCCMVAC are always UNDEFINED at EL0 using AArch32.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

HCR_EL2.TPU traps cache maintenance to point of unification instructions.

Note

IC IALLUIS and IC IALLU are always UNDEFINED at EL0 using AArch64.

ICIMVAU, ICIALLU, ICIALLUIS, and DCCMVAU are always UNDEFINED at EL0 using AArch32.

For more information about these instructions, see:

- *Cache maintenance instructions, and data cache zero operation* on page C5-381 for the AArch64 instructions.
- *Cache maintenance system instructions* on page K14-8064 for the AArch32 instructions.

Traps to EL2 of EL1 accesses to the Auxiliary Control Register

HCR_EL2.TACR traps EL1 accesses to the Auxiliary Control Registers to EL2 if enabled in the current Security state:

Note

- The ACTLR_EL1, ACTLR, and ACTLR2 are not accessible at EL0.
- The Auxiliary Control Registers are IMPLEMENTATION DEFINED registers that might implement global control bits for the PE.

Traps to EL2 of EL0 and EL1 accesses to lockdown, DMA, and TCM operations

The lockdown, DMA, and TCM features of the Armv8-A architecture are IMPLEMENTATION DEFINED. The architecture reserves the encodings of a number of System registers for control of these features.

HCR_EL2.TIDCP traps the execution of System register access instructions that access any of the encodings described in *Reserved encodings for IMPLEMENTATION DEFINED registers* on page D12-2866 and any of the following AArch32 encodings:

- `CRn==c9`, `opc1==`{0-7}, `CRm==`{c0-c2, c5-c8}, `opc2==`{0-7}.
- `CRn==c10`, `opc1==`{0-7}, `CRm==`{c0, c1, c4, c8}, `opc2==`{0-7}.
- `CRn==c11`, `opc1==`{0-7}, `CRm==`{c0-c8, c15}, `opc2==`{0-7}.

Execution at EL1 is trapped to EL2 if enabled in the current Security state. Execution at EL1 is an IMPLEMENTATION DEFINED choice between either a trap to EL2, or UNDEFINED with any resulting exception taken to EL1.

An implementation can also include IMPLEMENTATION DEFINED registers that provide additional controls, to give finer-grained control of the trapping of IMPLEMENTATION DEFINED features.

Note

- Arm expects the trapping of EL0 accesses to these functions to EL2 to be unusual, and used only when the hypervisor is virtualizing EL0 operation. Arm strongly recommends that unless the hypervisor must virtualize EL0 operation, a EL0 access to any of these functions is UNDEFINED, as it would be if the implementation did not include EL2. The PE then takes any resulting exception to EL1.
- The trapping of accesses to these registers from EL1 is higher priority than an exception resulting from the register access being UNDEFINED.

Traps to EL2 of EL1 execution of `SMC` instructions

HCR_EL2.TSC traps EL1 execution of `SMC` instructions to EL2 if enabled in the current Security state. the value of SCR_EL3.SMD is ignored.

If EL3 is not implemented, HCR_EL2.TSC is RES0.

For more information about `SMC` instructions, see *SMC* on page C6-1205.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

- This trap is implemented only if the implementation includes EL3.
- SMC instructions are UNDEFINED at EL0.
- HCR_EL2.TSC traps execution of the SMC instruction. It is not a routing control for the SMC exception. Trap exceptions and SMC exceptions have different preferred return addresses.

Traps to EL2 of EL0 and EL1 accesses to the ID registers

Other than the MIDR_EL1, MPIDR_EL1, and PMCR_EL0.N, the ID registers are divided into groups, with a trap control in the HCR_EL2 for each group.

HCR_EL2.TID0 traps accesses to primary device identification registers at EL1 and EL0 to EL2 if enabled in the current Security state. HCR_EL2.TID1 traps accesses to implementation identification registers at EL1 to EL2 if enabled in the current Security state. HCR_EL2.TID2 traps accesses to cache identification registers at EL1 and EL0 to EL2 if enabled in the current Security state. HCR_EL2.TID3 traps accesses to detailed feature identification registers at EL1 to EL2 if enabled in the current Security state.

Note

In AArch32 state, the detailed feature identification registers are called the CPUID registers. There is no requirement for this trap to apply to those registers that the CPUID Identification Scheme defines as reserved. See *The CPUID identification scheme* on page G8-6097.

For the MIDR_EL1 and MPIDR_EL1, and for PMCR_EL0.N, the architecture provides read/write aliases. The original register becomes accessible only from EL2 or Secure state, and an EL0 or EL1 read of the original register returns the value of the read/write alias. This substitution is invisible to the EL0 or EL1 software reading the register.

Table D1-25 ID register substitution

Register	Original	Alias, EL2 using AArch64
Main ID	MIDR_EL1	VPIDR_EL2
Multiprocessor Affinity	MPIDR_EL1	VMPIDR_EL2
Performance Monitors Control Register	PMCR_EL0.N	MDCR_EL2.HPMN

Note

- If the optional Performance Monitors Extension is not implemented, MDCR_EL2.HPMN is RES0 and PMCR_EL0 is reserved.
- MDCR_EL2.HPMN also affects whether a Performance Monitors counter can be accessed from EL0 or EL1. See the register description of MDCR_EL2 for more information.
- PMCR_EL0 contains other fields that identify the implementation. For more information about trapping accesses to the PMCR_EL0, see *Traps to EL2 of EL0 and EL1 accesses to Performance Monitors registers* on page D1-2389.

Traps to EL2 of EL0 and EL1 execution of WFE and WFI instructions

HCR_EL2.{TWE, TWI} trap EL0 and EL1 execution of WFE and WFI instructions to EL2 if the instruction would otherwise have caused the PE to enter a low-power state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

For more information about these instructions, and when they can cause the PE to enter a low-power state, see:

- *Wait for Event mechanism and Send event* on page D1-2397.
- *Wait For Interrupt* on page D1-2400.

Fine-grained traps to EL2 of EL0 and EL1 accesses to the debug, trace, and PMU registers

The fields in HDFGRTR_EL2 and HDFGWTR_EL2 trap read and write accesses to individual and groups of related debug and trace registers to EL2 if enabled in the current Security state. The values of the registers are treated as 0 for all purposes other than direct reads of the register when HCR_EL2.{E2H, TGE} is {1,1}.

Most RW registers have fine-grained traps for read and write accesses. However, PMCR_EL0 has a trap for write accesses only. For more details, see HDFGRTR_EL2 and HDFGWTR_EL2.

If a fine-grained trap selects a breakpoint or watchpoint that is not implemented, the access is UNDEFINED. Accesses to unimplemented registers and unimplemented event counters are UNDEFINED.

When ARMv8.6-FGT is implemented, access to an implemented Performance Monitors counter <n> when n is greater than, or equal to MDCR_EL2.HPMN is always trapped to EL2, and the value of the corresponding fine-grained trap field in HDFGRTR_EL2 or HDFGWTR_EL2 is ignored. If ARMv8.6-FGT is not implemented, access to an implemented Performance Monitors counter <n> when n is greater than or equal to MDCR_EL2.HPMN has CONSTRAINED UNPREDICTABLE behavior.

Fine-grained traps to EL2 of EL0 and EL1 accesses to System registers

The fields in HFGRTR_EL2 and HFGWTR_EL2 trap read or write accesses to individual or small groups of system registers to EL2 if enabled in the current Security state. The values of the register are treated as 0 for all purposes other than direct reads of the register when HCR_EL2.{E2H, TGE} is {1,1}.

EL1 accesses are trapped from AArch64. When EL1 is using AArch64 and the functionality is accessible from EL0, EL0 accesses are trapped from AArch64 or AArch32.

If a register is not implemented, the corresponding field is RES0.

Fine-grained Traps to EL2 of EL0 and EL1 accesses to instructions

The fields in HFGITR_EL2 trap specific system instructions to EL2 if enabled in the current Security state. The values of the register are treated as 0 for all purposes other than direct reads of the register when HCR_EL2.{E2H, TGE} is {1,1}.

If an instruction is not implemented and behaves as an unallocated instruction, the corresponding field in HFGITR_EL2 is RES0.

For cache maintenance instructions to a PoC, PoU, PoP, or PoDP, if no caches are defined to be affected in the implementation before that point in the memory system, it is IMPLEMENTATION DEFINED whether the instruction is trapped when the corresponding trap enable field is set.

General trapping to EL2 of EL0 and EL1 accesses to System registers, from AArch32 state only

HSTR_EL2.{T0-T3, T5-T13, T15} trap accesses to the AArch32 System registers in the `coproc==0b1111` encoding space, by the register number, {c0-c3, c5-c13, c15} used for:

- The `CRn` argument used when accessing the register using an `MCR` or `MRC` instruction.
- The `CRm` argument used when accessing the register using an `MCRR` or `MRRC` instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

These traps are from AArch32 state only. They are from both:

- EL1 using AArch32.
- EL0 using AArch32.

Note

HSTR_EL2[4, 14] is reserved, RES0. Although the Generic Timer AArch32 System registers are implemented in the `coproc==0b1111` encoding space and accessed using a `CRn` or `CRm` value of c14, EL2 does not provide a trap on accesses to the Generic Timer System registers.

System registers in the (`coproc==0b1111`) encoding space with IMPLEMENTATION DEFINED access permission from EL0

For an AArch32 System register in the (`coproc==0b1111`) encoding space, which is accessed using a CRn or CRm value that can be trapped by a HSTR_EL2.Tn control, if an access to the register from EL0 is UNDEFINED when the value of the corresponding HSTR_EL2.Tn trap control is 0, then when that HSTR_EL2.Tn trap control is 1, it is IMPLEMENTATION DEFINED whether an access from Non-secure EL0 using AArch32:

- Generates a Trap exception that is taken to EL2.
- Is UNDEFINED and generates an exception that is taken to Non-secure EL1.

If the instruction is treated as UNDEFINED and generates an exception that is taken to Non-secure EL1, and Non-secure EL1 is using AArch64, the exception is reported in ESR_EL1 as an exception for an unknown reason, using EC value `0x00`.

Note

Arm expects that trapping to EL2 of Non-secure EL0 accesses to AArch32 System register in the (`coproc==0b1111`) encoding space will be unusual, and used only when the hypervisor must virtualize EL0 operation. Arm recommends that, whenever possible, Non-secure EL0 accesses to the System registers behave as they would if the implementation did not include EL2. This means that, if the architecture does not support the Non-secure EL0 access, then the register access instruction is treated as UNDEFINED and generates an exception that is taken to Non-secure EL1.

Traps to EL2 of System register accesses to the trace filter control registers

MDCR_EL2.TTRF traps System register accesses to the trace filter control registers to EL2, if enabled in the current Security state.

Traps to EL2 of EL0 and EL1 System register accesses to debug registers

MDCR_EL2.{TDRA, TDOSA, TDA} trap System register accesses to the debug registers to EL2 if enabled in the current Security state, as follows:

- MDCR_EL2.TDRA traps EL0 and EL1 accesses to the Debug ROM registers to EL2 if enabled in the current Security state. This trap applies to Non-secure EL0 only if it is using AArch32.If MDCR_EL2.TDE or HCR_EL2.TGE is 1, behavior is as if MDCR_EL2.TDRA is 1 other than for the purpose of a direct read.
- MDCR_EL2.TDOSA traps EL1 accesses to powerdown debug registers to EL2 if enabled in the current Security state. These registers are not accessible at EL0.If MDCR_EL2.TDE or HCR_EL2.TGE is 1, behavior is as if MDCR_EL2.TDOSA is 1 other than for the purpose of a direct read.
- MDCR_EL2.TDA traps EL0 and EL1 accesses to those debug System registers that are not trapped by MDCR_EL2.TDRA and MDCR_EL2.TDOSA. The MDCR_EL2.TDA traps are to EL2 if enabled in the current Security state. If MDCR_EL2.TDE or HCR_EL2.TGE is 1, behavior is as if MDCR_EL2.TDA is 1 other than for the purpose of a direct read.

Note

EL2 does not provide traps on debug register accesses through the optional memory-mapped external debug interfaces.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

System register accesses to the debug registers can have side-effects. When a System register access is trapped to EL2, no side-effects occur before the exception is taken to EL2. See *Traps on instructions* on page D1-2374.

Traps to EL2 of EL0 and EL1 accesses to the Debug Communications Channel registers

If the PE is not in Debug state, MDCR_EL2.TDCC traps EL0 and EL1 accesses to DCC registers to EL2 if enabled for the current Security state.

If the PE is in Debug state, MDCR_EL2.TDCC does not trap accesses to DBGDTR_EL0, DBGDTRRX_EL0, DBGDTRTX_EL0, DBGDTRRXint, and DBGDTRTXint that would otherwise be trapped. See MDCR_EL2.TDCC for more information.

Traps to EL2 of EL0 and EL1 accesses to Performance Monitors registers

MDCR_EL2.{TPM, TPMCR} trap EL0 and EL1 accesses to the Performance Monitors registers to EL2 if enabled in the current Security state:

Note

EL2 does not provide traps on Performance Monitor register accesses through the optional memory-mapped external debug interface.

MDCR_EL2.HPMN affects whether a counter can be accessed from Non-secure EL0 or EL1. See the register description of MDCR_EL2 for more information.

Enabling use of the Pointer authentication instructions, EL2 translation regime

This control is implemented when ARMv8.3-PAuth is implemented.

Each of the SCTLR_EL2.{EnDA, EnDB, EnIA, EnIB} fields enables the pointer authentication functionality for the corresponding Pointer authentication instructions for the EL2 or EL2&0 translation regime. For more information see *System register control of pointer authentication* on page D5-2539.

Note

These controls cause the pointer authentication instructions to execute as `NOP`s. They never cause an exception to be generated.

Disabling Address tagging for instruction accesses, EL2 translation regime

This control is implemented when ARMv8.3-PAuth is implemented.

When a TCR_EL2.TBI or TCR_EL2.{TBI0, TBI1} field enables the use of address tagging for the EL2 translation regime, the corresponding TCR_EL2.TBID or TCR_EL2.{TBID0, TBID1} field determines whether address tagging is used for both data and instruction addresses, or only for data addresses. For more information see *Address tagging in AArch64 state* on page D5-2534.

Note

These controls determine the scope of address tagging. They never cause an exception to be generated.

D1.14.4 EL3 configurable controls

These controls are in _EL3 System registers. The resulting exceptions might be taken from either Execution state. SPSR_EL3.M[4] indicates which Execution state the exception was taken from.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-26 shows the _EL3 System registers that contain these controls.

Table D1-26 _EL3 registers that contain instruction enables and disables, and trap controls

Register name	Register description
SCTLR_EL3	System Control Register, EL3
SCR_EL3	Secure Configuration Register
CPTR_EL3	Architectural Feature Trap Register, EL3
MDCR_EL3	Monitor Debug Configuration Register, EL3
TCR_EL3	Translation Control Register, EL3

Table D1-27 summarizes the controls.

Table D1-27 Instruction enables and disables, and trap controls, provided by EL3

Control	Control type[a]	Description
CPTR_EL3.TCPAC	T	*Trapping to EL3 of EL2 accesses to the CPTR_EL2 or HCPTR, and EL2 and EL1 accesses to the CPACR_EL1 or CPACR* on page D1-2391
CPTR_EL3.TAM	T	*Traps to EL3 of EL2, EL1, and EL0 accesses to Activity Monitors registers* on page D1-2392
CPTR_EL3.TTA	T	*Traps to EL3 of System register accesses to the trace registers* on page D1-2392
CPTR_EL3.TFP	T	*Traps to EL3 of all accesses to the SIMD and floating-point registers* on page D1-2392
MDCR_EL3.TDCC	T	*Traps to EL3 of EL2, EL1, and EL0 accesses to Debug Communication Channel registers* on page D1-2392
MDCR_EL3.TTRF	T	*Traps to EL3 of System register accesses to the trace registers* on page D1-2392
MDCR_EL3.{TDOSA, TDA}	T	*Traps to EL3 of EL2, EL1, and EL0 System register accesses to debug registers* on page D1-2393
MDCR_EL3.TPM	T	*Traps to EL3 of EL2, EL1, and EL0 accesses to Performance Monitors registers* on page D1-2393
SCR_EL3.FGTEn	E	*Traps to EL3 of EL2 accesses to fine-grained trap registers* on page D1-2393
SCR_EL3.FIEN	T	*Traps to EL3 of EL1 and EL2 accesses to the RAS error record registers* on page D1-2393
SCR_EL3.API	T	*Trap to EL3 accesses to Pointer authentication instructions* on page D1-2393
SCR_EL3.APK	T	*Trap to EL3 accesses to Pointer authentication key registers* on page D1-2393
SCR_EL3.TERR	T	*Traps to EL3 of EL1 and EL2 accesses to the RAS error record registers* on page D1-2393
SCR_EL3.{TWE, TWI}	T	*Traps to EL3 of EL2, EL1, and EL0 execution of WFE and WFI instructions* on page D1-2393
SCR_EL3.ST	T	*Traps to EL3 of Secure EL1 accesses to the Counter-timer Physical Secure timer registers* on page D1-2394
SCR_EL3.HCE	E	*Enabling EL3, EL2, and Non-secure EL1 execution of HVC instructions* on page D1-2394

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-27 Instruction enables and disables, and trap controls, provided by EL3 (continued)

Control	Control type[a]	Description
SCR_EL3.SMD	D	*Disabling EL3, EL2, and EL1 execution of SMC instructions* on page D1-2394
SCTLR_EL3.{EnDA, EnDB, EnIA, EnIB}	E	*Enabling use of the Pointer authentication instructions, EL3 translation regime* on page D1-2394
TCR_EL3.TBID	D	*Disabling Address tagging for instruction accesses, EL3 translation regime* on page D1-2395

a. See Table D1-28.

Table D1-28 Control types, for exceptions taken to EL1

Abbreviation	Type	See
D	Disable	*Instruction enables and instruction disables* on page D1-2373
E	Enable	*Instruction enables and instruction disables* on page D1-2373
T	Trap	*Trap controls* on page D1-2373

Also see the following for more general information about traps to EL3:

- *Traps on instructions* on page D1-2374.
- *Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32*.

Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32

If EL1 is using AArch32, all of the following are trapped to EL3:

- Secure EL1 reads and writes to any of the SCR, NSACR, MVBAR or SDCR.
- Any attempt at Secure EL1 to execute any of the following:
  - ATS12NSO** instructions.
  - `SRS` instructions that use the R13_mon banked register.
  - `MRS` or `MSR` instructions that access any of the SPSR_mon, R13_mon or R14_mon banked registers.

In addition, if EL1 is using AArch32:

- Secure EL1 write accesses to the CNTFRQ register are UNDEFINED. They are not trapped to EL3.
- Any attempt at Secure EL1 to change the PE mode to Monitor mode, by using a `CPS` or an `MSR` instruction, or by performing an exception return, is treated as an illegal change of the CPSR.M field. See *Illegal changes to PSTATE.M* on page G1-5699.

Note

- Reads of the NSACR from either Non-secure EL1 using AArch32 or Non-secure EL2 using AArch32 return the value `0x00000C00`. See *Restricted access System registers* on page G5-6053.
- These operations are not available at EL0.

Trapping to EL3 of EL2 accesses to the CPTR_EL2 or HCPTR, and EL2 and EL1 accesses to the CPACR_EL1 or CPACR

CPTR_EL3.TCPAC traps all of the following to EL3:

- EL2 accesses to the CPTR_EL2 or HCPTR.
- EL2 and EL1 accesses to the CPACR_EL1 or CPACR.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When CPTR_EL3.TCPAC is:

For EL1, this trap control applies to accesses from both Security states.

Traps to EL3 of EL2, EL1, and EL0 accesses to Activity Monitors registers

CPTR_EL3.TAM traps EL2, EL1, and EL0 accesses to the Activity Monitor registers to EL3.

Traps to EL3 of System register accesses to the trace registers

CPTR_EL3.TTA traps System register accesses to the trace registers, from all Exception levels, to EL3:

Note

- The ETMv4 architecture does not permit EL0 to access the trace registers. If the Armv8-A architecture is implemented with an ETMv4 implementation, EL0 accesses to the trace registers are UNDEFINED, and any resulting exception is higher priority than a CPTR_EL3.TTA Trap exception.
- EL3 does not provide traps on trace register accesses through the Memory-mapped interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, no side-effects occur before the exception is taken, see *Traps on instructions* on page D1-2374.

For EL0 and EL1, this trap control applies to accesses from both Security states.

Traps to EL3 of all accesses to the SIMD and floating-point registers

CPTR_EL3.TFP traps all accesses to the following SIMD and floating-point registers, from all Exception levels, to EL3:

- FPCR, FPSR, FPEXC32_EL2, and any of the SIMD and floating-point registers V0-V31, including their views as D0-D31 registers or S0-S31 registers. See *The SIMD and floating-point registers, V0-V31* on page D1-2324.
- FPSID, MVFR0, MVFR1, MVFR2, FPSCR, FPEXC, and any of the SIMD and floating-point registers Q0-Q15, including their views as D0-D31 registers or S0-S31 registers. See *Advanced SIMD and floating-point System registers* on page G1-5774. Permitted `VMSR` accesses to the FPSID are ignored, but for the purposes of this trap the architecture defines a `VMSR` access to the FPSID from EL1 or higher is an access to a SIMD and floating-point register.

For EL0 and EL1, this trap control applies to accesses from both Security states.

Note

- FPEXC32_EL2 is not accessible from EL0 using AArch64.
- FPSID, MVFR0, MVFR1, and FPEXC are not accessible from EL0 using AArch32.

Traps to EL3 of EL2, EL1, and EL0 accesses to Debug Communication Channel registers

MDCR_EL3.TDCC traps EL2, EL1, and EL0 accesses to DCC registers to EL3.

If the PE is in Debug state, MDCR_EL3.TDCC does not trap accesses to DBGDTR_EL0, DBGDTRRX_EL0, DBGDTRTX_EL0, DBGDTRRXint, and DBGDTRTXint that would otherwise be trapped. See MDCR_EL3.TDCC for more information.

Traps to EL3 of all System register accesses to the filter trace control registers

MDCR_EL3.TTRF traps System register accesses to the trace registers, from all Exception levels, to EL3:

For EL0 and EL1, this trap control applies to accesses from both Security states.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Traps to EL3 of EL2, EL1, and EL0 System register accesses to debug registers

MDCR_EL3.TDOSA traps EL2, EL1, and EL0 System register accesses to the powerdown debug registers to EL3, from both Security states. For EL1, this trap control applies to accesses from both Security states.

MDCR_EL3.TDA traps EL2, EL1, and EL0 System register accesses to the debug registers that are not trapped by MDCR_EL3.TDOSA, to EL3, from both Security states.

Note

EL3 does not provide traps on debug register accesses through the Memory-mapped or External debug interfaces.

System register accesses to the debug registers can have side-effects. When a System register access is trapped to EL3, no side-effects occur before the exception is taken to EL3. See *Traps on instructions* on page D1-2374.

Traps to EL3 of EL2, EL1, and EL0 accesses to Performance Monitors registers

MDCR_EL3.TPM traps EL2, EL1, and EL0 accesses to the Performance Monitors registers to EL3:

For EL0 and EL1, this trap control applies to accesses from both Security states.

Traps to EL3 of EL2 accesses to fine-grained trap registers

If SCR_EL3.FGTEn is set to 0, EL2 accesses to the following fine-grained trap registers are trapped to EL3:

- HFGRTR_EL2, HFGWTR_EL2 for System register traps.
- HFGITR_EL2 for System instruction traps.
- HDFGRTR_EL2, HDFGWTR_EL2 for debug and trace register traps.
- HAFGRTR_EL2 for activity monitor register traps.

Traps to EL3 of EL1 and EL2 accesses to the RAS error record registers

SCR_EL3.FIEN traps EL1 and EL2 accesses to the RAS ERXP* registers to EL3.

SCR_EL3.TERR traps EL1 and EL2 read accesses to the RAS ER* registers that are not trapped by SCR_EL3.FIEN, to EL3.

Trap to EL3 accesses to Pointer authentication instructions

This control is implemented when ARMv8.3-PAuth is implemented.

SCR_EL3.API traps, to EL3, accesses to any of the Pointer authentication instructions for which pointer authentication is enabled, for instructions executed at an Exception level lower than EL3, in either Security state.

Because pointer authentication is supported only in AArch64 state this control only traps from AArch64 state.

For more information, including the description of when pointer authentication is enabled for an instruction, see *System register control of pointer authentication* on page D5-2539.

Trap to EL3 accesses to Pointer authentication key registers

This control is implemented when ARMv8.3-PAuth is implemented.

SCR_EL3.APK traps, to EL3, accesses to the Pointer authentication key registers from EL2 or from Secure or Non-secure EL1. Because pointer authentication is supported only in AArch64 state this control only traps from AArch64 state.

For more information see *System register control of pointer authentication* on page D5-2539.

Traps to EL3 of EL2, EL1, and EL0 execution of `WFE` and `WFI` instructions

SCR_EL3.{TWE, TWI} trap EL2, EL1, and EL0 execution of `WFE` and `WFI` instructions to EL3.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

For more information about these instructions, and when they can cause the PE to enter a low-power state, see:

- *Wait for Event mechanism and Send event* on page D1-2397.
- *Wait For Interrupt* on page D1-2400.

Traps to EL3 of Secure EL1 accesses to the Counter-timer Physical Secure timer registers

SCR_EL3.ST traps Secure EL1 accesses to the Counter-timer Physical Secure timer registers to EL3.

Note

- Accesses to the Counter-timer Physical Secure timer registers are always enabled at EL3.
- These registers are not accessible at EL0.

Enabling EL3, EL2, and Non-secure EL1 execution of HVC instructions

SCR_EL3.HCE enables HVC instruction execution at EL1 and above. Otherwise, HVC instructions are UNDEFINED at EL1, EL2, and EL3, and any resulting exception is taken from the current Exception level to the current Exception level

For EL1, this enable control applies to HVC instructions in Non-secure state only.

If EL2 is not implemented, this bit is RES0.

Note

HVC instructions are always UNDEFINED at EL0.

Disabling EL3, EL2, and EL1 execution of SMC instructions

SCR_EL3.SMD disables SMC instruction execution at EL1 and above. SMC instructions are UNDEFINED at EL1 and above, and any resulting exception is taken from the current Exception level to the current Exception level.

For EL1, this disable control applies to SMC instructions in both Security states.

Note

SMC instructions are always UNDEFINED at EL0.

If HCR_EL2.TSC or HCR.TSC traps attempted EL1 execution of SMC instructions to EL2, that trap has priority over this disable.

Enabling use of the Pointer authentication instructions, EL3 translation regime

This control is implemented when ARMv8.3-PAuth is implemented.

Each of the SCTLR_EL3.{EnDA, EnDB, EnIA, EnIB} fields enables the pointer authentication functionality for the corresponding Pointer authentication instructions for the EL3 translation regime. For more information see *System register control of pointer authentication* on page D5-2539.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

These controls cause the pointer authentication instructions to execute as `NOP`s. They never cause an exception to be generated.

Disabling Address tagging for instruction accesses, EL3 translation regime

This control is implemented when ARMv8.3-PAuth is implemented.

When the TCR_EL3.TBI field enables the use of address tagging for the EL3 translation regime, the TCR_EL3.TBID field determines whether address tagging is used for both data and instruction addresses, or only for data addresses. For more information see *Address tagging in AArch64 state* on page D5-2534.

Note

This control determines the scope of address tagging. It never causes an exception to be generated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D1.15 System calls

A system call is generated by the execution of an SVC, HVC, or SMC instruction:

- By default, the execution of an SVC instruction generates a Supervisor Call, a synchronous exception that targets EL1. This provides a mechanism for software executing at EL0 to make a call to an operating system or other software executing at EL1.
- In an implementation that includes EL2, the execution of an HVC instruction generates a Hypervisor Call, a synchronous exception that targets EL2 by default.

  The HVC instruction is UNDEFINED:

  — At EL0.

  — At EL1 in Secure state.

  Note

  Software executing at EL0 cannot directly generate a Hypervisor Call.

- In an implementation that includes EL3, by default the execution of an SMC instruction generates a Secure Monitor Call, a synchronous exception that targets EL3.

  The SMC instruction is UNDEFINED at EL0, meaning software executing at EL0 cannot directly generate a Secure Monitor Call.

The default behavior applies when the instruction is not UNDEFINED and both of the following are true:

- The instruction is executed at an Exception level that is the same as or lower than the target Exception level.
- The instruction is not trapped to a different Exception level.

If an SVC or HVC instruction is executed at an Exception level that is higher than the target Exception then it generates a synchronous exception that is taken to the current Exception level.

EL2 and EL3 can disable Hypervisor Call exceptions, see:

- *Disabling Non-secure state execution of HVC instructions* on page D1-2384.
- *Enabling EL3, EL2, and Non-secure EL1 execution of HVC instructions* on page D1-2394.

EL2 can trap use of the SMC instruction, see *Traps to EL2 of EL1 execution of SMC instructions* on page D1-2385.

EL3 can disable Secure Monitor Call exceptions, see *Disabling EL3, EL2, and EL1 execution of SMC instructions* on page D1-2394.

D1.15.1 Pseudocode description of system calls

The `AArch64.CallSupervisor()` pseudocode function performs an SVC call in AArch64 state.

The `AArch64.CallHypervisor()` pseudocode function performs an HVC call in AArch64 state.

The `AArch64.CallSecureMonitor()` pseudocode function performs an SMC call in AArch64 state.

The `AArch64.CallSupervisor()`, `AArch64.CallHypervisor()`, and `AArch64.CallSecureMonitor()` functions are described in Chapter J1 *Armv8 Pseudocode*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D1.16 Mechanisms for entering a low-power state

The Arm architecture provides mechanisms that software can use to indicate that the PE can enter a low-power state, if it supports that state. The following sections describe those mechanisms:

- *Wait for Event mechanism and Send event*.
- *Wait For Interrupt* on page D1-2400.

D1.16.1 Wait for Event mechanism and Send event

A PE can use the *Wait for Event* (WFE) mechanism to enter a low-power state, depending on the value of the Event Register for that PE. To enter the low-power state, the PE executes a Wait For Event instruction, `WFE`, and if the Event Register is clear, the PE can enter the low-power state.

If the PE does enter the low-power state, it remains in that low-power state until it receives a *WFE wake-up event*.

The architecture does not define the exact nature of the low-power state, except that the execution of a `WFE` instruction must not cause a loss of memory coherency.

WFE mechanism behavior depends on the interaction of all of the following, that are described in the subsections that follow:

- The Event Register for the PE. See subsection *The Event Register* on page D1-2398.
- The Wait For Event instruction, `WFE`. See subsection *The Wait For Event instruction* on page D1-2398.
- *WFE wake-up events*. See subsection *WFE wake-up events in AArch64 state* on page D1-2399
- The Send Event instructions, `SEV` and `SEVL` that can cause WFE wake-up events. See subsection *The Send Event instructions* on page D1-2399.

Note

Because the Wait for Event mechanism is associated with suspending execution on a PE for the purpose of power saving, Arm recommends that the Event Register is set only infrequently. However, software must only use the setting of the Event Register as a hint, and must not assume that any particular message is sent as a result of the setting of the Event Register.

Example D1-2 describes how a spinlock implementation might use the WFE mechanism to save energy.

Example D1-2 Spinlock as an example of using Wait For Event and Send Event

A multiprocessor operating system requires locking mechanisms to protect data structures from being accessed simultaneously by multiple PEs. These mechanisms prevent the data structures becoming inconsistent or corrupted if different PEs try to make conflicting changes. If a lock is busy, because a data structure is being used by one PE, it might not be practical for another PE to do anything except wait for the lock to be released. For example, if a PE is handling an interrupt from a device, it might need to add data received from the device to a queue. If another PE is removing data from the same queue, it will have locked the memory area that holds the queue. The first PE cannot add the new data until the queue is in a consistent state and the second PE has released the lock. The first PE cannot return from the interrupt handler until the data has been added to the queue, so it must wait.

Typically, a spin-lock mechanism is used in these circumstances:

- A PE requiring access to the protected data attempts to obtain the lock using single-copy atomic synchronization primitives such as the Load-Exclusive and Store-Exclusive operations described in *Synchronization and semaphores* on page B2-167.
- If the PE obtains the lock it performs its memory operation and then releases the lock.
- If the PE cannot obtain the lock, it reads the lock value repeatedly in a tight loop until the lock becomes available. When the lock becomes available, the PE again attempts to obtain it.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A spin-lock mechanism is not ideal for all situations:

- In a low-power system the tight read loop is undesirable because it uses energy to no effect.
- In a multiprocessor system the execution of spin-locks by multiple waiting PEs can degrade overall performance.

Using the Wait For Event and Send Event mechanism can improve the energy efficiency of a spinlock:

- A PE that fails to obtain a lock executes a `WFE` instruction to request entry to a low-power state, at the time when the Exclusives monitor is set holding the address of the location holding the lock.
- When a PE releases a lock, the write to the lock location causes the Exclusives monitor of any PE monitoring the lock location to be cleared. This clearing of the Exclusives monitors generates a WFE wake-up event for each of those PEs. Then, these PEs can attempt to obtain the lock again.

For large systems, more advanced locking systems, such as ticket locks, can avoid unfairness caused by having multiple PEs simultaneously reading the lock. In such systems, the WFE mechanism can be used in a similar way to monitor the next ticket value.

The Event Register

The Event Register is a single bit register for each PE. When set, an Event Register indicates that an event has occurred since the register was last cleared, that might require some action by the PE. Therefore, when the Event Register is set, the PE must not suspend operation on executing a `WFE` instruction.

The reset value of the Event Register is UNKNOWN.

The Event Register for a PE is set by any of the following:

- A Send Event instruction, `SEV`, executed by any PE in the system.
- A Send Event Local instruction, `SEVL`, executed by the PE.
- An exception return.
- The clearing of the global monitor for the PE.
- An event from a Generic Timer event stream, see *Event streams* on page D11-2845.
- An event sent by some IMPLEMENTATION DEFINED mechanism.

The Event Register is cleared only by a Wait For Event instruction.

Note

Software cannot read or write the value of the Event Register directly.

The Wait For Event instruction

The action of the Wait For Event instruction, `WFE`, depends on the state of the Event Register:

- If the Event Register is set, the instruction clears the register and completes immediately.
- If the Event Register is clear the PE can suspend execution and enter a low-power state. It remains in that state until the PE detects a WFE wake-up event, or earlier if the implementation chooses, or until a reset. When the PE detects a WFE wake-up event, or earlier if chosen, the WFE instruction completes. If the wake-up event sets the Event Register, it is IMPLEMENTATION DEFINED whether on restarting execution, the Event Register is cleared.

Note

Software using the Wait For Event mechanism must tolerate spurious wake-up events, including multiple wake-ups.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Trapping of WFE

The `WFE` instruction is available at all Exception levels. Attempts to enter a low-power state made by software executing at EL0, EL1, or EL2 might be trapped to a higher Exception level. See:

- *Traps to EL1 of EL0 execution of WFE and WFI instructions* on page D1-2377.
- *Traps to EL2 of EL0 and EL1 execution of WFE and WFI instructions* on page D1-2386.
- *Traps to EL3 of EL2, EL1, and EL0 execution of WFE and WFI instructions* on page D1-2393.

If ARMv8.6-TWED is implemented, the delay for taking a `WFE` trap is configurable.

The delay on the trap does not affect the priority of the traps. In particular, if execution is subject to a trap at EL1 as a result of SCTLR_EL1.nTWE==0 and HCR_EL2.TWE==1, the only trap that will be taken is a trap to EL1, even if the delay at EL1 is longer than the delay at EL2.

WFE wake-up events in AArch64 state

The following are *WFE wake-up events*:

- The execution of an `SEV` instruction on any PE in the multiprocessor system.
- Any physical SError interrupt, IRQ interrupt, or FIQ interrupt received by the PE, that is not disabled by EDSCR.INTdis and:
  - Is marked as A in the tables in *Asynchronous exception masking* on page D1-2367, regardless of the value of the corresponding PSTATE.{A, I, F} mask bit.
  - Is marked as B in the tables in *Asynchronous exception masking* on page D1-2367, if the value of the corresponding PSTATE.{A, I, F} mask bit is 0.

  Note

  Any physical SError interrupt, IRQ interrupt, or FIQ interrupt that is marked as A/B behaves as A or B. See *A/B* on page D1-2368.

- In EL1 or EL0, any virtual SError interrupt, IRQ interrupt, or FIQ interrupt received by the PE, that is not disabled by EDSCR.INTdis and is marked as B in Table D1-17 on page D1-2370 in *Virtual interrupts* on page D1-2369, if the value of the corresponding PSTATE.{A, I, F} mask bit is 0.
- An asynchronous External Debug Request debug event, if halting is allowed. For the definition of *halting is allowed* see *Halting allowed and halting prohibited* on page H2-6927.

  See also *External Debug Request debug event* on page H3-6983.
- An event sent by the timer event stream for the PE. See *Event streams* on page D11-2845.
- An event caused by the clearing of the global monitor for the PE.
- An event sent by some IMPLEMENTATION DEFINED mechanism.

Not all of these wake-up events set the Event Register.

Note

The disabling of interrupts, and WFE wake-up events, by EDSCR.INTdis is possible only when external debug is enabled.

The Send Event instructions

The Send Event instructions are:

SEV, Send Event This causes an event to be signaled to all PEs in the multiprocessor system.

SEVL, Send Event Local

This must set the local Event Register.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

It might signal an event to other PEs by some IMPLEMENTATION DEFINED mechanism, but is not required to do so.

The mechanism that signals an event to other PEs is IMPLEMENTATION DEFINED. The PE is not required to guarantee the ordering of this event with respect to the completion of memory accesses by instructions before the `SEV` instruction. Therefore, Arm recommends that software includes a `DSB` instruction before any `SEV` instruction.

Note

A `DSB` instruction ensures that no instructions, including any `SEV` instructions, that appear in program order after the `DSB` instruction, can execute until the `DSB` instruction has completed. See *Data Synchronization Barrier (DSB)* on page B2-138.

The `SEVL` instruction appears to execute in program order relative to any subsequent `WFE` instruction executed on the same PE, without the need for any explicit insertion of barrier instructions.

The receipt of a signaled `SEV` or `SEVL` event by a PE sets the Event Register on that PE.

The `SEV` and `SEVL` instructions are available at all Exception levels.

Pseudocode description of the Wait For Event mechanism

This section identifies pseudocode functions that describe the behavior of the Wait For Event mechanism.

The `ClearEventRegister()` pseudocode function clears the Event Register of the current PE.

The `IsEventRegisterSet()` pseudocode function returns TRUE if the Event Register of the current PE is set and FALSE if it is clear.

The `WaitForEvent()` pseudocode function optionally suspends execution until a WFE wake-up event or reset occurs, or until some earlier time if the implementation chooses. It is IMPLEMENTATION DEFINED whether restarting execution after the period of suspension causes `ClearEventRegister()` to be called.

The `SendEvent()` pseudocode function sets the Event Register of every PE in the multiprocessor system.

The `SendEventLocal()` pseudocode function sets the event register for the local PE.

D1.16.2 Wait For Interrupt

Software can use the *Wait for Interrupt* (WFI) instruction to cause the PE to enter a low-power state. The PE then remains in that low-power state until it receives a *WFI wake-up event*, or until some other IMPLEMENTATION DEFINED reason causes it to leave the low-power state. The architecture permits a PE to leave the low-power state for any reason, but requires that it must leave the low-power state on receipt of any architected WFI wake-up event.

Note

Because the architecture permits a PE to leave the low-power state for any reason, it is permissible for a PE to treat `WFI` as a `NOP`, but this is not recommended for lowest power operation.

When the PE leaves a low-power state that was entered as a result of a `WFI` instruction, that `WFI` instruction completes.

The architecture does not define the exact nature of the low-power state, except that the execution of a `WFI` instruction must not cause a loss of memory coherency.

Attempts to enter a low-power state made by software executing at EL0, EL1, or EL2 might be trapped to a higher Exception level. See:

- *Traps to EL1 of EL0 execution of WFE and WFI instructions* on page D1-2377.
- *Traps to EL2 of EL0 and EL1 execution of WFE and WFI instructions* on page D1-2386.
- *Traps to EL3 of EL2, EL1, and EL0 execution of WFE and WFI instructions* on page D1-2393.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

WFI wake-up events

The following are *WFI wake-up events*:

- Any physical SError interrupt, IRQ interrupt, or FIQ interrupt received by the PE, that is marked as A, Bor A/B in the tables in *Asynchronous exception masking* on page D1-2367, regardless of the value of the corresponding PSTATE.{A, I, F} mask bit.
- In EL1 or EL0, any virtual SError interrupt, IRQ interrupt, or FIQ interrupt received by the PE, that is marked as B in Table D1-17 on page D1-2370 in *Virtual interrupts* on page D1-2369, regardless of the value of the corresponding PSTATE.{A, I, F} mask bit.
- An asynchronous External Debug Request debug event, if halting is allowed. For the definition of *halting is allowed* see *Halting allowed and halting prohibited* on page H2-6927.

  See also *External Debug Request debug event* on page H3-6983.
- An event sent by some IMPLEMENTATION DEFINED mechanism.

Note

- WFI wake-up events are never disabled by EDSCR.INTdis, and are never masked by the PSTATE.{A, I, F} mask bits. If wake-up is invoked by an interrupt that is disabled or masked the interrupt is not taken.
- Because debug events are WFI wake-up events, Arm recommends that Wait For Interrupt is used as part of an idle loop rather than waiting for a single specific interrupt event to occur and then moving forward. This ensures that the intervention of debug while waiting does not significantly change the function of the program being debugged.
- Some implementations of the WFI mechanism drain down any pending memory activity before suspending execution. This increases power saving, by increasing the area over which clocks can be stopped. The architecture does not require this operation, therefore software must not rely on the WFI mechanism operating in this way.

Using WFI to indicate an idle state on bus interfaces

Software can use the WFI mechanism to force quiescence on a PE, and, combined with preventing any possible WFI wakeup events, this can be used to complete an entry into a powerdown state.

Because mechanisms for entering powerdown states are inherently IMPLEMENTATION DEFINED, whether an implementation uses the WFI mechanism is IMPLEMENTATION DEFINED. If it does, the `WFI` instruction forces the suspension of execution, and of all associated bus activity.

The control logic that does this also tracks the activity on the bus interfaces of the PE, so that when the PE has completed all current operations and any associated bus activity has completed, it can signal to an external power controller that there is no ongoing bus activity.

However, the PE must continue to process memory-mapped and external debug interface accesses to debug registers when in the WFI state. The indication of idle state to the system normally only applies to the non-debug functional interfaces used by the PE, not the debug interfaces.

If the OS Double Lock control is implemented and OSDLR_EL1.DLK is 1, the PE must not signal this idle state to the control logic unless it can also guarantee that the debug interface is idle. For more information about the OS Double Lock, see *Debug behavior when the OS Double Lock is locked* on page H6-7039.

Note

In a PE that implements separate Core and Debug power domains, the debug interface referred to in this section is the interface between the Core and Debug power domains, since the signal to the power controller indicates that the Core power domain is idle. For more information about the power domains see *Power domains and debug* on page H6-7029.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The exact nature of this interface is IMPLEMENTATION DEFINED, but the use of Wait For Interrupt as the only architecturally-defined mechanism that completely suspends execution makes it very suitable as the preferred powerdown entry mechanism.

Pseudocode description of Wait For Interrupt

The `WaitForInterrupt()` pseudocode function optionally suspends execution until a WFI wake-up event or reset occurs, or until some earlier time if the implementation chooses.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D1.17 Self-hosted debug

The Armv8-A architecture supports both of the following:

Self-hosted debug

The PE itself hosts a debugger. The debugger programs the PE to generate *debug exceptions*. Debug exceptions are accommodated in the Armv8-A Exception model.

External debug

The PE is controlled by an external debugger. The debugger programs the PE to generate *Debug events*, that cause the PE to enter *Debug state*. In Debug state, the PE is halted.

This section describes self-hosted debug. It includes:

- *Debug exceptions*.
- *The PSTATE debug mask bit, D*.

For external debug, see Part H *External Debug*.

D1.17.1 Debug exceptions

Debug exceptions occur during normal program flow, if a debugger has programmed the PE to generate them.

For example, a software developer might use a debugger contained in an operating system to debug an application. To do this, the debugger might enable one or more debug exceptions.

The possible debug exceptions are:

- Breakpoint Instruction exceptions.
- Breakpoint exceptions.
- Watchpoint exceptions.
- Vector Catch exceptions.
- Software Step exceptions.

Chapter D2 *AArch64 Self-hosted Debug* describes these in detail for AArch64.

For the PE to generate a debug exception requires that:

- The debug exception is enabled. *The debug exception enable controls* on page D2-2428 gives the controls for the different debug exceptions.
- Debug exceptions are enabled from the current Exception level and Security state. See *Enabling debug exceptions from the current Exception level* on page D2-2431.

Debug exceptions are synchronous exceptions, and are accommodated in the Armv8 Exception model.

Note

Breakpoints and Watchpoints can cause entry to Debug state instead of causing debug exceptions. See Chapter H1 *About External Debug*.

D1.17.2 The PSTATE debug mask bit, D

As with all other exceptions, when a debug exception is taken, software must take care to avoid generating another instance of an exception within the exception handler, to avoid recursive entry into the exception handler and loss of return state.

To help avoid this, the Armv8 architecture provides a debug exception mask bit, PSTATE.D, that can mask Watchpoint, Breakpoint, and Software Step exceptions when the target Exception level is the current Exception level.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

PSTATE.D is set to 1 on taking an exception. This means that while handling an exception in AArch64 state, Watchpoint, Breakpoint, and Software Step exceptions are masked. This prevents recursive entry at the Exception level that debug exceptions are targeted to.

When execution is in AArch64 state, debug exceptions are also masked implicitly when the target Exception level is lower than the current Exception level.

When the target Exception level is higher than the current Exception level, debug exceptions cannot be masked by PSTATE.D.

Because debug exceptions are synchronous, the architecture requires that debug exceptions are not generated when PSTATE.D is 1. By preventing debug exception generation, debug exceptions cannot be taken at a subsequent time when the Process state D mask bit is cleared to 0.

Note

This differs from the behavior for interrupts, where the PSTATE.{A, I, F} mask has the effect of preventing the interrupt from being taken, but instead the interrupt remains pending.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D1.18 Event monitors

The Armv8-A architecture supports the following non-invasive architectural components that allow for event monitoring:

Performance Monitors

The Performance Monitors have a wide feature set, flexible selection of counted events, and are read/write in operation. See *The Performance Monitors Extension*.

Activity Monitors

The Activity Monitors have a narrow feature set, limited selection of counted events, and are read-only in operation. See *The Activity Monitors Extension*.

D1.18.1 The Performance Monitors Extension

The System registers provide access to a Performance Monitors Unit (PMU), defined as the OPTIONAL Performance Monitors Extension to the architecture, a non-invasive debug resource that provides information about the operation of the PE. The PMU provides:

- A 64-bit cycle counter.
- An IMPLEMENTATION DEFINED number of event counters. If ARMv8.5-PMU is implemented, the event counters are 64-bit unsigned counters, otherwise the event counters are 32-bit event counters.

  Each event counter can be configured to count occurrences of a specified event. The events that can be counted are:

  — Architectural and microarchitectural events that are likely to be consistent across many microarchitectures. The PMU architecture uses event numbers to identify an event, and the PMU specification defines which event number must be used for each of these architectural and microarchitectural events.

  — Implementation-specific events. The PMU specification reserves event numbers for implementation-specific events. See Appendix K3 *Recommendations for Performance Monitors Event Numbers for IMPLEMENTATION DEFINED Events*.

For more information, see Chapter D7 *The Performance Monitors Extension*.

D1.18.2 The Activity Monitors Extension

When the OPTIONAL Activity Monitors Extension is implemented, the System registers provide access to controls and counters for the Activity Monitors Unit (AMU). For more information, see Chapter D8 *The Activity Monitors Extension*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D1.19 Interprocessing

Interprocessing is the term used to describe moving between the AArch64 and AArch32 Execution states.

The Execution state can change only on a change of Exception level. This means that the Execution state can change only on taking an exception to a higher Exception level, or returning from an exception to a lower Exception level.

On taking an exception to a higher Exception level, the Execution state either:

- Remains unchanged.
- Changes from AArch32 state to AArch64 state.

On returning from an exception to a lower Exception level, the Execution state either:

- Remains unchanged.
- Changes from AArch64 state to AArch32 state.

Note

If, on taking or returning from an exception, the Exception level remains the same, the Execution state cannot change.

For the description of:

- Exception entry to an Exception level using AArch64, see *Exception entry* on page D1-2339.
- Exception return from an Exception level using AArch64 state, see *Exception return* on page D1-2350.
- Exception return to AArch32 state, see *Exception return to an Exception level using AArch32* on page G1-5725.

  Note

  The description in *Handling exceptions that are taken to an Exception level using AArch32* on page G1-5703 is outside the scope of interprocessing, because such exceptions must have been taken from an Exception level that is using AArch32, and therefore there is no change of Execution state.

The following sections describe the behavior associated with interprocessing.

- *Register mappings between AArch32 state and AArch64 state*.
- *State of the general-purpose registers on taking an exception to AArch64 state* on page D1-2416.
- *SPSR, ELR, and AArch64 SP relationships on changing Execution state* on page D1-2418.

D1.19.1 Register mappings between AArch32 state and AArch64 state

This section defines the architectural mappings between AArch32 state registers and AArch64 state registers.

The mappings describe:

- For exceptions taken from AArch32 state to AArch64 state, where the AArch32 register content is found.
- For exception returns from AArch64 state to AArch32 state, how the AArch32 register content is derived.

The general model is:

- The AArch32 register contents are situated in the bottom 32 bits of the AArch64 registers.
- In AArch32 state, the upper 32 bits of AArch64 registers are inaccessible and are ignored.

Note

System software that executes in AArch64 state, such as an OS or Hypervisor, can use these mappings for context save and restore, or to interpret and modify the AArch32 registers of an application or virtual machine.

For more information see the following subsections:

- *Mapping of the general-purpose registers between the Execution states* on page D1-2407.
- *Mapping of the SIMD and floating-point registers between the Execution states* on page D1-2408.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *Mapping of the System registers between the Execution states* on page D1-2409.

Mapping of the general-purpose registers between the Execution states

Table D1-29 shows how each of the AArch32 general-purpose registers, R0-R12, SP, and LR, including the banked copies of these registers, maps to an AArch64 general-purpose register. A register in the *AArch64 register* column of the table provides the AArch64 view of the corresponding register in the *AArch32 register* column.

Note

For some exceptions, the exception syndrome given in the ESR_ELx identifies one or more register numbers from the issued instruction that generated the exception. Where the exception is taken from an Exception level using AArch32 these register numbers give the AArch64 view of the register. For example, if an exception is taken from AArch32 Abort mode, and the faulting instruction specified R14, the ESR_ELx.ISS field would report this using the EC value `0b10100`, because register X20 provides the AArch64 view of LR_abt. which is the copy of R14 used in Abort mode.

Table D1-29 General-purpose register mapping between AArch32 state and AArch64 state

AArch32 register	AArch64 register
R0	X0
R1	X1
R2	X2
R3	X3
R4	X4
R5	X5
R6	X6
R7	X7
R8_usr	X8
R9_usr	X9
R10_usr	X10
R11_usr	X11
R12_usr	X12
SP_usr	X13
LR_usr	X14
SP_hyp	X15
LR_irq	X16
SP_irq	X17
LR_svc	X18
SP_svc	X19
LR_abt	X20

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-29 General-purpose register mapping between AArch32 state and AArch64 state

AArch32 register	AArch64 register
SP_abt	X21
LR_und	X22
SP_und	X23
R8_fiq	X24
R9_fiq	X25
R10_fiq	X26
R11_fiq	X27
R12_fiq	X28
SP_fiq	X29
LR_fiq	X30

Note

For a description of the banking of AArch32 general-purpose registers R8-R12, SP, and LR, see *AArch32 general-purpose registers, the PC, and the Special-purpose registers* on page G1-5691.

Mapping of the SIMD and floating-point registers between the Execution states

Table D1-30 shows the mapping between the AArch64 V registers and the AArch32 Q registers.

Table D1-30 SIMD and floating-point register mapping between AArch64 state and AArch32 state

AArch64 register	AArch32 register
V0	Q0
V1	Q1
V2	Q2
. . .	. . .
V15	Q15

The AArch64 registers V16-V31 are not accessible from AArch32 state.

The mapping between the V, D, and S registers in AArch64 state is not the same as the mapping between the Q, D, and S registers in AArch32 state:

- In AArch64 state, there are:
  - 32 128-bit V registers, V0-V31.
  - 32 64-bit D registers, D0-D31.
  - 32 32-bit S registers, S0-S31.

  A smaller register occupies the least-significant bytes of the corresponding larger register. For example, S5 is the least-significant word of D5 and V5. Figure D1-3 on page D1-2409 shows this mapping.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

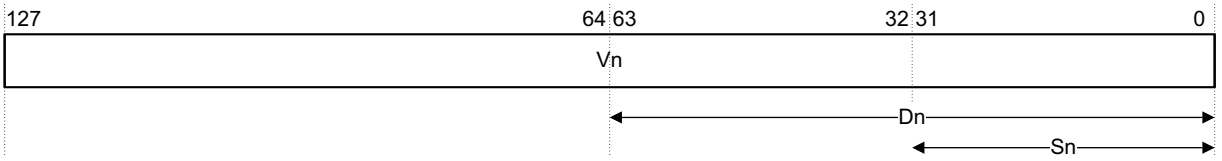


Figure D1-3 AArch64 state SIMD and floating-point register mappings

- In AArch32 state, there are:
  - 16 128-bit Q registers, Q0-Q15.
  - 32 64-bit D registers, D0-D31.
  - 32 32-bit S registers, S0-S31.

  Smaller registers are packed into larger registers. Figure D1-4 shows this mapping.

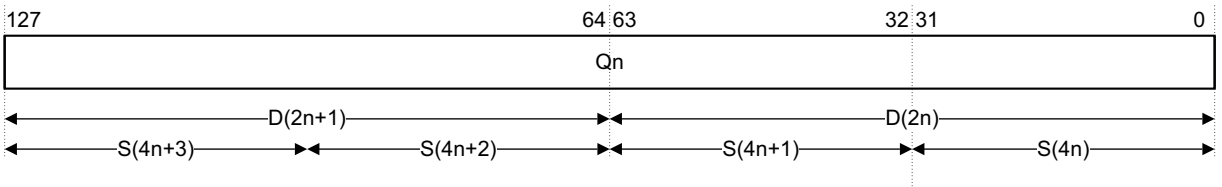


Figure D1-4 AArch32 state SIMD and floating-point register mappings

In AArch32 state:

- There are no S registers that correspond to Q8-Q15.
- D16-D31 pack into Q8-Q15. For example, D16 and D17 pack into Q8.

Note

A consequence of this mapping is that if software executing in AArch64 state interprets D or S registers from AArch32 state, it must unpack the D or S registers from the V registers before it uses them.

Mapping of the System registers between the Execution states

Armv8 architecturally defines the relationship between the AArch64 System registers and the AArch32 System registers, to allow supervisory code such as a hypervisor, that is executing in AArch64 state, to save, restore, and interpret the System registers belonging to a lower Exception level that is using AArch32.

Any modifications made to AArch32 System registers affects only those parts of those AArch64 registers that are mapped to the AArch32 System registers. Bits[63:32] of AArch64 registers, where they are not mapped to AArch32 registers, are unchanged by AArch32 state execution.

Note

This model is different to the model for the general-purpose registers described in *Mapping of the general-purpose registers between the Execution states* on page D1-2407. In this model, there are several cases where two AArch32 System registers are packed into a single AArch64 System register.

When EL3 is implemented and is using AArch32, some System registers are banked between the two Security states. When a register is banked in this way, there is an instance of the register in Secure state, and another instance of the register in Non-secure state. In Table D1-31 on page D1-2410 these banked registers are identified by footnote[a]. This banking is not supported when EL3 is using AArch64 or if EL3 is not implemented. This means that when EL3 is implemented and is using AArch64, exactly the same registers are accessed in the following states:

- Secure EL1 with EL1 using AArch32.
- Non-secure EL1 with EL1 using AArch32.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This means that, architecturally, it is not possible to determine whether an AArch64 register is mapped onto the Secure instance of the corresponding AArch32 register, or onto the Non-secure instance of that register. When EL3 is using AArch64, the interrupt asserted by the AArch64 CNTP_* timer is the same interrupt as is asserted by the Non-secure AArch32 CNTP_* timer when EL3 is using AArch32.

Note

Although the architecture does not require this, because it is not architecturally visible, Arm expects that implementations will map many of the AArch64 registers for use by EL3 to the Secure instances of the banked AArch32 registers, and will map many of the AArch64 registers for use by EL1 to the Non-secure instances of the banked AArch32 registers. However, if EL2 and EL3 are implemented and both support use of AArch32, this is not possible for the following registers:

IFAR This is because when EL3 is using AArch32, HIFAR is an alias of the Secure IFAR.

DFAR This is because when EL3 is using AArch32, HDFAR is an alias of the Secure DFAR.

Table D1-31 shows the mappings between the writable AArch64 System registers and the AArch32 System registers.

Table D1-31 Mapping of writable AArch64 System registers to the AArch32 System registers

AArch64 register	AArch32 register
ACTLR_EL1[31:0]	ACTLR[a]
ACTLR_EL1[63:32]	ACTLR2[a] if implemented
AFSR0_EL1[31:0]	ADFSR[a]
AFSR1_EL1[31:0]	AIFSR[a]
AMAIR_EL1[31:0]	AMAIR0[a]
AMAIR_EL1[63:32]	AMAIR1[a]
CONTEXTIDR_EL1[31:0]	CONTEXTIDR[a]
CPACR_EL1[31:0]	CPACR
CSSELR_EL1[31:0]	CSSELR[a]
DACR32_EL2[31:0]	DACR[a]
FAR_EL1[31:0]	DFAR[a]
ESR_EL1[31:0]	DFSR[a]
HACR_EL2[31:0]	HACR
ACTLR_EL2[31:0]	HACTLR
ACTLR_EL2[63:32]	HACTLR2 if implemented
AFSR0_EL2[31:0]	HADFSR
AFSR1_EL2[31:0]	HAIFSR
AMAIR_EL2[31:0]	HAMAIR0
AMAIR_EL2[63:32]	HAMAIR1
CPTR_EL2[31:0]	HCPTR
HCR_EL2[31:0]	HCR

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-31 Mapping of writable AArch64 System registers to the AArch32 System registers

AArch64 register	AArch32 register
HCR_EL2[63:32]	HCR2
MDCR_EL2[31:0]	HDCR
FAR_EL2[31:0]	HDFAR
FAR_EL2[63:32]	HIFAR
MAIR_EL2[31:0]	HMAIR0
MAIR_EL2[63:32]	HMAIR1
HPFAR_EL2[31:0]	HPFAR
SCTLR_EL2[31:0]	HSCTLR
ESR_EL2[31:0]	HSR
HSTR_EL2[31:0]	HSTR
TCR_EL2[31:0]	HTCR
TPIDR_EL2[31:0]	HTPIDR
TTBR0_EL2[47:1]	HTTBR
VBAR_EL2[31:0]	HVBAR
FAR_EL1[63:32]	IFAR[a]
IFSR32_EL2[31:0]	IFSR[a]
MAIR_EL1[63:32]	NMRR or MAIR1[a]
PAR_EL1[63:0]	PAR[a]
MAIR_EL1[31:0]	PRRR or MAIR0[a]
RMR_EL1[31:0]	RMR (at EL1)
RMR_EL2[31:0]	HRMR
RMR_EL3[31:0]	RMR (at EL3)
SCTLR_EL1[31:0]	SCTLR[a]
SDER32_EL3[31:0]	SDER
TPIDR_EL1[31:0]	TPIDRPRW[a]
TPIDRRO_EL0[31:0]	TPIDRURO[a]
TPIDR_EL0[31:0]	TPIDRURW[a]
TCR_EL1[31:0]	TTBCR[a]
TCR_EL1[63:32]	TTBCR2[a] if implemented
TTBR0_EL1[63:0]	TTBR0[a]
TTBR1_EL1[63:0]	TTBR1[a]
VBAR_EL1[31:0]	VBAR[a]

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-31 Mapping of writable AArch64 System registers to the AArch32 System registers

AArch64 register	AArch32 register
VMPIDR_EL2[31:0]	VMPIDR
VPIDR_EL2[31:0]	VPIDR
VTCR_EL2[31:0]	VTCR
VTTBR_EL2[63:0]	VTTBR
Timer registers	
CNTFRQ_EL0[31:0]	CNTFRQ
CNTHCTL_EL2[31:0]	CNTHCTL
CNTHP_CTL_EL2[31:0]	CNTHP_CTL
CNTHP_CVAL_EL2[63:0]	CNTHP_CVAL
CNTHP_TVAL_EL2[31:0]	CNTHP_TVAL
CNTHPS_CTL_EL2[31:0]	CNTHPS_CTL
CNTHPS_CVAL_EL2[31:0]	CNTHPS_CVAL
CNTHPS_TVAL_EL2[31:0]	CNTHPS_TVAL
CNTKCTL_EL1[31:0]	CNTKCTL
CNTP_CTL_EL0[31:0]	CNTP_CTL[a]
CNTP_CVAL_EL0[63:0]	CNTP_CVAL[a]
CNTP_TVAL_EL0[31:0]	CNTP_TVAL[a]
CNTPCT_EL0[63:0]	CNTPCT
CNTV_CTL_EL0[31:0]	CNTV_CTL
CNTV_CVAL_EL0[63:0]	CNTV_CVAL
CNTV_TVAL_EL0[31:0]	CNTV_TVAL
CNTHV_CTL_EL2[63:0]	CNTHV_CTL
CNTHV_CVAL_EL2[63:0]	CNTHV_CVAL
CNTHV_TVAL_EL2[63:0]	CNTHV_TVAL
CNTHVS_CTL_EL2[31:0]	CNTHVS_CTL
CNTHVS_CVAL_EL2[63:0]	CNTHVS_CVAL
CNTHVS_TVAL_EL2[63:0]	CNTHVS_TVAL
CNTVCT_EL0[63:0]	CNTVCT
CNTVOFF_EL2[63:0]	CNTVOFF
Debug System registers	
DBGAUTHSTATUS_EL1[31:0]	DBGAUTHSTATUS
DBGBCR<n>_EL1[31:0]	DBGBCR<n>

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-31 Mapping of writable AArch64 System registers to the AArch32 System registers

AArch64 register	AArch32 register
DBGBVR<n>_EL1[31:0]	DBGBVR<n>
DBGBVR<n>_EL1[63:32]	DBGBXVR<n>
DBGCLAIMCLR_EL1[31:0]	DBGCLAIMCLR
DBGCLAIMSET_EL1[31:0]	DBGCLAIMSET
DBGDTR_EL0[63:32]	DBGDTRRXint
DBGDTR_EL0[31:0]	DBGDTRTXint
DBGDTRRX_EL0[31:0]	DBGDTRRXint
DBGDTRTX_EL0[31:0]	DBGDTRRXint
DBGPRCR_EL1[31:0]	DBGPRCR
DBGVCR32_EL2[31:0]	DBGVCR
DBGWCR<n>_EL1[31:0]	DBGWCR<n>
DBGWVR<n>_EL1[31:0]	DBGWVR<n>
ID_DFR0_EL1[31:0]	ID_DFR0
MDCCSR_EL0[b][30:29]	DBGDSCRint[b]
MDCR_EL2[31:0]	HDCR
MDRAR_EL1[63:0]	DBGDRAR
MDSCR_EL1[b][31:0]	DBGDSCRext[b]
OSDLR_EL1[31:0]	DBGOSDLR
OSDTRRX_EL1[b][31:0]	DBGDTRRXext[b]
OSDTRTX_EL1[b][31:0]	DBGDTRTXext[b]
OSECCR_EL1[31:0]	DBGOSECCR
OSLAR_EL1[31:0]	DBGOSLAR
OSLSR_EL1[31:0]	DBGOSLSR
SDER32_EL3[31:0]	SDER
Performance Monitors System registers	
PMCCNTR_EL0[31:0]	PMCCNTR (MRC/MCR)
PMCEID0_EL0[31:0]	PMCEID0
PMCEID0_EL0[63:32]	PMCEID2
PMCEID1_EL0[31:0]	PMCEID1
PMCEID1_EL0[63:32]	PMCEID3
PMCNTENCLR_EL0[31:0]	PMCNTENCLR
PMCNTENSET_EL0[31:0]	PMCNTENSET

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-31 Mapping of writable AArch64 System registers to the AArch32 System registers

AArch64 register	AArch32 register
PMCR_EL0[31:0]	PMCR
PMEVCNTR<n>_EL0[31:0]	PMEVCNTR<n>
PMEVTYPER<n>_EL0[31:0]	PMEVTYPER<n>
PMINTENCLR_EL1[31:0]	PMINTENCLR
PMINTENSET_EL1[31:0]	PMINTENSET
PMSELR_EL0[31:0]	PMSELR
PMSWINC_EL0[31:0]	PMSWINC
PMUSERENR_EL0[31:0]	PMUSERENR
PMXEVCNTR_EL0[31:0]	PMXEVCNTR
PMXEVTYPER_EL0[31:0]	PMXEVTYPER
Activity Monitors System registers	
AMCNTENCLR0_EL0[31:0]	AMCNTENCLR0
AMCNTENCLR1_EL0[31:0]	AMCNTENCLR1
AMCNTENSET0_EL0[31:0]	AMCNTENSET0
AMCNTENSET1_EL0[31:0]	AMCNTENSET1
AMCR_EL0[31:0]	AMCR
AMEVCNTR0<n>_EL0[63:0]	AMEVCNTR0<n>
AMEVCNTR1<n>_EL0[63:0]	AMEVCNTR1<n>
AMEVTYPER1<n>_EL0[31:0]	AMEVTYPER1<n>
RAS System registers	
DISR_EL1[31:0]	DISR
ERRIDR_EL1[31:0]	ERRIDR
ERRSELR_EL1[31:0]	ERRSELR
ERXADDR_EL1[31:0]	ERXADDR
ERXADDR_EL1[63:32]	ERXADDR2
ERXCTLR_EL1[31:0]	ERXCTLR
ERXCTLR_EL1[63:32]	ERXCTLR2
ERXFR_EL1[31:0]	ERXFR
ERXFR_EL1[63:32]	ERXFR2
ERXMISC0_EL1[31:0]	ERXMISC0
ERXMISC0_EL1[63:32]	ERXMISC1
ERXMISC1_EL1[31:0]	ERXMISC2

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-31 Mapping of writable AArch64 System registers to the AArch32 System registers

AArch64 register	AArch32 register
ERXMISC1_EL1[63:32]	ERXMISC3
ERXMISC2_EL1[31:0]	ERXMISC4
ERXMISC2_EL1[63:32]	ERXMISC5
ERXMISC3_EL1[31:0]	ERXMISC6
ERXMISC3_EL1[63:32]	ERXMISC7
ERXSTATUS_EL1[31:0]	ERXSTATUS
VDISR_EL2[31:0]	VDISR
VSESR_EL2[31:0]	VDFSR

a. AArch32 registers that are banked if EL3 is using AArch32.

b. These registers have overlapping register content. One or more bits of one register appear in the other register.

There are a small number of AArch32 System registers that are not mapped to any AArch64 System registers. The AArch64 registers listed in Table D1-32 can be used to access these from a higher Exception level that is using AArch64. The registers shown in the table are UNDEFINED if EL1 cannot use AArch32.

Table D1-32 AArch64 registers for accessing registers that are only used in AArch32 state

AArch32 register	Register for access from AArch64 state	Short description
DACR	DACR32_EL2	Domain Access Control Register
DBGVCR	DBGVCR32_EL2	Debug Vector Catch Register
FPEXC	FPEXC32_EL2	Floating-Point Exception Control Register
IFSR	IFSR32_EL2	Instruction Fault Status Register
SDER	SDER32_EL3	AArch32 Secure Debug Enable Register

Table D1-33 shows the AArch64 System registers that allow access from AArch64 state to the AArch32 ID registers. These AArch64 registers are UNKNOWN if no Exception level can use AArch32.

Table D1-33 AArch64 registers that access the AArch32 ID registers

AArch32 register	Register for access from AArch64 state	Short description
ID_AFR0	ID_AFR0_EL1	AArch32 Auxiliary Feature Register 0
ID_DFR0	ID_DFR0_EL1	AArch32 Debug Feature Register 0
ID_ISAR0	ID_ISAR0_EL1	EL1, AArch32 Instruction Set Attribute Register 0
ID_ISAR1	ID_ISAR1_EL1	EL1, AArch32 Instruction Set Attribute Register 1
ID_ISAR2	ID_ISAR2_EL1	EL1, AArch32 Instruction Set Attribute Register 2
ID_ISAR3	ID_ISAR3_EL1	EL1, AArch32 Instruction Set Attribute Register 3
ID_ISAR4	ID_ISAR4_EL1	EL1, AArch32 Instruction Set Attribute Register 4
ID_ISAR5	ID_ISAR5_EL1	EL1, AArch32 Instruction Set Attribute Register 5

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-33 AArch64 registers that access the AArch32 ID registers (continued)

AArch32 register	Register for access from AArch64 state	Short description
ID_MMFR0	ID_MMFR0_EL1	AArch32 Memory Model Feature Register 0
ID_MMFR1	ID_MMFR1_EL1	AArch32 Memory Model Feature Register 1
ID_MMFR2	ID_MMFR2_EL1	AArch32 Memory Model Feature Register 2
ID_MMFR3	ID_MMFR3_EL1	AArch32 Memory Model Feature Register 3
ID_MMFR4	ID_MMFR4_EL1	AArch32 Memory Model Feature Register 4
ID_PFR0	ID_PFR0_EL1	AArch32 PE Feature Register 0
ID_PFR1	ID_PFR1_EL1	AArch32 PE Feature Register 1

D1.19.2 State of the general-purpose registers on taking an exception to AArch64 state

When an exception is taken from AArch32 state to AArch64 state, the state of a general-purpose register depends on whether, immediately before the exception, the register was accessible from AArch32 state, as follows:

If the general-purpose register was accessible from AArch32 state

The upper 32 bits either become zero, or hold the value that the same architectural register held before any AArch32 execution. The choice between these two options is IMPLEMENTATION DEFINED, and might vary dynamically within an implementation. Correspondingly, software must regard the value as being a CONSTRAINED UNPREDICTABLE choice between these two values.

This behavior applies regardless of whether any execution occurred at the Exception level that was using AArch32. That is, this behavior applies even if AArch32 state was entered by an exception return from AArch64 state, and another exception was immediately taken to AArch64 state without any instruction execution in AArch32 state.

Which general-purpose registers have their upper 32 bits affected in this way depends on both:

- The AArch64 state target Exception level.
- The values of both:
  - SCR_EL3.RW.
  - HCR_EL2.RW or HCR.RW, where HCR.RW is a notional bit that is RES0.

Table D1-34 shows which general-purpose registers can have their upper 32 bits set to zero.

Table D1-34 General-purpose registers that can have their upper 32 bits set to zero on taking an exception to AArch64 state from AArch32 state

SCR_EL3.RW	HCR_EL2.RW or HCR.RW[a]	Registers when the target Exception level is:		
		EL3	EL2	EL1
0	0	X0-X30	-[b]	-[b]
0	1	-[c]	-[c]	-[c]
1	0	X0-X14, X16-X30	X0-X14, X16-X30	-[b]
1	1	X0-X14	X0-X14	X0-X14

a. HCR.RW is a notional bit that is RES0.
b. The RW bit values are not valid for the targeted Exception level.
c. Not valid because the RW bit values would imply that EL2 is AArch32 and EL1 is AArch64.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

If EL2 is not implemented, or the SCR_EL3.NS or SCR.NS bit prevents its use, then as described in *The effects of supporting fewer than four Exception levels* on page D1-2421, the behavior is consistent with HCR_EL2.RW taking the value of SCR_EL3.RW.

If the general-purpose register was not accessible from AArch32 state

The general rule is that the register retains the state it had before any AArch32 execution.

There is one exception to this rule, that is when taking an exception to EL3 using AArch64 when either EL2 is not implemented or EL1 is in Secure state. In these cases, the X15 register must be treated as if it is accessible when the value of SCR_EL3.RW is 0, and therefore the upper bits of X15 might either be set to zero or retain their previous value.

Which general-purpose registers retain their state depends on both:

- The AArch64 state target Exception level.
- The values of both:
  - SCR_EL3.RW.
  - HCR_EL2.RW or HCR.RW, where HCR.RW is a notional bit that is RES0.

Table D1-35 shows which general-purpose registers can retain their state.

Table D1-35 General-purpose registers that can retain their state on taking an exception to AArch64 from AArch32

SCR_EL3.RW	HCR_EL2.RW or HCR.RW[a]	Registers when the target Exception level is:		
		EL3	EL2	EL1
0	0	None	-[b]	-[b]
0	1	-[c]	-[c]	-[c]
1	0	X15	X15	-[b]
1	1	X15-X30	X15-X30	X15-X30

a. HCR.RW is a notional bit that is RES0.
b. The RW bit values are not valid for the targeted Exception level.
c. Not valid because the RW bit values would imply that EL2 is AArch32 and EL1 is AArch64.

Note

If EL2 is not implemented, or the SCR_EL3.NS bit prevents its use, then as described in *The effects of supporting fewer than four Exception levels* on page D1-2421, the behavior is consistent with HCR_EL2.RW taking the value of SCR_EL3.RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D1.19.3 SPSR, ELR, and AArch64 SP relationships on changing Execution state

Table D1-36 shows the SPSR and ELR registers that are architecturally mapped between AArch32 state and AArch64 state.

Table D1-36 SPSR and ELR mappings between AArch32 state and AArch64 state

AArch32 register	AArch64 register
SPSR_svc	SPSR_EL1
SPSR_hyp	SPSR_EL2
ELR_hyp	ELR_EL2

On exception entry to EL3 using AArch64 state from an Exception level using AArch32 state, when EL2 has been using AArch32 state, the upper 32-bits of ELR_EL2 are either set to zero or they retain the value before the AArch32 state execution. The implementation determines the choice between these two options, and the choice might vary dynamically within an implementation. Therefore, software must regard the upper 32-bits as being UNKNOWN.

On exception entry to an Exception level using AArch64 state from an Exception level using AArch32 state, the AArch64 Stack Pointers and Exception Link Registers associated with an Exception level that are not accessible during execution in AArch32 state at that Exception level, retain the state that they had before the execution in AArch32 state.

The following AArch32 registers are used only during execution in AArch32 state. However, they retain their state when there is execution at EL1 with EL1 using AArch64 state:

- SPSR_abt.
- SPSR_und.
- SPSR_irq.
- SPSR_fiq.

Note

- These registers are accessible during execution in AArch64 state at Exception levels higher than EL1, for context switching.
- If EL1 does not support execution in AArch32 state then these registers are RES0.

On exception entry to an Exception level using AArch64 from an Exception level using AArch32, the AArch64 Stack Pointers and Exception Link Registers associated with an Exception level that are not accessible during AArch32 execution at that Exception level retain the state that they had before AArch32 execution. This applies to the following registers:

- SP_EL0.
- SP_EL1.
- SP_EL2.
- ELR_EL1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D1.20 The effect of implementation choices on the programmers' model

Three of the implementation choices in Armv8 are:

- The number of Exception levels implemented.
- Which Exception levels support AArch32 and which Exception levels support AArch64.
- Whether SIMD and floating-point support is implemented.

The following subsections give more information about how these choices affect the programmers' model:

- *Implication of Exception levels implemented*.
- *Support for Exception levels and Execution states* on page D1-2420.
- *Implementations not including Advanced SIMD and floating-point instructions* on page D1-2420.
- *The effects of supporting fewer than four Exception levels* on page D1-2421.

D1.20.1 Implication of Exception levels implemented

All implementations must include EL0 and EL1.

EL2 and EL3 are optional. The architecture permits all combinations of EL2 and EL3.

See also *Implementations not including Advanced SIMD and floating-point instructions* on page D1-2420 and *The effects of supporting fewer than four Exception levels* on page D1-2421.

For an implementation that includes all of the Exception levels Figure D1-5 shows the implemented Exception levels and the possible Execution states at lower Exception levels when EL3 is using AArch64. Figure D1-5 applies regardless of whether EL3 also supports use of AArch32.

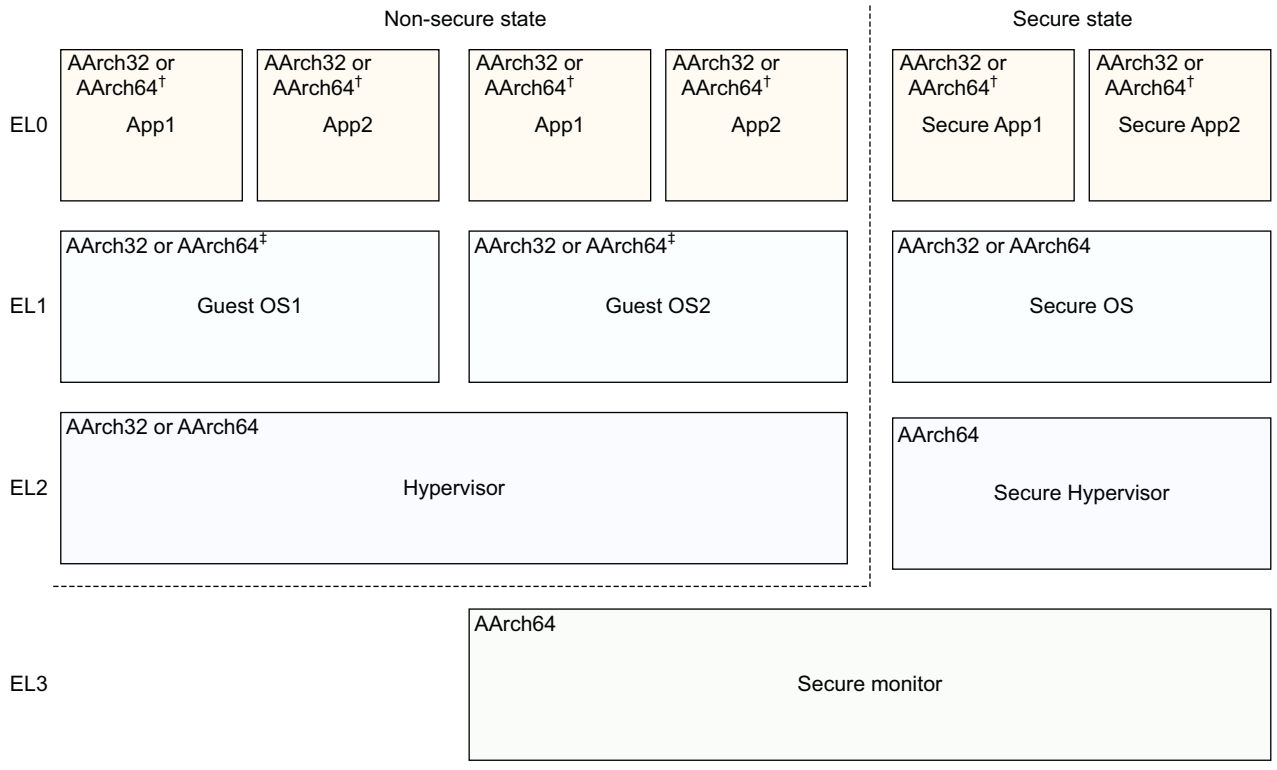


Figure D1-5 Armv8-A security model when EL3 is using AArch64

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The possible combinations of Exception levels are as follows:

- EL0, EL1, and EL2. The implementation supports only a single Security state. This might be either Secure state or Non-secure state.
- EL0, EL1, and EL3. The implementation does not support virtualization. The Exception levels and Execution states depend on whether EL3 is using AArch64 state or AArch32 state, as follows:
  - If EL3 is using AArch64, the Exception levels and Execution states are as shown in Figure D1-5 on page D1-2419 with EL2 removed and no virtualization of EL1 and EL0.
  - If EL3 is using AArch32, the Exception levels and Execution states are as shown in Figure G1-1 on page G1-5680 with EL2 removed and no virtualization of EL1 and EL0.
- EL0 and EL1 only. The implementation supports only a single Security state. This might be either Secure state or Non-secure state, see *Behavior when only EL1 and EL0 are implemented* on page D1-2422.
- EL0, EL1, EL2, and EL3, as described in this section.

For more information, see *The effects of supporting fewer than four Exception levels* on page D1-2421.

D1.20.2 Support for Exception levels and Execution states

Subject to the interprocessing rules defined in *Interprocessing* on page D1-2406, an implementation of the Arm architecture could support:

- AArch64 state only.
- AArch64 and AArch32 states.
- AArch32 state only.

This means the Armv8-A architecture can, potentially, support implementations with very large number of combinations of Execution state and Exception level. Arm intends to license only a subset of the possible combinations.

In an implementation that:

- Supports AArch64 state, all Exception levels are included.
- Has Secure and Non-secure states, EL3 should be implemented.
- Includes all Exception levels, EL3 cannot be included in AArch32 state.

D1.20.3 Implementations not including Advanced SIMD and floating-point instructions

In general, Armv8-A requires the inclusion of the Advanced SIMD and floating-point instructions in all instruction sets. Exceptionally, for implementations targeting specialized markets that do not require support for floating-point or use of Advanced SIMD, Arm might produce or license an Armv8-A implementation that does not provide any support for Advanced SIMD and floating-point instructions. In such an implementation:

In AArch64 state

- The CPACR_EL1.FPEN field is RES0.
- The CPTR_EL2.TFP bit is RES1.
- The CPTR_EL3.TFP bit is RES1.
- Each of the ID_AA64PFR0_EL1.{AdvSIMD, FP} fields is `0b1111`.
- The FPEXC32_EL2, FPCR, and FPSR registers are not implemented, and their encodings are UNDEFINED.
- Attempted accesses to Advanced SIMD and floating-point functionality are UNDEFINED. This means:
  - All Advanced SIMD and floating-point instructions are UNDEFINED.
  - Attempts to access the Advanced SIMD and floating-point System registers are UNDEFINED.
- If at least one Exception level supports execution in AArch32 state, the MVFR0_EL1, MVFR1_EL1 and MVFR2_EL1 registers are RAZ. When no Exception level supports execution in AArch32 state these registers are UNKNOWN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In AArch32 state

See *AArch32 implications of not including support for Advanced SIMD and floating-point* on page G1-5772.

D1.20.4 The effects of supporting fewer than four Exception levels

The effect of implementation choices on the programmers' model on page D1-2419 defines the permitted combinations of Exception levels in an Armv8-A implementation.

In every implementation that supports the highest Exception level using either AArch64 state or AArch32 state, an IMPLEMENTATION DEFINED mechanism determines whether the highest implemented Exception level uses AArch64 state or AArch32 state from a Cold reset. Typically, this mechanism is a configuration input. When the highest level is configured to be AArch64 state, then after a Cold reset execution starts at the reset vector in that Exception level.

The unimplemented Exception levels have no effect on execution:

- No interrupts are routed to these Exception levels.
- No traps that target these Exception levels are active
- All systems calls to unimplemented Exception levels from lower Exception levels are treated as UNDEFINED.
- There is no support for address translation from these Exception levels.
- Any exception return that targets an unimplemented Exception level is treated as an illegal exception return as described in *Illegal return events from AArch64 state* on page D1-2351.
- Every accessible register associated with an unimplemented Exception level is RES0 unless the register is associated with the Exception level only to provide the ability to transfer execution to a lower Exception level.

  Note

  If, for example, EL3 is not implemented and EL2 is the highest implemented Exception level, then because none of the EL3 registers are accessible from EL2, the content of those registers is not architecturally visible.

The following subsections give more information about each of the permitted combinations of Exception levels that do not include all Exception levels.

Behavior when EL3 is not implemented

If EL3 is not implemented:

- If EL2 is implemented and Secure EL2 is not implemented, the *Effective value* of SCR_EL3.NS is `0b1`.
- If Secure EL2 is implemented, the *Effective value* of SCR_EL3.EEL2 is `0b1` and the *Effective value* of SCR_EL3.NS is `0b0`.
- If EL2 is not implemented, it is IMPLEMENTATION DEFINED whether the *Effective value* of SCR_EL3.NS is `0b1` or `0b0`.

Behavior when EL2 is not implemented

If EL2 is not implemented and EL3 is implemented:

- If EL1 can use AArch32 then the following registers are not RES0:
  - DACR32_EL2.
  - IFSR32_EL2.
  - FPEXC32_EL2.
  - DBGVCR32_EL2.
- The VMPIDR_EL2 and VPIDR_EL2 behave as follows:
  - Reads of VMPIDR_EL2 return the value of MPIDR_EL1, writes to VMPIDR_EL2 are ignored.
  - Reads of VPIDR_EL2 return the value of MIDR_EL1, writes to VPIDR_EL2 are ignored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Behavior is consistent with the HCR_EL2.RW bit taking the value of the SCR_EL3.RW bit for all purposes other than reading the HCR_EL2.
- Virtual interrupts are disabled.
- The following address translation and TLB invalidation instructions are UNDEFINED:
  — AT S1E2R and AT S1E2W.
  — TLBI VAE2, TLBI VALE2, TLBI VAE2IS, TLBI VALE2IS, TLBI VAE2OS, TLBI VALE2OS, TLBI ALLE2, TLBI ALLE2IS, TLBI ALLE2OS, TLBI RVAE2, TLBI RVALE2, TLBI RVAE2IS, TLBI RVALE2IS, TLBI RVAE2OS, TLBI RVALE2OS.

  Note

  No other TLB or address translation instructions become UNDEFINED with this combination of Exception levels.

- The SCR_EL3.HCE bit is RES0.

If EL2 is not implemented, regardless of whether EL3 is implemented:

- The *Effective value* of CNTHCTL_EL2[1:0] is `0b11`.
- The *Effective value* of MDCR_EL2.HPMN is the value of PMCR_EL0.N.

Behavior when only EL1 and EL0 are implemented

If EL3 and EL2 are not implemented, it is IMPLEMENTATION DEFINED whether the *Effective value* of the SCR_EL3.NS bit is `0b1` or `0b0`.

This means that if the PE is part of a system that supports two Security states:

- When the *Effective value* of the SCR_EL3.NS bit is `0b1`, the PE can only access Non-secure memory.
- When the *Effective value* of the SCR_EL3.NS bit is `0b0`, the PE can access both Secure memory and Non-secure memory.

If the *Effective value* of the SCR_EL3.NS bit is `0b0`, then:

- The *Effective value* of MDCR_EL3.{EPMAD, EDAD} is {0b1, 0b1}.
- The *Effective value* of MDCR_EL3.{SPME, NSPB} is {0b1, 0b01}.
- The *Effective value* of MDCR_EL3.SPD32 is `0b11`.

If EL3 is not implemented, regardless of whether EL2 is implemented, the *Effective value* of MDCR_EL3.STE is the inverse of the *Effective value* of SCR_EL3.NS.

Note

- The behavior described in this subsection still applies if EL1 is configured to use AArch32.
- The implementation can provide a configuration input that determines, from reset, the *Effective value* of the SCR_EL3.NS bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter D2
AArch64 Self-hosted Debug

When the PE is using self-hosted debug, it generates *debug exceptions*. This chapter describes the AArch64 self-hosted debug exception model. It is organized as follows:

Introductory information

- *About self-hosted debug* on page D2-2424.
- *The debug exception enable controls* on page D2-2428.

The debug Exception model

- *Routing debug exceptions* on page D2-2429.
- *Enabling debug exceptions from the current Exception level* on page D2-2431.
- *The effect of powerdown on debug exceptions* on page D2-2433.
- *Summary of the routing and enabling of debug exceptions* on page D2-2434.
- *Pseudocode description of debug exceptions* on page D2-2436.

The debug exceptions

- *Breakpoint Instruction exceptions* on page D2-2437.
- *Breakpoint exceptions* on page D2-2439.
- *Watchpoint exceptions* on page D2-2457.
- *Vector Catch exceptions* on page D2-2471.
- *Software Step exceptions* on page D2-2472.

Synchronization requirements

The behavior of self-hosted debug after changes to System registers, or after changes to the authentication interface, but before a *Context synchronization event* guarantees the effects of the changes:

- *Synchronization and debug exceptions* on page D2-2485.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D2.1 About self-hosted debug

Self-hosted debug supports debugging through the generation and handling of *debug exceptions*, that are taken using the exception model described in Chapter D1 *The AArch64 System Level Programmers' Model*. This section introduces some terms that are used in describing self-hosted debug, and then introduces the debug exceptions. See:

- *Definition of a debugger in the context of self-hosted debug*.
- *Context ID and Process ID*.
- *About debug exceptions*.

D2.1.1 Definition of a debugger in the context of self-hosted debug

Within this chapter, *debugger* means that part of an operating system, or higher level of system software, that handles debug exceptions and programs the Debug System registers. An operating system with rich application environments might provide debug services that support a debugger user interface executing at EL0. From the architectural perspective, the debug services are the debugger.

D2.1.2 Context ID and Process ID

A CONTEXTIDR_ELx identifies the current *Context ID*, that is used by:

- The debug logic, for breakpoint and watchpoint matching.
- Implemented trace logic, to identify the current process.

In AArch64 state, the CONTEXTIDR_ELx has a single field, PROCID, that is defined as the *Process Identifier* (Process ID). Therefore, in AArch64 state, the Context ID and Process ID are identical.

D2.1.3 About debug exceptions

Debug exceptions occur during normal program flow if a debugger has programmed the PE to generate them. For example, a software developer might use a debugger contained in an operating system to debug an application. To do this, the debugger enables one or more debug exceptions. The debug exceptions that can be generated in stage 1 of an AArch64 translation regime are:

- *Breakpoint Instruction exceptions* on page D2-2425.
- *Breakpoint exceptions* on page D2-2425, generated by hardware breakpoints.
- *Watchpoint exceptions* on page D2-2425, generated by hardware watchpoints.
- *Software Step exceptions* on page D2-2426.

In addition, debug exceptions generated in an AArch32 translation regime might be routed to EL2 using AArch64. See *Routing debug exceptions* on page D2-2429. Chapter G2 describes the debug exceptions that can be generated in an AArch32 translation regime.

Vector Catch exceptions are exceptions that cannot be generated in an AArch64 translation regime but can be generated in stage 1 of an AArch32 translation regime and routed to EL2 using AArch64. *Vector Catch exceptions* on page D2-2471 describes the behavior for this case.

The PE can only generate a particular debug exception when both:

1. Debug exceptions are enabled from the current Exception level and Security state.

   See *Enabling debug exceptions from the current Exception level* on page D2-2431. Breakpoint Instruction exceptions are always enabled from the current Exception level and Security state.

2. A debugger has enabled that particular debug exception.

   All of the debug exceptions except for Breakpoint Instruction exceptions have an enable control contained in the MDSCR_EL1. See *The debug exception enable controls* on page D2-2428.

Note

If *halting is allowed* and EDSCR.HDE is 1, hardware breakpoints and watchpoints cause entry to Debug state instead of causing debug exceptions. In Debug state, the PE is halted.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the definition of halting is allowed, see *Halting allowed and halting prohibited* on page H2-6927.

The following list summarizes each of the debug exceptions:

Breakpoint Instruction exceptions

Breakpoint instructions generate these. Breakpoint instructions are instructions that software developers can use to cause exceptions at particular points in the program flow.

The breakpoint instruction in the A64 instruction set is `BRK #<immediate>`. Whenever one of these is committed for execution, the PE takes a Breakpoint Instruction exception.

PE behavior

Breakpoint Instruction exceptions cannot be masked. The PE takes Breakpoint Instruction exceptions regardless of both of the following:

- The current Exception level.
- The current Security state.

For more information, see *Breakpoint Instruction exceptions* on page D2-2437.

Breakpoint exceptions

The Armv8-A architecture provides 2-16 hardware breakpoints. These can be programmed to generate Breakpoint exceptions based on particular instruction addresses, or based on particular PE contexts, or both.

For example, a software developer might program a hardware breakpoint to generate a Breakpoint exception whenever the instruction with address `0x1000` is committed for execution.

The Armv8-A architecture supports the following types of hardware breakpoint for use in stage 1 of an AArch64 translation regime:

- Address.
  - Comparisons are made with the virtual address of each instruction in the program flow.
- Context:
  - Context ID Match. Matches with the Context ID held in the CONTEXTIDR_EL1.
  - VMID Match. Matches with the VMID value held in the VTTBR_EL2.
  - Context ID and VMID Match. Matches with both the Context ID and the VMID value.

An Address breakpoint can link to a Context breakpoint, so that the Address breakpoint only generates a Breakpoint exception if the PE is in a particular context when the address match occurs.

A breakpoint generates a Breakpoint exception whenever an instruction that causes a match is committed for execution.

PE behavior

If halting is allowed and EDSCR.HDE is 1, hardware breakpoints cause entry to Debug state. That is, they halt the PE. See Chapter H2 *Debug State*.

Otherwise:

- If debug exceptions are enabled, hardware breakpoints cause Breakpoint exceptions.
- If debug exceptions are disabled, hardware breakpoints are ignored.

For more information, see *Breakpoint exceptions* on page D2-2439.

Watchpoint exceptions

The Armv8-A architecture provides 2-16 hardware watchpoints. These can be programmed to generate Watchpoint exceptions based on accesses to particular data addresses, or based on accesses to any address in a data address range.

For example, a software developer might program a hardware watchpoint to generate a Watchpoint exception on an access to any address in the data address range `0x1000 - 0x101F`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A hardware watchpoint can link to a hardware breakpoint if the hardware breakpoint is a *Linked Context* type. In this case, the watchpoint only generates a Watchpoint exception if the PE is in a particular context when the data address match occurs.

The smallest data address size that a watchpoint can be programmed to match on is a byte. A single watchpoint can be programmed to match on one or more bytes.

A watchpoint generates a Watchpoint exception whenever an instruction that initiates an access that causes a match is committed for execution.

PE behavior

If halting is allowed and EDSCR.HDE is 1, hardware watchpoints cause entry to Debug state. That is, they halt the PE. See Chapter H2 *Debug State*.

Otherwise:

- If debug exceptions are enabled, hardware watchpoints cause Watchpoint exceptions.
- If debug exceptions are disabled, hardware watchpoints are ignored.

For more information, see *Watchpoint exceptions* on page D2-2457.

Vector Catch exceptions

These are not generated in an AArch64 translation regime. They can only be generated in an AArch32 translation regime. See *Vector Catch exceptions* on page D2-2471.

Software Step exceptions

Software step is a resource that a debugger can use to make the PE single-step instructions.

For example, by using software step, debugger software executing at a higher Exception level can debug software executing at a lower Exception level, by making it single-step instructions.

After the software being debugged has single-stepped an instruction, the PE takes a Software Step exception.

PE behavior

Software step can only be used by a debugger executing in an Exception level that is using AArch64. However, the instruction stepped might be executed in either Execution state, and therefore Software Step exceptions can be taken from either Execution state.

If debug exceptions are enabled, Software Step exceptions can be generated.

If debug exceptions are disabled, software step is inactive.

For more information, see *Software Step exceptions* on page D2-2472.

Table D2-1 summarizes PE behavior and shows the location of the pseudocode for each of the debug exceptions.

Table D2-1 PE behavior and pseudocode for each of the debug exceptions

Debug exception	PE behavior if debug exceptions are:		Pseudocode
	Enabled	Disabled	
Breakpoint Instruction exceptions	Takes the exception	Takes the exception	*Pseudocode description of Breakpoint Instruction exceptions* on page D2-2438
Breakpoint exceptions	Takes the exception[a]	Ignored	*Pseudocode description of Breakpoint exceptions taken from AArch64 state* on page D2-2456

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D2-1 PE behavior and pseudocode for each of the debug exceptions (continued)

Debug exception	PE behavior if debug exceptions are:		Pseudocode
	Enabled	Disabled	
Watchpoint exceptions	Takes the exception[a]	Ignored	*Pseudocode description of Watchpoint exceptions taken from AArch64 state* on page D2-2470
Vector Catch exceptions	Takes the exception	Ignored	*Pseudocode description of Vector Catch exceptions* on page G2-5875
Software Step exceptions	Takes the exception	Not applicable[b]	*Pseudocode description of Software Step exceptions* on page D2-2484

a. If halting is allowed and EDSCR.HDE is 1, hardware breakpoints and watchpoints cause the PE to enter Debug state instead of causing debug exceptions. See Chapter H2 *Debug State*.

b. Software Step is inactive if debug exceptions are disabled. No Software Step exceptions can be generated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D2.2 The debug exception enable controls

The enable controls for each debug exception are as follows:

Breakpoint Instruction exceptions

None. Breakpoint Instruction exceptions are always enabled.

Breakpoint exceptions

MDSCR_EL1.MDE, plus an enable control for each breakpoint, DBGBCR<n>_EL1.E.

Watchpoint exceptions

MDSCR_EL1.MDE, plus an enable control for each watchpoint, DBGWCR<n>_EL1.E.

Vector Catch exceptions

MDSCR_EL1.MDE.

Software Step exceptions

MDSCR_EL1.SS.

In addition, for all debug exceptions other than Breakpoint Instruction exceptions, software must configure the controls that enable debug exceptions from the current Exception level and Security state. See *Enabling debug exceptions from the current Exception level* on page D2-2431.

The PE cannot take a debug exception if debug exceptions are disabled from either the current Exception level or the current Security state.

Breakpoint Instruction exceptions are always enabled from the current Exception level and Security state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D2.3 Routing debug exceptions

Debug exceptions are enabled and routed according to the following controls:

- MDCR_EL2.TDE.
- HCR_EL2.TGE.
- MDCR_EL3.SDD.
- The Security state when the exception is taken.
- The Exception level where the exception is taken.

Breakpoint Instructions are enabled in some situations where other Debug exceptions are disabled.

If the OS Lock is locked, or if DoubleLockStatus() == TRUE, a Debug exception cannot be taken.

Note

If EL2 is not implemented, the *Effective value* of HCR_EL2.TGE is 0 and the *Effective value* of MDCR_EL2.TDE is 0. Throughout this section, references to the values of these fields are to the *Effective values* of the fields.

If EL3 is not implemented, and the implementation is a Secure state only implementation, the *Effective value* of MDCR_EL3.SDD is 0.

The routing of debug exceptions is as follows:

Table D2-2 shows when debug exceptions are enabled from the current Security state.

Table D2-2 Whether debug exceptions are enabled from the current Security state

Current Security state	Breakpoint Instruction exceptions	All other debug exceptions
Non-secure	Enabled	Enabled
Secure	Enabled	Disabled if MDCR_EL3.SDD is 1. See *Disabling debug exceptions from Secure state* on page D2-2431. Otherwise enabled.

Debug exceptions taken when EL2 is implemented and enabled in the current Security state

The routing of debug exceptions taken depends on the values of MDCR_EL2.TDE and HCR_EL2.TGE:

If the *Effective value* of {MDCR_EL2.TDE, HCR_EL2.TGE} is not {0, 0}

Debug exceptions are routed to EL2, ELD is EL2.

Otherwise

Debug exceptions behave as follows:

- Debug exceptions taken from EL1 and EL0 are routed to EL1. ELD is EL1
- Breakpoint Instruction exceptions taken from EL2 are routed to EL2.
- All other debug exceptions are disabled from EL2 using AArch64.

When EL3 is implemented

Breakpoint Instruction exceptions taken from EL3 are routed to EL3.

All other debug exceptions are disabled from EL3 using AArch64.

Otherwise Debug exceptions are routed to EL1.

This means that, for all debug exceptions, the *Debug target Exception level*, EL_D, is either EL1 or EL2. When executing in the same exception level as ELD, see *Enabling debug exceptions from the current Exception level* on page D2-2431.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D2-3, Table D2-4, and Table D2-5 show the routing of debug exceptions. In these tables:

NS Means the *Effective value* of SCR_EL3.NS.

EEL2 Means the *Effective value* of SCR_EL3.EEL2. If ARMv8.4-SecEL2 is not implemented, this is 0.

TDE or TGE Means the logical OR of the *Effective value* of MDCR_EL2.TDE and the *Effective value* of HCR_EL2.TGE.

(EL*x*) Means EL_D is EL*x*. However:

- All debug exceptions other than Breakpoint Instruction exceptions are disabled from this Exception level.
- Breakpoint Instruction exceptions taken when executing in this Exception level are routed to the same Exception level. This may not be the same as the ELD Exception level.

EL*x* Means EL_D is EL*x*.

Table D2-3 Routing when both EL3 and EL2 are implemented

NS	EEL2	TDE or TGE	EL_D when executing in:			
			EL0	EL1	EL2	EL3
0	0	x	EL1	EL1	n/a	(EL1)
0	1	0	EL1	EL1	(EL1)	(EL1)
0	1	1	EL2	EL2	EL2	(EL2)
1	x	0	EL1	EL1	(EL1)	(EL1)
1	x	1	EL2	EL2	EL2	(EL2)

Table D2-4 Routing when EL3 is implemented and EL2 is not implemented

EL_D when executing in:		
EL0	EL1	EL3
EL1	EL1	(EL1)

Table D2-5 Routing when EL3 is not implemented and EL2 is implemented

TDE	EL_D when executing in:		
	EL0	EL1	EL2
0	EL1	EL1	(EL1)
1	EL2	EL2	EL2

D2.3.1 Pseudocode description of routing debug exceptions

`DebugTarget()` returns the current debug target Exception level.

`DebugTargetFrom()` returns the debug target Exception level for the specified Security state.

These functions are described in Chapter J1 *Armv8 Pseudocode*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D2.4 Enabling debug exceptions from the current Exception level

A debug exception can only be taken if all of the following are true:

- The OS Lock is unlocked.
- `DoubleLockStatus()` == FALSE.
- The debug exception is enabled from the current Exception level.
- The debug exception is enabled from the current Security state.

Table D2-6 shows when debug exceptions are enabled from the current Exception level. In the table, EL_D is the Exception level that Table D2-3 on page D2-2430 defines.

Table D2-6 Whether debug exceptions are enabled from the current Exception level

Current Exception level	Breakpoint Instruction exceptions	All other debug exceptions
Any Exception level that is higher than EL_D[a]	Enabled	Disabled
EL_D	Enabled	Disabled if either of the following is true: • The Local (kernel) Debug Enable bit, MDSCR_EL1.KDE, is 0. • The Debug exception mask bit, PSTATE.D, is 1. Otherwise enabled. This means that a debugger must explicitly enable these debug exceptions from EL_D by setting MDSCR_EL1.KDE to 1 and PSTATE.D to 0.
Any Exception level that is lower than EL_D	Enabled	Enabled

a. This includes EL3. EL3 is always higher than EL_D.

Note

PSTATE.D is set to 1 at reset and on exception entry.

D2.4.1 Disabling debug exceptions from Secure state

If EL3 is implemented, software executing at EL3 can set the *Secure Debug Disable* bit, MDCR_EL3.SDD, to 1 to disable all debug exceptions taken from AArch64 Secure state other than Breakpoint Instruction exceptions.

The Armv8-A architecture does not support disabling debug in Non-secure state.

Note

- If the boot software executed when reset is deasserted sets MDCR_EL3.SDD to 1, software operating at EL3 never has to switch the debug registers between Secure state and Non-secure state.
- The PE cannot take a debug exception unless it is enabled from the current Exception level. See Table D2-6.
- If either the OS Lock or the OS Double Lock is locked, debug exceptions other than Breakpoint Instruction exceptions are disabled.
- If EL3 and EL2 are not implemented, and the implementation is a Secure state only implementation, the PE behaves as if MDCR_EL3.SDD is 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D2.4.2 Pseudocode description of enabling debug exceptions

`AArch64.GenerateDebugException`s() determines whether debug exceptions other than Breakpoint Instruction exceptions are enabled from the current Exception level and Security state.

`AArch64.GenerateDebugExceptionsFrom`() determines whether debug exceptions other than Breakpoint Instruction exceptions are enabled from the specified Exception level and Security state.

These functions are described in Chapter J1 *Armv8 Pseudocode*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D2.5 The effect of powerdown on debug exceptions

Debug OS Save and Restore sequences on page H6-7035 describes the *powerdown save routine* and the *restore routine*.

When executing either routine, software must use the OS Lock to disable generation of all of the following:

- Breakpoint exceptions.
- Watchpoint exceptions.
- Vector Catch exceptions.
- Software Step exceptions.

This is because the generation of these exceptions depends on the state of the debug registers, and the state of the debug registers might be lost over these routines.

If the OS Lock is unlocked, and `DoubleLockStatus()`== FALSE, debug exceptions other than Breakpoint Instruction exceptions are enabled.

If OS Lock is locked, or if `DoubleLockStatus()`==TRUE, debug exceptions other than Breakpoint Instruction exceptions are disabled.

Breakpoint Instruction exceptions are enabled regardless of the state of the OS Lock and the OS Double Lock.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D2.6 Summary of the routing and enabling of debug exceptions

Behavior is as follows:

Breakpoint Instruction exceptions

These are always enabled, regardless of the current Exception level and Security state. A Breakpoint Instruction exception taken from EL3 is always routed to EL3. A Breakpoint Instruction exception taken from EL2 is routed to EL2. A Breakpoint Instruction exception taken from EL0 or EL1 is always routed to ELD.

All other debug exceptions

Table D2-7 shows the valid combinations of MDCR_EL3.SDD, MDCR_EL2.TDE, MDSCR_EL1.KDE, and PSTATE.D, and for each combination shows where these exceptions are enabled from and where they are taken to.

In the table:

Lock Means the value of (OSLSR_EL1.OSLK == '1' || `DoubleLockStatus()`).

NS Means the *Effective value* of SCR_EL3.NS.

SDD Means the *Effective value* of MDCR_EL3.SDD. See *Disabling debug exceptions from Secure state* on page D2-2431.

EEL2 Means the *Effective value* of SCR_EL3.EEL2. If ARMv8.4-SecEL2 is not implemented, this is 0.

TGE Means the value of HCR_EL2.TGE. If EL2 is not implemented, the PE behaves as if this is 0.

TDE Means the value of MDCR_EL2.TDE. If EL2 is not implemented, the PE behaves as if this is 0.

KDE Means the value of MDSCR_EL1.KDE.

D Means the value of PSTATE.D.

n/a Means not applicable. The PE cannot be executing at this Exception level.

- Means that debug exceptions are disabled from that Exception level.

Table D2-7 Routing of Breakpoint, Watchpoint, Software Step, and Vector Catch exceptions

Debug state	Lock	NS	SDD	EEL2	TGE	TDE	KDE	D	EL_D when enabled from:			
									EL0	EL1	EL2	EL3
Yes	X	X	X	X	X	X	X	X	-	-	-	-
No	TRUE	X	X	X	X	X	X	X	-	-	-	-
	FALSE	0	1	X	X	X	X	X	-	-	-	-
			0	0	X	X	0	X	EL1	-	n/a	-
							1	0	EL1	EL1	n/a	-
								1	EL1	-	n/a	-
				1	0	0	0	X	EL1	-	-	-
							1	0	EL1	EL1	-	-
								1	EL1	-	-	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D2-7 Routing of Breakpoint, Watchpoint, Software Step, and Vector Catch exceptions (continued)

Debug state	Lock	NS	SDD	EEL2	TGE	TDE	KDE	D	EL_D when enabled from:			
									EL0	EL1	EL2	EL3
No	FALSE	0	0	1	0	1	0	X	EL2	EL2	-	-
							1	0	EL2	EL2	EL2	-
								1	EL2	EL2	-	-
					1	X	0	X	EL2	n/a	-	-
							1	0	EL2	n/a	EL2	-
								1	EL2	n/a	-	-
		1	X	X	0	0	0	X	EL1	-	-	-
							1	0	EL1	EL1	-	-
								1	EL1	-	-	-
						1	0	X	EL2	EL2	-	-
							1	0	EL2	EL2	EL2	-
								1	EL2	EL2	-	-
					1	X	0	X	EL2	n/a	-	-
							1	0	EL2	n/a	EL2	-
								1	EL2	n/a	-	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D2.7 Pseudocode description of debug exceptions

`AArch64.DebugFault()` returns a `FaultRecord` object that indicates that a memory access has generated a debug exception:

The `AArch64.Abort()` function processes `FaultRecord` objects, as described in *Abort exceptions* on page D4-2529, and generates a debug exception.

`AArch64.Abort()` calls one of the following:

- `AArch64.BreakpointException()`.
- `AArch64.WatchpointException()`.
- `AArch64.VectorCatchException()`.
- `AArch64.SoftwareStepException()`.

These functions are defined in Chapter J1 *Armv8 Pseudocode*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D2.8 Breakpoint Instruction exceptions

This section describes Breakpoint Instruction exceptions in an AArch64 translation regime.

The PE is using an AArch64 translation regime when it is executing either:

- In an Exception level that is using AArch64.
- At EL0 using AArch32 when EL1 is using AArch64.

For software executing in an Exception level that is using AArch64, a Breakpoint Instruction exception results from the execution of an A64 `BRK` instruction. However, within the AArch64 EL1&0 translation regime, executing a T32 or A32 `BKPT` instruction at EL0 using AArch32 generates a Breakpoint Instruction exception.

For more information about the T32 and A32 `BKPT` instructions, see:

- *Breakpoint instruction in the A32 and T32 instruction sets* on page G2-5827.
- *BKPT instructions as the first instruction in an IT block* on page G2-5828.

The following subsections describe Breakpoint Instruction exceptions in an AArch64 translation regime:

- *About Breakpoint Instruction exceptions*.
- *Breakpoint instructions*.
- *Exception syndrome information and preferred return address* on page D2-2438.
- *Pseudocode description of Breakpoint Instruction exceptions* on page D2-2438.

D2.8.1 About Breakpoint Instruction exceptions

A *breakpoint* is an event that results from the execution of an instruction, which is based on either:

- The instruction address, the PE context, or both. This type of breakpoint is called a *hardware breakpoint.*
- The instruction itself. That is, the instruction is a *breakpoint instruction*. These can be included in the program that the PE executes. This type of breakpoint is called a *software breakpoint*.

Breakpoint Instruction exceptions, that this section describes, are software breakpoints. *Breakpoint exceptions* on page D2-2439 describes hardware breakpoints.

There is no enable control for Breakpoint Instruction exceptions. They are always enabled, and cannot be masked.

A Breakpoint Instruction exception is generated whenever a breakpoint instruction is committed for execution, regardless of all of the following:

- The current Exception level.
- The current Security state.
- Whether the *debug target Exception level*, EL_D, is using AArch64 or AArch32.

Note

- The debug target Exception level, EL_D, is the Exception level that debug exceptions are targeting. *Routing debug exceptions* on page D2-2429 describes how EL_D is derived.
- Debuggers using breakpoint instructions must be aware of the Armv8 rules for concurrent modification and execution of instructions. See *Concurrent modification and execution of instructions* on page B2-120.

D2.8.2 Breakpoint instructions

The breakpoint instruction in the A64 instruction set is `BRK #<immediate>`. It is unconditional.

For details of the instruction encoding, see *BRK* on page C6-841.

The breakpoint instruction in the A32 and T32 instruction sets is `BKPT #<immediate>`.

For more information about the A32 and T32 breakpoint instruction, see *Breakpoint instruction in the A32 and T32 instruction sets* on page G2-5827.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D2.8.3 Exception syndrome information and preferred return address

See the following:

- *Exception syndrome information*.
- *Preferred return address*.

Exception syndrome information

On taking a Breakpoint Instruction exception, the PE records information about the exception in the *Exception Syndrome Register* (ESR) at the Exception level the exception is taken to. The ESR used is one of:

- ESR_EL1.
- ESR_EL2.
- ESR_EL3.

Note

Breakpoint Instruction exceptions are the only debug exception that can be taken to EL3 using AArch64.

Table D2-8 shows the information that the PE records.

Table D2-8 Information recorded in the ESR_ELx

ESR_ELx field	Information recorded in ESR_EL1, ESR_EL2, or ESR_EL3.
Exception Class, EC	Whether the breakpoint instruction was executed in AArch64 state or AArch32 state. The PE sets this to: • `0x3C` for an A64 `BRK` instruction. • `0x38` for an A32 or T32 `BKPT` instruction.
Instruction Length, IL	The PE sets this to: • 0 for a 16-bit T32 `BKPT` instruction. • 1 for an A64 `BRK` instruction, or an A32 `BKPT` instruction.
Instruction Specific Syndrome, ISS	**ISS[24:16]** RES0. **ISS[15:0]** The PE copies the instruction Comment field value into here, zero extended as necessary.

Note

- If debug exceptions are routed to EL2, it is the exception that is routed, not the instruction that is trapped. Therefore, if a Breakpoint Instruction exception is routed to EL2, ESR_EL2.EC is set to the same value as if the exception was taken to EL1.
- For information about how debug exceptions can be routed to EL2, see *Routing debug exceptions* on page D2-2429.

Preferred return address

The preferred return address is the address of the breakpoint instruction, not the next instruction. This is different to the behavior of other exception-generating instructions, like `SVC`.

D2.8.4 Pseudocode description of Breakpoint Instruction exceptions

`AArch64.SoftwareBreakpoint()` generates a Breakpoint Instruction exception that is taken to AArch64 state.

This function is defined in Chapter J1 *Armv8 Pseudocode*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D2.9 Breakpoint exceptions

This section describes Breakpoint exceptions in stage 1 of an AArch64 translation regime.

The PE is using an AArch64 translation regime when it is executing either:

- In an Exception level that is using AArch64.
- At EL0 using AArch32 when EL1 is using AArch64.

This section contains the following subsections:

- *About Breakpoint exceptions*.
- *Breakpoint types and linking of breakpoints* on page D2-2440.
- *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page D2-2448.
- *Breakpoint instruction address comparisons* on page D2-2450.
- *Breakpoint context comparisons* on page D2-2452.
- *Breakpoint usage constraints* on page D2-2453.
- *Preferred return address* on page D2-2456.
- *Pseudocode description of Breakpoint exceptions taken from AArch64 state* on page D2-2456.

D2.9.1 About Breakpoint exceptions

A *breakpoint* is an event that results from the execution of an instruction, which is based on either:

- The instruction address, the PE context, or both. This type of breakpoint is called a *hardware breakpoint.*
- The instruction itself. That is, the instruction is a *breakpoint instruction*. These can be included in the program that the PE executes. This type of breakpoint is called a *software breakpoint*.

Breakpoint exceptions are generated by *Breakpoint debug events*. Breakpoint debug events are generated by hardware breakpoints. Software breakpoints are described in *Breakpoint Instruction exceptions* on page D2-2437.

An implementation can include between 2-16 hardware breakpoints. ID_AA64DFR0_EL1.BRPs shows how many are implemented.

To use an implemented hardware breakpoint, a debugger programs the following registers for the breakpoint:

- The *Breakpoint Control Register*, DBGBCR<n>_EL1. This contains controls for the breakpoint, for example an enable control.
- The *Breakpoint Value Register*, DBGBVR<n>_EL1. This holds the value used for breakpoint matching, that is one of:
  - An instruction virtual address.
  - A Context ID.
  - A VMID value.
  - A concatenation of both a Context ID value and a VMID value.

These registers are numbered, so that:

- DBGBCR1_EL1 and DBGBVR1_EL1 are for breakpoint number one.
- DBGBCR2_EL1 and DBGBVR2_EL1 are for breakpoint number two.
- …
- …
- DBGBCR<n>_EL1 and DBGBVR<n>_EL1 are for breakpoint number n.

A debugger can link a breakpoint that is programmed with an address and a breakpoint that is programmed with anything other than an address together, so that a Breakpoint debug event is only generated if both breakpoints match.

For each instruction in the program flow, all of the breakpoints are tested. When a breakpoint is tested, it generates a Breakpoint debug event if all of the following are true:

- The breakpoint is enabled. That is, the breakpoint enable control for it, DBGBCR<n>_EL1.E, is 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The conditions specified in the DBGBCR<n>_EL1 are met.
- The comparison with the value held in the DBGBVR<n>_EL1 is successful.
- If the breakpoint is linked to another breakpoint, the comparisons made by that other breakpoint are also successful.
- The instruction is committed for execution.

If all of these conditions are met, the breakpoint generates the Breakpoint debug event regardless of the following:
- Whether the instruction passes its Condition code check.
- The instruction type.

If halting is allowed and EDSCR.HDE is 1, Breakpoint debug events cause entry to Debug state.

Otherwise, if debug exceptions are:
- Enabled, Breakpoint debug events generate Breakpoint exceptions.
- Disabled, Breakpoint debug events are ignored.

Note

The remainder of this Breakpoint exceptions section, including all subsections, describes breakpoints as generating Breakpoint exceptions.

However, the behavior described also applies if breakpoints are causing entry to Debug state.

The debug exception enable controls on page D2-2428 describes the enable controls for Breakpoint debug events.

D2.9.2 Breakpoint types and linking of breakpoints

Each implemented breakpoint is one of the following:

- A *context-aware* breakpoint. This is a breakpoint that can be programmed to generate a Breakpoint exception on any one of the following:
  - An instruction address match.
  - A Context ID match, with the value held in the CONTEXTIDR_EL1.
  - A VMID match, with the VMID value held in the VTTBR_EL2.
  - Both a Context ID match and a VMID match.
- A breakpoint that is not context-aware. These can only be programmed to generate a Breakpoint exception on an instruction address match.

ID_AA64DFR0_EL1.CTX_CMPs shows how many of the implemented breakpoints are context-aware breakpoints. At least one implemented breakpoint must be context-aware. The context-aware breakpoints are the highest numbered breakpoints.

Any breakpoint that is programmed to generate a Breakpoint exception on an instruction address match is categorized as an *Address breakpoint*. Breakpoints that are programmed to match on anything else are categorized as *Context breakpoints*.

When a debugger programs a breakpoint to be an Address or a Context breakpoint, it must also program that breakpoint so that it is either:
- Used in isolation. In this case, the breakpoint is called an *Unlinked breakpoint*.
- Enabled for linking to another breakpoint. In this case, the breakpoint is called a *Linked breakpoint*.

By linking an Address breakpoint and a Context breakpoint together, the debugger can create a breakpoint pair that only generates a Breakpoint exception if the PE is in a particular context when an instruction address match occurs. For example, a debugger might:

1. Program breakpoint number one to be a *Linked Address Match breakpoint*.
2. Program breakpoint number five to be a *Linked Context ID Match breakpoint*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

3. Link these two breakpoints together. A Breakpoint exception is only generated if both the instruction address matches and the Context ID matches.

The *Breakpoint Type* field for a breakpoint, DBGBCR<n>_EL1.BT, controls the breakpoint type and whether the breakpoint is enabled for linking. If BT[0] is 1, the breakpoint is enabled for linking.

If AArch32 state is implemented, Address breakpoints can be programmed to generate Breakpoint exceptions on addresses that are halfword-aligned but not word-aligned. This makes it possible to breakpoint on T32 instructions. See *Specifying the halfword-aligned address that an Address breakpoint matches on* on page D2-2451.

Note

Stage 1 of an AArch32 translation regimes supports two additional breakpoint types, Unlinked and Linked Address Mismatch breakpoints, BT == `0b0100` and BT == `0b0101`. For information about these, see Chapter G2 *AArch32 Self-hosted Debug*. These types are reserved in stage 1 of an AArch64 translation regime. See *Reserved BT values* on page D2-2453.

Rules for linking breakpoints

The rules for breakpoint linking are as follows:

- Only Linked breakpoint types can be linked.
- Any type of Linked Address breakpoint can link to any type of Linked Context breakpoint. The *Linked Breakpoint Number* field, DBGBCR<n>_EL1.LBN, for the Linked Address breakpoint specifies the particular Linked Context breakpoint that the Linked Address breakpoint links to, and:
  - DBGBCR<n>_EL1.{SSC, HMC, PMC} for the Linked Address breakpoint define the execution conditions that the breakpoint pair generates Breakpoint exceptions for. See *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page D2-2448.
  - DBGBCR<n>_EL1.{SSC, HMC, PMC} for the Linked Context breakpoint are ignored.
- Linked Context breakpoint types can only be linked to. The LBN field for Context breakpoints is therefore ignored.
- Linked Address breakpoints cannot link to watchpoints. The LBN field can therefore only specify another breakpoint.
- If a Linked Address breakpoint links to a breakpoint that is not context-aware, the behavior of the Linked Address breakpoint is CONSTRAINED UNPREDICTABLE. See *Other usage constraints for Address breakpoints* on page D2-2455.
- If a Linked Address breakpoint links to an Unlinked Context breakpoint, the Linked Address breakpoint never generates any Breakpoint exceptions.
- Multiple Linked Address breakpoints can link to a single Linked Context breakpoint.

  Note

  Multiple Linked watchpoints can also link to a single Linked Context breakpoint. *Watchpoint exceptions* on page D2-2457 describes watchpoints.

These rules mean that a single Linked Context breakpoint might be linked to by all, or any combination of, the following:

- Multiple Linked Address Match breakpoints.
- Multiple Linked watchpoints.

Note

If ARMv8.4-NV is implemented, the hypervisor must use the 0b1101, Linked CONTEXTIDR_EL2 Match breakpoint type to guarantee a linked match, see *Interaction with self-hosted and External debug* on page D5-2650.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

It is also possible that a Linked Context breakpoint might have no breakpoints or watchpoints linked to it.

Figure D2-1 shows an example of permitted breakpoint and watchpoint linking.

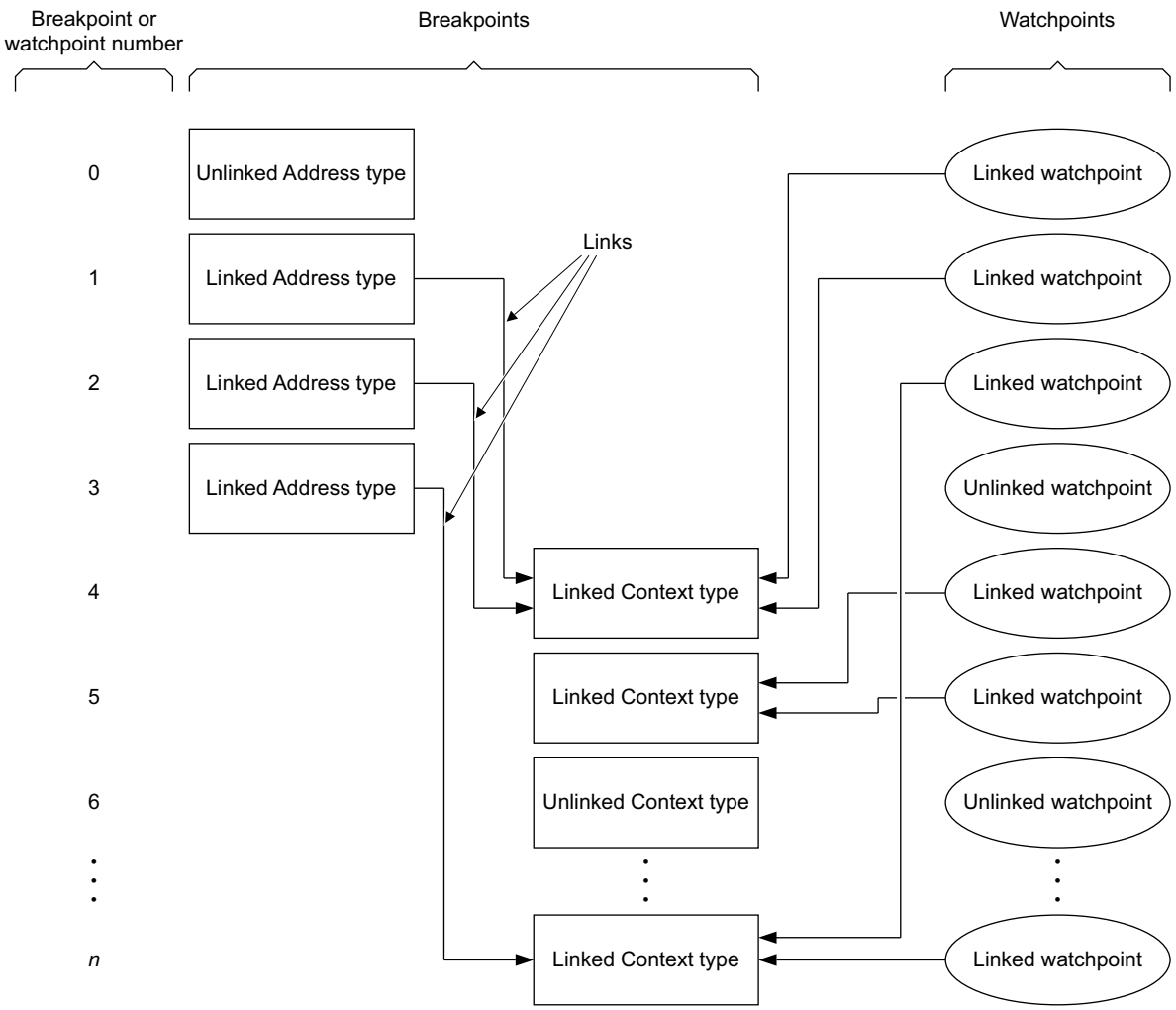


Figure D2-1 The role of linking in Breakpoint and Watchpoint exception generation

In Figure D2-1, each Linked Address breakpoint can only generate a Breakpoint exception if the comparisons made by both it, and the Linked Context breakpoint that it links, to are successful. Similarly, each Linked watchpoint can only generate a Watchpoint exception if the comparisons made by both it, and the Linked Context breakpoint that it links to, are successful.

Breakpoint types defined by DBGBCRn_EL1.BT

The following list provides more detail about each breakpoint type:

`0b0000`, Unlinked Address Match breakpoint

Generation of a Breakpoint exception depends on both:

- DBGBCR<n>_EL1.{SSC, HMC, PMC}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions. See *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page D2-2448.
- A successful address match, as described in *Breakpoint instruction address comparisons* on page D2-2450.

DBGBCR<n>_EL1.LBN for this breakpoint is ignored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b0001`, Linked Address Match breakpoint

Generation of a Breakpoint exception depends on all of the following:

- DBGBCR<n>_EL1.{SSC, HMC, PMC} for this breakpoint. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page D2-2448.
- A successful address match defined by this breakpoint, as described in *Breakpoint instruction address comparisons* on page D2-2450.
- A successful context match defined by the Linked Context breakpoint that this breakpoint links to.

DBGBCR<n>_EL1.LBN for this breakpoint selects the Linked Context breakpoint that this breakpoint links to.

`0b0010`, Unlinked Context ID Match breakpoint

BT == `0b0010` is a reserved value if the breakpoint is not a context-aware breakpoint.

For context-aware breakpoints, generation of a Breakpoint exception depends on both:

- DBGBCR<n>_EL1.{SSC, HMC, PMC}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions. See *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page D2-2448.
- A successful Context ID match, as described in *Breakpoint context comparisons* on page D2-2452.

The value of DBGBVR<n>_EL1.ContextID is compared with the current Context ID.

CONTEXTIDR_EL2 holds the current Context ID when all of:

- The implementation includes ARMv8.1-VHE.
- EL2 is implemented and enabled in the current Security state.
- EL2 using AArch64 and HCR_EL2.E2H is set to 1.
- The PE is executing at EL0 and HCR_EL2.TGE is 1, or the PE is executing at EL2.

Otherwise, CONTEXTIDR_EL1 holds the current Context ID.

DBGBCR<n>_EL1.{LBN, BAS} for this breakpoint are ignored

`0b0011`, Linked Context ID Match breakpoint

BT == `0b0011` is a reserved value if the breakpoint is not a context-aware breakpoint.

For context-aware breakpoints, one of the following applies:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint, then the breakpoint does not generate any Breakpoint exceptions.
- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address breakpoint that links to this breakpoint, see *Breakpoint instruction address comparisons* on page D2-2450.
  - A successful Context ID match defined by this breakpoint, as described in *Breakpoint context comparisons* on page D2-2452.
- Generation of a Watchpoint exception depends on both:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see *Watchpoint data address comparisons* on page D2-2462.
  - A successful Context ID match defined by this breakpoint, as described in *Breakpoint context comparisons* on page D2-2452.

The value of DBGBVR<n>_EL1.ContextID is compared with the current Context ID.

CONTEXTIDR_EL2 holds the current Context ID when all of:

- The implementation includes ARMv8.1-VHE.
- EL2 is implemented and enabled in the current Security state.
- EL2 using AArch64 and HCR_EL2.E2H is set to 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The PE is executing at EL0 and HCR_EL2.TGE is 1, or the PE is executing at EL2.

Otherwise, CONTEXTIDR_EL1 holds the current Context ID.

DBGBCR<n>_EL1.{LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

`0b0100`, Unlinked Address Mismatch breakpoint

BT == `0b0100` is a reserved value in stage 1 of an AArch64 translation regime. See *Reserved BT values* on page D2-2453.

0b0100, Unlinked Address Mismatch breakpoint on page G2-5835 describes the behavior of Address Mismatch breakpoints in stage 1 of an AArch32 translation regime.

`0b0101`, Linked Address Mismatch breakpoint

BT == `0b0101` is a reserved value in stage 1 of an AArch64 translation regime. See *Reserved BT values* on page D2-2453.

0b0101, Linked Address Mismatch breakpoint on page G2-5835 describes the behavior of Address Mismatch breakpoints in stage 1 of an AArch32 translation regime.

`0b0110`, Unlinked CONTEXTIDR_EL1 Match breakpoint

BT == `0b0110` is a reserved value if either:

- The breakpoint is not a context-aware breakpoint.
- The implementation does not include ARMv8.1-VHE.

In an implementation that includes ARMv8.1-VHE, for context-aware breakpoints, generation of a Breakpoint exception depends on both:

- DBGBCR<n>_EL1.{SSC, HMC, PMC}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions.
- A successful Context ID match defined by this breakpoint, as described in *Breakpoint context comparisons* on page D2-2452.

The Context ID check is made against the value in CONTEXTIDR_EL1. The value of DBGBVR<n>_EL1.ContextID is compared with the Context ID value held in CONTEXTIDR_EL1.

Note

The operation of this breakpoint does not depend on the value of HCR_EL2.E2H.

DBGBCR<n>_EL1.{LBN, BAS} for this breakpoint are ignored.

`0b0111`, Linked CONTEXTIDR_EL1 Match breakpoint

BT == `0b0111` is a reserved value if either:

- The breakpoint is not a context-aware breakpoint.
- The implementation does not include ARMv8.1-VHE.

In an implementation that includes ARMv8.1-VHE, for context-aware breakpoints, one of the following applies:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint, then the breakpoint does not generate any Breakpoint exceptions.
- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address match breakpoint that links to this breakpoint, see *Breakpoint instruction address comparisons* on page D2-2450.
  - A successful Context ID match defined by this breakpoint, as described in *Breakpoint context comparisons* on page D2-2452.
- Generation of a Watchpoint exception depends on both:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see *Watchpoint data address comparisons* on page D2-2462.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— A successful Context ID match defined by this breakpoint, as described in *Breakpoint context comparisons* on page D2-2452.

The Context ID check is made against the value in CONTEXTIDR_EL1. The value of DBGBVR<n>_EL1.ContextID is compared with the Context ID value held in CONTEXTIDR_EL1.

Note

The operation of this breakpoint does not depend on the value of HCR_EL2.E2H.

DBGBCR<n>_EL1.{LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

`0b1000`**, Unlinked VMID Match breakpoint**

BT == `0b1000` is a reserved value if either:

- The breakpoint is not a context-aware breakpoint.
- EL2 is not implemented.

For context-aware breakpoints, generation of a Breakpoint exception depends on both:

- DBGBCR<n>_EL1.{SSC, HMC, PMC}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions. See *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page D2-2448.
- A successful VMID match, as described in *Breakpoint context comparisons* on page D2-2452.

DBGBCR<n>_EL1.{LBN, BAS} for this breakpoint are ignored.

`0b1001`**, Linked VMID Match breakpoint**

BT == `0b1000` is a reserved value if either:

- The breakpoint is not a context-matching breakpoint.
- EL2 is not implemented.

For context-aware breakpoints, one of the following applies:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint, then the breakpoint does not generate any Breakpoint exceptions.
- Generation of a Breakpoint exception depends on both:
  - — A successful instruction address match, defined by a Linked Address Match breakpoint that links to this breakpoint. See *Breakpoint instruction address comparisons* on page D2-2450.
  - — A successful VMID match defined by this breakpoint, as described in *Breakpoint context comparisons* on page D2-2452.
- Generation of a Watchpoint exception depends on both:
  - — A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see *Watchpoint data address comparisons* on page D2-2462.
  - — A successful VMID match defined by this breakpoint, as described in *Breakpoint context comparisons* on page D2-2452.

DBGBCR<n>_EL1.{LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

`0b1010`**, Unlinked Context ID and VMID Match breakpoint**

BT == `0b1010` is a reserved value if either:

- The breakpoint is not a context-aware breakpoint.
- EL2 is not implemented.

When EL2 is implemented, for context-aware breakpoints, generation of a Breakpoint exception depends on all of the following:

- DBGBCR<n>_EL1.{SSC, HMC, PMC}. These define the execution conditions that the breakpoint generates a Breakpoint exception for. See *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page D2-2448.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- A successful Context ID match, as described in *Breakpoint context comparisons* on page D2-2452.
- A successful VMID match.

The value of DBGBVR<n>_EL1.ContextID is compared with CONTEXTIDR_EL1.

Breakpoint context comparisons on page D2-2452 describes the requirements for a successful Context ID match and a successful VMID match.

DBGBCR<n>_EL1.{LBN, BAS} for this breakpoint are ignored.

`0b1011`, Linked Context ID and VMID Match breakpoint

BT == `0b1011` is a reserved value if either:

- The breakpoint is not a context-aware breakpoint.
- EL2 is not implemented.

When EL2 is implemented, for context-aware breakpoints, one of the following applies:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint, then the breakpoint does not generate any Breakpoint exceptions.
- Generation of a Breakpoint exception depends on all of the following:
  - A successful instruction address match, defined by a Linked Address breakpoint that links to this breakpoint, see *Breakpoint instruction address comparisons* on page D2-2450.
  - A successful Context ID match defined by this breakpoint, as described in *Breakpoint context comparisons* on page D2-2452.
  - A successful VMID match defined by this breakpoint.
- Generation of a Watchpoint exception depends on all of the following:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see *Watchpoint data address comparisons* on page D2-2462.
  - A successful Context ID match defined by this breakpoint, as described in *Breakpoint context comparisons* on page D2-2452.
  - A successful VMID match defined by this breakpoint.

The value of DBGBVR<n>_EL1.ContextID is compared with CONTEXTIDR_EL1.

Breakpoint context comparisons on page D2-2452 describes the requirements for a successful Context ID match and a successful VMID match by this breakpoint.

DBGBCR<n>_EL1.{LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

`0b1100`, Unlinked CONTEXTIDR_EL2 Match breakpoint

BT == `0b1100` is a reserved value if either:

- The breakpoint is not a context-aware breakpoint.
- ARMv8.1-VHE is not implemented and ARMv8.2-Debug is not implemented, which means the implementation does not include CONTEXTIDR_EL2.

In an implementation in which ARMv8.1-VHE is implemented or ARMv8.2-Debug is implemented, for context-aware breakpoints, generation of a Breakpoint exception depends on both:

- DBGBCR<n>_EL1.{SSC, HMC, PMC}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions.
- A successful CONTEXTIDR_EL2 match, as described in *Breakpoint context comparisons* on page D2-2452.

The Context ID check is made against the value in CONTEXTIDR_EL2. The value of DBGBVR<n>_EL1 is compared with the Context ID value held in CONTEXTIDR_EL2.

Note

The operation of this breakpoint does not depend on the value of HCR_EL2.E2H.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

DBGBCR<n>_EL1.{LBN, BAS} for this breakpoint are ignored.

`0b1101`, Linked CONTEXTIDR_EL2 Match breakpoint

BT == `0b1101` is a reserved value if either:

- The breakpoint is not a context-aware breakpoint.
- ARMv8.1-VHE is not implemented and ARMv8.2-Debug is not implemented, which means the implementation does not include CONTEXTIDR_EL2.

In an implementation in which ARMv8.1-VHE is implemented or ARMv8.2-Debug is implemented, for context-aware breakpoints, either:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint, then the breakpoint does not generate any Breakpoint exceptions.
- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address match breakpoint that links to this breakpoint, see *Breakpoint instruction address comparisons* on page D2-2450.
  - A successful CONTEXTIDR_EL2 match, as described in *Breakpoint context comparisons* on page D2-2452.
- Generation of a Watchpoint exception depends on both:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see *Watchpoint data address comparisons* on page D2-2462.
  - A successful CONTEXTIDR_EL2 match, as described in *Breakpoint context comparisons* on page D2-2452.

The Context ID check is made against the value in CONTEXTIDR_EL2. The value of DBGBVR<n>_EL1 is compared with the Context ID value held in CONTEXTIDR_EL2.

Note

The operation of this breakpoint does not depend on the value of HCR_EL2.E2H.

DBGBCR<n>_EL1.{LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

`0b1110`, Unlinked Full Context ID Match breakpoint

BT == `0b1110` is a reserved value if either:

- The breakpoint is not a context-aware breakpoint.
- ARMv8.1-VHE is not implemented and ARMv8.2-Debug is not implemented, which means the implementation does not include CONTEXTIDR_EL2.

In an implementation in which ARMv8.1-VHE is implemented or ARMv8.2-Debug is implemented, for context-aware breakpoints, generation of a Breakpoint exception depends on both:

- DBGBCR<n>_EL1.{SSC, HMC, PMC}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions.
- A successful Context ID match, as described in *Breakpoint context comparisons* on page D2-2452.

The Context ID check is made against the values in both CONTEXTIDR_EL1 and CONTEXTIDR_EL2. The value of DBGBVR<n>_EL1[31:0] is compared with the Context ID value held in CONTEXTIDR_EL1, and the value of DBGBVR<n>_EL1[63:32] is compared with the Context ID value held in CONTEXTIDR_EL2. Both comparisons must match for the Context ID check.

Note

The operation of this breakpoint does not depend on the value of HCR_EL2.E2H.

DBGBCR<n>_EL1.{LBN, BAS} for this breakpoint are ignored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1111`, Linked Full Context ID Match breakpoint

BT == `0b1111` is a reserved value if either:

- The breakpoint is not a context-aware breakpoint.
- ARMv8.1-VHE is not implemented and ARMv8.2-Debug is not implemented, which means the implementation does not include CONTEXTIDR_EL2.

In an implementation in which ARMv8.1-VHE is implemented or ARMv8.2-Debug is implemented, for context-aware breakpoints, one of the following applies:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint, then the breakpoint does not generate any Breakpoint exceptions.
- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address match breakpoint that links to this breakpoint, see *Breakpoint instruction address comparisons* on page D2-2450.
  - A successful Context ID match, as described in *Breakpoint context comparisons* on page D2-2452.
- Generation of a Watchpoint exception depends on both:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see *Watchpoint data address comparisons* on page D2-2462.
  - A successful Context ID match, as described in *Breakpoint context comparisons* on page D2-2452.

The Context ID check is made against the values in both CONTEXTIDR_EL1 and CONTEXTIDR_EL2. The value of DBGBVR<n>_EL1[31:0] is compared with the Context ID value held in CONTEXTIDR_EL1, and the value of DBGBVR<n>_EL1[63:32] is compared with the Context ID value held in CONTEXTIDR_EL2. Both comparisons must match for the Context ID check.

Note

The operation of this breakpoint does not depend on the value of HCR_EL2.E2H.

DBGBCR<n>_EL1.{LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

Note

See *Reserved DBGBCR<n>_EL1.BT values* on page D2-2453 for the behavior of breakpoints programmed with reserved BT values.

D2.9.3 Execution conditions for which a breakpoint generates Breakpoint exceptions

Each breakpoint can be programmed so that it only generates Breakpoint exceptions for certain execution conditions. For example, a breakpoint might be programmed to generate Breakpoint exceptions only when the PE is executing at EL0 in Secure state.

DBGBCR<n>_EL1.{SSC, HMC, PMC} defines the execution conditions the breakpoint generates Breakpoint exceptions for, as follows:

Security State Control, SSC

Controls whether the breakpoint generates Breakpoint exceptions only in Secure state, only in Non-secure state, or in both Security states.

Note

This is determined by the Security state of the PE, not from the NS attribute returned by the translation of the virtual address on which the breakpoint is set.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Higher Mode Control, HMC, and Privileged Mode Control, PMC

HMC and PMC together control which Exception levels the breakpoint generates Breakpoint exceptions in.

Table D2-9 shows the valid combinations of the values of HMC, SSC, and PMC, and for each combination shows which Exception levels breakpoints generate Breakpoint exceptions in.

In the table:

Y Means that a breakpoint programmed with the values of HMC, SSC, and PMC shown in that row can generate Breakpoint exceptions in that Exception level and Security state.

- Means that a breakpoint programmed with the values of HMC, SSC, and PMC shown in that row cannot generate Breakpoint exceptions in that Exception level and Security state.

For information about which combinations of HMC, SSC and PMC are reserved if an Exception level or Security state are not implemented or enabled, see *Reserved DBGBCR<n>_EL1.{SSC, HMC, PMC} values* on page D2-2454.

Table D2-9 Summary of breakpoint HMC, SSC, and PMC encodings

HMC	SSC	PMC	Security state	EL3[a]	EL2	EL1	EL0
0	00	01	Both	-	-	Y	-
0	00	10		-	-	-	Y
0	00	11		-	-	Y	Y
0	01	01	Non-secure	n/a	-	Y	-
0	01	10		n/a	-	-	Y
0	01	11		n/a	-	Y	Y
0	10	01	Secure	-	-	Y	-
0	10	10		-	-	-	Y
0	10	11		-	-	Y	Y
0	11	00	Secure	-	Y	-	-
0	11	01		-	Y	Y	-
0	11	11		-	Y	Y	Y
1	00	01	Both	Y	Y	Y	-
1	00	11		Y	Y	Y	Y
1	01	00	Non-secure	n/a	Y	-	-
1	01	01		n/a	Y	Y	-
1	01	11		n/a	Y	Y	Y
1	10	00	Secure	Y	-	-	-
1	10	01		Y	Y	Y	-
1	10	11		Y	Y	Y	Y

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D2-9 Summary of breakpoint HMC, SSC, and PMC encodings (continued)

HMC	SSC	PMC	Security state	EL3[a]	EL2	EL1	EL0
1	11	00	Both	-	Y	-	-
1	11	01		-	Y	Y	-
1	11	11		-	Y	Y	Y

a. Debug exceptions are not generated at EL3 using AArch64. This means that these combinations of HMC, SSC, and PMC are only relevant if breakpoints cause entry to Debug state. Self-hosted debuggers must avoid combinations of HMC, SSC, and PMC that generate Breakpoint exceptions at EL3 using AArch64.

All combinations of HMC, SSC, and PMC that this table does not show are reserved. See *Reserved DBGBCR<n>_EL1.{SSC, HMC, PMC} values* on page D2-2454.

D2.9.4 Breakpoint instruction address comparisons

In this subsection, the term `AddrTop` represents the most significant bit of a virtual address used by breakpoint data address comparisons. `AddrTop` is:

- 55, if address tagging is used for the address. See *Address tagging in AArch64 state* on page D5-2534.
- 63, otherwise.

Note

When stage 1 translation is enabled, in AArch64 state, a virtual address has a maximum address width of either 48 bits or, when ARMv8.2-LVA is implemented and the 64KB translation granule is used, 52 bits. Software can configure a smaller address width for a virtual address, see *Input address size* on page D5-2549. Attempting to translate an address that is larger than the configured input address size generates a Translation fault.

When stage 1 translation is disabled, using an address that ia larger than the implemented PA size generates an Address size fault. The implemented PA size is IMPLEMENTATION DEFINED up to 52 bits, see *Physical address size* on page D5-2548.

These faults have a higher priority than breakpoints.

An address comparison is successful if bits [`AddrTop`:2] of the current instruction virtual address are equal to DBGBVR<n>_EL1[`AddrTop`:2].

Note

DBGBVR<n>_EL1 is a 64-bit register. The most significant bits of this register are sign-extension bits. DBGBVR<n>_EL1[1:0] are RES0 and are ignored.

If EL1 is using AArch64 and EL0 is using AArch32, A32 and T32 instructions can be executed in stage 1 of an AArch64 translation regime. In this case, the instruction addresses are zero-extended before comparison with the breakpoint.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Specifying the halfword-aligned address that an Address breakpoint matches on

For Address Match breakpoints, if the implementation supports AArch32 state, a debugger must program the *Byte Address Selection* field, DBGBCR<n>_EL1.BAS.

Table D2-10 Programmable BAS values

BAS	Match instruction at	Constraint for debuggers
`0b0011`	DBGBCR<n>_EL1	Use for T32 instructions.
`0b1100`	DBGBCR<n>_EL1 + 2	Use for T32 instructions.
`0b1111`	DBGBCR<n>_EL1	Use for A64 and A32 instructions.

If the implementation is an AArch64-only implementation, all instructions are word-aligned and DBGBCR<n>_EL1.BAS is RES1.

Figure D2-2 on page D2-2452 shows a summary of when Address Match breakpoints programmed with particular BAS values generate Breakpoint exceptions. The figure contains four parts:

- A column showing the row number, on the left.
- An instruction set and instruction size table.
- A location of instruction figure.
- A BAS field values table, on the right.

To use the figure, read across the rows. For example, row 7 shows that a breakpoint with DBGBCR<n>_EL1.BAS programmed as either `0b0011` or `0b1111` generates Breakpoint exceptions for A64 instructions. A64 instructions are always at word-aligned addresses.

Note

To breakpoint on an A64 instruction, Arm recommends that the debugger programs DBGBCR<n>_EL1.BAS as `0b1111`.

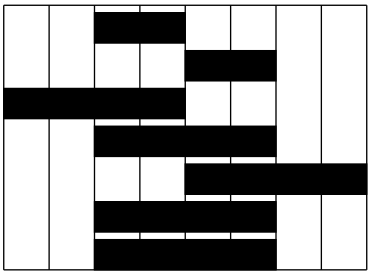
In the figure:

Yes Means that the breakpoint generates a Breakpoint exception.

No Means that the breakpoint does not generate a Breakpoint exception.

UNP Means that is it CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception. See *Other usage constraints for Address breakpoints* on page D2-2455.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta



	Instruction set	Size	Location of instruction[a] -2	-1	0	+1	+2	+3	+4	+5	BAS[3:0] 0b0011	0b1100	0b1111
Row 1	T32	16-bit			■	■					Yes	No	Yes
Row 2		16-bit					■	■			No	Yes	UNP
Row 3	T32	32-bit	■	■	■	■					UNP	No	UNP
Row 4		32-bit			■	■	■	■			Yes	UNP	Yes
Row 5		32-bit					■	■	■	■	No	Yes	UNP
Row 6	A32	32-bit			■	■	■	■			Yes	UNP	Yes
Row 7	A64	32-bit			■	■	■	■			Yes	UNP	Yes

a. 0 means the word-aligned address held in the DBGBVR<n>_EL1[maxAddressSize:2]:00. The other locations a
- -2 means ((DBGBVR<n>_EL1[maxAddressSize:2]:00) – 2).
- -1 means ((DBGBVR<n>_EL1[maxAddressSize:2]:00) – 1).
- ...
- ...
- +5 means ((DBGBVR<n>_EL1[maxAddressSize:2]:00) + 5).

The solid areas show the location of the instruction.

Figure D2-2 Summary of BAS field meanings for Address Match breakpoints

D2.9.5 Breakpoint context comparisons

The breakpoint type defined by DBGBCR<n>_EL1.BT determines what context comparison is required, if any. Table D2-11 shows the BT values that require a comparison, and the match required for the comparison to be successful.

Table D2-11 Breakpoint Context ID and VMID comparison tests

DBGBCR<n>.BT	Test required for successful context comparison
0b001x	• When ARMv8.1-VHE is implemented, EL2 is using AArch64, the *Effective value* of HCR_EL2.E2H is 1, and either the PE is executing at EL0 with HCR_EL2.TGE set to 1, or the PE is executing at EL2, CONTEXTIDR_EL2 must match the DBGBVR<n>_EL1. ContextID value. • Otherwise, CONTEXTIDR_EL1 must match the DBGBVR<n>_EL1.ContextID value.
0b011x	CONTEXTIDR_EL1 must match the DBGBVR<n>_EL1.ContextID value.
0b100x	VTTBR_EL2.VMID must match the DBGBVR<n>_EL1.VMID value.
0b101x	CONTEXTIDR_EL1 must match the DBGBVR<n>_EL1.ContextID value and VTTBR_EL2.VMID must match the DBGBVR<n>_EL1.VMID value.
0b110x	CONTEXTIDR_EL2 must match the DBGBVR<n>_EL1.ContextID2 value, DBGBVR<n>_EL1[63:32].
0b111x	Both: • CONTEXTIDR_EL1 must match the DBGBVR<n>_EL1.ContextID value, DBGBVR<n>_EL1[31:0]. • CONTEXTIDR_EL2 must match the DBGBVR<n>_EL1.ContextID2 value, DBGBVR<n>_EL1[63:32].

No Context ID or VMID comparison is required for other valid DBGBCR<n>.BT values.

Context breakpoints do not generate Breakpoint exceptions when any of:

- The comparison uses the value of CONTEXTIDR_EL1 and any of:
  - The PE is executing at EL3 using AArch64.
  - The PE is executing at EL2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

  - ARMv8.1-VHE is implemented, EL2 is using AArch64, EL2 is enabled in the current Security state, and HCR_EL2.{E2H, TGE} == {1, 1}.

- The comparison uses the value of CONTEXTIDR_EL2 and any of:
  - Neither ARMv8.1-VHE is implemented, nor ARMv8.2-Debug is implemented.
  - If the PE is in Secure state, and either ARMv8.4-SecEL2 is not implemented, or Secure EL2 is disabled.
  - EL2 is using AArch32.
  - EL2 is not implemented.

- The comparison uses the current VMID value and any of:
  - EL2 is not implemented.
  - If the PE is in Secure state, and either ARMv8.4-SecEL2 is not implemented, or Secure EL2 is disabled.
  - The PE is executing at EL2.
  - ARMv8.1-VHE is implemented, EL2 is using AArch64, EL2 is enabled in the current Security state, and HCR_EL2.{E2H, TGE} == {1, 1}.

Note

- For all Context breakpoints, DBGBCR<n>_EL1.BAS is RES1 and is ignored.
- For Linked Context breakpoints, DBGBCR<n>_EL1.{LBN, SSC, HMC, PMC} are RES0 and are ignored.

D2.9.6 Breakpoint usage constraints

See the following sections:

- *Reserved DBGBCR<n>_EL1.BT values*.
- *Reserved DBGBCR<n>_EL1.{SSC, HMC, PMC} values* on page D2-2454.
- *Reserved DBGBCR<n>_EL1.BAS values* on page D2-2455.
- *Reserved DBGBCR<n>_EL1.LBN values* on page D2-2455.
- *Other usage constraints for Address breakpoints* on page D2-2455.
- *Other usage constraints for Context breakpoints* on page D2-2456.

Reserved DBGBCR<n>_EL1.BT values

Table D2-12 shows when particular DBGBCR<n>_EL1.BT values are reserved.

Table D2-12 Reserved BT values

BT value	Breakpoint type	Reserved
`0b001x`	Context ID Match	If the breakpoint is not context-aware
`0b010x`	Address Mismatch	In stage 1 of an AArch64 translation regime, or if EDSCR.HDE is 1 and halting is allowed
`0b011x`	CONTEXTIDR_EL1 Match	If ARMv8.1-VHE is not implemented, or the breakpoint is not context-aware
`0b100x`	VMID Match	If EL2 is not implemented, or the breakpoint is not context-aware
`0b101x`	Context ID and VMID Match	If EL2 is not implemented, or the breakpoint is not context-aware
`0b110x`	CONTEXTIDR_EL2 Match	If ARMv8.1-VHE is not implemented and ARMv8.2-Debug is not implemented, or if the breakpoint is not context-aware
`0b111x`	Full Context ID Match	

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If a breakpoint is programmed with one of these reserved BT values:

- The breakpoint must behave as if it is either:
  - Disabled.
  - Programmed with a BT value that is not reserved, other than for a direct or external read of DBGBCR<n>_EL1.
- For a direct or external read of DBGBCR<n>_EL1, if the reserved BT value:
  - Has no function for any execution conditions, the value read back is UNKNOWN.
  - Has a function for execution conditions other than the current execution conditions, the value read back is the value written. This permits software to save and restore the BT value so that the breakpoint functions for the other execution conditions.

The behavior of breakpoints with reserved BT values might change in future revisions of the architecture. For this reason, software must not rely on the behavior described here.

Reserved DBGBCR<n>_EL1.{SSC, HMC, PMC} values

Table D2-13 shows when particular combinations of DBGBCR<n>_EL1.{SSC, HMC, PMC} are reserved in stage 1 of an AArch64 translation regime.

Table D2-13 Reserved HMC, SSC, and PMC combinations

HMC, SSC, and PMC combination	Reserved
All combinations with SSC set to `0b01` or `0b10`, except for the combination with HMC set to 1, SSC set to `0b01`, and PMC set to `0b00`	When EL3 is not implemented and EL2 is implemented
Any combination where HMC or SSC is nonzero, except for the combination with HMC set to 1, SSC set to `0b01`, and PMC set to `0b00`, or combinations when SSC is set to `0b11`	When both of EL2 and EL3 are not implemented
The combination with HMC set to 1, SSC set to `0b11`, and PMC set to `0b00`	When EL2 is not implemented
The combinations with SSC set to `0b11` except the combination with HMC set to 1, SSC set to `0b11` and PMC set to `0b00`	When Secure EL2 is not implemented
The combination with HMC set to 1, SSC set to `0b01` and PMC set to `0b00`	When Secure EL2 is not implemented
Combinations not included in Table D2-9 on page D2-2449	Always

For all breakpoints except Linked Context breakpoints, if a breakpoint is programmed with one of these reserved combinations:

- If the reserved combination has a function for other execution conditions:
  - The breakpoint must behave as if it is disabled.
  - A direct or external read of DBGBCR<n>_EL1.{SSC, HMC, PMC} returns the values written. This means that software can save and restore the combination so that the breakpoint can function for the other execution conditions.
- If the reserved combination does not have a function for other execution conditions:
  - It must behave either as if it is programmed with a combination that is not reserved or as if it is disabled.
  - A direct or external read of DBGBCR<n>_EL1.{SSC, HMC, PMC} returns UNKNOWN values.

If the breakpoint is a Linked Context breakpoint, then:

- The values of HMC, SSC, and PMC are ignored.
- A direct or external read of DBGBCR<n>_EL1.{SSC, HMC, PMC} returns UNKNOWN values

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The behavior of breakpoints with reserved combinations of HMC, SSC, and PMC might change in future revisions of the architecture. For this reason, software must not rely on the behavior described here.

Reserved DBGBCR<n>_EL1.BAS values

In an AArch64-only implementation, DBGBCR<n>_EL1.BAS for all breakpoints is RES1.

Otherwise:

For all Context breakpoints

DBGBCR<n>_EL1.BAS is RES1 and is ignored.

For all Address breakpoints

Table D2-10 on page D2-2451 gives the valid values of the DBGBCR<n>_EL1.BAS field.

If a breakpoint is programmed with a reserved BAS value:

- The breakpoint must behave as if it is either:
  - Disabled.
  - Programmed with a BAS value that is not reserved, other than for a direct or external read of DBGBCR<n>_EL1.
- A direct or external read of DBGBCR<n>_EL1.BAS returns an UNKNOWN value.

Software must not rely on these properties as the behavior of reserved values might change in a future revision of the architecture.

Reserved DBGBCR<n>_EL1.LBN values

For all Context breakpoints

DBGBCR<n>_EL1.LBN reads UNKNOWN and its value is ignored.

For Linked Address breakpoints

A Linked Address breakpoint must link to a context-aware breakpoint. For a Linked Address breakpoint, any DBGBCR<n>_EL1.LBN value that is not for a context-aware breakpoint is reserved.

If a Linked Address breakpoint links to a breakpoint that is not implemented, or that is not context-aware, then reads of DBGBCR<n>_EL1.LBN return an unknown value and behavior is CONSTRAINED UNPREDICTABLE. The Linked Address breakpoint behaves as if it is either:

- Disabled.
- Linked to an UNKNOWN context-aware breakpoint.

If a Linked Address breakpoint links to a breakpoint that is implemented and that is context-aware, but that is either not enabled or not programmed as a Linked Context breakpoint, it behaves as if it is disabled.

For Unlinked Address breakpoints

DBGBCR<n>_EL1.LBN reads UNKNOWN and its value is ignored.

Other usage constraints for Address breakpoints

For all Address breakpoints

- DBGBVR<n>_EL1[1:0] are RES0 and are ignored.
- If the implementation supports AArch32 state:
  - For 32-bit instructions, if a breakpoint matches on the address of the second halfword but not the address of the first halfword, it is CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— If DBGBCR<n>.BAS is `0b1111`, it is CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception for a T32 instruction starting at address ((DBGBVR<n>[48:2]:00) + 2). For T32 instructions, Arm recommends that the debugger programs the BAS field with either `0b0011` or `0b1100`.

Other usage constraints for Context breakpoints

For all Context breakpoints

Any bits of DBGBVR<n>_EL1 that are not used to specify Context ID or VMID are RES0 and are ignored.

For Linked Context breakpoints

If no Linked Address breakpoints or Linked watchpoints link to a Linked Context breakpoint, the Linked Context breakpoint does not generate any Breakpoint exceptions.

D2.9.7 Preferred return address

The preferred return address of a Breakpoint exception is the address of the instruction that was not executed because the PE took the Breakpoint exception instead.

This means that the preferred return address is the address of the instruction that caused the exception.

D2.9.8 Pseudocode description of Breakpoint exceptions taken from AArch64 state

`AArch64.BreakpointValueMatch()` tests the value in DBGBVR<n>_EL1.

`AArch64.StateMatch()` tests the values in DBGBCR<n>_EL1.{SSC, HMC, PMC} and, if the breakpoint links to a Linked Context breakpoint, also tests the Linked Context breakpoint.

For a watchpoint, `AArch64.StateMatch()` tests the values in DBGWCR<n>_EL1.{SSC, HMC, PAC} and, if the watchpoint links to a Linked Context breakpoint, also tests the Linked Context breakpoint.

`AArch64.BreakpointMatch()` tests a committed instruction against all breakpoints.

`AArch64.CheckBreakpoint()` generates a Breakpoint exception if all of the following are true:

- MDSCR_EL1.MDE is 1.
- Debug exceptions are enabled from the current Exception level and Security state. See *Enabling debug exceptions from the current Exception level* on page D2-2431.
- All of the conditions required for Breakpoint exception generation are met. See *About Breakpoint exceptions* on page D2-2439.

Note

`AArch64.CheckBreakpoint()` might halt the PE and cause it to enter Debug state. External debug uses Debug state.

`AArch64.BreakpointException()` is called to generate a Breakpoint exception.

These functions are defined in Chapter J1 *Armv8 Pseudocode*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D2.10 Watchpoint exceptions

This section describes Watchpoint exceptions in stage 1 of an AArch64 translation regime.

The PE is using an AArch64 translation regime when it is executing either:

- In an Exception level that is using AArch64.
- At EL0 using AArch32 when EL1 is using AArch64.

This section contains the following subsections:

- *About Watchpoint exceptions*.
- *Watchpoint types and linking of watchpoints* on page D2-2458.
- *Execution conditions for which a watchpoint generates Watchpoint exceptions* on page D2-2459.
- *Watchpoint data address comparisons* on page D2-2462.
- *Determining the memory location that caused a Watchpoint exception* on page D2-2465.
- *Watchpoint behavior on other instructions* on page D2-2466.
- *Watchpoint usage constraints* on page D2-2467.
- *Exception syndrome information and preferred return address* on page D2-2469.
- *Pseudocode description of Watchpoint exceptions taken from AArch64 state* on page D2-2470.

D2.10.1 About Watchpoint exceptions

A *watchpoint* is an event that results from the execution of an instruction, based on a data address. Watchpoints are also known as *data breakpoints*.

A watchpoint operates as follows:

1. A debugger programs the watchpoint with a data address, or a data address range.
2. The watchpoint generates a *Watchpoint debug event* on an access to the address, or any address in the address range.

A watchpoint never generates a Watchpoint debug event on an instruction fetch.

An implementation can include between 2-16 watchpoints. In an implementation, ID_AA64DFR0_EL1.WRPs shows how many are implemented.

To use an implemented watchpoint, a debugger programs the following registers for the watchpoint:

- The *Watchpoint Control Register*, DBGWCR<n>_EL1. This contains controls for the watchpoint, for example an enable control.
- The *Watchpoint Value Register*, DBGWVR<n>_EL1. This holds the data virtual address used for watchpoint matching.

These registers are numbered, so that:

- DBGWCR1_EL1 and DBGWVR1_EL1 are for watchpoint number one.
- DBGWCR2_EL2 and DBGWVR2_EL1 are for watchpoint number two.
- …
- …
- DBGWCR<n>_EL1 and DBGWVR<n>_EL1 are for watchpoint number n.

A watchpoint can:

- Be programmed to generate Watchpoint debug events on read accesses only, on write accesses only, or on both types of access.
- Link to a *Linked Context breakpoint*, so that a Watchpoint debug event is only generated if the PE is in a particular context when the address match occurs.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A single watchpoint can be programmed to match on one or more address bytes. A watchpoint generates a Watchpoint debug event on an access to any byte that it is watching. The number of bytes a watchpoint is watching is either:

- One to eight bytes, provided that these bytes are contiguous and that they are all in the same naturally-aligned doubleword. A debugger uses the *Byte Address Select* field, DBGWCR<n>_EL1.BAS, to select the bytes. See *Programming a watchpoint with eight bytes or fewer* on page D2-2463.
- Eight bytes to 2GB, provided that both of the following are true:
  - The number of bytes is a power-of-two.
  - The range starts at an address that is aligned to the range size.

  A debugger uses the *MASK* field, DBGWCR<n>_EL1.MASK, to program a watchpoint with eight bytes to 2GB. See *Programming a watchpoint with eight or more bytes* on page D2-2464.

A debugger must use either the BAS field or the MASK field. If it uses both, whether the watchpoint generates Watchpoint debug events is CONSTRAINED UNPREDICTABLE. See *Programming dependencies of the BAS and MASK fields* on page D2-2468.

For each memory access, all of the watchpoints are tested. When a watchpoint is tested, it generates a Watchpoint debug event if all of the following are true:

- The watchpoint is enabled. That is, the watchpoint enable control for it, DBGWCR<n>_EL1.E, is 1.
- The conditions specified in the DBGWCR<n>_EL1 are met.
- The comparison with the address held in the DBGWVR<n>_EL1 is successful.
- If the watchpoint links to a Linked Context breakpoint, the comparison or comparisons made by the Linked Context breakpoint also are successful. See Figure D2-1 on page D2-2442. See also *Breakpoint context comparisons* on page D2-2452.
- The instruction that initiates the memory access is committed for execution.
- The instruction that initiates the memory access passes its Condition code check.

If halting is allowed and EDSCR.HDE is 1, Watchpoint debug events cause entry to Debug state.

Otherwise, if debug exceptions are:
- Enabled, Watchpoint debug events generate Watchpoint exceptions.
- Disabled, Watchpoint debug events are ignored.

Note

The remainder of this Watchpoint Exceptions section, including all subsections, describes watchpoints as generating Watchpoint exceptions.

However, the behavior described also applies if watchpoints are causing entry to Debug state.

The debug exception enable controls on page D2-2428 describes the enable controls for Watchpoint debug events.

D2.10.2 Watchpoint types and linking of watchpoints

When a debugger programs a watchpoint, it must program that watchpoint so that it is either:
- Used in isolation. In this case, the watchpoint is called an *Unlinked watchpoint*.
- Enabled for linking to a Linked Context breakpoint. In this case, the watchpoint is called a *Linked watchpoint*.

When a Linked watchpoint links to a Linked Context breakpoint, the Linked watchpoint only generates a Watchpoint exception if the PE is in a particular context when the data address match occurs. For example, a debugger might:

1. Program watchpoint number one with a data address.
2. Program breakpoint number five to be a *Linked VMID Match breakpoint*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

3. Link the watchpoint and the breakpoint together. A Watchpoint exception is only generated if both the data address matches and the VMID matches.

The *Watchpoint Type* field for a watchpoint, DBGWCR<n>_EL1.WT, controls whether the watchpoint is enabled for linking. If DBGWCR<n>_EL1.WT is 1, the watchpoint is enabled for linking.

Rules for linking watchpoints

The rules for watchpoint linking are as follows:

- Only Linked watchpoints can be linked.
- A Linked watchpoint can link to any type of Linked Context breakpoint. The *Linked Breakpoint Number* field, DBGWCR<n>_EL1.LBN, for the Linked watchpoint specifies the particular Linked Context breakpoint that the Linked watchpoint links to, and:
  - DBGWCR<n>_EL1.WT.{SSC, HMC, PAC} for the Linked watchpoint defines the execution conditions that the watchpoint generates Watchpoint exceptions for. See *Execution conditions for which a watchpoint generates Watchpoint exceptions*.
  - DBGBCR<n>_EL1.{SSC, HMC, PMC} for the Linked Context breakpoint are ignored.
- A Linked watchpoint cannot link to another watchpoint. The LBN field can therefore only specify a breakpoint.
- If a Linked watchpoint links to a breakpoint that is not context-aware, the behavior of the Linked watchpoint is CONSTRAINED UNPREDICTABLE. See *Watchpoint usage constraints* on page D2-2467.
- If a Linked watchpoint links to an Unlinked Context breakpoint, the Linked watchpoint never generates any Watchpoint exceptions.
- Multiple Linked watchpoints can link to a single Linked Context breakpoint.

  Note

  Multiple Address breakpoints can also link to a single Linked Context breakpoint. *Breakpoint exceptions* on page D2-2439 describes breakpoints.

Figure D2-1 on page D2-2442 shows an example of permitted watchpoint linking.

D2.10.3 Execution conditions for which a watchpoint generates Watchpoint exceptions

Each watchpoint can be programmed so that it only generates Watchpoint exceptions for certain execution conditions. For example, a watchpoint might be programmed to generate Watchpoint exceptions only when the PE is executing at EL2 in Non-secure state.

DBGWCR<n>_EL1.{SSC, HMC, PAC} define the execution conditions a watchpoint generates Watchpoint exceptions for, as follows:

Security State Control, SSC

Controls whether the watchpoint generates Watchpoint exceptions only in Secure state, only in Non-secure state, or in both Security states.

Note

This is determined by the Security state of the PE, not from the NS attribute returned by the translation of the virtual address on which the watchpoint is set.

Higher Mode Control, HMC, and Privileged Access Control, PAC

HMC and PAC together control which Exception levels the watchpoint generates Watchpoint exceptions in.

The PAC control relates to the privilege of the memory access, not to the Exception level at which the access was made.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

This means that, if the PE executes a Load unprivileged or Store unprivileged instruction at EL1, the resulting data access triggers a watchpoint only if both:

- PAC is programmed to a value that generates watchpoints on EL0 accesses.
- All other conditions for generating the watchpoint are met.

Example A64 Load unprivileged and Store unprivileged instructions are `LDTR` and `STTR`.

Table D2-14 shows the valid combinations of HMC, SSC, and PAC, and for each combination shows which Exception levels watchpoints generate Watchpoint exceptions in.

In the table:

Y or - Means that a watchpoint programmed with the values of HMC, SSC, and PAC shown in that row:

Y Can generate Watchpoint exceptions in that Exception level and Security state.

- Cannot generate Watchpoint exceptions in that Exception level and Security state.

For information about which combinations of HMC, SSC and PMC are reserved if an Exception level or Security state are not implemented or enabled, see *Reserved DBGWCR<n>_EL1.{SSC, HMC, PAC} values* on page D2-2467.

Table D2-14 Summary of watchpoint HMC, SSC, and PAC encodings

HMC	SSC	PAC	Security state	EL3[a]	EL2	EL1	EL0
0	00	01	Both	-	-	Y	-
0	00	10		-	-	-	Y
0	00	11		-	-	Y	Y
0	01	01	Non-secure	n/a	-	Y	-
0	01	10		n/a	-	-	Y
0	01	11		n/a	-	Y	Y
0	10	01	Secure	-	-	Y	-
0	10	10		-	-	-	Y
0	10	11		-	-	Y	Y
0	11	00		-	Y	-	-
0	11	01		-	Y	Y	-
0	11	11		-	Y	Y	Y
1	00	01	Both	Y	Y	Y	-
1	00	11		Y	Y	Y	Y
1	01	00	Non-secure	n/a	Y	-	-
1	01	01		n/a	Y	Y	-
1	01	11		n/a	Y	Y	Y

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D2-14 Summary of watchpoint HMC, SSC, and PAC encodings (continued)

HMC	SSC	PAC	Security state	EL3[a]	EL2	EL1	EL0
1	10	00	Secure	Y	-	-	-
1	10	01		Y	Y	Y	-
1	10	11		Y	Y	Y	Y
1	11	00	Both	-	Y	-	-
1	11	01		-	Y	Y	-
1	11	11		-	Y	Y	Y

a. Debug exceptions are not generated at EL3 using AArch64. This means that these combinations of HMC, SSC, and PAC are only relevant if watchpoints cause entry to Debug state. Self-hosted debuggers must avoid combinations of HMC, SSC, and PMC that generate Watchpoint exceptions at EL3 using AArch64.

All combinations of HMC, SSC, and PAC that this table does not show are reserved. See *Reserved DBGWCR<n>_EL1.{SSC, HMC, PAC} values* on page D2-2467.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D2.10.4 Watchpoint data address comparisons

In this subsection, the term AddrTop represents the most significant bit of a virtual address used by watchpoint data address comparisons. AddrTop is:

- 55, if address tagging is used for the address. See *Address tagging in AArch64 state* on page D5-2534.
- 63, otherwise.

Note

When stage 1 translation is enabled, in AArch64 state, a virtual address has a maximum address width of either 48 bits or, when ARMv8.2-LVA is implemented and the 64KB translation granule is used, 52 bits. Software can configure a smaller address width for a virtual address. See *Input address size* on page D5-2549. Attempting to translate an address that is larger than the configured input address size generates a Translation fault.

When stage 1 translation is disabled, using an address that is larger than the implemented PA size generates an Address size fault. The implemented PA size is IMPLEMENTATION DEFINED up to 52 bits. See *Physical address size* on page D5-2548.

These faults have a higher priority than watchpoints.

An address comparison is successful if bits [AddrTop:2] of the current data address are equal to DBGWVR<n>_EL1[AddrTop:2], taking into account all of the following:

- The size of the access. See *Size of the data access*.

  If EL1 is using AArch64 and EL0 is using AArch32, AArch32 instructions can be executed in stage 1 of an AArch64 translation regime. In this case, data addresses are zero-extended before comparison with the watchpoint.

- The bytes selected by DBGWVR<n>_EL1.BAS. See *Programming a watchpoint with eight bytes or fewer* on page D2-2463.

- Any address ranges indicated by DBGWVR<n>_EL1.MASK. See *Programming a watchpoint with eight or more bytes* on page D2-2464.

Note

- DBGWVR<n>_EL1 is a 64-bit register. The most significant bits of this register are sign-extension bits.
- DBGWVR<n>_EL1[1:0] are RES0 and are ignored

Size of the data access

Because watchpoints can be programmed to generate Watchpoint exceptions on individual bytes, the size of each data access must be taken into account. See Example D2-1.

Example D2-1

1. A debugger programs a watchpoint to generate Watchpoint exceptions only when the byte at address 0x1009 is accessed.

2. The PE accesses the unaligned doubleword starting at address 0x1003.

In this scenario, the watchpoint must generate a Watchpoint exception.

The size of data accesses initiated by DC ZVA instructions is the DC ZVA block size that DCZID_EL0.BS defines.

The size of data accesses initiated by DC IVAC instructions is an IMPLEMENTATION DEFINED size that is both:

- From the inclusive range between:
  - The size that CTR_EL0.DminLine defines.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

  — 2KB.
- A power-of-two.

For both of these instructions:

- The lowest address accessed by the instruction is the address supplied to the instruction, rounded down to the nearest multiple of the access size initiated by that instruction.
- The highest address accessed is (size - 1) bytes above the lowest address accessed.

See also, *Watchpoint behavior on accesses by the DC IVAC instruction and the DC ZVA instruction* on page D2-2467.

Programming a watchpoint with eight bytes or fewer

The Byte Address Select field, DBGWCR<n>_EL1.BAS, selects which bytes in the doubleword starting at the address contained in the DBGWVR<n>_EL1 the watchpoint generates Watchpoint exceptions for.

If the address programmed into the DBGWVR<n>_EL1 is:

- Doubleword-aligned:
  — All eight bits of DBGWCR<n>_EL1.BAS are used, and the descriptions given in Table D2-15 apply.
- Word-aligned but not doubleword-aligned:
  — Only DBGWCR<n>_EL1.BAS[3:0] are used, and the descriptions given in Table D2-16 apply. In this case, DBGWCR<n>_EL1.BAS[7:4] are RES0.

Table D2-15 Supported BAS values when the DBGWVRn_EL1 address alignment is doubleword

BAS value	Description
`0b00000000`	Watchpoint never generates a Watchpoint exception.
BAS[0] == 1	Generates a Watchpoint exception if the byte at address DBGWVR<n>_EL1[`AddrTop`:3]:000 is accessed.
BAS[1] == 1	Generates a Watchpoint exception if the byte at address DBGWVR<n>_EL1[`AddrTop`:3]:001 is accessed.
BAS[2] == 1	Generates a Watchpoint exception if the byte at address DBGWVR<n>_EL1[`AddrTop`:3]:010 is accessed.
BAS[3] == 1	Generates a Watchpoint exception if the byte at address DBGWVR<n>_EL1[`AddrTop`:3]:011 is accessed.
BAS[4] == 1	Generates a Watchpoint exception if the byte at address DBGWVR<n>_EL1[`AddrTop`:3]:100 is accessed.
BAS[5] == 1	Generates a Watchpoint exception if the byte at address DBGWVR<n>_EL1[`AddrTop`:3]:101 is accessed.
BAS[6] == 1	Generates a Watchpoint exception if the byte at address DBGWVR<n>_EL1[`AddrTop`:3]:110 is accessed.
BAS[7] == 1	Generates a Watchpoint exception if the byte at address DBGWVR<n>_EL1[`AddrTop`:3]:111 is accessed.

Table D2-16 Supported BAS values when the DBGWVRn_EL1 address alignment is word

BAS value[a]	Description
`0b00000000`	Watchpoint never generates a Watchpoint exception
BAS[0] == 1	Generates a Watchpoint exception if byte at address DBGWVR<n>_EL1[`AddrTop`:2]:00 is accessed.
BAS[1] == 1	Generates a Watchpoint exception if byte at address DBGWVR<n>_EL1[`AddrTop`:2]:01 is accessed.
BAS[2] == 1	Generates a Watchpoint exception if byte at address DBGWVR<n>_EL1[`AddrTop`:2]:10 is accessed.
BAS[3] == 1	Generates a Watchpoint exception if byte at address DBGWVR<n>_EL1[`AddrTop`:2]:11 is accessed.

a. DBGWCR<n>_EL1.BAS[7:4] are RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the BAS field is programmed with more than one byte, the bytes that it is programmed with must be contiguous. For watchpoint behavior when its BAS field is programmed with non-contiguous bytes, see *Other usage constraints* on page D2-2469.

When programming the BAS field with anything other than `0b11111111`, a debugger must program DBGWCR<n>_EL1.MASK to be `0b00000`. See *Programming dependencies of the BAS and MASK fields* on page D2-2468.

A watchpoint generates a Watchpoint exception whenever a watched byte is accessed, even if:

- The access size is smaller or larger than the address region being watched.
- The access is misaligned, and the base address of the access is not in the doubleword or word of memory addressed by the DBGWVR<n>_EL1[`AddrTop`:3]. See Example D2-1 on page D2-2462.

The following are some example configurations of the BAS field:

- To program a watchpoint to generate a Watchpoint exception on the byte at address `0x1003`, program:
  - DBGWVR<n>_EL1 with `0x1000`.
  - DBGWCR<n>_EL1.BAS to be `0b00001000`.
- To program a watchpoint to generate a Watchpoint exception on the bytes at addresses `0x2003`, `0x2004` and `0x2005`, program:
  - DBGWVR<n>_EL1 with `0x2000`.
  - DBGWCR<n>_EL1.BAS to be `0b00111000`.
- If the address programmed into the DBGWVR<n>_EL1 is doubleword-aligned:
  - To generate a Watchpoint exception when any byte in the word starting at the doubleword-aligned address is accessed, program DBGWCR<n>_EL1.BAS to be `0b00001111`.
  - To generate a Watchpoint exception when any byte in the word starting at address DBGWVR<n>_EL1[31:3]:100 is accessed, program DBGWCR<n>_EL1.BAS to be `0b11110000`.

Note

Arm deprecates programming a DBGWVR<n>_EL1 with an address that is not doubleword-aligned.

Programming a watchpoint with eight or more bytes

A debugger can use the *MASK* field, DBGWCR<n>_EL1.MASK, to program a single watchpoint with a data address range. The range must meet all of the following criteria:

- It is a size that is:
  - A power-of-two.
  - A minimum of eight bytes.
  - A maximum of 2GB.
- It starts at an address that is aligned to the size.

The MASK field specifies the number of least significant data address bits that must be masked. Up to 31 least significant bits can be masked:

MASK	`0b00000`	No bits are masked.
	`0b00001`	Reserved.
	`0b00010`	Reserved.
	`0b00011`	Three least significant bits are masked.
	`0b00100`	Four least significant bits are masked.
	`0b00101`	Five least significant bits are masked.
	…	…
	`0b11111`	31 least significant bits are masked.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If *n* least significant address bits are masked, the watchpoint generates a Watchpoint exception on all of the following:

- Address DBGWVR<n>_EL1[`AddrTop`:*n*]:000…
- Address DBGWVR<n>_EL1[`AddrTop`:*n*]:111…
- Any address between these two addresses.

For example, if the four least significant address bits are masked, Watchpoint exceptions are generated for all addresses between DBGWVR<n>_EL1[`AddrTop`:4]:0000 and DBGWVR<n>_EL1[`AddrTop`:4]:1111, including these addresses.

Note

- The 17 most significant bits cannot be masked. This means that the full address cannot be masked.
- For watchpoint behavior when its MASK field is programmed with a reserved value, see *Reserved DBGWCR<n>_EL1.MASK values* on page D2-2469.

When masking address bits, a debugger must both:

- Program DBGWCR<n>_EL1.BAS to be `0b11111111`. See *Programming dependencies of the BAS and MASK fields* on page D2-2468.
- In the DBGWVR<n>_EL1, set the masked address bits to 0. For watchpoint behavior when any of the masked address bits are not 0, see *Other usage constraints* on page D2-2469.

D2.10.5 Determining the memory location that caused a Watchpoint exception

On taking a Watchpoint exception, the PE records an address in a *Fault Address Register* that the debugger can use to determine the memory location that triggered the watchpoint.

The Fault Address Register (FAR) used is either:

- FAR_EL1, if the exception is taken to EL1.
- FAR_EL2, if the exception is taken to EL2.

In cases where one instruction triggers multiple watchpoints, only one address is recorded.

On entering Debug state on a Watchpoint debug event, the PE records the address in the EDWAR.

For more information, see the subsections that follow. These are:

- *Address recorded for Watchpoint exceptions generated by instructions other than data cache maintenance instructions*
- *Address recorded for Watchpoint exceptions generated by data cache maintenance instructions* on page D2-2466

Address recorded for Watchpoint exceptions generated by instructions other than data cache maintenance instructions

Note

Despite its mnemonic, the `DC ZVA, Data Cache Zero by VA` instruction is not a data cache maintenance instruction.

The address recorded must be both:

- From the inclusive range between:
  - The lowest address accessed by the memory access that triggered the watchpoint.
  - The highest *watchpointed address* accessed by the memory access. A watchpointed address is an address that the watchpoint is watching.
- Within a naturally-aligned block of memory that is all of the following:
  - A power-of-two size.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— No larger than the `DC ZVA` block size.
— Contains a watchpointed address accessed by the memory access.

The size of the block is IMPLEMENTATION DEFINED. There is no architectural means of discovering the size.

Example D2-2 Address recorded for a watchpoint programmed on `0x8019`

A debugger programs a watchpoint to generate a Watchpoint exception on any access to the byte `0x8019`.

An A32 load multiple instruction then loads nine registers starting from address `0x8004` upwards. This triggers the watchpoint.

If the `DC ZVA` block size is:

- 32 bytes, the address that the PE records must be between `0x8004` and `0x8019` inclusive.
- 16 bytes, the address that the PE records must be between `0x8010` and `0x8019` inclusive.

Address recorded for Watchpoint exceptions generated by data cache maintenance instructions

The address recorded is the address passed to the instruction. This means that the address recorded might be higher than the address of the location that triggered the watchpoint.

D2.10.6 Watchpoint behavior on other instructions

Under normal operating conditions, the following do not generate Watchpoint exceptions:

- Instruction cache maintenance instructions.
- Address translation instructions.
- TLB maintenance instructions.
- Prefetch memory instructions.
- All data cache maintenance instructions except `DC IVAC`.

Note

Despite its mnemonic, the `DC ZVA, Data Cache Zero by VA` instruction is not a data cache maintenance instruction.

However, the debug architecture allows for IMPLEMENTATION DEFINED controls, such as those in ACTLR registers, to enable watchpoints on an implementation defined subset of these instructions. Whether a watchpoint treats the instruction as a load or a store, and the access size of instruction cache, address translation, and TLB operations are implementation defined.

The access size of the IMPLEMENTATION DEFINED instruction cache, address translation, and TLB operations which generate Watchpoint exceptions are IMPLEMENTATION DEFINED.

See also the following subsections:

- *Watchpoint behavior on accesses by Store-Exclusive instructions*.
- *Watchpoint behavior on accesses by the DC IVAC instruction and the DC ZVA instruction* on page D2-2467.

Watchpoint behavior on accesses by Store-Exclusive instructions

If a watchpoint matches on a data access caused by a Store-Exclusive instruction, then:

- If the store fails because an Exclusives monitor does not permit it, it is IMPLEMENTATION DEFINED whether the watchpoint generates a Watchpoint exception.
- Otherwise, the watchpoint generates a Watchpoint exception.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Watchpoint behavior on accesses by the DC IVAC instruction and the DC ZVA instruction

`DC ZVA` operations can generate Watchpoint exceptions. If the Point of Coherency is before any level of cache, it is IMPLEMENTATION DEFINED whether a DC IVAC instruction can generate a Watchpoint exception. Otherwise, DC IVAC operations can generate Watchpoint exceptions.

`DC IVAC` and `DC ZVA` operations are treated as data stores by DBGWCR<n>_EL1.LSC.

Note

For the size of data accesses performed by the `DC IVAC` instruction and the `DC ZVA` instruction, see *Watchpoint data address comparisons* on page D2-2462. The size of all data accesses must be considered because watchpoints can be programmed to match on individual bytes.

D2.10.7 Watchpoint usage constraints

See the following:

- *Reserved DBGWCR<n>_EL1.{SSC, HMC, PAC} values*.
- *Reserved DBGWCR<n>_EL1.LBN values* on page D2-2468.
- *Programming dependencies of the BAS and MASK fields* on page D2-2468.
- *Reserved DBGWCR<n>_EL1.BAS values* on page D2-2468.
- *Reserved DBGWCR<n>_EL1.MASK values* on page D2-2469.
- *Other usage constraints* on page D2-2469.

Reserved DBGWCR<n>_EL1.{SSC, HMC, PAC} values

Table D2-17 shows when particular combinations of DBGWCR<n>_EL1.{SSC, HMC, PAC} are reserved.

Table D2-17 Reserved SSC, HMC, and PAC combinations

HMC, SSC, and PAC combination	Reserved
All combinations with SSC set to `0b01` or `0b10` except for the combination with HMC set to 1, SSC set to `0b01`, and PAC set to `0b00`.	When EL3 is not implemented and EL2 is implemented.
All combinations where HMC or SSC is nonzero, except for the combination with HMC set to 1, SSC set to `0b01`, and PAC set to `0b00` or combinations with SSC set to `0b11`.	When both of EL2 and EL3 are not implemented.
The combination with HMC set to 1, SSC set to `0b11`, and PAC set to `0b00`.	When EL2 is not implemented.
The combinations with SSC set to `0b11` except the combination with HMC set to 1, SSC set to `0b11`, and PAC set to `0b00`.	When Secure EL2 is not implemented.
The combination with HMC set to 1, SSC set to `0b01`, and PAC set to `0b00`.	When Secure EL2 is not implemented.
Combinations not included in Table D2-14 on page D2-2460.	Always.

If a watchpoint is programmed with one of these reserved combinations:

- The watchpoint must behave as if it is either:
  - Disabled.
  - Programmed with a combination that is not reserved, other than for a direct or external read of DBGWCR<n>_EL1.
- For a direct or external read of DBGWCR<n>_EL1, if the reserved combination:
  - Has no function for any execution conditions, the value read back for each of SSC, HMC, and PMC is UNKNOWN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— Has a function for execution conditions other than the current execution conditions, the value read back is the value written. This permits software to save and restore the combination so that the watchpoint functions for the other execution conditions.

The behavior of watchpoints with reserved combinations of SSC, HMC, and PAC might change in future revisions of the architecture. For this reason, software must not rely on the behavior described here.

Reserved DBGWCR<n>_EL1.LBN values

For Linked Watchpoints

A Linked watchpoint must link to a context-aware breakpoint. For a Linked watchpoint, any DBGWCR<n>_EL1.LBN value that is not for a context-aware breakpoint is reserved.

If a Linked watchpoint links to a breakpoint that is not implemented, or that is not context-aware, then reads of DBGWCR<n>_EL1.LBN return an UNKNOWN value and the behavior is CONSTRAINED UNPREDICTABLE. The Linked watchpoint behaves as if it is either:

- Disabled
- Linked to an UNKNOWN context-aware breakpoint.

If a Linked watchpoint links to a breakpoint that is implemented and is context-aware, but that is either not enabled or not programmed as a Linked Context breakpoint, it behaves as if it is disabled.

For Unlinked Watchpoints For Unlinked watchpoints, DBGWCR<n>_EL1.LBN reads UNKNOWN and its value is ignored.

Programming dependencies of the BAS and MASK fields

When programming a watchpoint, a debugger must use either:

- The MASK field, to program the watchpoint with an address range that can be eight bytes to 2GB.
- The BAS field, to select which bytes in the doubleword or word starting at the address contained in the DBGWVR<n>_EL1 the watchpoint must generate Watchpoint exceptions for.

If the debugger uses the:

- MASK field, it must program BAS to be `0b11111111`, so that all bytes in the doubleword or word are selected.
- BAS field, it must program MASK to be `0b00000`, so that the MASK field does not indicate any address ranges.

If an enabled watchpoint has a MASK field that is non-zero and a BAS field that is not set to 0b11111111, then for each byte in the address range, it is CONSTRAINED UNPREDICTABLE whether or not a Watchpoint exception is generated.

Reserved DBGWCR<n>_EL1.BAS values

The BAS field must be programmed with a value `Zeros(8-n-m):Ones(n):Zeros(m)`, where:

- `n` is a non-zero positive integer less-than-or-equal-to 8.
- `m` is a positive integer less-than 8.
- `n+m` is less-than-or-equal-to 8.

All other values are reserved.

Note

If `x` is zero, then `Zeros(x)` is an empty bitstring.

If DBGWVR<n>_EL1[2] is 1, DBGWCR<n>_EL1.BAS[7:4] are RES0 and are ignored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If a watchpoint is programmed with a reserved BAS value:

- It is CONSTRAINED UNPREDICTABLE whether the watchpoint generates a Watchpoint exception for each byte in the doubleword or word of memory addressed by the DBGWVR<n>_EL1.
- A direct or external read of DBGWCR<n>_EL1.BAS returns an UNKNOWN value.

Software must not rely on these properties as the behavior of reserved values might change in a future revision of the architecture.

Reserved DBGWCR<n>_EL1.MASK values

If a watchpoint is programmed with a reserved MASK value:

- The watchpoint must behave as if it is either:
  - Disabled.
  - Programmed with an UNKNOWN value that is not reserved, that might be `0b00000`, other than for a direct or external read of DBGWCR<n>_EL1.
- A direct or external read of DBGWCR<n>_EL1.MASK returns an UNKNOWN value.

Other usage constraints

For all watchpoints:

- DBGWVR<n>_EL1[1:0] are RES0 and are ignored.
- If DBGWCR<n>_EL1.MASK is nonzero, and any masked bits of DBGWVR<n>_EL1 are not 0, it is CONSTRAINED UNPREDICTABLE whether the watchpoint generates a Watchpoint exception when the unmasked bits match.
- A watchpoint never generates any Watchpoint exceptions if DBGWCR<n>_EL1.LSC is `0b00`.

D2.10.8 Exception syndrome information and preferred return address

See the following:

- *Exception syndrome information*.
- *Preferred return address* on page D2-2470.

Exception syndrome information

On taking a Watchpoint exception, the PE records all of the following:

- Information about the exception in the *Exception Syndrome Register* (ESR_ELx) at the Exception level the exception is taken to.
- An address that the debugger can use to determine the memory location that caused the exception. The PE records this in a *Fault Address Register* (FAR).

The ESR and FAR used is either:

- ESR_EL1 and FAR_EL1, if the exception is taken to EL1.
- ESR_EL2 and FAR_EL2, if the exception is taken to EL2.

Note

Watchpoint exceptions cannot be taken to EL3 using AArch64.

See *ISS encoding for an exception from a Watchpoint exception* on page D13-3001 for more information.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Preferred return address

The preferred return address of a Watchpoint exception is the address of the instruction that was not executed because the PE took the Watchpoint exception instead.

This means that the preferred return address is the address of the instruction that caused the exception.

D2.10.9 Pseudocode description of Watchpoint exceptions taken from AArch64 state

`AArch64.WatchpointByteMatch()` tests an individual byte accessed by an operation.

`AArch64.StateMatch()` tests the values in DBGWCR<n>_EL1.{HMC, SSC, PAC}, and if the watchpoint is Linked, also tests the Linked Context breakpoint that the watchpoint links to.

`AArch64.WatchpointMatch()` tests the value in DBGWVR<n>_EL1.

`AArch64.CheckWatchpoint()` generates a `FaultRecord` that `AArch64.Abort()` raises a Watchpoint exception for if all of the following are true:

- MDSCR_EL1.MDE is 1.
- Debug exceptions are enabled from the current Exception level and Security state. See *Enabling debug exceptions from the current Exception level* on page D2-2431.
- All of the conditions required for Watchpoint exception generation are met. See *About Watchpoint exceptions* on page D2-2457.

Note

`AArch64.CheckWatchpoint()` might halt the PE and cause it to enter Debug state. External debug uses Debug state.

`AArch64.WatchpointException()` is called to generate a Watchpoint exception.

These functions are defined in Chapter J1 *Armv8 Pseudocode*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D2.11 Vector Catch exceptions

Vector Catch exceptions are not generated in AArch64 translation regimes.

Note

This means that they are never taken to EL1 using AArch64 and are only supported if at least EL1 using AArch32 is supported.

A debugger that is executing in EL2 using AArch64 can route Vector Catch exceptions to EL2 using AArch64. See *Routing debug exceptions* on page D2-2429.

`AArch64.VectorCatchException()` is called to generate a Vector Catch exception.

Vector Catch exceptions on page G2-5869 describes Vector Catch exceptions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D2.12 Software Step exceptions

The following subsections describe Software Step exceptions:

- *About Software Step exceptions*.
- *Rules for setting MDSCR_EL1.SS to 1*.
- *The software step state machine*.
- *Entering the active-not-pending state* on page D2-2474.
- *Behavior in the active-not-pending state* on page D2-2477.
- *Entering the active-pending state* on page D2-2479.
- *Behavior in the active-pending state* on page D2-2480.
- *Stepping T32 IT instructions* on page D2-2480.
- *Exception syndrome information and preferred return address* on page D2-2481.
- *Additional considerations* on page D2-2482.
- *Pseudocode description of Software Step exceptions* on page D2-2484.

D2.12.1 About Software Step exceptions

Software step is an Armv8-A resource that a debugger can use to make the PE single-step instructions.

For example, by using software step, debugger software executing at a higher Exception level can single-step instructions at a lower Exception level.

Operation is as follows:

1. A debugger:
   a. Enables software step by setting MDSCR_EL1.SS to 1. See *The debug exception enable controls* on page D2-2428.
   b. Executes an exception return instruction, `ERET`, to branch to the instruction to be single-stepped in the software being debugged.
2. The PE then:
   a. Executes the instruction to be single-stepped.
   b. Takes a Software Step exception on the next instruction, returning control to the debugger.

However, another exception might be generated while the instruction is being stepped. This exception is either:

- A synchronous exception that is generated by the instruction being stepped.
- An asynchronous exception that is taken before or after the instruction being stepped.

The PE can only take a Software Step exception if debug exceptions are enabled from the current Exception level and Security state. See *Enabling debug exceptions from the current Exception level* on page D2-2431.

A state machine describes the behavior of software step, shown in *The software step state machine*.

Throughout this *Software Step exceptions* section, including in all subsections, EL_D means the Exception level that Software Step exceptions are targeting. *Routing debug exceptions* on page D2-2429 defines EL_D as the *debug target Exception level*.

D2.12.2 Rules for setting MDSCR_EL1.SS to 1

Debugger software must be executing in an Exception level and Security state that debug exceptions are disabled from when it sets MDSCR_EL1.SS to 1.

The Exception level that hosts the debugger software must be using AArch64.

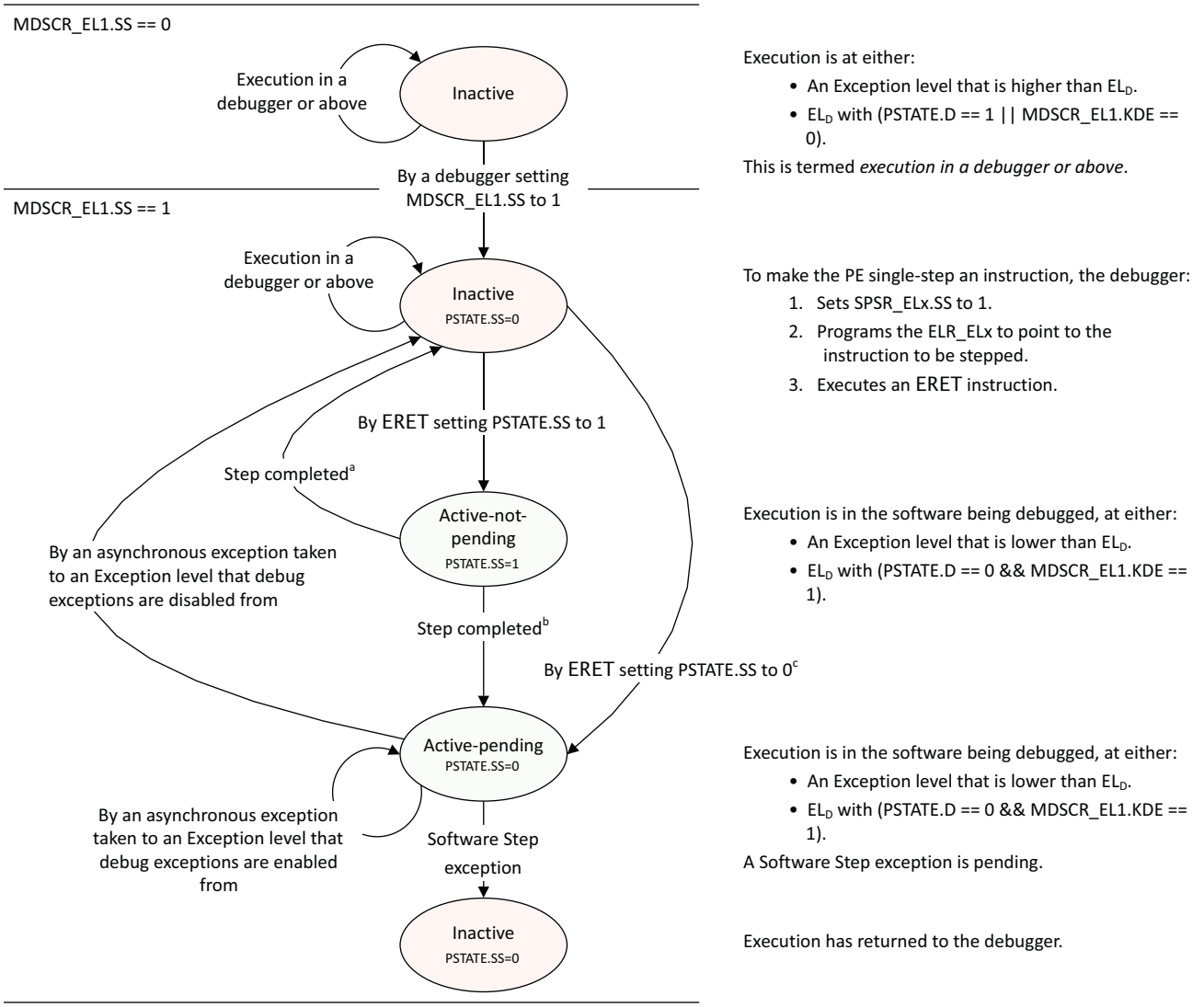
D2.12.3 The software step state machine

In Figure D2-3 on page D2-2473:

- The OS Lock is unlocked and `DoubleLockStatus()` == FALSE.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The PE is not in Secure state with MDCR_EL3.SDD set to 1. That is, the PE is in Non-secure state, or is in Secure state with MDCR_EL3.SDD set to 0, or the implementation does not include EL3.



a. The step is the PE either:
- Taking an exception to an Exception level that debug exceptions are disabled from.
- If execution is at EL_D with MDSCR_EL1.KDE == 1, executing an instruction that sets PSTATE.D to 1.

Software step is inactive when debug exceptions are disabled from the current Exception level, and debug exceptions are disabled from EL_D when PSTATE.D is 1.

b. The step is the PE either:
- Executing the instruction to be stepped without taking an exception.
- Taking an exception to an Exception level that debug exceptions are enabled from. The Exception level might be using AArch64 or AArch32.

c. Or, if execution is at EL_D with MDSCR_EL1.KDE == 1, by software setting PSTATE.D to 0.

Figure D2-3 Software step state machine

For a description of when debug exceptions are enabled or disabled from an Exception level, see *Enabling debug exceptions from the current Exception level* on page D2-2431.

For more information about how a step is completed, see *Behavior in the active-not-pending state* on page D2-2477.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The software step states are:

Inactive Software step is inactive. It cannot generate any Software Step exceptions or affect PE execution. Software step is inactive whenever any of the following are true:

- MDSCR_EL1.SS is 0.
- EL_D is using AArch32.
- Debug exceptions are disabled from the current Exception level or Security state.

Active-not-pending

None of the conditions mentioned in *Inactive* are true, therefore software step is active.

The current instruction is the instruction to be stepped.

Active-pending

None of the conditions mentioned in *Inactive* are true, therefore software step is active.

A Software Step exception is pending on the current instruction.

Whenever software step is active, whether the state machine is in the active-not pending state or the active-pending state depends on PSTATE.SS. Table D2-18 shows this.

Table D2-18 State machine states

EL_D using:	Debug exception enable status in the current Exception level and Security state	MDSCR_EL1.SS	PSTATE.SS	State machine state
AArch32	X	X	X	Inactive
AArch64	Disabled	X	X	Inactive
AArch64	Enabled	0	X	Inactive
AArch64	Enabled	1	1	Active-not-pending
AArch64	Enabled	1	0	Active-pending

D2.12.4 Entering the active-not-pending state

Software step can only enter the active-not-pending state from the inactive state.

Software step:

- Enters the active-not-pending state when an `ERET` instruction writes 1 to PSTATE.SS, by copying from SPSR_ELx.SS when it restores PSTATE.
- Might enter the active-not-pending state on exiting Debug state when DSPSR_EL0.SS or DSPSR.SS is 1. See *Exiting Debug state* on page H2-6963.

An `ERET` instruction only copies 1 from SPSR_ELx.SS to PSTATE.SS if all of the following are true:

- MDSCR_EL1.SS is 1.
- EL_D is using AArch64.
- Debug exceptions are disabled from the current Exception level.
- Debug exceptions are enabled from the Exception level that the `ERET` instruction targets.

Otherwise, `ERET` instructions set PSTATE.SS to 0, regardless of the value of SPSR_ELx.SS.

Table D2-19 on page D2-2475 shows this. In the table:

Lock Means the value of (OSLSR_EL1.OSLK == '1' || `DoubleLockStatus()`).

NS Means the *Effective value* of SCR_EL3.NS.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

SDD Means the *Effective value* of MDCR_EL3.SDD. See *Disabling debug exceptions from Secure state* on page D2-2431.

EEL2 Means the *Effective value* of SCR_EL3.EEL2. If ARMv8.4-SecEL2 is not implemented, this is 0.

TGE Means the value of HCR_EL2.TGE. If EL2 is not implemented, the PE behaves as if this is 0.

TDE Means the *Effective value* of MDCR_EL2.TDE. See *Routing debug exceptions* on page D2-2429.

EL1 is using The Execution state when the ELD is EL1.

EL2 is using The Execution state when the ELD is EL2.

Table D2-19 Value an ERET writes to PSTATE.SS

MDSCR_EL1.SS	Lock	NS	SDD	EEL2	TGE	TDE	EL1 is using	EL2 is using	Value an ERET writes to PSTATE.SS
0	X	X	X	X	X	X	X	X	0
1	TRUE	X	X	X	X	X	X	X	0
	FALSE	0	1	X	X	X	X	X	0
			0	0	X	X	AArch32	n/a	0
							AArch64	n/a	See Table D2-20 on page D2-2476
				1	0	0	AArch32	n/a	0
							AArch64	AArch64	See Table D2-20 on page D2-2476
						1	AArch32	AArch32	0
							X	AArch64	See Table D2-21 on page D2-2477
					1	X	n/a	AArch32	0
							n/a	AArch64	See Table D2-21 on page D2-2477
		1	X	X	0	0	AArch32	n/a	0
							AArch64	AArch64	See Table D2-20 on page D2-2476
						1	AArch32	AArch32	0
							X	AArch64	See Table D2-21 on page D2-2477
					1	X	n/a	AArch32	0
							n/a	AArch64	See Table D2-21 on page D2-2477

For:

- If EL_D is EL1 using AArch64, Table D2-20 on page D2-2476 shows the value an ERET writes to PSTATE.SS.
- If EL_D is EL2 using AArch64, Table D2-21 on page D2-2477 shows the value an ERET writes to PSTATE.SS.

In both tables:

From EL Means the Exception level at which the PE executes the ERET instruction.

Target EL Is the target Exception level of the ERET.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

If the ERET is an illegal exception return, the target Exception level of the ERET is the current Exception level. See *Illegal return events from AArch64 state* on page D1-2351.

KDE Is MDSCR_EL1.KDE. See *Enabling debug exceptions from the current Exception level* on page D2-2431.

Table D2-20 Value an ERET writes to PSTATE.SS if EL_D is EL1 using AArch64

From EL	Target EL	KDE	PSTATE.D	SPSR_ELx.D	Software step enable status at: From EL	Software step enable status at: Target EL	Value an ERET writes to PSTATE.SS
EL3	EL3	X	X	X	Disabled	Disabled	0
	EL2	X	X	X	Disabled	Disabled	0
	EL1	0	X	X	Disabled	Disabled	0
		1	X	1	Disabled	Disabled	0
				0	Disabled	Enabled	SPSR_EL3.SS
	EL0	X	X	X	Disabled	Enabled	SPSR_EL3.SS
EL2	EL2	X	X	X	Disabled	Disabled	0
	EL1	0	X	X	Disabled	Disabled	0
		1	X	1	Disabled	Disabled	0
				0	Disabled	Enabled	SPSR_EL2.SS
	EL0	X	X	X	Disabled	Enabled	SPSR_EL2.SS
EL1	EL1	0	X	X	Disabled	Disabled	0
		1	0	X	Enabled[a]	-[b]	0
			1	1	Disabled	Disabled	0
				0	Disabled	Enabled	SPSR_EL1.SS
	EL0	0	X	X	Disabled	Enabled	SPSR_EL1.SS
		1	0	X	Enabled[a]	Enabled	0
			1	X	Disabled	Enabled	SPSR_EL1.SS

a. Because MDSCR_EL1.SS == 1, it means that the ERET is itself being stepped.

b. Depends on SPSR_EL1.D.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D2-21 Value an ERET writes to PSTATE.SS if EL_D is EL2 using AArch64

From EL	Target EL	KDE	PSTATE.D	SPSR_ELx.D	Software step enable status at: From EL	Software step enable status at: Target EL	Value an ERET writes to PSTATE.SS
EL3	EL3	X	X	X	Disabled	Disabled	0
	EL2	0	X	X	Disabled	Disabled	0
		1	X	1	Disabled	Disabled	0
				0	Disabled	Enabled	SPSR_EL3.SS
	EL1	X	X	X	Disabled	Enabled	SPSR_EL3.SS
	EL0	X	X	X	Disabled	Enabled	SPSR_EL3.SS
EL2	EL2	0	X	X	Disabled	Disabled	0
		1	0	X	Enabled[a]	-[b]	0
			1	1	Disabled	Disabled	0
				0	Disabled	Enabled	SPSR_EL2.SS
	EL1	0	X	X	Disabled	Enabled	SPSR_EL2.SS
		1	0	X	Enabled[a]	Enabled	0
			1	X	Disabled	Enabled	SPSR_EL2.SS
	EL0	0	X	X	Disabled	Enabled	SPSR_EL2.SS
		1	0	X	Enabled[a]	Enabled	0
			1	X	Disabled	Enabled	SPSR_EL2.SS
EL1	EL1	X	X	X	Enabled[a]	Enabled	0
	EL0	X	X	X	Enabled[a]	Enabled	0

a. Because MDSCR_EL1.SS == 1, it means that the ERET is itself being stepped.
b. Depends on SPSR_EL2.D.

Note

No AArch32 instruction can set PSTATE.SS to 1.

D2.12.5 Behavior in the active-not-pending state

In this state, the PE does one of the following:

- Executes the instruction to be stepped and either:
  - Completes it without taking a synchronous exception.
  - Takes a synchronous exception if the instruction generates one.
- Takes an asynchronous exception without executing any instructions.
- Enters Debug state because of a *Halting debug event*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the PE executes the instruction without taking any exceptions, then the PE sets PSTATE.SS to 0, meaning that after the instruction has been executed:

- If the instruction has disabled debug by setting PSTATE.D to 1 then software step advances to the inactive state.
- If the instruction disables software step by a direct write to a System register, for example a write to MDSCR_EL1.KDE or MDSCR_EL1.SS, then software step might advance to the inactive state. These writes require explicit synchronization to guarantee their effect. See *Synchronization and the software step state machine* on page D2-2483.
- Otherwise, software step advances to the active-pending state. See *Behavior in the active-pending state* on page D2-2480.

If the PE takes either a synchronous or an asynchronous exception, behavior is as described in one of the following:

- *If the PE takes an exception to an Exception level that is using AArch64*.
- *If the PE takes an exception to an Exception level that is using AArch32* on page D2-2479.

If the PE enters Debug state because of a Halting debug event, behavior is as described in *Entering Debug state and Software Step* on page H2-6935.

If the PE takes an exception to an Exception level that is using AArch64

As part of exception entry, the PE does all of the following:

- Sets SPSR_ELx.SS to 0 or 1, depending on the exception. See Table D2-22.
- It is UNPREDICTABLE whether SPSR_ELx.SS to 0 or 1 when an SError interrupt is taken to ELx without executing the instruction.
- Sets PSTATE.SS to 0. This causes software step to enter either the active-pending state or the inactive state, depending on whether debug exceptions are enabled or disabled from the Exception level that the exception is taken to:

  Enabled Software step enters the active-pending state.

  Disabled Software step enters the inactive state.

  In either case, on taking the exception, a step is complete.
- Sets PSTATE.D to 1.

Table D2-22 Categorization of exceptions, for setting SPSR_ELx.SS to 0 or 1

Exception description	Exceptions	SPSR_ELx.SS
Exceptions whose preferred return address is for the instruction that follows the instruction to be stepped.	Supervisor Call (SVC) exceptions. Hypervisor Call (HVC) exceptions. Secure Monitor Call (SMC) exceptions.	0
Exceptions whose preferred return address is the address of the instruction to be stepped.	All other synchronous exceptions, and asynchronous exceptions that are taken before the instruction to be stepped.	1

Note

If an SMC instruction executed at Non-secure EL1 is trapped to EL2 because HCR_EL2.TSC is 1, the exception is a Trap exception, not a Secure Monitor Call exception, and so SPSR_ELx.SS is set to 1, not 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the PE takes an exception to an Exception level that is using AArch32

This can only happen when all of the following is true:

- EL2 is implemented and is using AArch64, and the *Effective value* of MDCR_EL2.TDE is 1. Because MDCR_EL2.TDE is 1, EL_D is EL2.
- The exception is taken to EL1 using AArch32.

As part of exception entry, the PE sets PSTATE.SS to 0. This causes software step to enter the active-pending state.

Note

- Software step always enters the active-pending state because the exception is taken to an Exception level that debug exceptions are enabled from, EL1. Debug exceptions are enabled from EL1 because EL_D is EL2, and debug exceptions are always enabled from Exception levels that are lower than EL_D.
- AArch32 SPSRs have no SS bit.

Summary of behavior in the active-not-pending state

Table D2-23 summarizes behavior in the active-not-pending state.

Table D2-23 Summary of behavior in the active-not-pending state

Event	Value written to PSTATE.SS	Target Exception level is using:	Details[a]	Value written to SPSR_ELx.SS	Next state
No exception	0	n/a	Disables Software step	n/a	Inactive
			Otherwise	n/a	Active-pending
Exception	0	AArch64	Supervisor Call (SVC) Hypervisor Call (HVC) Secure Monitor Call (SMC)	0	Active-pending or inactive[b]
			Other	1	
		AArch32	All	0	Active-pending

a. For the *No exception* rows, this column shows the effect of the event.
For the *Exception* rows, this column shows the exception taken.

b. Which state software step enters depends on whether debug exceptions are enabled or disabled from the target Exception level. See Figure D2-3 on page D2-2473.

D2.12.6 Entering the active-pending state

Software step enters the active-pending state after any of the following operations, provided that both:

- MDSCR_EL1.SS is 1.
- Debug exceptions are enabled from the Exception level and Security state that execution is in after the operation.

The operations are:

While software step is in the active-not-pending state

The PE either:

- Executing the instruction to be stepped without taking any exceptions.
- Taking an exception.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

While software step is in the active-pending state

The PE takes an asynchronous exception.

While software step is in the inactive state

The PE executes either:

- An ERET instruction when SPSR_ELx.SS is 0.
- An instruction that enables debug by setting PSTATE.D to 0.

Note

If entry to the active-pending state is because of the PE taking an exception, it means that the exception is one that is taken to EL1 when MDCR_EL2.TDE is 1 and EL2 is implemented and enabled in the current Security state. Otherwise, debug exceptions are masked by PSTATE.D, therefore they would be disabled from the target Exception level of the exception.

In addition, software step might enter the active-pending state either:

- After a direct write to a System register, for example a write to MDSCR_EL1.KDE or MDSCR_EL1.SS. These writes require explicit synchronization to guarantee their effect. See *Synchronization and the software step state machine* on page D2-2483.
- On exiting Debug state when DSPSR_EL0.SS or DSPSR.SS is 0. See *Exiting Debug state* on page H2-6963.

D2.12.7 Behavior in the active-pending state

When the PE is in the active-pending state, a Software Step exception is taken before the PE executes an instruction.

The Software Step exception has higher priority than all other types of synchronous exception. However, the prioritization of this exception with respect to any unmasked pending asynchronous exception is not defined by the architecture.

For more information, see the following:

- *Synchronous exception prioritization for exceptions taken to AArch64 state* on page D1-2355.
- *Prioritization and recognition of interrupts* on page D1-2371.
- *Architectural requirements for taking asynchronous exceptions* on page G1-5709.

D2.12.8 Stepping T32 IT instructions

The Armv8-A architecture permits a combination of an IT instruction and another 16-bit T32 instruction to comprise one 32-bit instruction.

For the purpose of stepping an item, it is IMPLEMENTATION DEFINED whether:

- The PE considers this combination to be one instruction.
- The PE considers this combination to be two instructions.

In an implementation that supports the ITD control, that can disable some uses of the IT instruction, it is then IMPLEMENTATION DEFINED whether this behavior depends on the value of the applicable ITD field. For example:

- The PE might consider this combination to be one instruction, regardless of the state of the applicable ITD field.
- The PE might consider this combination to be two instructions, regardless of the state of the applicable ITD field.
- The PE might consider this combination to be one instruction when the applicable ITD field is 1, and two instructions when it is 0.

The applicable ITD field is one of:

- SCTLR_EL1.ITD if execution is at EL0 using AArch32 when EL1 is using AArch64.
- SCTLR.ITD if execution is at EL0 or EL1 when EL1 is using AArch32.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- HSCTLR.ITD if execution is at Non-secure EL2 using AArch32.

D2.12.9 Exception syndrome information and preferred return address

See the following:

- *Exception syndrome information*.
- *Preferred return address* on page D2-2482.

Exception syndrome information

On taking a Software Step exception, the PE records information about the exception in the Exception Syndrome Register (ESR_ELx) at the Exception level the exception is taken to. See *ISS encoding for an exception from a Software Step exception* on page D13-3001 for more information.

If no instruction was stepped because software step entered the active-pending state from the inactive state without passing through the active-not-pending state, then ESR_ELx.{ISV, EX} are set to 0.

When an instruction has been stepped, if the stepped instruction was a conditional Load-Exclusive instruction that failed its Condition code test, then ESR_ELx.EX is set to a CONSTRAINED UNPREDICTABLE choice of 0 or 1.

When an instruction has been stepped, if the stepped instruction was an `ERET` or an `ISB`. then ESR_ELx.ISV is set to a CONSTRAINED UNPREDICTABLE choice of 0 or 1, and ESR_ELx.EX is set to 0.

If the *Effective value* of MDCR_EL2.TDE == 1, EL2 is implemented and enabled in the current Security state, and a different exception is taken before the Software Step exception, then ESR_ELx.ISV is set to a CONSTRAINED UNPREDICTABLE choice of 0 or 1. In this case:

- If ESR_ELx.ISV is set to 1, then ESR_ELx.EX is set to the correct value for the instruction.
- If ESR_ELx.ISV is set to 0, then ESR_ELx.EX is set to zero.

Other than for the cases described above, when an instruction has been stepped:

- ESR_ELx.ISV is set to 1, to indicate that the EX bit is valid.
- The value of ESR_ELx.EX is set according to the instruction stepped. When:
  - The instruction stepped was an instruction other than a Load-Exclusive instruction, an Exception Return instruction, or an `ISB`, and no other exception was taken before the Software Step exception, ESR_ELx.EX is set to 0.
  - The instruction stepped was a Load-Exclusive instruction that was either not conditional or did not fail its Condition code test, ESR_ELx.EX is set to 1.

Note

A Load-Exclusive instruction is any one of the following:

- In the A64 instruction set, any instruction that has a mnemonic starting with either `LDX` or `LDAX`.
- In the A32 and T32 instruction sets, any instruction that has a mnemonic starting with either `LDREX` or `LDAEX`.

Note

An implementation that always sets ISV to 0 and never sets EX is not compliant.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D2-24 summarizes the possible values that the PE can record in ESR_ELx.{ISV, EX}.

Table D2-24 Values that the PE can record in ESR_ELx.{ISV, EX}

Description	ESR_ELx.ISV	ESR_ELx.EX
Syndrome data is not available because no instruction was stepped.	0	0
Syndrome data is available because an instruction was stepped. The instruction stepped was a conditional Load-Exclusive instruction that failed its Condition code test.	1	0 or 1
Syndrome data is available because an instruction was stepped. The instruction stepped was an Exception Return instruction or an `ISB`.	0 or 1	0
A different exception is taken before the Software Step exception.	0	0
	1	Set to the correct value for the instruction.
Syndrome data is available because an instruction was stepped. The instruction stepped was an instruction other than a Load-Exclusive instruction, an Exception Return instruction, or an `ISB`, and no other exception was taken before the Software Step exception.	1	0
Syndrome data is available because an instruction was stepped. The instruction stepped was a Load-Exclusive instruction that was either not conditional or did not fail its Condition code test.	1	1

Preferred return address

The preferred return of a Software Step exception is the address of the instruction that was not executed because the PE took the Software Step exception instead.

D2.12.10 Additional considerations

This section contains the following:

- *Behavior when an ERET instruction is an illegal exception return*.
- *Behavior when the instruction stepped writes a misaligned PC value* on page D2-2483.
- *Stepping code that uses Exclusives monitors* on page D2-2483.
- *Synchronization and the software step state machine* on page D2-2483.

Behavior when an `ERET` instruction is an illegal exception return

If the conditions for entering the active-not-pending state in *Entering the active-not-pending state* on page D2-2474 are met, but the PE executes an `ERET` instruction that is an illegal exception return, the exception return must be taken to the same Exception level that it was taken from. In this scenario, even though the Exception level remains the same before and after the `ERET`, software step can advance from the inactive state to one of the active states. Consider the following case:

1. MDSCR_EL1.SS is 1 and software step is inactive. The current Exception level is EL1 using AArch64, the OS Lock and OS Double Lock are unlocked, and MDCR_EL2.TDE is 0, MDSCR_EL1.KDE is 1, and PSTATE.D is 1.

   PSTATE.D == 1 is the reason why software step is inactive, because PSTATE.D == 1 means that debug exceptions are disabled from the current Exception level.

2. The PE executes an `ERET` instruction.

3. The intended target of the `ERET` is EL2. This means that the `ERET` is an illegal exception return because the intended target is higher than the Exception level the `ERET` it is executed at. In this case, the `ERET` must target EL1 instead of EL2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If SPSR_EL1.D is 0, then on the ERET PSTATE.D becomes 0 and debug exceptions become enabled from the current Exception level. Software step therefore advances from the inactive state to one of the active states.

Which active state software step advances to depends on whether SPSR_ELx.SS is 1 or 0:

- If SPSR_ELx.SS is 1, software step advances to the active-not-pending state.

  In this case, an Illegal Execution state exception is pending on the instruction to be stepped, and the PE takes the Illegal Execution state exception instead of executing the instruction to be stepped.

- If SPSR_ELx.SS is 0, software step advances to the active-pending state.

  In this case, a Software Step exception and an Illegal Execution state exception are both pending. The Software Step exception has higher priority. On taking the Software Step exception, the PE sets SPSR_ELx.IL to 1.

Note

Synchronous exception prioritization for exceptions taken to AArch64 state on page D1-2355 shows the relative priorities of synchronous exceptions.

Behavior when the instruction stepped writes a misaligned PC value

An indirect branch that writes a misaligned PC value might generate a PC alignment fault exception at the target of the branch. However, if the indirect branch is stepped using software step, the PE takes a Software Step exception instead, because the Software Step exception has higher priority. Behavior on returning from the Software Step exception depends on which Execution state the Exception level being returned to is using:

AArch64 A PC alignment fault exception is generated.

AArch32 The return from the Software Step exception forces the PC to the correct alignment, and no PC alignment fault exception is generated.

Debugger software must therefore take care when using software step to single-step an indirect branch instruction executed in AArch32 state, that it does not hide a PC alignment fault exception.

Stepping code that uses Exclusives monitors

The Armv8-A architecture provides no mechanism for preserving the state of the Exclusives monitors when a Load-Exclusive or a Store-Exclusive instruction is stepped.

However, for certain progressions through the software step state machine, on taking a Software Step exception, the PE provides an indication of whether the instruction stepped was a Load-Exclusive instruction.

Debugger software can use this to detect the state of the Exclusives monitors. For example, if the PE reports that the instruction stepped was a Load-Exclusive instruction, the debugger is aware that the next Store-Exclusive operation will fail, because all Exclusives monitors are cleared on returning from the Software Step exception. The debugger must then take action to ensure that the code being stepped makes forwards progress.

For more information on how the PE reports whether the instruction stepped was a Load-Exclusive instruction, see *Exception syndrome information and preferred return address* on page D2-2481.

Synchronization and the software step state machine

Any of the following can cause transitions between software step states:

- A direct write to a System register.
- A direct write to a Special-purpose register.
- A write to an external debug register that affects the routing of debug exceptions.

Because the software step state machine indirectly reads these registers, it is not guaranteed to observe any new values until after a *Context synchronization event* has occurred.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In the time between a write to one of these registers and the next *Context synchronization event*, it is CONSTRAINED UNPREDICTABLE whether software step uses the state of the PE before the write, or the state of the PE after the write.

After a *Context synchronization event*, the state machine must use the state of the PE after the write.

Example D2-3

1. Software changes MDSCR_EL1.SS from 0 to 1 when debug exceptions are enabled.
2. The PE executes some instructions.
3. A *Context synchronization event* occurs.

During step 2, it is CONSTRAINED UNPREDICTABLE whether software step remains in the inactive state, as if MDSCR_EL1.SS is 0, or enters the active-pending state because MDSCR_EL1.SS is 1. If it is in the:

- Inactive state, then after the *Context synchronization event*, it must enter the active-pending state.
- Active-pending state, the PE might take a Software Step exception before the *Context synchronization event*.

Note

A direct write to a Special-purpose register does not require explicit synchronization.

D2.12.11 Pseudocode description of Software Step exceptions

`SSAdvance()` advances software step from the active-not-pending state to the active-pending state, by setting PSTATE.SS to 0. It is called on completing execution of each instruction.

`CheckSoftwareStep()` checks whether software step is in the active-pending state, and if it is, generates a Software Step exception. It is called before each instruction executed, regardless of Execution state, before checking for any other synchronous exceptions.

`DebugExceptionReturnSS()` returns the value to write to PSTATE.SS on an exception return or an exit from Debug state. See *Entering the active-not-pending state* on page D2-2474.

These functions are defined in Chapter J1 *Armv8 Pseudocode*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D2.13 Synchronization and debug exceptions

The behavior of debug depends on all of the following:

- The state of the external debug authentication interface.
- Indirect reads of:
  - External debug registers.
  - System registers, including system debug registers.
  - Special-purpose registers.

If a change is made to any of these, the effect of that change on debug exception generation cannot be relied on until after a *Context synchronization event* has occurred. Similarly, the effect of the change on the software step state machine cannot be relied on until after a *Context synchronization event* has occurred.

For any instructions executed between the time when the change is made and the time when the next *Context synchronization event* occurs, it is CONSTRAINED UNPREDICTABLE whether debug uses the state of the PE before the change, or the state of the PE after the change.

Example D2-4

1. Software changes MDSCR_EL1.MDE from 0 to 1.
2. An instruction is executed, that would cause a Breakpoint exception if self-hosted debug uses the state of the PE after the change.
3. A *Context synchronization event* occurs.

In this case, it is CONSTRAINED UNPREDICTABLE whether the instruction generates a Breakpoint exception.

Example D2-5

1. Software unlocks the OS Lock.
2. The PE executes some instructions.
3. A *Context synchronization event* occurs.

During the time when the PE is executing some instructions, step 2, it is CONSTRAINED UNPREDICTABLE whether debug exceptions other than Breakpoint Instruction exceptions can be generated.

Note

Some register updates are self-synchronizing. Others require an explicit *Context synchronization event*. For more information, see:

- *Accessing PSTATE fields* on page D1-2331.
- *Synchronization requirements for AArch64 System registers* on page D13-2869.
- *Synchronization of changes to the external debug registers* on page H8-7050.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter D3
AArch64 Self-hosted Trace

This chapter describes the AArch64 self-hosted trace:

Introductory information:

- *About self-hosted trace* on page D3-2488.
- *Trace sinks* on page D3-2488.
- *Register controls to enable self-hosted trace* on page D3-2488.

Prohibited regions in trace:

- *Controls to prohibit trace at Exception levels* on page D3-2489.
- *Self-hosted trace and visibility of virtual data* on page D3-2490.

Timestamps and Synchronization:

- *Self-hosted trace timestamps* on page D3-2491.
- *Synchronization in self-hosted trace* on page D3-2493.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D3.1 About self-hosted trace

A PE Trace Unit generates trace data to describe the program flow of the PE.

The PE Trace Unit may be an implementation of a standard Arm Embedded Trace Macrocell (ETM), or another type of Arm Trace Architecture, or an IMPLEMENTATION DEFINED trace function.

If an Armv8.4-compliant PE implements an ETM Architecture PE Trace Unit, ARMv8.4-Trace extension must be implemented.

If an Armv8.4-compliant PE implements a Trace Unit that is not an ETM Architecture PE Trace Unit, Arm recommends that ARMv8.4-Trace extension is implemented, but this is not mandatory.

Self-hosted trace happens when the agent controlling the trace collection is part of the same software stack as the software being traced. The agent controls prohibited regions. The information collected by the agent is sent to a trace sink.

If the self-hosted trace extensions are implemented, the PE Trace Unit must implement the system register interface. The PE Trace Unit and the PE must have the same view of the debug authentication interface. If ARMv8.4-Trace is implemented, `ExternalNoninvasiveDebugEnabled`() is always TRUE.

D3.1.1 Trace sinks

The PE Trace Unit sends the trace data to a trace sink. A system might include multiple trace sinks, and allow software to configure which trace sink or sinks are used.

An example of an internal trace sink is an Embedded Trace Router (ETR), which allows software to define a buffer in memory. Trace data is written to this buffer.

Arm recommends that a system that a system that includes ARMv8.4-Trace incorporates an ETR, and follows the system architecture described by the *CoreSight Base System Architecture (CS-BSA)*.

The self-hosted trace extensions do not describe the programmers' model trace sinks.

D3.1.2 Register controls to enable self-hosted trace

If ARMv8.4-Trace is implemented, self-hosted trace is enabled if one of the following is true:

- EDSCR.TFO == 0.
- EDSCR.TFO == 1, EL3 is implemented, MDCR_EL3.STE == 1 and `ExternalSecureNoninvasiveDebugEnabled`() == FALSE.
- EDSCR.TFO ==1, EL3 is not implemented, the PE executes in Secure state and `ExternalSecureNoninvasiveDebugEnabled`() = FALSE.

The pseudocode function `SelfHostedTraceEnabled`() shows these rules.

If ARMv8.4-Trace is not implemented, `SelfHostedTraceEnabled`() returns FALSE.

While `SelfHostedTraceEnabled`() == FALSE, `ExternalSecureNoninvasiveDebugEnabled`() and `ExternalNoninvasiveDebugEnabled`() control whether tracing is prohibited or allowed in each Security state.

The self-hosted trace extensions do not provide any mechanism to control software access to the PE Trace Unit external debug interface.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D3.2 Prohibited regions in self-hosted trace

Trace is not generated in prohibited regions. The pseudocode function `TraceAllowed`() indicates whether tracing is allowed in the current Security state and Exception level.

The IMPLEMENTATION DEFINED debug authentication interface can allow an external agent to disable the self-hosted trace extension.

If `SelfHostedTraceEnabled`() == TRUE, tracing is prohibited in Secure state when MDCR_EL3.STE == 0. If ARMv8.4-Trace is implemented but not enabled, tracing is prohibited in Secure state when `ExternalSecureNoninvasiveDebugEnabled`() == FALSE.

D3.2.1 Controls to prohibit trace at Exception levels

If `SelfHostedTraceEnabled`() == TRUE, TRFCR_EL1 and TRFCR_EL2 control whether trace is prohibited at an Exception level. While `SelfHostedTraceEnabled`() == FALSE, the registers TRFCR_EL1 and TRFCR_EL2 are ignored.

If `SelfHostedTraceEnabled`() == TRUE, tracing is prohibited at EL0 if one of the following is true:

- The Effective value of HCR_EL2.TGE == 0 and TRFCR_EL1.E0TRE == 0.
- The Effective value of HCR_EL2.TGE == 1 and TRFCR_EL2.E0HTRE == 0.

If `SelfHostedTraceEnabled`() == TRUE, tracing is prohibited at EL1 if TRFCR_EL1.E1TRE == 0.

If `SelfHostedTraceEnabled`() == TRUE, tracing is prohibited at EL2 if TRFCR_EL2.E2TRE == 0.

If `SelfHostedTraceEnabled`() == TRUE, tracing is prohibited at EL3 if one of the following is true:

- EL3 is using AArch64 state.
- EL3 is using AArch32 state and TRFCR.E1TRE == 0.

The pseudocode `TraceAllowed`() shows the above rules.

If `SelfHostedTraceEnabled`() == TRUE, Table D3-1 shows when export of PMU events to the ETM is prohibited.

Table D3-1 Export of PMU events prohibited

Tracing prohibited in all of these Exception levels	HCR_EL2.TGE	Export of PMU events Attributable to this Exception level to the ETM is prohibited at these Exception levels.
EL0, EL2, EL3	1	EL0, EL2, EL3
EL0, EL1, EL2, EL3	0	EL0, EL1, EL2, EL3
EL1, EL2, EL3	0	EL1, EL2, EL3
EL2, EL3	0	EL2, EL3
EL3	0	EL3

If `SelfHostedTraceEnabled`() == TRUE, Table D3-2 on page D3-2490 shows the prohibited regions by exception level and state.

In the table:

STE Means the Effective value of MDCR_EL3.STE or SDCR.STE, as applicable.

EEL2 Means the Effective value of SCR_EL3.EEL2.

TGE Means the Effective value of HCR_EL2.TGE.

P Means prohibited.

E2TRE Means allowed if TRFCR_EL2.E2TRE == 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

E1TRE Means allowed if TRFCR_EL1.E1TRE == 1.

E0HTRE Means allowed if TRFCR_EL2.E0HTRE == 1.

E0TRE Means allowed if TRFCR_EL1.E0TRE == 1.

n/a Not applicable.

Table D3-2 Prohibited regions

Controls					Tracing prohibited at			
State	**STE**	**EL3 using**	**EEL2**	**TGE**	**EL3**	**EL2**	**EL1**	**EL0**
Non-secure	X	X	X	0	n/a	E2TRE	E1TRE	E0TRE
	X	X	X	1	n/a	E2TRE	n/a	E0HTRE
Secure	0	X	X	X	P	P	P	P
	1	AArch64	0	|X	P	n/a	E1TRE	E0TRE
			1	0	P	E2TRE	E1TRE	E0TRE
			1	1	P	E2TRE	n/a	E0HTRE
		AArch32	X	X	E1TRE	n/a	n/a	E0TRE

D3.2.2 Self-hosted trace and visibility of virtual data

A hypervisor can use TRFCR_EL2.CX to control visibility of CONTEXTIDR_EL2 and VTTBR_EL2.VMID.

If `SelfHostedTraceEnabled`() == TRUE and TRFCR_EL2.CX == 0, or if EL2 is not implemented:

- The values of CONTEXTIDR_EL2 and VTTBR_EL2.VMID are not traced.
- Comparisons between CONTEXTIDR_EL2 and VTTBR_EL2.VMID do not match and results of comparison are not exposed through the comparators.

The PE Trace Unit may either prohibit trace for these values, or may record a CONTEXTIDR_EL2 or VTTBR_EL2.VMID value of zero in the trace.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D3.3 Self-hosted trace timestamps

The trace timestamp is a value that represents the passage of time in real-time. It is calculated from a counter which increments all the time, when the PE is generating trace and when the PE is in a prohibited region.

While `SelfHostedTraceEnabled`() == FALSE, the external trace provides the trace timestamp. If the external trace is a standard CoreSight system, the relationship between CoreSight time and the Generic Timer counter is IMPLEMENTATION DEFINED.

When `SelfHostedTraceEnabled`() == TRUE, the trace timestamp is one of the following:

- The physical counter value CNTPCT_EL0.
- An offset physical counter value, which is calculated from the physical counter value CNTPCT_EL0, minus an offset CNTPOFF_EL2. When any of the following are true, the *Effective value* of CNTPOFF_EL2 is 0 for all trace and SPE purposes:
  - EL3 is using AArch32.
  - EL2 is not implemented.
  - ARMv8.6-ECV is not implemented.
  - EL2 is enabled in the current Security state and is using AArch32.
  - CNTHCTL_EL2.ECV is 0.
  - SCR_EL3.ECVEn is 0.
- A virtual counter value, which is calculated from the physical counter value CNTPCT_EL0, minus an offset CNTVOFF_EL2.

The fields TRFCR_EL2.TS, HTRFCR.TS, TRFCR_EL1.TS and TRFCR.TS control which counter is used for self-hosted trace.

The timestamp used for trace is shown in Table D3-3.

Table D3-3 Timestamp used for trace.

`SelfHostedTraceEnabled`()	TRFCR_EL2.TS	TRFCR_EL1.TS	Timestamp traced
FALSE	xx	xx	CoreSight time
TRUE	0b00	0b01	CNTPCT_EL0 - CNTVOFF_EL2
	0b00	0b10	CNTPCT_EL0 - CNTPOFF_EL2[a]
	0b00	0b11	CNTPCT_EL0
	0b01	xx	CNTPCT_EL0 - CNTVOFF_EL2
	0b10	xx	CNTPCT_EL0 - CNTPOFF_EL2[a]
	0b11	xx	CNTPCT_EL0

a. This register is only implemented when ARMv8.6-ECV is implemented.

Note

The counter value used for the trace timestamp is not affected by the value of HCR_EL2.E2H, or whether EL2 is enabled or disabled in the current Security state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D3.4 Synchronization in self-hosted trace

The PE Trace Unit is an indirect observer of the System registers.

While `SelfHostedTraceEnabled`() == TRUE, indirect reads of the trace filter control fields, TRFCR_EL1.{E1TRE, E0TRE} and TRFCR_EL2.{E2TRE, E0HTRE} are treated as indirect reads made by the instruction being traced, and are subject to the standard requirements for synchronization of System register accesses.

The `TSB CSYNC` operation is used to ensure that a trace operation, due to a PE Trace Unit generating trace for an instruction has completed. The `TSB CSYNC` operation may be reordered with respect to other instructions, so must be combined with at least one context synchronization event to ensure the operations are executed in the required order. This means that a direct write to TRFCR_EL1 or TRFCR_EL2 is guaranteed to be observed by the PE Trace Unit only after a subsequent Context synchronization event. For more information, see *Trace Synchronization Barrier (TSB CSYNC)* on page B2-138.

While `SelfHostedTraceEnabled`() == FALSE, the PE Trace Unit might impose stronger synchronization requirements.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter D4
The AArch64 System Level Memory Model

This chapter provides a system level view of the general features of the memory system. It contains the following sections:

- *About the memory system architecture* on page D4-2496.
- *Address space* on page D4-2497.
- *Mixed-endian support* on page D4-2498.
- *Cache support* on page D4-2499.
- *External aborts* on page D4-2524.
- *Memory barrier instructions* on page D4-2526.
- *Pseudocode description of general memory System instructions* on page D4-2527.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D4.1 About the memory system architecture

The Arm architecture supports different implementation choices for the memory system microarchitecture and memory hierarchy, depending on the requirements of the system being implemented. In this respect, the memory system architecture describes a design space in which an implementation is made. The architecture does not prescribe a particular form for the memory systems. Key concepts are abstracted in a way that permits implementation choices to be made while enabling the development of common software routines that do not have to be specific to a particular microarchitectural form of the memory system. For more information about the concept of a hierarchical memory system see *Memory hierarchy* on page B2-143.

If ARMv8.5-MemTag is implemented and enabled the definitions of the memory model which apply to data accesses and data apply to Allocation Tag accesses and Allocation tags, unless otherwise specified in Chapter D6 *Armv8.5 Memory Tagging Extension*.

D4.1.1 Form of the memory system architecture

The Armv8 A-profile architecture includes a *Virtual Memory System Architecture* (VMSA). Chapter D5 *The AArch64 Virtual Memory System Architecture* describes the AArch64 view of the VMSA.

D4.1.2 Memory attributes

Memory types and attributes on page B2-153 describes the memory attributes, including how different memory types have different attributes. Each location in memory has a set of memory attributes, and the translation tables define the virtual memory locations, and the attributes for each location.

Table D4-1 shows the memory attributes that are visible at the system level.

Table D4-1 Memory attribute summary

Memory type	Shareability	Cacheability
Device[a]	Outer Shareable	Non-cacheable.
Normal	One of: • Non-shareable. • Inner Shareable. • Outer Shareable.	One of[b]: • Non-cacheable. • Write-Through Cacheable. • Write-Back Cacheable.

a. Takes additional attributes, see *Device memory* on page B2-157.

b. See also *Cacheability, cache allocation hints, and cache transient hints* on page D4-2502.

For more information on cacheability and shareability see *Shareable Normal memory* on page B2-154, *Non-shareable Normal memory* on page B2-155, and *Caches and memory hierarchy* on page B2-143.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D4.2 Address space

The Armv8 architecture is designed to support a wide range of applications with different memory requirements. It supports a range of *physical address* (PA) sizes, and provides associated control and identification mechanisms. For more information, see *Address size configuration* on page D5-2547.

D4.2.1 Virtual address space overflow

When a PE performs a *Simple sequential execution* of instructions, it calculates:

```
(address_of_current_instruction) + (size_of_executed_instruction)
```

This calculation is performed after each instruction to determine which instruction to execute next.

If the address calculation performed after executing an instruction overflows `0xFFFF FFFF FFFF FFFF`, the program counter becomes UNKNOWN.

Note

Address tags are not propagated to the program counter, so the tag does not affect the address calculation.

Where an instruction accesses a sequential set of bytes that crosses the `0xFFFF_FFFF_FFFF_FFFF` boundary when tagged addresses are not used, or the `0xxxFF_FFFF_FFFF_FFFF` boundary when tagged addresses are used, then the virtual address accessed for the bytes above this boundary is UNKNOWN. When tagged addresses are used, the value of the tag associated with the address also becomes UNKNOWN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D4.3 Mixed-endian support

A control bit, SCTLR_EL1.E0E is provided to allow the endianness of explicit data accesses made while executing at EL0 to be controlled independently of those made while executing at EL1. Table D4-2 shows the endianness of explicit data accesses and translation table walks.

Table D4-2 Endianness support

Exception level	Explicit data accesses	Stage 1 translation table walks	Stage 2 translation table walks
EL0	SCTLR_EL1.E0E	SCTLR_EL1.EE	SCTLR_EL2.EE
EL1	SCTLR_EL1.EE	SCTLR_EL1.EE	SCTLR_EL2.EE
EL2	SCTLR_EL2.EE	SCTLR_EL2.EE	N/A
EL3	SCTLR_EL3.EE	SCTLR_EL3.EE	N/A

Note

SCTLR_EL1.E0E has no effect on the endianness of the `LDTR`, `LDTRH`, `LDTRSH`, and `LDTRSW` instructions, or on the endianness of the `STTR` and `STTRH` instructions, when these are executed at EL1.

AArch64 state provides the following options for endianness support:

- All Exception levels support mixed-endianness:
  - SCTLR_ELx.EE is RW and SCTLR_EL1.E0E is RW.
- Only EL0 supports mixed-endianness and EL1, EL2, and EL3 support only little-endianness:
  - SCTLR_ELx.EE is RES0 and SCTLR_EL1.E0E is RW.
- Only EL0 supports mixed-endianness and EL1, EL2, and EL3 support only big-endianness:
  - SCTLR_ELx.EE is RES1 and SCTLR_EL1.E0E is RW.
- All Exception levels support only little-endianness:
  - SCTLR_ELx.EE is RES0 and SCTLR_EL1.E0E is res0.
- All Exception levels support only big-endianness:
  - SCTLR_ELx.EE is RES1 and SCTLR_EL1.E0E is res1.

If mixed endian support is implemented for an Exception level using AArch32, endianness is controlled by PSTATE.E. For exception returns to AArch32 state, PSTATE.E is copied from SPSR_ELx.E. If the target Exception level supports only little-endian accesses, SPSR_ELx.E is RES0. If the target Exception level supports only big-endian accesses, SPSR_ELx.E is RES1. PSTATE.E is ignored in AArch64 state.

The `BigEndian()` function determines whether the current Exception level and Execution state are using big-endian data. This function is defined in Chapter J1 *Armv8 Pseudocode*.

For more information about endianness in the Arm architecture see *Endian support* on page B2-150.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D4.4 Cache support

This section describes the Armv8 cache identification and control mechanisms, and the A64 cache maintenance instructions, in the following sections:

- *General behavior of the caches*.
- *Cache identification* on page D4-2500.
- *Cacheability, cache allocation hints, and cache transient hints* on page D4-2502.
- *Enabling and disabling the caching of memory accesses* on page D4-2503.
- *Behavior of caches at reset* on page D4-2505
- *Non-cacheable accesses and instruction caches* on page D4-2505.
- *About cache maintenance in AArch64 state* on page D4-2506.
- *A64 Cache maintenance instructions* on page D4-2510
- *Data cache zero instruction* on page D4-2520.
- *Cache lockdown* on page D4-2520.
- *System level caches* on page D4-2522.
- *Branch prediction* on page D4-2522.
- *Execution and data prediction restriction System instructions* on page D4-2522.

See also *Caches in a VMSAv8-64 implementation* on page D5-2685.

D4.4.1 General behavior of the caches

When a memory location has a Normal Cacheable memory attribute, determining whether a copy of the memory location is held in a cache still depends on many aspects of the implementation. The following non-exhaustive list of factors might be involved:

- The size, line length, and associativity of the cache.
- The cache allocation algorithm.
- Activity by other elements of the system that can access the memory.
- Speculative instruction fetching algorithms.
- Speculative data fetching algorithms.
- Interrupt behaviors.

Given this range of factors, and the large variety of cache systems that might be implemented, the architecture cannot guarantee whether:

- A memory location present in the cache remains in the cache.
- A memory location not present in the cache is brought into the cache.

Instead, the following principles apply to the behavior of caches:

- The architecture has a concept of an entry locked down in the cache. How lockdown is achieved is IMPLEMENTATION DEFINED, and lockdown might not be supported by:
  - A particular implementation.
  - Some memory attributes.
- An unlocked entry in a cache might not remain in that cache. The architecture does not guarantee that an unlocked cache entry remains in the cache or remains incoherent with the rest of memory. Software must not assume that an unlocked item that remains in the cache remains dirty.
- A locked entry in a cache is guaranteed to remain in that cache. The architecture does not guarantee that a locked cache entry remains incoherent with the rest of memory, that is, it might not remain dirty.

  Note

  For more information, see *The interaction of cache lockdown with cache maintenance instructions* on page D4-2521.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Any memory location that has a Normal Cacheable attribute at either the current Exception level or at a higher Exception level can be allocated to a cache at any time.
- It is guaranteed that no memory location that does not have a Normal Cacheable attribute is allocated into the cache.
- It is guaranteed that no memory location is allocated to the cache if it has a Normal Non-cacheable attribute or any type of Device memory attribute in both:
  - The translation regime at the current Exception level.
  - The translation regime at any higher Exception level.
- For data accesses, any memory location with a Normal Inner Shareable or Normal Outer Shareable attribute is guaranteed to be coherent with all masters in its shareability domain.
- Any memory location is not guaranteed to remain incoherent with the rest of memory.
- The eviction of a cache entry from a cache level can overwrite memory that has been written by another observer only if the entry contains a memory location that has been written to by an observer in the shareability domain of that memory location. The maximum size of the memory that can be overwritten is called the *Cache Write-back Granule*. In some implementations the CTR_EL0 identifies the Cache Write-back Granule.
- The allocation of a memory location into a cache cannot cause the most recent value of that memory location to become invisible to an observer if it was previously visible to that observer.

Note

The Cacheability attribute of an address is determined by the applicable translation table entry for that address, as modified by any applicable System register Cacheability controls, such as the SCTLR_EL1.{I, C} controls.

For the purpose of these principles, a cache entry covers at least 16 bytes and no more than 2KB of contiguous address space, aligned to the size of the cache entry.

D4.4.2 Cache identification

The Armv8 cache identification registers describe the implemented caches that are affected by cache maintenance instructions executed on the PE. This includes the cache maintenance instructions that:

- Affect the entire cache, for example `IC IALLU`.
- Operate by VA, for example `IC IVAU`.
- Operate by set/way, for example `DC ISW`.

The cache identification registers are:

- The Cache Type Register, CTR_EL0, that defines:
  - The minimum line length of any of the instruction caches affected by the instruction cache maintenance instructions.
  - The minimum line length of any of the data or unified caches, affected by the data cache maintenance instruction.
  - The cache indexing and tagging policy of the Level 1 instruction cache.

  Note

  It is IMPLEMENTATION DEFINED whether caches beyond the PoC will be reported by this mechanism, and because of the possible existence of system caches some caches before the PoC might not be reported. For more information about system caches see *System level caches* on page D4-2522.

- A single Cache Level ID Register, CLIDR_EL1, that defines:
  - The type of cache that is implemented and can be maintained using the architected cache maintenance instructions that operate by set/way or operate on the entire cache at each cache level, up to the maximum of seven levels.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

  - The *Level of Coherence* (LoC) for the caches. See *Terms used in describing the cache maintenance instructions* on page D4-2506 for the definition of LoC.
  - The *Level of Unification Uniprocessor* (LoUU) for the caches. See *Terms used in describing the cache maintenance instructions* on page D4-2506 for the definition of LoUU.
  - An optional ICB field to indicate the boundary between the caches use for caching Inner Cacheable memory regions and those used only for caching Outer Cacheable regions.

- A single Cache Size Selection Register, CSSELR_EL1, that selects the cache level and cache type of the current Cache Size Identification Register.

- For each implemented cache that is identifiable by this mechanism, across all the levels of caching, a Cache Size Identification Register, CCSIDR_EL1, that defines:
  - Whether the cache supports Write-Through, Write-Back, Read-Allocate and Write-Allocate.
  - The number of sets, associativity and line length of the cache. See *Terms used in describing the cache maintenance instructions* on page D4-2506 for a definition of these terms.

  Note

  From Armv8.3, multiple formats of the Cache Size Identification Register are supported. For more information, see *Possible formats of the Cache Size Identification Register, CCSIDR_EL1*.

To determine the cache topology associated with a PE:

1. Read the Cache Type Register to find the indexing and tagging policy used for the Level 1 instruction cache. This register also provides the size of the smallest cache lines used for the instruction caches, and for the data and unified caches. These values are used in cache maintenance instructions.

2. Read the Cache Level ID Register to find what caches are implemented. The register includes seven Cache type fields, for cache levels 1 to 7. Scanning these fields, starting from Level 1, identifies the instruction, data or unified caches implemented at each level. This scan ends when it reaches a level at which no caches are defined. The Cache Level ID Register also specifies the Level of Unification (LoU) and the Level of Coherence (LoC) for the cache implementation.

3. For each cache identified at stage 2:
   - Write to the Cache Size Selection Register to select the required cache. A cache is identified by its level, and whether it is:
     - An instruction cache.
     - A data or unified cache.
   - Read the Cache Size Identification Register to find details of the cache.

Possible formats of the Cache Size Identification Register, CCSIDR_EL1

From Armv8.3, the Cache Size Identification Register, CCSIDR_EL1 has two different formats available for defining the number of sets and associativity of the cache. For a definition of these terms, see *Terms used in describing the cache maintenance instructions* on page D4-2506.

When ARMv8.3-CCIDX is implemented:

- CCSIDR_EL1 is a 64-bit register.

- The length of the CCSIDR_EL1.Assoc field is 21 bits. This limits the associativity of the currently selected cache to 2^{21}.

- The length of the CCSIDR_EL1.NumSets field is 24 bits. This limits the number of sets in the currently selected cache to 2^{24}.

This is the 64-bit format of the Cache Size Identification Register.

When ARMv8.3-CCIDX is not implemented:

- CCSIDR_EL1 is a 32-bit register.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The length of the CCSIDR_EL1.Assoc field is 10 bits. This limits the associativity of the currently selected cache to 2^{10}.
- The length of the CCSIDR_EL1.NumSets field is 15 bits. This limits the number of sets in the currently selected cache to 2^{15}.

This is the 32-bit format of the Cache Size Identification Register.

When one of these formats is implemented, it is implemented across all the levels of caching.

D4.4.3 Cacheability, cache allocation hints, and cache transient hints

Cacheability only applies to Normal memory, and can be defined independently for Inner and Outer cache locations. All types of Device memory are always treated as Non-cacheable.

As described in *Memory types and attributes* on page B2-153, the memory attributes include a cacheability attribute that is one of:

- Non-cacheable.
- Write-Through cacheable.
- Write-Back cacheable.

In Armv8, Cacheability attributes other than Non-cacheable can be complemented by a *cache allocation hint*. This is an indication to the memory system of whether allocating a value to a cache is likely to improve performance. In addition, it is IMPLEMENTATION DEFINED whether a *cache transient hint* is supported, see *Transient cacheability hint*.

The cache allocation hints are assigned independently for read and write accesses, and therefore when the Transient hint is supported the following cache allocation hints can be assigned:

For read accesses: Read-Allocate, Transient Read-Allocate, or No Read-Allocate.

For write accesses: Write-Allocate, Transient Write-Allocate, or No Write-Allocate.

Note

- A Cacheable location with both No Read-Allocate and No Write-Allocate hints is not the same as a Non-cacheable location. A Non-cacheable location has coherency guarantees for all observers within the system that do not apply for a location that is Cacheable, No Read-Allocate, No Write-Allocate.
- Implementations can use the cache allocation hints to limit cache pollution to a part of a cache, such as to a subset of ways.
- For VMSAv8-64 translation table walks, the TCR_ELx.{IRGN*n*, ORGN*n*} fields define the memory attributes of the translation tables, including the cacheability. However, this assignment supports only a subset of the cacheability attributes described in this section.

The architecture does not require an implementation to make any use of cache allocation hints. This means an implementation might not make any distinction between memory locations with attributes that differ only in their cache allocation hint.

Transient cacheability hint

In Armv8, it is IMPLEMENTATION DEFINED whether a Transient hint is supported. In an implementation that supports the Transient hint, the Transient hint is a qualifier of the cache allocation hints, and indicates that the benefit of caching is for a relatively short period. It indicates that it might be better to restrict allocation of transient entries, to avoid possibly casting-out other, less transient, entries.

Note

The architecture does not specify what is meant by a *relatively short period*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The description of the AArch64 MAIR_EL1, MAIR_EL2, and MAIR_EL3 registers, and the AArch32 MAIR0, MAIR1, HMAIR0, and HMAIR1 registers, includes the assignment of the Transient hint in an implementation that supports this option. In this assignment:

- The Transient hint is defined independently for Inner Cacheable and Outer Cacheable memory regions.
- A single Transient hint applies to both read and write accesses to a memory region.

D4.4.4 Enabling and disabling the caching of memory accesses

In Armv8, Cacheability control fields can force all memory locations with the Normal memory type to be treated as Non-cacheable, regardless of their assigned Cacheability attribute. Independent controls are provided for each stage of address translation, with separate controls for:

- Data accesses. These controls also apply to accesses to the translation tables.
- Instruction accesses.

Note

These Cacheability controls replace the cache enable controls provided in previous versions of the Arm architecture.

The Cacheability control fields and their effects are as follows:

For the EL1&0 translation regime

- When the value of SCTLR_EL1.C is 0:
  - All stage 1 translations for data accesses to Normal memory are Non-cacheable.
  - All accesses to the EL1&0 stage 1 translation tables are Non-cacheable.
- When the value of SCTLR_EL1.I is 0:
  - All stage 1 translations for instruction accesses to Normal memory are Non-cacheable.
- When the value of HCR_EL2.CD is 1:
  - All stage 2 translations for data accesses to Normal memory are Non-cacheable.
  - All accesses to the EL1&0 stage 2 translation tables are Non-cacheable.
- When the value of HCR_EL2.ID is 1:
  - All stage 2 translations for instruction accesses to Normal memory are Non-cacheable.
- When the value of HCR_EL2.DC is 1, all stage 1 translations and all accesses to the EL1&0 stage 1 translation tables, are treated as accesses to Normal Non-shareable Inner Write-Back Cacheable Read-Allocate Write-Allocate, Outer Write-Back Cacheable Read-Allocate Write-Allocate memory, regardless of the value of SCTLR_EL1.{I, C}. This applies to translations for both data and instruction accesses.

Note

- The stage 1 and stage 2 cacheability attributes are combined as described in *Combining the stage 1 and stage 2 cacheability attributes for Normal memory* on page D5-2636.
- The SCTLR_EL1.{C, I} and HCR_EL2.DC fields have no effect on the EL2, EL2&0, and EL3 translation regimes.
- The HCR_EL2.{ID, CD} fields affect only stage 2 of the EL1&0 translation regime.
- When EL2 is using AArch64 and EL1 is using AArch32, the HCR_EL2.{ID, CD, DC} controls apply as described here, but the EL1 controls are SCTLR.{C, I}.

For the EL2 translation regime

- When the value of SCTLR_EL2.C is 0:
  - All data accesses to Normal memory using the EL2 translation regime are Non-cacheable.
  - All accesses to the EL2 translation tables are Non-cacheable.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- When the value of SCTLR_EL2.I is 0:
  - All instruction accesses to Normal memory using the EL2 translation regime are Non-cacheable.

Note

The SCTLR_EL2.{I, C} fields have no effect on the EL1&0 and EL3 translation regimes.

For the EL2&0 translation regime

- When the value of SCTLR_EL2.C is 0:
  - All stage 1 translations for data accesses to Normal memory are Non-cacheable.
  - All accesses to the EL2&0 stage 1 translation tables are Non-cacheable.
- When the value of SCTLR_EL2.I is 0:
  - All stage 1 translations for instruction accesses to Normal memory are Non-cacheable.

For the EL3 translation regime

- When the value of SCTLR_EL3.C is 0:
  - All data accesses to Normal memory using the EL3 translation regime are Non-cacheable.
  - All accesses to the EL3 translation tables are Non-cacheable.
- When the value of SCTLR_EL3.I is 0:
  - All instruction accesses to Normal memory using the EL3 translation regime are Non-cacheable.

Note

The SCTLR_EL3.{I, C} fields have no effect on the EL1&0, EL2, and EL2&0 translation regimes.

In addition:

- For translation regimes other than the EL1&0 translation regime, if the value of SCTLR_ELx.M is 0, indicating that stage 1 translations are disabled for that translation regime, then:
  - If the value of SCTLR_ELx.I is 0, instruction accesses to Normal memory from stage 1 of the translation regime are Outer Shareable, Inner Non-cacheable, Outer Non-cacheable.
  - If the value of SCTLR_ELx.I is 1, instruction accesses to Normal memory from stage 1 of the translation regime are Outer Shareable, Inner Write-Through cacheable, Outer Write-Through cacheable.
- For the EL1&0 translation regime, if the value of SCTLR_EL1.M is 0, indicating that stage 1 translations are disabled for that translation regime, and the value of HCR_EL2.DC is 0:
  - If the value of SCTLR_EL1.I is 0, instruction accesses to Normal memory from stage 1 of the translation regime are Outer Shareable, Inner Non-cacheable, Outer Non-cacheable.
  - If the value of SCTLR_EL1.I is 1, instruction accesses to Normal memory from stage 1 of the translation regime are Outer Shareable, Inner Write-Through Cacheable, Outer Write-Through Cacheable.

The effect of SCTLR_ELx.C, HCR_EL2.DC and HCR_EL2.CD is reflected in the result of the address translation instructions in the PAR when these bits have an effect on the stages of translation being reported in the PAR.

Note

- In conjunction with the requirements in *Non-cacheable accesses and instruction caches* on page D4-2505, the requirements in this section mean the architecturally required effect of SCTLR_ELx.I is limited to its effect on caching instruction accesses in unified caches.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- This specification can give rise to different cacheability attributes between instruction and data accesses to the same location. Where this occurs, the measures for mismatch memory attributes described in *Mismatched memory attributes* on page B2-164 must be followed to manage the corresponding loss of coherency.

D4.4.5 Behavior of caches at reset

In Armv8:

- All caches reset to IMPLEMENTATION DEFINED states that might be UNKNOWN.
- The Cacheability control fields described in *Enabling and disabling the caching of memory accesses* on page D4-2503 reset to values that force all memory locations to be treated as Non-cacheable.

  Note

  This applies only to the controls that apply to the Translation regime that is used by the Exception level and Security state entered on reset.

- An implementation can require the use of a specific cache initialization routine to invalidate its storage array before caching is enabled. The exact form of any required initialization routine is IMPLEMENTATION DEFINED, and the routine must be documented clearly as part of the documentation of the device.
- If an implementation permits cache hits when the Cacheability control fields force all memory locations to be treated as Non-cacheable then the cache initialization routine must:
  - Provide a mechanism to ensure the correct initialization of the caches.
  - Be documented clearly as part of the documentation of the device.

  In particular, if an implementation permits cache hits when the Cacheability controls force all memory locations to be treated as Non-cacheable, and the cache contents are not invalidated at reset, the initialization routine must avoid any possibility of running from an uninitialized cache. It is acceptable for an initialization routine to require a fixed instruction sequence to be placed in a restricted range of memory.
- Arm recommends that whenever an invalidation routine is required, it is based on the Armv8 cache maintenance instructions.

See also *TLB behavior at reset* on page D5-2666.

D4.4.6 Non-cacheable accesses and instruction caches

In AArch64 state, instruction accesses to Non-cacheable Normal memory can be held in instruction caches.

Correspondingly, the sequence for ensuring that modifications to instructions are available for execution must include invalidation of the modified locations from the instruction cache, even if the instructions are held in Normal Non-cacheable memory. This includes cases where System register Cacheability control fields force instruction accesses to memory to be Non-cacheable.

Therefore when using self-modified code in Non-cacheable space in a uniprocessor system, the following sequence is required:

```
; Enter this code with <Wt> containing the new 32-bit instruction
; to be held at a location pointed to by <Xn> in Normal Non-cacheable memory.
STR <Wt>, [Xn]
DSB ISH; Ensure visibility of the data stored
IC IVAU, [Xn] ; Invalidate instruction cache by VA to PoU
DSB ISH; Ensure completion of the invalidations
ISB ;
```

In a multiprocessor system, the `IC IVAU` is broadcast to all PEs within the Inner Shareable domain of the PE running this sequence, but additional software steps might be required to synchronize the threads with other PEs. This might be necessary so that the PEs executing the modified instructions can execute an `ISB` after completing the invalidation, and to avoid issues associated with concurrent modification and execution of instruction sequences.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Larger blocks of instructions can be modified using the `IC IALLU` instruction for a uniprocessor system, or a `IC IALLUIS` for a multiprocessor system.

Note

This section applies even when the Cacheability control fields force instruction accesses to memory in AArch64 state to be Non-cacheable, as described in *Enabling and disabling the caching of memory accesses* on page D4-2503.

D4.4.7 About cache maintenance in AArch64 state

The following sections give general information about cache maintenance:

- *Terms used in describing the cache maintenance instructions*.
- *The Armv8 abstraction of the cache hierarchy* on page D4-2509.

The following sections describe the A64 cache maintenance instructions:

- *The instruction cache maintenance instruction (IC)* on page D4-2511.
- *The data cache maintenance instruction (DC)* on page D4-2511.

Note

Some descriptions of the cache maintenance instructions refer to the cacheability of the address on which the instruction operates. The Cacheability of an address is determined by the applicable translation table entry for that address, as modified by any applicable System register Cacheability controls, such as the SCTLR_EL1.{I, C} controls.

Terms used in describing the cache maintenance instructions

Cache maintenance instructions are defined to act on particular memory locations. Instruction scope can be defined:

- By the virtual address of the memory location to be maintained, referred to as operating *by VA*.
- By a mechanism that describes the location in the hardware of the cache, referred to as operating *by set/way*.

In addition, for instruction caches, there are instructions that invalidate all entries.

The following subsections define the terms used in the descriptions of the cache maintenance instructions:

- *Terminology for cache maintenance instructions operating by set/way*.
- *Terminology for Clean, Invalidate, and Clean and Invalidate instructions* on page D4-2507.

Note

There is no terminology specific to cache maintenance instructions that operate by VA. When all applicable stages of translation are disabled, the VA used is identical to the PA. For more information about memory system behavior when address translation is disabled, see *The effects of disabling a stage of address translation* on page D5-2586.

Terminology for cache maintenance instructions operating by set/way

Cache maintenance instruction that operate by set/way refer to the particular structures in a cache. Three parameters describe the location in a cache hierarchy that an instruction works on. These parameters are:

Level The cache level of the hierarchy. The number of levels of cache is IMPLEMENTATION DEFINED. The cache levels that can be managed using the architected cache maintenance instructions that operate by set/way can be determined from the CLIDR_EL1.

In the Arm architecture, the lower numbered cache levels are those closest to the PE. See *Memory hierarchy* on page B2-143.

Set Each level of a cache is split up into a number of *sets*. Each set is a set of locations in a cache level to which an address can be assigned. Usually, the set number is an IMPLEMENTATION DEFINED function of an address.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In the Arm architecture, sets are numbered from 0.

Way The associativity of a cache is the number of locations in a set to which a specific address can be assigned. The *way* number specifies one of these locations.

In the Arm architecture, ways are numbered from 0.

Note

Because the allocation of a memory address to a cache location is entirely IMPLEMENTATION DEFINED, Arm expects that most portable software will use only the cache maintenance instructions by set/way as single steps in a routine to perform maintenance on the entire cache.

Terminology for Clean, Invalidate, and Clean and Invalidate instructions

Caches introduce coherency problems in two possible directions:

1. An update to a memory location by a PE that accesses a cache might not be visible to other observers that can access memory. This can occur because new updates are still in the cache and are not visible yet to the other observers that do not access that cache.
2. Updates to memory locations by other observers that can access memory might not be visible to a PE that accesses a cache. This can occur when the cache contains an old, or *stale*, copy of the memory location that has been updated.

The *Clean* and *Invalidate* instructions address these two issues. The definitions of these instructions are:

Clean A cache clean instruction ensures that updates made by an observer that controls the cache are made visible to other observers that can access memory at the point to which the instruction is performed. Once the Clean has completed, the new memory values are guaranteed to be visible to the point to which the instruction is performed, for example to the Point of Unification.

The cleaning of a cache entry from a cache can overwrite memory that has been written by another observer only if the entry contains a location that has been written to by an observer in the shareability domain of that memory location.

Invalidate A cache invalidate instruction ensures that updates made visible by observers that access memory at the point to which the invalidate is defined, are made visible to an observer that controls the cache. This might result in the loss of updates to the locations affected by the invalidate instruction that have been written by observers that access the cache, if those updates have not been cleaned from the cache since they were made.

If the address of an entry on which the invalidate instruction operates is Normal, Non-cacheable or any type of Device memory then an invalidate instruction also ensures that this address is not present in the cache.

Note

Entries for addresses that are Normal Cacheable can be allocated to the cache at any time, and so the cache invalidate instruction cannot ensure that the address is not present in a cache.

Clean and Invalidate

A cache *clean and invalidate* instruction behaves as the execution of a clean instruction followed immediately by an invalidate instruction. Both instructions are performed to the same location.

The points to which a cache maintenance instruction can be defined differ depending on whether the instruction operates by VA or by set/way:

- For instructions operating by set/way, the point is defined to be to the next level of caching. For the All operations, the point is defined as the Point of Unification for each location held in the cache.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- For instructions operating by VA, the following conceptual points are defined:

Point of Coherency (PoC)

The point at which all agents that can access memory are guaranteed to see the same copy of a memory location for accesses of any memory type or cacheability attribute. In many cases this is effectively the main system memory, although the architecture does not prohibit the implementation of caches beyond the PoC that have no effect on the coherency between memory system agents.

Note

The presence of system caches can affect the determination of the point of coherency as described in *System level caches* on page D4-2522.

Point of Unification (PoU)

The PoU for a PE is the point by which the instruction and data caches and the translation table walks of that PE are guaranteed to see the same copy of a memory location. In many cases, the Point of Unification is the point in a uniprocessor memory system by which the instruction and data caches and the translation table walks have merged.

The PoU for an Inner Shareable shareability domain is the point by which the instruction and data caches and the translation table walks of all the PEs in that Inner Shareable shareability domain are guaranteed to see the same copy of a memory location. Defining this point permits self-modifying software to ensure future instruction fetches are associated with the modified version of the software by using the standard correctness policy of:

1. Clean data cache entry by address.
2. Invalidate instruction cache entry by address.

Point of Persistence (PoP)

When ARMv8.2-DCPoP is implemented:

The point in a memory system, if it exists, at or beyond the Point of Coherency, where a write to memory is maintained when system power is removed, and reliably recovered when power is restored to the affected locations in memory.

When ARMv8.2-DCPoP and ARMv8.2-DCCVADP are implemented:

The point in a memory system where there is a system guarantee that there is sufficient energy within the system to ensure that a write to memory will be persistent if system power is removed.

Note

Such memory is sometimes called non-volatile memory. For example, the Storage-class memory shown in Figure B2-1 on page B2-144 could be used as target memory for this feature.

Point of Deep Persistence (PoDP)

The point in a memory system where any writes that have reached that point are persistent, even in the event of an instantaneous hardware failure of the power system.

The following fields in the CLIDR_EL1 relate to the PoC and PoU:

LoC, Level of Coherence

This field defines the last level of cache that must be cleaned or invalidated when cleaning or invalidating to the Point of Coherency. The LoC value is a cache level, so, for example, if LoC contains the value 3:

- A clean to the Point of Coherency operation requires the level 1, level 2 and level 3 caches to be cleaned.
- Level 4 cache is the first level that does not have to be maintained.

If the LoC field value is `0x0`, this means that no levels of cache need to cleaned or invalidated when cleaning or invalidating to the Point of Coherency.

If the LoC field value is a nonzero value that corresponds to a level that is not implemented, this indicates that all implemented caches are before the Point of Coherency.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

LoUU, Level of Unification, uniprocessor

This field defines the last level of data cache that must be cleaned or invalidated when cleaning or invalidating to the Point of Unification for the PE. As with LoC, the LoUU value is a cache level.

If the LoUU field value is `0x0`, this means that no levels of data cache need to be cleaned or invalidated when cleaning or invalidating to the Point of Unification.

If the LoUU field value is a nonzero value that corresponds to a level that is not implemented, this indicates that all implemented caches are before the Point of Unification.

LoUIS, Level of Unification, Inner Shareable

In any implementation:

- This field defines the last level of data or unified cache that must be cleaned or invalidated when cleaning or invalidating to the Point of Unification for the Inner Shareable shareability domain. As with LoC, the LoUIS value is a cache level.
- If the LoUIS field value is `0x0`, this means that no levels of data or unified cache need to cleaned or invalidated when cleaning or invalidating to the Point of Unification for the Inner Shareable shareability domain.
- If the LoUIS field value is a nonzero value that corresponds to a level that is not implemented, this indicates that all implemented caches are before the Point of Unification.

The Armv8 abstraction of the cache hierarchy

The following subsections describe the Armv8 abstraction of the cache hierarchy:

- *Cache maintenance instructions that operate by VA*.
- *Cache maintenance instructions that operate by set/way*.

Cache maintenance instructions that operate by VA

The VA-based cache maintenance instructions are described as operating by VA. Each of these instructions is always qualified as being one of:

- Performed to the Point of Coherency.
- Performed to the Point of Unification.
- When ARMv8.2-DCPoP is implemented, performed to the Point of Persistence.

See *Terms used in describing the cache maintenance instructions* on page D4-2506 for definitions of these terms, and for more information about possible meanings of VA.

A64 Cache maintenance instructions on page D4-2510 lists the VA-based maintenance instructions.

The CTR_EL0 holds minimum line length values for:

- The instruction caches.
- The data and unified caches.

These values support efficient invalidation of a range of VAs, because this value is the most efficient address stride to use to apply a sequence of VA-based maintenance instructions to a range of VAs.

For the Invalidate data or unified cache line by VA instruction, the Cache Write-back Granule field of the CTR_EL0 defines the maximum granule that a single invalidate instruction can invalidate. This meaning of the Cache Write-back Granule is in addition to its defining the maximum size that can be written back.

Cache maintenance instructions that operate by set/way

A64 Cache maintenance instructions on page D4-2510 lists the set/way-based maintenance instructions. Some encodings of these instructions include a required field that specifies the cache level for the instruction:

- A clean instruction cleans from the level of cache specified through to at least the next level of cache, moving further from the PE.
- An invalidate instruction invalidates only at the level specified.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D4.4.8 A64 Cache maintenance instructions

The A64 cache maintenance instructions are part of the A64 System instruction class in the register encoding space. For encoding details and other general information on these System instructions, see *System instructions* on page C3-204, *SYS* on page C6-1362 and *Cache maintenance instructions, and data cache zero operation* on page C5-381.

Table D4-3 shows the AArch64 System instructions that perform instruction or data cache maintenance. Instructions that take an argument include `Xt` in the entry in the *System instruction* column.

Note

- In Table D4-3 the Point of Unification is the Point of Unification of the PE executing the cache maintenance instruction.
- In general, the AArch32 instruction and data cache maintenance instructions provide equivalent functionality to the AArch64 cache maintenance instructions, see *AArch32 cache and branch predictor maintenance instructions* on page G4-5899. However, the data cache clean to the Point of Persistence instruction, implemented when ARMv8.2-DCPoP is implemented, is supported in AArch64 state only.

Table D4-3 System instructions for cache maintenance

System instruction	Instruction	Notes
Instruction cache maintenance instructions		
`IC IALLUIS`	Invalidate all to Point of Unification, Inner Shareable	EL1 or higher access.
`IC IALLU`	Invalidate all to Point of Unification	EL1 or higher access.
`IC IVAU, Xt`	Invalidate by virtual address to Point of Unification	When SCTLR_EL1.UCI[a] == 1, EL0 access. Otherwise, EL1 or higher access.
Data cache maintenance instructions		
`DC IVAC, Xt`	Invalidate by virtual address to Point of Coherency	EL1 or higher access.
`DC ISW, Xt`	Invalidate by set/way	EL1 or higher access.
`DC CVAC, Xt`	Clean by virtual address to Point of Coherency	When SCTLR_EL1.UCI[a] == 1, EL0 access. Otherwise EL1 or higher access.
`DC CVAP, Xt`	Clean by virtual address to Point of Persistence[b]	When SCTLR_EL1.UCI[a] == 1, EL0 access. Otherwise EL1 or higher access.
`DC CSW, Xt`	Clean by set/way	EL1 or higher access.
`DC CVAU, Xt`	Clean by virtual address to Point of Unification	When SCTLR_EL1.UCI[a] == 1, EL0 access. Otherwise EL1 or higher access.
`DC CIVAC, Xt`	Clean and invalidate by virtual address to Point of Coherency	When SCTLR_EL1.UCI[a] == 1, EL0 access. Otherwise EL1 or higher access.
`DC CISW, Xt`	Clean and invalidate by set/way	EL1 or higher access.

a. When HCR_EL2.{E2H,TGE} == {1, 1}, the control is from SCTLR_EL2.
b. Supported only when ARMv8.2-DCPoP is implemented.

A `DSB` or `DMB` instruction intended to ensure the completion of cache or branch predictor maintenance instructions must have an access type of both loads and stores.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following subsections give more information about these instructions:

- *The instruction cache maintenance instruction (IC)*.
- *The data cache maintenance instruction (DC)*.
- *EL0 accessibility of cache maintenance instructions* on page D4-2513.
- *General requirements for the scope of maintenance instructions* on page D4-2513.
- *Effects of instructions that operate by VA to the PoC* on page D4-2513.
- *Effects of instructions that operate by VA to the PoP* on page D4-2514.
- *Effects of instructions that operate by VA to the PoU* on page D4-2514.
- *Effects of All and set/way maintenance instructions* on page D4-2515.
- *Effects of virtualization and Security state on the cache maintenance instructions* on page D4-2515.
- *Boundary conditions for cache maintenance instructions* on page D4-2517.
- *Ordering and completion of data and instruction cache instructions* on page D4-2517.
- *Performing cache maintenance instructions* on page D4-2518.

The instruction cache maintenance instruction (`IC`)

System instructions on page C3-204 describes the A64 assembly syntax for this instruction.

When an `IC` instruction requires an address argument this takes the form of a 64-bit register that holds the VA argument. No alignment restrictions apply for this address.

Any cache maintenance instruction operating by VA includes as part of any required VA to PA translation:

- For an instruction executed at EL1, or at EL2 when HCR_EL2.E2H==1, the current ASID.
- The current Security state.
- Whether the instruction was executed at EL1 or EL2.
- For an instruction executed at EL1, the current VMID.

That VA to PA translation might fault. However, for an instruction cache maintenance instruction that operates by VA:

- It is IMPLEMENTATION DEFINED whether the instruction can generate:
  - An Access flag fault.
  - A Translation fault.
- The instruction cannot generate a Permission fault, except for:
  - The possible generation of a Permission fault by the execution of an `IC IVAU` instruction at EL0 when the specified address does not have read access at EL0, as described in *EL0 accessibility of cache maintenance instructions* on page D4-2513.
  - The possible Permission fault on a Stage 2 fault on a stage 1 translation table walk.

For more information about possible faults on a cache maintenance instruction that operates by VA see *VMSAv8-64 memory aborts* on page D5-2652.

See also *Ordering and completion of data and instruction cache instructions* on page D4-2517.

The data cache maintenance instruction (`DC`)

System instructions on page C3-204 describes the A64 assembly syntax for this instruction.

When a `DC` instruction requires a set/way/level argument this takes the form of a 64-bit register, the upper 32 bits of which are RES0.

If a data cache maintenance by set/way instruction specifies a set, way, or level argument that is larger than the value supported by the implementation then the instruction is CONSTRAINED UNPREDICTABLE, see *Out of range values of the Set/Way/Index fields in cache maintenance instructions* on page K1-7871 or the instruction description.

When a `DC` instruction requires an address argument this takes the form of a 64-bit register that holds the VA argument. No alignment restrictions apply for this address.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Any cache maintenance instruction operating by VA includes as part of any required VA to PA translation:

- For an instruction executed at EL1, or at EL2 when HCR_EL2.E2H is 1, the current ASID.
- The current Security state.
- Whether the instruction is executed at EL1 or EL2.
- For an instruction executed at EL1, the current VMID.

That VA to PA translation might fault. However, a data or unified cache maintenance instruction that operates by VA cannot generate a Permission fault except in the following cases:

- The possible generation of a Permission fault by:
  - The execution of a `DC IVAC` instruction when the specified address does not have write permission.
  - The execution of an enabled `DC *` instruction at EL0 when the specified address does not have read access at EL0, as described in *EL0 accessibility of cache maintenance instructions* on page D4-2513.

  The description of Permission faults includes possible constraints on the generation of Permission faults on cache maintenance by VA instructions.
- The possible Permission fault on a Stage 2 fault on a stage 1 translation table walk.

For more information about possible faults on a VA to PA translation see *VMSAv8-64 memory aborts* on page D5-2652.

When executed at EL1, a `DC ISW` instruction performs a clean and invalidate, meaning it performs the same maintenance as a `DC CISW` instruction, if all of the following apply:

- EL2 is implemented and enabled in the current Security state.
- Either:
  - The value of HCR_EL2.SWIO is 1, forcing a cache clean to perform a clean and invalidate.
  - The value of HCR_EL2.VM is 1, meaning EL1&0 stage two address translation is enabled.

When executed at EL1, a `DC IVAC` instruction performs a clean and invalidate, meaning it performs the same maintenance as a `DC CIVAC` instruction, if all of the following apply:

- EL2 is implemented and enabled in the current Security state.
- The value of HCR_EL2.VM is 1, meaning EL1&0 stage two address translation is enabled.

Note

The forcing of a clean instruction to perform a clean invalidate applies to the AArch32 cache maintenance instructions `DCIMVAC` and `DCISW`. See *AArch32 data cache maintenance instructions (DC*)* on page G4-5901.

When ARMv8.2-DCPoP is implemented, meaning the `DC CVAP` instruction is implemented, if the memory system does not support the Point of Persistence, a data cache clean to the PoP, `DC CVAP`, behaves as a data cache clean to the PoC, `DC CVAC`.

Note

- Support for the Point of Persistence does not change the definition or behavior of the CLIDR_EL1 System register.
- Because a `DSB SYS` instruction will not complete until all previous `DC CVAP` instructions have completed, the following sequence can be used to ensure the completion of any store to the Point of Persistence, where the store might be to Non-cacheable memory:

```
DMB            ; Note this can be any DMB that applies to both loads and stores
DC CVAP, Xt
DSB SYS
```

- If caches that are invisible to the programmer exist beyond the Point of Coherency but before the Point of Persistence and hold data that is marked as Non-cacheable, the `DC CVAP` operation causes the Non-cacheable locations to be cleaned from those caches.

If a memory fault that sets the FAR for the translation regime applicable for the cache maintenance instruction is generated from a data cache maintenance instruction, the FAR holds the address specified in the register argument of the instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

Despite its mnemonic, `DC ZVA` is not a cache maintenance instruction.

See also *EL0 accessibility of cache maintenance instructions* and *Ordering and completion of data and instruction cache instructions* on page D4-2517.

EL0 accessibility of cache maintenance instructions

The SCTLR_EL1.UCI bit enables EL0 access for the `DC CVAU`, `DC CVAC`, `DC CVAP`, `DC CIVAC`, and `IC IVAU` instructions. When EL0 use of these instructions is disabled because SCTLR_EL1.UCI == 0, executing one of these instructions at EL0 generates a trap to EL1, that is reported using EC = `0x18`. When HCR_EL2.{E2H,TGE} == 1, the control is from SCTLR_EL2.

Note

`DC CVAP` is implemented only if ARMv8.2-DCPoP is implemented.

For these instructions read access permission is required. When the value of SCTLR_EL1.UCI is 1:

- For the `DC CVAU`, `DC CVAC`, `DC CVAP`, and `DC CIVAC` instructions, if the instruction is executed at EL0 and the address specified in the argument cannot be read at EL0, a Permission fault might be generated.
- For the `IC IVAU` instruction, if the instruction is executed at EL0 and the address specified in the argument cannot be read at EL0, it is IMPLEMENTATION DEFINED whether a Permission fault is generated.

For more information see the description of Permission faults, In the case of a `DC *` instruction executed at EL0 when the address specified cannot be read at EL0 the Permission fault is generated unless one of the permitted constraints described in that section applies and means the fault cannot be generated.

Software can read the CTR_EL0 to discover the stride needed for cache maintenance instructions. The SCTLR_EL1.UCT bit enables EL0 access to the CTR_EL0. When EL0 access to the Cache Type register is disabled, a register access instruction executed at EL0 is trapped to EL1 using EC = `0x18`.

General requirements for the scope of maintenance instructions

The Armv8 specification of the cache maintenance instructions describes what each instruction is guaranteed to do in a system. It does not limit other behaviors that might occur, provided they are consistent with the requirements described in *General behavior of the caches* on page D4-2499, *Behavior of caches at reset* on page D4-2505, and *Preloading caches* on page B2-147.

This means that as a side-effect of a cache maintenance instruction:

- Any location in the cache might be cleaned.
- Any unlocked location in the cache might be cleaned and invalidated.

Note

Arm recommends that, for best performance, such side-effects are kept to a minimum. Arm strongly recommends that the side-effects of operations performed in Non-secure state do not have a significant performance impact on execution in Secure state.

Effects of instructions that operate by VA to the PoC

For Normal memory that is not Inner Non-cacheable, Outer Non-cacheable, cache maintenance instructions that operate by VA to the PoC must affect the caches of other PEs in the shareability domain described by the shareability attributes of the VA supplied with the instruction.

For Device memory and Normal memory that is Inner Non-cacheable, Outer Non-cacheable, these instructions must affect the caches of all PEs in the Outer Shareable shareability domain of the PE on which the instruction is operating.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In all cases, for any affected PE, these instructions affect all data and unified caches to the PoC. Table D4-4 shows the scope of these Data and unified cache maintenance instructions.

Table D4-4 PEs affected by cache maintenance instructions to the PoC

Shareability	PEs affected	Effective to
Non-shareable	The PE executing the instruction	The PoC of the entire system
Inner Shareable	All PEs in the same Inner Shareable shareability domain as the PE executing the instruction	The PoC of the entire system
Outer Shareable	All PEs in the same Outer Shareable shareability domain as the PE executing the instruction	The PoC of the entire system

Effects of instructions that operate by VA to the PoP

For Normal memory that is not Inner Non-cacheable, Outer Non-cacheable, cache maintenance instructions that operate by VA to the PoP must affect the caches of other PEs in the shareability domain described by the shareability attributes of the VA supplied with the instruction.

For Device memory and Normal memory that is Inner Non-cacheable, Outer Non-cacheable, these instructions must affect the caches of all PEs in the Outer Shareable shareability domain of the PE on which the instruction is operating.

In all cases, for any affected PE, these instructions affect all data and unified caches to the PoP. Table D4-5 shows the scope of these Data and unified cache maintenance to the PoP instructions.

Table D4-5 PEs affected by cache maintenance instructions to the PoP

Shareability	PEs affected	Effective to
Non-shareable	The PE executing the instruction	The PoPof the entire system
Inner Shareable	All PEs in the same Inner Shareable shareability domain as the PE executing the instruction	The PoP of the entire system
Outer Shareable	All PEs in the same Outer Shareable shareability domain as the PE executing the instruction	The PoP of the entire system

Effects of instructions that operate by VA to the PoU

For cache maintenance instructions that operate by VA to the PoU, Table D4-6 shows how, for a VA in a Normal or Device memory location, the shareability attribute of the VA determines the minimum set of PEs affected, and the point to which the instruction must be effective.

Table D4-6 PEs affected by cache maintenance instructions to the PoU

Shareability	PEs affected	Effective to
Non-shareable	The PE executing the instruction	The PoU of instruction cache fills, data cache fills and write-backs, and translation table walks, on the PE executing the instruction
Inner Shareable or Outer Shareable	All PEs in the same Inner Shareable shareability domain as the PE executing the instruction	The PoU of instruction cache fills, data cache fills and write-backs, and translation table walks, of all PEs in the same Inner Shareable shareability domain as the PE executing the instruction

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

The set of PEs guaranteed to be affected is never greater than the PEs in the Inner Shareable shareability domain containing the PE executing the instruction.

Effects of All and set/way maintenance instructions

The `IC IALLU` and `DC` set/way instructions apply only to the caches of the PE that performs the instruction.

The `IC IALLUIS` instruction can affect the caches of all PEs in the same Inner Shareable shareability domain as the PE that performs the instruction. This instruction has an effect to the Point of Unification of instruction cache fills, data cache fills, and write-backs, and translation table walks, of all PEs in the same Inner Shareable shareability domain.

Note

The possible presence of system caches, as described in *System level caches* on page D4-2522, means architecture does not guarantee that all levels of the cache can be maintained using set/way instructions.

Effects of virtualization and Security state on the cache maintenance instructions

Each Security state has its own physical address (PA) space, therefore cache entries are associated with PA space.

Table D4-7 shows the effects of virtualization and security on the cache maintenance instructions. In the table, the *Specified entries* are entries that the architecture requires the instruction to affect. The rules described in *General behavior of the caches* on page D4-2499 mean that an instruction might also affect other entries.

Table D4-7 Effects of virtualization and security on the maintenance instructions

Cache maintenance instructions	Security state	Specified entries
Data or unified cache maintenance instructions		
Invalidate, Clean, or Clean and Invalidate by VA: `DC IVAC`, `DC CVAC`, `DC CVAP`, `DC CVAU`, `DC CIVAC`, `DC CVAP`	Both	All lines that hold the PA that, in the current Security state, is mapped to by the combination of all of: • The specified VA. • For an instruction executed at EL1, EL0, or at EL2 when HCR_EL2.E2H is set to 1 the current ASID if the location is mapped to by a non-global page. • For an instruction executed at EL1 when SCR_EL3.NS == 1 or SCR_EL3.EEL2 == 1, the current VMID.[a] • For an instruction executed at EL0 when (SCR_EL3.NS == 1 or SCR_EL3.EEL2 == 1) and (HCR_EL2.E2H == 0 or HCR_EL2.TGE == 0), the current VMID.[a]
Invalidate, Clean, or Clean and Invalidate by set/way: `DC ISW`, `DC CSW`, `DC CISW`	Non-secure	Line specified by set/way provided that the entry comes from the Non-secure PA space.
	Secure	Line specified by set/way regardless of the PA space that the entry has come from.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D4-7 Effects of virtualization and security on the maintenance instructions (continued)

Cache maintenance instructions	Security state	Specified entries
Instruction cache maintenance instructions		
Invalidate by VA: `IC IVAU`	Both	All lines corresponding to the specified VA[b] in the current translation regime and: • For an instruction executed at EL1, EL0, or at EL2 when HCR_EL2.E2H is set to 1 the current ASID. • For an instruction executed at EL1 when SCR_EL3.NS == 1 or SCR_EL3.EEL2 == 1, the current VMID.[a] • For an instruction executed at EL0 when (SCR_EL3.NS == 1 or SCR_EL3.EEL2 == 1) and (HCR_EL2.E2H == 0 or HCR_EL2.TGE == 0), the current VMID.[a]
Invalidate All: `IC IALLU`, `IC IALLUIS`	Both	For an instruction executed at: • EL1 when SCR_EL3.NS ==0 and SCR_EL3.EEL2 == 1, all instruction cache lines containing entries associated with the current VMID. • EL1 when SCR_EL3.NS ==1, all instruction cache lines containing Non-secure entries associated with the current VMID. • EL2 when SCR_EL3.NS ==1, all instruction cache lines containing Non-secure entries. • EL1 when the Effective value of SCR_EL3.{EEL2, NS} is {0,0}, EL2 when SCR_EL3.EEL2 is 1, or EL3, all instruction cache lines.

a. Dependencies on the VMID apply even when HCR_EL2.VM is set to 0. VTTBR_EL2.VMID resets to zero, meaning there is a valid VMID.

b. The type of instruction cache used affects the interpretation of the specified entries in this table such that:
 For a PIPT instruction cache, the cache maintenance applies to all entries whose physical address corresponds to the specified address.
 For a VIPT instruction cache, the cache maintenance applies to entries whose virtual index and physical tag corresponds to the specified address.
For information on types of instruction cache see *Instruction caches* on page D5-2685.

For locked entries and entries that might be locked, the behavior of cache maintenance instructions described in *The interaction of cache lockdown with cache maintenance instructions* on page D4-2521 applies.

With an implementation that generates aborts if entries are locked or might be locked in the cache, when the use of lockdown aborts is enabled, these aborts can occur on any cache maintenance instructions.

In an implementation that includes EL2:

- The architecture does not require cache cleaning when switching between virtual machines. Cache invalidation by set/way must not present an opportunity for one virtual machine to corrupt state associated with a second virtual machine. To ensure this requirement is met, invalidate by set/way instructions can, instead, perform a clean and invalidate by set/way.
- As described in *The data cache maintenance instruction (DC)* on page D4-2511, the AArch64 Data cache invalidate instructions, `DC IVAC` and `DC ISW`, when executed at EL1 and EL0, and the AArch32 Data cache invalidate instructions `DCIMVAC` and `DCISW`, when executed at EL1, can be configured to perform a cache clean as well as a cache invalidation.
- TLB and instruction cache invalidate instructions executed at EL1 are broadcast across the Inner Shareable domain when all of the following is true:
  — When the value of HCR_EL2.FB is 1.
  — EL3 is not implemented, or EL3 is implemented and either SCR_EL3.NS == 1 or SCR_EL3.EEL2 == 1.

  When EL1 is using AArch64, this applies to the `IC IALLU` instruction. This means the instruction performs the invalidation that would be performed by the corresponding Inner Shareable instruction `IC IALLUIS`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For more information about the cache maintenance instructions, see *About cache maintenance in AArch64 state* on page D4-2506, *A64 Cache maintenance instructions* on page D4-2510, and Chapter D5 *The AArch64 Virtual Memory System Architecture*.

Boundary conditions for cache maintenance instructions

Cache maintenance instructions operate on the caches regardless of whether the System register Cacheability controls force all memory accesses to be Non-cacheable.

For VA-based cache maintenance instructions, the instruction operates on the caches regardless of the memory type and cacheability attributes marked for the memory address in the VMSA translation table entries. This means that the effects of the cache maintenance instructions can apply regardless of:

- Whether the address accessed:
  - Is Normal memory or Device memory.
  - Has the Cacheable attribute or the Non-cacheable attribute.
- Any applicable domain control of the address accessed.
- The access permissions for the address accessed, other than the effect of the stage two write permission on data or unified cache invalidation instructions.

Ordering and completion of data and instruction cache instructions

All data cache instructions, other than `DC ZVA`, that specify an address:

- Execute in program order relative to loads or stores that have all of the following properties:
  - Access an address in Normal memory with either Inner Write Through or Inner Write Back attributes within the same cache line of minimum size, as indicated by CTR_EL0.DMinLine.
  - Use an address with the same cacheability attributes as the address passed to the data cache instruction.
- Can execute in any order relative to loads or stores that have all of the following properties:
  - Access an address in Normal memory with either Inner Write Through or Inner Write Back attributes within the same cache line of minimum size, as indicated by CTR_EL0.DMinLine.
  - Use an address with different cacheability attributes as the address passed to the data cache instruction.
  - Do not have a DMB or DSB executed between the load or store instruction and the data cache instruction.
- Can execute in any order relative to loads or stores that access any address with the Device memory attribute, or with Normal memory with Inner Non-cacheable attribute unless a `DMB` or `DSB` is executed between the instructions.
- Execute in program order relative to other data cache instructions, other than `DC ZVA`, that specify an address within the same cache line of minimum size, as indicated by CTR_EL0.DMinLine.
- Can execute in any order relative to loads or stores that access an address in a different cache line of minimum size, as indicated by CTR_EL0.DMinLine, unless a `DMB` or `DSB` is executed between the instructions.
- Can execute in any order relative to other data cache instructions, other than `DC ZVA`, that specify an address in a different cache line of minimum size, as indicated by CTR_EL0.DMinLine, unless a `DMB` or `DSB` is executed between the instructions.
- Can execute in any order relative to data cache maintenance instructions that do not specify an address unless a `DMB` or `DSB` is executed between the instructions.

Note

Despite its mnemonic, the `DC ZVA, Data Cache Zero by VA` instruction is not a data cache maintenance instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Can execute in any order relative to instruction cache maintenance instructions unless a DSB is executed between the instructions.

Note

- Data cache ordering rules by address are consistent with physically indexed physically tagged caches. See *Data and unified caches* on page D5-2685.
- *Data cache zero instruction* on page D4-2520 describes the ordering and completion rules for Data Cache Zero.

All data cache maintenance instructions that do not specify an address:

- Can execute in any order relative to data cache maintenance instructions that do not specify an address unless a DMB or DSB is executed between the instructions.
- Can execute in any order relative to data cache maintenance instructions that specify an address, other than Data Cache Zero, unless a DMB or DSB is executed between the instructions.
- Can execute in any order relative to loads or stores unless a DMB or DSB is executed between the instructions.
- Can execute in any order relative to instruction cache maintenance instructions unless a DSB is executed between the instructions.

All instruction cache maintenance instructions can execute in any order relative to other instruction cache instructions, data cache instructions, loads, and stores unless a DSB is executed between the instructions.

A cache maintenance instruction can complete at any time after it is executed, but is only guaranteed to be complete, and its effects visible to other observers, following a DSB instruction executed by the PE that executed the cache maintenance instruction. See also the requirements for cache maintenance instructions in *Completion and endpoint ordering* on page B2-130.

In all cases, where the text in this section refers to a DMB or a DSB, this means a DMB or DSB whose required access type is both loads and stores.

Note

These ordering requirements are extended from the requirements in AArch32 state given in:

- *Ordering of cache and branch predictor maintenance instructions* on page G4-5907.
- *AArch32 instruction cache maintenance instructions (IC*)* on page G4-5900.

Performing cache maintenance instructions

To ensure all cache lines in a block of address space are maintained through all levels of cache Arm strongly recommends that software:

- For data or unified cache maintenance, uses the CTR_EL0.DMinLine value to determine the loop increment size for a loop of data cache maintenance by VA instructions.
- For instruction cache maintenance, uses the CTR_EL0.IMinLine value to determine the loop increment size for a loop of instruction cache maintenance by VA instructions.

Example code for cache maintenance instructions

The cache maintenance instructions by set/way can clean or invalidate, or both, the entirety of one or more levels of cache attached to a PE. However, unless all PEs attached to the caches regard all memory locations as Non-cacheable, it is not possible to prevent locations being allocated into the cache during such a sequence of the cache maintenance instructions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

Since the set/way instructions are performed only locally, there is no guarantee of the atomicity of cache maintenance between different PEs, even if those different PEs are each executing the same cache maintenance instructions at the same time. Since any cacheable line can be allocated into the cache at any time, it is possible for a cache line to migrate from an entry in the cache of one PE to the cache of a different PE in a way that means the line is not affected by set/way based cache maintenance. Therefore, Arm strongly discourages the use of set/way instructions to manage coherency in coherent systems. The expected use of the cache maintenance instructions that operate by set/way is limited to the cache maintenance associated with the powerdown and powerup of caches, if this is required by the implementation.

The limitations of cache maintenance by set/way mean maintenance by set/way does not happen on multiple PEs, and cannot be made to happen atomically for each address on each PE. Therefore in multiprocessor or multithreaded systems, the use of cache maintenance by set/way to clean, or clean and invalidate, the entire cache for coherency management with very large buffers or with buffers with unknown address can fail to provide the expected coherency results because of speculation by other PEs, or possibly by other threads. The only way that these instructions can be used in this way is to first ensure that all PEs that might cause speculative accesses to caches that need to be maintained are not capable of generating speculative accesses. This can be achieved by ensuring that those PEs have no memory locations with a Normal Cacheable attribute. Such an approach can have very large system performance effects, and Arm advises implementers to use hardware coherency mechanisms in systems where this will be an issue.

System level caches on page D4-2522 refers to other limitations of cache maintenance by set/way.

The following example code for cleaning a data or unified cache to the Point of Coherency illustrates a generic mechanism for cleaning the entire data or unified cache to the Point of Coherency. It assumes that the current Cache Size Identification Register is in 32-bit format. For more information, see *Possible formats of the Cache Size Identification Register, CCSIDR_EL1* on page D4-2501.

```
        MRS     X0, CLIDR_EL1
        AND     W3, W0, #0x07000000     // Get 2 x Level of Coherence
        LSR     W3, W3, #23
        CBZ     W3, Finished
        MOV     W10, #0                 // W10 = 2 x cache level
        MOV     W8, #1                  // W8 = constant 0b1
Loop1:  ADD     W2, W10, W10, LSR #1    // Calculate 3 x cache level
        LSR     W1, W0, W2              // extract 3-bit cache type for this level
        AND     W1, W1, #0x7
        CMP     W1, #2
        B.LT    Skip                    // No data or unified cache at this level
        MSR     CSSELR_EL1, X10         // Select this cache level
        ISB                             // Synchronize change of CSSELR
        MRS     X1, CCSIDR_EL1          // Read CCSIDR
        AND     W2, W1, #7              // W2 = log2(linelen)-4
        ADD     W2, W2, #4              // W2 = log2(linelen)
        UBFX    W4, W1, #3, #10         // W4 = max way number, right aligned
        CLZ     W5, W4                  // W5 = 32-log2(ways), bit position of way in DC operand
        LSL     W9, W4, W5              // W9 = max way number, aligned to position in DC operand
        LSL     W16, W8, W5             // W16 = amount to decrement way number per iteration
Loop2:  UBFX    W7, W1, #13, #15        // W7 = max set number, right aligned
        LSL     W7, W7, W2              // W7 = max set number, aligned to position in DC operand
        LSL     W17, W8, W2             // W17 = amount to decrement set number per iteration
Loop3:  ORR     W11, W10, W9            // W11 = combine way number and cache number ...
        ORR     W11, W11, W7            //  ... and set number for DC operand
        DC      CSW, X11                // Do data cache clean by set and way
        SUBS    W7, W7, W17             // Decrement set number
        B.GE    Loop3
        SUBS    X9, X9, X16             // Decrement way number
        B.GE    Loop2
Skip:   ADD     W10, W10, #2            // Increment 2 x cache level
        CMP     W3, W10
        DSB                             // Ensure completion of previous cache maintenance instruction
        B.GT    Loop1
Finished:
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Similar approaches can be used for all cache maintenance instructions.

D4.4.9 Data cache zero instruction

The Data Cache Zero by Address instruction, `DC ZVA`, writes `0x00` to each byte of a block of *N* bytes, aligned in memory to *N* bytes in size, where:

- The block in memory is identified by the address supplied as an argument to the `DC ZVA` instruction. There are no alignment restrictions on this address.

  Note

  This means that each byte of the block of memory that includes the supplied address is set to zero.

- The DCZID_EL0 register indicates the block size, *N* bytes, that is written with byte values of zero.

Software can restrict access to this instruction. See *Configurable instruction enables and disables, and trap controls* on page D1-2373 and the description of the `DC ZVA` instruction.

The `DC ZVA` instruction behaves as a set of stores to the location being accessed, and:

- Generates a Permission fault if the translation regime being used when the instruction is executed does not permit writes to the locations.
- Requires the same considerations for ordering and the management of coherency as any other store instruction.

In addition:

- When the instruction is executed, it can generate memory faults or watchpoints that are prioritized in the same way as other memory related faults or watchpoints. Where a synchronous Data Abort fault or a watchpoint is generated, the CM bit in the syndrome field is not set to 1, which would be the case for all other cache maintenance instructions. See *ISS encoding for an exception from a Data Abort* on page D13-2992 for more information about the encoding of the associated ESR_ELx.ISS field.

- If the memory region being zeroed is any type of Device memory, then `DC ZVA` generates an Alignment fault which is prioritized in the same way as other alignment faults that are determined by the memory type.

Note

The architecture makes no statements about whether or not a `DC ZVA` instruction causes allocation to any particular level of the cache, for addresses that have a cacheable attribute for those levels of cache.

Despite its mnemonic, the `DC ZVA` instruction is not a data cache maintenance instruction.

D4.4.10 Cache lockdown

The concept of an entry locked in a cache is allowed, but not architecturally defined. How lockdown is achieved is IMPLEMENTATION DEFINED and might not be supported by:

- An implementation.
- Some memory attributes.

An unlocked entry in a cache might not remain in that cache. The architecture does not guarantee that an unlocked cache entry remains in the cache or remains incoherent with the rest of memory. Software must not assume that an unlocked item that remains in the cache remains dirty.

A locked entry in a cache is guaranteed to remain in that cache. The architecture does not guarantee that a locked cache entry remains incoherent with the rest of memory, that is, it might not remain dirty.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The interaction of cache lockdown with cache maintenance instructions

The interaction of cache lockdown and cache maintenance instructions is IMPLEMENTATION DEFINED. However, an architecturally-defined cache maintenance instruction on a locked cache line must comply with the following general rules:

- The effect of the following instructions on locked cache entries is IMPLEMENTATION DEFINED:
  - Cache clean by set/way, `DC CSW`.
  - Cache invalidate by set/way, `DC ISW`.
  - Cache clean and invalidate by set/way, `DC CISW`.
  - Instruction cache invalidate all, `IC IALLU` and `IC IALLUIS`.

  However, one of the following approaches must be adopted in all these cases:

  1. If the instruction specified an invalidation, a locked entry is not invalidated from the cache.
  2. If the instruction specified a clean it is IMPLEMENTATION DEFINED whether locked entries are cleaned.
  3. If an entry is locked down, or could be locked down, an IMPLEMENTATION DEFINED Data Abort exception is generated, using the DFSC value defined for this purpose, see *ISS encoding for an exception from a Data Abort* on page D13-2992.

  This permits a usage model for cache invalidate routines to operate on a large range of addresses by performing the required operation on the entire cache, without having to consider whether any cache entries are locked.

The effect of the following instructions is IMPLEMENTATION DEFINED:

- Cache clean by virtual address, `DC CVAC`, `DC CVAP`, and `DC CVAU`.
- Cache invalidate by virtual address, `DC IVAC`.
- Cache clean and invalidate by virtual address, `DC CIVAC`.

However, one of the following approaches must be adopted in all these cases:

1. If the instruction specified an invalidation, a locked entry is invalidated from the cache. For the clean and invalidate instructions, the entry must be cleaned before it is invalidated.
2. If the instruction specified an invalidation, a locked entry is not invalidated from the cache. If the instruction specified a clean it is IMPLEMENTATION DEFINED whether locked entries are cleaned.
3. If an entry is locked down, or could be locked down, an IMPLEMENTATION DEFINED Data Abort exception is generated, using the DFSC value defined for this purpose. See *ESR_ELx* on page K14-8042.

In an implementation that includes EL2 enabled in the current Security state, if HCR_EL2.TIDCP is set to 1, any exception relating to lockdown of an entry is routed to EL2.

Note

An implementation that uses an abort mechanism for entries that can be locked down but are not actually locked down must:

- Document the IMPLEMENTATION DEFINED instruction sequences that perform the required operations on entries that are not locked down.
- Implement one of the other permitted alternatives for the locked entries.

Arm recommends that, when possible, such IMPLEMENTATION DEFINED instruction sequences use architecturally-defined instructions. This minimizes the number of customized instructions required.

In addition, an implementation that uses an abort to handle cache maintenance instructions for entries that might be locked must provide a mechanism that ensures that no entries are locked in the cache.

The reset setting of the cache must be that no cache entries are locked.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Additional cache functions for the implementation of lockdown

An implementation can add additional cache maintenance functions for the handling of lockdown in the IMPLEMENTATION DEFINED spaces reserved for Cache Lockdown, see *Reserved encodings for IMPLEMENTATION DEFINED registers* on page D12-2866.

D4.4.11 System level caches

The Arm Architecture defines a *system cache* as a cache that is not described in the PE Cache Identification registers, CCSIDR_EL1 and CLIDR_EL1, and for which the set/way cache maintenance instructions do not apply.

Conceptually, three classes of system cache can be envisaged:

1. System caches which lie before the point of coherency and cannot be managed by any cache maintenance instructions. Such systems fundamentally undermine the concept of cache maintenance instructions operating to the point of coherency, as they imply the use of non-architecture mechanisms to manage coherency. The use of such systems in the Arm architecture is explicitly prohibited.
2. System caches which lie before the point of coherency and can be managed by cache maintenance by address instructions that apply to the point of coherency, but cannot be managed by cache maintenance by set/way instructions. Where maintenance of the entirety of such a cache must be performed, as in the case for power management, it must be performed using non-architectural mechanisms.
3. System caches which lie beyond the point of coherency and so are invisible to the software. The management of such caches is outside the scope of the architecture.

D4.4.12 Branch prediction

Armv8 does not define any branch predictor maintenance instructions for AArch64 state.

If branch prediction is architecturally visible, cache maintenance must also apply to branch prediction.

D4.4.13 Execution and data prediction restriction System instructions

When ARMv8.0-PredInv is implemented, the System instructions listed in *A64 System instructions for prediction restriction* on page C5-760 prevent predictions based on information gathered from earlier execution within a particular execution context from affecting the later Speculative execution within that context, to the extent that the speculative execution is observable through side-channels.

The prediction restriction System instructions being used by a particular execution context apply to:

- All control flow prediction resources that predict execution addresses.
- Data value prediction.
- Cache allocation prediction.

For these System instructions, the execution context is defined by:

- The SCR_EL3.NS bit.
- The Exception level.
- When executing at EL1, the VMID.
- When executing at EL0 when using the EL1&0 translation regime, the ASID and VMID.
- When executing at EL0 when using the EL2&0 translation regime, the VMID.

Note

- The data value prediction applies to all prediction resources that use some form of training to speculate data values as part of an execution.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The cache allocation applies to all instruction and data caches, and TLB prefetching hardware used by the executing PE that applies to the supplied context.

The context information is passed as a register argument, and is restricted so that:

- Execution of the System instruction at EL0 only applies to the current hardware defined context.
- Execution of the System instruction at EL1only applies to the current VMID and Security state, and does not apply to EL2 or EL3.
- Execution of the System instruction at EL2 can only apply to the current Security state, and does not apply to EL3.

If the System instruction is specified to apply to Exception levels that are not implemented, or which are higher than the Exception level that the System instruction is executed at, then the System instruction is treated as a `NOP`.

When the System instruction is complete and synchronized, no predictions of the restricted type for the affected context are influenced by the execution of the program before the System instruction in a manner that can be observed by the use of any side channels.

Note

Prediction restriction System instructions do not require the invalidation of prediction structures so long as the behavior described for completion is met by an implementation.

These System instructions are guaranteed to be complete following a DSB that covers both read and write behavior on the same PE that executed the original instruction. A subsequent Context synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

In AArch64 state, EL0 access to the System instructions is controlled by:

- When HCR_EL2.{E2H, TGE} is not {1, 1}, SCTLR_EL1.EnRCTX.
- When HCR_EL2.{E2H, TGE} == {1, 1}, SCTLR_EL2.EnRCTX.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D4.5 External aborts

The Arm architecture defines External aborts as errors that occur in the memory system, other than those that are detected by the MMU or debug logic. An External abort might signal a data corruption to the PE. For example, a memory location might have been corrupted, and this corruption is detected by hardware using a parity or error correction code (ECC). The error might have been propagated. The RAS Extension provides mechanisms for software to determine the extent of the corruption and contain propagation of the error. For more information, see the *ARM® Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for the ARMv8-A architecture profile*.

An External abort is one of the following:

- Synchronous.
- Precise asynchronous.
- Imprecise asynchronous.

For more information, see *Exception terminology* on page D1-2315.

The RAS Extension provides a more granular taxonomy of aborts. When the RAS Extension is not implemented, the Arm architecture does not provide any method to distinguish between precise asynchronous and imprecise asynchronous External aborts.

It is IMPLEMENTATION DEFINED which External aborts, if any, are supported.

External aborts on data accesses and translation table walks on data accesses can be either synchronous or asynchronous.

When ARMv8.4-DFE is not implemented, External aborts on instruction fetches and translation table walks on instruction fetches can be either synchronous or asynchronous.

When ARMv8.4-DFE is implemented, all External abort exceptions on instruction fetches and translation table walks on instruction fetches must be synchronous.

A synchronous External abort on an instruction fetch, including a translation table walk on an instruction fetch, is taken precisely using the Instruction Abort exception.

A synchronous External abort on a data read or write, including a translation table walk on a data read or write, is taken precisely using the Data Abort exception.

See *Synchronous exception types, routing and priorities* on page D1-2354.

An asynchronous External abort is taken using the SError interrupt exception. See *Asynchronous exception types, routing, masking and priorities* on page D1-2363.

The effect of a failed memory access is described in *Effect of Data Aborts and Watchpoints* on page D1-2359.

Normally, External aborts are rare. An imprecise asynchronous External abort is likely to be fatal to the process that is running, Arm recommends that implementations make External aborts precise wherever possible.

The following subsections give more information about possible External aborts:

- *Provision for the classification of External aborts*.
- *Parity or ECC error reporting, RAS Extension not implemented* on page D4-2525.

D4.5.1 Provision for the classification of External aborts

In AArch64 state, an implementation can use ESR_ELx.EA, ISS[9], to provide more information about synchronous External aborts. For all synchronous aborts other than synchronous External aborts, ESR_ELx.EA, ISS[9], returns a value of 0.

If the RAS Extension is implemented:

- The ESR_ELx.SET field provides information about the state of the PE following a synchronous External abort.
- The ESR_ELx.AET field might contain more information following an asynchronous abort taken as an SError interrupt.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The implementation might define error record registers.

For more information, see:

- *ISS encoding for an exception from an Instruction Abort* on page D13-2990.
- *ISS encoding for an exception from a Data Abort* on page D13-2992.
- *ISS encoding for an SError interrupt* on page D13-2999.
- *ARM® Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for the ARMv8-A architecture profile*.

D4.5.2 Parity or ECC error reporting, RAS Extension not implemented

The Arm architecture supports the reporting of both synchronous and asynchronous parity or ECC errors from the cache system. It is IMPLEMENTATION DEFINED what parity or ECC errors in the cache systems, if any, result in synchronous or asynchronous parity or ECC errors.

A fault code is defined for reporting parity or ECC errors, see *Use of the ESR_EL1, ESR_EL2, and ESR_EL3* on page D1-2342. However, when parity or ECC error reporting is implemented, it is implementation defined whether a parity or ECC error is reported using the assigned fault code or using another appropriate encoding.

For all purposes other than the Fault status encoding, parity or ECC errors are treated as External aborts.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D4.6 Memory barrier instructions

Memory barriers on page B2-135 describes the memory barrier instructions. This section describes the system level controls of those instructions.

D4.6.1 EL2 control of the shareability of data barrier instructions executed at EL0 or EL1

In an implementation that includes EL2 enabled in the current Security state and supports shareability limitations on the data barrier instructions, the HCR_EL2.BSU field can modify the required shareability of an instruction that is executed at EL0 or EL1. Table D4-8 shows the encoding of this field.

Table D4-8 EL2 control of shareability of barrier instructions executed at EL0 or EL1

HCR_EL2.BSU	Minimum shareability of barrier instructions
00	No effect, shareability is as specified by the instruction
01	Inner Shareable
10	Outer Shareable
11	Full system

For an instruction executed at EL0 or EL1, Table D4-9 shows how the HCR_EL2.BSU is combined with the shareability specified by the argument of the `DMB` or `DSB` instruction to give the scope of the instruction.

Table D4-9 Effect of HCR_EL2.BSU on barrier instructions executed at EL1 or EL0

Shareability specified by the `DMB` or `DSB` argument	HCR_EL2.BSU	Resultant shareability
Full system	Any	Full system
Outer Shareable	`00`, `01`, or `10`	Outer Shareable
	`11`, Full system	Full system
Inner Shareable	`00` or `01`	Inner Shareable
	`10`, Outer Shareable	Outer Shareable
	`11`, Full system	Full system
Non-shareable	`00`, No effect	Non-shareable
	`01`, Inner Shareable	Inner Shareable
	`10`, Outer Shareable	Outer Shareable
	`11`, Full system	Full system

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D4.7 Pseudocode description of general memory System instructions

This section lists the pseudocode describing general memory operations:

- *Memory data type definitions*.
- *Basic memory access*.
- *Aligned memory access* on page D4-2528.
- *Unaligned memory access* on page D4-2528.
- *Exclusives monitors operations* on page D4-2528.
- *Access permission checking* on page D4-2529.
- *Abort exceptions* on page D4-2529.
- *Memory barriers* on page D4-2529.

D4.7.1 Memory data type definitions

This section lists the memory data types.

The memory data types are:

- Address descriptor, defined by the `AddressDescriptor` type.
- Full address, defined by the `FullAddress` type.
- Memory attributes, defined by the `MemoryAttributes` type.
- Memory type, defined by the `MemType` enumeration.
- Device memory type, defined by the `DeviceType` enumeration.
- Normal memory attributes, defined by the `MemAttrHints` type.
- Cacheability attributes, defined by the `MemAttr_NC`, `MemAttr_WT`, and `MemAttr_WB` constants.
- Allocation hints, defined by the `MemHint_No`, `MemHint_WA`, `MemHint_RA`, and `MemHint_RWA` constants.
- Access permissions, defined by the `Permissions` type.

These types are defined in Chapter J1 *Armv8 Pseudocode*.

D4.7.2 Basic memory access

The two forms of the `_Mem[]` accessor, non-assignment (memory read) _Mem and assignment (memory write) _Mem, are the operations that perform single-copy atomic, aligned, little-endian memory accesses of `size` bytes to or from the underlying physical memory array of bytes.

The functions address the array using `desc.paddress`, that supplies:

- The physical address.
- An NS bit that selects between the Secure and Non-secure parts of the array.

The attributes in `desc.memattrs` are used by the memory system to determine caching and ordering behaviors as described in *Memory types and attributes* on page B2-153, *Ordering and observability* on page B2-124, and *Atomicity in the Arm architecture* on page B2-118.

An additional parameter to the `_Mem[]` accessor defines the access type, for example normal, exclusive, ordered, or streaming, and whether the access is made as part of a translation table walk.

The actual implemented array of physical memory might be smaller than the maximum size permitted by the architecture. In this case the scheme for aliasing is IMPLEMENTATION DEFINED, or some parts of the address space might give rise to External aborts or a System Error.

Note

The permitted physical memory size is:

- 2^{52} bytes in an implementation that includes ARMv8.2-LPA. However, only 2^{48} bytes are accessible using the 4KB or 16KB translation granule.
- 2^{48} bytes otherwise.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

PAMax() returns the IMPLEMENTATION DEFINED size of the physical address.

D4.7.3 Aligned memory access

The two MemSingle[] accessors, non-assignment (memory read) AArch64.MemSingle[] and assignment (memory write) AArch64.MemSingle[], make atomic, little-endian accesses of size bytes. These functions are defined in Chapter J1 *Armv8 Pseudocode*.

D4.7.4 Unaligned memory access

The two Mem[] accessors, Non-assignment (memory read) Mem[] and Assignment (memory write) Mem[], make accesses of the required type. If an access is not architecturally defined to be atomic, Mem[] synthesizes accesses from multiple calls to AArch64.MemSingle[]. It also reverses the byte order if the access is big-endian.

The AArch64.CheckAlignment() function checks the alignment of memory accesses.

D4.7.5 Exclusives monitors operations

The AArch64.SetExclusiveMonitors() function sets the Exclusives monitors for a block of bytes, the size of which is determined by size, at the virtual address defined by address.

The AArch64.ExclusiveMonitorsPass() function checks whether the Exclusives monitors are set to include the location of a number of bytes specified by size, at the virtual address defined by address. The atomic write that follows after the Exclusives monitors have been set must be to the same physical address. It is permitted, but not required, for this function to return FALSE if the virtual address is not the same as that used in the previous call to AArch64.SetExclusiveMonitors().

The ExclusiveMonitorsStatus() function returns 0 if the previous atomic write was to the same physical memory locations selected by AArch64.ExclusiveMonitorsPass() and therefore succeeded. Otherwise the function returns 1, indicating that the address translation delivered a different physical address.

The MarkExclusiveGlobal() procedure takes as arguments a FullAddress paddress, the PE identifier processorid and the size of the transfer. The procedure records that the PE processorid has requested exclusive access covering at least size bytes from address paddress. The size of the location marked as exclusive is IMPLEMENTATION DEFINED, up to a limit of 2KB and no smaller than two words, and aligned in the address space to the size of the location. It is CONSTRAINED UNPREDICTABLE whether this causes any previous request for exclusive access to any other address by the same PE to be cleared.

The MarkExclusiveLocal() procedure takes as arguments a FullAddress paddress, the PE identifier processorid and the size of the transfer. The procedure records in a local record that PE processorid has requested exclusive access to an address covering at least size bytes from address paddress. The size of the location marked as exclusive is IMPLEMENTATION DEFINED, and can at its largest cover the whole of memory but is no smaller than two words, and is aligned in the address space to the size of the location. It is IMPLEMENTATION DEFINED whether this procedure also performs a MarkExclusiveGlobal() using the same parameters.

The IsExclusiveGlobal() function takes as arguments a FullAddress paddress, the PE identifier processorid and the size of the transfer. The function returns TRUE if the PE processorid has marked in a global record an address range as exclusive access requested that covers at least size bytes from address paddress. It is IMPLEMENTATION DEFINED whether it returns TRUE or FALSE if a global record has marked a different address as exclusive access requested. If no address is marked in a global record as exclusive access, IsExclusiveGlobal() returns FALSE.

The IsExclusiveLocal() function takes as arguments a FullAddress paddress, the PE identifier processorid and the size of the transfer. The function returns TRUE if the PE processorid has marked an address range as exclusive access requested that covers at least the size bytes from address paddress. It is IMPLEMENTATION DEFINED whether this function returns TRUE or FALSE if the address marked as exclusive access requested does not cover all of size bytes from address paddress. If no address is marked as exclusive access requested, then this function returns FALSE. It is IMPLEMENTATION DEFINED whether this result is ANDed with the result of IsExclusiveGlobal() with the same parameters.

The ClearExclusiveByAddress() procedure takes as arguments a FullAddress paddress, the PE identifier processorid and the size of the transfer. The procedure clears the global records of all PEs, other than processorid, for which an address region including any of size bytes starting from paddress has had a request for an exclusive access. It is

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

IMPLEMENTATION DEFINED whether the equivalent global record of the PE `processorid` is also cleared if any of `size` bytes starting from `paddress` has had a request for an exclusive access, or if any other address has had a request for an exclusive access.

The `ClearExclusiveLocal()` procedure takes as arguments the PE identifier `processorid`. The procedure clears the local record of PE `processorid` for which an address has had a request for an exclusive access. It is IMPLEMENTATION DEFINED whether this operation also clears the global record of PE `processorid` that an address has had a request for an exclusive access.

These functions are defined in Chapter J1 *Armv8 Pseudocode*.

D4.7.6 Access permission checking

The function `AArch64.CheckPermission()` is used by the architecture to perform access permission checking based on attributes derived from the translation tables or location descriptors. It returns the result of the call to `AArch64.NoFault()`.

These functions are defined in Chapter J1 *Armv8 Pseudocode*.

The interpretation of access permission is shown in *Memory access control* on page D5-2606.

D4.7.7 Abort exceptions

The function `AArch64.Abort()` generates either a Data Abort or an Instruction Abort exception by calling `AArch64.DataAbort()` or `AArch64.InstructionAbort()`. It also can generate a debug exception for debug related faults, see Chapter D2 *AArch64 Self-hosted Debug*.

The function `AArch64.DataAbort()` generates a Data Abort exception, routes the exception to EL2 or EL3, and records the information required for the Exception Syndrome registers, ESR_ELx. See *ISS encoding for an exception from a Data Abort* on page D13-2992. A second stage abort might also record the intermediate physical address, IPA, but this depends on the type of the abort.

For a synchronous abort, `AArch64.DataAbort()` also sets the FAR to the VA of the abort.

The function `AArch64.InstructionAbort()` generates an Instruction Abort exception, routes the exception to EL2 or EL3, and records the information required for the Exception Syndrome registers, ESR_ELx, see *ISS encoding for an exception from an Instruction Abort* on page D13-2990. A second stage abort might also record the intermediate physical address, IPA, but this depends on the type of the abort.

For a synchronous abort, `AArch64.InstructionAbort()` also sets the FAR to the VA of the abort.

The `FaultRecord` type describes a fault. Functions that check for faults return a record of this type appropriate to the type of fault. *Pseudocode description of the MMU faults* on page D5-2661 provides a number of wrappers to generate FaultRecords.

The function `AArch64.NoFault()` returns a null record that indicates no fault. The `IsFault()` function tests whether a FaultRecord contains a fault.

D4.7.8 Memory barriers

The definition for the memory barrier functions is given by the enumerations `MBReqDomain` and `MBReqTypes`.

These enumerations define the required shareability domains and required access types used as arguments for `DMB` and `DSB` instructions.

The procedures `DataMemoryBarrier`, `DataSynchronizationBarrier`, and `InstructionSynchronizationBarrier` perform the memory barriers.*

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter D5
The AArch64 Virtual Memory System Architecture

This chapter provides a system level view of the AArch64 *Virtual Memory System Architecture* (VMSAv8-64), the memory system architecture of an Armv8 implementation that is executing in AArch64 state. It contains the following sections:

- *About the Virtual Memory System Architecture (VMSA)* on page D5-2532.
- *The VMSAv8-64 address translation system* on page D5-2540.
- *VMSAv8-64 translation table format descriptors* on page D5-2593.
- *Memory access control* on page D5-2606.
- *Memory region attributes* on page D5-2628.
- *Virtualization Host Extensions* on page D5-2638.
- *Nested virtualization* on page D5-2644.
- *VMSAv8-64 memory aborts* on page D5-2652.
- *Nested virtualization* on page D5-2644.
- *Translation Lookaside Buffers (TLBs)* on page D5-2662.
- *TLB maintenance requirements and the TLB maintenance instructions* on page D5-2668.
- *Caches in a VMSAv8-64 implementation* on page D5-2685.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D5.1 About the Virtual Memory System Architecture (VMSA)

This chapter describes the Armv8 *Virtual Memory System Architecture* (VMSA), and in particular how it applies to a PE that is executing in AArch64 state. In this state the PE is using VMSAv8-64, as defined in *Armv8 VMSA naming*. See *The Armv8 VMSA when some Exception levels are using AArch32* for information about the VMSA in other contexts.

A VMSA provides a *Memory Management Unit* (MMU), that controls address translation, access permissions, and memory attribute determination and checking, for memory accesses made by the PE. The process of address translation maps the *virtual addresses* (VAs) used by the PE onto the *physical addresses* (PAs) of the physical memory system. The mapping of a VA to a PA requires either a single stage of translation, or two sequential stages of translation.

The translations are defined independently for different Exception levels and Security states, as described in *The VMSAv8-64 address translation system* on page D5-2540.

VMSAv8-64 supports tagging of VAs:

- Address tagging as described in *Address tagging in AArch64 state* on page D5-2534. As that section describes, this address tagging has no effect on the address translation process.
- If ARMv8.5-MemTag is implemented and enabled Memory tagging as described in Chapter D6 *Armv8.5 Memory Tagging Extension*.

The remainder of this chapter gives a full description of VMSAv8-64 for an implementation that includes all of the Exception levels. *The implemented Exception levels and the resulting translation stages and regimes* on page D5-2545 describes the differences in the VMSA if some Exception levels are not implemented.

The following sections give more information about the VMSA:

- *Armv8 VMSA naming*.
- *The Armv8 VMSA when some Exception levels are using AArch32*.
- *VMSA address types and address spaces* on page D5-2533.
- *Address tagging in AArch64 state* on page D5-2534.
- *Pointer authentication in AArch64 state* on page D5-2536.

D5.1.1 Armv8 VMSA naming

The Armv8 VMSA naming model reflects the possible stages of address translation, as follows:

VMSAv8 The overall translation scheme, within which an address translation has one or two stages.

VMSAv8-32 The translation scheme for a single stage of address translation that is managed from an Exception level that is using AArch32.

VMSAv8-32 is sometimes used to refer to the two stages of translation used to map a VA to a PA, where each stage is managed from an Exception level that is using AArch32.

VMSAv8-64 The translation scheme for a single stage of address translation that is managed from an Exception level that is using AArch64.

VMSAv8-64 is sometimes used to refer to the two stages of translation used to map a VA to a PA, where each stage is managed from an Exception level that is using AArch64.

D5.1.2 The Armv8 VMSA when some Exception levels are using AArch32

As stated at the start of the chapter, this chapter describes VMSAv8-64, the Armv8 VMSA that applies to an Exception level that is using AArch64. However, when a higher Exception level is using AArch64, and therefore using VMSAv8-64, lower Exception levels can be using AArch32. Chapter G5 *The AArch32 Virtual Memory System Architecture* describes VMSAv8-32, meaning it describes:

- The translation stages and translation regimes when EL3 is using AArch32.
- Any stages of address translation that are using VMSAv8-32 when EL3 is using AArch64.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

However, a PE can be executing at EL0 using AArch32 when the next higher Exception level is using AArch64, for example when EL0 is using AArch32 and EL1 is using AArch64. When this is the case execution at EL0 uses a VMSAv8-64 translation regime as described in *Constraints on accesses from EL0 when EL0 is using AArch32* on page D5-2543.

D5.1.3 VMSA address types and address spaces

A description of the VMSA refers to the following address types.

Note

These descriptions relate to the VMSAv8 description and therefore give more detail than the generic definitions given in the glossary.

Virtual address (VA)

An address used in an instruction, as a data or instruction address, is a Virtual Address (VA).

Note

This means that an address held in the PC, LR, SP, or an ELR, is a VA.

In AArch64 state, the VA has a maximum address width of either 48 bits or, when ARMv8.2-LVA is implemented and the 64KB translation granule is used, 52 bits. As *About address translation and supported input address ranges* on page D5-2544 describes, a stage of address translation can support one or two VA ranges:

Translation stage can support only a single VA range

For a translation stage that supports a single VA range, a 48-bit VA width gives a VA range of `0x0000000000000000` to `0x0000FFFFFFFFFFFF`.

If ARMv8.2-LVA is implemented and the 64KB translation granule is used, for a translation regime that supports a single VA range, the 52-bit VA width gives a VA range of `0x0000000000000000` to `0x000FFFFFFFFFFFFF`.

Translation stage can support two VA ranges

For a translation stage that supports two VA subranges, one at the bottom of the full 64-bit address range, and one at the top, as follows:

- The bottom VA range runs up from address `0x0000000000000000`.
  With a maximum VA width of 48 bits this gives a VA range of `0x0000000000000000` to `0x0000FFFFFFFFFFFF`.
  With a maximum VA width of 52 bits this gives a VA range of `0x0000000000000000` to `0x000FFFFFFFFFFFFF`.
- The top VA subrange runs up to address `0xFFFFFFFFFFFFFFFF`.
  With a maximum VA width of 48 bits this gives a VA range of `0xFFFF000000000000` to `0xFFFFFFFFFFFFFFFF`.
  With a maximum VA width of 52 bits this gives a VA range of `0xFFF0000000000000` to `0xFFFFFFFFFFFFFFFF`.
  Reducing the VA width for this subrange increases the bottom address of the range.

Note

- When ARMv8.1-VHE is not implemented, the only translation stage that can support two VA ranges is stage 1 of the EL1&0 translation regime.
- When ARMv8.1-VHE is implemented and the value of HCR_EL2.E2H is 1, stage 1 of the EL2, or EL2&0, translation regime also can support two VA ranges.

A 48-bit VA range corresponds to an address space of 256TB. A 52-bit VA range corresponds to an address space of 4PB.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Each translation regime that takes a VA as an input address can be configured to support fewer than the maximum number of bits of VA space, see *Address size configuration* on page D5-2547.

Intermediate physical address (IPA)

In a translation regime that provides two stages of address translation, the IPA is:

- The OA from the stage 1 translation.
- The IA for the stage 2 translation.

In a translation regime that provides only one stage of address translation, the IPA is identical to the PA. Alternatively, the translation regime can be considered as having no concept of IPAs.

The EL3, Secure EL1, and if ARMv8.4-SecEL2 is implemented, Secure EL2 Exception levels provide independent definitions of the PA spaces for Secure and Non-secure operation. This means they provide two independent address spaces, where:

- A VA accessed in Secure state can be translated to either the Secure or the Non-secure PA space.
- When in Non-secure state, a VA is always mapped to the Non-secure PA space.

For more information about maximum address widths, see *Address size configuration* on page D5-2547.

Physical address (PA)

The address of a location in a physical memory map. That is, an output address from the PE to the memory system.

The EL3, Secure EL1, and if ARMv8.4-SecEL2 is implemented, Secure EL2 Exception levels provide independent definitions of the PA spaces for Secure and Non-secure operation. This means they provide two independent address spaces, where:

- A VA accessed in Secure state can be translated to either the Secure or the Non-secure PA space.
- When in Non-secure state, a VA is always mapped to the Non-secure PA space.

For more information about maximum address widths, see *Address size configuration* on page D5-2547.

D5.1.4 Address tagging in AArch64 state

In AArch64 state, the Armv8 architecture supports the tagging of addresses. Address tagging in this section is not to be confused with Memory tagging if ARMv8.5-MemTag is implemented. For more information on Memory tagging see Chapter D6 *Armv8.5 Memory Tagging Extension*.

In the case of Address tagging the top eight bits of the VA are ignored when determining:

- If the translation system is enabled, whether the address is out of range and therefore causes a Translation fault.
- If the translation system is not enabled, whether the address is out of range and therefore causes an Address size fault.
- Whether the address requires invalidation when performing a TLB invalidation instruction by address.

The use of address tags is controlled as follows:

For addresses when stage 1 translation can support two VA ranges

The value of bit[55] of the VA determines the register bit that controls the use of address tags, as follows:

VA[55]==0 TCR_ELx.TBI0 determines whether address tags are used. If stage 1 translation is enabled, TTBR0_ELx holds the base address of the translation tables used to translate the address.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

VA[55]==1 TCR_ELx.TBI1 determines whether address tags are used. If stage 1 translation is enabled, TTBR1_ELx holds the base address of the translation tables used to translate the address.

For addresses when stage 1 translation supports only a single VA range

TCR_ELx.TBI determines whether address tags are used. If stage 1 translation is enabled, TTBR0_ELx holds the base address of the translation tables used to translate the address.

Note

The TCR_ELx.TBI{*n*} bit determines whether address tags are used regardless of whether the corresponding translation regime is enabled.

When ARMv8.3-PAuth is implemented, TBID{*n*} bits are added to TCR_ELx registers.

When a TCR_ELx.TBI{*n*} bit enables the use of address tagging, the corresponding TBID{*n*} bit determines whether address tagging is used for both instruction and data addresses, or only for data addresses.

The bits added are:

- TCR_EL1.{TBID1, TBID0}.
- If stage 1 of the EL2 or EL2&0 translation regime supports two VA ranges, TCR_EL2.{TBID1, TBID0}. Otherwise, TCR_EL2.TBID.
- TCR_EL3.TBID.

Note

Restricting address tagging to data addresses means instruction addresses can use larger Pointer authentication code fields. See *Pointer authentication in AArch64 state* on page D5-2536.

An address tag enable bit also has an effect on the PC value in the following cases:

- On taking an exception to the controlled Exception level, regardless of whether this is also the Exception level from which the exception was taken.
- Any branch within the controlled Exception level, unless that branch generates an Illegal exception return.
- On performing an exception return that is not an Illegal exception return to the controlled Exception level, regardless of whether this is also the Exception level from which the exception return was performed.

  Note

  On an Illegal exception return, bits[63:32] of the PC become UNKNOWN.

- Exiting from debug state to the controlled Exception level.

Note

As an example of what is meant by the *controlled Exception level*, TCR_EL3.TBI controls this effect for:

- A branch or procedure return within EL3.
- Taking an exception to EL3.
- Performing an exception return or a debug state exit to EL3.

The effect of the controlling TBI{*n*} bit is:

For a translation regime where stage 1 translation can support two VA ranges

If the controlling TBI*n* bit for the address being loaded into the PC is set to 1, then bits[63:56] of the PC are forced to be a sign-extension of bit[55] of that address.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For a translation regime where stage 1 translation supports only a single VA range

If the controlling TBI bit for the address being loaded into the PC is set to 1, then bits[63:56] of the PC are forced to be `0x00`.

However, when ARMv8.3-PAuth is implemented and the value of a TCR_ELx.TBID{n} field is 1, the *Effective value* of the corresponding TCR_ELx.TBI{n} field is 0 for any of:

- A branch or procedure return within an Exception level.
- Taking an exception to an Exception level.
- Exception return to an Exception level.
- Exit from Debug state to an Exception level.

The `AddrTop()` pseudocode function shows the algorithm determining the most significant bit of the VA, and therefore whether the VA is using tagging. For a translation regime where the stage 1 translation supports two VA ranges, this pseudocode includes the selection between TTBR0_ELx and TTBR1_ELx described in *Selection between TTBR0_ELx and TTBR1_ELx when two VA ranges are supported* on page D5-2578.

Note

The required behavior prevents a tagged address being propagated to the program counter.

When address tagging is enabled for an address that causes a Data Abort or a Watchpoint, the address tag is included in the VA returned in the FAR.

D5.1.5 Pointer authentication in AArch64 state

ARMv8.3-PAuth adds functionality that supports the authentication of the contents of a register before that register is used as the target of an indirect branch, or as a load. This functionality is supported only in AArch64 state.

For pointer authentication, this functionality provides:

- An instruction that inserts a *Pointer Authentication Code* (PAC) into the upper bits of a register. The bits used are the extension bits that do not hold valid address bits. The inserted PAC value is calculated from the value of the register and one other 64-bit value.
- An instruction that extracts the PAC from the upper bits of a register, and checks that the value is correct, based on the value of the register and one other 64-bit value, and:
  - If the value is correct, replaces the PAC with the extension bits.
  - Otherwise, replaces the PAC with the extension bits, except that two bits of the extension are set to a fixed unique number. This means that, if the register is used as the target of an indirect branch, execution branches to an address that generates a Translation fault because the VA is not mapped.
- An instruction that removes the PAC, replacing it with the extension bits, without any verification.

Multiple versions of these instructions are provided to support different use cases. These include instructions that combine a pointer authentication operation with another operation. *Pointer authentication instructions* on page C3-206 summarizes these instructions.

ARMv8.3-PAuth2 adds enhanced functionality that changes the mechanism by which a PAC is added to the pointer. The mechanism exclusive-ORs the upper bits of the pointer with the PAC instead of replacing the upper bits of the pointer with the PAC.

Note

If ARMv8.3-PAuth2 is implemented but ARMv8.3-FPAC is not implemented, when stage 1 translation is disabled, and TCR_ELx.T0SZ or TCR_ELx.T1SZ is set to indicate an address range that is smaller than the PA size, for some PAC values, the address generated by a failed PAC authentication can still be an address within the PA size. This means that using such an address that has failed PAC authentication for a memory access will not generate an Address size fault. Instead, the memory access will be performed that has its upper bits, those between the PA size and the size indicated by the TCR_ELx.T0SZ or TCR_ELx.T1SZ field, taken from the result of the authentication

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

process. As a result, if ARMv8.3-PAuth2 is implemented but ARMv8.3-FPAC is not implemented, when stage 1 translation is disabled, Arm recommends not setting the TCR_ELx.T0SZ or TCR_ELx.T1SZ values to indicate an address range that is smaller than the PA size.

For the Pointer authentication instructions, it is IMPLEMENTATION DEFINED whether PACs are generated using:

- The QARMA algorithm, see *The QARMA Block Cipher Family*. When this is the case, the value of ID_AA64ISAR1_EL1.APA is non-zero.
- An IMPLEMENTATION DEFINED algorithm. When this is the case, the value of ID_AA64ISAR1_EL1.API is `non-zero`.

ARMv8.3-PAuth provides a generic authentication instruction, `PACGA`, that generates a 32-bit PAC from two 64-bit values.

Note

The PACGA instruction can be used to provide protection for small blocks of memory. Instructions can be chained to allow protection of an arbitrary-sized block.

For the `PACGA` instruction, it is IMPLEMENTATION DEFINED whether PACs are generated using:

- The QARMA algorithm, see *The QARMA Block Cipher Family*. When this is the case, the value of ID_AA64ISAR1_EL1.GPA is `non-zero`.
- An IMPLEMENTATION DEFINED algorithm. When this is the case, the value of ID_AA64ISAR1_EL1.GPI is `non-zero`.

The pseudocode descriptions of the operation of these instructions describe the use of the QARMA algorithm. When an IMPLEMENTATION DEFINED algorithm is used the `ComputePAC`() function:

- Must have the same arguments as the function defined in this Manual.
- For a set of arguments passed to the function, must give the same result for all PEs that a thread of execution could migrate between.

Pointer authentication is implemented if the value of at least one of ID_AA64ISAR1_EL1.{APA, API, GPA, GPI} are not `0b0000`.

Note

Pointer authentication functionality is useful only when address translation is enabled. However, this functionality is the same whether address translation is enabled or disabled,

---

The following sections give more information about the ARMv8.3-PAuth, ARMv8.3-PAuth2, and ARMv8.3-FPAC functionality:

- *Supported PAC field and relation to the use of address tagging*.
- *Keys for PAC generation and verification* on page D5-2538.
- *System register control of pointer authentication* on page D5-2539.
- *Faulting on pointer authentication* on page D5-2539.

Supported PAC field and relation to the use of address tagging

As stated earlier in this section, the PAC is held in the extension bits of a register, that do not hold valid address bits. However, as described in *Address tagging in AArch64 state* on page D5-2534, when address tagging is used the tag is held in Xn[63:56]. Therefore, when Xn is a 64-bit register holding an address:

When address tagging is used

The PAC field is Xn[54:`bottom_PAC_bit`].

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When address tagging is not used

The PAC field is Xn[63:56, 54:`bottom_PAC_bit`].

In the PAC field definitions, `bottom_PAC_bit` == 64-TCR_ELx.T*n*SZ,

Note

Xn[55] determines whether the address lies in the upper or lower address range for the purpose of determining whether address tagging is used, see *Address tagging in AArch64 state* on page D5-2534. The value of Xn[55] is the value of *n* in T*n*SZ. Therefore, it also determines whether Xn[63:56] are part of the PAC field, and which of TCR_ELx.{T0SZ, T1SZ} determines the value of `bottom_PAC_bit`.

If the value of TCR_ELx.T*n*SZ is outside its permitted range then it is CONSTRAINED UNPREDICTABLE whether the value used to determine `bottom_PAC_bit` is the programmed value of the field, or is forced to the maximum or minimum permitted value of the field. However, if the PE treats an out of range TnSZ value as the maximum or minimum permitted value of the field for all purposes except reading the value of the field then that behavior also applies to determining `bottom_PAC_bit`.

ARMv8.3-PAuth adds a new control to TCR_ELx, that disables the use of address tagging for instruction addresses, see *Address tagging in AArch64 state* on page D5-2534.

Note

This control means software can use larger PAC field for instruction addresses, while using tagging and the smaller PAC field for data addresses.

Keys for PAC generation and verification

For pointer authentication, two 128-bit keys are provided for each of instruction addresses and data addresses, and a fifth 128-bit key is provided for the generic authentication instruction, as follows:

Keys for instruction address PACs

`APIAKey_EL1`
: The concatenation of the register values APIAKeyHi_EL1:APIAKeyLo_EL1.

`APIBKey_EL1`
: The concatenation of the register values APIBKeyHi_EL1:APIBKeyLo_EL1.

Keys for data address PACs

`APDAKey_EL1`
: The concatenation of the register values APDAKeyHi_EL1:APDAKeyLo_EL1.

`APDBKey_EL1`
: The concatenation of the register values APDBKeyHi_EL1:APDBKeyLo_EL1.

Key for generic authentication

`APGAKey_EL1`
: The concatenation of the register values APGAKeyHi_EL1:APGAKeyLo_EL1.

Note

Keys are not banked by Exception level. Arm expects software to switch the keys between Exception levels, typically by swapping the values with zero so that the current key values are not present in memory.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

System register control of pointer authentication

ARMv8.3-PAuth adds controls to the SCTLR_ELx registers that enable generation and validation of PACs for data and instruction addresses. Formally, the definition of these fields is that when the functionality is disabled the `AddPAC<I|D><A|B>()` and `Auth<I|D><A|B>()` pseudocode functions return the value of the first parameter passed to them. This means:

- Except for `PACGA`, the instructions listed in Table C3-12 on page C3-206, that add a PAC to an address in a register, execute as `NOP`s.
- The instructions listed in Table C3-13 on page C3-207, that authenticate a pointer, execute as `NOP`s.
- For the Combined instructions listed in Table C3-15 on page C3-208, the `Auth<I|D><A|B>()` function has no effect on the operation of the instruction, which operates as the equivalent non-Authenticate pointer instruction. This means that, for example:
  - A `RETAA` instruction operates as a `RET` instruction.
  - A `LDRAA Xt, [Xn, #<simm10>]!` instruction operates as a `LDR Xt, [Xn, #<simm10>:000]!` instruction.

These controls do not affect the `PACGA` and `XPAC*` instructions, that are always enabled.

The controls added to the SCTLR_ELx registers are:

EnIA Controls instructions that apply to PACs for instruction addresses that are generated using the `APIAKey_EL1` key.

EnIB Controls instructions that apply to PACs for instruction addresses that are generated using the `APIBKey_EL1` key.

EnDA Controls instructions that apply to PACs for data addresses that are generated using the `APDAKey_EL1` key.

EnDB Controls instructions that apply to PACs for data addresses that are generated using the `APDBKey_EL1` key.

See the SCTLR_ELx.{EnIA, EnIB, EnDA, EnDB} field descriptions for more information.

Note

These fields are RES0 in versions of the architecture before Armv8.3, and therefore should be written as 0 by legacy software.

Faulting on pointer authentication

In addition to ARMv8.3-PAuth2, ARMv8.3-FPAC introduces faulting on instructions that authenticate a PAC and, optionally, on the combined instructions that include pointer authentication. If the PAC supplied is incorrect on any instructions listed in Table C3-13 on page C3-207, that authenticate a PAC, the instruction generates a synchronous exception.

It is IMPLEMENTATION DEFINED whether the combined instructions listed in Table C3-15 on page C3-208 generate an exception directly from an authorization failure, rather than changing the address in a way that will generate a Translation fault when the address is accessed.

If an exception from an authorization failure is generated at EL0 and HCR_EL2.TGE==1, the exception is taken at EL2. Otherwise, the exception is taken at EL1.

An exception from the authorization failure generated at any other Exception level is taken at the same Exception level. The ESR_ELx.EC code used for such an exception is `0x1C`.

Synchronous exception prioritization for exceptions taken to AArch64 state on page D1-2355 describes the prioritization of exceptions taken from an authorization failure.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D5.2 The VMSAv8-64 address translation system

The following subsections describe the VMSAv8-64 address translation system, that maps VAs to PAs:

- *About the VMSAv8-64 address translation system*.
- *The implemented Exception levels and the resulting translation stages and regimes* on page D5-2545.
- *Controlling address translation stages* on page D5-2545.
- *Memory translation granule size* on page D5-2554.
- *Translation tables and the translation process* on page D5-2561.
- *Overview of the VMSAv8-64 address translation stages* on page D5-2564.
- *The VMSAv8-64 translation table format* on page D5-2575.
- *The algorithm for finding the translation table descriptors* on page D5-2582.
- *The effects of disabling a stage of address translation* on page D5-2586.
- *The implemented Exception levels and the resulting translation stages and regimes* on page D5-2545.
- *Pseudocode description of VMSAv8-64 address translation* on page D5-2588.
- *Address translation instructions* on page D5-2589.

Related to this:

- *VMSAv8-64 translation table format descriptors* on page D5-2593 describes the translation table entries.
- *Memory region attributes* on page D5-2628 describes the attributes that are held in the translation table entries, including how different attributes can interact.
- *Translation Lookaside Buffers (TLBs)* on page D5-2662 describes the caching of translation table lookups in TLBs, and the architected instructions for maintaining TLBs.
- *AArch64 Address translation examples* on page K7-7926 gives detailed descriptions of typical examples of translating a VA to a final PA, and obtaining the memory attributes of that PA.
- Chapter D6 *Armv8.5 Memory Tagging Extension*, gives details of ARMv8.5-MemTag and the modified behavior of the VMSAv8-64 address translation system when ARMv8.5-MemTag is implemented and enabled.

D5.2.1 About the VMSAv8-64 address translation system

The *Memory Management Unit* (MMU) controls address translation, memory access permissions, and memory attribute determination and checking, for memory accesses made by the PE.

The general model of MMU operation is that the MMU takes information about a required memory access, including an *input address* (IA), and either:

- Returns an associated *output address* (OA), and the *memory attributes* for that address.
- Is unable to perform the translation for one of a number of reasons, and therefore causes an exception to be generated. This exception is called an MMU fault. System registers are used to report any MMU faults that occur.

The process of mapping an IA to an OA is an *address translation*, or more precisely a single stage of address translation.

When using a VMSA, a *translation regime* maps a VA to a PA using one or two stages of translation, and:

- *The AArch64 translation regimes* on page D5-2541 defines the translation regimes.
- *VMSA address types and address spaces* on page D5-2533 give more information about VAs and PAs.

The *translation granule* specifies the granularity of the mapping from IA to OA. That is, it defines both:

- The *page size* for a stage of address translation, where a page is the smallest block of memory for which an IA to OA mapping can be specified.
- The size of a complete translation table for that stage of address translation.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The MMU is controlled by System registers, that provide independent control of each address translation stage, including a control to disable the stage of address translation. *The effects of disabling a stage of address translation* on page D5-2586 defines how the MMU handles an access for which a required address translation stage is disabled.

This section describes the address translation system for an implementation that includes all of the Exception levels, and gives a complete description of translations that are controlled by an Exception level that is using AArch64. In addition:

- *The Armv8 VMSA when some Exception levels are using AArch32* on page D5-2532 gives information about the VMSA when some Exception levels are using AArch32.
- *The implemented Exception levels and the resulting translation stages and regimes* on page D5-2545 describes the effect on the address translation model when some Exception levels are not implemented.

Each enabled stage of address translation uses a set of address translations and associated memory properties held in memory mapped tables called *translation tables*. A single translation table lookup can resolve only a limited number of bits of the IA, and therefore a single address translation can require multiple lookups. These are described as different levels of lookup.

Translation table entries can be cached in a *Translation Lookaside Buffer* (TLB).

As well as defining the OA that corresponds to the IA, the translation table entries define the following properties:

- For accesses made from Secure state, whether the access is to the Secure or Non-secure address map.
- Memory access permissions.
- Memory region attributes.

For more information, see *Memory attribute fields in the VMSAv8-64 translation table format descriptors* on page D5-2599.

The following subsections give more information:

- *The AArch64 translation regimes*.
- *About address translation and supported input address ranges* on page D5-2544.
- *The VMSAv8-64 translation table format* on page D5-2544.

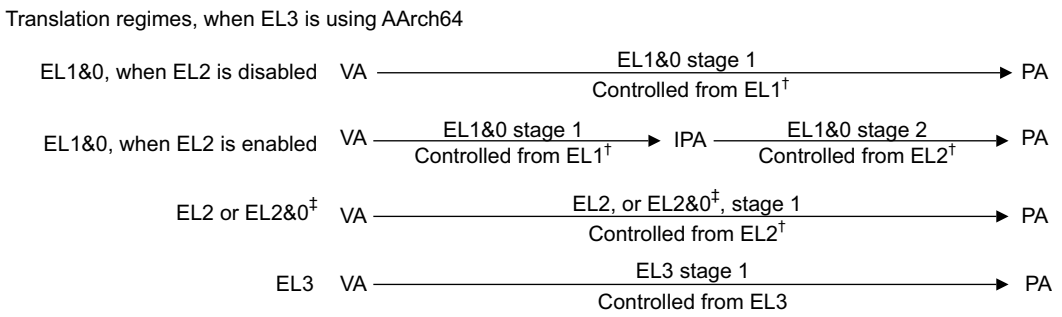
The AArch64 translation regimes

The architecture defines a number of *translation regimes*, where a translation regime comprises either:

- A single stage of address translation.
  This maps an input VA to an output PA.
- Two, sequential, stages of address translation, where:
  - Stage 1 maps an input VA to an output IPA.
  - Stage 2 maps an input IPA to an output PA.

Figure D5-1 on page D5-2542 shows these translation stages and translation regimes when EL3 is using AArch64.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta



† Typically controlled from this Exception level, but also accessible from higher Exception levels

‡Only when the implementation includes ARMv8.1-VHE and the value of HCR_EL2.E2H is 1

Figure D5-1 VMSAv8 AArch64 translation regimes, translation stages, and associated controls

This means that in VMSAv8-64 the set of translation regimes is:

The Secure EL1&0 translation regime, when EL2 is disabled

This has a single stage of translation, stage 1, that maps VAs to PAs and can support two VA ranges and the use of ASIDs.

This translation regime is used:

- For memory accesses from EL1 or EL0 when the value of HCR_EL2.{E2H, TGE} is {0,0}.

The Non-secure EL1&0 translation regime, when EL2 is disabled

This has a single stage of translation, stage 1, that maps VAs to PAs and can support two VA ranges and the use of ASIDs.

This translation regime is used:

- For memory accesses from EL1 or EL0 when the value of HCR_EL2.{E2H, TGE} is {0,0}.

The memory access will be Non-secure when SCR_EL3.NS is 1.

The Secure EL1&0 translation regime, when EL2 is enabled

If cached in a TLB, a translation table lookup for this regime is associated with the VMID that identifies the current *virtual machine*. This regime has two stages of lookup:

Stage 1 Maps VAs to IPAs. This stage can support two VA ranges and the use of ASIDs.

Stage 2 Maps IPAs to PAs. This stage supports a single IPA range.

This translation regime is used:

- For memory access from EL1 or EL0 when the value of HCR_EL2.{E2H, TGE} is {0, 0}.

The Non-secure EL1&0 translation regime, when EL2 is enabled

If cached in a TLB, a translation table lookup for this regime is associated with the VMID that identifies the current *virtual machine*. This regime has two stages of lookup:

Stage 1 Maps VAs to IPAs. This stage can support two VA ranges and the use of ASIDs.

Stage 2 Maps IPAs to PAs. This stage supports a single IPA range.

This translation regime is used:

- For memory access from EL1 or EL0 when the value of HCR_EL2.{E2H, TGE} is {0,0}.

The Secure EL2&0 translation regime

When ARMv8.1-VHE is implemented, this regime has a single stage of translation, stage 1, that maps VAs to PAs and can support two VA ranges and the use of ASIDs.

This translation regime is used:

- For memory accesses from EL0 when the value of HCR_EL2.{E2H, TGE} is {1,1}.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- For memory accesses from EL2 when the value of HCR_EL2.E2H is 1.

This translation regime is present when ARMv8.4-SecEL2 is implemented and enabled.

The Non-secure EL2&0 translation regime

When ARMv8.1-VHE is implemented, this regime has a single stage of translation, stage 1, that maps VAs to PAs and can support two VA ranges and the use of ASIDs.

This translation regime is used:

- For memory accesses from EL0 when the value of HCR_EL2.{E2H, TGE} is {1, 1}.
- For memory accesses from EL2 when the value of HCR_EL2.E2H is 1.

The Secure EL2 translation regime

This has a single stage of translation, stage 1, that maps VAs to PAs and supports a single VA range.

This translation regime is used:

- For all memory accesses from EL2 in implementations that do not include ARMv8.1-VHE.
- For all memory access from EL2, when ARMv8.1-VHE is implemented and HCR_EL2.E2H is 0.

This translation regime is present when ARMv8.4-SecEL2 is implemented and enabled.

The Non-secure EL2 translation regime

This has a single stage of translation, stage 1, that maps VAs to PAs and supports a single VA range.

This translation regime is used:

- For all memory accesses from EL2 in implementations that do not include ARMv8.1-VHE.
- For all memory access from EL2, when ARMv8.1-VHE is implemented and HCR_EL2.E2H is 0.

The Secure EL3 translation regime

This has a single stage of translation, stage 1, that maps VAs to PAs and supports a single VA range.

An MMU fault might be generated by a particular stage of translation. An MMU fault is described as either a stage 1 MMU fault or a stage 2 MMU fault.

Note

- In the Arm architecture, a software agent, such as an operating system, that uses or defines stage 1 memory translations, might be unaware of the second stage of translation, and of the distinction between IPA and PA.
- A more generalized description of the translation regimes is that a regime always comprises two sequential stages of translation, but in some regimes the stage 2 translation both:
  - Returns an OA that equals the IA. This is called a *flat mapping* of the IA to the OA.
  - Does not change the memory attributes returned by the stage 1 address translation.

Constraints on accesses from EL0 when EL0 is using AArch32

Armv8 permits execution with EL0 using AArch32 when the next higher Exception level is using AArch64. This happens in the following situations:

- EL1 is using AArch64. Execution at EL0 using AArch32 uses the VMSAv8-64 EL1&0 translation regime.
- EL2 is using AArch64 and the *Effective value* of HCR_EL2.{E2H, TGE} is {0, 1} or {1, 0}. Execution at EL0 using AArch32 uses the VMSAv8-64 EL1&0 translation regime.
- In an implementation that includes ARMv8.1-VHE, EL2 is using AArch64 and the value of HCR_EL2.{E2H, TGE} is {1, 1}. Execution at EL0 using AArch32 uses the VMSAv8-64 EL2&0 translation regime.

In this case, accesses from EL0 using AArch32 are using:

- The stated VMSAv8-64 translation regime, EL1&0 or EL2&0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The AArch32 memory model.

In particular, this means the accesses from EL0 are limited to a 32-bit VA range.

About address translation and supported input address ranges

For a single stage of address translation, a *Translation table base register* (TTBR_ELx) indicates the start of the first translation table required for a mapping from input address (IA) to output address (OA). For a stage of address translation that supports two VA ranges each VA range is an independent mapping from IA to OA. This means that each implemented translation stage shown in *VMSAv8 AArch64 translation regimes, translation stages, and associated controls* on page D5-2542 requires:

- Two associated sets of translation tables if it supports two IA ranges.
- One associated set of translation tables if it supports a single IA range.

Note

- Stage 2 translations never support two IA ranges. This means that, for the translation stages that support two IA ranges the IA is always a VA.
- *Example use of the split VA range, and the TTBR0_ELx and TTBR1_ELx controls* on page D5-2579 shows how two supported VA ranges might be used.

Controlling address translation stages on page D5-2545 summarizes the System registers that control address translation by the MMU, and *Selection between TTBR0_ELx and TTBR1_ELx when two VA ranges are supported* on page D5-2578 gives more information about the address translation stages that support two VA ranges.

A full translation table lookup is called a *translation table walk*. It is performed automatically by hardware, and can have a significant cost in execution time. To support fine granularity of the VA to PA mapping, a single IA to OA translation can require multiple accesses to the translation tables, with each access giving finer granularity. Each access is described as a *level* of address lookup. The final level of the lookup defines:

- The high bits of the required output address.
- The *attributes* and *access permissions* of the addressed memory.

Translation table entries can be cached in a *Translation Lookaside Buffer*, see *Translation Lookaside Buffers (TLBs)* on page D5-2662.

The VMSAv8-64 translation table format

Stages of address translation that are controlled by an Exception level that is using AArch64 use the VMSAv8-64 translation table format. This format uses 64-bit descriptor entries in the translation tables.

Note

This format is an extension of the VMSAv8-32 Long-descriptor translation table format originally defined by the Armv7 Large Physical Address Extension, and extended slightly by Armv8. VMSAv8-32 also supports a Short-descriptor translation table format. Chapter G5 *The AArch32 Virtual Memory System Architecture* describes both of these formats.

The VMSAv8-64 translation table format provides:

- Up to four levels of address lookup.
- A translation granule size of 4KB, 16KB, or 64KB.
- Input addresses of:
  - Up to 52 bits if ARMv8.2-LVA is implemented and the 64KB translation granule is used.
  - Otherwise, up to 48 bits.
- Output addresses of:
  - Up to 52 bits if ARMv8.2-LPA is implemented and the 64KB translation granule is used.
  - Otherwise, up to 48 bits.

For more information about input address and output address sizes see *Address size configuration* on page D5-2547.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D5.2.2 The implemented Exception levels and the resulting translation stages and regimes

About the VMSAv8-64 address translation system on page D5-2540 describes an implementation that includes all Exception levels. *Controlling address translation stages* describes the control of address translation by Exception levels that are using AArch64. This subsection describes how the address translation scheme changes if an implementation does not include all of the Exception levels.

If an implementation does not include EL3, it has only a single Security state, with MMU controls equivalent to the Secure state MMU controls.

If an implementation does not include EL2 then:

- If it also does not include EL3, the MMU provides only a single EL1&0 stage 1 translation regime.
- If it includes EL3, the MMU provides an EL1&0 stage 1 translation regime in each Security state.

Figure D5-1 on page D5-2542 shows the set of translation regimes for an implementation that implements all of the Exception levels. Table D5-1 shows how the supported translation stages depend on the implemented Exception levels, and in some cases on the Execution state being used by the highest implemented Exception level.

Table D5-1 The relation between the implemented translation stages and Exception levels for AArch64

Translation stage	Requires
Secure EL3 stage 1	EL3 implemented and using AArch64.
Secure EL2[a] stage 1	EL2 implemented and using AArch64.
Secure EL2&0[ab] stage 1	EL2 implemented and using AArch64.
Secure EL1&0[a] stage 2	EL2 implemented and using AArch64.
Secure EL1&0 stage 1	Either: • EL3 implemented and using AArch64. • Only EL1 and EL0 implemented, all operation is in Secure state, and EL1 is using AArch64.
Non-secure EL2 stage 1	EL2 implemented.
Non-secure EL2&0[b] stage 1	EL2 implemented.
Non-secure EL1&0 stage 2	EL2 implemented.
Non-secure EL1&0 stage 1	Any implementation except: • Only EL1 and EL0 implemented, with all operation in the Secure state.

a. This translation regime is supported only if an implementation includes ARMv8.4-SecEL2. When supported, it is used when the value of SCR_EL3.EEL2 is 1.

b. The EL2&0 translation regime is supported only if an implementation includes ARMv8.1-VHE. When supported, it is used when the value of HCR_EL2.E2H is 1.

D5.2.3 Controlling address translation stages

The implemented Exception levels and the resulting translation stages and regimes defines the translation regimes and stages. For each supported address translation stages controlled from AArch64, Table D5-2 on page D5-2546 shows:

- A System register bit enables the stage of address translation, SCTLR_ELx.M or HCR_ELx.VM.
- A System register bit determines the endianness of the translation table lookups, SCTLR_ELx.EE.
- A *Translation Control Register* (TCR_ELx) controls the stage of address translation.
- If a stage of address translation supports two VA ranges then that stage of translation provides:
  - A single TCR_ELx.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— A TTBR_ELx for each VA range. TTBR0_ELx points to the translation tables for the address range that starts at `0x0000000000000000`, and TTBR1_ELx points to the translation tables for the address range that ends at `0xFFFFFFFFFFFFFFFF`.

Otherwise, a stage of translation provides a single TCR_ELx and a single TTBR_ELx that holds the address of the translation table that must be used for the first lookup for the stage of address translation.

Table D5-2 Enable and endianness bits for the AArch64 translation stages

Translation stage	Controlled from	Controlling registers	
Secure EL3 stage 1	EL3	SCTLR_EL3.{EE, M}	TCR_EL3 TTBR0_EL3
Secure EL2[a] stage 1	Secure EL2	SCTLR_EL2.{EE, M}	TCR_EL2 TTBR0_EL2
Secure EL2&0[b] stage 1	Secure EL2	SCTLR_EL2.{EE, M}	TCR_EL2 TTBR0_EL2 TTBR1_EL2
Secure EL1&0[a] stage 2	Secure EL2	SCTLR_EL2.EE HCR_EL2.VM	VSTCR_EL2 VSTTBR_EL2 VTCR_EL2 VTTBR_EL2
Secure EL1&0 stage 1	Secure EL1	SCTLR_EL1.{EE, M}	TCR_EL1 TTBR0_EL1 TTBR1_EL1
Non-secure EL2 stage 1	Non-secure EL2	SCTLR_EL2.{EE, M}	TCR_EL2 TTBR0_EL2
Non-secure EL2&0[b] stage 1	Non-secure EL2	SCTLR_EL2.{EE, M}	TCR_EL2 TTBR0_EL2 TTBR1_EL2
Non-secure EL1&0 stage 2	Non-secure EL2	SCTLR_EL2.EE HCR_EL2.VM	VTCR_EL2 VTTBR_EL2
Non-secure EL1&0 stage 1	Non-secure EL1	SCTLR_EL1.{EE, M}	TCR_EL1 TTBR0_EL1 TTBR1_EL1

a. This translation regime is supported only if an implementation includes ARMv8.4-SecEL2. When supported, it is used when the value of SCR_EL3.EEL2 is 1.

b. The EL2&0 translation regime is supported only if an implementation includes ARMv8.1-VHE. When supported, it is used when the value of HCR_EL2.E2H is 1.

Note

If the PA of the software that enables or disables a particular stage of address translation differs from its VA, speculative instruction fetching can cause complications. Arm strongly recommends that the PA and VA of any software that enables or disables a stage of address translation are identical if that stage of translation controls translations that apply to the software currently being executed.

The following subsections give more information about controlling address translation:

- *System registers relevant to MMU operation* on page D5-2547.
- *Address size configuration* on page D5-2547.
- *Atomicity of register changes on changing virtual machine* on page D5-2553.
- *Use of out-of-context translation regimes* on page D5-2554.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If ARMv8.5-MemTag is implemented and enabled, Chapter D6 *Armv8.5 Memory Tagging Extension* provides further controls for the checking of Tagged and Untagged addresses.

System registers relevant to MMU operation

In AArch64 state, System registers have a suffix, that indicates the lowest Exception level from which they can be accessed. In some general descriptions of MMU control and address translation, this chapter uses a *Common abbreviation* for each of the System registers that affects MMU operation, as Table D5-3 shows. The common abbreviation is used when describing features that apply to multiple translation regimes or stages.

Note

The only translation regime that supports a stage 2 translation is the EL1&0 translation regime, when EL2 is enabled.

Table D5-3 Abbreviations for System registers used in this chapter

Common abbreviation	Translation stage	Exception level		
		EL1	EL2	EL3
HCR_ELx	-	-	HCR_EL2	-
SCTLR_ELx	-	SCTLR_EL1	SCTLR_EL2	SCTLR_EL3
TCR_ELx	Stage 1	TCR_EL1	TCR_EL2	TCR_EL3
	Stage 2	-	VTCR_EL2 VSTCR_EL2[a]	-
TTBR_ELx	Stage 1	TTBR0_EL1, TTBR1_EL1	TTBR0_EL2	TTBR0_EL3
	Stage 2	-	VTTBR_EL2, VSTTBR_EL2[a]	-
TTBR0_ELx	Stage 1	TTBR0_EL1	TTBR0_EL2	TTBR0_EL3
TTBR1_ELx	Stage 1	TTBR1_EL1	TTBR1_EL2[b]	-

a. Only when both the implementation includes ARMv8.4-SecEL2 and the value of SCR_EL3.EEL2 is 1.
b. Only when both the implementation includes ARMv8.1-VHE and the value of HCR_EL2.E2H is 1.

Address size configuration

The following subsubsections specify the configuration of the PA size and of the input and output address sizes for each of the stages of address translation:

- *Physical address size* on page D5-2548.
- *Output address size* on page D5-2548.
- *Input address size* on page D5-2549.
- *Supported IPA size* on page D5-2551.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Physical address size

The ID_AA64MMFR0_EL1.PARange field indicates the implemented PA size, as Table D5-4 shows.

Table D5-4 Physical address size implementation options

ID_AA64MMFR0_EL1.PARange	Total PA size	PA address size
0000	4 GB	32 bits, PA[31:0]
0001	64 GB	36 bits, PA[35:0]
0010	1 TB	40 bits, PA[39:0]
0011	4 TB	42 bits, PA[41:0]
0100	16 TB	44 bits, PA[43:0]
0101	256 TB	48 bits, PA[47:0]
0110	4PB	52 bits, PA[51:0][a]

a. Only when an implementation includes ARMv8.2-LPA and the 64KB translation granule is used, see *Extending addressing above 48 bits* on page D5-2552.

All other PARange values are reserved.

Output address size

For each enabled stage of address translation, TCR_ELx.{I}PS must be programmed to maximum output address size for that stage of translation, using the encodings as shown in Table D5-5.

Table D5-5 Output address size implementation options

TCR_ELx.{I}PS	Total output size	Output address size
000	4 GB	32 bits, PA[31:0]
001	64 GB	36 bits, PA[35:0]
010	1 TB	40 bits, PA[39:0]
011	4 TB	42 bits, PA[41:0]
100	16 TB	44 bits, PA[43:0]
101	256 TB	48 bits, PA[47:0]
110	4PB	52 bits, PA[51:0][a]

a. Only when an implementation includes ARMv8.2-LPA and is using the 64KB translation granule, see *Extending addressing above 48 bits* on page D5-2552.

Note

- The naming of this field is as follows:
  - **IPS**
    - In TCR_EL1.
    - In an implementation that includes ARMv8.1-VHE, in TCR_EL2 when the value of HCR_EL2.E2H is 1.
  - **PS** Otherwise.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The {I}PS fields are 3-bit fields, corresponding to the least-significant PARange bits shown in Table D5-4 on page D5-2548.

If {I}PS is programmed to a value larger than the implemented PA size, then the PE behaves as if programmed with the implemented PA size, but software must not rely on this behavior. That is, the output address size is never larger than the implemented PA size. Table D5-4 on page D5-2548 shows the implemented PA size.

The PE checks that the TTBR_ELx, translation table entries, and the output address for the stage of address translation have the address bits above the output address size set to zero. If this is not the case, an Address size fault is generated for the level and stage of translation that caused the fault. An Address size fault from the TTBR_ELx is always reported as a level 0 fault. When ARMv8.2-LPA is implemented and a translation granule of 4KB or 16KB is in use, all output addresses are treated as having bits[51:48] set to `0b0000`.

If stage 1 translation is disabled and the input address is larger than the implemented PA size, then a stage 1 level 0 Address size fault is generated.

Note

These faults are reported as level 0 faults even if they occur in a translation stage that does not perform level 0 lookups.

When using two stages of translation:

- If stage 2 translation is disabled and the output address from the stage 1 translation is larger than the implemented PA size, then a stage 1 Address size fault is generated for the level of the stage 1 translation that generated the output address.
- If stage 2 translation is enabled and the output address from the stage 1 translation does not generate a stage 1 Address size fault, but is larger than the input address size specified for the stage 2 translation, then a stage 2 Translation fault is generated.

Input address size

For each enabled stage of address translation, the TCR_ELx.T*x*SZ fields specify the input address size:

For a stage of translation that can support two VA ranges

The TCR_ELx has two T*x*SZ fields, corresponding to the two VA ranges:

- TCR_ELx.T0SZ specifies the size for the lower VA range, translated using TTBR0_ELx.
- TCR_ELx.T1SZ specifies the size for the upper VA range, translated using TTBR1_ELx.

For a stage of translation that supports only a single input address (IA) range

The TCR_ELx has a single T0SZ field, and IAs are translated using TTBR0_ELx.

Attempting to translate an address that is larger than the configured input address size generates a Translation fault. This means:

- For a TCR_ELx with a single T0SZ field and a 48-bit address size, Figure D5-2 on page D5-2550 shows the input address map:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

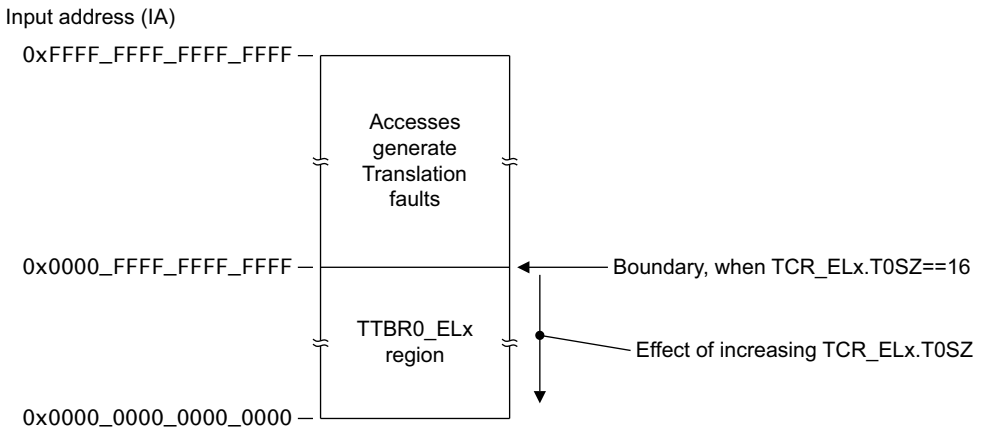


Figure D5-2 AArch64 input address map when using a single TTBR and 48-bit input address size

- For a TCR_ELx with two T*x*SZ fields, the input address is always a VA, and *Selection between TTBR0_ELx and TTBR1_ELx when two VA ranges are supported* on page D5-2578 describes the VA address map.

For the EL1&0 translation regime when EL2 is enabled, when both stages of translation are enabled, if the output address from the stage 1 translation does not generate a stage 1 address size fault, and is larger than the input address specified by VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ, then the input address size check for the stage 2 translation generates a Translation fault.

Although software can configure the input address size to be smaller than 48 bits, all implemented AArch64 TTBR_ELxs must support input address sizes of up to 48 bits, and in an implementation that includes Armv8.2-LVA, all TTBR_ELxs must support input address sizes of up to 52 bits.

Overview of the VMSAv8-64 address translation stages on page D5-2564 gives more information about the relationship between the required input address size, the value of T*x*SZ, and the required initial lookup level, and how these are affected by the translation granule size. However:

For all translation stages

If ARMv8.4-TTST is implemented, while the PE is executing in AArch64 state and is using 4KB or 16KB translation granules, the maximum TxSZ value is 48.

If ARMv8.4-TTST is implemented, while the PE is executing in AArch64 state and is using 64KB translation granules, the maximum TxSZ value is 47.

If ARMv8.4-TTST is not implemented or while the PE is executing in AArch32 state, the maximum T*x*SZ value is 39.

If T*x*SZ is programmed to a value larger than the defined maximum then it is IMPLEMENTATION DEFINED whether:

- The implementation behaves as if the field is programmed to the maximum for all purposes other than reading back the value of the field.
- Any use of the T*x*SZ value generates a Level 0 Translation fault for the stage of translation at which T*x*SZ is used.

For a stage 1 translation

The effective minimum value of T*x*SZ is 16 if ARMv8.2-LVA is not supported or if the translation granule that is in use is 4KB or 16KB in size. When ARMv8.2-LVA is supported, for the 64KB translation granule size only, the effective minimum value of T*x*SZ is 12.

If T*x*SZ is programmed to a value smaller than the effective minimum value, and if ARMv8.2-LVA is not supported, then it IMPLEMENTATION DEFINED whether:

- The implementation behaves as if the field were programmed to 16 for all purposes other than reading back the value of the field.
- Any use of the T*x*SZ value generates a stage 1 level 0 Translation fault.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If T*x*SZ is programmed to a value smaller than the effective minimum value when ARMv8.2-LVA is supported, then any use of the T*x*SZ value generates a stage 1 level 0 Translation fault.

For more information, see *Extending addressing above 48 bits* on page D5-2552.

For a stage 2 translation

Supported IPA size defines the effective minimum value of T0SZ, that depends on the supported PA size, and also describes the possible effects of programming T0SZ to a value that is smaller than this effective minimum value.

Supported IPA size

When EL2 is enabled in the current Security state, for the EL1&0 translation regime, the maximum IPA size is the maximum input address size for the second stage of translation is specified by VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ. For more information, see *Input address size* on page D5-2549 and *Output address size* on page D5-2548.

The maximum IPA size is constrained by the implemented PA size that is specified by ID_AA64MMFR0_EL1.PARange, see *Physical address size* on page D5-2548.

The implemented PA size also constrains the value of VTCR_EL2.SL0 and VSTCR_EL2.SL0, that specifies the level of the initial lookup. SL0 also depends on the translation granule, as described in *Overview of the VMSAv8-64 address translation stages* on page D5-2564.

Table D5-6 PA size implications for the VTCR_EL2.{T0SZ, SL0} and VSTCR_EL2.{T0SZ, SL0} fields

Supported PA size	Effective minimum T0SZ value	Valid initial lookup levels		
		4KB granule	16KB granule	64KB granule
32 bits	32 if EL1 is using AArch64 24 if EL1 is using AArch32	3[a], 2, 1	3, 2	3, 2
36 bits	28 if EL1 is using AArch64 24 if EL1 is using AArch32	3[a], 2, 1	3, 2	3, 2
40 bits	24	3[a], 2, 1	3, 2	3, 2
42 bits	22	3[a], 2, 1	3, 2, 1	3, 2
44 bits	20	3[a], 2. 1, 0	3, 2, 1	3, 2, 1
48 bits	16	3[a], 2, 1, 0	3, 2, 1	3, 2, 1
52 bits	12	-[b]	-[b]	3, 2, 1

a. Only supported if ARMv8.4-TTST is implemented, while the PE is executing in AArch64 state.

b. Only supported if ARMv8.2-LPA is implemented, and the PE is using the 64KB translation granule size.

If VTCR_EL2.SL0 is programmed to represent an initial lookup level not shown in Table D5-6, or is programmed to a reserved value, then any memory access that uses the second stage of translation generates a stage 2 level 0 Translation fault.

If VTCR_EL2.T0SZ is programmed to a value smaller than the effective minimum value shown in Table D5-6, and if ARMv8.2-LPA is not implemented, then the implementation consistently does one of the following:

- Treats the VTCR_EL2.T0SZ field as being programmed to the effective minimum value for all purposes other than reading back the value of the field.
- Treats the VTCR_EL2.T0SZ field as being programmed to the effective minimum value for all purposes other than:
  - Reading back the value of the field.
  - Checking whether the value of VTCR_EL2.T0SZ is consistent with the value of VTCR_EL2.SL0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Generates a stage 2 level 0 Translation fault on any memory access that uses the second stage of translation.

If T0SZ is programmed to a value smaller than the effective minimum value when ARMv8.2-LPA is supported, then any use of the T0SZ value generates a stage 2 level 0 Translation fault.

For more information, see *Extending addressing above 48 bits*.

Note

Programming VTCR_EL2.T0SZ to a value smaller than the effective minimum value shown in Table D5-6 on page D5-2551 can never provide support for a larger address range than the range given by the effective minimum value, because the stage 1 output address will give an Address size fault if it is larger than either:

- The PA size, for a VMSAv8-64 stage 1 translation.
- 40 bits, for a VMSAv8-32 stage 1 translation.

Extending addressing above 48 bits

Armv8.2 defines the following options for supporting 52-bit addressing:

ARMv8.2-LVA Supports 52-bit VAs when using the 64KB translation granule. The maximum IPA and PA sizes remain 48-bit unless ARMv8.2-LPA is implemented.

ARMv8.2-LPA Supports 52-bit IPAs and PAs when using the 64KB translation granule. The maximum VA size remains 48-bit unless ARMv8.2-LVA is implemented.

ARMv8.2-LPA and ARMv8.2-LVA can be implemented independently of each other.

When using the 64KB translation granule, ARMv8.2-LPA supports Block descriptors in level 1 translation tables. In this case, a block covers a 4TB address range.

In all cases, 52-bit address ranges are supported only when using the 64KB translation granule. Maximum address sizes when using the other translation granules remain 48-bit.

See *Address size configuration* on page D5-2547 for how to configure use of 52-bit VAs when an implementation includes ARMv8.2-LVA.

When using the 64KB translation granule, the 52-bit input address size is supported as follows:

- In an implementation that includes ARMv8.2-LVA, for stage 1 translations the minimum value of TCR_ELx.T*n*SZ field is 12.

  If TCR_ELx.T*n*SZ is programmed to a value less than 12, any use of the TCR_ELx.T*n*SZ bit generates a stage 1 level 0 Translation fault.

- In an implementation that includes ARMv8.2-LVA, for a stage 2 translation the effective minimum value of VTCR_EL2.T0SZ and VSTCR_EL2.T0SZ is 12.

  If VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ is programmed to a smaller value than the effective minimum size, then any use of a stage 2 translation generates a stage 2 level 0 Translation fault.

Table D5-25 on page D5-2585 shows the translation table descriptor addressing for each level of lookup when using the 64KB translation granule.

In an implementation that includes ARMv8.2-LPA and is using the 64KB translation granule:

- Bits[15:12] of each valid translation table descriptor hold bits[51:48] of the output address, or of the address of the translation table to be used for the initial lookup at the next level of translation. If the implementation does not support 52-bit physical addresses, then it is IMPLEMENTATION DEFINED whether non-zero values for these bits generate an Address size fault. In this case, not generating an Address Size Fault is deprecated.

- For a stage 1 translation, bits[5:2] of TTBR0_ELx or TTBR1_ELx holds bits[51:48] of the address of the translation table to be used for the initial lookup of that translation regime. If the implementation does not support 52-bit physical addresses, then non-zero values for these bits generate an Address size fault.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- For a stage 2 translation, bits[5:2] of VTTBR_EL2 or VSTTBR_EL2 holds bits[51:48] of the address of the translation table to be used for the initial lookup of the stage 2 translation. If the implementation does not support 52-bit physical addresses, then non-zero values for these bits generate an Address size fault.
- The minimum alignment of a translation table containing fewer than eight entries is 64 bytes.

  Note

  This is because, when the OA space is more than 48 bits, TTBR_ELx[5:2] specifies bits[51:48] of the translation table base address, and a translation table of fewer than eight entries would require one or more bits of TTBR_ELx[5:2] to be RES0 if the table was aligned to its size.

For more information, see *VMSAv8-64 translation table level 0, level 1, and level 2 descriptor formats* on page D5-2593 and *Armv8 translation table level 3 descriptor formats* on page D5-2597.

The ID_AA64MMFR2_EL1.VA field indicates the supported VA size. This field has the value `0x1` if the implementation includes ARMv8.2-LVA.

The ID_AA64MMFR0_EL1.PArange field indicates the supported PA and IPA size. This field has the value `0x6` if the implementation supports 52-bit PAs.

In addition to the System registers discussed in this subsection, the ARMv8.2-LPA and ARMv8.2-LVA features affect the following System registers that contain addresses:

- HPFAR_EL2.
- PAR_EL1.
- VBAR_EL1, VBAR_EL2, VBAR_EL3.
- DBGBVR<n>_EL1.
- DBGWVR<n>_EL1.
- MDRAR_EL1.
- LOREA_EL1.
- LORSA_EL1.

The larger VA, IPA, and PA addresses also affect the following System instructions for TLB maintenance:

- TLBI IPAS2E1.
- TLBI IPAS2E1IS.
- TLBI IPAS2E1OS.
- TLBI IPAS2LE1.
- TLBI IPAS2LE1IS.
- TLBI IPAS2LE1OS.
- TLBI RIPAS2E1.
- TLBI RIPAS2E1IS.
- TLBI RIPAS2E1OS.
- TLBI RIPAS2LE1.
- TLBI RIPAS2LE1IS.
- TLBI RIPAS2LE1OS.

Atomicity of register changes on changing virtual machine

From the viewpoint of software executing at EL1 or EL0, when there is a switch from one virtual machine to another, the registers that control or affect address translation must be changed atomically. This applies to the registers for the EL1&0, when EL2 is enabled, translation regime. This means that all of the following registers must change atomically:

- The registers associated with the stage 1 translations:
  - MAIR_EL1 and AMAIR_EL1.
  - TTBR0_EL1, TTBR1_EL1, TCR_EL1, and CONTEXTIDR_EL1.
  - SCTLR_EL1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The registers associated with the stage 2 translations:
  - VTTBR_EL2 and VTCR_EL2.
  - SCTLR_EL2.

Note

Only some bits of SCTLR_EL1 affect the stage 1 translation, and only some bits of SCTLR_EL2 affect the stage 2 translation. However, in each case, changing these bits requires a write to the register, and that write must be atomic with the other register updates.

These registers apply to execution using the EL1&0, when EL2 is enabled, translation regime. However, when updated as part of a switch of virtual machines they are updated by software executing at EL2. This means the registers are *out of context* when they are updated, and no synchronization precautions are required.

Similar considerations apply when ARMv8.1-VHE is implemented.

Use of out-of-context translation regimes

The architecture requires that:

- When executing at EL3 or EL2, the PE must not use the registers associated with the EL1&0 translation regime for speculative memory accesses.
- When executing at EL3 the PE must not use the registers associated with the EL2 or EL2&0 translation regime for speculative memory accesses.

If SPE is not in use for a lower Exception level when entering an Exception level, on completion of a `DSB` instruction, then no new memory accesses using any translation table entries from a translation regime of an Exception level lower than the Exception level that has been entered will be observed by any observers, to the extent that those accesses are required to be observed as determined by the shareability and cacheability of those translation table entries.

If SPE is in use for a lower Exception level when entering an Exception level, on completion of a `PSB CSYNC` and a subsequent `DSB` instruction, then no new memory accesses using any translation table entries from a translation regime of an Exception level lower than the Exception level that has been entered will be observed by any observers, to the extent that those accesses are required to be observed as determined by the shareability and cacheability of those translation table entries.

Note

- This does not require that speculative memory accesses cannot be performed using those entries if it is impossible to tell that those memory accesses have been observed by the observers.
- This requirement does not imply that, on taking an exception to a higher Exception level, any translation table walks started before the exception was taken will be completed by the time the higher Exception level is entered, and therefore memory accesses required for such a translation table walk might, in effect, be performed speculatively. However, the execution of a `DSB` on entry to the higher Exception level ensures that these accesses are complete.

D5.2.4 Memory translation granule size

The memory translation granule size defines both:

- The maximum size of a single translation table.
- The memory *page* size. That is, the granularity of a translation table lookup.

VMSAv8-64 supports translation granule sizes of 4KB, 16KB, and 64KB. Support for each granule size is optional. If ARMv8.5-GTG is implemented, support for granule size in Stage 1 is indicated as shown in Table D5-7 on page D5-2555, and support for granule size in Stage 2 is indicated as shown in Table D5-8 on page D5-2555. Otherwise, support for granule size in both Stages 1 and 2 is indicated as shown in Table D5-7 on page D5-2555:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D5-7 Identifying supported granule sizes

Granule size	Support indicated by: Field	Values	
4KB	ID_AA64MMFR0_EL1.TGran4	`0b0000`	4KB granule size supported.
		`0b1111`	4KB granule size not supported.
16KB	ID_AA64MMFR0_EL1.TGran16	`0b0000`	16KB granule size not supported.
		`0b0001`	16KB granule size supported.
64KB	ID_AA64MMFR0_EL1.TGran64	`0b0000`	64KB granule size supported.
		`0b1111`	64KB granule size not supported.

Table D5-8 Identifying supported granule sizes for Level 2 translation

Granule size	Support indicated by: Field	Values	
4KB	ID_AA64MMFR0_EL1.TGran4_2	`0b0010`	4KB granule size supported at stage 2.
		`0b0001`	4KB granule size not supported at stage 2.
16KB	ID_AA64MMFR0_EL1.TGran16_2	`0b0010`	16KB granule size supported.at stage 2.
		`0b0001`	16KB granule size not supported at stage 2.
64KB	ID_AA64MMFR0_EL1.TGran64_2	`0b0010`	64KB granule size supported.at stage 2.
		`0b0001`	64KB granule size not supported at stage 2.

Note

From a hardware viewpoint, the TGran*_2 fields hold the same information as the corresponding TGran* fields.

In VMSAv8-64, each address translation stage is configured, independently, to use one of the supported granule sizes.

Note

- Using a larger granule size can reduce the maximum required number of levels of address lookup because:
  - The increased translation table size means the translation table holds more entries. This means a single lookup can resolve more bits of the input address.
  - The increased page size means more of the least-significant address bits are required to address a page. These address bits are flat mapped from the input address to the output address, and therefore do not require translation.
- Arm recommends that memory-mapped peripherals are separated by an integer multiple of the largest granule size supported by the operating system or hypervisor, to allow each peripheral to be managed independently.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D5-9 summarizes the effects of the different granule sizes.

Table D5-9 Effect of granule size on a stage of address translation

Property	4KB granule	16KB granule	64KB granule	Notes
Maximum number of entries in a translation table	512	2048 (2K)	8192 (8K)	-
Address bits resolved in one level of lookup	9	11	13	2^9=512, 2^{11}=2K, 2^{13}=8K
Page size	4KB	16KB	64KB	-
Page address range	VA[11:0]= PA[11:0]	VA[13:0]= PA[13:0]	VA[15:0]= PA[15:0]	2^{12}=4K, 2^{14}=16K, 2^{16}=64K

How the granule size affects the address translation process

As Table D5-9 shows, the translation granule determines the number of address bits:

- Required to address a memory page.
- That can be resolved in a single translation table lookup.

This means the translation granule determines how the *input address* (IA) is resolved to an *output address* (OA) by the translation process.

Because a single translation table lookup can resolve only a limited number of address bits, the IA to OA resolution requires multiple *levels* of lookup.

Considering the resolution of an IA range of 48 bits, with a translation granule size of 2^n bytes:

- The least-significant n bits of the IA address the memory page. This means OA[(n-1):0]=IA[(n-1):0].
- The remaining (48-n) bits of the IA, IA[47:n], must be resolved by the address translation.
- A translation table descriptor is 8 bytes. Therefore:
  - A complete translation table holds $2^{(n-3)}$ descriptors.
  - A single level of translation can resolve a maximum of (n-3) bits of address.

  Consider the translation process, working back from the final level of lookup, that resolves the least significant of the address bits that require translation. Because the translation needs to resolve IA[47:n] and a level of lookup can resolve (n-3) bits of address:
  - The final level of lookup resolves IA[(2n-4):n].
  - The previous level of lookup resolves IA[(3n-7):(2n-3)].

  However, the level of lookup that resolves the most significant bits of the IA might not require a full-sized translation table. Therefore, in general, for a 48-bit IA the address bits resolved in a level of lookup are:

  IA[Min(47, ((3-m)(n-3)+2n-4)):(n+(3-m)(n-3))], where:

  Min(*a*, *b*) Is a function that returns the minimum of a and b.

  m Indicates the level of lookup. This is defined so that the level that resolves the least significant bit of the translated IA bits is level 3.

The following diagrams show this model, for each of the permitted granule sizes.

Figure D5-3 on page D5-2557 shows how a 48-bit IA is resolved when using the 4KB translation granule.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

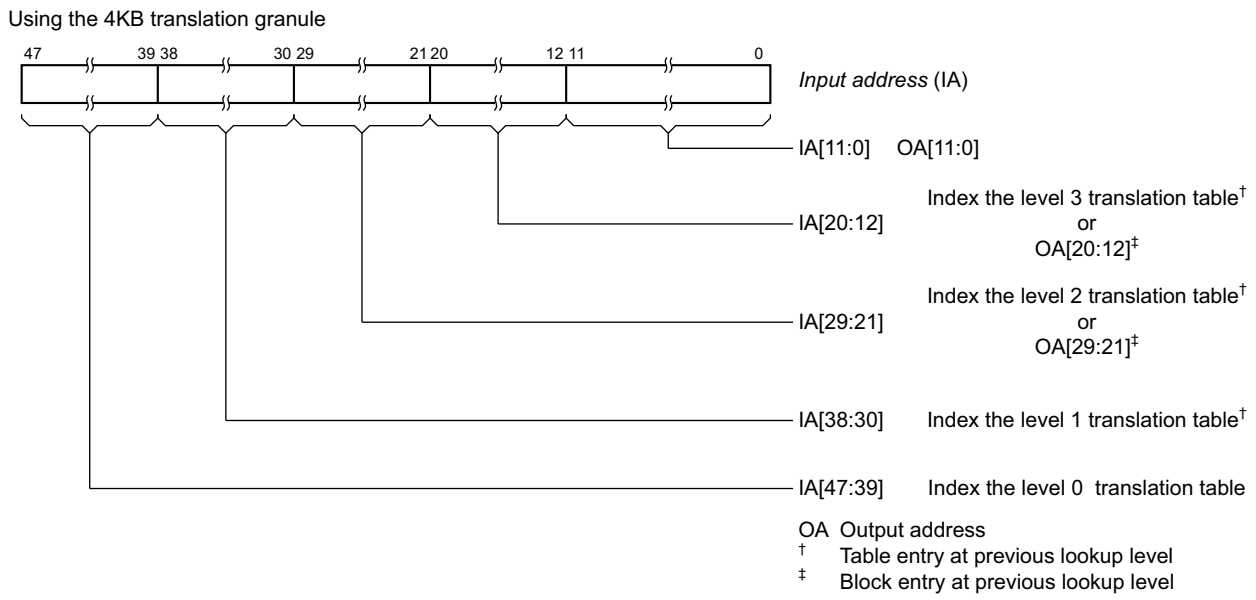


Figure D5-3 How a 48-bit IA is resolved when using the 4KB translation granule

Figure D5-4 shows how a 48-bit IA is resolved when using the 16KB translation granule.

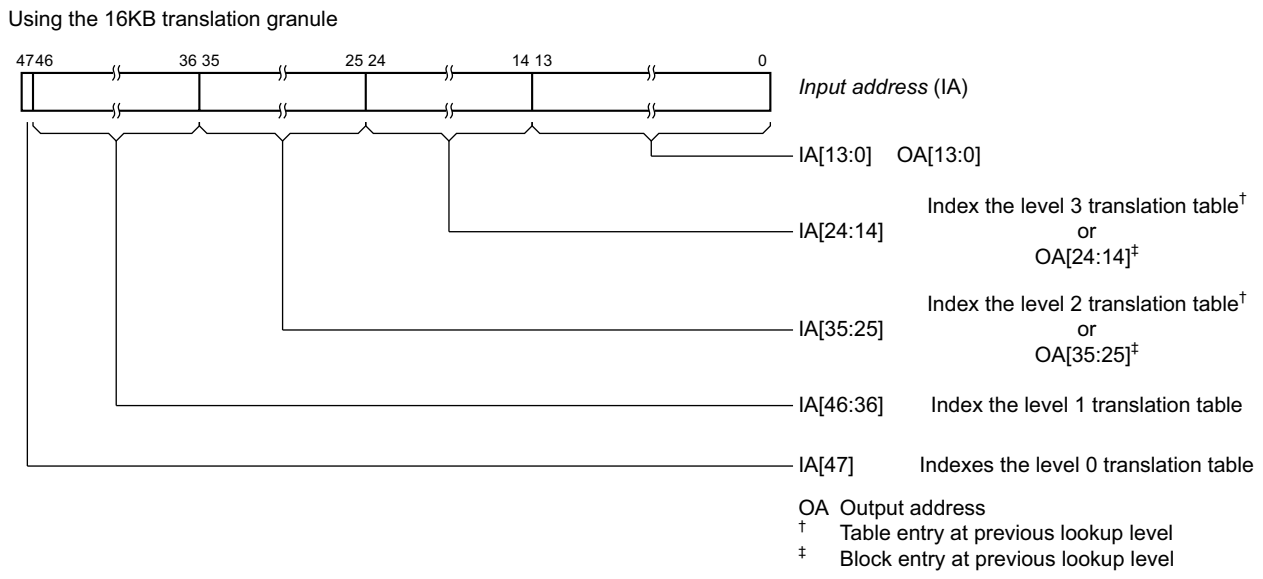


Figure D5-4 How a 48-bit IA is resolved when using the 16KB translation granule

Figure D5-5 on page D5-2558 shows how a 48-bit IA is resolved when using the 64KB translation granule.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

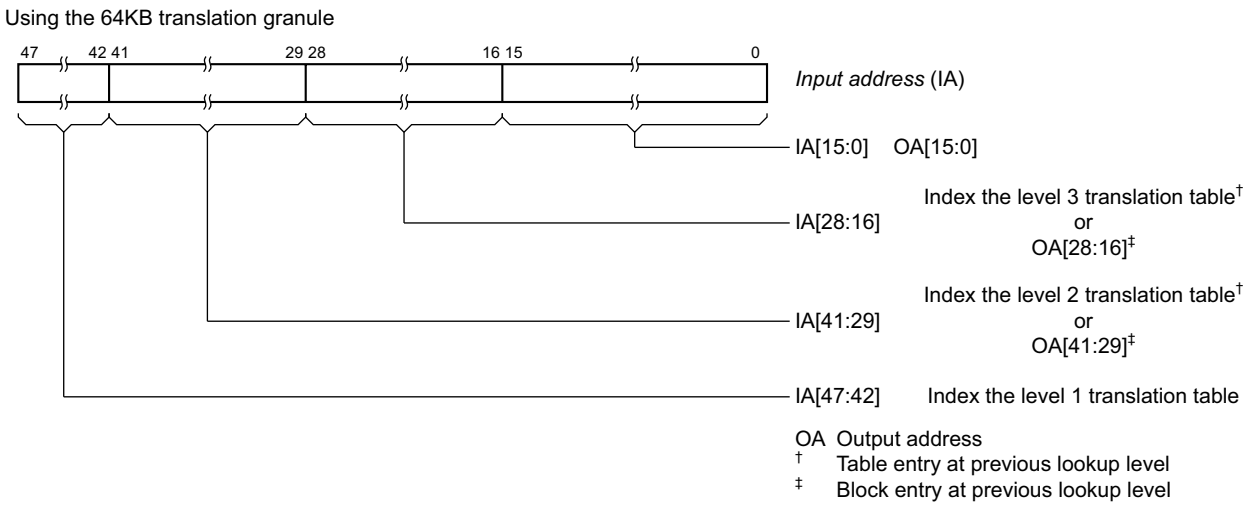


Figure D5-5 How a 48-bit IA is resolved when using the 64KB translation granule

In an implementation that includes ARMv8.2-LVA and is using the 64KB translation granule:

- The IA range that can be addressed by the level 1 lookup is IA[51:42].
- The level 1 lookup can directly address a block of memory, returning OA[51:42].

Note

The increased IA range means the size of the indexed level 1 translation table is increased.

Later sections of this chapter give more information about the translation process, and explain the terminology used in these figures.

Effect of granule size on translation table addressing and indexing

Table D5-10 shows the effect of the translation granule size on the addressing and indexing of the TTBR_ELx, and on the input address range that must be resolved.

Table D5-10 The effect of translation granule size on the translation tables

Granule size	Translation table		Translation resolves[a]	Notes
	Addressed by	Indexed by[b]		
4KB	TTBR_ELx[47:12]	IA[(x + 8):x]	IA[47:12]	One level of lookup resolves up to[c] 9 bits of IA
16KB	TTBR_ELx[47:14]	IA[(x + 10):x]	IA[47:14]	One level of lookup resolves up to[c] 11 bits of IA
64KB	TTBR_ELx[47:16][d] TTBR_ELx[5:2, 47:16]	IA[(x + 12):x]	IA[47:16][d] IA[51:16]	One level of lookup resolves up to[c] 13 bits of IA

a. When translating a maximum-sized input address, and accessing a page of memory.
b. Where the value of x depends on the lookup level, see Table D5-11 on page D5-2559.
c. Depending on the IA size, the initial lookup might resolve fewer bits of the IA.
d. For the 64KB granule entries in the *Addressed by* and *Translation resolves* columns, the second entry applies to an implementation that includes ARMv8.2-LVA and has selected an IA space larger than 47 bits, see *Extending addressing above 48 bits* on page D5-2552. The first entry applies otherwise.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D5-11 shows the IA bits resolved at each level of lookup, and how these correspond to the possible values of *x* in Table D5-10 on page D5-2558.

Table D5-11 IA bits resolved at different levels of lookup

Lookup level	4KB granule size	16KB granule size	64KB granule size
Zero	IA[47:39], $x = 39$	IA[47[a]], $x = 47$	-[b]
First	IA[38:30], $x = 30$	IA[46:36], $x = 36$	IA[47[a]:42], $x = 42$[c] IA[51[a]:42], $x = 42$
Second	IA[29:21], $x = 21$	IA[35:25], $x = 25$	IA[41:29], $x = 29$
Third	IA[20:12], $x = 12$	IA[24:14], $x = 14$	IA[28:16], $x = 16$

a. Smaller value than indicated in Table D5-10 on page D5-2558, as explained in this section.
b. Level 0 lookup not possible with 64KB granule size
c. The second entry applies to an implementation that includes ARMv8.2-LVA and has selected an IA space larger than 47 bits, see *Extending addressing above 48 bits* on page D5-2552, The first entry applies otherwise.

Table D5-10 on page D5-2558 refers to accessing a complete translation table, of 4KB, 16KB, or 64KB. However, the Armv8 translation system supports the following possible variations from the information in Table D5-10 on page D5-2558:

Reduced IA width

Depending on the configuration and implementation choices, the required input address width for the initial level of lookup might be smaller than the number of address bits that can be resolved at that level. This means that, for this initial level of lookup:

- The translation table size is reduced. For each 1 bit reduction in the input address size the size of the translation table is halved.

  Note

  — This has no effect on the translation table size for subsequent levels of lookup, for which the lookups always use full-sized translation tables.

  — For a stage 2 translation, it might be possible to start the translation at a lower level, see *Concatenated translation tables* on page D5-2560.

- More low-order TTBR_ELx bits are needed to hold the translation table base address.

Example D5-1 shows how this applies to translating a 35-bit input address range using the 4KB granule.

Example D5-1 Effect of an IA width of 35 bits when using the 4KB granule size

With a 4KB granule size, a single level of lookup can resolve up to 9 bits of IA. If an implementation has a 35-bit input address range, IA[34:0], Table D5-11 shows that lookup must start at level 1, and that the initial lookup must resolve IA[34:30], meaning it resolves 5 bits of address: This 4-bit reduction in the required resolution means:

- The translation table size is divided by 2^4, giving a size of 256B.
- The TTBR_ELx requires 4 more bits for the translation table base address, which becomes TTBR_ELx[47:8].

When using the 64KB translation granule to translate the maximum IA size of 48 bits, Table D5-11 shows that a level 1 lookup must resolve only IA[47:42]. This is 6 bits of address, compared to the 13 bits that can be resolved at a single level of lookup. This 7-bit reduction in the required resolution means:

- The translation table size is divided by 2^7, giving a size of 512B.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The TTBR_ELx requires 7 more bits for the translation table base address, which becomes TTBR_ELx[47:9].

Concatenated translation tables

For stage 2 address translations, for the initial lookup, up to 16 translation tables can be concatenated. This means additional IA bits can be resolved at that lookup level. The block of concatenated translation tables must be aligned to the size of the block of translation tables.

This means that each additional IA bit resolved:

- Doubles the number of translation tables required. Resolving an additional *n* bits requires 2^n concatenated translation tables at the initial lookup level.
- Reduces by 1 bit the width of the translation table base address held in the TTBR_ELx.

This means that, for the initial lookup of a stage 2 translation table, the IA ranges shown in Table D5-11 on page D5-2559 can be extended by up to 4 bits. Example D5-2 shows how concatenation can be used to resolve a 40-bit IA when using the 4KB translation granule.

Example D5-2 Concatenating translation tables to resolve a 40-bit IA range, with the 4K granule

---

Table D5-11 on page D5-2559 shows that, when using the 4KB translation granule, a level 1 lookup can resolve a 39-bit IA, with the first lookup resolving IA[38:30]. For a stage 2 translation, to extend the IA width to 40 bits and resolve IA[39:30] with the first lookup:

- Two translation tables are concatenated, giving a total size of 8KB.
- The TTBR_ELx requires 1 fewer bit for the translation table base address, which becomes TTBR_ELx[47:13].

For more information, see *Use of concatenated translation tables for the initial stage 2 lookup* on page D5-2580.

In all cases, the translation table, or block of concatenated translation tables, must be aligned to the actual size of the table or block of concatenated tables.

The translation table base address held in the TTBR_ELx is defined in the OA map for that stage of address translation. The information given in this section assumes this stage of translation has the maximum OA size, meaning the translation table base address is:

- TTBR_ELx[47:12] if using the 4KB translation granule.
- TTBR_ELx[47:14] if using the 16KB translation granule.
- TTBR_ELx[47:16] if using the 64KB translation granule with an OA of 48 bits.
- In an implementation that includes ARMv8.2-LPA and is using the 64KB translation granule, OA[51:16], where:
  - TTBR_ELx[5:2] holds OA[51:48].
  - TTBR_ELx[47:16] holds OA[47:16].

If the OA address is smaller than 48 bits then the upper bits of this field must be written as zero. For example, for a 40-bit OA range:

- If using the 4KB translation granule:
  - TTBR_ELx[47:40] must be set to zero.
  - TTBR_ELx[39:12] holds the translation table base address.
- If using the 16KB translation granule:
  - TTBR_ELx[47:40] must be set to zero.
  - TTBR_ELx[39:14] holds the translation table base address.
- If using the 64KB translation granule:
  - TTBR_ELx[47:40] must be set to zero.
  - TTBR_ELx[39:16] holds the translation table base address.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In all cases, if TTBR_ELx[47:40] is not zero, any attempt to access the translation table generates an Address size fault.

D5.2.5 Translation tables and the translation process

The following subsections describe general properties of the translation tables and translation table walks, that are largely independent of the translation table format:

- *Translation table walks*.
- *Ordering of memory accesses from translation table walks* on page D5-2563.
- *Security state of translation table lookups* on page D5-2563.
- *Control of translation table walks* on page D5-2564.

See also *Selection between TTBR0_ELx and TTBR1_ELx when two VA ranges are supported* on page D5-2578.

Translation table walks

A *translation table walk* comprises one or more *translation table lookups*. The translation table walk is the set of lookups that are required to translate the VA to the PA. For the EL1&0, when EL2 is enabled, translation regime, this set includes lookups for both the stage 1 translation and the stage 2 translation, but *translation table walk* can also be used to refer to either:

- The set of lookups required for the stage 1 translation, that translates the VA to the IPA. This is the *stage 1 translation table walk*.
- The set of lookups required for the stage 2 translation, that translates the IPA to the PA. This is the *stage 2 translation table walk*.

The information returned by a successful translation table walk is:

- The required PA. If the access is from Secure state this includes identifying whether the access is to the Secure PA space or the Non-secure PA space, see *Security state of translation table lookups* on page D5-2563.
- The memory attributes for the target memory region, as described in *Memory types and attributes* on page B2-153. For more information about how the translation table descriptors specify these attributes see *Memory region attributes* on page D5-2628.
- The access permissions for the target memory regions. For more information about how the translation table descriptors specify these permissions see *Memory access control* on page D5-2606.

The translation table walk starts with a read of the translation table for the initial lookup. The TTBR_ELx for the stage of translation holds the base address of this table. Each translation table lookup returns a descriptor, that indicates one of the following:

- The entry is the final entry of the walk. In this case, the entry contains the OA, and the permissions and attributes for the access.
- An additional level of lookup is required. In this case, the entry contains the translation table base address for that lookup. In addition:
  - The descriptor provides hierarchical attributes that are applied to the final translation, see *Hierarchical control of Secure or Non-secure memory accesses* on page D5-2605 and *Hierarchical control of data access permissions* on page D5-2611.
  - If the translation is in a Secure translation regime, the descriptor indicates whether that base address is in the Secure or Non-secure address space, unless a hierarchical control at a previous level of lookup has indicated that it must be in the Non-secure address space.
- The descriptor is invalid. In this case, the memory access generates a Translation fault.

Figure D5-6 on page D5-2562 gives a generalized view of a single stage of address translation where three levels of lookup are required.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

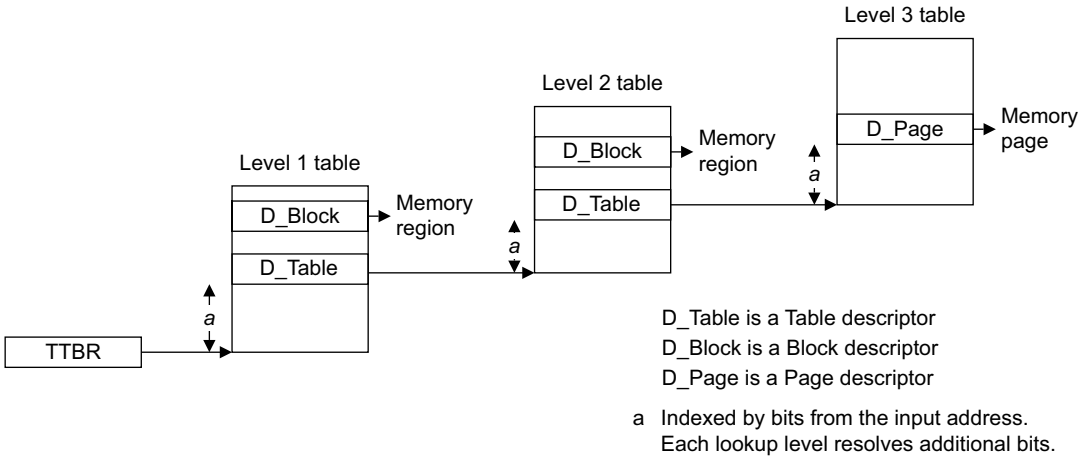


Figure D5-6 Generalized view of a stage of address translation

A translation table lookup from VMSAv8-64 performs a single-copy atomic 64-bit access to the translation table entry. This means the translation table entry is treated as a 64-bit object for the purpose of endianness. SCTLR_ELx.EE determines the endianness of the translation table lookups.

Note

Dynamically changing translation table endianness

Because any change to an SCTLR_ELx.EE, bit requires synchronization before it is visible to subsequent operations, Arm strongly recommends that any EE bit is changed only when either:

- Executing at an Exception level that does not use the translation tables affected by the EE bit being changed.
- Executing with address translation disabled for any stage of translation affected by the EE bit being changed.

Address translation stages are disabled by setting an SCTLR_ELx.M bit or the HCR_EL2.VM bit to 0. See the appropriate register description for more information.

The appropriate TTBR_ELx holds the output address of the base of the translation table used for the initial lookup, and:

- For all address translation stages other than EL1&0, when EL2 is enabled, stage 1 translations, the output address held in the TTBR_ELx, and any translation table base address returned by a translation table descriptor, is the PA of the base of the translation table.
- For EL1&0, when EL2 is enabled, stage 1 translations, the output address held in the TTBR_ELx, and any translation table base address returned by a translation table descriptor, is the IPA of the base of the translation table. This means that if stage 2 address translation is enabled, each of these OAs is subject to second stage translation.

Note

TLB caching can be used to minimize the number of translation table lookups that must be performed. For the EL1&0, when EL2 is enabled, translation regime, because each stage 1 OA generated during a translation table walk is subject to a stage 2 translation, if the caching of translation table entries is ineffective, a VA to PA address translation with two stages of translation can give rise to multiple translation table lookups. The number of lookups required is given by the following equation:

$$(S1+1)*(S2+1) - 1$$

Where, for this translation regime, S1 is the number of levels of lookup required for a stage 1 translation, and S2 is the number of levels of lookup required for a stage 2 translation.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The TCR_ELx determines the memory cacheability and shareability attributes that apply, for the corresponding stage of translation, to all translation table lookups generated by that stage of translation.

The Normal memory type is the memory type defined for a translation table lookup for a stage of translation.

Note

- In a two stage translation regime, a translation table lookup from stage 1, that has the Normal memory type defined at stage 1 by this rule, can still be given the Device memory type as part of the stage 2 translation of that address. Arm strongly recommends against such a remapping of the memory type, and the architecture includes a trap of this behavior to EL2. For more information, see *Stage 2 fault on a stage 1 translation table walk* on page D5-2658.
- The rules about mismatched attributes given in *Mismatched memory attributes* on page B2-164 apply to the relationship between translation table walks and explicit memory accesses to the translation tables in the same way that they apply to the relationship between different explicit memory accesses to the same location. For this reason, Arm strongly recommends that the attributes that the TCR_ELx applies to the translation tables are the same as the attributes that are applied for explicit accesses to the memory that holds the translation tables.

For more information see *Overview of the VMSAv8-64 address translation stages* on page D5-2564.

See also *Selection between TTBR0_ELx and TTBR1_ELx when two VA ranges are supported* on page D5-2578.

Ordering of memory accesses from translation table walks

A translation table walk is considered to be a separate observer in the same shareability domain as the PE doing the translation table walk, and:

- A write to the translation tables can be observed by that separate observer at any time after the execution of the instruction that performed that write, but is only guaranteed to be observed to the extent required by the shareability attributes after the execution of a `DSB` instruction by the PE that executed the instruction that performed that write to the translation tables.
- Any writes to the translation tables are not seen by any explicit memory access generated by a load or store that occurs in program order before the instruction that performs the write to the translation tables.

If ARMv8.0-ETS is implemented, and a memory access RW_1 is Ordered-before a second memory access RW_2, then RW_1 is also Ordered-before any translation table walk generated by RW_2 that generates any of the following:

- A Translation fault.
- An Address size fault.
- An Access flag fault.

Security state of translation table lookups

For a Non-secure translation regime, all translation table lookups are performed to Non-secure output addresses.

For a Secure translation regime, for the first stage of translation, the initial translation table lookup is performed to a Secure IPA.

If the translation table descriptor returned as a result of that initial lookup points to a second translation table, then the NSTable bit in that descriptor determines whether that translation table lookup is made to a Secure or to a Non-secure IPA.

This applies for all subsequent translation table lookups as part of that translation table walk, with the additional rule that any translation table descriptor that is returned from Non-secure memory is treated as if the NSTable bit in that descriptor indicates that the subsequent translation table lookup is to Non-secure memory.

Where the Secure IPA from a first stage translation table is translated by the second stage translation, the security of the output address of that memory access is determined by:

- For accesses made to Secure IPA space, the VSTCR_EL2.SA bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- For accesses made to Non-secure IPA space, the VTCR_EL2.NSA bit.

For a Secure translation regime, for the second stage of translation, the security of the output address of the translation table walk is determined by:

- For translation table walks for the Secure IPA space, the VSTCR_EL2.SW bit.
- For translation table walks for the Non-secure IPA space, the VTCR_EL2.NSW bit.

Control of translation table walks

When stage 1 translations of a translation can support two VA ranges the TCR_ELx.{EPD0, EPD1} bits determine whether, for that regime, the two sets of translation tables for stage 1 are valid. EPD0 indicates whether the tables that TTBR0_ELx points to is valid, and EPD1 indicates whether the tables that TTBR1_ELx points to is valid. The effect of these bits is:

EPD*n* == 0 The translation tables are valid, and can be used for a translation table lookup.

EPD*n* == 1 If a TLB miss occurs based on TTBR_ELx, a Translation fault is returned, and no translation table walk is performed. The fault is reported as a level 0 fault.

D5.2.6 Overview of the VMSAv8-64 address translation stages

As shown in *Memory translation granule size* on page D5-2554, the granule size determines significant aspects of the address translation process. *Effect of granule size on translation table addressing and indexing* on page D5-2558 shows, for each granule size:

- How the required input address range determines the required initial lookup levels.
- For stage 2 translations, the possible effect described in *Concatenated translation tables* on page D5-2560.
- The TTBR_ELx addressing and indexing for the initial lookup.

The following subsections summarize the multiple levels of lookup that can be required for a single stage of address translation that might require the maximum number of lookups:

- *Overview of VMSAv8-64 address translation using the 4KB translation granule*.
- *Overview of VMSAv8-64 address translation using the 16KB translation granule* on page D5-2567.
- *Overview of VMSAv8-64 address translation using the 64KB translation granule* on page D5-2572.

Overview of VMSAv8-64 address translation using the 4KB translation granule

The requirements for the level of the initial lookup are different for stage 1 and stage 2 translations.

Overview of stage 1 translations, 4KB granule

For a stage 1 translation, the required initial lookup level is determined only by the required input address range specified by the corresponding TCR_ELx.T*n*SZ field. When using the 4KB translation granule, Table D5-12 shows this requirement.

Table D5-12 TCR_ELx.T*n*SZ values and IA ranges, 4KB granule with no concatenation of tables

Initial lookup level	T*n*SZ values for and input address ranges[a] for starting at this level			
	$TnSZ_{min}$	IA_{max}	$TnSZ_{max}$	IA_{min}
0	16	IA[47:12]	24	IA[39:12]
1	25	IA[38:12]	33	IA[30:12]
2	34	IA[29:12]	42[b]	IA[21:12]
3[c]	43	IA[20:12]	48[c]	IA[15:12]

a. The IAs show the address bits to be resolved when addressing a page of memory, see the *Note* that follows.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

b. If ARMv8.4-TTST is not implemented, or while the PE is executing in AArch32 state, TnSZmax is 39.
c. Only available if ARMv8.4-TTST is implemented, while the PE is executing in AArch64 state.

These configuration options are also permitted for stage 2 translations.

Note

Some bits of the IA do not require resolution by the translation table lookup, because they always map directly to the OA, When using the 4KB translation granule, IA[11:0] = OA[11:0] for all translations.

Figure D5-7 shows the stage 1 address translation, for an address translation using the 4KB granule with an input address size greater than 39 bits.

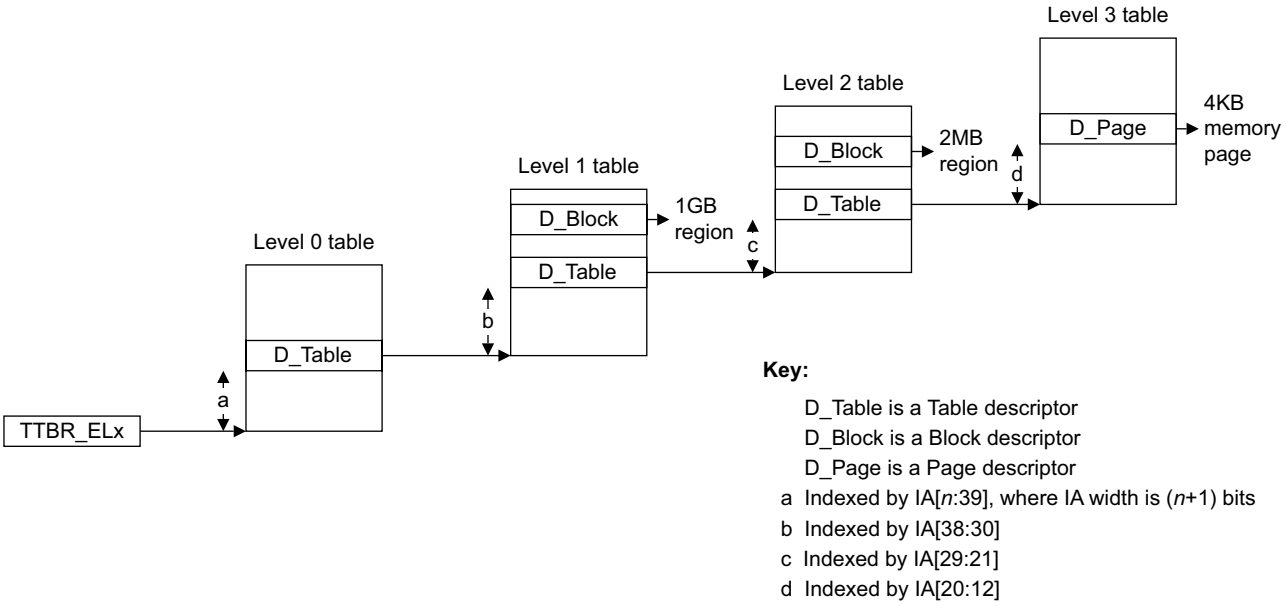


Figure D5-7 General view of VMSAv8-64 stage 1 address translation, 4KB granule

Overview of stage 2 translations, 4KB granule

For a stage 2 translation, up to 16 translation tables can be concatenated at the initial lookup level. For certain input address sizes, concatenating tables in this way means that the lookup starts at a lower level than would otherwise be the case. For more information see *Use of concatenated translation tables for the initial stage 2 lookup* on page D5-2580.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When using the 4KB translation granule, Table D5-13 shows all possibilities for the initial lookup for a stage 2 translation.

Table D5-13 VTCR_EL2.T0SZ values and IA ranges, 4KB granule with possible concatenation of translation tables

Tables[a]	1		2		4		8		16	
Initial lookup level (SL0 value)	T0SZ values and input address ranges[b] for starting at this level									
	T0SZ	IA	T0SZ	IA	T0SZ	IA	T0SZ	IA	T0SZ	IA
0 (2)	16-24	IA[47:12]-IA[39:12]	-	-	-	-	-	-	-	-
1 (1)	25-33	IA[38:12]-IA[30:12]	24	IA[39:12]	23	IA[40:12]	22	IA[41:12]	21	IA[42:12]
2 (0)	34-42[c]	IA[29:12]-IA[21:12]	33	IA[30:12]	32	IA[31:12]	31	IA[32:12]	30	IA[33:12]
3[d] (3)	43-48	IA[20:12]-IA[15:12]	42	IA[21:12]	41	IA[22:12]	40	IA[23:12]	39	IA[24:12]

a. Number of concatenated translation tables at the initial lookup level. *1 table* corresponds to no concatenation, also shown in Table D5-12 on page D5-2564.

b. The IAs shown in the table indicate the address bits to be resolved by an address translation addressing a page of memory, see the Note that follows.

c. If ARMv8.4-TTST is not implemented or while the PE is executing in AArch32 state, the maximum value of T0SZ is 39 with corresponding IA[29:12]-IA[24:12].

d. If ARMv8.4-TTST is implemented, while the PE is executing in AArch64 state, and is using 4KB granules, an initial lookup level 3, (VTCR_EL2.SL0 ==3) is possible.

Note

- Because concatenating translation tables reduces the number of levels of lookup required, when using the 4KB translation granule, tables cannot be concatenated at level 0.
- Some bits of the IA do not require resolution by the translation table lookup, because they always map directly to the OA. When using the 4KB translation granule, IA[11:0] = OA[11:0] for all translations.

Because the maximum number of concatenated translation tables is 16, there is a relationship between the permitted VTCR_EL2.{T0SZ, SL0} values. Table D5-13 shows the permitted T0SZ values for each initial lookup level.

If, when a translation table walk is started, the T0SZ value is not consistent with the SL0 value, or VTCR_EL2.SL0 is programmed to a reserved value, a stage 2 level 0 Translation fault is generated.

Figure D5-8 on page D5-2567 shows the stage 2 address translation, in a PE without ARMv8.4-TTST, for an input address size of between 40 and 43 bits. For an input address size in this range, the lookup can start at either level 0 or level 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

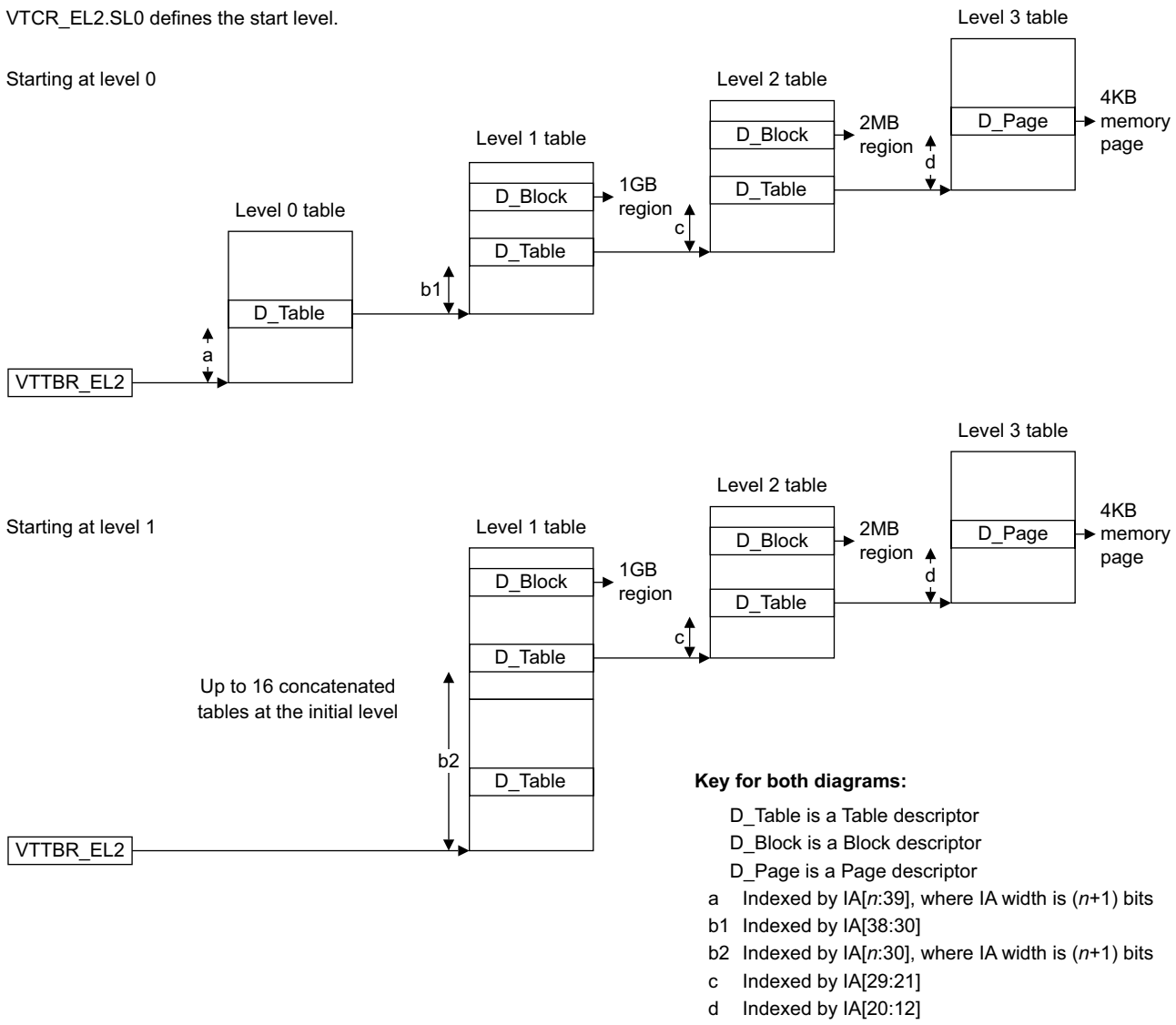


Figure D5-8 General view of VMSAv8-64 stage 2 address translation, 4KB granule

Overview of VMSAv8-64 address translation using the 16KB translation granule

The requirements for the level of the initial lookup are different for stage 1 and stage 2 translations.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Overview of stage 1 translations, 16KB granule

For a stage 1 translation, the required initial lookup level is determined only by the required input address range specified by the corresponding TCR_ELx.T*n*SZ field. When using the 16KB translation granule, Table D5-14 shows this requirement.

Table D5-14 TCR_ELx.T*n*SZ values and IA ranges, 16KB granule with no concatenation of tables

Initial lookup level	T*n*SZ values for and input address ranges[a] for starting at this level			
	$TnSZ_{min}$	IA_{max}	$TnSZ_{max}$	IA_{min}
0	16	IA[47:14]	16	IA[47:14]
1	17	IA[46:14]	27	IA[36:14]
2	28	IA[35:14]	38	IA[25:14]
3	39	IA[24:14]	48[b]	IA[15:14]

a. The IAs show the address bits to be resolved when addressing a page of memory, see the *Note* that follows.
b. If ARMv8.4-TTST is not implemented, the maximum is 39.

The configuration options for an initial lookup at level 1, level 2, or level 3 are also permitted for stage 2 translations, but stage 2 translation does not permit an initial lookup at level 0.

Note

- When using the 16KB translation granule, a maximum of 1 bit of IA is resolved by a level 0 lookup.
- Some bits of the IA do not require resolution by the translation table lookup, because they always map directly to the OA, When using the 16KB translation granule, IA[13:0] = OA[13:0] for all translations.

Figure D5-9 shows the stage 1 address translation, for an address translation using the 16KB granule with an input address size of 48 bits.

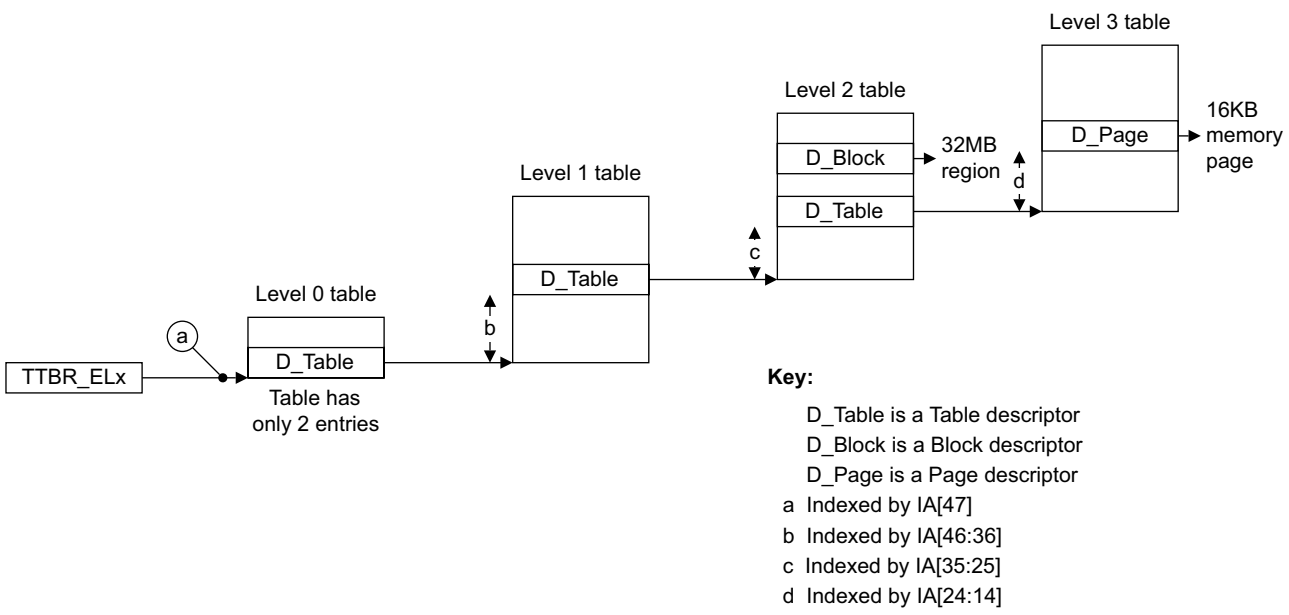


Figure D5-9 General view of VMSAv8-64 stage 1 address translation, 16KB granule

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Overview of stage 2 translations, 16KB granule

For a stage 2 translation, up to 16 translation tables can be concatenated at the initial lookup level. For certain input address sizes, concatenating tables in this way means that the lookup starts at a lower level than would otherwise be the case. For more information see *Use of concatenated translation tables for the initial stage 2 lookup* on page D5-2580.

When using the 16KB granule, for a stage 2 translation with an input address sized of 48 bits, the initial lookup must be at level 1, with two concatenated translation tables at this level.

When using the 16KB translation granule, Table D5-15 shows all possibilities for the initial lookup for a stage 2 translation.

Table D5-15 VTCR_EL2.T0SZ values and IA ranges, 16KB granule with possible concatenation of translation tables

Tables[a]	1		2		4		8		16	
Initial lookup level (SL0 value)	T0SZ values and input address ranges[b] for starting at this level									
	T0SZ	IA	T0SZ	IA	T0SZ	IA	T0SZ	IA	T0SZ	IA
1 (2)	17-27	IA[46:14]-IA[36:14]	16	IA[47:14]	-	-	-	-	-	-
2 (1)	28-38	IA[35:14]-IA[25:14]	27	IA[36:14]	26	IA[37:14]	25	IA[38:14]	24	IA[39:14]
3 (0)	39-48[c]	IA[24:14]-IA[15:14]	38	IA[25:14]	37	IA[26:14]	36	IA[27:14]	35	IA[28:14]

a. Number of concatenated translation tables at the initial lookup level. *1 table* corresponds to no concatenation, also shown in Table D5-14 on page D5-2568.

b. The IAs shown in the table indicate the address bits to be resolved by an address translation addressing a page of memory, see the Note that follows.

c. If ARMv8.4-TTST is not implemented or while the PE is executing in AArch32 state, the maximum value of T0SZ is 39 with corresponding IA[24:12].

Note

- When using the 16KB translation granule for a stage 2 translation, the initial lookup cannot be at level 0. When a 48-bit input address is required, translation must start with a level 1 lookup using two concatenated translation tables.
- Some bits of the IA do not require resolution by the translation table lookup, because they always map directly to the OA. When using the 16KB translation granule, IA[13:0] = OA[13:0] for all translations.

Because the maximum number of concatenated translation tables is 16, there is a relationship between the permitted VTCR_EL2.{T0SZ, SL0} values. Table D5-15 shows the permitted values of T0SZ for each initial lookup level.

If, when a translation table walk is started, the T0SZ value is not consistent with the SL0 value, or VTCR_EL2.SL0 is programmed to a reserved value, a stage 2 level 0 Translation fault is generated.

When stage 2 translation supports a 48-bit input address range, translation must start with a level 1 lookup using two concatenated translation tables. Figure D5-10 on page D5-2570 shows the translation for this case.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

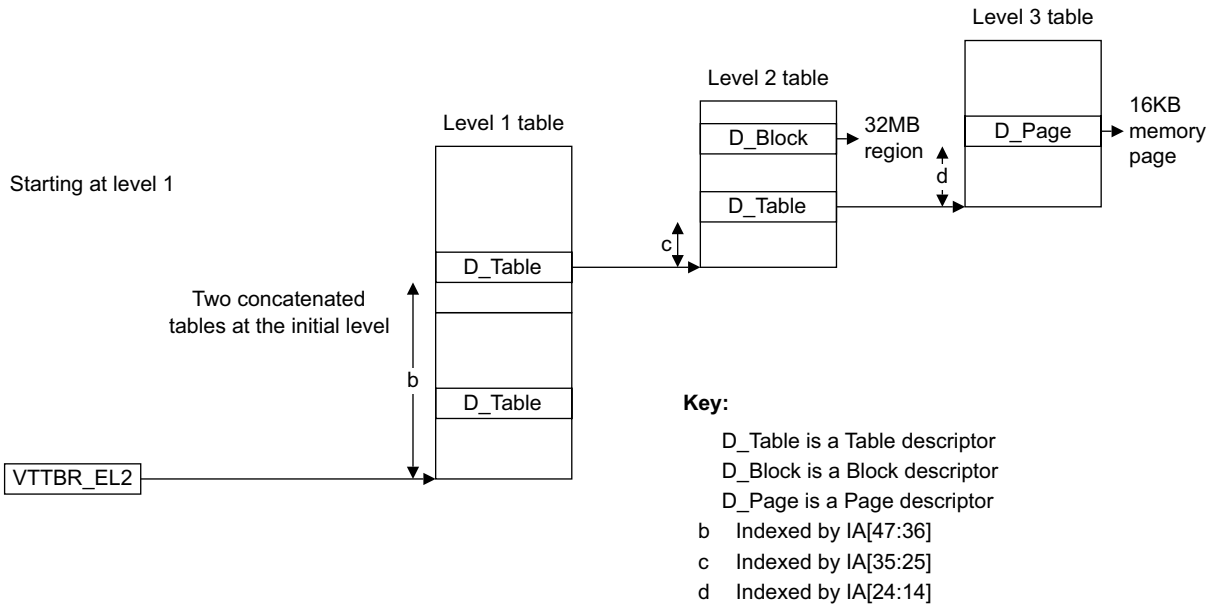


Figure D5-10 VMSAv8-64 stage 2 address translation, 16KB granule, 48 bit input address

However, for an input address size of between 37 and 40 bits, Table D5-15 on page D5-2569 shows that translation can start with either a level 1 lookup or a level 2 lookup, and Figure D5-11 on page D5-2571 shows these options.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

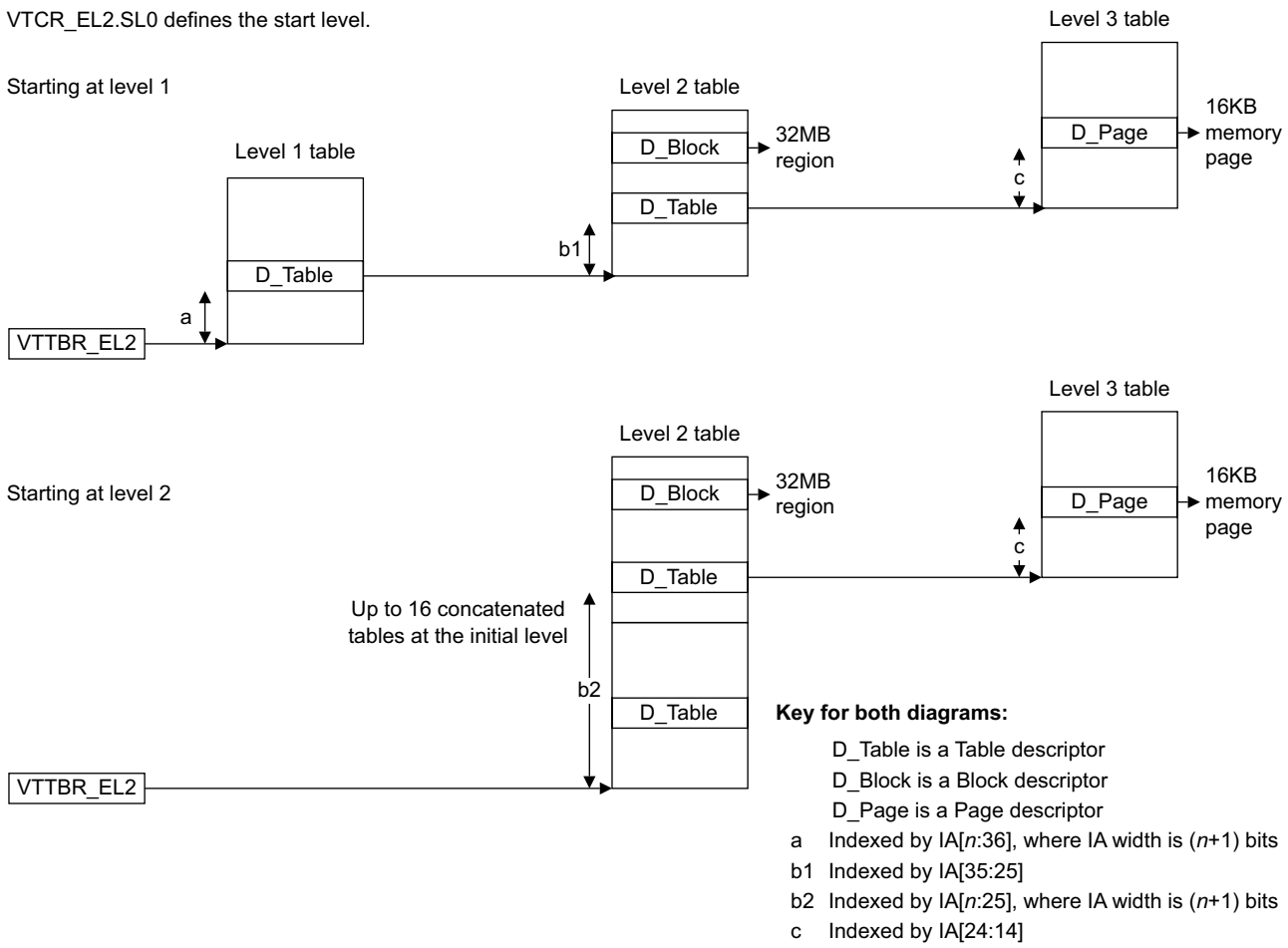


Figure D5-11 General view of VMSAv8-64 stage 2 address translation, 16KB granule

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Overview of VMSAv8-64 address translation using the 64KB translation granule

The requirements for the level of the initial lookup are different for stage 1 and stage 2 translations.

Overview of stage 1 translations, 64KB granule

For a stage 1 translation, the required initial lookup level is determined only by the required input address range specified by the corresponding TCR_ELx.TxSZ field. When using the 64KB translation granule, Table D5-16 shows this requirement.

Table D5-16 TCR_ELx.T*n*SZ values and IA ranges, 64KB granule with no concatenation of tables

Lookup level	T*n*SZ values for and input address ranges[a] for starting at this level			
	$TnSZ_{min}$	IA_{max}	$TnSZ_{max}$	IA_{min}
1[b]	12	IA[51:16]	21	IA[42:16]
1	16	IA[47:16]	21	IA[42:16]
2	22	IA[41:16]	34	IA[29:16]
3	35	IA[28:16]	47[c]	IA[16:16]

a. The IAs show the address bits to be resolved when addressing a page of memory, see the *Note* that follows.

b. Supported only if ARMv8.2-LVA is implemented and the 64KB translation granule is used, see *Extending addressing above 48 bits* on page D5-2552.

c. If ARMv8.4-TTST is not implemented or while the PE is executing in AArch32 state, the maximum value of TnSZ is 39 with IA[24:16].

These configuration options are also permitted for stage 2 translations.

Note

- When using the 64KB translation granule, there are no level 0 lookups.
- Some bits of the IA do not require resolution by the translation table lookup, because they always map directly to the OA. When using the 64KB translation granule, IA[15:0] = OA[15:0] for all translations.
- When ARMv8.2-LPA is implemented, a level 1 block attribute is supported when using the 64KB granule.

Figure D5-12 on page D5-2573 shows the stage 1 address translation, for an address translation using the 64KB granule with an input address size greater than 42 bits.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

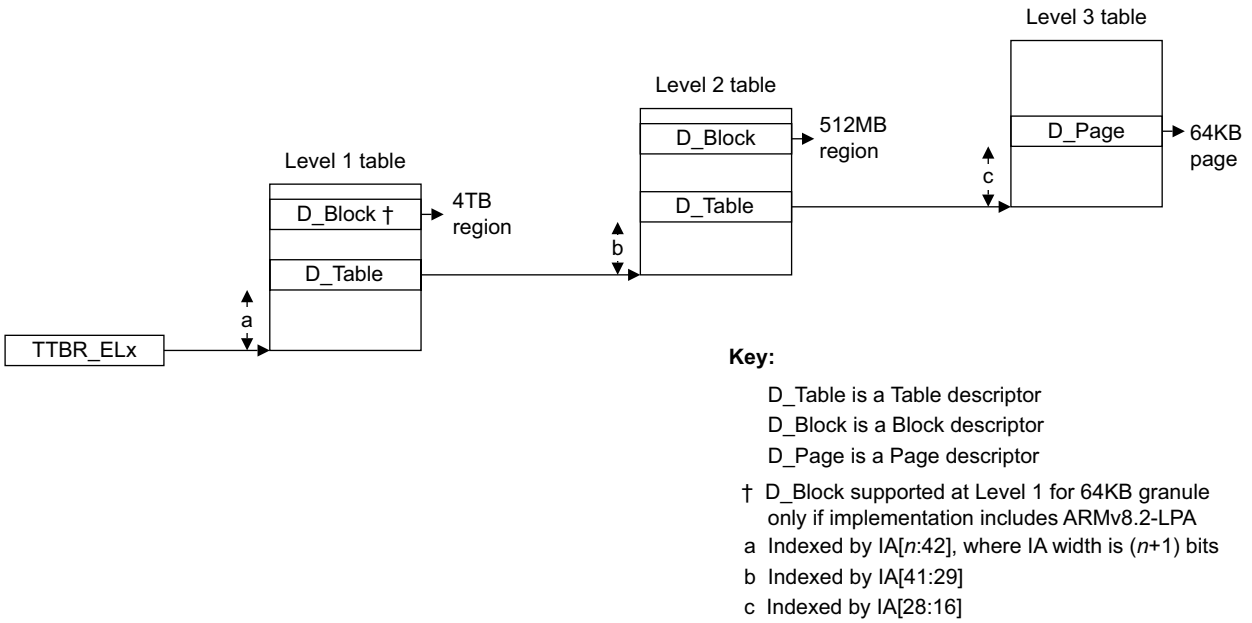


Figure D5-12 General view of VMSAv8-64 stage 1 address translation, 64KB granule with 52-bit VA support

Overview of stage 2 translations, 64KB granule

For a stage 2 translation, up to 16 translation tables can be concatenated at the initial lookup level. For certain input address sizes, concatenating tables in this way means that the lookup starts at a lower level than would otherwise be the case. For more information see *Use of concatenated translation tables for the initial stage 2 lookup* on page D5-2580.

When using the 64KB translation granule, Table D5-17 shows all possibilities for the initial lookup for a stage 2 translation.

Table D5-17 VTCR_EL2.T0SZ values and IA ranges, 64KB granule with possible concatenation of translation tables

Tables[a]	**1**		**2**		**4**		**8**		**16**	
Initial lookup level (SL0 value)	**T0SZ values and input address ranges[b] for starting at this level**									
	T0SZ	**IA**	**T0SZ**	**IA**	**T0SZ**	**IA**	**T0SZ**	**IA**	**T0SZ**	**IA**
1[c] (2)	12-21	IA[51:16]-IA[48:16]	-	-	-	-	-	-	-	-
1 (2)	16-21	IA[47:16]-IA[42:16]	-	-	-	-	-	-	-	-
2 (1)	22-34	IA[41:16]-IA[29:16]	21	IA[42:16]	20	IA[43:16]	19	IA[44:16]	18	IA[45:16]
3 (0)	35-47[d]	IA[28:16]-IA[16:16]	34	IA[29:16]	33	IA[30:16]	32	IA[31:16]	31	IA[32:16]

a. Number of concatenated translation tables at the initial lookup level. *1 table* corresponds to no concatenation, also shown in Table D5-16 on page D5-2572.

b. The IAs shown in the table indicate the address bits to be resolved by an address translation addressing a page of memory, see the *Note* that follows.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

c. Only supported if the PA size is 52 bits, see *Extending addressing above 48 bits* on page D5-2552.
d. If ARMv8.4-TTST is not implemented or while the PE is executing in AArch32 state, the maximum T0SZ value is 39, with IA[24:16].

Note

- When using the 64KB translation granule, there are no level 0 lookups.
- Because concatenating translation tables reduces the number of levels of lookup required, when using the 64KB translation granule, tables cannot be concatenated at level 1.
- Some bits of the IA do not require resolution by the translation table lookup, because they always map directly to the OA. When using the 64KB translation granule, IA[15:0] = OA[15:0] for all translations.

Because the maximum number of concatenated translation tables is 16, there is a relationship between the permitted VTCR_EL2.{T0SZ, SL0} values. Table D5-17 on page D5-2573 shows the permitted values of T0SZ for each initial lookup level.

If, when a translation table walk is started, the T0SZ value is not consistent with the SL0 value, or VTCR_EL2.SL0 is programmed to a reserved value, a stage 2 level 0 Translation fault is generated.

Figure D5-13 shows the stage 2 address translation, for an input address size of between 43 and 46 bits. This means the lookup can start at either level 1 or level 2.

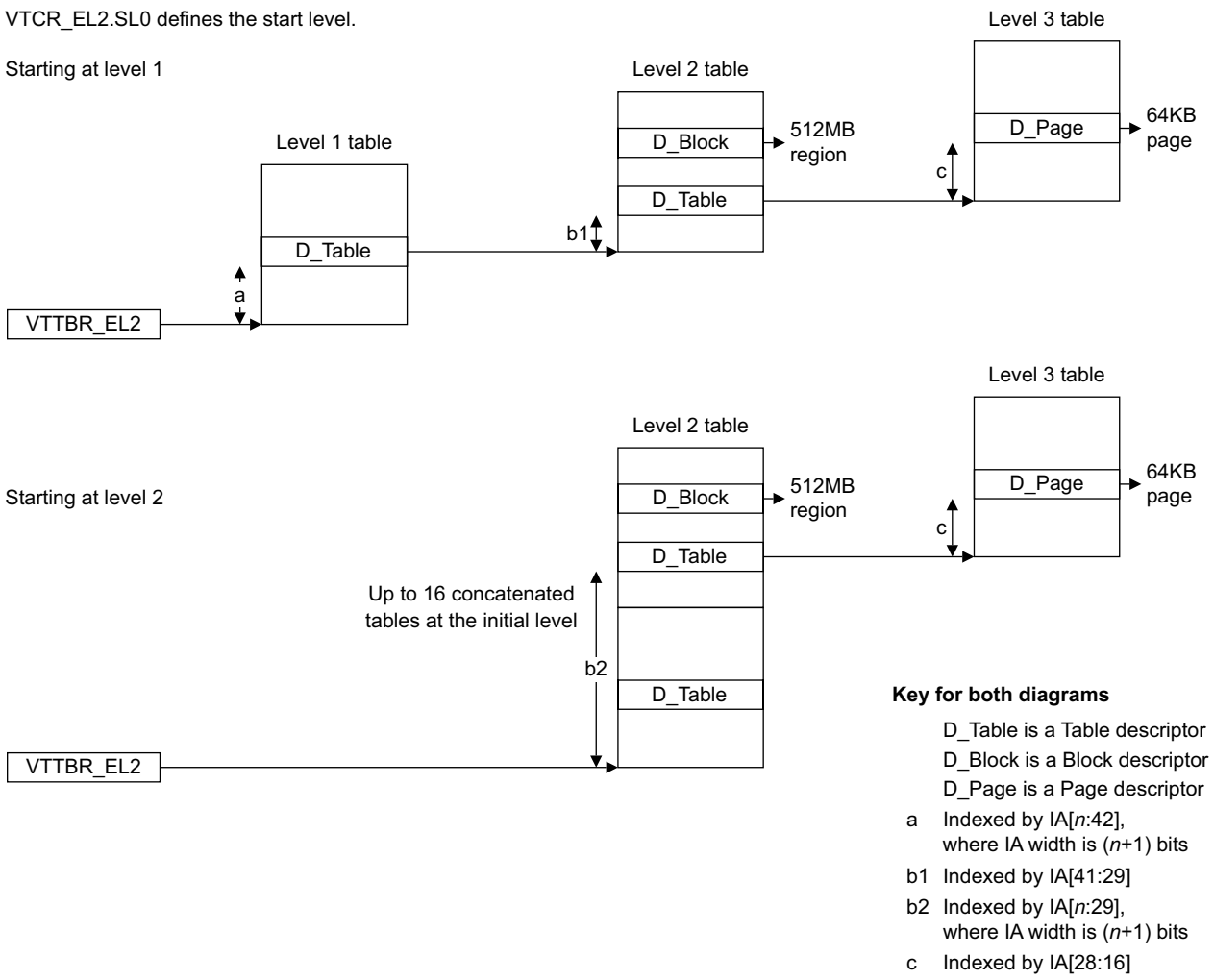


Figure D5-13 General view of VMSAv8-64 stage 2 address translation, 64KB granule

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D5.2.7 The VMSAv8-64 translation table format

This section provides the full description of the VMSAv8-64 translation table format, its use for address translations that are controlled by an Exception level using AArch64. For these translation regimes:

For a stage 1 translation that can support two VA ranges

- For the lower VA range, that uses TTBR0_ELx:
  - The TCR_ELx.{SH0, ORGN0, IRGN0} fields define memory region attributes for the translation table walks.
  - The TCR_ELx.TG0 field defines the Translation granule size.
- For the upper VA range, that uses TTBR1_ELx:
  - The TCR_ELx.{SH1, ORGN1, IRGN1} fields define memory region attributes for the translation table walks.
  - The TCR_ELx.TG1 field defines the Translation granule size.
- Each of TTBR0_ELx and TTBR1_ELx contains an ASID field, and the TCR_ELx.A1 field selects which of these specifies the ASID to use.

For a stage 1 translation that supports only one VA range

The translation table walks use TTBR0_ELx, and:

- The TCR_ELx.{SH0, ORGN0, IRGN0} fields define memory region attributes for the translation table walks.
- The TCR_ELx.TG0 field defines the Translation granule size.

For a stage 2 translation

The Non-secure translation table walks use VTTBR_EL2, and:

- The VTCR_EL2.{SH0, ORGN0, IRGN0} fields define memory region attributes for the translation table walks.
- The VTCR_EL2.TG0 field defines the Translation granule size.

The Secure translation table walks use VSTTBR_EL2, and:

- The VSTCR_EL2.SH0 and VTCR_EL2.{ORGN0, IRGN0} fields define memory region attributes for the translation table walks.
- The VSTCR_EL2.TG0 field defines the Translation granule size.

For the VMSAv8-64 translation table format, *Overview of the VMSAv8-64 address translation stages* on page D5-2564 summarizes the lookup levels, and *Descriptor encodings, Armv8 level 0, level 1, and level 2 formats* on page D5-2596 describes the translation table entries.

The following subsections describe the use of this translation table format:

- *Translation granule size and associated block and page sizes* on page D5-2576.
- *Selection between TTBR0_ELx and TTBR1_ELx when two VA ranges are supported* on page D5-2578.
- *Use of concatenated translation tables for the initial stage 2 lookup* on page D5-2580.
- *Possible errors in programming the translation table registers* on page D5-2581.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Translation granule size and associated block and page sizes

Table D5-18 shows the supported granule sizes, block sizes and page sizes, for the different granule sizes. For completeness, this table includes information for AArch32 state. In the table, the OA bit ranges are the OA bits that the translation table descriptor specifies to address the block or page of memory, in an implementation that supports a 48-bit OA range.

Table D5-18 Translation granule sizes, with block and page sizes, and output address ranges

Granule size	Table level	Block size and OA bit range	Page size and OA bit range
4KB	Zero	-	-
	One	1GB, OA[47:30]	-
	Two	2MB, OA[47:21]	-
	Three	-	4KB, OA[47:12]
16KB	Zero	-	-
	One	-	-
	Two	32MB, OA[47:25]	-
	Three	-	16KB, OA[47:14]
64KB	One	4TB, OA[51:42][a]	-
	Two	512MB, OA[47:29]	-
	Three	-	64KB, OA[47:16]

a. Only available when ARMv8.2-LPA is implemented, see *Extending addressing above 48 bits* on page D5-2552.

Bit[1] of a translation table descriptor identifies whether the descriptor is a block descriptor, and:

- The 4KB granule size supports block descriptors only in level 1 and level 2 translation tables.
- The 16KB granule size supports block descriptors only in level 2 translation tables.
- The 64KB granule size supports block descriptors in level 2 translation tables, and in level 1 translation tables when ARMv8.2-LPA is implemented and the implementation supports 52 bits of physical address. If the implementation does not support 52 bits of physical address, then encoding a block descriptor in a level 1 translation table generates a level 1 Translation fault.

If bit[1] of a descriptor is 0 in a translation table that does not support block descriptors then a translation table walk that accesses that descriptor generates a Translation fault.

For translations managed from AArch64 state, the following tables expand the information for each granule size, showing for an access to a single translation table at each lookup level:

- The maximum IA size, and the address bits that are resolved for that maximum size.
- The maximum OA range resolved by the translation table descriptors at this level, and the corresponding memory region size.
- The maximum size of the translation table. This is the size required for the maximum IA size.

Table D5-19 on page D5-2577 shows this information for the 4KB translation granule size, Table D5-20 on page D5-2577 shows this information for the 16KB translation granule size, and Table D5-21 on page D5-2577 shows this information for the 64KB translation granule size.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D5-19 Properties of the address lookup levels, 4KB granule size

Level	Maximum input address		Maximum output address		Number of entries	Block entries supported?
	Size	Address range	Address range	Size of addressed region[a]		
Zero	256TB	Address[47:39]	Address[47:39]	512GB	Up to 512	No
One	512GB	Address[38:30]	Address[47:30]	1GB	Up to 512	Yes
Two	1GB	Address[29:21]	Address[47:21]	2MB	Up to 512	Yes
Three	2MB	Address[20:12]	Address[47:12]	4KB	512	Page only

a. That is, the size of the region either addressed by descriptors at this level or to be resolved at this and the subsequent levels of lookup.

Table D5-20 Properties of the address lookup levels, 16KB granule size

Level	Maximum input address		Maximum output address		Number of entries	Block entries supported?
	Size	Address range	Address range	Size of addressed region[a]		
Zero	256TB	Address[47]	Address[47]	128TB	2[b]	No
One	128TB	Address[46:36]	Address[47:36]	64GB	Up to 2048	No
Two	64GB	Address[35:25]	Address[47:25]	32MB	Up to 2048	Yes
Three	32MB	Address[24:14]	Address[47:14]	16KB	2048	Page only

a. That is, the size of the region either addressed by descriptors at this level or to be resolved at this and the subsequent levels of lookup.
b. The translation table size is less than the maximum for this granule size, and therefore the number of entries is reduced.

Table D5-21 Properties of the address lookup levels, 64KB granule size

Level	Maximum input address		Maximum output address		Number of entries	Block entries supported?
	Size	Address range	Address range	Size of addressed region[a]		
One	4PB	Address[51:42]	Address[51:42]	4TB	Up to 1024[b]	Yes
One	256TB	Address[47:42]	Address[47:42]	4TB	Up to 64[b]	No
Two	4TB	Address[41:29]	Address[47:29]	512MB	Up to 8192	Yes
Three	512MB	Address[28:16]	Address[47:16]	64KB	8192	Page only

a. That is, the size of the region either addressed by descriptors at this level or to be resolved at this and the subsequent levels of lookup.
b. The translation table size is less than the maximum for this granule size, and therefore the number of entries is reduced.

For the initial lookup level:

- If the IA range specified by the TCR_ELx.T*x*SZ field is smaller than the maximum size shown in these table then this reduces the number of addresses in the table and therefore reduces the table size. The smaller translation table is aligned to its table size.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- For stage 2 translations, multiple translation tables can be concatenated to extend the maximum IA size beyond that shown in these tables. For more information see the stage 2 translation overviews in *Overview of the VMSAv8-64 address translation stages* on page D5-2564 and *Use of concatenated translation tables for the initial stage 2 lookup* on page D5-2580.

If a supplied input address is larger than the configured input address size, a Translation fault is generated.

Note

Larger translation granule sizes typically requires fewer levels of translation tables to translate a particular size of VA.

For the TCR_ELx programming requirements for the initial lookup, see *Overview of the VMSAv8-64 address translation stages* on page D5-2564.

Selection between TTBR0_ELx and TTBR1_ELx when two VA ranges are supported

Every translation table walk starts by accessing the translation table addressed by the TTBR_ELx for the stage 1 translation for the required translation regime.

For a stage 1 translation that can support two VA ranges, Figure D5-14 shows this VA range split when using 48-bit VAs, and:

- TTBR0_ELx points to the initial translation table for the lower VA range, that starts at address `0x0000000000000000`,
- TTBR1_ELx points to the initial translation table for the upper VA range, that runs up to address `0xFFFFFFFFFFFFFFFF`.

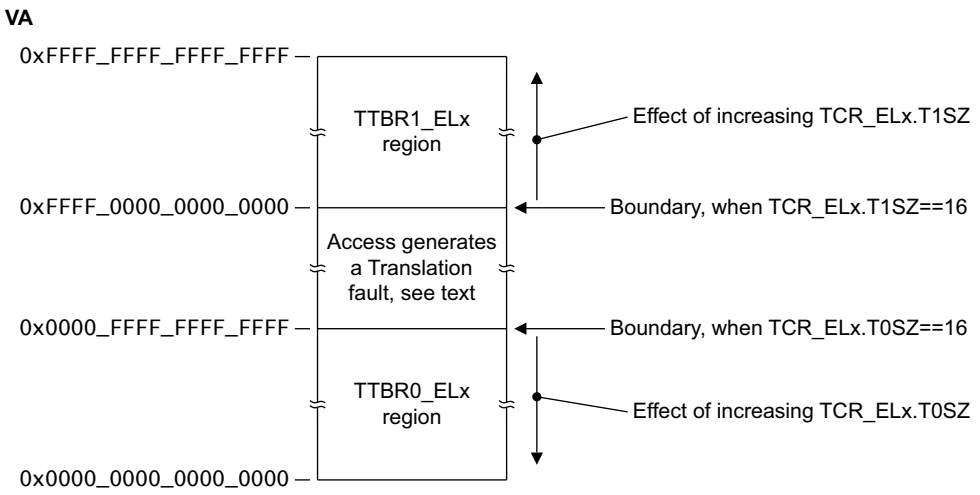


Figure D5-14 AArch64 TTBR*n* boundaries and VA ranges for 48-bit VAs

As Figure D5-14 shows, for 48-bit VAs:

- The address range translated using TTBR0_ELx is `0x0000000000000000` to `0x0000FFFFFFFFFFFF`.
- The address range translated using TTBR1_ELx is `0xFFFF000000000000` to `0xFFFFFFFFFFFFFFFF`.

In an implementation that includes ARMv8.2-LVA and is using the 64KB translation granule, for 52-bit VAs:

- The address range translated using TTBR0_ELx is `0x0000000000000000` to `0x000FFFFFFFFFFFFF`.
- The address range translated using TTBR1_ELx is `0xFFF0000000000000` to `0xFFFFFFFFFFFFFFFF`.

Which TTBR_ELx is used depends only on the VA presented for translation. The most significant bits of the VA must all be the same value and:

- If the most significant bits of the VA are zero, then TTBR0_ELx is used.
- If the most significant bits of the VA are one, then TTBR1_ELx is used.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

However, it is configurable whether VA[63:56] are considered when determining which TTBR_ELx is used, that is:

- In an implementation that includes ARMv8.2-LVA and is using the 64KB translation granule, whether the determination depends on VA[63:52] or on VA[55:52].
- Otherwise, whether the determination depends on VA[63:48] or on VA[55:48].

For more information about whether VA[63:56] are considered for this determination see *Address tagging in AArch64 state* on page D5-2534.

Note

The handling of the Contiguous bit can mean that the boundary between the translation regions defined by the TCR_ELx.TnSZ values and the region for which an access generates a Translation fault is wider than shown in Figure D5-14 on page D5-2578. That is, if the descriptor for an access to the region shown as generating a fault has the Contiguous bit set to 1, the access might not generate a fault. *Possible errors in programming the translation table registers* on page D5-2581 describes this possibility.

Example D5-3 shows a typical application of this VA split.

Example D5-3 Example use of the split VA range, and the TTBR0_ELx and TTBR1_ELx controls

An example of using the split VA range is:

TTBR0_ELx Used for process-specific addresses.

Each process maintains a separate level 1 translation table. On a context switch:

- TTBR0_ELx is updated to point to the level 1 translation table for the new context
- TCR_ELx is updated if this change changes the size of the translation table
- CONTEXTIDR_ELx is updated.

TTBR1_ELx Used for operating system and I/O addresses, that do not change on a context switch.

For each VA subrange, the input address size is $2^{(64-TnSZ)}$, where T*n*SZ is one of TCR_ELx.{T0SZ, T1SZ},

This means the two VA subranges are:

Lower VA subrange `0x0000_0000_0000_0000` to $(2^{(64-T0SZ)} - 1)$.

Upper VA subrange $(2^{64} - 2^{(64-T1SZ)})$ to `0xFFFF_FFFF_FFFF_FFFF`.

If ARMv8.5-E0PD is implemented, the TCR_ELx.E0PD1 field can prevent unprivileged access to the addresses translated by TTBR1_ELx.

For the situation where the minimum T*n*SZ value is 16, corresponding to a maximum input address range of 48 bits, Example D5-4 shows the two VA subranges when T0SZ and T1SZ are both set to this minimum value.

Example D5-4 Maximum VA ranges when a stage of translation supports two ranges

The maximum VA subranges correspond to T0SZ and T1SZ each having a minimum value of 16. In this case the subranges are:

Lower VA subrange `0x0000_0000_0000_0000` to `0x0000_FFFF_FFFF_FFFF`.

Upper VA subrange `0xFFFF_0000_0000_0000` to `0xFFFF_FFFF_FFFF_FFFF`.

Figure D5-14 on page D5-2578 indicates the effect of varying the T*n*SZ values.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

As described in *Overview of the VMSAv8-64 address translation stages* on page D5-2564, the T*n*SZ values also determine the initial lookup level for the translation.

Use of concatenated translation tables for the initial stage 2 lookup

Overview of the VMSAv8-64 address translation stages on page D5-2564 introduced the ability to concatenate translation tables for the initial stage 2 translation lookup. This section gives more information about that concatenation.

If a stage 2 translation would require 16 entries or fewer in its top-level translation table, that stage of translation can, instead, be configured so that:

- It requires the corresponding number of concatenated translation tables at the next translation level, aligned to the size of the block of concatenated translation tables.
- The stage 2 translation starts at that next translation level.

When using the 16KB translation granule, if a 48-bit input address size is required for the stage 2 translations, lookup must start with two concatenated translation tables at level 1.

The use of concatenated translation tables requires the software that is defining the translation to:

- Define the concatenated translation tables with the required overall alignment.
- Program VTTBR_EL2 or VSTTBR_EL2 to hold the address of the first of the concatenated translation tables.
- Program VTCR_EL2 or VSTCR_EL2 to indicate the required input address range and initial lookup level.

Note

The use of concatenated translation tables avoids the overhead of an additional level of translation.

Concatenating additional translation tables at the initial level of look up resolves additional address bits at that level. To resolve *n* additional address bits requires 2^n concatenated translation tables. Example D5-5 shows how, for level 1 lookups using the 4KB translation granule, translation tables can be concatenated to resolve three additional address bits.

Example D5-5 Adding three bits of address resolution at level 1 lookup, using the 4KB granule

When using the 4KB translation granule, a level 1 lookup with a single translation table resolves address bits[38:30]. To add three more address bits requires 2^3 translation tables, that is, eight translation tables. This means:

- The total size of the concatenated translation tables is 8×4KB=32KB.
- This block of concatenated translation tables must be aligned to 32KB.
- The address range resolved at this lookup level is A[41:30], of which:
  - Bits A[41:39] select the 4KB translation table.
  - Bits A[38:30] index a descriptor within that translation table.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

As an example of the concatenation of translation tables at the initial lookup level, when using the 4KB translation granule, Table D5-22 shows the possible uses of concatenated translation tables to permit lookup to start at level 1 rather than at level 0. For completeness, the table starts with the case where the required IPA range means lookup starts at level 1 with a single translation table at that level.

Table D5-22 Possible uses of concatenated translation tables for level 1 lookup, 4KB granule

Configured stage 2 IA size		Lookup starts at level 0	Lookup starts at level 1	
IPA range	**Size**	**Required level 0 entries**	**Number of concatenated tables**	**Required alignment[a]**
IPA[38:0]	2^{36} bytes	-	1	4KB
IPA[39:0]	2^{37} bytes	2	2	8KB
IPA[40:0]	2^{38} bytes	4	4	16KB
IPA[41:0]	2^{39} bytes	8	8	32KB
IPA[42:0]	2^{40} bytes	16	16	64KB

a. Required alignment of the set of concatenated level 2 tables.

Note

Because concatenation is permitted only for a stage 2 translation, the input addresses in the table are IPAs.

Overview of the VMSAv8-64 address translation stages on page D5-2564 identifies all of the possible uses of concatenation. In all cases, the block of concatenated translation tables must be aligned to the block size.

Possible errors in programming the translation table registers

This subsection describes possible errors in programming the translation table registers.

Misprogramming the VTCR_EL2.{T0SZ, SL0} and VSTCR_EL2.{T0SZ, SL0} fields

For a stage 2 translation, the programming of the VTCR_EL2 or VSTCR_EL2 T0SZ and SL0 fields must be consistent. If these fields are not consistent, or if SL0 is programmed to a reserved value, any translation table walk that uses stage 2 translation generates a stage 2 level 0 Translation fault. For more information see *Overview of the VMSAv8-64 address translation stages* on page D5-2564.

Misprogramming of the Contiguous bit

For more information about the Contiguous bit, and the range of translation table entries that must have the bit set to 1 to mark the entries as contiguous, see *The Contiguous bit* on page D5-2633.

If one or more of the following errors is made in programming the translation tables, the TLB might contain overlapping entries:

- One or more of the contiguous translation table entries does not have the Contiguous bit set to 1.
- One or more of the contiguous translation table entries holds an output address that is not consistent with all of the entries pointing to the same aligned contiguous address range.
- The attributes and permissions of the contiguous entries are not all the same.

Such misprogramming of the translation tables means the output address, memory permissions, or attributes for a lookup might be corrupted, and might be equal to values that are not consistent with any of the programmed translation table values.

In some implementations, such misprogramming might also give rise to a TLB Conflict abort.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The architecture guarantees that misprogramming of the Contiguous bit cannot provide a mechanism for any of the following to occur:

- Software executing at EL1 or EL0 accessing regions of physical memory that are not accessible by programming the translation tables, from EL1, with arbitrary chosen values that do not misprogram the Contiguous bit.
- Software executing at EL1 or EL0 accessing regions of physical memory with attributes or permissions that are not possible by programming the translation tables, from EL1, with arbitrary chosen values that do not misprogram the Contiguous bit.
- Software executing in Non-secure state accessing Secure physical memory.

Note

Hardware implementations must ensure that use of the Contiguous bit cannot provide a mechanism for avoiding output address range checking. This might occur if a Contiguous bit block size of 0.5 GB or 1 GB is used in a system with the output address size configured to 4 GB. The architecture permits the implemented mechanism for preventing any avoidance of output address range checking to suppress the use of the Contiguous bit for such entries in such a system.

Where the Contiguous bit is used to mark a set of blocks as contiguous, if the address range translated by a set of blocks marked as contiguous is larger than the size of the input address supported at a stage of translation used to translate that address at that stage of translation, as defined by the TCR_ELx.TxSZ field, then this is a programming error. An implementation is permitted, but not required, to:

- Treat such a block within a contiguous set of blocks as causing a Translation fault, even though the block is valid, and the address accessed within that block is within the size of the input address supported at a stage of translation, as defined by the TCR_ELx.TxSZ field.
- Treat such a block within a contiguous set of blocks as not causing a Translation fault, even though the address accessed within that block is outside the size of the input address supported at a stage of translation, as defined by the TCR_ELx.TxSZ field, provided that both of the following apply:
  - The block is valid.
  - At least one address within the block, or contiguous set of blocks, is within the size of the input address supported at a stage of translation.

When ARMv8.2-LVA is implemented, level 1 block descriptors for the 64KB granule do not support the Contiguous bit, and that field is RES0.

D5.2.8 The algorithm for finding the translation table descriptors

This subsection gives the algorithms for finding the translation table descriptor that corresponds to a given IA, for each required level of lookup. The algorithms encode the descriptions of address translation given earlier in this section. The algorithm details depend on the translation granule size for the stage of address translation, see:

- *Finding the translation table descriptor when using the 4KB translation granule* on page D5-2583.
- *Finding the translation table descriptor when using the 16KB translation granule* on page D5-2584.
- *Finding the translation table descriptor when using the 64KB translation granule* on page D5-2585.

Each subsection uses the following terms:

BaseAddr The base address for the level of lookup, as defined by:
- For the initial lookup level, the value of the appropriate TTBR_ELx.BADDR field.
- Otherwise, the translation table address returned by the previous level of lookup.

PAMax The supported PA width, in bits.

IA The supplied IA for this stage of translation.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T*n*SZ The translation table size for this stage of translation:

For EL1&0 stage 1
TCR_EL1.T0SZ or TCR_EL1.T1SZ, as appropriate.

For Non-secure EL1&0 stage 2
VTCR_EL2.T0SZ.

For Secure EL1&0 stage 2
VSTCR_EL2.T0SZ

For EL2 stage 1 TCR_EL2.T0SZ.

For EL2&0 stage 1 TCR_EL2.T0SZ or TCR_EL2.T1SZ, as appropriate.

For EL3 stage 1 TCR_EL3.T0SZ.

SL0 The initial lookup level for this stage of translation:

For Non-secure EL1&0 stage 2 translation
VTCR_EL2.SL0

For Secure EL1&0 stage 2 translation
VSTCR_EL2.SL0

These subsections show only architecturally-valid programming of the TCR_ELx. See also *Possible errors in programming the translation table registers* on page D5-2581.

Finding the translation table descriptor when using the 4KB translation granule

Table D5-23 shows the translation table descriptor address, for each level of lookup, when using the 4KB translation granule. See the start of *The algorithm for finding the translation table descriptors* on page D5-2582 for more information about terms used in the table.

Table D5-23 Translation table entry addresses when using the 4KB translation granule

Lookup level	Entry address and conditions		General conditions
	Stage 1 translation	Stage 2 translation	
Zero	BaseAddr[PAMax-1:*x*]:IA[*y*:39]:`0b000` if[a] $z \le$ T*n*SZ ≤ 24 then $x = (28 - $ T*n*SZ)	BaseAddr[PAMax-1:*x*]:IA[*y*:39]:`0b000` if SL0[b] == 2 then if[a] $z \le$ T0SZ ≤ 24 then $x = (28 - $ T0SZ)	$y = (x + 35)$ $z = 16$
One	BaseAddr[PAMax-1:*x*]:IA[*y*:30]:`0b000` if[a] $25 \le$ T*n*SZ ≤ 33 then $x = (37 - $ T*n*SZ) else[c] $x = 12$	BaseAddr[PAMax-1:*x*]:IA[*y*:30]:`0b000` if SL0[b] == 1 then if[a] $21 \le$ T0SZ ≤ 33 then $x = (37 - $ T0SZ) elsif SL0[b, c] == 2 then $x = 12$	$y = (x + 26)$
Two	BaseAddr[PAMax-1:*x*]:IA[*y*:21]:`0b000` if[a] $34 \le$ T*n*SZ ≤ 39 then $x = (46 - $ T*n*SZ) else[c] $x = 12$	BaseAddr[PAMax-1:*x*]:IA[*y*:21]:`0b000` if SL0[b] == 0 then if[a] $30 \le$ T0SZ ≤ 39 then $x = (46 - $ T0SZ) elsif SL0[b, c] ==1 or 2 then $x = 12$	$y = (x + 17)$
Three	BaseAddr[PAMax-1:x]:IA[y:12]:`0b000` if[a] $43 \le$ T*n*SZ ≤ 48 then x=(55-T*n*SZ) else[c] x=12	BaseAddr[PAMax-1:x]:IA[y:12]:`0b000` if SL0[b] == 3 then if[a] $39 \le$ T0SZ ≤ 48 then $x = (55 - $ T0SZ) elsif SL0[b, c] =0, 1, or 2 then x=12	y = (x+ 8)

a. This line indicates the range of permitted values for T*n*SZ, for a lookup that starts at this level, see *Overview of VMSAv8-64 address translation using the 4KB translation granule* on page D5-2564.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

b. SL0 == 0 if the initial lookup is level 2, SL0 == 1 if the initial lookup is level 1, SL0 == 2 if the initial lookup level is level 0, and SL0 == 3 if the initial lookup level is level 3.
c. This is the case where this level of lookup is not the initial level of lookup.

Table D5-7 on page D5-2555 shows how software can determine whether an implementation supports the 4KB granule size.

Finding the translation table descriptor when using the 16KB translation granule

Table D5-24 shows the translation table descriptor address, for each level of lookup, when using the 16KB translation granule. See the start of *The algorithm for finding the translation table descriptors* on page D5-2582 for more information about terms used in the table.

Table D5-24 Translation table entry addresses when using the 16KB translation granule

Lookup level	Entry address and conditions		General conditions
	Stage 1 translation	Stage 2 translation	
Zero	BaseAddr[PAMax-1:4]:IA[47]:`0b000` [a] 16 == T*n*SZ	-	Only applies to stage 1
One	BaseAddr[PAMax-1:*x*]:IA[*y*:36]:`0b000` if[a] 17 ≤ T*n*SZ ≤ 27 then *x* = (31 - T*n*SZ) else[c] *x* =14	BaseAddr[PAMax-1:*x*]:IA[*y*:36]:`0b000` if SL0[b] == 2 then if[a] 16 ≤ T0SZ ≤ 27 then *x* = (31 - T0SZ)	*y* = (*x* + 32)
Two	BaseAddr[PAMax-1:*x*]:IA[*y*:25]:`0b000` if[a] 28 ≤ T*n*SZ ≤ 38 then *x* = (42 - T*n*SZ) else[c] *x* =14	BaseAddr[PAMax-1:*x*]:IA[*y*:25]:`0b000` if SL0[b] == 1 then if[a] 24 ≤ T0SZ ≤ 38 then *x* = (42 - T0SZ) elsif SL0[b, c] == 2 then *x* = 14	*y* = (*x* + 21)
Three	BaseAddr[PAMax-1:x]:IA[y:14]:`0b000` if[a] 39 ≤ T*n*SZ ≤ 48 then x = (53 - T*n*SZ) else[c] x=14	BaseAddr[PAMax-1:*x*]:IA[*y*:14]:`0b000` if SL0[b] == 0 then if[a] 35 ≤ T0SZ ≤ 48 then *x* = (53 - T0SZ) elsif SL0[b, c] ==1 or 2 then *x* = 14	*y* = (*x* + 10)

a. This line indicates the range of permitted values for T*n*SZ, for a lookup that starts at this level, see *Overview of VMSAv8-64 address translation using the 16KB translation granule* on page D5-2567.
b. SL0 == 0 if the initial lookup is level 3, SL0 == 1 if the initial lookup is level 2, and SL0 ==2 if the initial lookup level is level 1.
c. This is the case where this level of lookup is not the initial level of lookup.

Table D5-7 on page D5-2555 shows how software can determine whether an implementation supports the 16KB granule size.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Finding the translation table descriptor when using the 64KB translation granule

Table D5-25 shows the translation table descriptor address, for each level of lookup, when using the 64KB translation granule. See the start of *The algorithm for finding the translation table descriptors* on page D5-2582 for more information about terms used in the table.

Table D5-25 Translation table entry addresses when using the 64KB translation granule

Lookup level	Entry address and conditions		General conditions
	Stage 1 translation	Stage 2 translation	
One	BaseAddr[PAMax-1:*x*]:IA[*y*:42]:`0b000` if[a] z[b] ≤ T*n*SZ ≤ 21 then x = (25 - T*n*SZ)	BaseAddr[PAMax-1:*x*]:IA[*y*:42]:`0b000` if SL0[c] == 2 then if[a] z[b] ≤ T0SZ ≤ 21 then x = (25 - T0SZ)	y = (x + 38) z = 16 or 12[b]
Two	BaseAddr[PAMax-1:*x*]:IA[*y*:29]:`0b000` if[a] 22 ≤ T*n*SZ ≤ 34 then x = (38 - T*n*SZ) else[d] x =16	BaseAddr[PAMax-1:*x*]:IA[*y*:29]:`0b000` if SL0[c] == 1 then if[a] 18 ≤ T0SZ ≤ 34 then x = (38 - T0SZ) elsif SL0[c, d] == 2 then x = 16	y = (x + 25)
Three	BaseAddr[PAMax-1:*x*]:IA[*y*:16]:`0b000` if[a] 35 ≤ T*n*SZ ≤ 47 then x = (51 - T*n*SZ) else[d] x =16	BaseAddr[PAMax-1:*x*]:IA[*y*:16]:`0b000` if SL0[c] == 0 then if[a] 31 ≤ T0SZ ≤ 47 then x = (51 - T0SZ) elsif SL0[c, d] ==1 or 2 then x = 16	y = (x + 12)

a. This line indicates the range of permitted values for T*n*SZ, for a lookup that starts at this level, see *Overview of VMSAv8-64 address translation using the 64KB translation granule* on page D5-2572.

b. If ARMv8.2-LVA is implemented, the value of z is 12, see *Extending addressing above 48 bits* on page D5-2552. Otherwise, the value of z is 16.

c. SL0 == 0 if the initial lookup is level 3, SL0 == 1 if the initial lookup is level 2, and SL0 ==2 if the initial lookup level is at level 1.

d. This is the case where this level of lookup is not the initial level of lookup.

Table D5-7 on page D5-2555 shows how software can determine whether an implementation supports the 64KB granule size.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D5.2.9 The effects of disabling a stage of address translation

The following sections describe the effect on MMU behavior of disabling each stage of translation:

- *Behavior when stage 1 address translation is disabled*.
- *Behavior when stage 2 address translation is disabled* on page D5-2587.
- *Behavior of instruction fetches when all associated stages of translation are disabled* on page D5-2587.

Behavior when stage 1 address translation is disabled

When a stage 1 address translation is disabled, memory accesses that would otherwise be translated by that stage of translation are treated as follows:

EL1 and EL0 accesses if the HCR_EL2.DC bit is set to 1

For the EL1&0, when EL2 is enabled, translation regime, when the value of HCR_EL2.DC is 1, the stage 1 translation assigns the Normal Non-shareable, Inner Write-Back Read-Allocate Write-Allocate, Outer Write-Back Read-Allocate Write-Allocate memory attributes.

Note

This applies for both instruction and data accesses.

All other accesses

For all other accesses, when stage 1 address translation is disabled, the assigned attributes depend on whether the access is a data access or an instruction access, as follows:

Data access

The stage 1 translation assigns the Device-nGnRnE memory type.

Instruction access

The stage 1 translation assigns the Normal memory attribute, with the cacheability and shareability attributes determined by the value of the SCTLR_ELx.I bit for the translation regime, as follows:

When the value of I is 0

The stage 1 translation assigns the Non-cacheable and Outer Shareable attributes.

When the value of I is 1

The stage 1 translation assigns the Cacheable, Inner Write-Through Read-Allocate No Write-Allocate, Outer Write-Through Read-Allocate No Write-Allocate Outer Shareable attribute.

Secure accesses and Non-secure accesses

For accesses from the Non-secure state, the output address is to the Non-secure output address space.

For accesses from the Secure state, the output address is to the Secure output address space.

For this stage of translation, no memory access permission checks are performed. Therefore no MMU faults can be generated for this stage of address translation.

Note

Alignment checking is performed, and therefore Alignment faults can occur.

For every access, the input address of the stage 1 translation is flat-mapped to the output address.

For a EL1 or EL0 access, if EL1&0 stage 2 address translation is enabled, the stage 1 memory attribute assignments and output address can be modified by the stage 2 translation.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When the value of HCR_EL2.DC is 1:

- The SCTLR_EL1.M bit behaves as if it is 0, for all purposes other than reading the value of the bit. This means EL1&0 stage 1 address translation is disabled.
- The HCR_EL2.VM bit behaves as if it is 1, for all purposes other than reading the value of the bit. This means that EL1&0 stage 2 address translation is enabled.

See also *Behavior of instruction fetches when all associated stages of translation are disabled*.

Effect of disabling address translation on maintenance and address translation instruction instructions

Cache maintenance instructions act on the target cache regardless of whether any stages of address translation are disabled, and regardless of the values of the memory attributes. However, if a stage of address translation is disabled, they use the flat address mapping for that translation stage.

TLB invalidate operations act on the target TLB regardless of whether any stage of address translation is disabled.

The value of HCR_EL2.DC affect some address translation instructions, see *Address translation instructions, AT** on page D5-2589.

Behavior when stage 2 address translation is disabled

When stage 2 address translation is disabled:

- The IPA output from the stage 1 translation maps flat to the PA.
- The memory attributes and permissions from the stage 1 translation apply to the PA.

When both stages of address translation are disabled, see also *Behavior of instruction fetches when all associated stages of translation are disabled*.

Secure accesses and Non-secure accesses

For accesses from the Non-secure IPA address space, the output address is to the Non-secure physical address space.

For accesses from the Secure IPA address space, the output address is to the Secure physical address space.

Behavior of instruction fetches when all associated stages of translation are disabled

When EL3 is using AArch64, this section applies to:

- The Secure EL1&0, when EL2 is disabled, translation regime when stage 1 address translation is disabled in that regime.
- The EL3 translation regime when stage 1 address translation is disabled in that regime.
- The Secure EL2, or Secure EL2&0, translation regime when stage 1 address translation is disabled in that regime
- The Non-secure EL1&0, when EL2 is enabled, translation regime, when both stages of address translation are disabled.

Note

- The behaviors in Non-secure state apply regardless of the Execution state that EL3 is using.
- When the value of HCR_EL2.DC is 1, then the behavior of the EL1&0 translation regime is as if stage 1 translation is disabled and stage 2 translation is enabled, as described in *Behavior when stage 1 address translation is disabled* on page D5-2586.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In these cases, when execution is in AArch64 state, a memory location might be accessed as a result of an instruction fetch if either:

- The memory location is in the same block of memory as, or in the next contiguous block of memory to, an instruction that a simple sequential execution of the program either requires to be fetched now or has required to be fetched since the last reset.
- The memory location is the target of a direct branch that a simple sequential execution of the program would have taken since the most recent of:
  - The last reset.
  - The last synchronization of instruction cache maintenance targeting the address of the branch instruction.

In this description, the blocks of memory referred to are of the size of the minimum implemented translation granule and are aligned to that size.

These accesses can be caused by speculative instruction fetches, regardless of whether the prefetched instruction is committed for execution.

Note

To ensure architectural compliance, software must ensure that both of the following apply:

- Instructions that will be executed when all associated stages of address translation are disabled are located in blocks of the address space, of the translation granule size, that contain only memory that is tolerant to speculative accesses.
- Each block of the address space, of the translation granule size, that immediately follows a similar block that holds instructions that will be executed when all associated stages address translation are disabled, contains only memory that is tolerant to speculative accesses.

D5.2.10 Pseudocode description of VMSAv8-64 address translation

The following subsections outline a pseudocode description of the translation table walk:

- *Definitions required for address translation*.
- *Performing the full address translation*.
- *Stage 1 translation* on page D5-2589.
- *Stage 2 translation* on page D5-2589.
- *Translation table walk* on page D5-2589.
- *Support functions* on page D5-2589.

Definitions required for address translation

In pseudocode, the result of a translation table lookup, in either Execution state, is returned in a `TLBRecord` structure.

Memory data type definitions on page D4-2527 includes definitions of the `Permissions` and `AddressDescriptor` parameters.

Performing the full address translation

The function `AArch64.FullTranslate()` performs a full translation table walk. For any translation regime it performs a stage 1 translation for the supplied VA, and for the EL1&0, when EL2 is enabled, translation regime it then performs a stage 2 translation of the returned address.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Stage 1 translation

The function `AArch64.FirstStageTranslate()` performs a stage 1 translation, calling the function `AArch64.TranslationTableWalk()`, described in *Translation table walk*, to perform the required translation table walk. However, if stage 1 translation is disabled, it calls the function `AArch64.TranslateAddressS1Off()` to set the memory attributes.

Stage 2 translation

In the EL1&0, when EL2 is enabled, translation regime, a descriptor address returned by stage 1 lookup is in the IPA address space, and must be mapped to a PA by a stage 2 translation. Function `AArch64.SecondStageWalk()` performs this translation, by calling the `AArch64.SecondStageTranslate()` function. When called from `AArch64.SecondStageWalk()`, the `AArch64.SecondStageTranslate()` function performs a second stage translation, from IPA to PA, of the supplied address, including checking that the access has read permission at the second stage. If the access does not have second stage read permission it generates a second stage Permission fault on the first stage translation table walk. The second stage translation might hit in a TLB, or might involve a translation table walk, which will use the algorithm described in this section.

Translation table walk

The function `AArch64.FirstStageTranslate()` returns the result, in the form of a `TLBRecord`, of a translation table walk made for a memory access from an Exception level that is using AArch64.

Support functions

In the translation table walk functions, the `WalkAttrDecode()` function determines the attributes for a translation table lookup.

The function `AArch64.S1AttrDecode()` decodes the attributes from a stage 1 translation table lookup.

The function `AArch64.CheckPermission()` checks the access permissions returned by a stage 1 translation table lookup, see *Access permission checking* on page D4-2529.

The function `AArch64.CheckS2Permission()` checks the access permissions returned by a stage 2 translation table lookup.

The function `AddrTop()` returns the bit number of the most significant valid bit of a VA in the current translation regime. If EL1 is using AArch64 and EL0 is using AArch32 then an address from EL0 is zero-extended to 64 bits.

D5.2.11 Address translation instructions

Each of the Armv8 instruction sets provides instructions that return the result of translating an input address, su.pplied as an argument to the instruction, using a specified translation stage or regime.

The available instructions only perform translations that are accessible from the Security state and Exception level at which the instruction is executed. That is:

- No instruction executed in Non-secure state can return the result of a Secure address translation stage.
- No instruction can return the result of an address translation stage that is controlled by an Exception level that is higher than the Exception level at which the instruction is executed.

*Address translation instructions, AT** summarizes the A64 address translation instructions.

See also *A64 System instructions for address translation* on page C5-536.

If ARMv8.5-MemTag is implemented and enabled the behavior of AT* instructions in AArch64 state are modified. For more information see *Virtual address translation* on page D6-2693.

Address translation instructions, AT*

The A64 assembly language syntax for address translation instructions is:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
AT <operation>, <Xt>
```

Where:

`<operation>` Is one of `S1E1R`, `S1E1RP`, `S1E1W`, `S1E1WP`, `S1E0R`, `S1E0W`, `S12E1R`, `S12E1W`, `S12E0R`, `S12E0W`, `S1E2R`, `S1E2W`, `S1E3R`, or `S1E3W`.

`<operation>` has a structure of `<stages><level><read|write><pan>`, where:

`<stages>` Is one of:

- `S1` Stage 1 translation.
- `S12` Stage 1 translation followed by stage 2 translation.

`<level>` Describes the Exception Level that the translation applies to. Is one of:

- `E0` EL0.
- `E1` EL1.
- `E2` EL2.
- `E3` EL3.

If `<level>` is higher than the current Exception Level, the instruction is UNDEFINED.

`<read|write>`

Is one of:

- `R` Read.
- `W` Write.

`<pan>`

Only available when ARMv8.2-ATS1E1 is implemented. Optional, but if present:

- `P` Determines action based on value of PSTATE.PAN.

Only permitted for `<stages>`=`S1` and `<level>`=`E1`.

`<Xt>` The address to be translated. No alignment restrictions apply for the address.

If EL2 is not implemented, the `AT S1E2R` and `AT S1E2W` instructions are UNDEFINED.

Note

If EL2 is not implemented but EL3 is implemented, the `AT S12E**` instructions are not UNDEFINED, but behave the same way as the equivalent `AT S1E**` instructions. This is consistent with the behavior if EL2 is implemented but stage 2 translation is disabled.

In each case, the address being translated is held in the 64-bit address argument register, `Xt`. If the address translation instruction uses a translation regime that is using AArch32, meaning it requires a VA of only 32 bits, then VA[63:32] is RES0.

If the address translation is successful, the resulting PA is returned in PAR_EL1.PA, and PAR_EL1.F is set to 0 to indicate that the translation was successful. Otherwise, see *Synchronous faults generated by address translation instructions* on page D5-2591.

Note

The architecture provides a single PAR, PAR_EL1, that is used regardless of:

- The Exception level at which the instruction was executed.
- The Exception level that controls the stage or stages of translation used by the instruction.

For all of these instructions, the current context information determines which entries in TLB caching structures are used, and how the translation table walk is performed. However, it is IMPLEMENTATION DEFINED whether the Address translation instructions return the values held in a TLB or the result of a translation table walk. Therefore, Arm recommends that these instructions are not used at a time when the TLB entries might be different from the underlying translation tables held in memory.

If EL3 is implemented, then for instructions that apply to the EL1 or EL0 Exception level, SCR_EL3.NS determines the translation regime to which the instruction applies, as follows:

SCR_EL3.NS == 0 Secure EL1&0 translation regime.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

SCR_EL3.NS == 1 Non-secure EL1&0 translation regime.

All relevant context information used for the translation depends on this determination.

When EL1&0 stage 1 address translation is disabled, any AT S1E0*, AT S1E1*, AT S12E0*, or AT S12E1* address translation instruction that accesses the Non-secure state translation reflects the effect of the HCR_EL2.DC bit as described in *Behavior when stage 1 address translation is disabled* on page D5-2586.

If Secure EL2 translation regime is disabled, executing `AT S1E2R` or `AT S1E2W` at EL3 with SCR_EL3.NS == 0 is UNDEFINED.

Note

`AT S12E**` instructions at EL3 with SCR_EL3.NS == 0 are not UNDEFINED but behave the same way as the equivalent `AT S1E**` instructions.

Synchronous faults generated by address translation instructions

The address translation instructions use the translation mechanism, and that mechanism can generate the following synchronous faults:

- Translation fault.
- Access flag fault.
- Permission fault.
- Domain fault, when translating using the AArch32 translation systems.
- Address size fault.
- TLB conflict fault.
- Synchronous External aborts during a translation table walk.

In addition:

- If the address translation instruction requires two stages of translation then these faults could arise from either stage 1 or stage 2.
- For a stage 1 translation for the EL1&0 translation regime, the fault might be generated on the stage 2 translation of an address accessed as part of the stage 1 translation table walk, see *Stage 2 fault on a stage 1 translation table walk* on page D5-2658.

Except as described in this section, these faults are not taken as an exception for the address translation instructions, but instead the PAR_EL1.FST field holds the Fault status information. In these cases the PAR_EL1.PA field does not hold the output address of the translation.

The exceptions to this reporting the fault in PAR_EL1 are:

- Synchronous External aborts during a translation table walk are taken as a Data Abort exception.

  For an address translation instruction executed at a particular Exception level, if the synchronous External abort is generated on a stage 1 translation table walk, the Data Abort exception is taken to the Exception level to which a synchronous External abort on a stage 1 translation table walk for a memory access from that Exception level would be taken.

  If the synchronous External abort is generated on a stage 2 translation table walk then:

  — If the address translation instruction was executed at EL3, the synchronous Data Abort exception is taken to EL3.

  — If the address translation instruction was executed at EL2 or EL1, the Data Abort exception is taken to the Exception level to which a synchronous External abort on a stage 2 translation table walk for a memory access from that Exception level would be taken.

  In any case where the address translation instruction causes a synchronous Data Abort exception to be taken:

  — The PAR_EL1 is UNKNOWN.

  — The ESR_ELx of the target Exception Level of the exception indicates that the fault was due to a translation table walk for a cache maintenance instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— The FAR_ELx of the target Exception Level holds the VA for the translation request.

- For the `AT S1E0*` and `AT S1E1*` instructions executed from EL1, if there is a synchronous stage 2 fault on a memory access made as part of the translation table walk then:
  — If the fault is a synchronous External abort on a stage 2 translation table and SCR_EL3,EA is 1, then a synchronous External abort on a stage 2 translation table walk is taken to EL3.
  — Otherwise the fault is taken as an exception to EL2.

  If the exception is taken to EL2 the following apply:
  — PAR_EL1 is UNKNOWN.
  — ESR_EL2 indicates that the fault occurred on a translation table walk, and that the operation that faulted was a cache maintenance instruction.
  — HPFAR_EL2 holds the IPA that faulted.
  — FAR_EL2 holds the VA that the executing software supplied to the address translation instruction.

  This fault can occur for any of the following reasons:
  — Stage 2 Translation fault.
  — Stage 2 Access fault.
  — Stage 2 Permission fault.
  — Stage 2 Address size fault.
  — Synchronous External abort on a stage 2 translation table walk.

Synchronization requirements of the address translation instructions

Where an instruction results in an update to a System register, as is the case with the `AT *` address translation instructions, explicit synchronization must be performed before the result is guaranteed to be visible to subsequent direct reads of the PAR_EL1.

Note

This is consistent with the AArch32 requirement, where the VA to PA translation instructions are executed as writes to the (`coproc==0b1111`) System register encoding space, and the effect of those writes to other registers require explicit synchronization before the result is guaranteed to be visible to subsequent instructions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D5.3 VMSAv8-64 translation table format descriptors

In general, a descriptor is one of:

- An invalid or fault entry.
- A table entry, that points to the next-level translation table.
- A block entry, that defines the memory properties for the access.
- A reserved format.

Bit[1] of the descriptor indicates the descriptor type, and bit[0] indicates whether the descriptor is valid.

The following sections describe the Armv8 translation table descriptor formats:

- *VMSAv8-64 translation table level 0, level 1, and level 2 descriptor formats*.
- *Armv8 translation table level 3 descriptor formats* on page D5-2597.

Memory attribute fields in the VMSAv8-64 translation table format descriptors on page D5-2599 then gives more information about the descriptor attribute fields, and *Control of Secure or Non-secure memory access* on page D5-2605 describe how the NS and NSTable together control whether a memory access from Secure state accesses the Secure memory map or the Non-secure memory map.

D5.3.1 VMSAv8-64 translation table level 0, level 1, and level 2 descriptor formats

In the VMSAv8-64 translation table format, the difference in the formats of the level 0, level 1 and level 2 descriptors is:

- Whether a Block descriptor is permitted.
- If a Block descriptor is permitted, the size of the memory region described by that entry.
- The maximum OA size, depending on whether ARMv8.2-LPA is implemented.

These differences depend on the translation granule, as follows:

4KB granule Level 0 translation tables do not support Block descriptors.

A block descriptor:

- In a level 1 table describes the mapping of the associated 1GB input address range.
- In a level 2 table describes the mapping of the associated 2MB input address range.

The maximum OA size of a lookup is 48 bits.

16KB granule Level 0 and level 1 translation tables do not support Block descriptors.

A Block descriptor in a level 2 table describes the mapping of the associated 32MB input address range.

The maximum OA size of a lookup is 48 bits.

64KB granule Level 0 lookup is not supported.

Other properties depend on whether ARMv8.2-LPA is implemented:

If ARMv8.2-LPA is implemented

A block descriptor:

- In a level 1 table describes the mapping of the associated 4TB input address range.
- In a level 2 table describes the mapping of the associated 512MB input address range.

The maximum OA size of a lookup is 48 bits.

If ARMv8.2-LPA is not implemented

Level 1 translation tables do not support Block descriptors.

A Block descriptor in a level 2 table describes the mapping of the associated 512MB input address range.

The maximum OA size of a lookup is 48 bits.

When a lookup returns a Table descriptor, the OA is the next-level table address.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Figure D5-15 shows the Armv8 level 0, level 1, and level 2 descriptor formats that provide 48-bit OAs:

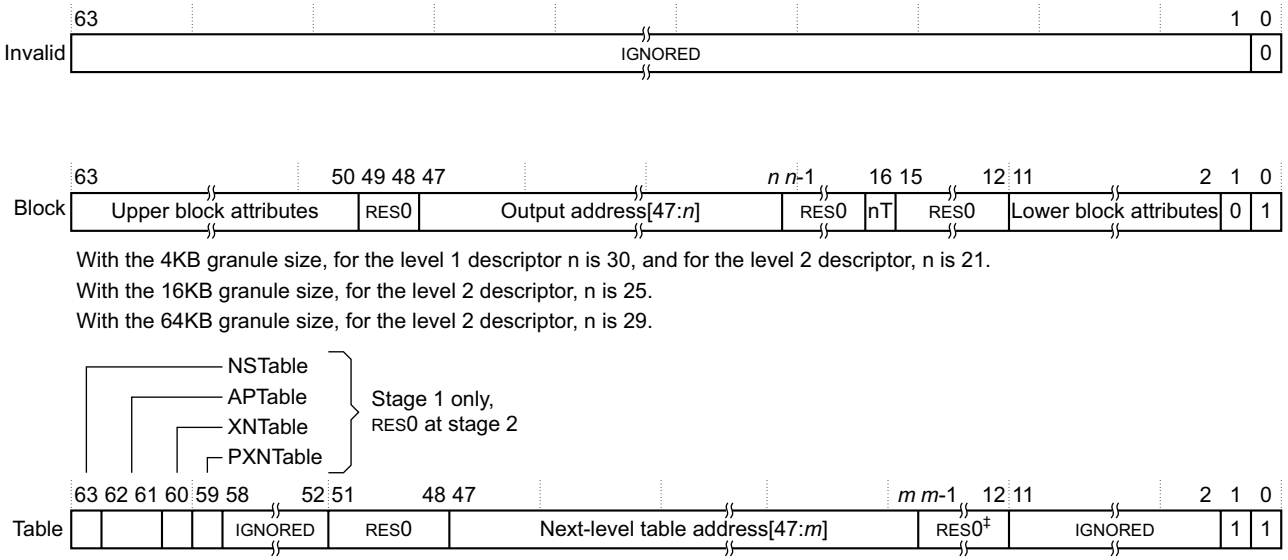
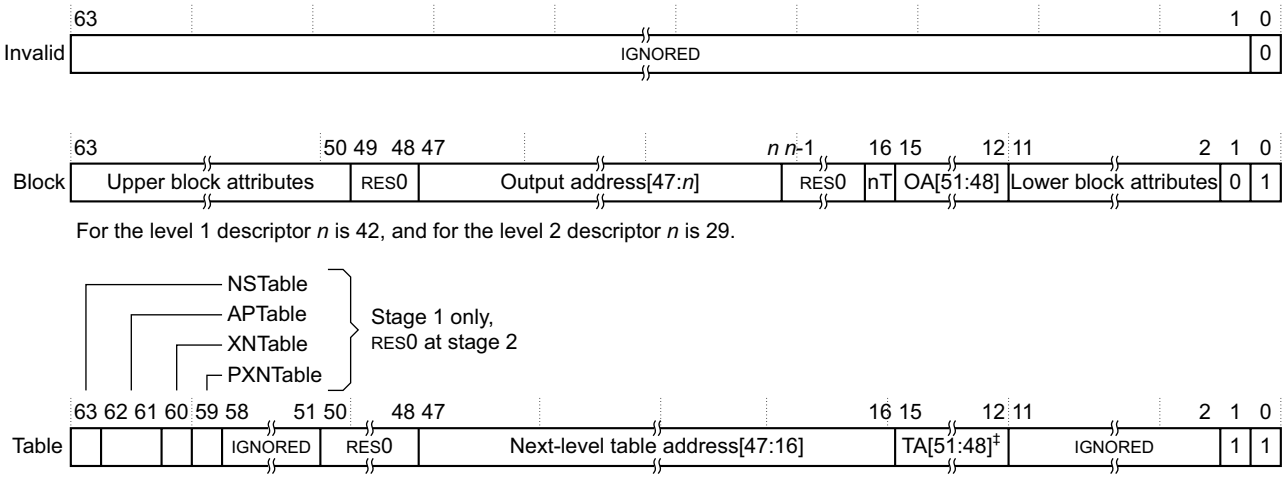


Figure D5-15 VMSAv8-64 level 0, level 1 and level 2 descriptor formats with 48-bit OAs

In an implementation that includes ARMv8.2-LPA, when the 64KB granule is used, the Block and Table descriptors are redefined as Figure D5-16 on page D5-2595 shows:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta



For the level 1 descriptor *n* is 42, and for the level 2 descriptor *n* is 29.

A level 1 Table descriptor returns the address of the level 2 table.
A level 2 Table descriptor returns the address of the level 3 table.

‡ TA[51:48] indicates bits[51:48] of the next-level table address.

Figure D5-16 VMSAv8-64 level 1 and level 2 descriptor formats, 64KB granule with ARMv8.2-LPA

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

The effects on the Non-secure EL1 descriptors when ARMv8.1-HPD is enabled and HCR_EL2.{NV, NV1} ={1,1} are detailed in *Effect of HCR_EL2.{NV, NV1}* on page D5-2644.

Descriptor encodings, Armv8 level 0, level 1, and level 2 formats

Descriptor bit[0] identifies whether the descriptor is valid, and is 1 for a valid descriptor. If a lookup returns an invalid descriptor, the associated input address is unmapped, and any attempt to access it generates a Translation fault.

Descriptor bit[1] identifies the descriptor type, and is encoded as:

0, Block The descriptor gives the base address of a block of memory, and the attributes for that memory region.

1, Table The descriptor gives the address of the next level of translation table, and for a stage 1 translation, some attributes for that translation.

The other fields in the valid descriptors are:

Block descriptor

Gives the base address and attributes of a block of memory, as follows:

4KB translation granule

- For a level 1 Block descriptor, bits[47:30] are bits[47:30] of the output address. This output address specifies a 1GB block of memory.
- For a level 2 descriptor, bits[47:21] are bits[47:21] of the output address.This output address specifies a 2MB block of memory.

16KB translation granule

For a level 2 Block descriptor, bits[47:25] are bits[47:25] of the output address.This output address specifies a 32MB block of memory.

64KB translation granule

For a level 1 Block descriptor:

- If ARMv8.2-LPA is implemented, bits[15:12] are bits[51:48] of the output address and bits[47:42] are bits [47:42] of the output address. This output address specifies a 4TB block of memory.
- If ARMv8.2-LPA is not implemented, there is no level 1 Block descriptor.

For a level 2 Block descriptor:

- If ARMv8.2-LPA is implemented, bits[15:12] are bits[51:48] of the output address, and bits[47:29] are bits [47:29] of the output address.This output address specifies a 512MB block of memory.
- If ARMv8.2-LPA is not implemented, bits[47:29] are bits[47:29] of the output address.This output address specifies a 512MB block of memory.

In Armv8.0, bits[63:52, 11:2] provide attributes for the target memory block. When ARMv8.1-TTHM is implemented, bits[63:51, 11:2] provide the attributes for the target memory block. For more information, see *Memory attribute fields in the VMSAv8-64 translation table format descriptors* on page D5-2599.

Note

- In Armv8.0, the position and contents of bits[63:52, 11:2] are identical to bits[63:52, 11:2] in the Page descriptors.
- When ARMv8.1-TTHM is implemented, the position and contents of bits[63:51, 11:2] are identical to bits[63:51, 11:2] in the Page descriptors.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- When ARMv8.2-TTPBHA is implemented, hardware can use bits[62:59] of the Block descriptors for IMPLEMENTATION DEFINED purposes, see *Memory attribute fields in the VMSAv8-64 translation table format descriptors* on page D5-2599.

Table descriptor

Gives the translation table address for the next-level lookup, as follows:

4KB translation granule

- Bits[47:12] are bits[47:12] of the address of the required next-level table, which is:
  - For a level 0 Table descriptor, the address of a level 1 table.
  - For a level 1 Table descriptor, the address of a level 2 table.
  - For a level 2 Table descriptor, the address of a level 3 table.
- Bits[11:0] of the table address are zero.

16KB translation granule

- Bits[47:14] are bits[47:14] of the address of the required next-level table, which is:
  - For a level 0 Table descriptor, the address of a level 1 table.
  - For a level 1 Table descriptor, the address of a level 2 table.
  - For a level 2 Table descriptor, the address of a level 3 table.
- Bits[13:0] of the table address are zero.

64KB translation granule

- Bits[47:16] are bits[47:16] of the address of the required next-level table, which is:
  - For a level 1 Table descriptor, the address of a level 2 table.
  - For a level 2 Table descriptor, the address of a level 3 table.

  When ARMv8.2-LPA is implemented, bits[15:12] are bits[51:48] of the required next-level table.
- Bits[15:0] of the table address are zero.

For a stage 1 translation only, bits[63:59] provide attributes for the next-level lookup, see *Memory attribute fields in the VMSAv8-64 translation table format descriptors* on page D5-2599.

If the translation table defines either the Secure or Non-secure EL1&0, when EL2 is enabled, stage 1 translations, then the output address in the descriptor is the IPA of the target block or table. Otherwise, it is the PA of the target block or table.

D5.3.2 Armv8 translation table level 3 descriptor formats

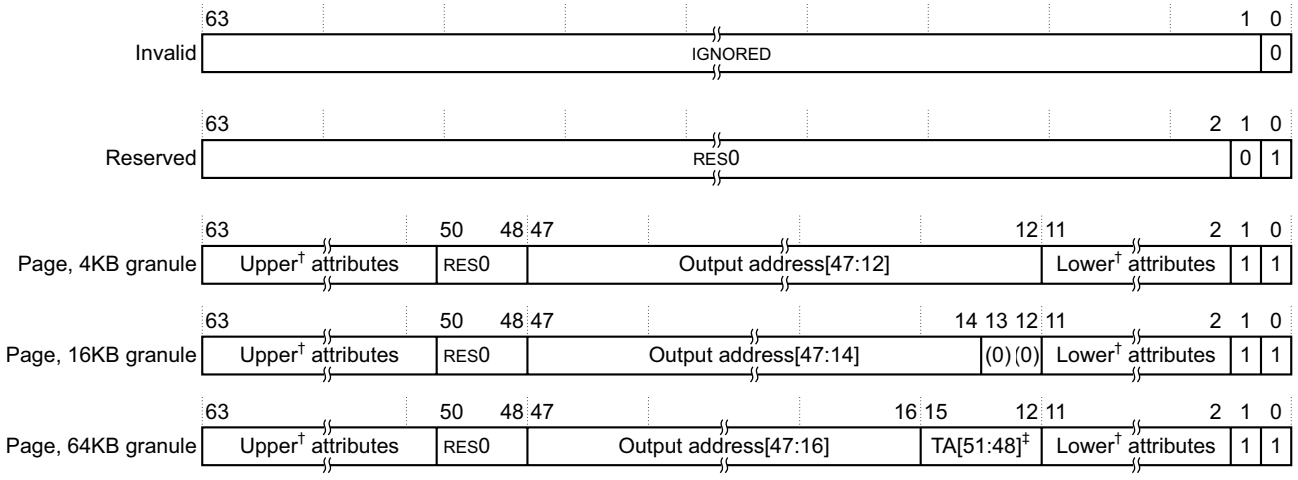
For the 4KB granule size, each entry in a level 3 table describes the mapping of the associated 4KB input address range.

For the 16KB granule size, each entry in a level 3 table describes the mapping of the associated 16KB input address range.

For the 64KB granule size, each entry in a level 3 table describes the mapping of the associated 64KB input address range.

Figure D5-17 on page D5-2598 shows the Armv8 level 3 descriptor formats.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta



† Upper page attributes and Lower page attributes
‡ TA[51:48] indicates bits[51:48] of the page address.

Figure D5-17 VMSAv8-64 level 3 descriptor format

Descriptor bit[0] identifies whether the descriptor is valid, and is 1 for a valid descriptor. If a lookup returns an invalid descriptor, the associated input address is unmapped, and any attempt to access it generates a Translation fault.

Descriptor bit[1] identifies the descriptor type, and is encoded as:

0, Reserved, invalid

Behaves identically to encodings with bit[0] set to 0.

This encoding must not be used in level 3 translation tables.

1, Page Gives the address and attributes of a 4KB, 16KB, or 64KB page of memory.

At this level, the only valid format is the Page descriptor. The other fields in the Page descriptor are:

Page descriptor

Gives the output address of a page of memory, as follows:

4KB translation granule

Bits[47:12] are bits[47:12] of the output address for a page of memory.

16KB translation granule

Bits[47:14] are bits[47:14] of the output address for a page of memory.

64KB translation granule

If ARMv8.2-LPA is implemented, bits[15:12] are bits[51:48] and bits[47:16] are bits[47:16] of the output address for a page of memory.

If ARMv8.2-LPA is not implemented, bits[47:16] are bits[47:16] of the output address for a page of memory.

Bits[63:52, 11:2] provide attributes for the target memory page, see *Memory attribute fields in the VMSAv8-64 translation table format descriptors* on page D5-2599.

Note

- In Armv8.0, the position and contents of bits[63:52, 11:2] are identical to bits[63:52, 11:2] in the level 0, level 1, and level 2 block descriptors.
- When ARMv8.1-TTHM is implemented, the position and contents of bits[63:51, 11:2] are identical to bits[63:51, 11:2] in the level 0, level 1, and level 2 block descriptors.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- When ARMv8.2-TTPBHA is implemented, hardware can use bits[62:59] of the Page descriptors for IMPLEMENTATION DEFINED purposes, see *Memory attribute fields in the VMSAv8-64 translation table format descriptors*.

For either the Secure or Non-secure EL1&0, when EL2 is enabled, stage 1 translations, the output address in the descriptor is the IPA of the target page. Otherwise, it is the PA of the target page.

D5.3.3 Memory attribute fields in the VMSAv8-64 translation table format descriptors

Memory region attributes on page D5-2628 describes the region attribute fields. The following subsections summarize the descriptor attributes as follows:

Table descriptor

Table descriptors for stage 2 translations do not include any attribute field. For a summary of the attribute fields in a stage 1 table descriptor, that define the attributes for the next lookup level, see *Next-level attributes in stage 1 VMSAv8-64 Table descriptors*.

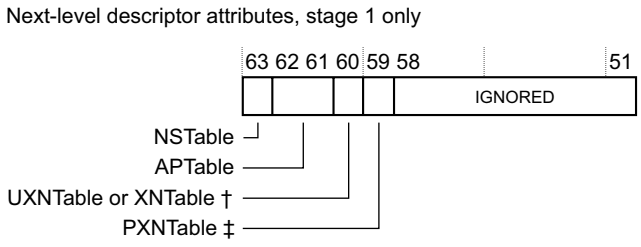
Block and page descriptors

These descriptors define memory attributes for the target block or page of memory. Stage 1 and stage 2 translations have some differences in these attributes, see:

- *Attribute fields in stage 1 VMSAv8-64 Block and Page descriptors* on page D5-2601
- *Attribute fields in stage 2 VMSAv8-64 Block and Page descriptors* on page D5-2603.

Next-level attributes in stage 1 VMSAv8-64 Table descriptors

In a Table descriptor for a stage 1 translation, bits[63:59] of the descriptor define the attributes for the next-level translation table access, and bits[58:51] are IGNORED:



† UXNTable for a translation regime that can apply to execution at EL0, otherwise XNTable.
‡ RES0 for a translation regime that cannot apply to execution at EL0.

These attributes are:

NSTable, bit[63]

For memory accesses from Secure state, specifies the Security state for subsequent levels of lookup, see *Hierarchical control of Secure or Non-secure memory accesses* on page D5-2605.

For memory accesses from Non-secure state, including all accesses in the EL2 or EL2&0 translation regime, this bit is RES0 and is ignored by the PE.

APTable, bits[62:61]

Access permissions limit for subsequent levels of lookup, see *Hierarchical control of data access permissions* on page D5-2611.

APTable[0] is RES0:

- In the EL2 translation regime.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

In an implementation that includes ARMv8.1-VHE, when the value of HCR_EL2.E2H is 1 the translation regime for memory accesses from EL2 is the EL2&0 translation regime. APTable[0] can be valid (not RES0) in the EL2&0 translation regime.

- In the EL3 translation regime.

From Armv8.1, when ARMv8.1-HPD is implemented, this field can be disabled. When the value of TCR_ELx.HPD{0} or TCR_ELx.HPD1 is 1:

- The value of the corresponding APTable field is IGNORED by hardware, allowing the field to be used by software.
- The behavior of the system is as if the value of the corresponding APTable field is 0.

Note

From Armv8.3, if EL2 is enabled in the current Security state, in the EL1 translation regime, when the value of HCR_EL2.{NV, NV1} == {1, 1}, bit[61] is treated as 0 regardless of the actual value, see *Additional behaviors when HCR_EL2.NV == 1 and HCR_EL2.NV1 == 1* on page D5-2645.

UXNTable or XNTable, bit[60]

XN limit for subsequent levels of lookup, see *Hierarchical control of instruction fetching* on page D5-2616.

The naming of this field depends on whether stage 1 of the translation regime can support two VA ranges:

Stage 1 can support two VA ranges

This field is UXNTable, and determines whether execution at EL0 of instructions fetched from the region identified at a lower level of lookup permitted.

Note

PXNTable is the equivalent control of execution at a higher Exception level.

Stage 1 supports only one VA range

This field is XNTable.

From Armv8.1, when ARMv8.1-HPD is implemented, this field can be disabled. When the value of TCR_ELx.HPD{0} or TCR_ELx.HPD1 is 1:

- The value of the corresponding UXNTable field is IGNORED by hardware, allowing the field to be used by software.
- The behavior of the system is as if the value of the corresponding UXNTable field is 0.

Note

From Armv8.3, if EL2 is enabled in the current Security state, in the EL1 translation regime, when the value of HCR_EL2.{NV, NV1} == {1, 1}, bit[60] holds PXNTable, see *Additional behaviors when HCR_EL2.NV == 1 and HCR_EL2.NV1 == 1* on page D5-2645.

PXNTable, bit[59]

PXN limit for subsequent levels of lookup, see *Hierarchical control of instruction fetching* on page D5-2616.

This field is valid only for a stage 1 translation that can support two VA ranges. It is RES0 for stage 1 translations that can support only one VA range.

From Armv8.1, when ARMv8.1-HPD is implemented, this field can be disabled. When the value of TCR_ELx.HPD{0} or TCR_EL1.HPD1 is 1:

- The value of the corresponding PXNTable field is IGNORED by hardware, allowing the field to be used by software.
- The behavior of the system is as if the value of the corresponding PXNTable field is 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

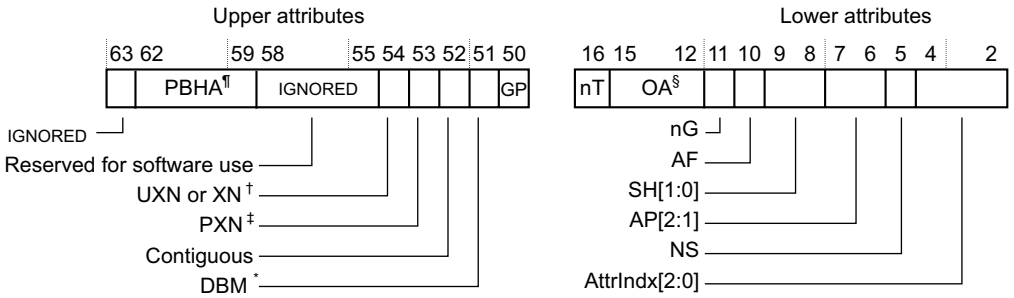
From Armv8.3, if EL2 is enabled in the current Security state, in the EL1&0 translation regime, when the value of HCR_EL2.{NV, NV1} == {1, 1}, bit[59] is RES0, see *Additional behaviors when HCR_EL2.NV == 1 and HCR_EL2.NV1 == 1* on page D5-2645.

The definition of IGNORED means the architecture guarantees that the PE makes no use of the field, see *IGNORED* on page Glossary-8115. For more information about these fields see *Other fields in the VMSAv8-64 translation table format descriptors* on page D5-2633.

Attribute fields in stage 1 VMSAv8-64 Block and Page descriptors

In Block and Page descriptors, the memory attributes are split into an upper block and a lower block, as shown for a stage 1
translation:

Attribute fields for VMSAv8-64 stage 1 Block and Page descriptors



¶ IGNORED if ARMv8.2-TTPBHA is not implemented.
† UXN for a translation regime that can apply to execution at EL0, otherwise XN.
‡ RES0 for a translation regime that cannot apply to execution at EL0.
* RES0 if ARMv8.1-TTHM is not implemented.
§ RES0 if ARMv8.2-LPA is not implemented.

For a stage 1 descriptor, the attributes are:

PBHA, bits[62:59]

Page-based Hardware Attributes bits.

These bits are IGNORED when ARMv8.2-TTPBHA is not implemented.

When ARMv8.2-TTPBHA is implemented, each TCR_ELx has a control bit for each PBHA bit in the translation tables that it controls. When the value of that control bit is 1, and the value of the corresponding Hierarchical permission disables bit, TCR_ELx.HPD{n} is 1, hardware can use that PBHA bit for IMPLEMENTATION DEFINED purposes.

The TCR_ELx control bits for this feature are:

For a translation regime that supports only a single VA range

HWU0nn Controls whether Block or Page descriptor bit[nn] can be used by hardware.

These controls apply only when the value of TCR_ELx.HPD0 is 1.

For a translation regime that can support two VA ranges

HWU0nn For the translation tables indicated by TTBR0_ELx, controls whether Block or Page descriptor bit[nn] can be used by hardware.

These controls apply only when the value of TCR_ELx.HPD0 is 1.

HWU1nn For the translation tables indicated by TTBR1_ELx, controls whether Block or Page descriptor bit[nn] can be used by hardware.

These controls apply only when the value of TCR_ELx.HPD1 is 1.

If ARMv8.2-TTPBHA is not implemented, then the TCR_ELx control bits are RAZ/WI.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

XN or UXN, bit[54]

The Execute-never or Unprivileged execute-never field, see *Access permissions for instruction execution* on page D5-2612.

Note

From Armv8.3, in the Non-secure EL1 translation regime, when the value of HCR_EL2.{NV, NV1} == {1, 1}, bit[54] holds PXN, see *Additional behaviors when HCR_EL2.NV == 1 and HCR_EL2.NV1 == 1* on page D5-2645.

PXN, bit[53] The Privileged execute-never field, see *Access permissions for instruction execution* on page D5-2612.

This field is valid only when stage 1 of the translation regime can support two VA ranges. It is RES0 when stage 1 can support only one VA range.

Note

From Armv8.3, in the Non-secure EL1 translation regime, when the value of HCR_EL2.{NV, NV1} == {1, 1}, bit[53] is RES0, see *Additional behaviors when HCR_EL2.NV == 1 and HCR_EL2.NV1 == 1* on page D5-2645.

Contiguous, bit[52]

A hint bit indicating that the translation table entry is one of a contiguous set or entries, that might be cached in a single TLB entry, see *The Contiguous bit* on page D5-2633.

DBM, bit[51] Dirty Bit Modifier, see *The dirty state* on page D5-2618.

GP, bit[50] Guarded Page.

If ARMv8.5-BTI is implemented, this field is present in stage 1 block and page translation table entries. Otherwise, this field is RES0.

This field is RES0 in stage 2 block and page translation table entries.

nT, bit[16] Block translation entry, see *Block translation entry* on page D5-2618.

If ARMv8.4-TTRem is implemented, this field is present in stage 1 block translation table entries. Otherwise, this field is RES0.

nG, bit[11] The not global bit. If a lookup using this descriptor is cached in a TLB, determines whether the TLB entry applies to all ASID values, or only to the current ASID value. See *Global and process-specific translation table entries* on page D5-2665.

This field is valid only when stage 1 of the translation regime can support two VA ranges. It is RES0 when stage 1 can support only one VA range.

AF, bit[10] The Access flag, see *The Access flag* on page D5-2618.

SH, bits[9:8] Shareability field, see *Memory region attributes* on page D5-2628.

AP[2:1], bits[7:6]

Data Access Permissions bits, see *Memory access control* on page D5-2606.

Note

The Armv8 translation table descriptor format defines AP[2:1] as the Access Permissions bits, and does not define an AP[0] bit.

AP[1] is valid only for stage 1 of a translation regime that can support two VA ranges. It is RES1 when stage 1 translations can support only one VA range.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

From Armv8.3, in the Non-secure EL1 translation regime, when the value of HCR_EL2.{NV, NV1} == {1, 1}, bit[6] is treated as 0 regardless of its actual value, see *Additional behaviors when HCR_EL2.NV == 1 and HCR_EL2.NV1 == 1* on page D5-2645.

NS, bit[5]

Non-secure bit. For memory accesses from Secure state, specifies whether the output address is in the Secure or Non-secure address map, see *Control of Secure or Non-secure memory access* on page D5-2605.

For memory accesses from Non-secure state this bit is RES0 and is ignored by the PE.

AttrIndx[2:0], bits[4:2]

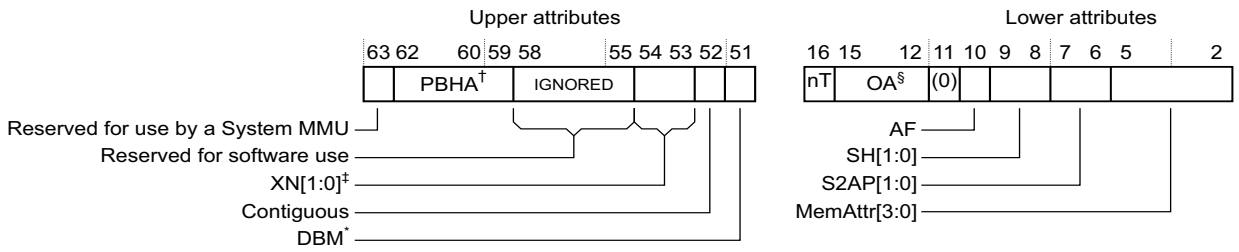
Stage 1 memory attributes index field, for the MAIR_ELx, see *Stage 1 memory region type and Cacheability attributes* on page D5-2628.

The definition of IGNORED means the architecture guarantees that the PE makes no use of the field, see *IGNORED* on page Glossary-8115. For more information about these fields see *Other fields in the VMSAv8-64 translation table format descriptors* on page D5-2633.

Attribute fields in stage 2 VMSAv8-64 Block and Page descriptors

In Block and Page descriptors, the memory attributes are split into an upper block and a lower block, as shown for a stage 2 translation:

Attribute fields for VMSAv8-64 stage 2 Block and Page descriptors when ARMv8.4-S2FWB is implemented



‡ Bit[53] is RES0 if ARMv8.2-TTS2UXN is not implemented.
† Bits [62:60] are IGNORED and reserved for use by System MMU if ARMv8.2-TTPBHA is not implemented.
Bits [59] is IGNORED if ARMv8.2-TTPBHA is not implemented.
* RES0 if ARMv8.1-TTHM is not implemented.
§ RES0 if ARMv8.2-LPA is not implemented.

For a stage 2 descriptor, the attributes are:

PBHA[3:1], bits[62:60]

Page-based hardware attributes bits.

These bits are IGNORED and reserved for System MMU use when ARMv8.2-TTPBHA is not implemented.

When ARMv8.2-TTPBHA is implemented, VTCR_EL2 has a control bit for each PBHA bit in the EL1&0 stage 2 translation tables. When the value of that control bit is 1, hardware can use the corresponding PBHA bit for IMPLEMENTATION DEFINED purposes. When the value of the control bit is 0, the corresponding PBHA bit is IGNORED and reserved for System MMU use.

PBHA[0], bit[59]

Page-based hardware attributes bit.

This bit is IGNORED when ARMv8.2-TTPBHA is not implemented.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When ARMv8.2-TTPBHA is implemented, VTCR_EL2 has a control bit for this bit in the EL1&0 stage 2 translation tables. When the value of that control bit is 1, hardware can use this bit for IMPLEMENTATION DEFINED purposes. When the value of the control bit is 0, this bit is IGNORED.

XN[1:0], bits[54:53]

The Execute-never field, see *Access permissions for instruction execution* on page D5-2612.

If ARMv8.2-TTS2UXN is not implemented, bit[53] is RES0.

Contiguous, bit[52]

A hint bit indicating that the translation table entry is one of a contiguous set or entries, that might be cached in a single TLB entry, see *The Contiguous bit* on page D5-2633.

DBM, bit[51] Dirty Bit Modifier, see *The dirty state* on page D5-2618.

nT, bit[16] Block translation entry, see *Block translation entry* on page D5-2618.

If ARMv8.4-TTRem is implemented, this field is present in stage 2 block translation table entries. Otherwise, this field is RES0.

AF, bit[10] The Access flag, see *The Access flag* on page D5-2618.

SH, bits[9:8] Shareability field, see *The stage 2 memory region attributes, EL1&0 translation regime* on page D5-2630.

S2AP, bits[7:6]

Stage 2 data Access Permissions bits, see *The S2AP data access permissions, Secure or Non-secure EL1&0, when EL2 is enabled, translation regime* on page D5-2611.

Note

In the original VMSAv7-32 Long-descriptor attribute definition, this field was called HAP[2:1], for consistency with the AP[2:1] field in the stage 1 descriptors and despite there being no HAP[0] bit. Armv8 renames the field for greater clarity.

MemAttr, bits[5:2]

Stage 2 memory attributes, see *The stage 2 memory region attributes, EL1&0 translation regime* on page D5-2630.

The definition of IGNORED means the architecture guarantees that the PE makes no use of the field, see *IGNORED* on page Glossary-8115. For more information about these fields see *Other fields in the VMSAv8-64 translation table format descriptors* on page D5-2633.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D5.3.4 Control of Secure or Non-secure memory access

As this section describes, the NS bit in the translation table entries:

- For accesses from Secure state, if the translation table entry was held in secure memory, determines whether the access is to Secure or Non-secure memory.
- Is ignored by:
  - Accesses from Non-secure state.
  - Accesses from Secure state if the translation table entry was held in Non-secure memory.

In the VMSAv8-64 translation table format:

- The NS bit relates only to the memory block or page at the output address defined by the descriptor.
- The descriptors also include an NSTable bit, that affects accesses at lower levels of lookup, see *Hierarchical control of Secure or Non-secure memory accesses*.

The NS and NSTable bits are valid only for memory accesses from Secure state described by translation table descriptors that are fetched from Secure memory, and:

- In the translation table descriptors in a Non-secure translation table, the NS and NSTable bits are SBZ.
- Memory accesses from Non-secure state, including all accesses from EL2, ignore the values of these bits.

In the Secure translation regimes, for translation table descriptors that are fetched from Secure memory, the NS bit in a descriptor indicates whether the descriptor refers to the Secure or the Non-secure address map, as follows:

NS == 0 Access the Secure PA space.

NS == 1 Access the Non-secure PA space.

For Non-secure translation regimes, and for translation table descriptors fetched from Non-secure memory, the corresponding bit is RES0 and is ignored by the PE. The access is made to Non-secure memory, regardless of the value of the bit.

Hierarchical control of Secure or Non-secure memory accesses

For VMSAv8-64 table descriptors for stage 1 translations, the descriptor includes an NSTable bit, that indicates whether the table identified in the descriptor is in Secure or Non-secure memory. For accesses from Secure state, the meaning of the NSTable bit is:

NSTable == 0 The defined table address is in the Secure PA space. In the descriptors in that translation table, NS bits and NSTable bits have their defined meanings.

NSTable == 1 The defined table address is in the Non-secure PA space. Because this table is fetched from the Non-secure address space, the NS and NSTable bits in the descriptors in this table must be ignored. This means that, for this table:

- The value of the NS bit in any block or page descriptor is ignored. The block or page address refers to Non-secure memory.
- The value of the NSTable bit in any table descriptor is ignored, and the table address refers to Non-secure memory. When this table is accessed, the NS bit in any block or page descriptor is ignored, and all descriptors in the table refer to Non-secure memory.

In addition, an entry fetched in Secure state is treated as non-global if it is read from the Non-secure IPA space memory. That is, these entries must be treated as if nG==1, regardless of the value of the nG bit. For more information about the nG bit, see *Global and process-specific translation table entries* on page D5-2665.

The effect of NSTable applies to later entries in the translation table walk, and so its effects can be held in one or more TLB entries. Therefore a change to NSTable requires coarse-grained invalidation of the TLB to ensure that the effect of the change is visible to subsequent memory transactions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D5.4 Memory access control

The access control fields in the translation table descriptors determine whether the PE, in its current state, is permitted to perform the required access to the output address given in the translation table descriptor. If a translation stage does not permit the access then an MMU fault is generated for that translation stage, and no memory access is performed.

The following sections describe the memory access controls:

- *About access permissions*.
- *About PSTATE.PAN* on page D5-2607.
- *About PSTATE.UAO* on page D5-2608.
- *About PSTATE.BTYPE* on page D5-2608.
- *Data access permission controls* on page D5-2609.
- *Access permissions for instruction execution* on page D5-2612.
- *The Access flag* on page D5-2618.
- *The dirty state* on page D5-2618.
- *Software management of the Access flag* on page D5-2618.
- *Hardware management of the Access flag and dirty state* on page D5-2619.
- *Ordering of hardware updates to the translation tables* on page D5-2625.
- *Restriction on memory types for hardware updates on translation tables* on page D5-2626.
- *Use of the Contiguous bit with hardware updates of the translation table entries* on page D5-2626.

Note

This section describes the access controls for each of the translation regimes, and for each stage of translation in the EL1&0, when EL2 is enabled, translation regime.

A translation applies to memory accesses from either:

- Only a single Exception level, for example the EL3 translation regime.
- EL0 and one higher Exception level, for example the EL1&0 translation regime.

In addition to an output address, a translation table entry that refers to a page or region of memory includes fields that define properties of the target memory region. These fields can be classified as address map control, access control, and region attribute fields. *Control of Secure or Non-secure memory access* on page D5-2605 describes the address map control, and *Memory region attributes* on page D5-2628 describes the other fields.

D5.4.1 About access permissions

The translation table descriptors include fields that define access permissions for data accesses and for instruction fetches. This section introduces those fields. In addition:

- System register controls can prevent execution from writable locations, see *Preventing execution from writable locations* on page D5-2617.
- For the effect of disabling a stage of address translation on the access permissions see *The effects of disabling a stage of address translation* on page D5-2586.
- From Armv8.1, the PSTATE.PAN bit can affect the access permissions for privileged data accesses, see *About PSTATE.PAN* on page D5-2607.
- From Armv8.2, the PSTATE.UAO bit can affect the access permissions for unprivileged instructions, see *About PSTATE.UAO* on page D5-2608.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

This section gives a general description of memory access permissions. In an implementation that includes EL2, software executing at EL1 can see only the access permissions defined by the EL1&0, when EL2 is enabled, stage 1 translations. However, software executing at EL2 can modify these permissions. This modification is invisible to the software executing at EL1 or EL0.

The access permission bits control access to the corresponding memory region. The VMSAv8-64 translation table format:

- In stage 1 translations, uses AP[2:1] to define the data access permissions, see *The AP[2:1] data access permissions, for stage 1 translations* on page D5-2610.

  Note

  The description of the access permission field as AP[2:1] is for consistency with the VMSAv8-32 Short-descriptor translation table format, see *The VMSAv8-32 Short-descriptor translation table format* on page G5-5939. The VMSAv8-64 translation table format does not define an AP[0] bit.

- In stage 2 translations, uses S2AP[1:0] to define the data access permissions, see *The S2AP data access permissions, Secure or Non-secure EL1&0, when EL2 is enabled, translation regime* on page D5-2611.
- Uses the UXN, XN and PXN fields to define access controls for instruction fetches, see *Access permissions for instruction execution* on page D5-2612.

An attempt to perform a memory access that the translation table access permission bits do not permit generates a Permission fault, for the corresponding stage of translation.

Note

In an implementation that includes EL2, each stage of the translation of a memory access made using the Secure or Non-secure EL1&0, when EL2 is enabled, translation regime has its own, independent, permission check.

D5.4.2 About PSTATE.PAN

When the value of PSTATE.PAN is 1, any privileged data access from EL1, or EL2 when HCR_EL2.E2H is 1, to a virtual memory address that is accessible at EL0, generates a Permission fault.

When the value of PSTATE.PAN is 0, the translation system is the same as in Armv8.0.

When ARMv8.1-PAN is implemented, the SPSR_EL1.PAN, SPSR_EL2.PAN, and SPSR_EL3.PAN bits are used for exception returns, and the DSPSR_EL0 register is used for entry to or exit from Debug state.

When ARMv8.1-PAN is implemented, the SCTLR_EL1.SPAN and SCTLR_EL2.SPAN bits are used to control whether the PAN bit is set on an exception to EL1 or EL2.

When HCR_EL2.{E2H, TGE} == {1, 1} SCTLR_EL1.SPAN and SCTLR_EL2.SPAN are ignored.

The PAN bit has no effect on:

- Data Cache instructions other than `DC ZVA`.
- Address translation instructions, other than `ATS1E1RP` and `ATS1E1WP` when ARMv8.2-ATS1E1 is implemented.
- Unprivileged instructions, `LDTR`, `LDTRB`, `LDTRH`, `LDTRSB`, `LDTRSH`, `LDTRSW`, `STTR`, `STTRB`, and `STTRH`, unless HCR_EL2.{E2H, TGE} == {1, 0}.
- Instruction accesses.

The PAN bit has no effect when the first stage of translation is disabled for the current translation regime or when the first stage of translation for the current translation regime does not describe the permissions for access at EL0.

If access is disabled, then the access will give rise to a stage 1 Permission fault.

On an exception that is taken from AArch64 state to AArch64 state, PSTATE.PAN is copied to SPSR_ELx.PAN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

On an exception return from AArch64 state:

- SPSR_ELx.PAN is copied to PSTATE.PAN, when the target Exception level is in AArch64 state.
- SPSR_ELx.PAN is copied to CPSR.PAN, when the target Exception level is in AArch32 state.

Note

- In Non-debug state, in AArch64 state:
  - Software can use an `MSR PAN, #Imm4` or `MSR PAN, Xt` instruction to modify PSTATE.PAN, or an `MRS Xt, PAN` instruction to read PSTATE.PAN.
  - In EL1, when HCR_EL2.{NV, NV1} == {1, 1}, PSTATE.PAN is treated as 0 for all purposes except reading the value of the bit.
- In Debug state, in AArch64 state, a debugger can use the `DRPS` instruction to modify PSTATE.PAN.

D5.4.3 About PSTATE.UAO

When the value of PSTATE.UAO is 1, a Load/Store unprivileged instruction executed at EL1, or executed at EL2 when the *Effective value* of HCR_EL2.{E2H, TGE} is {1, 1} is subject to the memory access permissions that apply to the Exception level at which it is executed, rather than being subject to the EL0 access permissions. This means the Load/Store unprivileged instruction is subject to the same access permissions as the corresponding Load/Store register instruction. See *Load/Store unprivileged* on page C3-213 and *Load/Store register* on page C3-209.

When ARMv8.2-UAO is implemented and PSTATE.UAO is 0, it has no effect on the described behavior of any Load/Store unprivileged instruction.

A corresponding UAO bit is added to SPSR_EL1, SPSR_EL2, and SPSR_EL3 for exception returns, and DSPSR_EL0 for entry to or exit from Debug state.

On an exception that is taken from AArch64 state to AArch64 state, PSTATE.UAO is copied to SPSR_ELx.UAO and then set to 0.

On an exception that is taken from AArch32 state to AArch64 state:

- PSTATE.UAO is set to 0.
- SPSR_ELx.UAO is set to 0.

On an exception return from AArch64 state to AArch64 state, SPSR_ELx.UAO is copied to PSTATE.UAO.

Note

- In Non-debug state, in AArch64 state, software can use an `MSR UAO, #Imm4` or `MSR UAO, Xt` instruction to modify PSTATE.UAO, or an `MRS Xt, UAO` instruction to read PSTATE.UAO.
- In Debug state, in AArch64 state, a debugger can use the `DRPS` instruction to modify PSTATE.UAO.

D5.4.4 About PSTATE.BTYPE

When ARMv8.5-BTI is implemented, on execution of an instruction, the guarded status of the memory region and the register that is accessed by the instruction determines the value that the PSTATE.BTYPE field is set at the end of the execution of the instruction as shown in Table D5-26:

Table D5-26 Values taken by PSTATE.BTYPE on execution of an instruction

Instruction executed	Memory region	Register accessed	PSTATE.BTYPE
`BR`, `BRAA, BRAAZ, BRAB, BRABZ`	Guarded	Any register other than X16 or X17	`0b11`
`BLR`, `BLRAA, BLRAAZ, BLRAB, BLRABZ`	Any	Any register	`0b10`
`BR`, `BRAA, BRAAZ, BRAB, BRABZ`	Guarded	X16 or X17	`0b01`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D5-26 Values taken by PSTATE.BTYPE on execution of an instruction (continued)

Instruction executed	Memory region	Register accessed	PSTATE.BTYPE
BR, BRAA, BRAAZ, BRAB, BRABZ	Non-guarded	Any register	0b01
RET, RETAA, RETAB	Any	Any register	0b00
Any instruction other than BR, BRAA, BRAAZ, BRAB, BRABZ, BLR, BLRAA, BLRAAZ, BLRAB, BLRABZ, RET, RETAA, RETAB	Any	Any register	0b00

The BTI instructions <targets> operand identifies the compatibility of the BTI instruction to different PSTATE.BTYPE values, as seen in Table D5-27.

Table D5-27 Compatibility of BTI instruction to different PSTATE.BTYPE values

<targets>	PSTATE.BTYPE value			
	0b00	0b01	0b10	0b11
(omitted)	N/A	Not compatible	Not compatible	Not compatible
c	N/A	Compatible	Compatible	Not compatible
j	N/A	Compatible	Not compatible	Compatible
jc	N/A	Compatible	Compatible	Compatible

When accessing a guarded memory region and if PSTATE.BTYPE has a value of 0b01, 0b10, or 0b11, then a BTI instruction that is compatible with the current value of PSTATE.BTYPE will not generate a Branch Target exception and will allow execution of subsequent instructions within the memory region.

When accessing a guarded memory region, PACIASP and PACIBSP instructions have an implicit branch target identification instruction. This means that they are a target that is compatible with:

- A PSTATE.BTYPE value of 0b10 or 0b01,
- When the associated SCTLR_ELx.{BT0, BT1, BT} bits are 0, a PSTATE.BTYPE value of 0b11.

Note

- The implicit branch target identification property of PACIASP and PACIBSP is independent of the setting of the SCTLR_ELx.{EnIA, EnIB} bits.
- The Branch Target Identification instructions are NOPs in a non-guarded page.
- There is no direct way of reading or writing to the PSTATE.BTYPE field.

The ESR_ELx.EC code for a Branch Target exception is 0x0D, see *ISS encoding for an exception from Branch Target Identification instruction* on page D13-3003.

D5.4.5 Data access permission controls

The following subsubsections describe the data access permission controls:

- *Preventing EL0 access to halves of the address map* on page D5-2610
- *The AP[2:1] data access permissions, for stage 1 translations* on page D5-2610.
- *The S2AP data access permissions, Secure or Non-secure EL1&0, when EL2 is enabled, translation regime* on page D5-2611.
- *Hierarchical control of data access permissions* on page D5-2611.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Preventing EL0 access to halves of the address map

If ARMv8.5-E0PD is implemented, the TCR_ELx.{E0PD0, E0PD1} fields can prevent unprivileged access to the addresses translated by TTBR0_ELx or TTBR1_ELx. If access is prevented, the fault is reported as a level 0 fault, and should take the same time to generate, whether the address is present in the TLB or not, to mitigate attacks that use fault timing.

The AP[2:1] data access permissions, for stage 1 translations

In VMSAv8-64, for a translation regime that applies to both EL0 and a higher Exception level, the AP[2:1] bits control the stage 1 data access permissions, and:

AP[2] Selects between read-only and read/write access.

AP[1] Selects between Application level (EL0) control and the higher Exception level control.

This provides four permission settings for data accesses:

- Read-only at all levels.
- Read/write at all levels.
- Read-only at the higher Exception level, no access by software executing at EL0.
- Read/write at the higher Exception level, no access by software executing at EL0.

Note

In an implementation that does not include ARMv8.1-VHE, the only translation regime that applies to EL0 and a higher Exception level is the EL1&0 translation regime. In an implementation that includes ARMv8.1-VHE, the EL2&0 translation regime applies to both Non-secure EL0 and EL2 when the value of HCR_EL2.{E2H, TGE} is {1, 1}.

For translation regimes that apply only to accesses from a single Exception level, AP[2] determines the stage 1 data access permissions, and AP[1] is RES1, meaning it is ignored by hardware and is treated as if it is 1.

Table D5-28 shows the meaning of the AP[2:1] field for stage 1 of a translation regime that applies to both EL0 and a higher Exception level. In this table, an entry of None indicates that any access from that Exception level faults.

Table D5-28 Data access permissions for stage 1 translations that applies to EL0 and a higher Exception level

AP[2:1]	Access from higher Exception level	Access from EL0
00	Read/write	None
01	Read/write	Read/write
10	Read-only	None
11	Read-only	Read-only

For the Secure or Non-secure EL1&0, when EL2 is enabled, translation regime:

- The stage 2 translation also defines data access permissions, see *The S2AP data access permissions, Secure or Non-secure EL1&0, when EL2 is enabled, translation regime* on page D5-2611.
- When both stages of translation are enabled, *Combining the stage 1 and stage 2 data access permissions* on page D5-2635 describes how these permissions are combined.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D5-29 shows the effect of the AP[2] field for stage 1 of a translation regime that applies to only a single Exception level.

Table D5-29 Data access permissions for stage 1 translations that apply to only a single Exception level

AP[2]	Access permission
0	Read/write
1	Read-only

The S2AP data access permissions, Secure or Non-secure EL1&0, when EL2 is enabled, translation regime

In the Secure or Non-secure EL1&0, when EL2 is enabled, translation regime, when stage 2 address translation is enabled, the S2AP field in the stage 2 translation table descriptors define the data access permissions as Table D5-30 shows. In this table, an entry of None indicates that any access generates a Permission fault.

Table D5-30 Data access permissions for stage 2 of the Secure or Non-secure EL1&0, when EL2 is enabled, translation regime

S2AP	Access from Non-secure EL1 or Non-secure EL0
00	None
01	Read-only
10	Write-only
11	Read/write

The S2AP access permissions make no distinction between Non-secure accesses from EL1 and Non-secure accesses from EL0. However, when both stages of address translation are enabled, these permissions are combined with the stage 1 access permissions defined by AP[2:1], see *Combining the stage 1 and stage 2 data access permissions* on page D5-2635.

Combining the stage 1 and stage 2 attributes, EL1&0 translation regime on page D5-2634 gives more information about the use of the stage 1 and stage 2 access permissions in an implementation of virtualization.

Hierarchical control of data access permissions

The VMSAv8-64 translation table format includes mechanisms by which entries at one level of translation table lookup can set limits on the permitted entries at subsequent levels of lookup. This subsection describes how these controls apply to the data access permissions.

Note

Similar hierarchical controls apply to instruction fetching, see *Hierarchical control of instruction fetching* on page D5-2616.

However, in an implementation that includes ARMv8.1-HPD, when the value of a TCR_ELx.HPD{0} field is 1, or the value of the TCR_ELx.HPD1 field is 1, the hierarchical control of data access permissions is disabled for the translation stage controlled by that TCR_ELx, and the information in this subsection does not apply.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The restrictions apply only to subsequent levels of lookup for the same stage of translation. The APTable[1:0] field restricts the access permissions, as Table D5-31 shows. As stated in the table footnote, for a translation regime that applies to only a single Exception level, APTable[0] is RES0, meaning it is ignored by the hardware.

Table D5-31 Effect of APTable[1:0] on subsequent levels of lookup

APTable[1:0]	Effect
00	No effect on permissions in subsequent levels of lookup.
01 [a]	Access at EL0 not permitted, regardless of permissions in subsequent levels of lookup.
10	Write access not permitted, at any Exception level, regardless of permissions in subsequent levels of lookup.
11 [a]	Regardless of permissions in subsequent levels of lookup: • Write access not permitted, at any Exception level. • Read access not permitted at EL0.

a. Not valid any translation regime that applies to only a single Exception level. In the translation tables for such a regime, APTable[0] is RES0.

Note

The APTable[1:0] settings are combined with the translation table access permissions in the translation tables descriptors accessed in subsequent levels of lookup. They do not restrict or change the values entered in those descriptors.

The VMSAv8-64 provides APTable[1:0] control only for stage 1 translations. The corresponding bits are RES0 in the stage 2 translation table descriptors.

The effect of APTable applies to later entries in the translation table walk, and so its effects can be held in one or more TLB entries. Therefore, a change to APTable requires coarse-grained invalidation of the TLB to ensure that the effect of the change is visible to subsequent memory transactions.

D5.4.6 Access permissions for instruction execution

Execute-never controls determine whether instructions can be executed from a memory region. These controls are:

UXN, Unprivileged execute-never, stage 1 only

Descriptor bit[54], defined as UXN only for stage 1 of any translation regime for which stage 1 translation can support two VA ranges.

This field applies only to execution at EL0. A value of 0 indicates that this control permits execution.

XN, Execute-never

Descriptor bit[54], defined as XN for:

- Stage 1 of any translation regime for which the stage 1 translation can support only a single VA range.
- Stage 2 translations when ARMv8.2-TTS2UXN is not implemented.

Note

XN[1:0], Execute-never, stage 2 only describes the stage 2 control when ARMv8.2-TTS2UXN is implemented.

This field applies to execution at any Exception level to which the stage of translation applies. A value of 0 indicates that this control permits execution.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

PXN, Privileged execute-never, stage 1 only

Descriptor bit[53], used only for stage 1 of any translation regime for which stage 1 translation can support two VA ranges.

- For stage 1 of a translation regime for which the stage 1 translation supports only a single VA range the stage 1 descriptors define a PXN field that is RES0, meaning it is ignored by hardware.

This field applies only to execution at an Exception level higher than EL0. A value of 0 indicates that this control permits execution.

XN[1:0], Execute-never, stage 2 only

Descriptor bits[54:53], defined as XN[1:0] for:

- Stage 2 translations when ARMv8.2-TTS2UXN is implemented.

Table D5-32 shows the operation of this control.

Table D5-32 XN[1:0] stage 2 access permissions model

XN[1]	XN[0]	Access
0	0	The stage 2 control permits execution at EL1 and EL0
0	1	The stage 2 control does not permit execution at EL1, but permits execution at EL0
1	0	The stage 2 control does not permit execution at EL1 or EL0
1	1	The stage 2 control permits execution at EL1, but does not permit execution at EL0

Note

For stage 2 translations when ARMv8.2-TTS2UXN is not implemented, descriptor bit[53] is RES0, meaning it is ignored by hardware.

Note

In an implementation that does not include ARMv8.1-VHE, the only translation regime for which stage 1 translation can support two VA ranges is the EL1&0 translation regime. In an implementation that includes ARMv8.1-VHE:

- When the value of HCR_EL2.E2H is 1, TCR_EL2 controls the EL2&0 translation regime, and this regime:
  - Supports two VA ranges, corresponding to TTBR0_EL2 and TTBR1_EL2.
  - Always supports both UXN and PXN fields.
- Memory accesses from EL0 are translated using the EL2&0 translation regime only when the value of HCR_EL2.{E2H, TGE} is {1, 1}.

Table D5-32 shows the operation of the stage 2 XN[1:0] control, and for each single-bit execute-never field a value of 1 indicates that, at an exception level to which the control applies, instructions cannot be executed from the target memory region. In addition:

- For a translation regime that applies to EL0 and a higher Exception level, if the value of the AP[2:1] bits is `0b01`, permitting write access from EL0, then the PXN field is treated as if it has the value 1, regardless of its actual value.
- In a translation regime with two stages of translation, a region is execute-never if execution is not permitted by the value of the applicable execute-never field in one or both of:
  - The stage 1 translation table descriptor.
  - The stage 2 translation table descriptor.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- For each translation regime, if the value of the corresponding SCTLR_ELx.WXN field is 1 then any memory region that is writable is treated as XN, regardless of the value of the corresponding UXN, XN, or PXN field. For more information, see *Preventing execution from writable locations* on page D5-2617.
- The SCR_EL3.SIF bit prevents execution in Secure state of any instruction fetched from Non-secure memory, see *Restriction on Secure instruction fetch* on page D5-2618.

The execute-never controls apply to speculative instruction fetching, meaning speculative instruction fetch from a memory region that is execute-never at the current Exception level is prohibited.

Note

- Although the execute-never controls apply to speculative fetching, on a speculative instruction fetch from an execute-never location, no Permission fault is generated unless the PE attempts to execute the instruction that would have been fetched from that location. This means that, if a speculative fetch from an execute-never location is attempted, but there is no attempt to execute the corresponding instruction, a Permission fault is not generated.
- The software that defines a translation table must mark any region of memory that is read-sensitive as execute-never, to avoid the possibility of a speculative fetch accessing the memory region. This means it must mark any memory region that corresponds to a read-sensitive peripheral as execute-never. Hardware does not prevent speculative accesses to a region of any Device memory type unless that region is also marked as execute-never for all Exception levels from which it can be accessed.
- When no stage of address translation for the translation regime is enabled, memory regions cannot have UXN, XN, or PXN attributes assigned. *Behavior of instruction fetches when all associated stages of translation are disabled* on page D5-2587 describes how disabling all stages of address translation affects instruction fetching.

The following subsubsections give more information about the data access permission controls:

- *Stage 1 instruction access and execution permissions*.
- *Stage 2 instruction execution permissions* on page D5-2616.
- *Hierarchical control of instruction fetching* on page D5-2616.
- *Preventing execution from writable locations* on page D5-2617.
- *Restriction on Secure instruction fetch* on page D5-2618.

Stage 1 instruction access and execution permissions

Table D5-33 on page D5-2615 and Table D5-34 on page D5-2616 include the AP[2:1] read and write permissions shown in Table D5-28 on page D5-2610 and Table D5-29 on page D5-2611. These permissions are shown as:

R Indicates Read permission granted.

W Indicates Write permission granted.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D5-33 shows the stage 1 access permissions for instruction execution when using a translation regime that applies to EL0 and a higher Exception level.

Table D5-33 Stage 1 access permissions for instruction execution for a translation regime that applies to EL0 and a higher Exception level

UXN	PXN	AP[2:1]	SCTLR_ELx.WXN[a]	Access from higher Exception level	Access from EL0
0	0	00	0	R, W, Executable	Executable
			1	R, W, Not executable[b]	Executable
		01	0	R, W, Not executable[c]	R, W, Executable
			1	R, W, Not executable	R, W, Not executable[d]
		10	x	R, Executable	Executable
		11	x	R, Executable	R, Executable
0	1	00	x	R, W, Not executable	Executable
		01	0	R, W, Not executable	R, W, Executable
			1	R, W, Not executable	R, W, Not executable[d]
		10	x	R, Not executable	Executable
		11	x	R, Not executable	R, Executable
1	0	00	0	R, W, Executable	Not executable
			1	R, W, Not executable[b]	Not executable
		01	x	R, W, Not executable[c]	R, W, Not executable
		10	x	R, Executable	Not executable
		11	x	R, Executable	R, Not executable
	1	00	x	R, W, Not executable	Not executable
		01	x	R, W, Not executable	R, W, Not executable
		10	x	R, Not executable	Not executable
		11	x	R, Not executable	R, Not executable

a. Where ELx is the higher Exception level to which the translation regime applies.
b. Not executable because of SCTLR_ELx.WXN control, because region is writable at ELx.
c. Not executable, because AArch64 execution treats all regions writable at EL0 as being PXN.
d. Not executable because of SCTLR_ELx.WXN control, because region is writable at EL0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D5-34 shows the stage 1 access permissions for instruction execution when using a translation regime that applies to only a single Exception level.

Table D5-34 Access permissions for instruction execution for a translation regime that applies to only a single Exception level

XN	AP[2]	SCTLR_ELx.WXN[a]	Access permission
0	0	0	R, W, Executable
		1	R, W, Not executable[b]
	1	x	R, Executable
1	0	x	R, W, Not executable
	1	x	R, Not executable

a. Where ELx is the higher Exception level to which the translation regime applies.

b. Not executable because of the SCTLR_ELx.WXN control, because region is writable at ELx.

Note

The Access permissions for an AArch64 translation regime that applies to only a single Exception level are consistent with the following fields in the translation table entries being treated as shown:

- AP treated as RES1.
- APTable[0] treated as RES0.
- PXN treated as RES0.
- PXNTable treated as RES0.

Stage 2 instruction execution permissions

For the Secure or Non-secure EL1&0, when EL2 is enabled, stage 2 translation, the XN fields in the stage 2 translation table descriptors control the execution permission, and this control is completely independent of the S2AP access permissions:

- When ARMv8.2-TTS2UXN is not implemented the stage 2 XN field is a 1-bit field that applies to execution at both EL0 and EL1, see *XN, Execute-never* on page D5-2612.
- When ARMv8.2-TTS2UXN is implemented the stage 2 XN field is a 2-bit field that provides independent control of execution from EL0 and execution from EL1, see *XN[1:0], Execute-never, stage 2 only* on page D5-2613.

See also *Combining the stage 1 and stage 2 instruction execution permissions* on page D5-2635.

Hierarchical control of instruction fetching

The VMSAv8-64 translation table format includes mechanisms by which entries at one level of translation table lookup can set limits on the permitted entries at subsequent levels of lookup. This subsection describes how these controls apply to the instruction fetching controls.

Note

Similar hierarchical controls apply to data accesses, see *Hierarchical control of data access permissions* on page D5-2611.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

However, in an implementation that includes ARMv8.1-HPD, when the value of a TCR_ELx.HPD{0} field is 1, or the value of the TCR_ELx.HPD1 field is 1, the hierarchical control of instruction fetching is disabled for the translation stage controlled by that TCR_ELx, and the information in this subsection does not apply.

The restrictions apply only to subsequent levels of lookup at the same stage of translation, and:

- UXNTable or XNTable restricts the execute-never control:
  - When the value of the XNTable bit is 1, the UXN bit is treated as 1 in all subsequent levels of lookup, regardless of its actual value.
  - When the value of the UXNTable bit is 1, the XN bit is treated as 1 in all subsequent levels of lookup, regardless of its actual value.
  - When the value of a UXNTable or XNTable bit is 0 the bit has no effect.
- For a translation regime that applies to EL0 and a higher Exception level, PXNTable restricts the PXN control:
  - When the value of PXNTable is 1, the PXN bit is treated as 1 in all subsequent levels of lookup, regardless of the actual value of the bit.
  - When the value of PXNTable is 0 it has no effect.

Note

The UXNTable, XNTable, and PXNTable settings are combined with the XN, UXN, and PXN bits in the translation table descriptors accessed at subsequent levels of lookup. They do not restrict or change the values entered in those descriptors.

The UXNTable, XNTable, and PXNTable controls are provided only for stage 1 translations. The corresponding bits are RES0 in the stage 2 translation table descriptors.

The effect of UXNTable, XNTable, or PXNTable applies to later entries in the translation table walk, and so its effects can be held in one or more TLB entries. Therefore, a change to UXNTable, XNTable, or PXNTable requires coarse-grained invalidation of the TLB to ensure that the effect of the change is visible to subsequent memory transactions.

Preventing execution from writable locations

Armv8 provides control bits that, when corresponding stage 1 address translation is enabled, force writable memory to be treated as XN, PXN, or UXN, regardless of the value of the XN, PXN, or UXN bit:

- For a translation regime that applies to EL0 and a higher Exception level, when the value of the applicable SCTLR_ELx.WXN field is 1:
  - All regions that are writable from EL0 at stage 1 of the address translation are treated as XN.
  - All regions that are writable from EL1 at stage 1 of the address translation are treated as PXN.
- For a translation regime that applies to only a single Exception level, when the value of the applicable SCTLR_ELx.WXN field is 1, all regions that are writable at stage 1 of the address translation are treated as UXN.

Note

- The SCTLR_ELx.WXN controls are intended to be used in systems with very high security requirements.
- Setting a WXN field to 1 changes the interpretation of the translation table entry, overriding a zero value of a XN, UXN, or PXN field. It does not cause any change to the translation table entry.

For any given virtual machine, Arm expects WXN to remain static in normal operation. In particular, it is IMPLEMENTATION DEFINED whether TLB entries associated with a particular VMID reflect the effect of the values of these fields. This means that any change of these fields without a corresponding change of VMID might require synchronization and TLB invalidation, as described in *TLB maintenance requirements and the TLB maintenance instructions* on page D5-2668.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Restriction on Secure instruction fetch

EL3 provides a Secure instruction fetch bit, SCR_EL3.SIF. When the value of this bit is 1, and execution is using the EL3 translation regime, the Secure EL2 translation regime, or the Secure EL1&0 translation regime, any attempt to execute an instruction fetched from memory marked in the first stage of translation as Non-secure memory causes a Permission fault. TLB entries might reflect the value of this bit, and therefore any change to the value of this bit requires synchronization and TLB invalidation, as described in *TLB maintenance requirements and the TLB maintenance instructions* on page D5-2668.

In an implementation that does not implement EL3, the *Effective value* of this bit is 0.

D5.4.7 The Access flag

The Access flag indicates when a page or section of memory is accessed for the first time since the Access flag in the corresponding translation table descriptor was set to 0.

The AF bit in the translation table descriptors is the Access flag.

In Armv8.0, the Access flag is managed by software as described in *Software management of the Access flag*.

From Armv8.1, the Access flag can be managed by hardware as described in *Hardware management of the Access flag* on page D5-2619.

Note

The support for hardware management of the Access flag applies only to the VMSAv8-64 translation regimes.

D5.4.8 The dirty state

The dirty state indicates whether a page or section of memory is modified.

The dirty state can be managed by hardware as described in *Hardware management of dirty state* on page D5-2620.

Where the dirty state is managed in hardware, the dirty state information is encoded using the access permission bits AP[2] and S2AP[1] in conjunction with the DBM bit.

D5.4.9 Block translation entry

While the nT bit is set, if the implementation meets either level 1 or level 2 support, the PE either:

- Generates a translation fault when using a translation table entry that has the nT bit set. Such an entry is not permitted to be cached within the TLB.
- Guarantees that using a translation table entry that has the nT bit set does not break coherency, ordering guarantees or uniprocessor semantics, or fail to clear the Exclusives monitors when an entry that does not have the nT bit set is translating the same address cached within the TLB.

Note

Using a translation table entry that has the nT bit set might significantly impact the performance of the translation.

For more information, see *Support levels for changing block size* on page D5-2670.

D5.4.10 Software management of the Access flag

Armv8.0 requires that software manages the Access flag. This means an Access flag fault is generated whenever an attempt is made to read into the TLB a translation table descriptor entry for which the value of Access flag is 0.

The Access flag mechanism expects that, when an Access flag fault occurs, software resets the Access flag to 1 in the translation table entry that caused the fault. This prevents the fault occurring the next time that memory location is accessed. Entries with the Access flag set to 0 are never held in the TLB, meaning software does not have to flush the entry from the TLB after setting the flag.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

If a system incorporates components that can autonomously update translation table entries that are shared with the Arm PE, then the software must be aware of the possibility that such components can update the access flag autonomously.

In such a system, system software should perform any changes of translation table entries with an Access flag of 0, other than changes to the Access flag value, by using an Load-Exclusive/Store-Exclusive loop, to allow for the possibility of simultaneous updates.

D5.4.11 Hardware management of the Access flag and dirty state

Armv8.1 introduces the following optional features that perform hardware updates to the translation tables:

- *Hardware management of the Access flag*.
- *Hardware management of dirty state* on page D5-2620.

The support for hardware management of the Access flag and dirty state is identified by the feature ARMv8.1-TTHM.

When the hardware management of the Access flag is enabled, in situations where, without this feature, an Access flag fault would be generated, the hardware instead performs an atomic read-modify-write of the appropriate translation table descriptor to update the Access flag from 0 to 1.

When the hardware management of dirty state is enabled, if the Block or Page descriptor in a translation table indicates that a data access does not have write permission, then in situations where, without this feature, a data access would generate a Permission fault only because of this lack of write permission, the hardware checks the value of the DBM field in the Block or Page descriptor. If this field is 1, then instead of generating a Permission fault, the hardware performs an atomic read-modify-write of the translation table descriptor, to change the value of the bit that prohibits the write access.

It is permissible, but not required, that a stage 2 permission failure on the stage 1 translation table walk is generated (and has priority over the stage 1 abort generated by the stage 1 translation table entry) if all of the following are true:

- Stage 1 hardware updating of either access or dirty information is enabled.
- A stage 1 translation table entry would result in the stage 1 translation table entry having the access or dirty bit updated.
- The stage 1 translation table entry has stage 2 read permission but not stage 2 write permission.
- The stage 1 translation entry generates an abort (which might be one of an address size fault, an alignment fault caused by memory type or a permission fault).

Hardware management of the Access flag

Hardware management of the Access flag is enabled, for the corresponding stage of address translation, by the following configuration fields:

For stage 1 translations

- TCR_EL1.HA.
- TCR_EL2.HA.
- TCR_EL3.HA.

For stage 2 translations

- VTCR_EL2.HA.

Implementations are not required to support the hardware management of the Access flag. If ARMv8.1-TTHM is not supported, then the HA bit in TCR_EL1, TCR_EL2, TCR_EL3, and VTCR_EL2 is RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When the value of a configuration bit, HA, is 1, then when a memory access is made using a translation table Block or Page descriptor from the corresponding stage of address translation:

- The PE sets the value of the Access flag to 1 in the translation table descriptor in memory, in a coherent manner, by an atomic read-modify-write of the translation table descriptor, if both of the following conditions are true:
  - The descriptor does not generate a Permission fault or an Alignment fault based on the memory type.
  - If the hardware update mechanism was disabled or not implemented, the access would have generated an Access flag fault.

  When the PE updates the Access flag in this way no Access flag fault is generated.
- It is CONSTRAINED UNPREDICTABLE whether the PE sets the value of the Access flag in the translation table entry in memory to 1, in a coherent manner, by an atomic read-modify-write of the translation table descriptor, if both of the following conditions are true.
  - The descriptor generates a Permission fault or an Alignment fault based on the memory type.
  - If the hardware update mechanism was disabled or not implemented, the access would have generated an Access flag fault.

  This means that the value of the Access flag becomes UNKNOWN if the above conditions are all true.

The Access flag might be set to 1 as a result of speculative accesses by the PE.

Note

A consequence of the architectural rules for translation table accesses is that the architecture requires that for any translation to which an architecturally executed memory access occurs, the Access flag is set to 1, except as indicated in *Using break-before-make when updating translation table entries* on page D5-2669. However, because the architecture permits speculative accesses, the Access flag is permitted to be set to 1, even if there is no architecturally executed memory accesses by the processor.

When hardware updating of the Access flag is enabled, each stage of translation is treated independently. This means that a single memory access can cause a hardware update to either or both:

- The stage 1 Access flag.
- The stage 2 Access flag.

Note

Since speculative accesses are permitted to update the Access flags, it is permissible for:

- The stage 1 Access flag for a translation of a virtual address to be updated in situations where the stage 2 translation of the associated intermediate physical address that is returned by the stage 1 of the virtual address does not permit access.
- The stage 2 Access flag for a translation of an intermediate physical address to be updated in situations where the stage 1 translation of the associated virtual address which returned that intermediate physical address does not permit access.

An address translation instruction for an address is permitted, but not required, to set the Access flag in the translation table entries for that address. Correspondingly, it is IMPLEMENTATION DEFINED whether such an instruction can generate a Data Abort if the Access flag for a stage of translation is updated to be set.

When hardware updates of the Access flag are enabled for a stage of translation an address translation instruction that uses that stage of translation will not report that the address will give rise to an Access flag fault in the PAR, and the result in PAR will be as if the value of the Access flag in the translation table entries for that address was 1.

Hardware management of dirty state

The hardware management of dirty state mechanism can only be enabled if hardware management of the Access flag is enabled. For information on the hardware management of the Access flag, see *Hardware management of the Access flag* on page D5-2619.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

The hardware management of dirty state mechanism uses:

- In a stage 1 translation table access, the AP[2] bit in conjunction with the DBM bit in the translation table descriptors.
- In a stage 2 translation table access, the S2AP[1] bit in conjunction with the DBM bit in the translation table descriptors.

Hardware management of dirty state is enabled, for the corresponding stage of address translation, by the following configuration fields:

For stage 1 translations

- TCR_EL1.HD.
- TCR_EL2.HD.
- TCR_EL3.HD.

For stage 2 translations

- VTCR_EL2.HD.

Implementations are not required to support the dirty state mechanism. If this mechanism is not supported, then the HD bit in TCR_EL1, TCR_EL2, TCR_EL3, and VTCR_EL2 is RES0.

When hardware management of dirty state is enabled, and a memory access is made using a translation table Block or Page descriptor:

- For a stage 1 address translation, if the value of the TCR_ELx.HD field corresponding to the address translation is 1, then the PE sets AP[2] to 0 in the translation descriptor in memory, in a coherent manner by an atomic read-modify-write of the translation table descriptor, if both of the following conditions are true:
  - The value of the DBM field in the descriptor is 1.
  - If the hardware update mechanism was disabled or not implemented, the access using this descriptor would have generated a Permission fault only because the value of the AP[2] field is 1, indicating that the access does not have write permission.

  When the PE updates AP[2] in this way no Permission fault is generated because of the value of the AP[2] field.
- For a stage 2 address translation, if the value of the VTCR_EL2.HD field is 1, then the PE sets S2AP[1] to 1 in the translation descriptor in memory, in a coherent manner by an atomic read-modify-write of the translation table descriptor, if both of the following conditions are true:
  - The value of the DBM field in the descriptor is 1.
  - If the hardware update mechanism was disabled or not implemented, the access using this descriptor would have generated a Permission fault only because the value of the S2AP[1] field is 0, indicating that the access does not have write permission.

  When the PE updates S2AP[1] in this way no Permission fault is generated because of the value of the S2AP[1] field.

Note

The PE that does the atomic update of the translation table descriptor is expected to ensure that any cached copy of that translation table descriptor for that PE is similarly updated, or removed from the TLB, so that multiple writes from the same thread on the same PE do not lead to multiple updates to the table. This is only a performance expectation.

If, for a write access, the PE finds that a cached copy of the descriptor in a TLB had the DBM bit set to 1 and the AP[2] or S2AP[1] bit set to the value that forbids writes, then the PE must check that the cached copy is not stale with regard to the descriptor entry in memory, and if necessary perform an atomic read-modify-write update of the descriptor in memory. This applies if the cached copy of the descriptor in a TLB is either:

- A stage 1 descriptor in which DBM has the value 1 and AP[2] has the value 1.
- A stage 2 descriptor in which DBM has the value 1 and S2AP[1] has the value 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

Arm expects that, in many implementations, any atomic update of a translation table entry required by the dirty state management mechanism will cause a translation table walk.

For the hardware updating of the AP[2] and S2AP[1] bits, each translation stage is treated independently. This means a single memory access can update either or both of:

- The stage 1 AP[2] bit.
- The stage 2 S2AP[1] bit.

The architecture does not permit updates to AP[2] and S2AP[1] by the hardware management of the dirty state mechanism to occur as a result of speculative accesses by the PE that are not performed architecturally, except that for translation table entries for which the value of DBM is 1:

- A non-speculative access that passes stage 1 permissions check can update AP[2], if writes to that stage translation table are permitted and subsequently encounter a stage 2 fault. A non-speculative access that passes its stage 1 permission check but subsequently encounters a stage 2 fault is also permitted (but not required) to generate a stage 2 permission fault on the stage 1 translation table walk if all of the following is true:
  - The stage 1 hardware updating of the access flag or dirty state is enabled.
  - The stage 2 translation table entry translating the last level stage 1 translation entry has S2AP[1] ==0 and either DBM ==0 or hardware updating of the dirty state information is not enabled.

  Note

  These are cases where there is no stage 2 write permission for the hardware updating of the last level stage 1 translation table entry.

- A non-speculative access that generates an Alignment fault only because the memory type accessed is Device memory by a stage of translation can update AP[2] or S2AP[1] of that stage of translation if the memory access would have updated that translation table bit had the memory access not generated the Alignment fault.

- If the stage 2 hardware management of dirty state mechanism is enabled, the S2AP[1] field of a stage 2 translation table entry that is translating a stage 1 translation table without generating a stage 2 MMU fault:
  - Is updated from 0 to 1 as a result of a speculative update of the Access flag in an entry of that stage 1 translation table.
  - Is permitted to be updated speculatively from 0 to 1 as a result of performing a translation table walk using that stage1 translation table, even if the entry in the stage 1 translation table is not updated. The speculative update is permitted to generate a synchronous External abort or an IMPLEMENTATION DEFINED abort caused by the memory type not supporting an atomic read-modify-write.

    Note

    This applies even if the stage 1 translation table contains entries that are not the final level entries and therefore would not be updated. This relaxation avoids the hardware complexity of having to detect whether the stage 1 entry is a final level entry before deciding to set the stage 2 dirty state information.

- If an instruction that generates more than one single-copy atomic memory access has a fault on some, but not all, of those memory accesses, then AP[2] and S2AP[1] bits associated with accesses from that instruction, which do not fault are permitted to be updated if the associated hardware update of dirty state mechanism is enabled.

- If the hardware update of dirty state mechanism is enabled and a write to memory is prevented by a Synchronous Tag Check Fault, the AP[2] and S2AP[1] bits associated with that write are permitted to be updated. For more information see Chapter D6 *Armv8.5 Memory Tagging Extension*.

- When enabled, the Statistical Profiling Extension can update the AP[2] or S2AP[1] for any Page or Block translation table entry in the Profiling Buffer. See *Hardware management of dirty state and the Access flag by the Statistical Profiling Extension* on page D9-2805.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For a Block or Page translation table descriptor for which the AF bit is 0, the DBM bit is 1, and either the value of the stage 1 AP[2] bit is 1 or the value of the stage 2 S2AP[1] bit is 0, both AF can be set to 1, and either AP[2] set to 0 or S2AP[1] set to 1, in a single atomic read-modify-write operation, as a result of an attempted write to a memory location that uses the translation table entry.

Implications of enabling the dirty state management mechanism

This subsection describes behaviors that result from having the dirty state management mechanism enabled for a particular stage of address translation.

For the final level of lookup in a stage 1 translation:

In the EL3 translation regime

The OA of the lookup is treated as writable if all of the following conditions apply:

- In the descriptor for the final level of lookup, the value of DBM is 1 and the value of AP[2] is 1.
- In the descriptor for every higher level of lookup, the value of APTable[1] is 0.

In this case, if the value of SCTLR_EL3.WXN is 1 then the OA is treated as Execute-never.

In the EL2 or EL2&0 translation regime, when the value of HCR_EL2.{E2H, TGE} is not {1, 1}

Note

When the value of HCR_EL2.E2H is 1, TCR_EL2 controls the EL2&0 translation regime, and otherwise it controls the EL2 translation regime.

The OA of the lookup is treated as writable if all of the following conditions apply:

- In the descriptor for the final level of lookup, the value of DBM is 1 and the value of AP[2] is 1.
- In the descriptor for every higher level of lookup the value of APTable[1] is 0.

In this the value of SCTLR_EL2.WXN is 1 then the OA is treated as Execute-never.

In addition, if the value of HCR_EL2.E2H is 1, the OA is treated as Privileged execute-never if all of the following conditions apply:

- In the descriptor for the final level of lookup, the value of DBM is 1 and the value of AP[2:1] is `0b11`.
- In the descriptor for every higher level of lookup, the value of APTable[1:0] is `0b00`.

Note

When the value of HCR_EL2.{E2H, TGE} is not {1, 1}, memory accesses from EL0 do not use the EL2, or EL2&0, translation regime.

In the EL2&0 translation regime, when the value of HCR_EL2.{E2H, TGE} is {1, 1}

The OA of the lookup is treated as writable at EL2 and EL0, Privileged execute-never, if all of the following conditions apply:

- In the descriptor for the final level of lookup, the value of DBM is 1 and the value of AP[2:1] is `0b11`.
- In the descriptor for every higher level of lookup the value of APTable[1:0] is `0b00`.

In this case, if the value of SCTLR_EL2.WXN is 1 then the OA is also treated as Unprivileged execute-never.

The OA of the lookup is treated as writable at EL2 but not writable at EL0 if either:

- Both:
  - In the descriptor for the final level of lookup, the value of DBM is 1 and the value of AP[2:1] is `0b10`.
  - In the descriptor for every higher level of lookup the value of APTable[1:0] is `0b0x`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In this case, if the value of SCTLR_EL2.WXN is 1 then the OA is treated as Privileged execute-never.

- Both:
  - In the descriptor for the final level of lookup, the value of DBM is 1 and the value of AP[2:1] is `0b11`.
  - In at least one of the descriptors for higher levels of lookup the value of APTable[1:0] is `0b01`.

  In this case, if the value of SCTLR_EL2.WXN is 1 then the OA is treated as Privileged execute-never.

In the EL1&0 translation regime

The OA of the lookup is treated as writable at EL1 and EL0, Privileged execute-never, if all of the following conditions apply:

- In the descriptor for the final level of lookup, the value of DBM is 1 and the value of AP[2:1] is `0b11`.
- In the descriptor for every higher level of lookup the value of APTable[1:0] is `0b00`.

In this case, if the value of SCTLR_EL1.WXN is 1 then the OA is treated as Unprivileged execute-never.

The OA of the lookup is treated as writable at EL1 but not writable at EL0 if either:

- Both:
  - In the descriptor for the final level of lookup, the value of DBM is 1 and the value of AP[2:1] is `0b11`.
  - In at least one of the descriptors for higher levels of lookup the value of APTable[1:0] is `0b01`.

  In this case, if the value of SCTLR_EL1.WXN is 1 then the OA is treated as Privileged execute-never.
- Both:
  - In the descriptor for the final level of lookup, the value of DBM is 1 and the value of AP[2:1] is `0b10`.
  - In the descriptor for every higher level of lookup the value of APTable[1:0] is `0b0x`.

  In this case, if the value of SCTLR_EL1.WXN is 1 then the OA is treated as Privileged execute-never.

The OA of a translation table entry where the DBM bit is 1, and the stage 1 AP[2] bit is 1 or the stage 2 S2AP[1] bit is 0, is treated as writable:

- For data cache invalidation instructions that require write permission, that is for the DC IVAC instruction.
- For address translation instructions that require write permission, that is for the AT S12E0W, AT S12E1W, AT S1E0W, AT S1E1W, AT S1E2W, and AT S1E3W instructions.

Cache invalidation and address translation instructions never cause the stage 1 AP[2] bit or the stage 2 S2AP[1] bit in the translation table entry to be updated.

For a Store-Exclusive instruction to a memory location for which the DBM bit is 1 and the stage 1 AP[2] bit is 1, if the Store-Exclusive fails because the Exclusives monitor is not in the exclusive state, it is IMPLEMENTATION DEFINED whether the AP[2] bit in the translation table is updated.

For a Store-Exclusive instruction to a memory location for which the DBM bit is 1, and the stage 2 S2AP[1] bit is 0, if the Store-Exclusive fails because the Exclusives monitor is not in the Exclusive access state, it is IMPLEMENTATION DEFINED whether the S2AP[1] bit in the translation table is updated.

For a store to a memory location for which the DBM bit is 1, and the stage 1 AP[2] bit is 1, it is IMPLEMENTATION DEFINED whether the AP[2] bit in the translation table is updated:

- If the memory location generates a synchronous External abort on a write for a store to a memory location.
- If the memory location generates a watchpoint on a write.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For a store to a memory location for which the DBM bit is 1, and the stage 2 S2AP[1] bit is 0, it is IMPLEMENTATION DEFINED whether the S2AP[1] bit in the translation table is updated:

- If the memory location generates a synchronous External abort on a write for a store to a memory location.
- If the memory location generates a watchpoint on a write.

In the event of a PE setting the stage 1 AP[2] bit to 0, it is not required that all associated entries are removed from the TLBs of other PEs in the system.

In the event of a PE setting the stage 2 S2AP[1] bit to 1, it is not required that all associated entries are removed from the TLBs of other PEs in the system.

For the stage 2 translation tables, it is CONSTRAINED UNPREDICTABLE whether the stage 2 S2AP[1] entry is updated in response to a stage 1 translation table walk where the stage 1 translation system is configured to perform hardware updates to the Access flag or stage 1 AP[2] bit, but the values of the Access flag and AP[2] bit are such that a hardware update to the stage 1 translation table entry being accessed is not required.

In the event of a PE encountering a situation for a data write for which the DBM bit is 1 and the stage 1 AP[2] bit is 1 in a TLB, it is required that the hardware checks that the cached copy is not stale with regards to the translation table entry in memory and performs the atomic read-modify-write update with respect to table entry in memory.

In the event of a PE encountering a situation for a data write for which the DBM bit is 1 and stage 2 S2AP[1] bit is 0 in a TLB, it is required that the hardware checks that the cached copy is not stale with regards to the translation table entry in memory and performs the atomic read-modify-write update with respect to table entry in memory.

For a `CAS` or `CASP` instruction to a memory location for which the DBM bit is 1, and the stage 1 AP[2] bit is 1, if the compare fails, and the location is not updated, it is CONSTRAINED UNPREDICTABLE whether the AP[2] bit in the translation table is updated.

For a `CAS` or `CASP` instruction to a memory location for which the DBM bit is 1, and the stage 2 S2AP[1] bit is 0, if the compare fails, and the location is not updated, it is CONSTRAINED UNPREDICTABLE whether the S2AP[1] bit in the translation table is updated.

For an atomic instruction to a memory location for which the DBM bit is 1, and the stage 2 S2AP[0:1] is 00, if the instruction generates a stage 2 Permission fault as a result of not having read permission, it is CONSTRAINED UNPREDICTABLE whether the S2AP[1] bit in the translation table is updated.

D5.4.12 Ordering of hardware updates to the translation tables

A hardware update to the translation table that is caused by a load or a store, including an atomic instruction, is guaranteed to be observed, to the extent required by the shareability attributes:

- Before a load or store, including an atomic instruction, to an arbitrary address, other than the address of the translation table entry, that appears in program order after the load or store, including an atomic instruction, causing the update to the translation table entry only if a `DSB` with the appropriate shareability attributes, where the DSB applies to both loads and stores, is executed between the load or store, including an atomic instruction, that caused the update to the translation table and the subsequent load or store.

- Before a load to the translation table entry that is being updated that appears in program order after the load or store, including an atomic instruction, causing the update to the translation table entry only if a `DSB` with the appropriate shareability attributes, where the DSB applies to both loads and stores, is executed between the load or store, including an atomic instruction, that caused the update to the translation table and the subsequent load.

- Before a store or atomic access to the translation table entry that is being updated that appears in program order after the load or store, including an atomic instruction, causing the update to the translation table entry.

- Before a cache maintenance instruction to an arbitrary address appearing in program order after the load or store, including an atomic instruction, causing the update to the translation table entry only if a `DSB` with the appropriate shareability attributes, where the DSB applies to both loads and stores, is executed between the load or store, including an atomic instructing that caused the update to the translation table entry and the subsequent cache maintenance instruction.

An update to the translation table that is caused by a load is not ordered with respect to the load itself.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

An update to the translation table that is caused by a store or an atomic access is observed by all observers, to the extent required by the shareability attributes, before the store itself in the case that the store is to the same location as the translation table update.

An update to the translation table that is caused by a store or an atomic access is not ordered with respect to the store itself in the case that the store is not the same location as the translation table update.

D5.4.13 Restriction on memory types for hardware updates on translation tables

Translation tables can be placed in Normal memory with any cacheability, but the hardware updates to the translation tables require an atomic update of memory. The properties of the atomicity can be met only by functionality outside the PE. Some system implementations might not implement this functionality for all regions of memory. This can apply to:

- Any type of memory in the system that does not support hardware cache coherency.
- Non-cacheable memory, or memory that is treated as Non-cacheable, in an implementation that does not support hardware cache coherency.

An implementation can choose which memory type is treated as Non-cacheable.

The memory types for which it is architecturally guaranteed that the hardware updates of the translation tables will be atomic are:

- Inner Shareable, Inner Write-Back, Outer Write-Back Normal memory with Read allocation hints and Write allocation hints and not transient.
- Outer Shareable, Inner Write-Back, Outer Write-Back Normal memory with Read allocation hints and Write allocation hints and not transient.

If the hardware updates of the translation tables are not atomic in regard to other agents that access memory, then performing a hardware update to such a location can have one or more of the following effects:

- The hardware update generates a synchronous External abort, which is presented as an External abort on a translation table walk.
- The instruction generates a SError interrupt.
- The hardware update generates an Unsupported atomic hardware update MMU fault reported using the Fault status code of:
  - ESR_ELx.DFSC = `110001` for Data Abort exceptions.
  - ESR_ELx.IFSC = `110001` for Instruction Abort exceptions.
  - PMBSR_EL1.FSC = `110001` for an abort on a write to the Statistical Profiling buffer.

  For the Secure or Non-secure EL1&0 translation regime, when EL2 is implemented and enabled in the current Security state, if atomic hardware update is not supported because of the memory type that is defined in the first stage of translation, or the second stage of translation is not enabled, then this exception is a first stage abort and is taken to EL1. Otherwise, the exception is a second stage abort and is taken to EL2.

  The priority of this MMU fault for a stage of the translation is immediately before or immediately after the priority of a Permission fault generated by the same stage of translation as the stage of this MMU fault, as determined by an IMPLEMENTATION DEFINED choice.
- The hardware updates are performed, but there is no guarantee that the memory accesses were performed atomically in regard to other agents that access memory. In this case, the instruction might also generate a SError interrupt.

D5.4.14 Use of the Contiguous bit with hardware updates of the translation table entries

Hardware updates of the Access flag, and the AP[2] or S2AP[1] bit, only apply to a single translation table entry. An update to one of these bits in a translation table entry that also has the Contiguous bit set to 1 can give rise to translation table entries that have different Access flag, or different AP[2] or S2AP[1] bits, within the members of a group of contiguous translation table entries.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This is acceptable under the architecture when using hardware updates of the translation table entries. In addition, an access or a write to a location translated by an entry that has the Contiguous bit set might not result in a hardware update of the Access flag or the AP[2] or S2AP[1] bit, if at least one entry in the set of contiguous translation table entries has the Access flag set to 1, or the AP[2] or S2AP[1] bit indicating that the entry is dirty.

Note

- The provision of the Contiguous bit permits, but does not require, the hardware to hold a single entry in a TLB for the set of translation table entries in the group, and to have updated only one or more of the Access flags and the AP[2] bit or S2AP[1] bit for the single translation table entry that gave rise to the TLB entry.
- A consequence of this is that software must combine the Access flag values, and AP[2] or S2AP[1] values, across all translation table entries in a contiguous group to determine whether any of the entries have been accessed or written to.

For more information on the Contiguous bit, see *The Contiguous bit* on page D5-2633.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D5.5 Memory region attributes

The memory region attribute fields control the memory type, accesses to the caches, and whether the memory region is Shareable and therefore is coherent. This section also describes some additional translation table fields, that this manual groups with the memory region attributes.

In the EL1&0 translation regime, each enabled stage of address translation assigns memory region attributes, as described in this section. When both stages of translation are enabled, *Combining the stage 1 and stage 2 attributes, EL1&0 translation regime* on page D5-2634 describes how the assignments from the two stages are combined.

Note

In a virtualization implementation, a hypervisor, executing at EL2, might usefully:

- Reduce the permitted cacheability of a region.
- Increase the required shareability of a region.

The combining of attributes from stage 1 and stage 2 translations supports both of these options.

The following sections describe these attributes:

- *The stage 1 memory region attributes*.
- *The stage 2 memory region attributes, EL1&0 translation regime* on page D5-2630.
- *Other fields in the VMSAv8-64 translation table format descriptors* on page D5-2633.
- *Combining the stage 1 and stage 2 attributes, EL1&0 translation regime* on page D5-2634.

Note

- This section describes the memory region attributes for each of the translation regimes, and for each stage of translation in the Secure or Non-secure EL1&0, when EL2 is enabled, translation regime.
- A translation applies to memory accesses from either:
  - Only a single Exception level, for example the EL3 translation regime.
  - EL0 and one higher Exception level, for example the EL1&0 translation regime.
- In general, attribute assignment is simpler in a regime that applies to only a single Exception level, and in these regimes behavior is consistent with fields in the translation tables being treated as follows:
  - AP[1] is RES1, meaning the PE ignores the value of the bit and behaves as if it is 1.
  - APTable[0] is RES0, meaning the PE ignores the value of the bit and behaves as if it is 0.
  - The PXN field is RES0, meaning the PE ignores the value of the bit and behaves as if it is 0.
  - The PXNTable bit is RES0, meaning the PE ignores the value of the bit and behaves as if it is 0.

D5.5.1 The stage 1 memory region attributes

The description of the memory region attributes in a translation descriptor divides into:

Memory type and Cacheability

These are described indirectly, by registers referenced by bits in the table descriptor. This is described as *remapping* the memory type and attribute description. *Stage 1 memory region type and Cacheability attributes* describes this encoding.

Shareability The SH[1:0] field in the translation table descriptor encodes shareability information. *Stage 1 Shareability attribute, for Normal memory* on page D5-2629 describes this encoding.

Stage 1 memory region type and Cacheability attributes

In the VMSAv8-64 translation table format, the AttrIndx[2:0] field in a block or page translation table descriptor for a stage 1 translation indicates the 8-bit field in the MAIR_ELx that specifies the attributes for the corresponding memory region. The required field is Attr*n*, where *n* = AttrIndx[2:0]. For more information about AttrIndx[2:0] see *Attribute fields in stage 1 VMSAv8-64 Block and Page descriptors* on page D5-2601.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

Each MAIR_ELx is a 64-bit register that is architecturally mapped to a pair of AArch32 registers. See the MAIR_ELx register descriptions for more information.

Each MAIR_ELx.Attr*n* field defines, for the corresponding memory region:

- The memory type, Device or Normal.
- For Device memory, the Device memory type, one of:
  - Device-nGnRnE.
  - Device-nGnRE.
  - Device-nGRE.
  - Device-GRE.
- For Normal memory:
  - The inner and outer cacheability, Non-cacheable, Write-Through, or Write-Back.
  - For Write-Through Cacheable and Write-Back Cacheable regions, the Read-Allocate and Write-Allocate policy hints, each of which is *Allocate* or *No Allocate*, and the Transient allocation hints, if supported.

For more information about the memory type and attributes, see *Memory types and attributes* on page B2-153 and *Cacheability, cache allocation hints, and cache transient hints* on page D4-2502.

Stage 1 Shareability attribute, for Normal memory

When using the VMSAv8-64 translation table format, the SH[1:0] field in a block or page translation table descriptor specifies the Shareability attributes of the corresponding memory region. Table D5-35 shows the encoding of this field.

Table D5-35 SH[1:0] field encoding for Normal memory, VMSAv8-64 translation table format

SH[1:0]	Normal memory
00	Non-shareable
01	Reserved, CONSTRAINED UNPREDICTABLE[a]
10	Outer Shareable
11	Inner Shareable

a. See *Reserved values in System and memory-mapped registers and translation table entries* on page K1-7871 for the permitted CONSTRAINED UNPREDICTABLE behavior.

Note

The shareability field is only relevant if the memory is a Normal Cacheable memory type. All Device and Normal Non-cacheable memory regions are always treated as Outer Shareable, regardless of the translation table shareability attributes

See *Combining the stage 1 and stage 2 shareability attributes for Normal memory* on page D5-2637 for constraints on the Shareability attributes of a Normal memory region that is Inner Non-cacheable, Outer Non-cacheable.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D5.5.2 The stage 2 memory region attributes, EL1&0 translation regime

In the stage 2 translation table descriptors for memory regions and pages, the MemAttr[3:0] and SH[1:0] fields describe the stage 2 memory region attributes:

- *Stage 2 memory region type and Cacheability attributes* describes how the MemAttr[3:0] field defines these attributes.
- The SH[1:0] field in the translation table descriptor encodes shareability information. *Stage 2 Shareability attribute, for Normal memory* on page D5-2631 describes this encoding.

The following sections describe how, when both stages of address translation are enabled, the memory region attributes assigned at stage 2 of the translation are combined with those assigned at stage 1:

- *Combining the stage 1 and stage 2 memory type attributes* on page D5-2635.
- *Combining the stage 1 and stage 2 cacheability attributes for Normal memory* on page D5-2636.
- *Combining the stage 1 and stage 2 shareability attributes for Normal memory* on page D5-2637.

D5.5.3 Stage 2 memory region type and Cacheability attributes

Table D5-36 shows how MemAttr[3:2] gives a top-level definition of the memory type, and of the Outer cacheability of a Normal memory region.

Table D5-36 VMSAv8-64 MemAttr[3:2] encoding, stage 2 translation

MemAttr[3:2]	Memory type	Outer cacheability
00	Device. MemAttr[1:0] encodes the Device memory type.	Not applicable
01	Normal. MemAttr[1:0] encodes the Inner Cacheability.	Outer Non-cacheable
10		Outer Write-Through Cacheable
11		Outer Write-Back Cacheable

The encoding of MemAttr[1:0] depends on the Memory type indicated by MemAttr[3:2]:

- When MemAttr[3:2]==`0b00`, indicating Device memory, Table D5-37 shows the encoding of MemAttr[1:0].

Table D5-37 MemAttr[1:0] encoding for Device memory

MemAttr[1:0]	Meaning when MemAttr[3:2] == `0b00`
00	Region is Device-nGnRnE memory
01	Region is Device-nGnRE memory
10	Region is Device-nGRE memory
11	Region is Device-GRE memory

- When MemAttr[3:2]!=`0b00`, indicating Normal memory, Table D5-38 shows the encoding of MemAttr[1:0].

Table D5-38 MemAttr[1:0] encoding for Normal memory

MemAttr[1:0]	Meaning when MemAttr[3:2] != `0b00`
00	Reserved, CONSTRAINED UNPREDICTABLE[a]

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D5-38 MemAttr[1:0] encoding for Normal memory (continued)

MemAttr[1:0]	Meaning when MemAttr[3:2] != `0b00`
01	Inner Non-cacheable
10	Inner Write-Through Cacheable
11	Inner Write-Back Cacheable

a. See *Reserved values in System and memory-mapped registers and translation table entries* on page K1-7871 for the permitted CONSTRAINED UNPREDICTABLE behavior.

Note

- The stage 2 translation does not assign any allocation hints.
- The following stage 2 translation table attribute settings leave the stage 1 settings unchanged:
  — MemAttr[3:2] == `0b11`, Normal memory, Outer Write-Back Cacheable.
  — MemAttr[1:0] == `0b11`, Inner Write-Back Cacheable.

D5.5.4 Stage 2 Shareability attribute, for Normal memory

When using the VMSAv8-64 translation table format, the SH[1:0] field in a block or page translation table descriptor specifies the Shareability attributes of the corresponding memory region. Table D5-39 shows the encoding of this field.

Table D5-39 SH[1:0] field encoding for Normal memory, VMSAv8-64 translation table format

SH[1:0]	Normal memory
00	Non-shareable
01	Reserved, CONSTRAINED UNPREDICTABLE[a]
10	Outer Shareable
11	Inner Shareable

a. See *Reserved values in System and memory-mapped registers and translation table entries* on page K1-7871 for the permitted CONSTRAINED UNPREDICTABLE behavior.

Note

- This encoding is the same as the shareability encoding described in *Stage 1 Shareability attribute, for Normal memory* on page D5-2629.
- The shareability field is only relevant if the memory is a Normal Cacheable memory type. All Device and Normal Non-cacheable memory regions are always treated as Outer Shareable, regardless of the translation table shareability attributes.

See *Combining the stage 1 and stage 2 shareability attributes for Normal memory* on page D5-2637 for constraints on the Shareability attributes of a Normal memory region that is Inner Non-cacheable, Outer Non-cacheable.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D5.5.5 Stage 2 memory region type and Cacheability attributes when Armv8.4-S2FWB is implemented

When ARMv8.4-S2FWB is implemented and HCR_EL2.FWB is set to 1, then the MemAttr[5:0] is divided as follows:

- Bit[5] is RES0.
- Bit[4] determines the interpretation of bits [3:2].

When bit[4] is one the effects of bits [3:2] are defined in Table D5-40.

Table D5-40 Effect of bit[4] == 1 on Cacheability and Memory Type

Stage 1 Memory Type and Inner or Outer Cacheability attribute	Stage 2 Block/ Descriptor Bits[3:2]	Resultant Memory type and Cacheability attribute
Normal Write-Back	0b11	Normal Write-Back
Normal Write-Through		Normal Write-Though
Normal Non-cacheable		Normal Non-cacheable
Device<attr>		Device<attr>
Normal Write-Back	0b10	Normal Write-Back
Normal Write-Through		
Normal Non-cacheable		
Device<attr>		
Normal Write-Back	0b01	Normal Non-cacheable
Normal Write-Through		
Normal Non-cacheable		
Device<attr>		Device<attr>
-	0b00	RESERVED

When HCR_EL2.FWB is set to 1 and Bit[4] is 0, then the stage 2 memory type is Device. Bits[3:2] of the Stage 2 page or block descriptor define the stage 2 Device memory attributes. The stage 2 Device Memory attributes are defined in Table D5-41.

The following are unaffected by the value of HCR_EL2.FWB:

- The way that Shareability attributes from stage 1 and stage 2 are combined.
- The way that stage 1 memory types and attributes are combined with stage 2 Device type and attributes.

Table D5-41 Device Memory Attributes when Bit[4] == 0

Stage 2 page/block descriptor bits [3:2]	Device Memory Attribute
0b00	Device-nGnRnE
0b01	Device-nGnRE
0b10	Device-nGRE
0b11	Device-GRE

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D5.5.6 Other fields in the VMSAv8-64 translation table format descriptors

The following subsections describe the other fields in the translation table block and page descriptors:

- *The Contiguous bit*.
- *IGNORED fields* on page D5-2634.
- *Field reserved for software use* on page D5-2634.

The Contiguous bit

When the value of the Contiguous bit is 1, it indicates that the entry is one of a number of adjacent translation table entries that point to a *contiguous output address range*. The required number of adjacent entries depends on the current translation granule size, as follows:

4KB granule 16 adjacent translation table entries point to a contiguous output address range that has the same permissions and attributes. These 16 entries must be aligned in the translation table. If accessing a full-sized 4KB translation table, this means that the top 5 of the 9 input addresses bits that index the descriptor positions in the translation table are the same for all of the entries.

The contiguous output address range must be aligned to size of 16 translation table entries at the same translation table level.

16KB granule This bit indicates that adjacent translation table entries point to contiguous output address range that has the same permissions and attributes. With the 16KB granule, the number of contiguous entries indicated by setting this bit to 1 depends on the lookup level of the translation table:

Level 2 lookup The bit indicates 32 contiguous entries, giving a 1GB block of memory. These entries must be aligned in the translation table. When accessing a full-sized 16KB translation table, this means the top 6 of the 11 input addresses bits that index the descriptor positions in the translation table are the same for all of the entries.

The contiguous output address range must be aligned to size of 32 translation table entries at the same translation table level.

Level 3 lookup The bit indicates 128 contiguous entries, giving a 2MB block of memory. These entries must be aligned in the translation table. When accessing a full-sized 16KB translation table, this means the top 4 of the 11 input addresses bits that index the descriptor positions in the translation table are the same for all of the entries.

The contiguous output address range must be aligned to size of 128 translation table entries at the same translation table level.

64KB granule 32 adjacent translation table entries point to a contiguous output address range that has the same permissions and attributes. These 32 entries must be aligned in the translation table. If accessing a full-sized 64KB translation table, this means that the top 8 of the 13 input addresses bits that index the descriptor positions in the translation table are the same for all of the entries.

The contiguous output address range must be aligned to size of 32 translation table entries at the same translation table level.

Setting this bit to 1 means that the TLB can cache a single entry to cover the contiguous translation table entries.

This section defines the requirements for programming the Contiguous bit. *Possible errors in programming the translation table registers* on page D5-2581 describes the effect of not meeting these requirements.

The architecture does not require a PE to cache TLB entries in this way. To avoid TLB coherency issues, any TLB maintenance by address must not assume any optimization of the TLB tables that might result from use of the Contiguous bit.

TLB maintenance must be performed based on the size of the underlying translation table entries, to avoid TLB coherency issues.

Use of the Contiguous bit with hardware updates of the translation table entries on page D5-2626 describes the effect of hardware management of the Access flag and dirty state on the Contiguous bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

When ARMv8.2-LVA is implemented, the level 1 block size for the 64KB granule does not support the Contiguous bit, and that field is RES0.

IGNORED fields

In the VMSAv8-64 translation table descriptors, the following fields are identified as IGNORED, meaning the architecture guarantees that a PE makes no use of these fields:

- In the stage 1 and stage 2 Table descriptors, bits[58:51] and bits[11:2].
- In the stage 1 and stage 2 Block and Page descriptors, bit[63] and bits[58:55].
- In the stage 1 and stage 2 Block and Page descriptors in an implementation that does not include ARMv8.2-TTPBHA, bits[62:59].

Of these fields:

- In the stage 1 and stage 2 block and page descriptors, bits[58:55] are reserved for software use, see *Field reserved for software use*.
- In the stage 2 block and page descriptors:
  - Bit[63] is reserved for use by a System MMU.
  - In an implementation that does not include ARMv8.2-TTPBHA, bits[62:59] are reserved for use by a System MMU.

Field reserved for software use

The architecture reserves a 4-bit IGNORED field in the Block and translation table descriptors, bits[58:55], for software use. The definition of IGNORED means the architecture guarantees that hardware makes no use of this field.

Note

This means there is no need to invalidate the TLB if these bits are changed.

D5.5.7 Combining the stage 1 and stage 2 attributes, EL1&0 translation regime

When EL2 is enabled, the Secure or Non-secure EL1&0 translation regime comprises two stage of translation, each of which can be enabled independently:

- Stage 1 translation is configured and controlled from EL1. When enabled, stage 1 translation can define access permissions independently for access from EL0 and for accesses from EL1.

  Stage 1 MMU faults are taken to EL1.
- When stage 2 translation is enabled, the stage 2 access controls defined at EL2:
  - Affect the stage 1 access permissions settings.
  - Take no account of whether the accesses are at EL1 or EL0.
  - Permit software executing at EL2 to assign a write-only attribute to a memory region.

  Stage 2 MMU faults are taken to EL2.

Note

In an implementation of virtualization, the attributes defined in the stage 2 translation tables mean a hypervisor can define additional access restrictions to those defined by a Guest OS in the stage 1 translation tables. For a particular access, the actual access permission is the more restrictive of the permissions defined by:

- The Guest OS, in the stage 1 translation tables.
- The hypervisor, in the stage 2 translation tables.

The effects of the combination of attributes defined by the Hypervisor are functionally transparent to the Guest OS.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When HCR_EL2.FWB is 1 and the final memory type is Normal Cacheable:

- If the stage 1 page or block descriptor specifies a cacheable memory type, then the final cache allocation hint is the stage 1 cache allocation hint.
- If the stage 1 page or block descriptor does not specify a cacheable memory type, then the final cache allocation hint is Read Allocate, Write Allocate.

The effects of HCR_EL2.FWB apply to both Secure and Non-secure stage 2 translation regime.

When ARMv8.4-S2FWB is implemented, the architecture requires that CLIDR_EL1.{LOUU, LOIUS} are zero so that no levels of data cache need to be cleaned in order to manage coherency with instruction fetches.

When HCR_EL2.FWB is set to 1, and the stage 2 page or block descriptor [4:2] is set to `0b110`, the resultant memory type is Normal WriteBack cacheable regardless of the value of the stage 1 memory type.

Combining the stage 1 and stage 2 data access permissions

When both stages of translation are enabled, the following access permissions are combined:

- The stage 1 permissions described in *The AP[2:1] data access permissions, for stage 1 translations* on page D5-2610.
- The stage 2 permissions described in *The S2AP data access permissions, Secure or Non-secure EL1&0, when EL2 is enabled, translation regime* on page D5-2611.

The stage 1 and stage 2 permissions are combined as follows:

1. If an access is not permitted by the stage 1 permissions, then it generates a stage 1 Permission fault, regardless of the stage 2 permissions.
2. If an access is permitted by the stage 1 permissions, but is not permitted by the stage 2 Permissions, then it generates a stage 2 Permission fault.
3. If an access is permitted by both the stage 1 permissions and the stage 2 permissions, then it does not generate a Permission fault.

Combining the stage 1 and stage 2 instruction execution permissions

When both stages of translation are enabled, the following access permissions are combined:

- The stage 1 permissions described in *Stage 1 instruction access and execution permissions* on page D5-2614.
- The stage 2 permissions described in *Stage 2 instruction execution permissions* on page D5-2616.

The stage 1 and stage 2 permissions are combined as follows:

1. If an instruction fetch is not permitted by the stage 1 permissions, then it generates a stage 1 Permission fault, regardless of the stage 2 permissions.
2. If an instruction fetch is permitted by the stage 1 permissions, but is not permitted by the stage 2 Permissions, then it generates a stage 2 Permission fault.
3. If an instruction fetch is permitted by both the stage 1 permissions and the stage 2 permissions, then it does not generate a Permission fault.

Combining the stage 1 and stage 2 memory type attributes

The combining of memory type attributes from the two stages of translation applies only if HCR_EL2.FWB is set to 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D5-42 shows the rules for combining the stage 1 and stage 2 memory type assignments.

Table D5-42 Combining the stage 1 and stage 2 memory type assignments

Rule	If either stage of translation assigns:	The resultant memory type is:
Device has precedence over Normal	Any Device memory type	A Device memory type
non-Gathering has precedence over Gathering	A Device-nGxx memory type	A Device-nGxx memory type
non-Reordering has precedence over Reordering	A Device-nGnRx memory type	A Device-nGnRx memory type
No Early write acknowledge has precedence over Early write acknowledge	The Device-nGnRnE memory type	The Device-nGnRnE memory type

Regardless of any shareability attribute obtained as described in *Combining the stage 1 and stage 2 shareability attributes for Normal memory* on page D5-2637:

- Any location for which the resultant memory type is any type of Device memory is always treated as Outer Shareable.
- Any location for which the resultant memory type is Normal Inner Non-cacheable, Outer Non-cacheable is always treated as Outer Shareable.

  For information about how the cacheability attribute is obtained from the attributes assigned at each stage of translation see *Combining the stage 1 and stage 2 cacheability attributes for Normal memory*.

The combining of the memory type attributes from the two stages of translation means a translation table walk for stage 1 translation can be made to a type of Device memory. If this occurs then:

- If the value of HCR_EL2.PTW is 0, then the translation table walk occurs as if it is to Normal Non-cacheable memory. This means it can be done speculatively.
- If the value of HCR_EL2.PTW is 1, then the memory access generates a stage 2 Permission fault.

When the first stage of the translation regime specifies Device memory, HCR_EL2.FWB is set to 1, and the stage 2 page or block descriptor [4:2] is set to `0b110`, does not prevent:

- Instruction fetches from Device memory being a CONSTRAINED UNPREDICTABLE choice between:
  - Generating a prefetch abort.
  - Accessing memory as the resultant memory type of Normal WriteBack cacheable.
- A misaligned memory access generating a first stage alignment fault.

When the first stage of the translation regime specifies Device memory, HCR_EL2.FWB is set to 1, and the stage 2 page or block descriptor [4:2] is set to `0b110`:

- It is IMPLEMENTATION DEFINED whether Atomic memory accesses or Exclusives are supported, in the same way as it is for accesses to memory locations whose resultant memory type is Device memory.
- It is CONSTRAINED UNPREDICTABLE whether a misaligned access can generate a stage 1 alignment fault as a result of the memory type described in the stage 1 translation.

Combining the stage 1 and stage 2 cacheability attributes for Normal memory

The combining of cacheability attributes from the two stages of translation applies only if HCR_EL2.FWB is set to 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For a Normal memory region, Table D5-43 shows how the stage 1 and stage 2 cacheability assignments are combined. This combination applies, independently, for the Inner cacheability and Outer cacheability attributes.

Table D5-43 Combining the stage 1 and stage 2 cacheability assignments for Normal memory

Assignment in stage 1	Assignment in stage 2	Resultant cacheability
Non-cacheable	Any	Non-cacheable
Any	Non-cacheable	Non-cacheable
Write-Through Cacheable	Write-Through or Write-Back Cacheable	Write-Through Cacheable
Write-Through or Write-Back Cacheable	Write-Through Cacheable	Write-Through Cacheable
Write-Back Cacheable	Write-Back Cacheable	Write-Back Cacheable

Combining the stage 1 and stage 2 shareability attributes for Normal memory

A memory region is treated as Outer Shareable, regardless of any shareability assignments at either stage of translation, if either:

- The resultant memory type attribute, described in *Combining the stage 1 and stage 2 memory type attributes* on page D5-2635, is any type of Device memory.
- The resultant memory type attribute, described in *Combining the stage 1 and stage 2 memory type attributes* on page D5-2635, is Normal memory, and the resultant cacheability, described in *Combining the stage 1 and stage 2 cacheability attributes for Normal memory* on page D5-2636, is Inner Non-cacheable, Outer Non-cacheable.

For a memory region with a resultant memory type attribute of Normal, that is not Inner Non-cacheable, Outer Non-cacheable, Table D5-44 shows how the stage 1 and stage 2 shareability assignments are combined.

Table D5-44 Combining the stage 1 and stage 2 Shareability assignments for Normal memory[a]

Assignment in stage 1	Assignment in stage 2	Resultant shareability
Outer Shareable	Any	Outer Shareable
Inner Shareable	Outer Shareable	Outer Shareable
Inner Shareable	Inner Shareable	Inner Shareable
Inner Shareable	Non-shareable	Inner Shareable
Non-shareable	Outer Shareable	Outer Shareable
Non-shareable	Inner Shareable	Inner Shareable
Non-shareable	Non-shareable	Non-shareable

a. Applies only if the Normal memory is not Inner Non-cacheable, Outer Non-cacheable, see text.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D5.6 Virtualization Host Extensions

Armv8.1 introduces the Virtualization Host Extensions that provide enhanced support for a Type 2 virtualization solution, where there is a Host OS, which is either more privileged than the hypervisor, or is a peer of the hypervisor.

The Virtualization Host Extensions only apply to an implementation that includes EL2 using AArch64.

D5.6.1 State added by the Virtualization Host Extensions

The following state is added as part of ARMv8.1-VHE:

- A configuration bit, E2H, is added to HCR_EL2.
- New registers:
  - CONTEXTIDR_EL2, which has the same format and contents as CONTEXTIDR_EL1.
  - TTBR1_EL2, which has the same format and contents as TTBR1_EL1.
- An EL2 virtual timer which is accessed using the registers CNTHV_CTL_EL2, CNTHV_CVAL_EL2, and CNTHV_TVAL_EL2. The registers take the same format as CNTV_CTL_EL0, CNTV_CVAL_EL0, and CNTV_TVAL_EL0 respectively. The virtual offset is treated as 0 for this timer.

D5.6.2 Behavior of HCR_EL2.E2H

When the value of HCR_EL2.E2H is 0:

- There are no changes to the Armv8 functionality other than the new state described in *State added by the Virtualization Host Extensions*.

  Note

  This means the translation regime controlled by TCR_EL2 is called the EL2 translation regime.

- The contents of TTBR1_EL2 are ignored by hardware, other than reads by an `MRS` instruction and writes by an `MSR` instruction.
- The Context ID matching breakpoint is disabled at EL2, and uses the value of CONTEXTIDR_EL1 at EL0 and EL1.

When the value of HCR_EL2.E2H is 1, and EL2 is enabled for the current Security state:

- The translation regime controlled by TCR_EL2 is the EL2&0 translation regime, and the behaviors of this translation regime differ from those of the EL2 translation regime.
- The EL2&0 translation regime behaves in the same way as stage 1 of the EL1&0 translation regime, with an upper address range translated by tables pointed to by TTBR1_EL2. The existing TTBR0_EL2 translates the lower address range of the EL2&0 translation regime and is extended to have the same contents and format as the TTBR0_EL1.
- The translation tables used in the EL2&0 translation regime take the same format as the EL1&0 translation regime. EL2 accesses are treated as privileged in this format.
- Context ID matching can occur at EL2. When executing at EL2, a Context ID matching breakpoint uses CONTEXTIDR_EL2.
- VMID and VMID + Context ID matching breakpoints do not match at EL2.
- The virtual offset is treated as 0 when CNTVCT_EL0 is read from EL2.
- The Privileged Access Never mechanism applies to accesses from EL2 to a virtual address which has access permitted in the EL2&0 translation regime.
- The following registers are redefined:
  - CNTHCTL_EL2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— CPTR_EL2.
— TCR_EL2.

If HCR_EL2.{E2H, TGE}== {1, 0}, then all accesses from EL1 and EL0 are not included in the EL2&0 translation regime.

If HCR_EL2.{E2H, TGE} == {1, 1}:

- The EL2&0 translation regime is used when executing at EL0 as well as when executing at EL2, where EL0 accesses are treated as unprivileged.

  Note

  Accesses from EL1 are not possible under this configuration.

- In EL2, the unprivileged instructions `LDTR`, `LDTRB`, `LDTRH`, `LDTRSB`, `LDTRSH`, `LDTRSW`, `STTR`, `STTRB`, and `STTRH` act as if they are executing at EL0 for permission and watchpoint checking.
- Except for the purpose of reading the value held in the register, some fields in HCR_EL2 and all fields in HSTR_EL2 are treated as having a specific value.
- SCTLR_EL2 is redefined to include additional fields from SCTLR_EL1, and to apply to execution at EL0.
- The following timer registers, and their equivalent AArch32 registers, are redefined to access the associated _EL2 register, rather than accessing the _EL0 register when in EL0:
  — CNTP_CTL_EL0.
  — CNTP_CVAL_EL0.
  — CNTP_TVAL_EL0.
  — CNTV_CTL_EL0.
  — CNTV_CVAL_EL0.
  — CNTV_TVAL_EL0.

  For some information on registers that are redirected, see *System and Special-purpose register redirection*.
- When executing at EL0, a Context ID matching breakpoint uses CONTEXTIDR_EL2.
- VMID and VMID + Context ID matching breakpoints do not match at EL0.
- The CPACR_EL1 register does not cause any instructions to be trapped to EL1, regardless of the contents of CPACR_EL1.
- The CNTKCTL_EL1 register does not cause any instructions to be trapped to EL1, and the event stream event caused by the CNTKCTL_EL1 is disabled, regardless of the contents of CNTKCTL_EL1.
- The virtual offset is treated as 0 when CNTVCT_EL0 is read from EL0 or EL2.
- The TLB maintenance and address translation instructions that apply to the EL1&0 translation regime are redefined to apply to the EL2&0 translation regime. See *A64 System instructions for address translation* on page C5-536 and *A64 System instructions for TLB maintenance* on page C5-561.
- When executing at EL2 or EL0, any physical interrupt that is configured to be taken at EL2 is subject to the PSTATE.{D, A, I, F} interrupt masks. If the mask bit is set, then the corresponding interrupt will not be taken. If the mask bit is not set, then the corresponding interrupt will be taken. See *Asynchronous exception masking* on page D1-2367.
- When an exception is taken from EL0 to EL2, the value of the HCR_EL2.RW bit is not considered when determining the exception vector offset to use. Table D1-7 on page D1-2342 lists the vector offsets used when an exception is taken from EL0.

D5.6.3 System and Special-purpose register redirection

When ARMv8.1-VHE is implemented, and HCR_EL2.E2H is set to 1, when executing at EL2, some EL1 System register access instructions are redefined to access the equivalent EL2 register.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D5-45 shows the System register access instruction encodings that are redirected to the equivalent EL2 register when the named mnemonic is used.

Table D5-45 System register redirection

System register access instruction encoding					Mnemonic	Equivalent register accessed at EL2
op0	op1	CRn	CRm	op2		
3	0	1	0	0	SCTLR_EL1	SCTLR_EL2
				2	CPACR_EL1	CPTR_EL2
			2	1	TRFCR_EL1	TRFCR_EL2
		2	0	0	TTBR0_EL1	TTBR0_EL2
				1	TTBR1_EL1	TTBR1_EL2
				2	TCR_EL1	TCR_EL2
		5	1	0	AFSR0_EL1	AFSR0_EL2
				1	AFSR1_EL1	AFSR1_EL2
			2	0	ESR_EL1	ESR_EL2
		6	0	0	FAR_EL1	FAR_EL2
		10	2	0	MAIR_EL1	MAIR_EL2
			3	0	AMAIR_EL1	AMAIR_EL2
		12	0	0	VBAR_EL1	VBAR_EL2
		13	0	1	CONTEXTIDR_EL1	CONTEXTIDR_EL2
		14	1	0	CNTKCTL_EL1	CNTHCTL_EL2
	3	14	2	0	CNTP_TVAL_EL0	CNTHP_TVAL_EL2
						CNTHPS_TVAL_EL2[a]
				1	CNTP_CTL_EL0	CNTHP_CTL_EL2
						CNTHPS_CTL_EL2[a]
				2	CNTP_CVAL_EL0	CNTHP_CVAL_EL2
						CNTHPS_CVAL_EL2[a]
3	3	14	3	0	CNTV_TVAL_EL0	CNTHV_TVAL_EL2
						CNTHVS_TVAL_EL2[a]
				1	CNTV_CTL_EL0	CNTHV_CTL_EL2
						CNTHVS_CTL_EL2[a]
				2	CNTV_CVAL_EL0	CNTHV_CVAL_EL2
						CNTHVS_CVAL_EL2[a]

a. This register is accessed when ARMv8.4-SecEL2 is implemented and enabled, when the value of SCR_EL3.EEL2 is 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D5-46 shows the Special-purpose register access instruction encodings that are redirected to the equivalent EL2 register when the named mnemonic is used.

Table D5-46 Special-purpose register redirection

Special-purpose register access instruction encoding			Mnemonic	Equivalent register accessed at EL2
op1	CRm	op2		
0	0	0	SPSR_EL1	SPSR_EL2
		1	ELR_EL1	ELR_EL2

D5.6.4 System and Special-purpose register aliasing

New register encodings, and aliases, are provided so that software executing at EL2 can access the EL1 registers for which accesses from EL2 are redirected as described in *System and Special-purpose register redirection* on page D5-2639. These aliases can also be used at EL3, but are UNDEFINED at EL1 and EL0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D5-47 shows the System register access instruction encodings that are aliased.

Table D5-47 System register aliases

System register access instruction encoding					Mnemonic	Register accessed
op0	op1	CRn	CRm	op2		
3	5	1	0	0	SCTLR_EL12	SCTLR_EL1
				2	CPACR_EL12	CPACR_EL1
			2	0	ZCR_EL12	ZCR_EL1[a]
				1	TRFCR_EL12	TRFCR_EL1
		2	0	0	TTBR0_EL12	TTBR0_EL1
				1	TTBR1_EL12	TTBR1_EL1
				2	TCR_EL12	TCR_EL1
		5	1	0	AFSR0_EL12	AFSR0_EL1
				1	AFSR1_EL12	AFSR1_EL1
			2	0	ESR_EL12	ESR_EL1
		6	0	0	FAR_EL12	FAR_EL1
		9	9	0	PMSCR_EL12	PMSCR_EL1
		10	2	0	MAIR_EL12	MAIR_EL1
			3	0	AMAIR_EL12	AMAIR_EL1
		12	0	0	VBAR_EL12	VBAR_EL1
		13	0	1	CONTEXTIDR_EL12	CONTEXTIDR_EL1
		14	1	0	CNTKCTL_EL12	CNTKCTL_EL1
			2	0	CNTP_TVAL_EL02	CNTP_TVAL_EL0
				1	CNTP_CTL_EL02	CNTP_CTL_EL0
				2	CNTP_CVAL_EL02	CNTP_CVAL_EL0
3	5	14	3	0	CNTV_TVAL_EL02	CNTV_TVAL_EL0
				1	CNTV_CTL_EL02	CNTV_CTL_EL0
				2	CNTV_CVAL_EL02	CNTV_CVAL_EL0

a. Scalable Vector Extension System register, see *The Scalable Vector Extension (SVE)* on page A2-99.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D5-48 shows the Special-purpose register aliasing.

Table D5-48 Special-purpose register aliases

Special-purpose register access instruction encoding					Register name	Register accessed
op0	op1	CRn	CRm	op2		
3	5	4	0	0	SPSR_EL12	SPSR_EL1
				1	ELR_EL12	ELR_EL1

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D5.7 Nested virtualization

From Armv8.3, nested virtualization is supported in AArch64 state:

- If ARMv8.3-NV is implemented, a Host hypervisor executing at EL2 can run a Guest hypervisor at EL1, see *Armv8.3 nested virtualization functionality*.
- If ARMv8.4-NV is implemented, the PE further transforms System register accesses into memory accesses, see *Enhanced support for nested virtualization* on page D5-2646.

D5.7.1 Armv8.3 nested virtualization functionality

Note

- When running a Guest hypervisor with HCR_EL2.E2H == 0, the Host hypervisor must set HCR_EL2.TVM and CPTR_EL2.TCPAC to trap any Guest hypervisor accesses to the EL1 System registers that would be accesses from any Guest OS running under the Guest hypervisor
- ARMv8.3-NV does not introduce any changes to either debug or to the Performance Monitors. Arm assumes that the Host hypervisor will trap accesses to the Breakpoint and Performance Monitors registers to EL2, so that it can process any accesses to these registers made by a Guest hypervisor or by a Guest OS running under the Guest hypervisor.

ARMv8.3-NV adds the fields HCR_EL2.{NV, NV1, AT}, see:

- *Effect of HCR_EL2.{NV, NV1}*.
- *Effect of HCR_EL2.AT* on page D5-2646.

Effect of HCR_EL2.{NV, NV1}

The following subsections describe the effect of HCR_EL2.{NV, NV1}:

- *Behavior when HCR_EL2.NV==1*.
- *Additional behavior when HCR_EL2.NV == 1 and HCR_EL2.NV1 == 0* on page D5-2645.
- *Additional behaviors when HCR_EL2.NV == 1 and HCR_EL2.NV1 == 1* on page D5-2645.
- *Behavior when HCR_EL2.NV == 0 and HCR_EL2.NV1 == 1* on page D5-2645.

HCR_EL2.{NV, NV1} are both permitted to be cached in a TLB.

Behavior when HCR_EL2.NV==1

The following behaviors apply when the value of HCR_EL2.NV is 1, regardless of the value of HCR_EL2.NV1. At EL1:

- Reads or writes to any allocated and implemented System register or Special-purpose register named *_EL2, *_EL02, or *_EL12 in the `MRS` or `MSR` instruction, other than SP_EL2, are trapped to EL2 rather than being UNDEFINED. In this case, ESR_EL2 uses the EC code of `0x18`.

  Only accesses that are permitted at EL2 are trapped. This means that, for example, if the register is a read-only register at EL2, then an `MSR` from Non-secure EL1 to the register is not trapped by this mechanism. Instead the register access remains UNDEFINED.

  Note

  The priority of this trapping relative to other configurable traps follows the standard hierarchy of exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64 state* on page D1-2355.

- Reads or writes to SPSR_irq, SPSR_abt, SPSR_und, or SPSR_fiq using `MRS` and `MSR` instructions are trapped to EL2 rather than being UNDEFINED. In this case the exception is reported in ESR_EL2 using the EC code `0x18`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Reads or writes to SP_EL1 using the dedicated MRS and MSR instruction for accessing that register are trapped to EL2 rather than being UNDEFINED. In this case the exception is reported in ESR_EL2 using the EC code 0x18.
- Execution of the EL2 translation regime Address translation instructions and TLB maintenance instructions are trapped to EL2 rather than being UNDEFINED. In this case the exception is reported in ESR_EL2 using the EC code 0x18.
- Execution of the EL1 translation regime Address translation instructions and TLB maintenance instructions that are only accessible from EL2 and above are trapped to EL2 rather than being UNDEFINED. In this case the exception is reported in ESR_EL2 using the EC code 0x18.
- The ERETAA, ERETAB, and ERET instructions are trapped to EL2. In this case the exception is reported in ESR_EL2 using the EC code 0x1A.

  Note

  The ERETAA and ERETAB instructions are only available when ARMv8.3-PAuth is implemented.

- A read of CurrentEL returns the value 0x2 in bits[3:2].
- If EL3 is not implemented and HCR_EL2.TSC == 1, an SMC instruction executed at EL1 is trapped to EL2 rather than being UNDEFINED, and HCR_EL2.TSC is not RES0. In this case the exception is reported in ESR_EL2 using the EC code 0x17.

Additional behavior when HCR_EL2.NV == 1 and HCR_EL2.NV1 == 0

At EL1, all the behaviors described in *Behavior when HCR_EL2.NV==1* on page D5-2644 apply.

In addition, when HCR_EL2.{NV, NV1}=={1, 0}, any exception taken from EL1 to EL1 causes SPSR_EL1.M[3:2] to be set to 0b10 rather than 0b01.

Additional behaviors when HCR_EL2.NV == 1 and HCR_EL2.NV1 == 1

At Non-secure EL1, all the behaviors described in *Behavior when HCR_EL2.NV==1* on page D5-2644 apply.

In addition, when HCR_EL2.{NV, NV1}=={1, 1}:

- Accesses to VBAR_EL1, ELR_EL1, and SPSR_EL1 from EL1 are trapped to EL2. In this case the exception is reported in ESR_EL2 using the EC code 0x18.
- In the EL1 translation table Block and Page descriptors:
  - Bit[54] holds PXN, not UXN.
  - Bit[53] is RES0.
  - Bit[6] is treated as 0 regardless of the actual value.
- If Hierarchical permissions are enabled, then in the EL1 translation table Table descriptor:
  - Bit[61] is treated as 0 regardless of the actual value.
  - Bit[60] holds PXNTable, not UXNTable.
  - Bit[59] is RES0.
- When in EL1, PSTATE.PAN is treated as 0 for all purposes except reading the value of the bit.
- When executed at EL1, the LDTR* behave as the corresponding LDR* instructions, and the STTR* instructions behave as the equivalent STR* instructions.

Behavior when HCR_EL2.NV == 0 and HCR_EL2.NV1 == 1

When HCR_EL2.{NV, NV1}=={0, 1}, the behavior is a CONSTRAINED UNPREDICTABLE choice of:

- Behaving as if HCR_EL2.NV==1 and HCR_EL2.NV1==1 for all purposes other than reading back the value of the HCR_EL2.NV bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Behaving as if HCR_EL2.NV==0 and HCR_EL2.NV1==0 for all purposes other than reading back the value of the HCR_EL2.NV1 bit.
- Behaving as defined for HCR_EL2.NV==0, with HCR_EL2.NV1==1 having the effect of causing accesses to VBAR_EL1, ELR_EL1, and SPSR_EL1 from EL1 to be trapped to EL2.

Effect of HCR_EL2.AT

When ARMv8.3-NV is implemented, if HCR_EL2.AT is 1, then EL1 accesses to AT S1E0R, AT S1E0W, AT S1E1R, AT S1E1W, AT S1E1RP, and AT S1E1WP, are trapped to EL2. In this case the exception is reported in ESR_EL2 using the EC code `0x18`.

D5.7.2 Enhanced support for nested virtualization

If ARMv8.4-NV is implemented, the PE can access the VNCR_EL2 register and the control bit HCR_EL2.NV2.

When HCR_EL2.NV2 is 1:

- When in EL1, the PE redirects EL2 register accesses to EL1 register accesses, see *Redirection of register accesses from EL2 to EL1*.
- When a Guest hypervisor issues System register access instructions to a Guest Guest OS, the PE transforms the System register access instructions into memory access instructions, see *Loads and stores generated by transforming register accesses*.

When HCR_EL2.NV2 is 0, the behavior of HCR_EL2.NV and HCR_EL2.NV1 are as described in *Armv8.3 nested virtualization functionality* on page D5-2644.

Redirection of register accesses from EL2 to EL1

When HCR_EL2.NV and HCR_EL2.NV2 are set to 1, instructions accessing certain Special-purpose EL2 registers executed at EL1 are redefined to access the corresponding EL1 register:

Table D5-49 Redirection of accesses to special-purpose registers at EL2

Special register access instruction [a]	Named EL2 register	Actual register accessed
op1 = 4, CRm=0, op2=0	SPSR_EL2	SPSR_EL1
op1 = 4, CRm=0, op2=1	ELR_EL2	ELR_EL1

a. For further information see *op0==0b11, Moves to and from Special-purpose registers* on page C5-387.

When HCR_EL2.NV and HCR_EL2.NV2 are set to 1, instructions accessing certain System registers executed at EL1 are redefined to access the corresponding EL1 register:

Table D5-50 Redirection of accesses to System registers at EL2

System register access instruction [a]	Named EL2 register	Actual register accessed
op0 = 3, op1=4, CRn=5, CRm=2, op2=0	ESR_EL2	ESR_EL1
op0 = 3, op1=4, CRn=6, CRm=0, op2=0	FAR_EL2	FAR_EL1

a. For further information see *Instructions for accessing non-debug System registers* on page D12-2853.

Loads and stores generated by transforming register accesses

When an MRS or MSR instruction is executed at EL1 and is accessing a register listed in Table D5-51 on page D5-2647, the PE transforms that access into a loads or store, respectively.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When the PE transforms a System register access into a memory access, the address of the resulting memory access is defined using a combination of a base address and an offset according to the formula `SignExtend(VNCR_EL2.BADDR : Offset<11:0>, 64)`:

- VNCR_EL2 holds the base memory address used for memory redirection of System register accesses.
- Each register which supports redirection to memory has a unique offset value, see Table D5-51.

Table D5-51 Memory address offsets associated with each transformed register access

Register access		Offset
If HCR_EL2.{NV, NV1, NV2} == {1, 0, 1}	**If HCR_EL2.{NV, NV1, NV2} == {1, 1, 1}**	
VTTBR_EL2	VTTBR_EL2	`0x20`
VSTTBR_EL2	VSTTBR_EL2	`0x30`
VTCR_EL2	VTCR_EL2	`0x40`
VSTCR_EL2	VSTCR_EL2	`0x48`
VMPIDR_EL2	VMPIDR_EL2	`0x50`
CNTVOFF_EL2	CNTVOFF_EL2	`0x60`
HCR_EL2	HCR_EL2	`0x78`
HSTR_EL2	HSTR_EL2	`0x80`
VPIDR_EL2	VPIDR_EL2	`0x88`
TPIDR_EL2	TPIDR_EL2	`0x90`
VNCR_EL2	VNCR_EL2	`0xB0`
CPACR_EL12	CPACR_EL1	`0x100`
CONTEXTIDR_EL12	CONTEXTIDR_EL1	`0x108`
SCTLR_EL12	SCTLR_EL1	`0x110`
ACTLR_EL1	ACTLR_EL1	`0x118`
TCR_EL12	TCR_EL1	`0x120`
AFSR0_EL12	AFSR0_EL1	`0x128`
AFSR1_EL12	AFSR1_EL1	`0x130`
ESR_EL12	ESR_EL1	`0x138`
MAIR_EL12	MAIR_EL1	`0x140`
AMAIR_EL12	AMAIR_EL1	`0x148`
MDSCR_EL1	MDSCR_EL1	`0x158`
SPSR_EL12	SPSR_EL1	`0x160`
CNTV_CVAL_EL02	CNTV_CVAL_EL0	`0x168`
CNTV_CTL_EL02	CNTV_CTL_EL0	`0x170`
CNTP_CVAL_EL02	CNTP_CVAL_EL0	`0x178`
CNTP_CTL_EL02	CNTP_CTL_EL0	`0x180`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D5-51 Memory address offsets associated with each transformed register access

Register access		Offset
If HCR_EL2.{NV, NV1, NV2} == {1, 0, 1}	**If HCR_EL2.{NV, NV1, NV2} == {1, 1, 1}**	
SCXTNUM_EL12	SCXTNUM_EL1	0x188
TFSR_EL12	TFSR_EL1	0x190
CNTPOFF_EL2	CNTPOFF_EL2	0x1A8
HFGRTR_EL2	HFGRTR_EL2	0x1B8
HFGWTR_EL2	HFGWTR_EL2	0x1C0
HFGITR_EL2	HFGITR_EL2	0x1C8
HDFGRTR_EL2	HDFGRTR_EL2	0x1D0
HDFGWTR_EL2	HDFGWTR_EL2	0x1D8
ZCR_EL12	ZCR_EL1	0x1E0
HAFGRTR_EL2	HAFGRTR_EL2	0x1E8
TTBR0_EL12	TTBR0_EL1	0x200
TTBR1_EL12	TTBR1_EL1	0x210
FAR_EL12	FAR_EL1	0x220
ELR_EL12	ELR_EL1	0x230
SP_EL1	SP_EL1	0x240
VBAR_EL12	VBAR_EL1	0x250
ICH_LR<n>_EL2	ICH_LR<n>_EL2	0x400+8*n
ICH_AP0R<n>_EL2	ICH_AP0R<n>_EL2	0x480+8*n
ICH_AP1R<n>_EL2	ICH_AP1R<n>_EL2	0x4A0+8*n
ICH_HCR_EL2	ICH_HCR_EL2	0x4C0
ICH_VMCR_EL2	ICH_VMCR_EL2	0x4C8
VDISR_EL2	VDISR_EL2	0x500
VSESR_EL2	VSESR_EL2	0x508
PMBLIMITR_EL1	PMBLIMITR_EL1	0x800
PMBPTR_EL1	PMBPTR_EL1	0x810
PMBSR_EL1	PMBSR_EL1	0x820
PMSCR_EL12	PMSCR_EL1	0x828
PMSEVFR_EL1	PMSEVFR_EL1	0x830
PMSICR_EL1	PMSICR_EL1	0x838
PMSIRR_EL1	PMSIRR_EL1	0x840
PMSLATFR_EL1	PMSLATFR_EL1	0x848

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D5-51 Memory address offsets associated with each transformed register access

Register access		Offset
If HCR_EL2.{NV, NV1, NV2} == {1, 0, 1}	If HCR_EL2.{NV, NV1, NV2} == {1, 1, 1}	
TRFCR_EL12	TRFCR_EL1	`0x880`
AMEVCNTVOFF0<n>_EL2	AMEVCNTVOFF0<n>_EL2	`0xA00+8*n`
AMEVCNTVOFF1<n>_EL2	AMEVCNTVOFF1<n>_EL2	`0xA80+8*n`

Note

Software should assume that future expansion of the architecture will allocate offset values up to but not including `0x1000`.

Registers that affect hypervisor execution by controlling the event stream are not included in Table D5-51 on page D5-2647:

- CNTHCTL_EL2
- When HCR_EL2.NV1 is 0, CNTKCTL_EL12.
- When HCR_EL2.NV1 is 1, CNTKCTL_EL1.

When a System register access is transformed into a memory access, that memory access has a defined format:

- The addressees the memory access is translated by the EL2 translation regime.
- The endianness of the memory access is defined by SCTLR_EL2.EE.
- The memory access is 64-bit single-copy atomic aligned to 64 bits.
- The memory access does not have acquire or release semantics.

  Note

  The value of the transformed System register access is not affected by fields that are defined to be RES0 or RES1 in the associated System register.

- When there is no context synchronizing operation between the read or write of the register and the load or store instruction accessing the address, the PE is permitted, but not required, to reorder the memory accesses with respect to any EL1 reads or writes generated by load or store instructions to the same address.
- The memory accesses behave as if PSTATE.PAN == 0 regardless of the value of PSTATE.PAN.

When a register access instruction targets a register that is not implemented, the PE treats access to that register as UNALLOCATED.

Any attempt to trap a register access instruction is subject to the exception prioritization rules, unless it is trapped by either or both of HCR_EL2.{NV, NV1}. When a System register access instruction is trapped by either or both of HCR_EL2.{NV, NV1}, then the instruction is transformed into a memory access instruction instead of creating a trap, see *Synchronous exception prioritization for exceptions taken to AArch64 state* on page D1-2355.

Exceptions from transformed register accesses

When HCR_EL2.{NV2, NV} == {1,1} any exception taken from EL1 and taken to EL1 causes the SPSR_EL1.M[3:2] to be set to `0b10` and not `0b01`.

When the memory access generates a Data Abort, then the resulting fault has a defined format:

- The fault is taken to EL2, using the standard vector offset for exceptions from EL1 to EL2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The fault is reported as a Data Abort from the current exception level with the ESR_EL2.EC code `0x25`, see *ISS encoding for an exception from a Data Abort* on page D13-3033.
- FAR_EL2 is updated to hold the faulting address.

When the memory access generates a synchronous External abort, and when External aborts are not configured to be taken to EL3, then the resulting fault has a defined format:

- The fault is taken to EL2 using the standard vector offset for exceptions from EL1 to EL2.
- The fault is reported as a Data Abort from the current Exception level with ESR_EL2.EC code `0x25`, see *ISS encoding for an exception from a Data Abort* on page D13-3033.

The VNCR field in ESR_EL2 and ESR_EL3 identifies whether the fault came from use of VNCR_EL2 by EL1, see *ISS encoding for an exception from a Data Abort* on page D13-3033.

Interaction with self-hosted and External debug

When a register access is transformed into a memory access, the PE checks the memory access against the watchpoint registers as an EL2 access.

When the memory access matches an EL2 access in the watchpoint registers, while a watchpoint is linked to a context-aware breakpoint that is programmed to match the value held in CONTEXTIDR_EL1, then it is CONSTRAINED UNPREDICTABLE whether there is a watchpoint match.

Note

A breakpoint programmed to match the value held in CONTEXTIDR_EL1 falls under one of four classes of breakpoint:

- 0b0011, Linked Context ID Match breakpoint.
- 0b0111, Linked CONTEXTIDR_EL1 Match.
- 0b1011, Linked Context ID and VMID Match breakpoint.
- 0b1111, Linked Full Context ID Match.

If ARMv8.4-NV is implemented, the hypervisor must use the 0b1101, Linked CONTEXTIDR_EL2 Match breakpoint type to guarantee a linked match.

When there is a watchpoint match, while EDSCR.HDE is set to 1 and halting is allowed, the watchpoint match generates a Watchpoint debug event.

When there is a watchpoint match, while EDSCR.HDE is set to 0 and debug exceptions are enabled at EL2, then the watchpoint match generates a Watchpoint exception.

When the watchpoint match generates a Watchpoint exception, the resulting exception has a defined format:

- The exception is taken to EL2.
- The exception is reported as a Watchpoint from the current Exception level with the ESR_EL2.EC code `0x35`, see *ISS encoding for an exception from a Watchpoint exception* on page D13-3042.
- FAR_EL2 is updated to hold the watchpointed address.

The VNCR field in ESR_EL2 identifies whether the Watchpoint exception came from use of VNCR_EL2 by EL1, see *ISS encoding for an exception from a Watchpoint exception* on page D13-3042.

When in Debug state, or while using performance monitoring, Statistical Profiling, or permission checking, breakpoints, trace, and sampling of operations all treat the read and write of registers as executed at EL1.

When in Debug state, or while using performance monitoring, Statistical Profiling, or permission checking, the MMU and watchpoint unit treat the loads and stores generated by the transformation of reads and writes of registers as an EL2 access.

While using performance monitoring, any MRS or MSR instructions that have been transformed to loads or stores by this mechanism are treated as loads or stores.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When the Statistical Profiling Extension selects the instruction generating the memory access for profiling, *Operation Type packet payload (Load/store)* on page D10-2831 records the operation as a Load/Store operation, with a SUBCLASS value of `0b0000100x`, where bit[0] is 0 for a load and 1 for a store

When the Statistical Profiling Extension selects the instruction generating the memory access for profiling while Statistical Profiling is disabled at EL2, the virtual address for the memory access is not recorded.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D5.8 VMSAv8-64 memory aborts

In a VMSAv8-64 implementation, the following mechanisms cause a PE to take an exception on a failed memory access:

Debug exception An exception caused by the debug configuration, see Chapter D2 *AArch64 Self-hosted Debug*.

Alignment fault An Alignment fault is generated if the address used for a memory access does not have the required alignment for the operation. For more information, see *Alignment support* on page B2-148.

MMU fault An MMU fault is a fault generated by the fault checking sequence for the current translation regime. See *Types of MMU faults*.

External abort Any memory system fault other than a Debug exception, an Alignment fault, or an MMU fault.

Collectively, these mechanisms are called *aborts*. Chapter D2 *AArch64 Self-hosted Debug* and on page H3-6965 describe Debug exceptions, and the remainder of this section describes Alignment faults, MMU faults, and External aborts.

An access that causes an abort is said to be aborted, and uses the *Fault Address Registers* (FARs) and *Exception Syndrome Registers* (ESRs) to record context information.

In AArch64 state MMU faults are synchronous exceptions that are reported as either:

- Data Aborts.
- Instruction Aborts

Note

Instruction Aborts report any synchronous memory abort on an instruction fetch.

The Exception level that an MMU fault is taken to depends on the translation regime and stage that generated the fault. The fault context saved in the appropriate ESR_ELx, where ELx is the Exception level that the fault is taken to, is dependent on whether:

- The MMU fault is reported as an Instruction or as a Data Abort.
- The exception is taken from the same or a lower Exception level.

For more information, see *Synchronous exception types, routing and priorities* on page D1-2354.

External aborts can be reported synchronously or asynchronously. Asynchronous External aborts are reported using the SError interrupt. For more information, see *External aborts* on page D4-2524.

Software stepping, which is a debug feature, and a PC alignment fault exception are the only exceptions that are higher priority than an Instruction Abort. Only watchpoints are at a lower priority than Data Aborts in the exception priority hierarchy. For more information, see *Synchronous exception prioritization for exceptions taken to AArch64 state* on page D1-2355.

The following sections describe the abort mechanisms:

- *Types of MMU faults*.
- *The MMU fault-checking sequence* on page D5-2655.
- *AArch64 state prioritization of synchronous aborts from a single stage of address translation* on page D5-2659.
- *Pseudocode description of the MMU faults* on page D5-2661.

D5.8.1 Types of MMU faults

This section describes the faults that might be detected during one of the fault-checking sequences described in *The MMU fault-checking sequence* on page D5-2655. The following list includes all the types of exceptions that can occur:

- Alignment fault on a data access, see *Alignment support* on page B2-148.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Permission fault.
- Translation fault.
- Address size fault.
- Synchronous External abort on a translation table walk.
- Access flag fault.
- TLB conflict abort.

When an MMU fault generates an abort for a region of memory, no memory access is made if that region is or could be marked as Device.

The following subsections describe the MMU faults that are not described elsewhere this Manual.

Permission fault

A Permission fault can be generated at any level of lookup, and the reported fault code identifies the lookup level. See *About access permissions* on page D5-2606 for information about conditions that cause a Permission fault.

A TLB might hold a translation table entry that causes a Permission fault. Therefore, if the handling of a Permission fault results in an update to the associated translation tables, the software that updates the translation tables must invalidate the appropriate TLB entry, to prevent the stale information in the TLB being used on a subsequent memory access.

This maintenance requirement applies to Permission faults in both stage 1 and stage 2 translations.

Cache maintenance instructions cannot generate Permission faults, except that:

- A stage 1 translation table walk performed as part of a cache maintenance instruction can generate a stage 2 Permission fault as described in Stage 2 fault on a stage 1 translation table walk.
- When the value of SCTLR_EL1.UCI is 1, enabling EL0 execution of the `DC CVAU`, `DC CVAC`, `DC CVAP`, `DC CIVAC`, and `IC IVAU` instructions:
  - Executing a `DC CVAU`, `DC CVAC`, `DC CVAP`, or `DC CIVAC` instruction at EL0 to a location that does not have read permission at EL0 generates a Permission fault, subject to the constraints described in this section.
  - It is IMPLEMENTATION DEFINED whether executing an `IC IVAU` instruction at EL0 to a location that does not have read permission at EL0 generates a Permission fault.
- A `DC IVAC` instruction requires write permission to the address it invalidates, otherwise it generates a Permission fault, subject to the constraints described in this section.

  Note

  - Execution of the `DCIMVAC` instruction in AArch32 state does not have this write permission requirement.
  - When EL1&0 stage 2 address translation is enabled, a `DC IVAC` instruction executed in Non-secure state performs a cache clean and invalidate, meaning it performs the same invalidation as a `DC CIVAC` instruction, as described in *Effects of virtualization and Security state on the cache maintenance instructions* on page D4-2515.

In all cases where the execution of a cache maintenance instruction might generate a Permission fault:

- If the Point of Coherency is before any level of cache, it is IMPLEMENTATION DEFINED whether a cache maintenance by VA to the Point of Coherency instruction can generate a Permission fault.
- If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether a data or unified cache clean by VA to the Point of Unification instruction can generate a Permission fault.

The Data Cache Zero instruction, `DC ZVA`, operates as set of stores to each byte within the block being accessed, and therefore it generates a Permission fault if the translation system does not permit writes to these locations.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Translation fault

A Translation fault can be generated at any level of lookup, and the reported fault code identifies the lookup level. A Translation fault is generated if bits[1:0] of a translation table descriptor identify the descriptor as either a Fault encoding or a reserved encoding. For more information, see *VMSAv8-64 translation table format descriptors* on page D5-2593.

In addition, a Translation fault is generated if the input address for a translation either does not map onto an address range of a TTBR_ELx, or the TTBR_ELx range that it maps onto is disabled. In these cases, the fault is reported as a level 0 Translation fault on the translation stage at which the mapping to a region described by a TTBR_ELx failed.

A data or unified cache maintenance by VA instruction can generate a Translation fault, except that:

- If the Point of Coherency is before any level of cache, it is IMPLEMENTATION DEFINED whether a data or unified cache maintenance by VA instruction can generate a Translation fault.
- If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether a data or unified cache clean by VA to the Point of Unification instruction can generate a Translation fault.

It is IMPLEMENTATION DEFINED whether an instruction cache invalidate by VA operation can generate a Translation fault.

The architecture guarantees that any translation table entry that causes a Translation fault is not cached, meaning the TLB never holds such an entry. Therefore, when a Translation fault occurs, the fault handler does not have to perform any TLB maintenance instructions to remove the faulting entry.

If ARMv8.5-E0PD is implemented and enabled, the TCR_ELx.{E0D0, E0D1} fields can prevent unprivileged access to the addresses translated by TTBR0_ELx or TTBR1_ELx. If access is prevented, the fault is reported as a level 0 Translation fault, and should take the same time to generate, whether the address is present in the TLB or not, to mitigate attacks that use fault timing.

Address size fault

An Address size fault can be generated at any level of lookup.

An Address size fault is generated if one of the following has nonzero address bits above the output address size, for the current stage of translation:

- The TTBR_ELx used for the translation.
- A translation table entry.
- The output address of the translation.

For an Address size fault generated because the TTBR_ELx used for the translation has nonzero address bits above the output address size, the reported fault code indicates a fault at level 0. Otherwise, the reported fault code indicates the lookup level at which the fault occurred.

A data or unified cache maintenance by VA instruction can generate an Address size fault, except that:

- If the Point of Coherency is before any level of cache, it is IMPLEMENTATION DEFINED whether a data or unified cache maintenance by VA instruction can generate an Address size fault.
- If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether a data or unified cache clean by VA to the Point of Unification instruction can generate an Address size fault.

It is IMPLEMENTATION DEFINED whether an instruction cache invalidate by VA operation can generate an Address size fault.

The architecture guarantees that any translation table entry that causes an Address size fault is not cached, meaning the TLB never holds such an entry. Therefore, when an Address size fault occurs, the fault handler does not have to perform any TLB maintenance instructions to remove the faulting entry.

For more information on Address size faults, see *Output address size* on page D5-2548.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

External abort on a translation table walk

An External abort on a translation table walk can be either synchronous or asynchronous. An External abort on a translation table walk is reported:

- If the external abort is synchronous, using:
  - A synchronous Instruction Abort exception if the translation table walk is for an instruction fetch.
  - A synchronous Data Abort exception if the translation table walk is for a data access.
- If the External abort is asynchronous, using the SError interrupt exception.

Behavior of External aborts on a translation table walk caused by address translation instructions

The address translation instructions summarized in *Address translation instructions* on page C5-383 require translation table walks. An External abort can occur in the translation table walk. This is reported as follows:

- If the External abort is synchronous, using a synchronous Data Abort exception.
- If the External abort is asynchronous, using the SError interrupt exception.

For more information, see *Synchronous faults generated by address translation instructions* on page D5-2591.

Access flag fault

An Access flag fault can be generated at any level of lookup, and the reported fault code identifies the lookup level. An Access flag fault is generated only if a translation table descriptor with the Access flag bit set to 0 is used.

For more information about the Access flag bit, see *VMSAv8-64 translation table format descriptors* on page D5-2593.

The architecture guarantees that any translation table entry that causes an Access flag fault is not cached, meaning the TLB never holds such an entry. Therefore, when an Access flag fault occurs, the fault handler does not have to execute any TLB maintenance instructions to remove the faulting entry.

Whether any cache maintenance by VA instructions can generate Access flag faults is IMPLEMENTATION DEFINED.

For more information, see *The Access flag* on page D5-2618.

D5.8.2 The MMU fault-checking sequence

This section describes the MMU checks made for the memory accesses required for instruction fetches and for explicit memory accesses:

- If an instruction fetch faults, it generates an Instruction Abort.
- If a data memory access faults, it generates a Data Abort.

MMU fault checking is performed for each stage of address translation.

The fault-checking sequence shows a translation from an Input address to an Output address. For more information about this terminology, see *About address translation and supported input address ranges* on page D5-2544.

Note

The descriptions in this section do not include the possibility that the attempted address translation generates a TLB conflict abort, as described in *TLB conflict aborts* on page D5-2667.

Types of MMU faults on page D5-2652 describes the faults that an MMU fault-checking sequence can report.

Figure D5-18 on page D5-2656 shows the process of fetching a descriptor from the translation table. For the top-level fetch for any translation, the descriptor is fetched only if the input address passes any required alignment check. As the figure shows, if the translation is stage 1 of the Secure or Non-secure EL1&0 translation regime, when EL2 is enabled, then the descriptor address is in the IPA address space, and is subject to a stage 2 translation to obtain the required PA. This stage 2 translation requires a recursive entry to the fault checking sequence.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

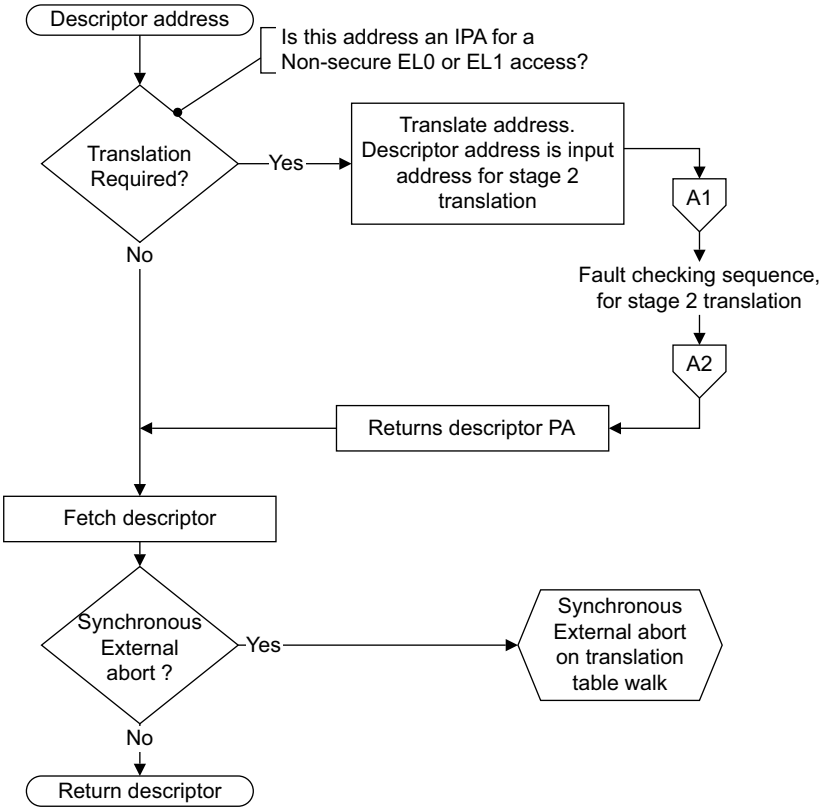


Figure D5-18 Fetching the descriptor in a VMSAv8-64 translation table walk

Figure D5-19 on page D5-2657 shows the full VMSA fault checking sequence, including the alignment check on the initial access.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

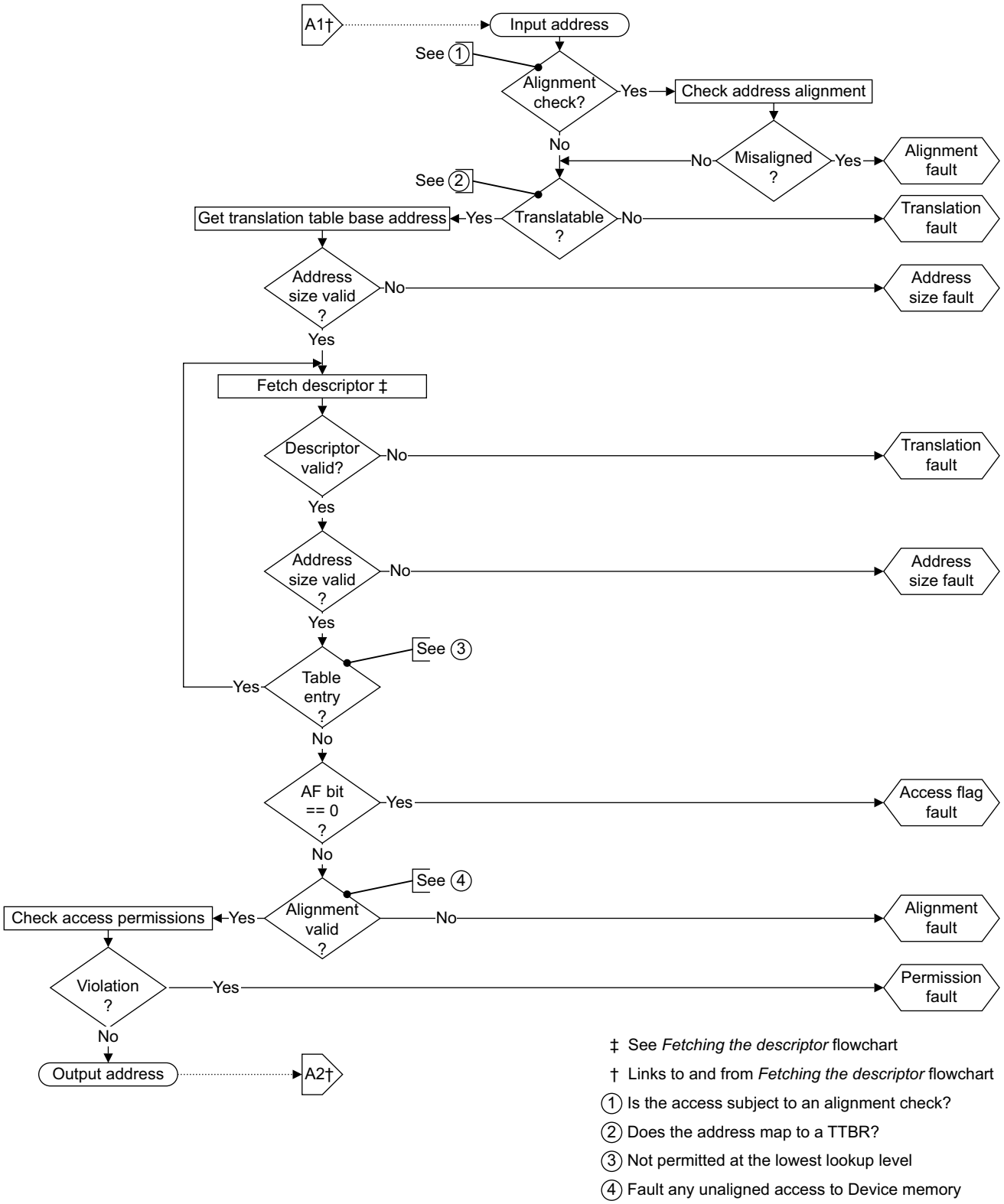


Figure D5-19 VMSAv8-64 fault checking sequence

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Stage 2 fault on a stage 1 translation table walk

On performing a translation table walk for the stage 1 translations, the descriptor addresses must be translated from IPA to PA, using a stage 2 translation. This means that a memory access made as part of a stage 1 translation table lookup might generate, on a stage 2 translation:

- A Translation fault, Access flag fault, or Permission fault.
- A synchronous External abort on the memory access.

If SCR_EL3.EA is set to 1, a synchronous External abort is taken to EL3. Otherwise, these faults are reported as stage 2 memory aborts. ESR_EL2.ISS[7] is set to 1, to indicate a stage 2 fault during a stage 1 translation table walk, and the part of the ISS field that might contain details of the instruction is invalid. For more information see *Use of the ESR_EL1, ESR_EL2, and ESR_EL3* on page D1-2342.

Alternatively, a memory access made as part of a stage 1 translation table lookup might target an area of memory with the Device attribute assigned on the stage 2 translation of the address accessed. When the HCR_EL2.PTW bit is set to 1, such an access generates a stage 2 Permission fault.

Note

On most systems, such a mapping to Device memory on the stage 2 translation is likely to indicate a Guest OS error, where the stage 1 translation table is corrupted. Therefore, it is appropriate to trap this access to the hypervisor.

A TLB might hold entries that depend on the effect of HCR_EL2.PTW. Therefore, if HCR_EL2.PTW is changed without changing the current VMID, the TLBs must be invalidated before executing in EL1 or EL0 state.

A cache maintenance instruction executed at EL1 or EL0 can cause a stage 1 translation table walk that might generate a stage 2 Permission fault as described in this section. However:

- If the Point of Coherency is before any level of cache, it is IMPLEMENTATION DEFINED whether a cache maintenance by VA instruction can generate a Permission fault in this way.
- If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether a data or unified cache clean by VA to the Point of Unification instruction can generate a Permission fault in this way.
- It is IMPLEMENTATION DEFINED whether an instruction cache invalidation by VA instruction can generate a Permission fault in this way.

Note

This is an exception to the general rule that a cache maintenance instruction cannot generate a Permission fault.

The level associated with MMU faults

For MMU faults, Table D5-52 shows how the LL bits in the ESR_ELx.STATUS fields encode the lookup level associated with the fault.

Table D5-52 Use of LL bits to encode the lookup level at which the fault occurred

LL bits	Meaning
00	Level 0 of translation or translation table base register.
01	Level 1.
10	Level 2.
11	Level 3. When xFSR.STATUS indicates a Domain fault, this value is reserved.

The lookup level associated with a fault is:

- For a fault generated on a translation table walk, the lookup level of the walk being performed.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- For a Translation fault, the lookup level of the translation table that gave the fault. If a fault occurs because a stage of address translation is disabled, or because the input address is outside the range specified by the appropriate base address register or registers, or because ARMv8.5-E0PD is enabled and prevents access to the translation table, the fault is reported as a level 0 fault.
- For an Access flag fault, the lookup level of the translation table that gave the fault.
- For a Permission fault, including a Permission fault caused by hierarchical permissions, the lookup level of the final level of translation table accessed for the translation. That is, the lookup level of the translation table that returned a Block or Page descriptor.

Also see *Synchronous External aborts from address translation caching structures* on page D5-2661

D5.8.3 AArch64 state prioritization of synchronous aborts from a single stage of address translation

Synchronous exception prioritization for exceptions taken to AArch64 state on page D1-2355 describes the prioritization of exceptions taken to an Exception level that is using AArch64. This section gives additional information about the prioritization of MMU faults from VMSAv8-64 translation regimes.

Note

The priority numbering in this list only shows the relative priorities of aborts from a single stage of address translation in a VMSAv8-64 translation regime. This numbering has no global significance and, for example, does not correlate with the equivalent AArch32 list in *AArch32 state prioritization of synchronous aborts from a single stage of address translation* on page G5-6021.

For a single stage of translation in a VMSAv8-64 translation regime, the following numbered list shows the priority of the possible memory management faults on a memory access. In this list:

- For memory accesses that undergo two stages of translation, the *italic entries show where the faults from the stage 2 translation can occur*. A stage 2 fault within a stage 1 translation table walk follows the same prioritization of faults:
- For synchronous External aborts from translation table walks, see also *Synchronous External aborts from address translation caching structures* on page D5-2661.

The prioritization between the stage 2 permission failure on the stage 1 translation table walk and the stage 1 abort generated by the stage 1 translation table entry is IMPLEMENTATION DEFINED if all the following are true:

- Stage 1 hardware updating of either access or dirty information is enabled.
- A stage 1 translation table entry results in the stage 1 translation table entry having the access or dirty bit updated.
- The stage 1 translation table entry has stage 2 read permission but not stage 2 write permission.
- The stage 1 translation entry generates an abort (which might be one of an address size fault, an alignment fault caused by memory type or a permission fault).

The priority order, from highest priority to lowest priority, is:

1. Alignment fault not caused by memory type. This is possible for a stage 1 translation only.
2. Translation fault due to the input address being out of the address range to be translated or requiring a TTBR_ELx that is disabled. This includes VTCR_EL2.SL0 being inconsistent with VTCR_EL2.T0SZ, VSTCR_EL2.SL0 being inconsistent with VSTCR_EL2.T0SZ, or SL0 programmed to a reserved value.
3. Address size fault on a TTBR_ELx caused by either:
   - The check on TCR_EL1.IPS, TCR_EL2.PS, TCR_EL3.PS, or VTCR_EL2.PS.
   - The PA being out of the range implemented.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

4. *Second stage abort on a level 0 memory access of a a stage 1 table walk. When stage 2 address translation is enabled this includes an Address size fault caused by the* PA *being out of the range implemented. This is second stage abort during a first stage translation table walk*.

5. Synchronous parity or ECC error on a level 0 lookup of a translation table walk.

6. Synchronous External abort on a level 0 lookup level of a translation table walk.

7. Translation fault on a level 0 translation table entry.

8. Address size fault a level 0 lookup translation table entry caused by either:
   - The check on TCR_EL1.IPS, TCR_EL2.PS, TCR_EL3.PS, or VTCR_EL2.PS.
   - The output address being out of the range implemented.

9. *Second stage abort on a level 1 memory access of a a stage 1 table walk. When stage 2 address translation is enabled this includes an Address size fault caused by the* PA *being out of the range implemented. This is second stage abort during a first stage translation table walk*.

10. Synchronous parity or ECC error on a level 1 lookup of a translation table walk.

11. Synchronous External abort on a level 1 lookup level of a translation table walk.

12. Translation fault on a level 1 translation table entry.

13. Address size fault on a level 1 lookup translation table entry caused by either:
    - The check on TCR_EL1.IPS, TCR_EL2.PS, TCR_EL3.PS, or VTCR_EL2.PS.
    - The output address being out of the range implemented.

14. *Second stage abort on a level 2 memory access of a a stage 1 table walk. When stage 2 address translation is enabled this includes an Address size fault caused by the* PA *being out of the range implemented*. This is second stage abort during a first stage translation table walk.

15. Synchronous parity or ECC error on a level 2 lookup of a translation table walk.

16. Synchronous External abort on a level 2 lookup level of a translation table walk.

17. Translation fault on a level 2 translation table entry.

18. Address size fault on a level 2 lookup translation table entry caused by either:
    - The check on TCR_EL1.IPS, TCR_EL2.PS, TCR_EL3.PS, or VTCR_EL2.PS.
    - The output address being out of the range implemented.

19. *Second stage abort on a level 3 memory access of a a stage 1 table walk. When stage 2 address translation is enabled this includes an Address size fault caused by the* PA *being out of the range implemented*. This is second stage abort during a first stage translation table walk.

20. Synchronous parity or ECC error on a level 3 lookup of a translation table walk.

21. Synchronous External abort on a level 3 lookup level of a translation table walk.

22. Translation fault on a level 3 translation table entry.

23. Address size fault on a level 3 lookup translation table entry caused by either:
    - The check on TCR_EL1.IPS, TCR_EL2.PS, TCR_EL3.PS, or VTCR_EL2.PS.
    - The output address being out of the range implemented.

24. Access Flag fault.

25. Alignment fault caused by the memory type.

26. Permission fault.

27. *A fault from the stage 2 translation of the memory access. When stage 2 address translation is enabled this includes an Address size fault caused by the* PA *being out of the range implemented*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

28. Synchronous parity or ECC error on the memory access.

29. Synchronous External abort on the memory access.

Note

- The prioritization of TLB Conflict aborts is IMPLEMENTATION DEFINED, as the exact cause of these aborts depends on the form of TLBs implemented. However, the TLB conflict abort must have higher priority than any abort that depends on a value held in the TLB.
- The prioritization of IMPLEMENTATION DEFINED MMU faults for a Load-Exclusive or Store-Exclusive to an unsupported memory type is IMPLEMENTATION DEFINED.

Synchronous External aborts from address translation caching structures

A caching structure used for caching translation table walks might support:

- An arbitrary number of levels of translation table lookup.
- One or more stages of translation, that might not correspond to the stages of an address translation lookup.

This might mean that, on a synchronous External aborts arising from the caching structure, including parity or ECC errors, the PE cannot precisely determine one or both of the translation stage and level of lookup at which the error occurred. In this case:

- If the PE cannot determine precisely the translation stage at which the error occurred, it is reported and prioritized as a stage 1 error.
- If the PE cannot determine precisely the lookup level at which the error occurred, the level is reported and prioritized as either:
  - The lowest-numbered level that could have given rise to the error.
  - Level 0 if it the PE cannot determine any information about the level.

D5.8.4 Pseudocode description of the MMU faults

The following functions generate fault records that describe MMU faults:

- `AArch64.AccessFlagFault()`.
- `AArch64.AddressSizeFault()`.
- `AArch64.PermissionFault()`.
- `AArch64.TranslationFault()`.

Abort exceptions on page D4-2529 describes how fault records are used.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D5.9 Translation Lookaside Buffers (TLBs)

Translation Lookaside Buffers (TLBs) reduce the average cost of a memory access by caching the results of translation table walks. TLBs behave as caches of the translation table information, and the VMSA provides TLB maintenance instructions for the management of TLB contents.

Note

The Arm architecture permits TLBs to hold any translation table entry that does not directly cause a Translation fault, an Address size fault, or an Access flag fault.

The following sections describe the architectural requirements for Translation Lookaside Buffers (TLBs) and their maintenance:

- *Use of ASIDs and VMIDs to reduce TLB maintenance requirements*.
- *About Armv8 Translation Lookaside Buffers (TLBs)* on page D5-2664.
- *TLB maintenance requirements and the TLB maintenance instructions* on page D5-2668.

In these descriptions, TLB entries for a translation regime for a particular Exception level are *out of context* when executing at a higher Exception level.

Note

In addition to the functions described in this section, the TLB might cache information from control registers that are described as being "permitted to be cached in a TLB", even when any or all of the stages of translation are disabled. This caching of information gives rise to the maintenance requirements described in *General TLB maintenance requirements* on page D5-2668.

D5.9.1 Use of ASIDs and VMIDs to reduce TLB maintenance requirements

To reduce the need for TLB maintenance on context switches, the lookups from some translation regimes can be associated with an ASID, or with an ASID and a VMID, as follows:

ASID For stage 1 of a translation regime that can support two VA ranges the VMSA can distinguish between *Global pages* and *Process-specific pages*. The *Address Space Identifier* (ASID) identifies pages associated with a specific process and provides a mechanism for changing process-specific tables without having to maintain the TLB structures.

For these stage 1 translations, each of TTBR0_ELx and TTBR1_ELx has a valid ASID field, and TCR_ELx.A1 determines which of these holds the current ASID.

Note

The selected ASID applies regardless of which set of translation tables are used. For example, when the value of TCR_ELx.A1 is 0, any translation table lookup using this stage of translation is associated with the ASID from TTBR0_ELx.ASID, regardless of whether the translation lookup uses TTBR0_ELx or TTBR1_ELx.

See also *ASID size* on page D5-2663 and *Global and process-specific translation table entries* on page D5-2665.

For a symmetric multiprocessor cluster where a single operating system is running on the set of processing elements, the Arm architecture requires all ASID values to be assigned uniquely within any single Inner Shareable domain. In other words, each ASID value must have the same meaning to all processing elements in the system.

VMID For the Secure or Non-secure EL1&0 translation regime, when EL2 is enabled, the *virtual machine identifier* (VMID) identifies the current virtual machine, with its own independent ASID space. The TLB entries include this VMID information, meaning TLBs do not require explicit invalidation when changing from one virtual machine to another if the virtual machines have different VMIDs.

VTTBR_EL2.VMID holds the current VMID.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Common not private translations

In an implementation that includes ARMv8.2-TTCNP, multiple PEs in the same Inner Shareable domain can use the same translation table entries for a given stage of translation in a particular translation regime. This sharing is enabled by the TTBR_ELx.CnP field for the stage of address translation.

For all PEs that are sharing translation table entries for a stage of translation, all system registers bits which apply to that stage of translation and which are described as being permitted to be cached in a TLB must be the same for all the PEs that are sharing the translation table entry. If this condition is not met by hardware, then it is CONSTRAINED UNPREDICTABLE whether the value of such a control bit which has a different value between PEs, interpreted by a PE, PE1, takes the value configured for:

- The system register bit of PE1.
- The system register bit of one of the PEs that is sharing the translation table entry.

When the value of a TTBR_ELx.CnP field is 1, translation table entries pointed to by that TTBR_ELx are shared with all other PEs in the Inner Shareable domain for which for which the following conditions are met:

- The corresponding TTBR_ELx.CnP field has the value 1.
- That TTBR_ELx relates to the same translation regime.

  Note

  — For TTBR0_EL1 the current Security state determines whether the register relates to the Secure EL1&0, when EL2 is disabled, translation regime, or to the Non-secure EL1&0, when EL2 is enabled, translation regime.

  — For TTBR0_EL2 the value of HCR_EL2.E2H determines whether the register relates to the EL2 translation regime, or to the EL2&0 translation regime.

- If an ASID applies to the stage of translation corresponding to that TTBR_ELx then the current ASID value must be the same for all of the PEs that are sharing entries for any translation table entry that is not global or leaf level.
- If a VMID applies to the stage of translation corresponding to that TTBR_ELx then the current VMID value must be the same for all of the PEs that are sharing entries.

For a translation regime with both stage 1 and stage 2 translations, where a TLB holds only stage 1 translation tables or where a TLB combines information from stage 1 and stage 2 translation table entries into a single entry, this entry can be shared between different PEs only if the value of the TTBR_ELx.CnP bit is 1 for both stage 1 and stage 2 of the translation table walk.

The TTBR_ELx.CnP bit can be cached in a TLB.

For a given TTBR_ELx, if the value of TTBR_ELx.CnP is 1 on multiple PEs in the same Inner Shareable domain, and those PEs meet the other conditions for sharing translation table entries as defined in this section, but those TTBR_ELxs do not point to the same translation table entries, then the system is misconfigured, and performing an address translation using that TTBR_ELx:

- Might generate multiple hits in the TLB, and as a result generate an exception that is reported using the TLB conflict fault code, see *TLB conflict aborts* on page D5-2667.
- Otherwise, has a CONSTRAINED UNPREDICTABLE result, as described in *CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values* on page K1-7856.

ASID size

In VMSAv8-64, the ASID size is an IMPLEMENTATION DEFINED choice of 8 bits or 16 bits, and ID_AA64MMFR0_EL1.ASIDBits identifies the supported size.

When an implementation supports a 16-bit ASID, TCR_ELx.AS selects whether the top 8 bits of the ASID are used.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When the value of TCR_ELx.AS is 0, ASID[15:8]:

- Are ignored by hardware for every purpose other than direct reads of TTBR0_ELx.ASID and TTBR1_ELx.ASID.
- Are treated as if they are all zeros when used for allocating and matching entries in the TLB.

Note

VMSAv8-32 uses an 8-bit ASID. For backwards compatibility, when executing using translations controlled from an Exception level that is using AArch32, the ASID size remains at 8 bits. If the implementation supports 16-bit ASIDs, the 8-bit ASID used is zero-extended to 16 bits.

VMID size

From Armv8.1, the VMID size is an IMPLEMENTATION DEFINED choice of 8 bits or 16 bits, and ID_AA64MMFR1_EL1.VMIDBits identifies the supported size.

When ARMv8.1-VMID16 is implemented, VTTBR_EL2[63:48] contains the 16-bit VMID.

When an implementation supports a 16-bit VMID, VTCR_EL2.VS selects whether the top 8 bits of the VMID are used.

When the value of VTCR_EL2.VS is 0, VMID[63:56]:

- Are ignored by hardware for every purpose other than reads of ID_AA64MMFR1_EL1.
- Are treated as if they are all zeros when used for allocating and matching entries in the TLB.

ARMv8.1-VMID16 is only supported when EL2 is using AArch64.

D5.9.2 About Armv8 Translation Lookaside Buffers (TLBs)

Translation Lookaside Buffers (TLBs) are an implementation technique that caches translations or translation table entries. TLBs avoid the requirement for every memory access to perform a translation table walk in memory. The Arm architecture does not specify the exact form of the TLB structures for any design. In a similar way to the requirements for caches, the architecture only defines certain principles for TLBs:

- The architecture has a concept of an entry locked down in the TLB. The method by which lockdown is achieved is IMPLEMENTATION DEFINED, and an implementation might not support lockdown.
- The architecture does not guarantee that an unlocked TLB entry remains in the TLB.
- The architecture guarantees that a locked TLB entry remains in the TLB. However, a locked TLB entry might be updated by subsequent updates to the translation tables. Therefore, when a change is made to the translation tables, the architecture does not guarantee that a locked TLB entry remains incoherent with an entry in the translation table.
- The architecture guarantees that a translation table entry that generates a Translation fault, an Address size fault, or an Access flag fault is not held in the TLB. However a translation table entry that generates a Permission fault might be held in the TLB.
- When address translation is enabled, any translation table entry that does not generate a Translation fault, an Address size fault, or an Access flag fault and is not from a translation regime for an Exception level that is lower than the current Exception level can be allocated to a TLB at any time. The only translation table entries guaranteed not to be held in a TLB are those that generate a Translation fault, an Address size fault, or an Access flag fault.

  Note

  A TLB can hold a translation table entry that does not itself generate a Translation fault but that points to a subsequent table in the translation table walk. This is referred to as *intermediate caching* of TLB entries.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Software can rely on the fact that between disabling and re-enabling a stage of address translation, entries in the TLB relating to that stage of translation have not have been corrupted to give incorrect translations.

The following sections give more information about TLB implementation:

- *Global and process-specific translation table entries*.
- *TLB matching*.
- *TLB behavior at reset* on page D5-2666.
- *TLB lockdown* on page D5-2666.
- *TLB conflict aborts* on page D5-2667.

See also *TLB maintenance requirements and the TLB maintenance instructions* on page D5-2668.

Global and process-specific translation table entries

In a VMSA implementation, system software can divide the virtual memory map used by a stage of translation that can support two VA ranges into global and non-global regions, indicated by the nG bit in the translation table descriptors:

nG == 0 The translation is global, meaning the region is available for all processes.

nG == 1 The translation is non-global, or process-specific, meaning it relates to the current ASID, as defined by:

- TTBR0_ELx.ASID, if the value of TCR_ELx.A1 is 0.
- TTBR1_ELx.ASID, if the value of TCR_ELx.A1 is 1.

As indicated by the nG field definitions, each non-global region has an associated ASID. These identifiers mean different translation table mappings can co-exist in a caching structure such as a TLB. This means that software can create a new mapping of a non-global memory region without removing previous mappings.

Note

- The selected ASID applies to the translation of any address for which the value of the nG bit is 1, regardless of whether the address is translated based on TTBR0_ELx or on TTBR1_ELx.
- In an implementation that does not include ARMv8.1-VHE, the only stage of translation that can support two VA ranges is stage 1 of the EL1&0 translation regime. In an implementation that includes ARMv8.1-VHE stage 1 of the EL2&0 translation regime also can support two VA ranges.

ASIDs are supported only when stage 1 translations can support two VA ranges. Stage 2 translations, and stage 1 translations that can support only a single VA range do not support ASIDs, and all descriptors in these regimes are treated as global.

In a translation regime that supports global and non-global translations, translation table entries from lookup levels other than the final level of lookup are treated as being non-global, regardless of the value of the nG bit.

When a PE is using the VMSAv8-64 translation table format, and is in Secure state, a translation must be treated as non-global, regardless of the value of the nG bit, if NSTable is set to 1 at any level of the translation table walk.

For more information, see *Control of Secure or Non-secure memory access* on page D5-2605.

TLB matching

A TLB is a hardware caching structure for translation table information. Like other hardware caching structures, it is mostly invisible to software. However, there are some situations where it can become visible. These are associated with coherency problems caused by an update to the translation table that has not been reflected in the TLB. Use of the TLB maintenance instructions described in *TLB maintenance requirements and the TLB maintenance instructions* on page D5-2668 can prevent any TLB incoherency becoming a problem.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A particular case where the presence of the TLB can become visible is if the translation table entries that are in use under a particular ASID and VMID are changed without suitable invalidation of the TLB. This can occur only if the architecturally-required break-before-make sequence described in *Using break-before-make when updating translation table entries* on page D5-2669 is not used. If the break-before make sequence is not used, the TLB can hold two mappings for the same address, and this:

- Might generate an exception that is reported using the TLB conflict fault code, see *TLB conflict aborts* on page D5-2667.
- Might lead to CONSTRAINED UNPREDICTABLE behavior. In this case, behavior will be consistent with one of the mappings held in the TLB, or with some amalgamation of the values held in the TLB, but cannot give access to regions of memory with permissions or attributes that could not be assigned by valid translation table entries in the translation regime being used for access. In addition, where all the entries being amalgamated come from Non-secure memory, the amalgamation cannot give rise to an output address that accesses Secure memory. For more information, see *CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values* on page K1-7856.

TLB behavior at reset

The Arm architecture does not require a reset to invalidate the TLBs. The architecture recognizes that an implementation might require caches, including TLBs, to maintain their contents over a system reset. Possible reasons for doing so include power management and debug requirements.

Therefore, for Armv8:

- All TLBs reset to an IMPLEMENTATION DEFINED state that might be UNKNOWN.
- All TLBs are disabled from reset. All stages of address translation are disabled from reset, and the contents of the TLBs have no effect on address translation. For more information, see *Controlling address translation stages* on page D5-2545.
- An implementation can require the use of a specific TLB invalidation routine, to invalidate the TLB arrays before they are enabled after a reset. The exact form of this routine is IMPLEMENTATION DEFINED, but if an invalidation routine is required it must be documented clearly as part of the documentation of the device.

  Arm recommends that if an invalidation routine is required for this purpose, the routine is based on the TLB maintenance instructions described in *TLB maintenance instructions* on page D5-2671.

Similar rules apply to cache behavior, see *Behavior of caches at reset* on page D4-2505.

TLB lockdown

The Arm architecture recognizes that any TLB lockdown scheme is heavily dependent on the microarchitecture, making it inappropriate to define a common mechanism across all implementations. This means that:

- VMSAv8-64 does not require TLB lockdown support.
- If TLB lockdown support is implemented, the lockdown mechanism is IMPLEMENTATION DEFINED. However, key properties of the interaction of lockdown with the architecture must be documented as part of the implementation documentation.

This means that a region of the System instruction encoding space is reserved for IMPLEMENTATION DEFINED functions, see *Reserved encodings for IMPLEMENTATION DEFINED registers* on page D12-2866. An implementation might use some of these encodings to implement TLB lockdown functions. These functions might include:

- Unlock all locked TLB entries.
- Preload into a specific level of TLB. This is beyond the scope of the `PLI` and `PLD` hint instructions.

In an implementation that includes EL2, exceptions generated as a result of TLB lockdown when executing in EL1 or EL0 state can be routed to either:

- EL1, as a Data Abort exception.
- EL2, as a Hyp Trap exception.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For more information, see *Traps to EL2 of EL0 and EL1 accesses to lockdown, DMA, and TCM operations* on page D1-2385.

TLB conflict aborts

If an address matches multiple entries in the TLB, it is IMPLEMENTATION DEFINED whether a TLB conflict abort is generated.

Note

An address can hit multiple entries in the TLB if the TLB has been invalidated inappropriately, for example if TLB invalidation required by the architecture has not been performed.

An implementation can generate TLB conflict aborts on either or both instruction fetches and data accesses. A TLB conflict abort:

- On an instruction fetch is reported as an Instruction Abort, see *ISS encoding for an exception from an Instruction Abort* on page D13-2990.
- On a data access is reported as a Data Abort, see *ISS encoding for an exception from a Data Abort* on page D13-3033.

Armv8 defines the Fault status encoding of `0b110000` for TLB conflict aborts. On a TLB conflict abort, the returned syndrome includes the address that generated the fault. That is, it includes the address that was being looked up in the TLB.

It is IMPLEMENTATION DEFINED whether a TLB conflict abort is a stage 1 abort or a stage 2 abort.

Note

A stage 2 abort cannot be generated if stage 2 of the Secure or Non-secure EL1&0, when EL2 is enabled, translation regime is disabled.

The priority of the TLB conflict abort is IMPLEMENTATION DEFINED, because it depends on the form of a TLB that can generate the abort. However, the TLB conflict abort must have higher priority than any abort that depends on a value held in the TLB.

If an address matches multiple entries in the TLB and no TLB conflict abort is generated, the resulting behavior is CONSTRAINED UNPREDICTABLE, see *CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values* on page K1-7856. The CONSTRAINED UNPREDICTABLE behavior must not permit access to regions of memory with permissions or attributes that mean they cannot be accessed in the current Security state at the current Exception level.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D5.10 TLB maintenance requirements and the TLB maintenance instructions

Translation Lookaside Buffers (TLBs) are an implementation mechanism that caches translations or translation table entries. The Arm architecture does not specify the form of any TLB structures, but defines the mechanisms by which TLBs can be maintained. The following sections describe the VMSA TLB maintenance instructions:

- *General TLB maintenance requirements*.
- *TLB maintenance instructions* on page D5-2671.

See also *Atomicity of register changes on changing virtual machine* on page D5-2553.

D5.10.1 General TLB maintenance requirements

TLB maintenance instructions provide a mechanism for invalidating entries from TLB caching structures to ensure that changes to the translation tables are reflected correctly in those TLB caching structures.

The architecture permits the caching of any translation table entry that has been returned from memory without a fault, provided that the entry does not, itself, cause a Translation fault, an Address size fault, or an Access Flag fault. This means that the entries that can be cached include:

- Entries in translation tables that point to subsequent tables to be used in that stage of translation.
- Stage 2 translation table entries used as part of a stage 1 translation table walk
- Stage 2 translation table entries used to translate the output address of the stage 1 translation.

Such entries might be held in intermediate TLB caching structures that are used during a translation table walk and that are distinct from the data caches in that they are not required to be invalidated as the result of writes of the data. The architecture makes no restriction of the form of these intermediate TLB caching structures.

The architecture does not intend to restrict the form of TLB caching structures used for holding translation table entries, and in particular for a translation regime that involves two stages of translation, it is recognized that such caching structures might contain:

- Entries containing information from stage 1 translation table entries, at any level of the translation table walk.
- Entries containing information from stage 2 translation table entries, at any level of the translation table walk.
- Entries that combine information from stage 1 and stage 2 translation table entries, at any level of the translation table walk.

Note

For the purpose of TLB maintenance, the term *TLB entry* denotes any structure, including temporary working registers in translation table walk hardware, that holds a translation table entry.

Where a TLB maintenance instruction is:

- Required to apply to stage 1 entries, then it must apply to any cached entries in caching structures that include any stage 1 information that are used to translate the address being invalidated.

  Note

  — Where stage 1 information has been cached in multiple TLB entries, as could occur from splintering a page when caching in the TLB, then the invalidation must apply to each cached entry containing stage 1 information from the page that is used to translate the address being invalidated, regardless of whether or not that cached entry would be used to translate the address being invalidated.

  — As stated in *Global and process-specific translation table entries* on page D5-2665, translation table entries from levels of translation other than the final level are treated as being non-global. Arm expects that, in at least some implementations, cached copies of levels of the translation table walk other than the last level are tagged with their ASID, regardless of whether the final level is global. This means that TLB invalidations that involve the ASID require the ASID to match such entries to perform the required invalidation.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Required to apply to stage 2 entries only, then:
  - It is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.
  - It must apply to caching structures that contain information only from stage 2 translation table entries.
- Required to apply to both stage 1 and stage 2 entries, then it must apply to any entry in the caching structures that includes information from either a stage 1 translation table entry or a stage 2 translation table entry, including any entry that combines information from both stage 1 and stage 2 translation table entries.

Whenever translation tables entries associated with a particular VMID or ASID are changed, the corresponding entries must be invalidated from the TLB to ensure that these changes are visible to subsequent execution, including speculative execution, that uses the changed translation table entries.

Some System register field descriptions state that the effect of the field is permitted to be cached in a TLB. This means that all TLB entries that might be affected by a change of the field must be invalidated whenever that field is changed, to ensure that the effect of the change of that control field is visible to subsequent execution, including speculative execution, that uses that control field. This invalidation is required in addition to, and after, the normal synchronization of the System registers described in *Synchronization requirements for AArch64 System registers* on page D13-2869, and applies to any stage of address translation that is implemented for the translation regime, and VMID and ASID as appropriate, that is affected by that control field. A control field that is permitted to be cached in a TLB requires this maintenance even when all stages of address translation are disabled.

In addition to any TLB maintenance requirement, when changing the cacheability attributes of an area of memory, software must ensure that any cached copies of affected locations are removed from the caches. For more information see *Cache maintenance requirement created by changing translation table attributes* on page D5-2687.

Because a TLB never holds any translation table entry that generates a Translation fault, an Address size fault, or an Access Flag fault, a change from a translation table entry that causes a Translation, Address size, or Access flag fault to one that does not fault, does not require any TLB invalidation. However, a *Context synchronization event* is required to ensure that instruction fetches are affected by a completed change to translation table entries that, before the change, generated a Translation, Address size, or Access flag fault.

Special considerations apply to translation table updates that change the memory type, cacheability, or output address of an entry, see *Using break-before-make when updating translation table entries*.

Using break-before-make when updating translation table entries

To avoid possibly creating multiple TLB entries for the same address, and to avoid the effects of TLB caching possibly breaking coherency, single-copy atomicity properties, ordering guarantees or uniprocessor semantics, or possibly failing to clear the Exclusives monitors, the architecture requires the use of a break-before-make sequence when changing translation table entries whenever multiple threads of execution can use the same translation tables and the change to the translation table entries involves any of:

- A change of the memory type.
- A change of the cacheability attributes.
- A change of the output address (OA), if the OA of at least one of the old translation table entry and the new translation table entry is writable, including if the DBM bit is set and hardware updates to the dirty bits are enabled.
- A change to the size of block used by the translation system. This applies both:
  - When changing from a smaller size to a larger size, for example by replacing a table mapping with a block mapping in a stage 2 translation table.
  - When changing from a larger size to a smaller size, for example by replacing a block mapping with a table mapping in a stage 2 translation table.
- A change of the output address (OA), if the contents of memory at the new OA do not match the contents of memory at the previous OA.
- Creating a global entry when there might be non-global entries in a TLB that overlap with that global entry.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A break-before-make sequence on changing from an old translation table entry to a new translation table entry requires the following steps:

1. Replace the old translation table entry with an invalid entry, and execute a `DSB` instruction.
2. Invalidate the translation table entry with a broadcast TLB invalidation instruction, and execute a `DSB` instruction to ensure the completion of that invalidation.
3. Write the new translation table entry, and execute a `DSB` instruction to ensure that the new entry is visible.

This sequence ensures that at no time are both the old and new entries simultaneously visible to different threads of execution, and therefore the problems described at the start of this subsection cannot arise.

In Armv8.1, with the introduction of hardware updates to the translation table entries, the effects of not following the break-before-make rules are extended.

If the break-before-make rules are not followed for changing the translation table entries, the Armv8.1 architecture permits that the following failures associated with the hardware updates of the translation table entries could occur:

- The Access flag is not set on such a translation table entry despite the fact that the memory location associated with that entry was accessed.
- The AP[2] or S2AP[1] bit is modified by the hardware on such a translation table entry despite the fact that the memory location associated with that entry was not written to.
- The AP[2] or S2AP[1] bit is not modified by the hardware on such a translation table entry despite the fact that the memory location associated with that entry was written to.
- The ordering required between hardware updates to such a translation table entry and stores appearing later in program order is not followed.

Support levels for changing block size

If ARMv8.4-TTRem is implemented, the PE provides three levels of support when changing block size, without changing any other parameters that require break-before-make:

Level 0 Software must use break-before-make to avoid breaking coherency, ordering guarantees or uniprocessor semantics, or failing to clear the Exclusives monitors when changing block size. See *Using break-before-make when updating translation table entries* on page D5-2669.

Level 1 Software can use the level 0 approach, or software can use the nT block translation entry to avoid breaking coherency, ordering guarantees or uniprocessor semantics, or failing to clear the Exclusives monitors when changing block size. See *Block translation entry* on page D5-2618.

Level 2 Software can use the level 0 or level 1 approach and, in addition, changing block size does not break coherency, ordering guarantees or uniprocessor semantics, or fail to clear the Exclusives monitors. If there has not been a TLB invalidation of the entries that have changed since the writes that changed those entries were completed, this change might cause Conflict aborts. This is because multiple translation entries might exist within the TLB for the same input address.

In addition, an implementation that uses the level 1 or level 2 approach supports the following without breaking coherency, ordering guarantees or uniprocessor semantics, or failing to clear the Exclusives monitors:

- A change to a set of blocks or pages from having the Contiguous bit set to having the Contiguous bit not set.
- A change to a set of blocks or pages from having the Contiguous bit not set to having the Contiguous bit set.

If multiple translation entries exist within the TLB for the same input address, this change might cause Conflict aborts when translating the address.

If a Conflict abort is detected in a translation regime for which stage 2 translations are enabled, the Conflict abort is reported to EL2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Clearing entries associated with a Conflict abort

While using level 1 or level 2 support, on a Conflict abort, the following instructions are guaranteed to clear the entries associated with the conflict:

- For the EL1&0 Translation regime, while stage 2 translations are in use: `TLBI VMALLS12E1`, `TLBI ALLE1`.
- For the EL1&0 Translation regime, while stage 2 translations are not in use: `TLBI VMALLE1`, `TLBI ALLE1`.
- For the EL2&0 Translation regime: `TLBI VMALLE1`, `TLBI ALLE1`.
- For the EL2 Translation regime: `TLBI ALLE2`.
- For the EL3 Translation regime: `TLBI ALLE3`.

D5.10.2 TLB maintenance instructions

The architecture defines TLB maintenance instructions, that provide the following:

- Invalidate all entries in the TLB.
- Invalidate a single TLB entry by ASID for a non-global entry.
- Invalidate all TLB entries that match a specified ASID.
- Invalidate all TLB entries that match a specified VA, regardless of the ASID.
- Invalidate all TLB entries within a range of addresses.

Each instruction can be specified as applying only to the PE that executes the instruction, or as applying to all PEs in the same shareability domain as the PE that executes the instruction.

The following subsubsections describe these instructions:

- *TLB maintenance instruction syntax*.
- *Operation of the TLB maintenance instructions* on page D5-2675.
- *Scope of the A64 TLB maintenance instructions* on page D5-2676.
- *TLB range maintenance instructions* on page D5-2679.
- *Invalidation of TLB entries from stage 2 translations* on page D5-2680.
- *Broadcast TLB maintenance between AArch32 and AArch64* on page D5-2681.
- *Broadcast TLB maintenance with different translation granule sizes* on page D5-2682.
- *Ordering and completion of TLB maintenance instructions* on page D5-2683.
- *TLB maintenance in the event of TLB conflict* on page D5-2683.
- *The interaction of TLB lockdown with TLB maintenance instructions* on page D5-2684.

TLB maintenance instructions on page C5-383 describes the encoding of the TLB maintenance instructions.

TLB maintenance instruction syntax

The A64 syntax for TLB maintenance instructions is:

```
TLBI <operation>{, <Xt>}
```

Where:

`<operation>` Is one of `ALLE1`, `ALLE2`, `ALLE3`, `ALLE1IS`, `ALLE2IS`, `ALLE3IS`, `ALLE10S`, `ALLE20S`, `ALLE30S`, `VMALLE1`, `VMALLE1IS`, `VMALLE10S`, `VMALLS12E1`, `VMALLS12E1IS`, `VMALLS12E10S`, `ASIDE1`, `ASIDE1IS`, `ASIDE10S`, `{R}VA{L}E1`, `{R}VA{L}E2`, `{R}VA{L}E3`, `{R}VA{L}E1IS`, `{R}VA{L}E2IS`, `{R}VA{L}E3IS`, `{R}VA{L}E10S`, `{R}VA{L}E20S`, `{R}VA{L}E30S`, `{R}VAA{L}E1`, `{R}VAA{L}E1IS`, `{R}VAA{L}E10S`, `{R}IPAS2{L}E1`, `{R}IPAS2{L}E1IS`, or `{R}IPAS2{L}E10S`.

`<operation>` has a structure of `{R}<type><level><shareability>` where:

`R` When present, indicates that the function applies to all TLBs that are within a determined address range, see *TLB range maintenance instructions* on page D5-2679. When not present, indicates that the function applies to all TLBs at a single address that contain entries that could be used by the PE that executes the TLBI instruction.

`<type>` Is one of:

`ALL` All translations used at `<level>`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the scope of `ALL` instructions see *ALL* on page D5-2676.

The `ALL` instructions are valid for all values of `<level>`.

`VMALL` All stage 1 translations used at <level> with the current VMID, if appropriate.

For the scope of the `VMALL` instructions see *VMALL* on page D5-2676.

The `VMALL` instructions are valid only when `level` == `E1`.

`VMALLS12` All stage 1 and stage 2 translations used at EL1 with the current VMID, if appropriate.

For the scope of the `VMALLS12` instructions see *VMALLS12* on page D5-2676.

The `VMALLS12` instructions are valid only when `level` == `E1`.

`ASID` All translations used at EL1 with the supplied ASID.

For the scope of the `ASID` instructions see *ASID* on page D5-2677.

The `ASID` instructions are valid only when `level` == `E1`.

`VA{L}` Translations used at `<level>` for the specified address and, if appropriate, the specified ASID.

For the scope of the `VA` instructions see *VA* on page D5-2677. For the scope of the `VAL` instructions see *VAL* on page D5-2677.

The `VA{L}` instructions are valid for all values of `<level>`.

`VAA{L}` Translations used at `<level>` for the specified address, for all ASID values, if appropriate.

For the scope of the `VAA` instructions see *VAA* on page D5-2678. For the scope of the `VAAL` instructions see *VAAL* on page D5-2678.

The `VAA{L}` instructions are valid only when `level` == `E1`.

`IPAS2{L}` Translations used at `<level>` for the specified IPA that are held in stage 2 only caching structures.

For the scope of the `IPAS2` instructions see *IPAS2* on page D5-2678. For the scope of the `IPAS2L` instructions see *IPAS2L* on page D5-2678

The `IPAS2{L}` instructions are valid only when `level` == `E1`.

In the `VA{L}`, `VAA{L}`, and `IPAS2{L}` types:

`L` An optional parameter that indicates that the invalidation only applies to caching of entries returned from the final lookup level of the translation table walk.

`<level>` Defines the Exception level of the translation regime that the invalidation applies to. Is one of:

`E1` EL1.

`E2` EL2.

`E3` EL3.

An instruction that applies to the translation regime of an Exception level higher than the Exception level at which the instruction is executed is UNDEFINED.

`TLBI ALLE1`, `TLBI ALLE1IS`, `TLBI ALLE10S`, `TLBI {R}IPAS2{L}E1`, `TLBI {R}IPAS2{L}E1IS`, `TLBI {R}IPAS2{L}E10S`, `TLBI VMALLS12E1`, `TLBI VMALLS12E1IS`, and `TLBI VMALLS12E10S` are UNDEFINED at EL1.

Note

All TLB maintenance instructions are UNDEFINED at EL0.

`<shareability>`

Is one of:

`IS` When present, it indicates that the function applies to all TLBs in the Inner Shareable shareability domain.

`OS` When present, it indicates that the function applies to all TLBs in the Outer Shareable shareability domain.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<blank>` When no shareability is present, it indicates that the function applies to all TLBs that contain entries that could be used by the PE that executes the TLBI instruction.

Note

When a TLB entry has been invalidated for one PE, it is not consistent with the architecture to allow another PE to refill that TLB entry where the new entry might give the appearance to software that the invalidation has not occurred.

`<Xt>` Passes one or both of an address and an ASID as an argument, where required. `<Xt>` is required for the `TLB ASID`, `TLB VA{L}`, `TLB VAA{L}`, and `TLB IPAS2{L}` instructions.

If EL2 is not implemented, the `TLBI VA{L}E2`, `TLBI VA{L}E2IS`, `TLBI VA{L}E2OS`, `TLBI ALLE2`, `TLBI ALLE2IS`, `TLBI ALLE2OS`, `TLBI RVA{L}E2`, `TLBI RVA{L}E2IS`, and `TLBI RVA{L}E2OS` instructions are UNDEFINED.

VMSAv8-64 TLB maintenance instructions that take a register argument that holds a VA, an ASID, or both, and that do not apply to a range of addresses, use the register argument format:

Bits[63:48] ASID. These bits are RES0 if the instruction does not require an ASID argument.

Bits[47:44] TTL. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated, see *Translation table level hints* on page D5-2674. This field is RES0 if the instruction does not require an VA argument, or if ARMv8.4-TTL is not implemented.

Bits[43:0] VA[55:12]. For an instruction that requires a VA argument, the treatment of the low-order bits of this field depends on the translation granule size, as follows:

4KB granule size All bits are valid and used for the invalidation.

16KB granule size Bits[1:0] RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.

64KB granule size Bits[3:0] are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

These bits are RES0 if the instruction does not require a VA argument.

For TLB maintenance instructions that take an address argument, hardware interprets VA[63:56] as each having the same value as VA[55].

If a TLB maintenance instruction targets a translation regime that is using AArch32, meaning the VA is only 32-bit, then software must treat VA[55:32] as RES0, and these bits are ignored when the instruction is executed.

If the implementation supports 16 bits of ASID then the upper 8 bits of the ASID are RES0 when the context being invalidated only uses 8 bits.

VMSAv8-64 TLB maintenance instructions that take a register argument that holds an IPA, and that do not apply to a range of addresses, use the register argument format:

Bits[63:48] RES0.

Bits[47:44] TTL. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated, see *Translation table level hints* on page D5-2674. This field is RES0 if the instruction does not require an IPA argument, or if ARMv8.4-TTL is not implemented.

Bits[43:36] RES0.

Bits[35:0] IPA[47:12]. For an instruction that requires a VA argument, the treatment of the low-order bits of this field depends on the translation granule size, as follows:

4KB granule size All bits are valid and used for the invalidation.

16KB granule size Bits[1:0] RES0 and ignored when the instruction is executed, because IPA[13:12] have no effect on the operation of the instruction.

64KB granule size Bits[3:0] are RES0 and ignored when the instruction is executed, because IPA[15:12] have no effect on the operation of the instruction.

For the register argument format of TLB instructions that apply to a range of addresses, see *TLB range maintenance instructions* on page D5-2679.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Translation table level hints

When ARMv8.4-TTL is implemented, the TTL field indicates the level of page table walk holding the leaf entry for the address being invalidated. Hardware can use this information to determine if there was a risk of splintering.

If an incorrect value for the entry being invalidated by the instruction is specified in the TTL field, then no entries are required by the architecture to be invalidated from the TLB.

The TTL field in TLB maintenance instructions that take a register argument that holds a VA or an IPA, and that do not apply to a range of addresses, use the encodings in Table D5-53.

Table D5-53 TTL field encodings in TLB instructions that apply to a single address

TTL[3:2]	TTL[1:0]	Information supplied
00	RES0	No information supplied about the translation level. Hardware must assume that the entry can be from any level.
01	00	The entry comes from a 4KB translation granule. This value is reserved, and hardware should treat this as if TTL[3:2] is `0b00`.
	01	The entry comes from a 4KB translation granule. The leaf entry for the address being invalidated is on level 1 of the page table walk.
	10	The entry comes from a 4KB translation granule. The leaf entry for the address being invalidated is on level 2 of the page table walk.
	11	The entry comes from a 4KB translation granule. The leaf entry for the address being invalidated is on level 3 of the page table walk.
10	00	The entry comes from a 16KB translation granule. This value is reserved, and hardware should treat this as if TTL[3:2] is `0b00`.
	01	The entry comes from a 16KB translation granule. This value is reserved, and hardware should treat this as if TTL[3:2] is `0b00`.
	10	The entry comes from a 16KB translation granule. The leaf entry for the address being invalidated is on level 2 of the page table walk.
	11	The entry comes from a 16KB translation granule. The leaf entry for the address being invalidated is on level 3 of the page table walk.
11	00	The entry comes from a 64KB translation granule. This value is reserved, and hardware should treat this as if TTL[3:2] is `0b00`.
	01	The entry comes from a 64KB translation granule. The leaf entry for the address being invalidated is on level 1 of the page table walk.
	10	The entry comes from a 64KB translation granule. The leaf entry for the address being invalidated is on level 2 of the page table walk.
	11	The entry comes from a 64KB translation granule. The leaf entry for the address being invalidated is on level 3 of the page table walk.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The TTL field in TLB maintenance instructions that take a register argument that holds a VA or an IPA, and that do not apply to a range of addresses, use the encodings in Table D5-54.

Table D5-54 TTL field encodings in TLB instructions that apply to multiple addresses

TTL	Information supplied
00	The entries in the range can be using any level for the translation table entries.
01	When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries. When using a 16KB translation granule, this value is reserved, and hardware should treat the TTL field as `0b00`.
10	All entries to invalidate are Level 2 translation table entries.
11	All entries to invalidate are Level 3 translation table entries.

Operation of the TLB maintenance instructions

Any TLB maintenance instruction can affect any TLB entries that are not locked down.

The TLB maintenance instructions specify the Exception level of the translation regime to which they apply.

Note

Because there is no guarantee that an unlocked TLB entry remains in the cache, architecturally it is not possible to tell whether a TLB maintenance instruction has affected any TLB entries that were not specified by the instruction.

If a TLB maintenance instruction specifies a VA, and a data or instruction access to that VA would generate an MMU abort, the TLB maintenance instruction does not generate an abort. VAs for which a TLB maintenance instruction does not generate an abort include VAs that are not in the range of VAs that can be translated.

When EL3 is implemented:

- The TLB maintenance instructions that apply to the EL1&0 translation regime take account of the current Security state, as part of the address translation required for the TLB operation.
- SCR_EL3.NS modifies the effect of the TLB maintenance instructions as follows:
  - For instructions that apply to the EL1&0 translation regime, the SCR_EL3.NS bit identifies whether the maintenance instructions apply to the Secure or Non-secure EL1&0 translation regime.

    Note

    If EL3 is not implemented, then there is only a single EL1&0 translation regime.

  - When SCR_EL3.EEL2 is 0 instructions that apply to the EL2 translation regime, or to the EL2&0 translation regime, the SCR_EL3.NS bit must be 1 or the instruction is UNDEFINED.
  - For instructions that apply to the EL3 translation regime, the SCR_EL3.NS bit has no effect.

Note

- An address-based TLB maintenance instruction that applies to the Inner Shareable domain or the Outer Shareable domain does so regardless of the Shareability attributes of the address supplied as an argument to the instruction.
- Previous versions of the Arm architecture included TLB maintenance instructions that operated only on instruction TLBs, or only on data TLBs. From the introduction of Armv7, Arm deprecated any use of these instructions. In Armv8:
  - AArch64 state does not include any of these instructions.
  - AArch32 state includes some of these instructions, but Arm deprecates their use.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The Arm architecture does not dictate the form in which the TLB stores translation table entries. However, when a TLB maintenance instruction is executed, the minimum size of the table entry that is invalidated from the TLB must be at least the size that appears in the translation table entry.

Note

The Contiguous bit does not affect the minimum size of entry that must be invalidated from the TLB.

Scope of the A64 TLB maintenance instructions

The TLB invalidation instruction `<type>` affects the different possible cached entries in the TLB as follows:

`ALL`

The invalidation applies to all cached copies of the stage 1 and stage 2 translation table entries from any level of the translation table walk required to translate any address at the specified Exception level, that would be used with the state specified by SCR_EL3.NS and SCR_EL3.EEL2.

For entries from the Secure or Non-secure EL1&0, when EL2 is enabled, translation regime, `ALL` applies to entries with any VMID.

For entries from a translation regime for which an ASID is valid, the invalidation applies to:

- All entries above the final level of lookup.
- All entries at the final level of lookup.

Note

This means the invalidation applies to both:

— Global entries.

— Non-global entries with any ASID.

`VMALL`

The invalidation applies to all cached copies of the stage 1 translation table entries, from any level of the translation table walk required to translate any address at the specified Exception level, that would be used with all of:

- The Security state specified by SCR_EL3.NS and SCR_EL3.EEL2.
- For the Secure or Non-secure EL1&0, when EL2 is enabled, translation regime, the current VMID.

For entries from a translation regime for which an ASID is valid that meet the other specified conditions, the invalidation applies to:

- All entries above the final level of lookup.
- All entries at the final level of lookup.

Note

This means the invalidation applies to both:

— Global entries.

— Non-global entries with any ASID.

`VMALL` is valid for:

- EL1.
- EL2, when HCR_EL2.{E2H, TGE} is {1, 1}.

`VMALLS12`

The invalidation applies to all cached copies of the stage 1 and stage 2 translation table entries from any level of the translation table walk required to translate any address at the specified Exception level, that would be used with all of:

- The Security state specified by SCR_EL3.NS and SCR_EL3.EEL2.
- For the Secure or Non-secure EL1&0, when EL2 is enabled, translation regime, the current VMID.

For entries from a translation regime for which an ASID is valid that meet the other specified conditions, the invalidation applies to:

- All entries above the final level of lookup.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- All entries at the final level of lookup.

  Note

  This means the invalidation applies to both:

  — Global entries.

  — Non-global entries with any ASID.

`VMALLS12` is valid for EL1.

If EL2 is not implemented, or if the `TLBI VMALLS12` instruction is executed when the value of SCR_EL3.NS is 0 and EL2 is disabled, the instruction is not UNDEFINED but it has the same effect as `TLBI VMALL`. This is because there are no stage 2 translations to invalidate.

`ASID`

The invalidation applies to all cached copies of the stage 1 translation table entries from any level of the translation table walk required to translate any address at the specified Exception level, that would be used with all of:

- The Security state specified by SCR_EL3.NS and SCR_EL3.EEL2.
- For the Secure or Non-secure EL1&0, when EL2 is enabled, translation regime, the current VMID.

For entries from a translation regime for which an ASID is valid that meet the other specified conditions, the invalidation applies only if either:

- The entry is from a level of lookup above the final level and matches the specified ASID.
- The entry is a non-global entry from the final level of lookup and matches the specified ASID.

`ASID` is valid for:

- EL1.
- EL2, when HCR_EL2.{E2H, TGE} is {1, 1}.

`VA`

The invalidation applies to all cached copies of the stage 1 translation table entries from any level of the translation table walk required to translate the address specified in the invalidation instruction at the specified Exception level that would be used with all of:

- The Security state specified by SCR_EL3.NS and SCR_EL3.EEL2.
- For the Secure or Non-secure EL1&0, when EL2 is enabled, translation regime, the current VMID.

For entries from a translation regime for which an ASID is valid that meet the other specified conditions, the invalidation applies only if one of the following applies:

- The entry is from a level of lookup above the final level and matches the specified ASID.
- The entry is a global entry from the final level of lookup.
- The entry is a non-global entry from the final level of lookup that matches the specified ASID.

`VAL`

The invalidation applies to all cached copies of the stage 1 translation table entry from the final level of the translation table walk required to translate the address specified in the invalidation instruction at the specified Exception level, that would be used with all of:

- The Security state specified by SCR_EL3.NS and SCR_EL3.EEL2.
- For the Secure or Non-secure EL1&0, when EL2 is enabled, translation regime, the current VMID.

For entries from a translation regime for which an ASID is valid that meet the other specified conditions, the invalidation applies only if either:

- The entry is a global entry from the final level of lookup.
- The entry is a non-global entry from the final level of lookup that matches the specified ASID.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`VAA` The invalidation applies to all cached copies of the stage 1 translation table entries from any level of the translation table walk required to translate the address specified in the invalidation instruction at the specified Exception level that would be used with all of:

- The Security state specified by SCR_EL3.NS and SCR_EL3.EEL2.
- For the Secure or Non-secure EL1&0, when EL2 is enabled, translation regime, the current VMID.

For entries from a translation regime for which an ASID is valid that meet the other specified conditions, the invalidation applies to all of:

- All entries above the final level of lookup.
- All entries at the final level of lookup.

Note

This means the invalidation applies to both:

— Global entries.

— Non-global entries with any ASID.

`VAAL` The invalidation applies to all cached copies of the stage 1 translation table entry from the final level of the translation table walk required to translate the address specified in the invalidation instruction at the specified Exception level that would be used with all of:

- The Security state specified by SCR_EL3.NS and SCR_EL3.EEL2.
- For the Secure or Non-secure EL1&0, when EL2 is enabled, translation regime, the current VMID.

For entries from a translation regime for which an ASID is valid that meet the other specified conditions, the invalidation applies to all entries at the final level of lookup.

Note

This means the invalidation apples to both:

- Global entries.
- Non-global entries with any ASID.

`IPAS2` The invalidation applies to all cached copies of the stage 2 translation table entries from any level of the translation table walk required to translate the specified IPA, that both:

- Are held in TLB caching structures holding stage 2 only entries.
- Would be used with the current VMID.

It is not required that this instruction invalidates TLB caching structures holding entries that combine stage 1 and stage 2 of the translation.

The only translation regime to which this instruction can apply is the Secure or Non-secure EL1&0, when EL2 is enabled, translation regime.

When executed with the SCR_EL3.NS == 0, or in an implementation that does not implement EL2, this instruction is a `NOP`.

For more information about the architectural requirements for the `IPAS2` instruction see *Invalidation of TLB entries from stage 2 translations* on page D5-2680.

`IPAS2L` The invalidation applies to cached copies of the stage 2 translation table entry from the final level of the stage 2 translation table walk required to translate the specified IPA, that both:

- Are held in TLB caching structures holding stage 2 only entries.
- Would be used with the current VMID.

It is not required that this instruction invalidates TLB caching structures holding entries that combine stage 1 and stage 2 of the translation.

The only translation regime to which this instruction can apply is the Secure or Non-secure EL1&0, when EL2 is enabled, translation regime.

When executed with the SCR_EL3.NS == 0, or in an implementation that does not implement EL2, this instruction is a `NOP`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For more information about the architectural requirements for the `IPAS2L` instruction see *Invalidation of TLB entries from stage 2 translations* on page D5-2680.

The entries that the invalidations apply to are not affected by the state of any other control bits involved in the translation process. Therefore, the following is a non-exhaustive list of control bits that do not affect how a TLB maintenance instruction updates the TLB entries:

In AArch64 state

SCTLR_EL1.M, SCTLR_EL2.M, SCTLR_EL3.{M, RW], HCR_EL2.{VM, RW}, TCR_EL1.{TG1, EPD1, T1SZ, TG0, EPD0, T0SZ, AS, A1}, TCR_EL2.{TG0, T0SZ}, TCR_EL3.{TG0, T0SZ}, VTCR_EL2.{SL0, T0SZ}, TTBR0_EL1.ASID, TTBR1_EL1.ASID.

In AArch32 state

SCTLR.M, HCR.VM, TTBCR.{EAE, PD1, PD0, N, EPD1, T1SZ, EPD0, T0SZ, A1}, HTCR.T0SZ, VTCR.{SL0, T0SZ}, TTBR0.ASID, TTBR1.ASID, CONTEXTIDR.ASID.

Note

- Arm expects most TLB maintenance performed by an operating system to occur to the last level entries of the stage 1 translation table walks, and the purpose of the address-based TLB invalidation instructions where the invalidation need only apply to caching of entries returned from the last level of translation table walk of stage 1 translation is to avoid unnecessary loss of the intermediate caching of the translation table entries. Similarly, for stage 2 translations Arm expects that most TLB maintenance performed by a hypervisor for a given Guest operation system will affect only the last level entries of the stage 2 translations. Therefore, similar capability is provided for instructions that invalidate single stage 2 entries.
- The architecture permits the invalidation of entries in TLB caching structures at any time, so for each of these instructions the definition is in terms of the minimum set of entries that must be invalidated from TLB caching structures, and an implementation might choose to invalidate more entries. In general, for best performance, Arm recommends not invalidating entries that are not required to be invalidated.
- Dependencies on the VMID for the Secure or Non-secure EL1&0, when EL2 is enabled, translation regime apply even when the value of HCR_EL2.VM is 0. The VTTBR_EL2.VMID field resets to a value that is architecturally UNKNOWN, and therefore VTTBR_EL2.VMID[7:0] must be set to a known value, that might be zero, as part of the PE initialization sequence, even if stage 2 translation is not in use.

TLB range maintenance instructions

Specific TLB invalidation instructions apply to a range of input addresses rather than a single address. All TLB range maintenance instructions invalidate TLB entries that are within the address range determined by the formula: `[BaseADDR <= input_address < BaseADDR + ((NUM +1)*2^(5*SCALE +1) * Translation_Granule_Size)]`.

Note

The set of masters containing TLBs that can be affected by the TLB range maintenance instructions are defined by the system architecture. This means that all masters in a system might not contain TLBs within the defined shareability domains.

VMSAv8-64 TLB range maintenance instructions that take a register argument that holds a VA, or a VA and an ASID, use the following register argument format:

Bits[63:48] ASID. These bits are RES0 if the instruction does not require an ASID argument.

Bits[47:46] TG. This field gives the translation granule size for the translations that are being invalidated. If the translations use a different translation granule size than the one specified, then the architecture does not require that the instruction invalidates any entries.

Bits[45:44] SCALE. This field gives the exponent element of the calculation that is used to produce the upper range.

Bits[43:39] NUM. This field gives the base element of the calculation that is used to produce the upper range.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bits[38:37] TTL level hint, see *Translation table level hints* on page D5-2674. This field is RES0 if the instruction does not require a VA argument, or if ARMv8.4-TTL is not implemented.

Bits[36:0] BaseADDR. This field gives the starting address for the range of the maintenance instruction.

4KB granule size	BaseADDR[48:12].
16KB granule size	BaseADDR[50:14].
64KB granule size	BaseADDR[52:16].

VMSAv8-64 TLB range maintenance instructions that take a register argument that holds an IPA, use the following register argument format:

Bits[63] NS. This bit is RES0 if the instruction is executed in Non-secure state.

Bits[62:48] RES0.

Bits[47:46] TG. This field gives the translation granule size for the translations that are being invalidated. If the translations use a different translation granule size than the one specified, then the architecture does not require that the instruction invalidates any entries.

Bits[45:44] SCALE. This field gives the exponent element of the calculation that is used to produce the upper range.

Bits[43:39] NUM. This field gives the base element of the calculation that is used to produce the upper range.

Bits[38:37] TTL level hint, see *Translation table level hints* on page D5-2674. This field is RES0 if the instruction does not require a VA argument, or if ARMv8.4-TTL is not implemented.

Bits[36:0] BaseADDR. This field gives the starting address for the range of the maintenance instruction.

4KB granule size	BaseADDR[48:12].
16KB granule size	BaseADDR[50:14].
64KB granule size	BaseADDR[52:16].

The range of addresses invalidated is UNPREDICTABLE when:

- When a 4K translation granule used, if the TTL field is 0b01 and BaseADDR[29:12] does not equal 0b000000000000000000.
- When a 4K translation granule used, if the TTL field is 0b10 and BaseADDR[20:12] does not equal 0b000000000.
- When a 16K translation granule used, if the TTL field is 0b10 and BaseADDR[24:12] does not equal 0b00000000000.
- When a 64K translation granule used, if the TTL field is 0b01 and BaseADDR[41:16] does not equal 0b0000000000000000000000000000.
- When a 64K translation granule used, if the TTL field is 0b10 and BaseADDR[28:16] does not equal 0b0000000000000.

Invalidation of TLB entries from stage 2 translations

The architectural requirements of the `IPAS2` instruction are that:

1. The following code is sufficient to invalidate all cached copies of the stage 2 translation of the IPA held in `Xt` for the current VMID, with the corresponding requirement for the broadcast versions of the instructions:

```
TLBI IPAS2E1, Xt
DSB
TLBI VMALLE1
```

2. The following code is sufficient to invalidate all cached copies of the stage 2 translations of the IPA held in `Xt` used to translate the VA (and the specified ASID when executing `TLBI VAE1`) held in `Xt2`, with the corresponding requirement for the broadcast versions of the instructions:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
TLBI IPAS2E1, Xt
DSB
TLBI VAE1, Xt2     ; or TLBI VAAE1, Xt2
```

3. The following code is sufficient to invalidate all cached copies of the stage 2 translations of the IPA held in `Xt` used to translate the IPA produced by the last level of stage 1 translation table lookup for the VA (and ASID when executing `TLBI VALE1`) held in `Xt2`, with the corresponding requirement for the broadcast versions of the instructions:

```
TLBI IPAS2E1, Xt
DSB
TLBI VALE1, Xt2    ; or TLBI VAALE1, Xt2
```

Note

Depending on the invalidation required, software must use the entire sequence 1, 2, or 3, even when Secure or Non-secure EL1&0, when EL2 is enabled, stage 1 translation is disabled.

Equivalent architectural requirements apply to the `IPAS2L` instruction, except that the only TLB entries that must be invalidated by an `IPAS2L` instruction are those that come from the final level of the translation table lookup.

Broadcast TLB maintenance between AArch32 and AArch64

In most cases, a TLB maintenance instruction affecting the shareability domain executed by a PE in an Exception level that is using AArch64 also affects any other PE in the same shareability domain that is executing at the same Exception level and is using AArch32, provided that the address, qualify the scope of the ASID and VMID matching requirements of the original instruction are met, as specified in *Scope of the A64 TLB maintenance instructions* on page D5-2676.

Note

The requirement to match means that the invalidation only occurs on the PE that is using AArch32 if, for the PE that executed the TLB maintenance instruction at an Exception level that is using AArch64, both of the following apply:

- If VA matching is required, the VA is `0x0000FFFFFFFF` or lower in the memory map.
- If ASID matching is required and the PE is using a 16-bit ASID, then the top 8 bits of the ASID are zero.

Except for the cases identified here, a TLB maintenance instruction affecting the Inner Shareable shareability domain executed by a PE in an Exception level that is using AArch32 also affects any other PE in the same Inner Shareable domain that is executing at the same Exception level and is using AArch64, provided that the address, ASID, and VMID matching requirements of the original instruction are met, as specified in *Scope of the A64 TLB maintenance instructions* on page D5-2676. In addition, for the instruction executed in AArch32 state:

- For a `TLBIMVAAIS`, `TLBIMVAALIS`, `TLBIMVAHIS`, `TLBIMVAIS`, `TLBIMVALHIS`, or `TLBIMVALIS` instruction, the VA supplied as an argument is zero-extended.
- For a `TLBIIPAS2IS` or `TLBIIPAS2LIS` instruction, the IPA supplied as an argument is zero-extended.
- For a `TLBIASIDIS`, `TLBIMVAIS`, or `TLBIMVALIS` instruction, the ASID supplied as an argument is zero-extended if the PE executing in AArch64 state is using a 16-bit ASID.

The VA from the instruction executed in AArch32 state is zero-extended, and the ASID is zero-extended if the PE executing in AArch64 state is using a 16-bit ASID.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The exceptions to these general rules are as follows:

1. An Armv7 PE in the same Inner Shareable domain is treated in the same way as an Armv8 PE for which EL3 is using AArch32, except that if an Armv8 PE issues a broadcast instruction that is not defined in Armv7, then that instruction is not required to have an effect on the TLBs of the Armv7 PE. The instructions that do not exist in Armv7 include the following TLB maintenance instructions that Armv8 adds to the T32 and A32 instruction sets:
   - The following instructions that operate on TLB entries for the final level of translation table walk for stage 1 translations:

     `TLBIMVALIS`, `TLBIMVAALIS`, `TLBIMVALHIS`, `TLBIMVAL`, `TLBIMVAAL`, and `TLBIMVALH`.
   - The following instructions that operate by IPA on TLB entries for stage 2 translations:

     `TLBIIPAS2IS`, `TLBIIPAS2LIS`, `TLBIIPAS2`, and `TLBIIPAS2L`.
2. The number of Exception levels in Secure state depends on whether EL3 is using AArch32 or EL3 is using AArch64. This means that, within the Inner Shareable domain, there might be PEs with different numbers of Exception levels in Secure state. Therefore, the following exceptions are made to the general rules:
   - If a PE with EL3 using AArch32 issues a broadcast AArch32 TLB maintenance instruction affecting Secure entries, and the Inner Shareable domain also contains PEs with EL3 using AArch64, then the architecture does not require that the broadcast AArch32 TLB maintenance instruction has any effect on either:
     — The EL3 translation regime of the PEs with EL3 using AArch64.
     — The Secure or Non-secureEL1&0, when EL2 is disabled, translation regime of the PEs with EL3 using AArch64, regardless of whether the Secure or Non-secure EL1&0, when EL2 is disabled, translation regime is using AArch64 or AArch32.
   - If a PE with EL3 using AArch64 issues a broadcast AArch64 TLB maintenance instruction affecting EL3 entries, and the Inner Shareable domain also contains PEs with EL3 using AArch32, then the architecture does not require that the broadcast AArch64 TLB maintenance instruction has any effect on the EL3 translation regime of the PEs with EL3 using AArch32.
   - If a PE with EL3 using AArch64 issues a broadcast AArch64 TLB maintenance instruction affecting Secure EL1 entries, and the Inner Shareable domain also contains PEs with EL3 using AArch32 then the architecture does not require that the broadcast AArch64 TLB maintenance instruction has any effect on the EL3 translation regime of the PEs with EL3 using AArch32.

Note

While the exceptions to the general rule mean the architecture does not require the specified TLB invalidations, the architecture also does not require that entries in the TLB remain in the TLB at any time, and so it is permissible that such broadcast instructions affect these translation regimes.

Broadcast TLB maintenance with different translation granule sizes

In the following cases, a broadcast TLB maintenance instruction is not required to perform any invalidation on the recipient PE:

- The TLB maintenance instruction specifying a VA and affecting the EL2 translation regime, the EL2&0 translation regime, or the EL3 translation regime is broadcast from a PE using one translation granule size for that translation regime to a PE using a different translation granule size for that same translation regime
- The TLB maintenance instruction specifying a VA and affecting the EL1&0 translation regime is broadcast from a PE using one stage 1 translation granule size for that translation regime for a particular ASID (if applicable), VMID (if applicable), and Security state, to a PE where EL1 for the same ASID (if applicable), VMID (if applicable), and Security state, is using a different stage 1 translation granule size.
- The TLB maintenance instruction specifying a VA and affecting the Secure or Non-secure EL1&0, when EL2 is enabled, translation regime is broadcast from a PE using one stage 2 translation granule size for a particular ASID (if applicable) and VMID, to a PE where EL1 for the same ASID (if applicable) and VMID is using a different stage 2 translation granule size.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The TLB maintenance instruction specifying an IPA and affecting the Secure or Non-secure EL1&0, when EL2 is enabled, translation regime is broadcast from a PE using one stage 2 translation granule size for a particular VMID to a PE where EL1 for the same VMID is using a different stage 2 translation granule size.

Ordering and completion of TLB maintenance instructions

For AArch64 execution, a TLB maintenance instruction can be executed in any order relative to:

- Any load or store instruction, unless a `DSB` is executed between the load or store and the TLB maintenance instruction.

  Note

  In the Arm architecture, a translation table walk is considered to be a separate observer, and a store to the translation tables can be observed by that separate observer at any time after the instruction has been executed, but is only guaranteed to be observable after the execution of a `DSB` instruction by the PE that executed the store to the translation tables.

- Another TLB maintenance instruction, unless a `DSB` is executed between the instructions.
- A data or instruction cache maintenance instruction, unless a `DSB` is executed between the instructions.

For AArch64 execution, the completion rules are:

- A TLB invalidate instruction is complete for a PE when all memory accesses generated by that PE using the TLB entries that have been invalidated are complete.

  A TLB invalidate instruction is complete when it is complete for all PEs.

  After the TLB invalidate instruction is complete, no new memory accesses using the invalidated TLB entries will be architecturally performed by any observer that is affected by the TLB invalidation.

  Note

  This requirement does not mean that speculative memory accesses cannot be performed using those entries if it is impossible for software running on any observer to tell that those memory accesses have been performed.

- A TLB maintenance instruction executed by a PE, PEx, can complete at any time after it is issued, but is only guaranteed to be complete for a PE other than PEx after the execution of DSB by the PEx.
- A TLB maintenance instruction executed by a PE, PEx, can complete at any time after it is issued, but is only guaranteed to be complete for a PEx after the execution of DSB by the PEx followed by a *Context synchronization event*.

In all cases in this section where a `DMB` or `DSB` is referred to, it refers to a `DMB` or `DSB` whose required access type is both loads and stores. A `DSB NSH` is sufficient to ensure completion of TLB maintenance instructions that apply to a single PE. A `DSB ISH` is sufficient to ensure completion of TLB maintenance instructions that apply to PEs in the same Inner Shareable domain.

TLB maintenance in the event of TLB conflict

In the event that multiple entries in the TLB are being used to translate a given address (which implies that an attempt to access the given address might give rise to a TLB Conflict abort), it is IMPLEMENTATION DEFINED as to the form of TLB maintenance operation that the software must perform in order to be guaranteed that all TLB entries associated with the given address and translation regime have been invalidated. In all cases, an `ALL` or `VMALL` form of TLB maintenance operation that targets the given translation regime is guaranteed to remove all entries within that regime, even if there are multiple, conflicting TLB entries for any given address within that regime.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The interaction of TLB lockdown with TLB maintenance instructions

The precise interaction of TLB lockdown with the TLB maintenance instructions is IMPLEMENTATION DEFINED. However, the architecturally-defined TLB maintenance instructions must comply with these rules:

- The effect on a locked TLB entry of a TLB invalidate all operation that would invalidate that entry if the entry was not locked must be one of the following, and it is IMPLEMENTATION DEFINED which behavior applies:
  - — The operation has no effect on entries that are locked down.
  - — The operation generates an IMPLEMENTATION DEFINED Data Abort exception if an entry is locked down, or might be locked down.

    Any such exceptions taken from Non-secure EL1 can be trapped to EL2, see *Traps to EL2 of EL0 and EL1 accesses to lockdown, DMA, and TCM operations* on page D1-2385.

  Note

  These options permit a usage model for TLB invalidate routines, where the routine invalidates a large range of addresses, without considering whether any entries are locked in the TLB.

- The effect on a locked TLB entry of a TLB invalidate by VA or invalidate by ASID match operation that would invalidate that entry if the entry was not locked must be one of the following, and it is IMPLEMENTATION DEFINED which behavior applies:
  - — The locked entry is invalidated in the TLB.
  - — The operation has no effect on any locked entry in the TLB. In the case of an invalidate single entry by VA, this means the PE treats the operation as a `NOP`.
  - — The operation generates an IMPLEMENTATION DEFINED Data Abort exception if it operates on an entry that is locked down, or might be locked down.

The exception syndrome definitions include a fault code for cache and TLB lockdown faults, see *ESR_EL1, Exception Syndrome Register (EL1)* on page D13-2968.

Note

Any implementation that uses an abort mechanism for entries that can be locked down but are not actually locked down must:

- Document the IMPLEMENTATION DEFINED instruction sequences that perform the required operations on entries that are not locked down.
- Implement one of the other specified alternatives for the locked entries.

Arm recommends that, when possible, such IMPLEMENTATION DEFINED instruction sequences use the architecturally-defined operations. This minimizes the number of customized operations required.

In addition, an implementation that uses an abort mechanism for handling the effect of TLB maintenance instructions on entries that can be locked down but are not actually locked down must provide an IMPLEMENTATION DEFINED mechanism that ensures that no TLB entries are locked.

Similar rules apply to cache lockdown, see *The interaction of cache lockdown with cache maintenance instructions* on page D4-2521.

The architecture does not guarantee that any unlocked entry in the TLB remains in the TLB. This means that, as a side effect of any TLB maintenance instruction, any unlocked entry in the TLB might be invalidated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D5.11 Caches in a VMSAv8-64 implementation

The Arm architecture describes the required behavior of an implementation of the architecture. As far as possible it does not restrict the implemented microarchitecture, or the implementation techniques that might achieve the required behavior.

In particular, maintaining this level of abstraction is difficult when describing the relationship between memory address translation and caches, especially regarding the indexing and tagging policy of caches. This section:

- Summarizes the architectural requirements for the interaction between caches and address translation.
- Gives some information about the likely implementation impact of the required behavior.

The following sections give this information:

- *Data and unified caches*.
- *Instruction caches*.

In addition, *Cache maintenance requirement created by changing translation table attributes* on page D5-2687 describes the cache maintenance required after updating the translation tables to change the attributes of an area of memory.

For more information about cache maintenance see *A64 Cache maintenance instructions* on page D4-2510, that describes the cache maintenance instructions in the A64 instruction set.

D5.11.1 Data and unified caches

For data and unified caches, the use of address translation is entirely transparent to any data access other than as described in *Mismatched memory attributes* on page B2-164.

This means that the behavior of accesses from the same observer to different VAs, that are translated to the same PA with the same memory attributes, is fully coherent. This means these accesses behave as follows, regardless of which VA is accessed:

- Two writes to the same PA occur in program order.
- A read of a PA returns the value of the last successful write to that PA.
- A write to a PA that occurs, in program order, after a read of that PA, has no effect on the value returned by that read.

The memory system behaves in this way without any requirement to use barrier or cache maintenance instructions.

In addition, if cache maintenance is performed on a memory location, the effect of that cache maintenance is visible to all aliases of that physical memory location.

These properties are consistent with implementing all caches that can handle data accesses as *Physically-indexed, physically-tagged* (PIPT) caches.

D5.11.2 Instruction caches

In the Arm architecture, an instruction cache is a cache that is accessed only as a result of an instruction fetch. Therefore, an instruction cache is never written to by any load or store instruction executed by the PE.

The Arm architecture permits different behaviors for instruction caches. These are identified by descriptions of the associated expected implementation. The following subsections describe the behavior associated with these cache types, including any occasions where explicit cache maintenance is required to make the use of address translation transparent to the instruction cache:

- *PIPT (Physically-indexed, physically-tagged) instruction caches* on page D5-2686.
- *VPIPT (VMID-aware PIPT) instruction caches* on page D5-2686.
- *VIPT (Virtually-indexed, physically-tagged) instruction caches* on page D5-2686.
- *The IVIPT Extension* on page D5-2687.

The CTR_EL0.L1Ip field identifies the form of the instruction caches.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

For software to be portable between implementations that might use any of PIPT instruction caches, VPIPT instruction caches, or VIPT instruction caches, software must invalidate the instruction cache whenever any condition occurs that would require instruction cache maintenance for at least one of the instruction cache types.

PIPT (*Physically-indexed, physically-tagged*) instruction caches

For a PIPT instruction cache:

- The use of memory address translation is entirely transparent to all instruction fetches other than as described in *Mismatched memory attributes* on page B2-164.
- If cache maintenance is performed on a memory location, the effect of that cache maintenance is visible to all aliases of that physical memory location.

An implementation that provides PIPT instruction caches implements the IVIPT Extension, see *The IVIPT Extension* on page D5-2687.

VPIPT (*VMID-aware PIPT*) instruction caches

An Armv8.2 implementation can implement VPIPT instruction caches. If it does so then it is described as implementing ARMv8.2-VPIPT.

The CTR_EL0.L1Ip field identifies the implemented cache type, meaning it identifies whether ARMv8.2-VPIPT is implemented.

For a VPIPT instruction cache:

- Instruction fetches from Non-secure EL1 and Non-secure EL0 are only permitted to hit in the cache if the instruction fetch is made using the VMID that was used when the entry in the instruction cache was fetched.
- An instruction cache maintenance instruction executed at Non-secure EL0 or at Non-secure EL1 is required to have an effect on entries in the instruction cache only if those entries were fetched using the VMID that is current when the cache maintenance instruction is executed.

All other requirements for the use of cache maintenance instructions are the same as for *PIPT (Physically-indexed, physically-tagged) instruction caches*.

An implementation that provides VPIPT instruction caches implements the IVIPT Extension, see *The IVIPT Extension* on page D5-2687.

VIPT (*Virtually-indexed, physically-tagged*) instruction caches

For a VIPT instruction cache:

- The use of memory address translation is transparent to all instruction fetches other than for the effect of memory address translation on instruction cache invalidate by address operations or as described in *Mismatched memory attributes* on page B2-164.

  Note

  Cache invalidation is the only cache maintenance that can be performed on an instruction cache.

- If instruction cache invalidation by address is performed on a memory location, the effect of that invalidation is visible only to the VA supplied with the operation. The effect of the invalidation might not be visible to any other aliases of that physical memory location.

The only architecturally-guaranteed way to invalidate all aliases of a PA from a VIPT instruction cache is to invalidate the entire instruction cache.

An implementation that provides VIPT instruction caches implements the IVIPT Extension, see *The IVIPT Extension* on page D5-2687.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The IVIPT Extension

In Armv8, any permitted instruction cache implementation can be described as implementing the *IVIPT Extension* to the Arm architecture.

The formal definition of the Arm IVIPT Extension is that it reduces the instruction cache maintenance requirement to the following condition:

- Instruction cache maintenance is required only after writing new data to a PA that holds an instruction.

Note

Previous versions of the Arm architecture have permitted an instruction cache option that does not implement the Arm IVIPT Extension.

D5.11.3 Cache maintenance requirement created by changing translation table attributes

Any change to the translation tables to change the attributes of an area of memory can require maintenance of the translation tables, as described in *General TLB maintenance requirements* on page D5-2668. If the change affects the cacheability attributes of the area of memory, including any change between Write-Through and Write-Back attributes, software must ensure that any cached copies of affected locations are removed from the caches, typically by cleaning and invalidating the locations from the levels of cache that might hold copies of the locations affected by the attribute change. Any of the following changes to the inner cacheability or outer cacheability attribute creates this maintenance requirement:

- Write-Back to Write-Through
- Write-Back to Non-cacheable
- Write-Through to Non-cacheable
- Write-Through to Write-Back.

The cache clean and invalidate avoids any possible coherency errors caused by mismatched memory attributes.

Similarly, to avoid possible coherency errors caused by mismatched memory attributes, the following sequence must be followed when changing the shareability attributes of a cacheable memory location:

1. Make the memory location Non-cacheable, Outer Shareable.
2. Clean and invalidate the location from them cache.
3. Change the shareability attributes to the required new values.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter D6
Armv8.5 Memory Tagging Extension

This chapter provides details of the optional ARMv8.5-MemTag Extension.

- *Introduction* on page D6-2690.
- *Allocation Tags* on page D6-2691.
- *Tag checking* on page D6-2692.
- *Tagged and Untagged Addresses* on page D6-2693.
- *PE access to Allocation Tags* on page D6-2694.
- *Enabling the Memory Tagging Extension* on page D6-2695.
- *PE handling of Tag Check Fault* on page D6-2696.
- *PE generation of Tag Checked and Tag Unchecked accesses* on page D6-2697.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D6.1 Introduction

ARMv8.5-MemTag is an optional extension to the Armv8.5 architecture in AArch64 state only.

The extension implements:

- Tag load and store instructions to access Allocation tags in a tag physical address space, separate to the data physical address space accessed by data load and store instructions to access data in normal memory and devices.
- 4-bit Allocation Tags at a granularity of one Allocation Tag for each naturally-aligned set of 16 physically addressed locations.

  Note

  Implementations are expected to provide one Allocation Tag for each 16 byte granule of bulk data memory.

- System register and page level control over access to Allocation Tags in memory.
- A Logical Address Tag from which a Physical Address Tag can be derived and held in the upper bits of the address that is used in load and store data instructions.
- A Tag Check of the Physical Address Tag against the 4-bit Allocation Tag for each data location that is accessed by a Tag Checked load or store of data.
- Instructions to generate and manipulate the Logical Address Tag in a register.
- A random tag generator using state that is held in RGSR_EL1 and controlled by GCR_EL1.
- Configurable behavior on a Tag Check returning a fail result causing a synchronous exception or recording the failure in a System register, one of TFSR_ELx or TFSRE0_EL1.
- Cache maintenance operations operating on Allocation Tags, or Allocation Tags and data.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D6.2 Allocation Tags

The tag physical address (tag PA) space provides access to Allocation Tags stored in memory. The data physical address (data PA) space provides access to data held in memory.

An Allocation Tag is 4 bits.

Each naturally-aligned set of 16 tag PA locations is a Tag Granule. Each Tag Granule is associated with one Allocation Tag.

Note

- The value `0b1111` may incur a higher performance overhead than other Allocation Tag encodings.
- Arm recommends that software does not use instructions which write `0b1111` as an Allocation Tag to memory.
- Arm recommends that implementations provide storage for Allocation Tags at each tag PA where general-purpose memory exists at the same physical address in the data PA space.

It is IMPLEMENTATION DEFINED what happens when an access is made to the tag PA where Allocation Tag storage is not provided.

It is IMPLEMENTATION DEFINED whether Allocation Tags are permitted to be accessed via regions of the data PA space. If Allocation Tags are accessible via the data PA space, then the layout of Allocation Tags is IMPLEMENTATION DEFINED.

It is not architecturally required for an Allocation Tag accessed via the tag PA space to be coherent with the same Allocation Tag accessed via the data PA space. A write to one location can be made visible at the other location by the use of the cache maintenance operations.

Unless otherwise stated the definitions in Chapter B2 *The AArch64 Application Level Memory Model* and Chapter D4 *The AArch64 System Level Memory Model* which apply to data accesses and data apply separately to Allocation Tag accesses and Allocation tags.

It is IMPLEMENTATION DEFINED whether the global monitor monitors access to the tag PA space.

D6.2.1 Cache activity and Allocation Tags

For more information on DC operations that affect Allocation Tags see *A64 System instructions for cache maintenance* on page C5-475.

The eviction of data from a cache entry from a cache level can overwrite data in memory that has been written by another observer only if the entry contains a memory location where either or both the data or associated Allocation Tags have been written to by an observer in the Shareability domain of that memory location, where the maximum size of the memory that can be overwritten is defined by the Cache Write-Back Granule in CTR_EL0.

The eviction of Allocation Tags from a cache entry from a cache level can overwrite Allocation Tags in memory that have been written by another observer only if the entry contains a memory location where the Allocation Tags have been written to by an observer in the Shareability domain of that memory location, where the maximum size of the memory that can be overwritten is defined by the Cache Write-Back Granule in CTR_EL0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D6.3 Tag checking

A memory access is either Tag Checked or Tag Unchecked.

An access to the data PA space is either Tag Checked or Tag Unchecked.

An access to the tag PA space is always Tag Unchecked.

A data access which is performed as part of a prefetch operation is Tag Unchecked.

When the value of PSTATE.TCO is 1, all loads and stores are Tag Unchecked.

A Tag Checked memory access includes a physical address Tag. A physical address Tag is 4 bits.

A Tag Checked access causes a Tag Check operation to be performed.

If the Allocation Tag and Physical Address Tag in a Tag Check operation do not match, the Tag Check operation generates a Tag Check Fault.

A Tag Check Fault can be configured to cause one of:

- A synchronous exception.
- A bit to be asynchronously set in TFSR_ELx.
- It to be ignored.

If a Tag Check Fault on a store causes a synchronous exception, memory locations associated with the fault are unchanged.

If a Tag Check Fault does not cause a synchronous exception, there is no effect on the data access, or any side effects on the data access, so that the data access is always performed, unless prevented by another exception being taken or a Data Abort.

The read of an Allocation Tag due to a Tag Check operation and the dependent data access are not required to form an atomic operation.

If the Tag Checked property of a memory access due to a Store-Exclusive instruction and the Tag Checked property of a memory access due to the preceding Load-Exclusive instruction in a Load-Exclusive/Store-Exclusive instruction pair to the same location from the same PE do not match, then software cannot rely on the Load-Exclusive/Store-Exclusive pair to eventually succeed.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D6.4 Tagged and Untagged Addresses

A virtual address is either Tagged or Untagged.

An access to memory at an Untagged virtual address generates an Unchecked access.

An access to memory at a Tagged virtual address permits the generation of Tag Checked or Tag Unchecked access.

A read of an Allocation Tag from an Untagged virtual address returns the value `0b0000`.

A write of an Allocation Tag to an Untagged address is IGNORED.

All virtual addresses in AArch32 state are Untagged.

D6.4.1 Virtual address translation

If stage 1 translation at the current Exception level is enabled, stage 1 translations are Tagged or Untagged depending on the Memory Attributes for the memory location being accessed.

If stage 1 translation at the current Exception level is disabled:

- When the value of HCR_EL2.DC is one, stage 1 translations are Tagged or Untagged depending on the value of HCR_EL2.DCT.
- When the value of HCR_EL2.DC is zero, stage 1 translations are treated as Untagged.

Memory locations are treated as Tagged where all of the following is true:

- The combined effects of stage 1 and stage 2 translations define the memory attributes as:
  - Normal memory,
  - Inner, and Outer Write-Back Non-Transient Read-Allocate Write-Allocate
- The stage 1 translation is treated as Tagged.

Otherwise memory locations are Untagged.

If a memory location is marked as Untagged, a data cache invalidation operation that would invalidate Allocation Tags at that location cleans and invalidates the Allocation Tags.

When the EL1&0 stage 1 translation regime is disabled and HCR_EL2.DC is one, in the current Security state, the execution of any `AT S1E0`, `AT S1E1`, `AT S12E0`, or `AT S12E1` address translation instruction will reflect the effect of HCR_EL2.DCT in PAR_EL1.ATTR.

For more information on Virtual address translation, see *The VMSAv8-64 address translation system* on page D5-2540.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D6.5 PE access to Allocation Tags

A PE can access an Allocation tag using the following instructions:

- LDG.
- LDGM.
- ST2G.
- STG.
- STGM.
- STGP.
- STZ2G.
- STZG.
- STZGM.

Instructions that load or store Allocation Tags apply the same address translation and permission checks as a load or store of data to a virtual address.

An instruction that loads or stores an Allocation Tag:

- Is considered a load or store of data to each location associated with the Allocation Tag for the purpose of triggering Watchpoints and PMU events.
- Generates a tag PA with the same physical address as a load or store of data to a virtual address.

If ID_AA64PFR1_EL1.MTE is `0b0010` the Memory Tagging Extension is fully implemented.

If ID_AA64PFR1_EL1.MTE is `0b0001` all of the following is true:

- SCR_EL3.ATA is RES0.
- HCR_EL2.ATA is RES0.
- SCTLR_ELx.ATA is RES0.
- SCTLR_ELx.ATA0 is RES0.
- All operations which read Allocation Tags treat the Allocation Tag as zero, any traps or permission checks continue to apply.
- All operations which write an Allocation Tag to memory do not modify the Allocation Tag in memory. Any traps or permission checks continue to apply.
- Instructions which insert Allocation Tags into addresses treat the Allocation Tag as zero. This means that:
  - IRG inserts `0b0000`.
  - ADDG or SUBG insert `0b0000`.
  - LDG inserts `0b0000`.
- The System instructions that are introduced by the Memory Tagging Extension are Unallocated.
- All other instructions which read or write Allocation Tags behave as if ID_AA64PFR1_EL1.MTE is `0b0010`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D6.6 Enabling the Memory Tagging Extension

Access to Allocation Tags in memory can be enabled by the use of the following controls:

- SCR_EL3.ATA.
- HCR_EL2.ATA.
- SCTLR_ELx.ATA.
- SCTLR_ELx.ATA0.

When access to Allocation Tags is disabled for an Exception level, instructions that are executed at that Exception level, that:

- Load or store data are Unchecked.
- Load or store Allocation Tags treat the Allocation Tag as RAZ/WI.
- Insert Logical Address Tags into addresses treat the Allocation Tag used to generate the Logical Address Tag as zero.

For the purpose of determining Allocation Tag access, unprivileged load and store instructions are treated as if executed at EL0 when executed either:

- At EL1 when the Effective value of PSTATE.UAO is 0.
- At EL2 when both the Effective value of HCR_EL2.{E2H, TGE} is {1, 1} and the Effective value of PSTATE.UAO is 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D6.7 PE handling of Tag Check Fault

If SCTLR_ELx.TCF has the value of `0b00`, a Tag Check Fault due to a load or store at ELx has no effect on the PE.

If SCTLR_ELx.TCF has the value of `0b01`, a Tag Check Fault due to a load or store at ELx generates a synchronous exception.

If SCTLR_ELx.TCF has the value of `0b10`, a Tag Check Fault due to a load or store at ELy using TTBR_ELx causes TFSR_ELx.TFx to be asynchronously set to 1.

A Tag Check Fault due to a load or store at EL0 has no effect on the PE if either of the following conditions are true:

- SCTLR_EL1.TCF0 has the value of `0b00` and HCR_EL2{E2H, TGE} does not have the value of {1,1}.
- SCTLR_EL2.TCF0 has the value of `0b00` and HCR_EL2{E2H, TGE} has the value of {1,1}.

A Tag Check Fault due to a load or store at EL0 generates a synchronous exception if either of the following conditions are true:

- SCTLR_EL1.TCF0 has the value of `0b01` and HCR_EL2{E2H, TGE} does not have the value of {1,1}.
- SCTLR_EL2.TCF0 has the value of `0b01` and HCR_EL2{E2H, TGE} has the value of {1,1}.

A Tag Check Fault due to a load or store at EL0 using TTBRy_EL1 or TTBRy_EL2 causes TFSRE0_EL1.TFy to be set to 1 if either of the following conditions are true:

- SCTLR_EL1.TCF0 has the value of `0b10` and HCR_EL2{E2H, TGE} does not have the value of {1,1}
- SCTLR_EL2.TCF0 has the value of `0b10` and HCR_EL2{E2H, TGE} has the value of {1,1}.

TFSR_ELx and TFSRE0_EL1 are unchanged by a memory data access causing a Tag Check pass.

A synchronous exception due to a Tag Check Fault is reported as a Data Abort with a Data Fault status code of Synchronous Tag Check Fault and the faulting virtual address is reported in FAR_ELx.

A Data Abort due to a Tag Check Fault is taken to:

- EL1 from EL0 if HCR_EL2.TGE has the value of 0.
- EL2 from EL0 if HCR_EL2.TGE has the value of 1.
- ELx from ELx where x is 1, 2 or 3.

A Data Abort due to a Tag Check Fault is prioritized as a Data Abort exception generated by a synchronous External abort that was not generated by a translation table walk.

If an access generates both a Data Abort due to a Synchronous Tag Check Fault and a Data Abort due to a synchronous External abort that was not generated by a translation table walk, it is IMPLEMENTATION DEFINED which abort is reported. For more information on prioritization of exceptions see *Synchronous exception types, routing and priorities* on page D1-2354.

If an instruction that stores to memory generates a Data Abort that is a Synchronous Tag Check Fault, the value of each memory location that the instruction stores to is UNKNOWN for any location for which no exceptions and no Debug event is generated. The size of a memory location is defined as being the size for which a memory access is single-copy atomic.

For the purpose of determining Tag Check Fault handling, unprivileged load and store instructions are treated as if executed at EL0 when executed either:

- At EL1 when the Effective value of PSTATE.UAO is 0.
- At EL2 when both the Effective value of HCR_EL2.{E2H, TGE} is {1, 1} and the Effective value of PSTATE.UAO is 0.

If SCTLR_ELy.ITFSB has the value of `0b1`, then updates to TFSRE0_EL1, and any TFSR_ELx accessible at ELy, are synchronized on exception entry to ELy.

When SVE is implemented, if a load of an element in a SVE Non-faulting or First-faulting load instruction causes a Tag Check Fault, and is not the first active element in a First-faulting instruction, the Tag Check Fault:

- Is recorded in the corresponding FFR register.
- Does not generate a Synchronous Tag Check Fault exception.
- Does not cause any bit in any TFSR_ELx or TFSRE0_EL1 registers to be set.
- The value loaded into the element is UNKNOWN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D6.8 PE generation of Tag Checked and Tag Unchecked accesses

Bits [59:56] of the 64-bit address that is used for a load or store instruction from a Logical Address Tag.

The PE generates a Physical Address Tag from the Logical Address Tag for each Tag Checked access to memory.

Unless an access is explicitly defined as a Tag Unchecked access, it is a Tag Checked access.

D6.8.1 Tag Unchecked accesses

The following operations generate a Tag Unchecked access:

- An instruction fetch.
- A load instruction that loads an Allocation Tag.
- A store instruction that stores an Allocation Tag.

When PSTATE.TCO has the value of one all loads and stores generate Tag Unchecked accesses.

A cache maintenance by virtual address operation other than `DC ZVA, Data Cache Zero by VA` generates a Tag Unchecked access.

An access due to a translation table walk generates a Tag Unchecked access.

If ARMv8.4-NV is implemented, Loads and Stores relative to VNCR_EL2 generate a Tag Unchecked access.

If the Statistical Profiling Extension is implemented all accesses to the Profiling Buffer are Tag Unchecked accesses, see Chapter D9 *The Statistical Profiling Extension* for more information.

Data accesses by an external Debugger may generate Tag Checked accesses, see Chapter H2 *Debug State* for more information.

An access which would be translated using TTBR0_ELx if the stage 1 translation regime were enabled is Tag Unchecked, irrespective of whether the stage 1 translation regime is disabled or not, where either of the following conditions apply:

- TCR_ELx.TBI has the value of zero.
- TCR_ELx.TBI0 has the value of zero.

If TCR_ELx.TBI1 has the value of zero, an access which would be translated using TTBR1_ELx if the stage 1 translation regime were enabled is Tag Unchecked, irrespective of whether the stage 1 translation regime is enabled or not.

An access will be Tag Unchecked, irrespective of whether the stage 1 translation regime is disabled or not, where all of the following conditions apply:

- The access would be translated using TTBR0_ELx.
- The Logical Address Tag is `0b0000`.
- TCR_ELx.TCMA has the value of one, or TCR_ELx.TCMA0 has the value of one.

An access will be Tag Unchecked, irrespective of whether the stage 1 translation regime is disabled or not, when all of the following conditions apply:

- The access would be translated using TTBR1_ELx.
- The Logical Address Tag is `0b1111`.
- TCR_ELx.TCMA1 has the value of one.

A Tag Unchecked access will be generated for a load or store that uses either of the following:

- A base register only, with the SP as the base register.
- A base register plus immediate offset addressing form, with the SP as the base register.

Literal (PC-relative) loads generate a Tag Unchecked access.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D6.8.2 Constrained Unpredictable behavior

When executing a Store Exclusive instruction that if Tag Unchecked would not perform the store and return a status value of one, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction generates a Tag Checked access.
- The instruction generates a Tag Unchecked access.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter D7
The Performance Monitors Extension

This chapter describes the Armv8 implementation of the Arm Performance Monitors, that are an optional non-invasive debug component. It describes version 3 of the *Performance Monitor Unit* (PMU) architecture, PMUv3. It contains the following sections:

- *About the Performance Monitors* on page D7-2700.
- *Accuracy of the Performance Monitors* on page D7-2703.
- *Behavior on overflow* on page D7-2705.
- *Attributability* on page D7-2707.
- *Prohibiting event counting* on page D7-2709.
- *Event filtering* on page D7-2711
- *Performance Monitors and Debug state* on page D7-2713.
- *Counter enables* on page D7-2714.
- *Counter access* on page D7-2715.
- *PMU events and event numbers* on page D7-2716.
- *Performance Monitors Extension registers* on page D7-2772.

Note

Table K14-2 on page K14-8040 disambiguates the general register references used in this chapter.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D7.1 About the Performance Monitors

In Armv8-A, the Performance Monitors Extension is an OPTIONAL feature of an implementation, but Arm strongly recommends that Armv8-A implementations include version 3 of the Performance Monitors Extension, PMUv3,

Note

No previous versions of the Performance Monitors Extension can be implemented in Armv8.

The basic form of the Performance Monitors is:

- A 64-bit cycle counter, see *Time as measured by the Performance Monitors cycle counter* on page D7-2702.
- A number of 64-bit or 32-bit event counters. If ARMv8.5-PMU is implemented and the highest Exception level is using AArch64, the event counters are 64-bit. If ARMv8.5-PMU is not implemented, the event counters are 32-bit.
- The event counted by each event counter is programmable. Armv8 provides space for up to 31 event counters. The actual number of event counters is IMPLEMENTATION DEFINED, and the specification includes an identification mechanism.

  Note

  Arm recommends that at least two event counters are implemented, and that hypervisors provide at least this many event counters to guest operating systems.

- When EL2 is implemented, the required controls to partition the implemented event counters into the following sets:
  - A set which is available for use by the guest operating system accessible at all Exception levels.
  - A set which is available for use by the hypervisor accessible at EL3 and EL2, and, if ARMv8.4-SecEL2 is not implemented or if Secure EL2 is disabled, in Secure state.
- Controls for:
  - Enabling and resetting counters.
  - Flagging overflows.
  - Enabling interrupts on overflow.

  Monitoring software can enable the cycle counter independently of the event counters, and if ARMv8.5-PMU is implemented, disable the cycle counter in Secure state or in EL2.

The PMU architecture uses event numbers to identify an event. It:

- Defines event numbers for common events, for use across many architectures and microarchitectures.

  Note

  Implementations that include PMUv3 must, as a minimum requirement, implement a subset of the common events. See *Common event numbers* on page D7-2722.

- Reserves a large event number space for IMPLEMENTATION DEFINED events.

The full set of events for an implementation is IMPLEMENTATION DEFINED. Arm recommends that implementations include all of the events that are appropriate to the architecture profile and microarchitecture of the implementation.

When an implementation includes the Performance Monitors Extension, Armv8 defines the following possible interfaces to the Performance Monitors Extension registers:

- A System register interface. This interface is mandatory.

  Note

  In AArch32 state, the interface is in the (`coproc==0b1111`) encoding space.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- An external debug interface which optionally supports memory-mapped accesses. Implementation of this interface is OPTIONAL. See Chapter I3 *Recommended External Interface to the Performance Monitors*.

An operating system can use the System registers to access the counters.

Also, if required, the operating system can enable application software to access the counters. This enables an application to monitor its own performance with fine-grain control without requiring operating system support. For example, an application might implement per-function performance monitoring.

To enable interaction with external monitoring, an implementation might consider additional enhancements, such as providing:

- A set of events, from which a selection can be exported onto a bus for use as external events.
- The ability to count external events. This enhancement requires the implementation to include a set of external event input signals.

The Performance Monitors Extension is common to AArch64 operation and AArch32 operation. This means the Armv8 architecture defines both AArch64 and AArch32 System registers to access the Performance Monitors. For example, the Performance Monitors Cycle Count Register is accessible as:

- When executing in AArch64 state, PMCCNTR_EL0.
- When executing in AArch32 state, PMCCNTR.

When executing in AArch32 state, if ARMv8.5-PMU is implemented, bits [63:32] of the event counters are not accessible. If the implementation does not support AArch64 at any Exception level, 64-bit event counters are not required to be implemented.

D7.1.1 Interaction with EL3

Software executing at EL3 can trap attempts by lower Exception levels to access the PMU. This means that the Secure monitor can identify any software which is using the PMU and switch contexts, if required.

Software executing at EL3 can:

- Prohibit counting of events Attributable to Secure state.
- If ARMv8.5-PMU is implemented, prohibit counting of cycles in Secure state, see *Prohibiting event counting* on page D7-2709.

In AArch32 state, the Performance Monitors registers are Common registers, see *Classification of System registers* on page G5-6052.

If ARMv8.6-MTPMU is implemented and EL3 is implemented, MDCR_EL3.MTPME and SDCR.MTPME enable and disable the PMEVTYPER<n>.MT bit.

D7.1.2 Interaction with EL2

Software can program HDCR.HPMN to reserve the highest-numbered event counters by partitioning the event counters into two sets. This does not depend on whether EL2 is enabled in the current Security state. Each set has its own global controls.

Software executing at EL3, and when EL2 is implemented and enabled in the current Security state, software executing at EL2 can:

- Trap an access at EL0 or EL1 to the PMU. This means the hypervisor can identify which Guest OSs are using the PMU and intelligently employ switching of the PMU state. There is a separate trap for the PMCR register and if ARMv8.6-FGT is implemented and enabled, fine-grained traps are provided.
- If ARMv8.1-PMU is implemented, prohibit counting of events Attributable to EL2 by the counters accessible to EL1 and EL0.
- If ARMv8.5-PMU is implemented, prohibit counting of cycles at EL2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When EL2 is implemented and enabled in the current Security state, software executing at EL1 and, if enabled by PMUSERENR, EL0:

- Will read the value of HDCR.HPMN for PMCR.N.
- Cannot access the highest-numbered event counters, or the controls associated with them.

If ARMv8.6-MTPMU is implemented, EL3 is not implemented, and EL2 is implemented, MDCR_EL2.MTPME and HDCR.MTPME enable and disable the PMEVTYPER<n>.MT bit.

For more information, see:

- *Counter enables* on page D7-2714.
- *Counter access* on page D7-2715.
- *Prohibiting event counting* on page D7-2709.
- *Multithreaded implementations* on page D7-2710.

D7.1.3 Time as measured by the Performance Monitors cycle counter

The Performance Monitors cycle counter, accessed through PMCCNTR_EL0 or PMCCNTR, increments from the hardware processor clock, not PE clock cycles.

The relationship between the count recorded by the Performance Monitors cycle counter and the passage of real time is IMPLEMENTATION DEFINED.

See *Prohibiting event counting* on page D7-2709 for information about when the cycle counter does not increment.

Note

- This means that, in an implementation where PEs are multithreaded, the counter continues to increment across all PEs, rather than only counting cycles for which the current PE is active.
- Although the architecture requires that direct reads of PMCCNTR_EL0 or PMCCNTR occur in program order, there is no requirement that the count increments between two such reads. Even when the counter is incrementing on every clock cycle, software might need check that the difference between two reads of the counter is nonzero.

  The architecture requires that an indirect write to the PMCCNTR_EL0 or PMCCNTR is observable to direct reads of the register in finite time. The counter increments from the hardware processor clock are indirect writes to these registers.

D7.1.4 Interaction with trace

It is IMPLEMENTATION DEFINED whether the implementation exports counter events to a PE Trace Unit, or other external monitoring agent, to provide triggering information. The form of any exporting is also IMPLEMENTATION DEFINED. If implemented, this exporting might be enabled as part of the performance monitoring control functionality.

Arm recommends system designers include a mechanism for importing a set of external events to be counted, but such a feature is IMPLEMENTATION DEFINED. When implemented, this feature enables the PE Trace Unit to pass in events to be counted.

Exporting PMU events to the ETM is prohibited for some Exception levels when `SelfHostedTraceEnabled`() == TRUE. For more information, see *Controls to prohibit trace at Exception levels* on page D3-2489.

D7.1.5 Interaction with power saving operations

All counters are subject to any changes in clock frequency, including clock stopping caused by the `WFI` and `WFE` instructions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D7.2 Accuracy of the Performance Monitors

The Performance Monitors:

- Are a non-invasive debug component. See *Non-invasive behavior*.
- Must provide broadly accurate and statistically useful count information.

However, the Performance Monitors allow for:

- A reasonable degree of inaccuracy in the counts to keep the implementation and validation cost low. See *A reasonable degree of inaccuracy*.
- IMPLEMENTATION DEFINED controls, such as those in ACTLR registers, to put the PE in an operating state that might do one or both of the following:
  - Change the level of non-invasiveness of the Performance Monitors so that enabling an event counter can impact the performance or behavior of the PE.
  - Allow inaccurate counts. This includes, but is not limited to, cycle counts.

D7.2.1 Non-invasive behavior

The Performance Monitors are a non-invasive debug feature. A non-invasive debug feature permits the observation of data and program flow. Performance Monitors, PC Sample-based Profiling and Trace are non-invasive debug features.

Non-invasive debug components do not guarantee that they do not make any changes to the behavior or performance of the processor. Any changes that do occur must not be severe however, as this will reduce the usefulness of event counters for performance measurement and profiling. This does not include any change to program behavior that results from the same program being instrumented to use the Performance Monitors, or from some other performance monitoring process being run concurrently with the process being profiled in a multitasking operating system. As such, a reasonable variation in performance is permissible.

Note

Power consumption is one measure of performance. Therefore, a reasonable variation in power consumption is permissible.

Arm does not define a *reasonable variation in performance*, but recommends that such a variation is kept within 5% of normal operating performance, when averaged across a suite of code that is representative of the application workload.

Note

For profiles other than A-profile, there is the potential for stronger requirements. Ultimately, performance requirements are determined by end-users, and not set by the architecture.

For some common architectural events, this requirement to be non-invasive can conflict with the requirement to present an accurate value of the count under normal operating conditions. Should an implementation require more performance-invasive techniques to accurately count an event, there are the following options:

- If the event is optional, define an alternative implementation defined event that accurately counts the event and document the impact on performance of enabling the event.
- Provide an implementation defined control that disables accurate counting of the event to restore broadly accurate performance, and document the impact on performance of accurate counting.

D7.2.2 A reasonable degree of inaccuracy

The Performance Monitors provide broadly accurate and statistically useful count information. To keep the implementation and validation cost low, a reasonable degree of inaccuracy in the counts is acceptable. Arm does not define a *reasonable degree of inaccuracy* but recommends the following guidelines:

- Under normal operating conditions, the counters must present an accurate value of the count.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- In exceptional circumstances, such as a change in Security state or other boundary condition, it is acceptable for the count to be inaccurate.
- Under very unusual, non-repeating pathological cases, the counts can be inaccurate. These cases are likely to occur as a result of asynchronous exceptions, such as interrupts, where the chance of a systematic error in the count is very unlikely.

Note

An implementation must not introduce inaccuracies that can be triggered systematically by the execution of normal pieces of software. For example, it is not reasonable for the count of branch behavior to be inaccurate when caused by a systematic error generated by the loop structure producing a dropping in branch count.

However, dropping a single branch count as the result of a rare interaction with an interrupt is acceptable.

The permitted inaccuracy limits the possible uses of the Performance Monitors. In particular, the architecture does not define the points in a pipeline where the event is generated and where it is counted, relative to the point where a read of the counters is made. This means that pipelining effects can cause some imprecision.

Where a direct write to a Performance Monitors control register disables a counter, and is followed by a *Context synchronization event*, any subsequent indirect read of the control register by the Performance Monitors to determine whether the counter is enabled will return the updated value. Any subsequent direct read of the counter will return the value at the point the counter was disabled.

Note

The imprecision means that the counter might have counted an event around the time the counter was disabled, but does not allow the event to be observed as counted after the counter was disabled.

A change of Security state can also affect the accuracy of the Performance Monitors, see *Interaction with EL3* on page D7-2701.

In addition to this, entry to and exit from Debug state can disturb the normal running of the PE, causing further inaccuracy in the Performance Monitors. Disabling the counters while in Debug state limits the extent of this inaccuracy. An implementation can employ methods to limit this inaccuracy, for example by promptly disabling the counters during the Debug state entry sequence.

An implementation must document any particular scenarios where significant inaccuracies are expected.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D7.3 Behavior on overflow

The event counters, PMEVCNTR<n> are either 32-bit or 64-bit unsigned counters that overflow in the following situations:

- If ARMv8.5-PMU is not implemented, 32-bit event counters are implemented, and if incrementing PMEVCNTR<n> causes an unsigned overflow of an event counter, the PE sets PMOVSCLR[n] to 1.
- If ARMv8.5-PMU is implemented, 64-bit event counters are implemented, and either n is in the range [0 .. (HDCR.HPMN-1)] or EL2 is not implemented, then event counter overflow is configured by PMCR.LP:
  - When PMCR.LP is set to 0, if incrementing PMEVCNTR<n> causes an unsigned overflow of bits [31:0] of the event counter, the PE sets PMOVSCLR[n] to 1.
  - When PMCR.LP is set to 1, if incrementing PMEVCNTR<n> causes an unsigned overflow of bits [63:0] of the event counter, the PE sets PMOVSCLR[n] to 1.
- If ARMv8.5-PMU is implemented, 64-bit event counters are implemented, and EL2 is implemented, when n is in the range [HDCR.HPMN .. (PMCR.N-1)], event counter overflow is configured by HDCR.HLP:
  - When HDCR.HLP is set to 0, if incrementing PMEVCNTR<n> causes an unsigned overflow of bits [31:0] of the event counter, the PE sets PMOVSCLR[n] to 1.
  - When HDCR.HLP is set to 1, if incrementing PMEVCNTR<n> causes an unsigned overflow of bits [63:0] of the event counter, the PE sets PMOVSCLR[n] to 1.

The cycle counter, PMCCNTR, is a 64-bit unsigned counter, that is configured by PMCR.LC:

- If PMCR.LC is set to 0, if incrementing PMCCNTR causes an unsigned overflow of bits [31:0] of the cycle counter, the PE sets PMOVSCLR[31] to 1.
- If PMCR.LC is set to 1, if incrementing PMCCNTR causes an unsigned overflow of bits [63:0] of the cycle counter, the PE sets PMOVSCLR[31] to 1.

For all 64-bit counters, incrementing the counter is the same whether an unsigned overflow occurs at [31:0] or [63:0]. If the counter increments for an event, bits [63:0] are always incremented,

When any overflow occurs, an interrupt request is generated if the PE is configured to generate counter overflow interrupts. For more information, see *Generating overflow interrupt requests*.

Note

Software executing at EL1 or higher must take care that setting PMCR.LP or HDCR.HLP does not cause software executing at lower Exception levels to malfunction. If legacy software accesses the PMU at lower Exception levels, software at the higher Exception levels should not set the PMCR.LP or HDCR.HLP fields to 1. However, if the legacy software does not use the counter overflow, it is not affected by setting the PMCR.LP or HDCR.HLP to 1.

D7.3.1 Generating overflow interrupt requests

Software can program the Performance Monitors so that an overflow interrupt request is generated when a counter overflows. See PMINTENSET and PMINTENCLR.

Note

- The mechanism by which an interrupt request from the Performance Monitors generates an FIQ or IRQ exception is IMPLEMENTATION DEFINED.
- Arm recommends that the overflow interrupt requests:
  - Translate into a **PMUIRQ** signal, so that they are observable to external devices.
  - Connect to inputs on an IMPLEMENTATION DEFINED generic interrupt controller as a *Private Peripheral Interrupt* (PPI) for the originating processor. See the *ARM Generic Interrupt Controller Architecture Specification* for information about PPIs.
  - Connect to a *Cross Trigger Interface* (CTI), see Chapter H5 *The Embedded Cross-Trigger Interface*.
- Arm strongly discourages implementations from connecting overflow interrupt requests from multiple PEs to the same *System Peripheral Interrupt* (SPI) identifier.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- From GICv3, the *ARM® Generic Interrupt Controller Architecture Specification* recommends that the *Private Peripheral Interrupt* (PPI) with ID 23 is used for overflow interrupt requests.

Software can write to the counters to control the frequency at which interrupt requests occur. For example, software might set a 32-bit counter to `0xFFFF0000`, to generate another counter overflow after 65 536 increments, and reset it to this value every time an overflow interrupt occurs.

Note

If an event can occur multiple times in a single clock cycle, then counter overflow can occur without the counter registering a value of zero.

The overflow interrupt request is a level-sensitive request. The PE signals a request for:

- Any given PMEVCNTR<n> counter, when the value of PMOVSSET[*n*] is 1, the value of PMINTENSET[*n*] is 1, and one of the following is true:
  - EL2 is not implemented and the value of PMCR.E is 1.
  - EL2 is implemented, *n* is less than the value of HDCR.HPMN, and the value of PMCR.E is 1.
  - EL2 is implemented, *n* is greater than or equal to the value of HDCR.HPMN, and the value of HDCR.HPME is 1.
- The cycle counter, when the values of PMOVSSET[31], PMINTENSET[31], and PMCR.E are all 1.

The overflow interrupt request is active in both Secure and Non-secure states. In particular, if EL3 and EL2 are both implemented, overflow events from PMEVCNTR<n> where *n* is greater than or equal to the value of HDCR.HPMN can be signaled from all modes and states but only if the value of HDCR.HPME is 1.

The interrupt handler for the counter overflow request must cancel the interrupt request, by writing to PMOVSCLR[n] to clear the overflow bit to 0.

Pseudocode description of overflow interrupt requests

See Chapter J1 *Armv8 Pseudocode* for a pseudocode description of overflow interrupt requests. The `AArch64.CheckForPMUOverflow()` and `AArch32.CheckForPMUOverflow()` pseudocode functions signal PMU overflow interrupt requests to an interrupt controller and PMU overflow trigger events to the cross-trigger interface.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D7.4 Attributability

An event caused by the PE counting the event is *Attributable*. If an agent other than the PE that is counting the events causes an event, these events are Unattributable.

An event is defined as being either Attributable or Unattributable. If the event is Attributable, it is further defined whether it is Attributable to:

- The current Security state of the PE.
- The current Exception level of the PE.
- When the PE is in Debug state, operations issued to the PE by the debugger through the external debug interface.

In a multithreaded implementation, an event might be generated by another PE with the same values for affinity level 1 and higher. This event is further defined as Attributable to:

- The current Security state of that PE.
- The current Exception level of that PE.
- When that PE is in Debug state, operations issued to that PE by the debugger through the external debug interface.

See *Multithreaded implementations* on page D7-2710 for information about enabling and restricting counting events in a multithreaded implementation.

Note

- In an implementation containing multiple PEs, each PE is identified by a unique *affinity* value reported by MPIDR_EL1{Aff3, Aff2, Aff1, Aff0}, where the value of affinity level 0 is the most significant for determining the PE behavior, and the values of higher affinity levels are less significant. Affinity level 3 is only supported in AArch64 state.
- An implementation is described as multithreaded when the lowest level of affinity consists of logical PEs that are implemented using a multithreading type approach. In this section, when referring to a multithreaded implementation, *thread* is used to mean processing elements with:
  - MPIDR_EL1.MT or MPIDR.MT set to 1,
  - Different values for affinity level 0.
  - The same values for affinity level 1 and higher.

An event can be defined as the combination of multiple subevents, which can be either Attributable or Unattributable.

All architecturally defined events are Attributable, unless otherwise stated.

Unattributable events might be counted when Attributable events are not counted. See:

- *Interaction with EL3* on page D7-2701.
- *Event filtering* on page D7-2711.
- *Performance Monitors and Debug state* on page D7-2713.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

These sections are summarized by Table D7-1 for events Attributable to the processor, and Unattributable events.

Table D7-1 Counting events

Counter and PMU enabled	State	Allowed or prohibited	Filtered	Event type		
				If Attributable to:	Then	Else
Yes	Non-debug	Allowed	Not filtered	X	Count	Count
			Filtered	Current Exception level	Do not count	IMPLEMENTATION DEFINED
		Prohibited	X	Current Security state	Do not count	IMPLEMENTATION DEFINED
	Debug	X	X	Debugger operations or raw cycles	Do not count	IMPLEMENTATION DEFINED
No	X	X	X	X	Do not count	Do not count

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D7.5 Prohibiting event counting

Counting Attributable events in Secure state is prohibited unless any one of the following is true:

- EL3 is not implemented.
- EL3 is implemented, is using AArch64, and the value of MDCR_EL3.SPME is 1.
- EL3 is implemented, is using AArch32, and the value of SDCR.SPME is 1.
- EL3 is implemented, EL3 or EL1 is using AArch32, executing at EL0, and the value of SDER32_EL3.SUNIDEN is 1.
- If ARMv8.2-Debug is not implemented, EL3 is implemented, and counting is permitted by an IMPLEMENTATION DEFINED authentication interface, `ExternalSecureNoninvasiveDebugEnabled()` == TRUE.

  Note

  Software can read the Authentication Status register, DBGAUTHSTATUS to determine the state of an IMPLEMENTATION DEFINED authentication interface.

Counting Attributable events at EL2 is prohibited unless any of the following are true:

- ARMv8.1-PMU is not implemented.
- HDCR.HPMD is 0.
- The event is being counted by an event counter reserved by HDCR.HPMN for use by the hypervisor.

If ARMv8.4-SecEL2 is implemented, counting Attributable events at Secure EL2 is prohibited if counting events in Secure state is prohibited or counting events at EL2 is prohibited.

The accessibility of Performance Monitors registers is unaffected by whether event counting is enabled or prohibited.

The cycle counter, PMCCNTR, counts unless one of the following is true:

- Event counting is prohibited and PMCR.DP is set to 1.
- The PE is in Debug state.
- ARMv8.5-PMU is implemented, EL3 is implemented, the PE is in Secure state, and SDCR.SCCD is set to 1.
- ARMv8.5-PMU is implemented, EL2 is implemented, the PE is executing at EL2, and HDCR.HCCD is set to 1.

For each Unattributable event, it is IMPLEMENTATION DEFINED whether it is counted when counting Attributable events is prohibited.

See `AArch64.CountEvents()` and `AArch32.CountEvents()` in Chapter J1 *Armv8 Pseudocode* for more information. The `CountEvents()` functions return TRUE if PMEVCNTR<n> counts events or the cycle counter counts cycles at the current Exception level and state. However, these functions do not completely describe the behavior for Unattributable events.

The Performance Monitors are intended to be broadly accurate and statistically useful, see *Accuracy of the Performance Monitors* on page D7-2703. Some inaccuracy is permitted at the point of changing Security state, however. To avoid the leaking of information from the Secure state, the permitted inaccuracy is that transactions that are not prohibited can be uncounted. Where possible, prohibited transactions must not be counted, but if they are counted, then that counting must not degrade security.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D7.6 Multithreaded implementations

If an implementation is multithreaded and the *Effective value* of PMEVTYPER<n>.MT ==1, events on other PEs with the same level 1 Affinity are also counted. A pair of PEs have the same level 1 Affinity if they have the same values for all fields in MPIDR_EL1 or MPIDR except the Aff0 field.

Events on other PEs are not counted when the *Effective value* of PMEVTYPER<n>.MT is 0.

If the CPU implements multithreading, and ARMv8.6-MTPMU is not implemented, for Armv8.5 and earlier, it is IMPLEMENTATION DEFINED whether PMEVTYPER<n>.MT is implemented as RW or RES0. From Armv8.6, if the optional ARMv8.6-MTPMU feature is implemented, PMEVTYPER<n>.MT is RW, and if ARMv8.6-MTPMU is not implemented, PMEVTYPER<n>.MT is RES0.

If ARMv8.6-MTPMU is implemented, EL3 is implemented, and MDCR_EL3.MTPME is 0 or SDCR.MTPME is 0, ARMv8.6-MTPMU is disabled and the *Effective value* of PMEVTYPER<n>.MT is 0.

If ARMv8.6-MTPMU is implemented, EL3 is not implemented, EL2 is implemented, and MDCR_EL2.MTPME is 0 or HDCR.MTPME is 0, ARMv8.6-MTPMU is disabled and the *Effective value* of PMEVTYPER<n>.MT is 0.

If ARMv8.6-MTPMU is disabled on a Processing Element PE_A, it is IMPLEMENTATION DEFINED whether ARMv8.6-MTPMU is disabled on another Processing Element PE_B, if all of the following are true:

- ARMv8.6-MTPMUis implemented on PE_A and PE_B.
- PE_A and PE_B have the same values for Affinity level 1 and higher.
- PE_A and PE_B both have MPIDR_EL1.MT or MPIDR.MT set to 1.

However, even when the *Effective value* of PMEVTYPER<n>.MT is 1, the PE does not count an event that is Attributable to Secure state or EL2 on another thread if counting events Attributable to Secure state or EL2 is prohibited on the PE that is counting the events.

Example D7-1 The effect of having PMEVTYPER<n>.MT == 1

If the value of MDCR_EL3.SPME is 0, and <n> is less than PMCR.N on one thread, then event counter <n> on this thread does not count events Attributable to Secure state on another thread, even if one or both of the following applies:

- This thread is in Non-secure state.
- MDCR_EL3.SPME==1 on the other thread.

Example D7-2 The effect of having PMEVTYPER<n>.MT == 1

If the value of MDCR_EL2.HPMD is 1 and <n> is less than MDCR_EL2.HPMN on one thread, then event counter <n> on this thread does not count events Attributable to EL2 on another thread, even if one of the following applies:

- MDCR_EL2.HPMD==0 on the other thread.
- This thread is not executing at EL2.

When the current configuration prohibits counting of events Attributable to Secure state or EL2 in Secure state or at EL2, it is IMPLEMENTATION DEFINED whether:

- Counting events Attributable to Secure state on this PE in Non-secure state is permitted.
- Counting events Attributable to EL2 when this PE is using another Exception level is permitted
- Counting Unattributable events related to other secure operations in the system or at EL2 is permitted.

If not specified above, counting events that are not prohibited on either PE is permitted.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D7.7 Event filtering

The PMU can filter events by various combinations of Exception level and Security state. This gives software the flexibility to count events across multiple processes.

D7.7.1 Filtering by Exception level and PE state

In AArch64 state:

- For each event counter PMEVTYPER<n>_EL0 specifies the Exception levels in which the counter counts events Attributable to Exception levels.
- PMCCFILTR_EL0 specifies the Exception levels in which the cycle counter counts.

In an implementation that supports multithreading:

- When the *Effective value* value of PMEVTYPER<n>_EL0.MT is 1, if an event is Attributable to another thread, then the specified filtering applies to the current Exception level and PE state of the thread to which the event is attributable, regardless of the Exception level and state of the counting thread.
- When the *Effective value* value of PMEVTYPER<n>_EL0.MT is 0. the event only counts events that are attributable to the counting thread, and the filtering applies to the Exception level and PE state of the counting state, see Example D7-3.

Example D7-3 Example of the effect of the PMEVTYPER<n>_EL0.MT control

---

If the value of PMEVTYPER<n>_EL0.U is 0 on the current thread, then it does not count events Attributable to EL0 on the other thread, even if this thread is not executing at EL0.

Otherwise, for each Unattributable event, it is IMPLEMENTATION DEFINED whether the filtering applies.

In AArch32 state, the filtering controls are provided by the PMEVTYPER<n> and PMCCFILTR registers.

For more information, see the individual register descriptions and *Multithreaded implementations* on page D7-2710.

D7.7.2 Accuracy of event filtering

For most events, it is acceptable that, during a transition between states, events generated by instructions executed in one state are counted in the other state. The following sections describe the cases where event counts must not be counted in the wrong state:

- *Exception-related events*.
- *Software increment events* on page D7-2712.

Exception-related events

The PMU must filter events related to exceptions and exception handling according to the Exception level in which the event occurred. These events are:

- EXC_TAKEN, Exception taken.
- EXC_RETURN, Instruction architecturally executed, Condition code check pass, exception return.
- CID_WRITE_RETIRED, Instruction architecturally executed, Condition code check pass, write to CONTEXTIDR.
- TTBR_WRITE_RETIRED, Instruction architecturally executed, Condition code check pass, write to translation table base.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The PMU must not count an exception after it has been taken because this could systematically report a result of zero exceptions at EL0. Similarly, it is not acceptable for the PMU to count exception returns or writes to CONTEXTIDR after the return from the exception.

Software increment events

The PMU must filter software increment events according to the Exception level in which the software increment occurred. Software increment counting must also be precise, meaning the PMU must count every architecturally executed software increment event, and must not count any Speculatively executed software increment.

Software increment events must also be counted without the need for explicit synchronization. For example, two software increments executed without an intervening *Context synchronization event* must increment the event counter twice.

For more information, see SW_INCR, Instruction architecturally executed, Condition code check pass, software increment.

D7.7.3 Pseudocode description of event filtering

See `AArch64.CountEvents()` and `AArch32.CountEvents()` in Chapter J1 *Armv8 Pseudocode* for a pseudocode description of event filtering. However, this function does not completely describe the behavior for Unattributable events.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D7.8 Performance Monitors and Debug state

Events that count cycles are not counted in Debug state.

Events Attributable to the operations issued by the debugger through the external debug interface are not counted in Debug state.

In an implementation that supports multithreading, when the *Effective value* value of PMEVTYPER<n>_EL0.MT is 1, if an event is Attributable to an operation issued by the debugger through the external debug interface to another thread that is in Debug state, then the event is not counted, and it is IMPLEMENTATION DEFINED whether the event is counted when the counting thread is in Debug state.

For each Unattributable event, it is IMPLEMENTATION DEFINED whether it is counted when the counting PE is in Debug state. If the event might be counted, then the rules in *Filtering by Exception level and PE state* on page D7-2711 apply for the current Security state in Debug state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D7.9 Counter enables

Table D7-2 shows an implementation that does not include EL2, and where the PMCR.E bit is a global counter enable bit, and PMCNTENSET provides an enable bit for each counter.

Table D7-2 Event counter enables when an implementation does not include EL2

PMCR.E	PMCNTENSET[n] == 0	PMCNTENSET[n] == 1
0	PMEVCNTR<n> disabled	PMEVCNTR<n> disabled
1	PMEVCNTR<n> disabled	PMEVCNTR<n> enabled

If the implementation includes EL2, then in addition to the PMCR.E and PMCNTENSET enable bits:

- HDCR.HPME overrides the value of PMCR.E for counters configured for access in Hyp mode.
- HDCR.HPMN specifies the number of performance counters that the Guest OS can access. The minimum permitted value of HDCR.HPMN is 1, meaning there must be at least one counter that the Guest OS can access.

Table D7-3 shows the combined effect of all the counter enable controls.

Table D7-3 Event counter enables when an implementation includes EL2

HDCR.HPME	PMCR.E	PMCNTENSET[n] == 0	PMCNTENSET[n] == 1	
			n < HDCR.HPMN	n ≥ HDCR.HPMN
0	0	PMEVCNTR<n> disabled	PMEVCNTR<n> disabled	PMEVCNTR<n> disabled
0	1	PMEVCNTR<n> disabled	PMEVCNTR<n> enabled	PMEVCNTR<n> disabled
1	0	PMEVCNTR<n> disabled	PMEVCNTR<n> disabled	PMEVCNTR<n> enabled
1	1	PMEVCNTR<n> disabled	PMEVCNTR<n> enabled	PMEVCNTR<n> enabled

Note

- The effect of HDCR.{HPME, HPMN} on the counter enables applies at all Exception levels and in both Security states.
- The value returned for PMCR.N is not affected by HDCR.HPMN at:
  - EL3.
  - EL2.
  - Secure EL1, if ARMv8.4-SecEL2 is not implemented or Secure EL2 is disabled.
  - Secure EL0, if ARMv8.4-SecEL2 is not implemented or Secure EL2 is disabled.

EL2 does not affect the enabling of PMCCNTR. Table D7-4 shows the PMCCNTR enables, for all implementations.

Table D7-4 Cycle counter enables

PMCR.E	PMCNTENSET[31] == 0	PMCNTENSET[31] == 1
0	PMCCNTR disabled	PMCCNTR disabled
1	PMCCNTR disabled	PMCCNTR enabled

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D7.10 Counter access

All implemented counters are accessible in EL3 and EL2. If EL2 is implemented the hypervisor uses HDCR.HPMN to reserve an event counter, with the effect that if EL2 is enabled in the current Security state, software cannot access that counter and its associated state from EL0 or EL1.

If ARMv8.6-FGT is implemented, if PMSELR.SEL or n indicates an unimplemented event counter, access to PMXEVTYPER, PMXEVCNTR, PMEVTYPER<n>, or PMEVCNTR<n> is UNDEFINED.

D7.10.1 PMEVCNTR<n> event counters

Table D7-5 shows how the number of implemented counters, PMCR.N, and if EL2 is implemented, the value of the HDCR.HPMN field affects the behavior of permitted accesses to the PMEVCNTR<n> event counter registers for values of n from 0 to 30.

Table D7-5 Result of PMEVCNTR<n> event counter accesses

Condition	Access at Exception level			
	EL3	EL2	EL1	EL0
n < PMCR.HPMN and either EL2 is not implemented or EL2 is disabled in the current Security state	Succeeds	n/a	Succeeds	Succeeds
n < HDCR.HPMN and EL2 is implemented and enabled in the current Security state	Succeeds	Succeeds	Succeeds	Succeeds
n ≥ HDCR.HPMN and *n* < PMCR.HPMN and EL2 is implemented and enabled in the current Security state	Succeeds	Succeeds	No access	No access
n ≥ PMCR.N	No access	No access	No access	No access

Where Table D7-5 shows access succeeds for an event counter <n>, the access might be UNDEFINED or generate a trap exception. See the descriptions of PMEVCNTR<n> and PMXEVCNTR for details.

Where Table D7-5 shows no access for an event counter <n>:

- When PMSELR.SEL is *n*, the PE prevents direct reads and direct writes of PMXEVTYPER or PMXEVCNTR. See the register descriptions for more information.
- The PE prevents direct reads and direct writes of PMEVTYPER<n> or PMEVCNTR<n>. See the register descriptions for more information.
- Direct reads and direct writes of the following registers are RAZ/WI. PMOVSCLR[*n*], PMOVSSET[*n*], PMCNTENSET[*n*], PMCNTENCLR[*n*], PMINTENSET[*n*], and PMINTENCLR[*n*].
- Direct writes to PMSWINC[n] are ignored.
- A direct write of 1 to PMCR.P does not reset PMEVCNTR<n>.

D7.10.2 Cycle counter

The PMU does not provide any control that a hypervisor can use to reserve the cycle counter for its own use. However, access to the PMU registers are subject to the access permissions described in *Configurable instruction enables and disables, and trap controls* on page D1-2373.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D7.11 PMU events and event numbers

The following sections describe the events that can be counted and their associated event numbers, and the mnemonics for the events:

- *Definitions*.
- *The PMU event number space and common events* on page D7-2721.
- *Common event numbers* on page D7-2722.
- *Cycle event counting on multithreaded implementations* on page D7-2768.
- *Meaningful ratios between common microarchitectural events* on page D7-2769.
- *Required events* on page D7-2769.
- *IMPLEMENTATION DEFINED event numbers* on page D7-2771.

D7.11.1 Definitions

The following subsections give more information about terms used in the event definitions:

- *Definition of terms*.
- *Levels of caches and TLBs* on page D7-2721.
- *Shared caches and buses* on page D7-2721.

Definition of terms

CSIZE Container size, in bits, that corresponds to the largest non-overlapping SVE or Advanced SIMD vector element size or scalar register size that is encoded in the instruction opcode. This excludes the 64-bit elements of the wide element variants of the SVE bitwise shift and integer compare instructions that overlap the narrower source and destination elements.

Instruction architecturally executed

Instruction architecturally executed is a class of event that counts for each instruction of the specified type. Architecturally executed means that the program flow is such that the counted instruction would be executed in a *Simple sequential execution* of the program. Therefore an instruction that has been executed and retired is defined to be *architecturally executed*. When a PE can perform speculative execution, an instruction is not architecturally executed if the PE discards the results of the speculative execution.

If an instruction that would be executed in a *Simple sequential execution* of the program generates a synchronous exception, it is IMPLEMENTATION DEFINED whether the instruction is counted.

Each architecturally executed instruction is counted once, even if the implementation splits the instruction into multiple operations. Instructions that have no visible effect on the architectural state of the PE are architecturally executed if they form part of the architecturally executed program flow. The point where such instructions are retired is IMPLEMENTATION DEFINED.

Examples of instructions that have no visible effect are:

- A `NOP`.
- A conditional instruction that fails its Condition code check.
- A Compare and Branch on Zero, `CBZ`, instruction that does not branch.
- A Compare and Branch on Nonzero, `CBNZ`, instruction that does not branch.

The point at which an event causes an event counter to be updated is not defined.

Unless otherwise stated, all instructions of the specified type are counted even if they have no visible effect on the architectural state of the PE. This includes a conditional instruction that fails its Condition code check.

For events that count only the execution of instructions that update context state, such as writes to the CONTEXTIDR, if such an instruction is executed twice without an intervening *Context synchronization event*, it is CONSTRAINED UNPREDICTABLE whether the first instruction is counted.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Instruction architecturally executed, Condition code check pass

Instruction architecturally executed, Condition code check pass is a class of events that explicitly do not occur for:

- A conditional instruction that fails its Condition code check.
- A Compare and Branch on Zero, `CBZ`, instruction that does not branch.
- A Compare and Branch on Nonzero, `CBNZ`, instruction that does not branch.
- A Test and Branch on Zero, `TBZ`, instruction that does not branch.
- A Test and Branch on Nonzero, `TBNZ`, instruction that does not branch.
- A Store-Exclusive instruction that does not write to memory.

Otherwise, the definition of architecturally executed is the same as for Instruction architecturally executed.

A branch that is architecturally executed, with condition code check pass is also described as a branch taken.

Instruction memory access

A PE acquires instructions for execution through instruction fetches. Instruction fetches might be due to:

- Fetching instructions that are architecturally executed.
- The result of the execution of an instruction preload instruction, `PLI`.
- Speculation that a particular instruction might be executed in the future.

The relationship between the fetch of an individual instruction and an instruction memory access is IMPLEMENTATION DEFINED. For example, an implementation might fetch many instructions including a non-integer number of instructions in a single instruction memory access.

Memory-read operations

A PE accesses memory through memory-read operations and Memory-write operations. A memory-read operation might be due to:

- The result of an architecturally executed memory-reading instructions.
- The result of a Speculatively executed memory-reading instructions.
- A translation table walk.

For levels of cache hierarchy beyond the Level 1 caches, memory-read operations also include accesses made as part of a refill of another cache closer to the PE. Such refills might be due to:

- Memory-read operations or Memory-write operations that miss in the cache
- The execution of a data preload instruction.
- The execution of an instruction preload instruction on a unified cache.
- The execution of a cache maintenance instruction.

  Note

  A preload instruction or cache maintenance instruction is not, in itself, an access to that cache. However, it might generate cache refills which are then treated as memory-read operations beyond that cache.

- Speculation that a future instruction might access the memory location.
- Instruction memory accesses.

This list is not exhaustive.

The relationship between memory-read instructions and memory-read operations is IMPLEMENTATION DEFINED. For example, for some implementations an `LDP` instruction that reads two 64-bit registers might generate one memory-read operation if the address is quadword-aligned, but for other addresses it generates two or more memory-read operations.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Memory-write operations

Memory-write operations might be due to:

- The result of an architecturally executed memory-writing instructions.
- The result of a Speculatively executed memory-writing instructions.

Note

Speculatively executed memory-writing instructions that do not become architecturally executed must not alter the architecturally defined view of memory. They can, however, generate a memory-write operation that is later undone in some implementation specific way.

For levels of cache hierarchy beyond the Level 1 caches, memory-write operations also include accesses made as part of a write-back from another cache closer to the PE. Such write-backs might be due to:

- Evicting a dirty line from the cache, to allocate a cache line for a cache refill, see Memory-read operations.
- The execution of a cache maintenance instruction.

  Note

  A cache maintenance instruction is not in itself an access to that cache. However, it might generate write-backs which are then treated as memory-write operations beyond that cache.

- The result of a coherency request from another PE.

This list is not exhaustive.

The relationship between memory-writing instructions and memory-write operations is IMPLEMENTATION DEFINED. For example, for some implementations an `STP` instruction that writes two 64-bit registers might generate one memory-write operation if the address is quadword-aligned, but for other addresses it generates two or more memory-write operations. In some implementations, the result of two `STR` instructions that write to adjacent memory might be merged into a single memory-write operation.

Note

The data written back from a cache that is shared with other PEs might not be data that was written by the PE that performs the operation that leads to the write-back. Nevertheless, the event is counted as a write-back event for that PE.

Microarchitectural operation

It is permissible for an implementation of a PE to break down instructions into separate, smaller, operations. The use of Microarchitectural operations (micro-ops) is IMPLEMENTATION DEFINED.

An instruction might create one or more micro-ops at any point in the execution pipeline. For the purpose of event counting, the micro-ops are counted. The definition of a micro-op is implementation specific. An architecture instruction might create more than one micro-op for each instruction. micro-ops might also be removed or merged in the execution stream, so an architecture instruction might create no micro-ops for an instruction. Any arbitrary translation of instructions to an equivalent sequence of micro-ops is permitted.

The counting of operations can indicate the workload on the PE. However, there is no requirement for operations to represent similar amounts of work, and direct comparisons between different microarchitectures are not meaningful.

For example, an implementation might split an A32 or T32 `LDM` instruction of six registers into six micro-ops, one for each load, and a seventh address-generation operation to determine the base address or writeback address. Also, for doubleword alignment, the six load micro-ops might combine into four operations, that is, a word load, two doubleword loads, and a second word load. This single instruction can then be counted as five, or possibly six, events:

- Four (Operations speculatively executed - Load) events.
- One (Operations speculatively executed - Integer data processing) event.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- One (Operations speculatively executed - Software change of the PC) event if the PC was one of the six registers in the `LDM` instruction.

MSIZE Memory element access size, in bits, that corresponds to a load or store instruction mnemonic suffix, where B=8, H=16, W=32 and D=64. When an instruction mnemonic does not end with B, H, W or D, the memory access size is implied by the scalar transfer register size or SIMD transfer register element size.

non-SIMD SVE instructions

These are instructions listed in the non-SIMD SVE instruction category in the *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*.

Operation counts for dot-product and multiply-accumulate operations

Table D7-6 gives the operation counts for any combination of an applicable instruction when a PMU event in the range 0x80C0 to 0x80CF is implemented.

In this table:

Input size The element size of input operands other than the accumulator.

Acc size The element size of the accumulator operand.

Count The number of addition and multiply operations per 128 bits of input:

- Scalable vector operations increment the counter by the Count value for an applicable *_SCALE_OPS_SPEC event.
- Advanced SIMD operations operating on a 128-bit register increment the counter by the Count value for an applicable *_FIXED_OPS_SPEC event.
- Advanced SIMD operations operating on a 64-bit register increment the counter by half the Count value for an applicable *_FIXED_OPS_SPEC event.

Type The data type classification for the operations. This determines for which events the event counter counts the operation.

Predicated operations are counted even if the Governing predicate for the element is FALSE.

Note

The FP64 `FMMLA` instruction works on 256-bit segments, and performs 16 operations per 256-bit segment. The table represents counts per 128 bits of input, so the counter increments by 8.

Table D7-6 Operation counts

Operation	Input size	Acc size	Count	Type
`SDOT`, `UDOT`, `USDOT`, `SUDOT`	8 bits	32 bits	32	Integer
`SDOT`, `UDOT`	16 bits	64 bits	16	Integer
`BFDOT`	16 bits	32 bits	16	Single-precision floating point
`BFMMLA`	16 bits	32 bits	32	Single-precision floating point
`BFMLAL`, `FMLAL`, and `FMLSL`	16 bits	32 bits	8	Single-precision floating point
`SMMLA`, `UMMLA`, or `USMMLA`	8 bits	32 bits	64	Integer
`FMMLA`	32 bits	32 bits	16	Single-precision floating point
`FMMLA`	64 bits	64 bits	8	Double-precision floating point

Note

Predicated operations are counted even if the governing predicate for the element is FALSE.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operations speculatively executed

There is no architecturally guaranteed relationship between a Speculatively executed micro-op and an architecturally executed instruction.

The results of such an operation can also be discarded, if it transpires that the operation was not required, following a mispredicted branch. Therefore, Armv8-A defines these events as *operations speculatively executed*, where appropriate.

Slot

An implementation of a PE might be able to execute multiple micro-ops in a single processor cycle. The maximum number of micro-ops that can be executed might vary at different points in the execution pipeline.

To allow profiling of the utilization of the resource of the PE, an implementation specific point in the execution pipeline is chosen where the maximum number of micro-ops that can be executed is an IMPLEMENTATION DEFINED fixed value.

Each possible micro-op that can be executed at that point in a cycle is called a Slot. The maximum number of micro-ops that can be executed is defined by PMMIR.SLOTS.

Software change of the PC

Some event relate to instructions that cause a software change of the PC. This includes all:

- Branch instructions.
- Memory-reading instructions that explicitly write to the PC.
- Data-processing instructions that explicitly write to the PC.
- Exception return instructions.

It is IMPLEMENTATION DEFINED whether any or all of the following are treated as software changes of the PC:

- `BRK` and `BKPT` instructions.
- An exception generated because an instruction is UNDEFINED.
- The exception-generating instructions, `SVC`, `HVC`, and `SMC`.
- Context synchronization barrier `ISB` instructions.

Speculatively executed

Many events relate to speculatively executed operations. Here, speculatively executed means the PE did some work associated with one or more instructions but the instructions were not necessarily architecturally executed.

See Operations speculatively executed for speculation of micro-ops.

Note

The definition of *speculatively executed* does not mean only those operations that are executed speculatively and later abandoned, for example due to a branch misprediction or fault. That is, speculatively executed operations must count operations on both false and correct execution paths.

Different groups of events can have different IMPLEMENTATION DEFINED definitions of speculatively executed. Such groups share a common base type, which the event name denotes. Each of the events in the previous example is of the base type, operation speculatively executed.

For groups of events with a common base type, speculatively executed operations are all counted on the same basis, which normally means at the same point in the pipeline. It is possible to compare the counts and make meaningful observations about the program being profiled.

Within these groups, events are commonly defined with reference to a particular architecture instruction or group of instructions. In the case of speculatively executed operations this means operations with semantics that map to that type of instruction.

VL The current SVE vector length, in bits.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Levels of caches and TLBs

The mapping of levels of cache and TLB to the PMU events is IMPLEMENTATION DEFINED. Although CLIDR_EL1 and CLIDR define the implemented levels of cache, these are not required to correspond with the levels of cache defined for PMU events. The architecture does not provide any way of determining implemented levels of TLB. Also, many implementations include structures that provide some caching at a higher level than the level 1 caches or TLBs. Typically, these structures, that might be called Level 0 caches, or mini caches, or microcaches, are invisible to software. The implementation-specific nature of cache and TLB implementations mean that, in general, PMU event counts cannot be used reliably to make direct comparisons between different implementations.

Shared caches and buses

There is no architectural concept of a *shared* component. However, when a cache, a bus, or any other system component that might generate countable events is implemented, and:

- The extent of the first-order effects due to an event from that component are only applicable to a single PE, then the event is not shared.
- Otherwise, the event is shared.

Second-order effects are not considered when determining if an event is shared.

Example D7-4 First and second order effects of a cache miss in a multiple-PE implementation

In an implementation that consists of two PEs, each with its own L1 cache, a cache miss by one of the PEs is a first-order effect of an access to its cache. Any snoop that is performed on the L1 cache of the other PE in the implementation as a result of that cache miss is a second order effect.

Note

Shared events are inherently linked to microarchitectures and so the implementer must make an informed decision about how such events are implemented.

D7.11.2 The PMU event number space and common events

In Armv8.0, the event number space is 10 bits. Armv8.1 extends the event number space, and therefore the PMEVTYPER<n>_EL0.evtCount field to 16 bits, and is allocated as Table D7-7 shows. For more information about the entries in the *Allocation* column see the text that follows this table:

Table D7-7 Allocation of the PMU event number space

Event numbers	Allocation
In all versions of Armv8	
`0x0000-0x003F`	Common architectural and microarchitectural events.
`0x0040-0x00BF`	Arm-recommended common architectural and microarchitectural events.
`0x00C0-0x03FF`	IMPLEMENTATION DEFINED events.
From Armv8.1	
`0x0400-0x3FFF`	IMPLEMENTATION DEFINED events.
`0x4000-0x403F`	Common architectural and microarchitectural events.
`0x4040-0x40BF`	Arm-recommended common architectural and microarchitectural events.
`0x40C0-0x7FFF`	IMPLEMENTATION DEFINED events.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D7-7 Allocation of the PMU event number space (continued)

Event numbers	Allocation
`0x8000-0x80FF`	Common architectural and microarchitectural events.
`0x8100-0x81FF`	Previously Reserved. From Armv8.6, Common architectural and microarchitectural events.
`0x8200-0xC0BF`	Reserved.
`0xC0C0-0xFFFF`	IMPLEMENTATION DEFINED events.

The meaning of the entries in the *Allocation* on page D7-2721 column of Table D7-7 on page D7-2721 is as follows:

Common architectural and microarchitectural events

Arm defines the use of these event numbers. For more information see *Common event numbers*.

Arm-recommended common architectural and microarchitectural events

The use of these event numbers is IMPLEMENTATION DEFINED. For more information see:

- *IMPLEMENTATION DEFINED event numbers* on page D7-2771.
- Appendix K3 *Recommendations for Performance Monitors Event Numbers for IMPLEMENTATION DEFINED Events*.

Common architectural and microarchitectural SVE events

Arm defines the use of these event numbers. See *Common event numbers*. An implementation is not required to implement SVE in order to implement these events.

IMPLEMENTATION DEFINED event numbers

For more information about the use of these event numbers see *IMPLEMENTATION DEFINED event numbers* on page D7-2771.

D7.11.3 Common event numbers

The event numbers of the common architectural and microarchitectural events are reserved for the specified events. Each of these event numbers must either:

- Be used for its assigned event.
- Not be used.

However, see *Required events* on page D7-2769.

When an implementation supports monitoring of an event that is assigned a common architectural or microarchitectural event number, Arm strongly recommends that it uses that number for the event. However, software might encounter implementations where an event assigned a number in this range is monitored using an event number from an IMPLEMENTATION DEFINED range.

Note

Arm might define other common architectural and microarchitectural event numbers. This is one reason why software must not assume that an event with an assigned common architectural or microarchitectural event number is never monitored using an event number from the IMPLEMENTATION DEFINED range.

Common events:

- *Common architectural events* on page D7-2723.
- *Common microarchitectural events* on page D7-2730.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The supported common architectural and microarchitectural events in the ranges 0x0000-0x003F and 0x4000-0x403F are discoverable to software through:

- The PMCEID0_EL0 and PMCEID1_EL0 registers in AArch64 state.
- The PMCEID0, PMCEID1, PMCEID2, and PMCEID3 registers in AArch32 state.

Arm recommends that the value of 0 is used for the PMCEID0_EL0 or PMCEID1_EL0 bit corresponding to any event that an implementation never generates, even if the implementation is considered to support but never count the event.

Note

- For example, if an implementation never generates the L1D_CACHE_ALLOCATE event, event 31, Arm recommends that PMCEID0_EL0[31] is RAZ.
- In an implementation that supports both Execution states, each bit in the AArch64 PMCEID0_EL0 and PMCEID1_EL0 registers corresponds to a single bit in the AArch32 PMCEID0, PMCEID1, PMCEID2, and PMCEID3 registers, and corresponding bits must have the same behavior.

However, for some implementations, an event in the common events range might be generated by the system, meaning behavior can vary between systems. In such a case, the corresponding PMCEID*n*_EL0 bit might be RAO.

Event numbers that Table D7-7 on page D7-2721 shows as allocated for common architectural and microarchitectural events that are not described in *Common architectural events* and *Common microarchitectural events* on page D7-2730 are reserved. Future revisions of this manual, or of the architecture, might assign these reserved values to additional common events. Events that do not require additional features in the PMU can be implemented retrospectively, meaning an implementation of a particular version of the PMU specification might support common events that are first defined in a later version of the PMU specification.

Note

- The requirement that an event that is implemented retrospectively *does not require additional features in the PMU* means that it must be possible to represent the event n the PMEVTYPER<n>_EL0.evtCount field. This means, for example, that an implementation with a 12-bit PMEVTYPER<n>_EL0.evtCount field can only implement events with event numbers `0x000-0xFFF`.
- This means that, for example, an Armv7 PMUv2 implementation, for which the evtCount field is 8 bits, can include support for any of the event numbers that are described in *Common architectural events* and *Common microarchitectural events* on page D7-2730 define in the range `0x00-0xFF`.

Common architectural events

This section describes the use of the defined common architectural event numbers.

For the common features, normally the counters must increment only once for each event. The event descriptions include any exceptions to this rule.

In these definitions, the term *architecturally executed* means that the instruction flow is such that the counted instruction would have been executed in a *Simple sequential execution* model.

The events corresponding to the common architectural event numbers are:

`0x0000`, SW_INCR, Instruction architecturally executed, Condition code check pass, software increment

The counter increments on writes to the PMSWINC register.

If the PE performs two architecturally executed writes to the PMSWINC register without an intervening *Context synchronization event*, then the counter is incremented twice.

If PMEVTYPER<n>_EL0.evtCount is set to 0x0000, then in AArch64 state, counts `MSR` writes to PMSWINC_EL0 with bit [n] set to 1.

If the value of PMEVTYPER<n>_EL0.MT is 1 then, in a multithreaded implementation, this counts writes by all PEs that have the same affinity at level 1 and above.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0x0006`, **LD_RETIRED, Instruction architecturally executed, Condition code check pass, load**

The counter increments for every executed memory-reading instruction.

Note

The counter `0x006` does not count the return status value of a Store-Exclusive instruction.

Whether the preload instructions `PRFM`, `PLD`, `PLDW`, `PLI`, count as memory-reading instructions is IMPLEMENTATION DEFINED. Arm recommends that if the instruction is not implemented as a `NOP` then it is counted as a memory-reading instruction.

`0x0007`, **ST_RETIRED, Instruction architecturally executed, Condition code check pass, store**

The counter increments for every executed memory-writing instruction.

`DC ZVA` is counted as a store.

The counter does not increment for a Store-Exclusive instruction that fails.

`0x0008`, **INST_RETIRED, Instruction architecturally executed**

The counter increments for every architecturally executed instruction.

It is IMPLEMENTATION DEFINED whether the counter increments for the `MOVPRFX` instruction.

`0x0009`, **EXC_TAKEN, Exception taken**

The counter increments for each exception taken. See *Exception-related events* on page D7-2711.

Note

The counter counts the PE exceptions described in:

- For exceptions taken to an Exception level using AArch64, *Exception entry* on page D1-2339.
- For exceptions taken to an Exception level using AArch32, *AArch32 state exception descriptions* on page G1-5738.

`0x000A`, **EXC_RETURN, Instruction architecturally executed, Condition code check pass, exception return**

The counter increments for each executed exception return instruction. See also *Exception-related events* on page D7-2711. The following sections define the counted instructions:

- For an exception return from an Exception level using AArch64, *Exception return* on page D1-2350.
- For an exception return from an Exception level using AArch32, *Exception return instructions* on page G1-5725.

However, is CONSTRAINED UNPREDICTABLE whether this event counts the execution of an exception return instruction if either:

- Execution of the instruction is, itself, CONSTRAINED UNPREDICTABLE.

  Note

  Examples of when an exception return instruction is CONSTRAINED UNPREDICTABLE are if the instruction is executed at EL0, or in AArch32 state in System mode.

- Execution of the instruction sets PSTATE.IL and does not generate an exception return.

Note

A particular consequence of this CONSTRAINED UNPREDICTABLE behavior is that an implementation that does not support AArch32 state at EL1 or higher does not have to treat AArch32 `MOVS PC, LR` instructions, and related instructions, as exception return instructions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

0x000B, CID_WRITE_RETIRED, Instruction architecturally executed, Condition code check pass, write to CONTEXTIDR

The counter increments for every write to CONTEXTIDR. See *Exception-related events* on page D7-2711.

If the PE performs two architecturally-executed writes to CONTEXTIDR without an intervening *Context synchronization event*, it is CONSTRAINED UNPREDICTABLE whether the first write is counted.

When Armv8.1-VHE is implemented, the counter is:

- Incremented as a result of the retirement of an instruction accessing the named register CONTEXTIDR_EL1, even when executing at EL2.
- Not incremented as a result of the retirement of an instruction accessing the named CONTEXTIDR_EL12.

Note

The event is defined by the name used to access the register. The counter does not count writes to the named register CONTEXTIDR_EL2.

0x000C, PC_WRITE_RETIRED, Instruction architecturally executed, Condition code check pass, software change of the PC

The counter increments for every Software change of the PC.

The counter does not increment for exceptions other than those explicitly identified as a Software change of the PC.

If PC_WRITE_RETIRED and BR_SKIP_RETIRED events are both implemented, the PE must have a consistent definition of Software change of the PC instructions. This means the definition must treat the following instructions in the same way for both events:

- `BRK` and `BKPT` instructions.
- An exception generated because an instruction is UNDEFINED.
- The exception-generating instructions, `SVC`, `HVC`, and `SMC`.
- Context synchronization barrier `ISB` instructions.

From Armv8.6, when BR_RETIRED is also implemented, the PE must treat these instructions in the same way for BR_RETIRED, PC_WRITE_RETIRED and BR_SKIP_RETIRED.

Note

Conditional branches are only counted if the branch is taken.

0x000D, BR_IMMED_RETIRED, Instruction architecturally executed, immediate branch

The counter counts all immediate branch instructions on the architecturally executed path.

In AArch32 state, the counter increments each time the PE executes one of the following instructions:

- `B{<c>} <label>`.
- `BL{<c>} <label>`.
- `BLX{<c>} <label>`.
- `CBZ <Rn>, <label>`.
- `CBNZ <label>`.

In AArch64 state, the counter increments each time the PE executes an immediate branch instructions:

- `B <label>`.
- `B.cond <label>`.
- `BL <label>`.
- `CBZ <Rn>, <label>`.
- `CBNZ <Rn>, <label>`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- `TBZ <Rn>, <label>`.
- `TBNZ <Rn>, <label>`.

Note

Conditional branches are always counted, regardless of whether the branch is taken.

If an `ISB` is counted as a software change of the PC instruction, then it is IMPLEMENTATION DEFINED whether an `ISB` is counted as an immediate branch instruction.

`0x000E`, BR_RETURN_RETIRED, Instruction architecturally executed, Condition code check pass, procedure return

In AArch32 state, the counter counts the following procedure return instructions:

- `BX R14`.
- `MOV PC, LR`.
- `POP {…, PC}`.
- `LDR PC, [SP], #offset`.

Note

The counter counts only the listed instructions as procedure returns. For example, it does not count the following as procedure return instructions:

- `BX R0`, because Rm != R14.
- `MOV PC, R0`, because Rm != R14.
- `LDM SP, {…, PC}`, because writeback is not specified.
- `LDR PC, [SP, #offset]`, because this specifies the wrong addressing mode.

In AArch64 state, the counter counts all architecturally executed `RET` instructions.

`0x000F`, UNALIGNED_LDST_RETIRED, Instruction architecturally executed, Condition code check pass, unaligned load or store

The counter counts each memory-reading instruction or memory-writing instruction access that would generate an Alignment fault when Alignment fault checking is enabled.

The counter does not count accesses that would generate an SP alignment fault exception if the applicable stack pointer alignment check is enabled, unless that access would also generate an Alignment fault Data Abort exception if Alignment fault checking is enabled.

It is IMPLEMENTATION DEFINED whether this event counts accesses that generate an exception, including accesses that do generate Alignment fault Data Abort exceptions.

See *SP alignment checking* on page D1-2333 for more information.

See *Unaligned data access* on page E2-4004 for more information.

`0x001C`, TTBR_WRITE_RETIRED, Instruction architecturally executed, Condition code check pass, write to TTBR

The counter counts writes to TTBR0_EL1 and TTBR1_EL1 in AArch64 state and TTBR0 and TTBR1 in AArch32 state. When EL3 is implemented and using AArch32, this includes counting writes to both banked copies of TTBR0 and TTBR1. See *Exception-related events* on page D7-2711.

If the PE executes two writes to the same TTBR, without an intervening *Context synchronization event*, it is CONSTRAINED UNPREDICTABLE whether the first write to the TTBR, is counted.

If EL3 is implemented and using AArch64, the counter does not count writes to TTBR0_EL3.

If EL2 is implemented and using AArch64, the counter does not count writes to TTBR0_EL2 and to VTTBR_EL2.

If EL2 is implemented and using AArch32, the counter does not count writes to HTTBR and to VTTBR.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When Armv8.1-VHE is implemented, the counter is:

- Incremented as a result of the retirement of an instruction accessing the named registers TTBR0_EL1 and TTBR1_EL1.
- Not incremented as a result of the retirement of an instruction accessing the named registers TTBR0_EL12 and TTBR1_EL12.

`0x001E`, **CHAIN**

Even-numbered counters never increment as a result of this event. For an odd-numbered counter *n+1*, if ARMv8.5-PMU is implemented, the odd-numbered event counter does not increment if:

- EL2 is not implemented and PMCR.LP is set to 1.
- EL2 is implemented, <*n*> is less than HDCR.HPMN and PMCR.LP is set to 1.
- EL2 is implemented, <*n+1*> is greater than or equal to HDCR.HPMN and HDCR.HLP is set to 1.

Otherwise, the odd-numbered event counter *n+1* increments when an event increments the preceding even-numbered counter *n* on the same PE and causes an unsigned overflow of bits [31:0] of event counter *n*.

This means the CHAIN event can be used to link the odd-numbered counter with the preceding even-numbered counter to provide a 64-bit counter.

Note

When ARMv8.5-PMU is not implemented, the CHAIN event can be used by software to provide *N* 32-bit counters, *N*/2 64-bit counters, or a mixture of 32-bit counters and 64-bit counters.

The CHAIN event only counts overflows from the preceding even-numbered counter on the same PE. This means it is unaffected by the value of PMEVTYPER<n>_EL0.MT.

To filter the Exception levels and Security states in which the event is counted, software must:

- Program PMEVTYPER<n>_EL0 to count the event in the required conditions.
- Program PMEVTYPER<n+1>_EL0 to count the CHAIN event in all Exception levels and states.

This allows, but does not require, hardware to ignore the filter settings for the CHAIN event and behave as if they are set to count in all Exception levels and states.

If software does not program the event in this way, the count becomes UNPREDICTABLE.

There is no atomic access to a pair of counters, so if software reads a counter-pair that is enabled, it must use a high-low-high read sequence, or employ reasonable heuristics, to avoid tearing. Similarly, if using CHAIN events, when disabling the counters software must take care that the result is not torn by the low counter overflowing at the same time as the counters are disabled Example D7-5 shows suitable sequences for disabling and enabling CHAIN counters.

Example D7-5 Usage examples for 64-bit counters

An example high-low-high read sequence for a 64-bit counter created by a pair of 32-bit counters paired by a CHAIN event is:

```
retry:
 MRS W2,PMEVCNTR1_EL0   ;; read high counter, must be odd-numbered
 ISB                    ;; force ordering
 MRS W0,PMEVCNTR0_EL0   ;; read low counter
                        ;; must return the previous counter to PMEVCNTR1_EL0
 ISB                    ;; force ordering
 MRS W1,PMEVCNTR1_EL0   ;; read high counter
 CMP W1,W2
 BNE retry              ;; if the high counter has changed, then retry
```

When disabling a pair of counters that are paired by a CHAIN event, software must:

1. Disable the low counter, by setting PMEVCNTR<n>_EL0[n] to 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Typically, software uses a read-modify-write sequence to update PMCNTENCLR_EL0.

2. Execute an ISB instruction, or perform another *Context synchronization event*.
3. Disable the high counter, by setting PMCNTENCLR_EL0[n+1] to 1, or setting PMCR_EL0.E to 0.

When enabling a pair of counters that are paired by a CHAIN event, software must:

1. Enable the high counter, by setting PMCNTENCLR_EL0[n+1] to 0 and, if necessary, setting PMCR_EL0.E to 1.
2. Execute an ISB instruction, or perform another *Context synchronization event*.
3. Enable the low counter by setting PMCNTENCLR_EL0[n] to 0.

When using 64-bit counters created by a pair of 32-bit counters paired by a CHAIN event, the architecture does not define the latency between the first counter overflowing and the second counter incrementing the CHAIN event. There is no requirement for updates to occur synchronously, but software reading or enabling the counter pair using a *low*-ISB-*high* sequence, as shown in Example D7-5 on page D7-2727, must not observe the low counter incrementing and overflowing for the event and the high counter not incrementing for the resulting CHAIN event. This means that the ISB executed after reading the low counter must ensure the completion of the update of the high counter by the CHAIN event.

0x0021, BR_RETIRED, Instruction architecturally executed, branch

The counter counts all branches on the architecturally executed path that would incur cost if mispredicted.

Counts all branch instructions, memory-reading and data-processing instructions that explicitly write to the PC, at retirement.

Note

Conditional branches are always counted, whether the branch is taken or not taken.

It is IMPLEMENTATION DEFINED whether this includes each of:

- Unconditional direct branch instructions. Arm recommends these are included.
- Exception-generating instructions.
- Exception return instructions. Arm recommends these are included.
- Context synchronization instructions.

From Armv8.6, when PC_WRITE_RETIRED and BR_RETIRED are both implemented, the PE must treat the following types of instruction in the same way for both events:

- BRK and BKPT instructions.
- UNDEFINED instructions.
- The exception-generating instructions, SVC, HVC, and SMC.
- Context synchronization barrier instructions.

0x8000, SIMD_INST_RETIRED, SIMD Instruction architecturally executed

The counter counts the following architecturally executed SIMD instructions.:

- SVE instructions, but not non-SIMD SVE instructions.
- Advanced SIMD instructions, but not Advanced SIMD scalar instructions.

0x8001, ASE_INST_RETIRED, Advanced SIMD Instruction architecturally executed

The counter counts architecturally executed Advanced SIMD instructions. It is IMPLEMENTATION DEFINED whether this event counts Advanced SIMD scalar instructions.

0x8002, SVE_INST_RETIRED, Instruction architecturally executed, SVE

The counter counts architecturally executed SVE instructions. It is IMPLEMENTATION DEFINED whether this event counts non-SIMD SVE instructions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

0x8003, ASE SVE_INST_RETIRED, Advanced SIMD and SVE Instruction architecturally executed

The counter counts architecturally executed instructions that are counted by ASE_INST_RETIRED or SVE_INST_RETIRED.

0x8107, BR_SKIP_RETIRED, Instruction architecturally executed, branch not taken

The counter counts the each conditional Software change of the PC instruction, on the architecturally executed path, that is not taken.

Note

Many of these instructions can only be conditional in the AArch32 instruction sets.

The counter does not increment for exceptions not listed as a Software change of the PC.

If PC_WRITE_RETIRED and BR_SKIP_RETIRED events are both implemented, the PE must have a consistent definition of Software change of the PC instructions. This means the definition must treat the following instructions in the same way for both events:

- BRK and BKPT instructions.
- UNDEFINED instructions.
- The exception-generating instructions, SVC, HVC, and SMC.
- Context synchronization barrier instructions.

From Armv8.6, then the BR_RETIRED event is implemented, the PE must treat these instructions in the same way for the BR_RETIRED event.

0x8108, BR_IMMED_TAKEN_RETIRED, Instruction architecturally executed, immediate branch taken

The counter counts the instructions, on the architecturally executed path, counted by both BR_IMMED_RETIRED and PC_WRITE_RETIRED. These are all immediate branch instructions where the branch is taken.

0x8109, BR_IMMED_SKIP_RETIRED, Instruction architecturally executed, immediate branch not taken

The counter counts the instructions on the architecturally executed path, counted by both BR_IMMED_RETIRED, and BR_SKIP_RETIRED. These are all immediate branch instructions where the branch is not taken.

0x810A, BR_IND_TAKEN_RETIRED, Instruction architecturally executed, indirect branch taken

The counter counts the instructions, on the architecturally executed path, counted by both BR_IND_RETIRED, and PC_WRITE_RETIRED. These are branch instructions where the branch is taken, but does not include immediate instructions.

A64 does not include conditional indirect branches. If A32 is not supported at any Exception level, this event is not implemented because BR_IND_RETIRED counts the same events.

0x810B, BR_IND_SKIP_RETIRED, Instruction architecturally executed, indirect branch not taken

The counter counts the instructions on the architecturally executed path, counted by both BR_IND_RETIRED, and BR_SKIP_RETIRED. These are branch instructions, where the branch is not taken, but does not include immediate instructions.

A64 does not include conditional indirect branches. If A32 is not supported at any Exception level, this event is not implemented.

0x810C, BR_INDNR_TAKEN_RETIRED, Instruction architecturally executed, indirect branch taken excluding returns

The counter counts the instructions, on the architecturally executed path, counted by both BR_IND_RETIRED, and PC_WRITE_RETIRED, but not BR_RETURN_RETIRED. These are branch instructions, where the branch is taken, but does not include returns or immediate instructions.

0x810D, BR_INDNR_SKIP_RETIRED, Instruction architecturally executed, indirect branch not taken excluding returns

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The counter counts the instructions, on the architecturally executed path, counted by both BR_INDNR_RETIRED and BR_SKIP_RETIRED. These are branch instructions, where the branch is not taken, but does not include returns or immediate instructions.

A64 does not include conditional indirect branches. If A32 is not supported at any Exception level, this event is not implemented.

`0x810E`, **BR_RETURN_ANY_RETIRED, Instruction architecturally executed, procedure return**

The counter counts the instructions counted on the architecturally executed path by BR_IND_RETIRED where, if taken, the branch would be counted by BR_RETURN_RETIRED.

A64 does not include conditional indirect branches. If A32 is not supported at any Exception level, this event is not implemented because BR_RETURN_RETIRED counts the same events.

`0x810F`, **BR_RETURN_SKIP_RETIRED, Instruction architecturally executed, procedure return not taken**

The counter counts the instructions on the architecturally executed path, counted by both BR_RETURN_ANY_RETIRED and BR_SKIP_RETIRED. These are branch return instructions, where the branch is not taken.

A64 does not include conditional indirect branches. If A32 is not supported at any Exception level, this event is not implemented.

`0x811D`, **BR_IND_RETIRED, Instruction architecturally executed, indirect branch**

The counter counts each Software change of the PC on the architecturally executed path that is not counted by BR_IMMED_RETIRED. These are all branch instructions that are not immediate,

Note

Conditional branches are always counted, whether the branch is taken or not taken.

`0x811E`, **BR_INDNR_RETIRED, Instruction architecturally executed, indirect branch excluding procedure return**

The counter counts the instructions on the architecturally executed path counted by BR_RETIRED, but not counted by BR_RETURN_ANY_RETIRED. These are branch instructions but does not include returns or immediate instructions.

If A32 is not supported at any Exception level, this event is not implemented because BR_INDNR_TAKEN_RETIRED counts the same events.

Common microarchitectural events

This section describes the use of the defined common microarchitectural event numbers.

The common microarchitectural events are features that are likely to be implemented across a wide range of implementations. Unlike the common architectural events, there can be some IMPLEMENTATION DEFINED variation between definitions on different implementations.

Unless otherwise stated, the common microarchitectural features relate only to events resulting from the operation of the PE counting the events. Events resulting from the operation of other PEs that might share a resource must not be counted. Where a resource can be subject to events that do not result from the operation of any of the PEs that share it, Arm recommends that the resource implements its own event counters. An example of a resource that might require its own event counters is a shared Level 2 cache that is subject to accesses from a system coherency port on that cache.

The event definitions relating to Level 2 caches generally assume the Level 2 cache is shared. The event definitions relating to Level 1 caches generally assume the Level 1 cache is not shared.

The events corresponding to the common microarchitectural event numbers are:

`0x0001`, **L1I_CACHE_REFILL, Level 1 instruction cache refill**

The counter counts each access counted by L1I_CACHE that causes a demand refill of any of the Level 1 caches outside the Level 1 caches of this PE.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A refill includes any access that causes data to be fetched from outside the cache, even if the data is ultimately not allocated into the cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

If the cache is shared, only events Attributable to this PE are counted. If the cache is not shared, all events are counted. If ARMv8.4-PMU is not implemented, the counter does not count cache maintenance instructions. If ARMv8.4-PMU is implemented, it is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted by this event.

See also:

- *Attributability* on page D7-2707.
- *Meaningful ratios between common microarchitectural events* on page D7-2769.

`0x0002`, L1I_TLB_REFILL, Attributable Level 1 instruction TLB refill

The counter counts Attributable instruction memory accesses that cause a TLB refill of at least the Level 1 instruction TLB. This includes each Instruction memory access that causes an access to a level of memory system due to a translation table walk or an access to another level of TLB caching. It is IMPLEMENTATION DEFINED whether the count increments when:

- A refill results in a Translation fault.
- A refill is not allocated in the TLB.

The counter does not count:

- A TLB miss that does not cause a refill but does generate a translation table walk.
- TLB maintenance instructions.

See also:

- *Attributability* on page D7-2707.
- *Meaningful ratios between common microarchitectural events* on page D7-2769.

`0x0003`, L1D_CACHE_REFILL, Level 1 data cache refill

The counter counts each access counted by L1D_CACHE that causes a demand refill of at least the Level 1 data or unified cache from outside the Level 1 cache. Each access to a cache line that causes a new linefill is counted, including those from instructions that generate multiple accesses, such as load or store multiples, and `PUSH` and `POP` instructions. In particular, the counter counts accesses to the Level 1 cache that cause a refill that is satisfied by another Level 1 data or unified cache, or a Level 2 cache, or memory.

A refill includes any access that causes data to be fetched from outside the cache, even if the data is ultimately not allocated into the cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

The counter does not count:

- A miss that does not cause a new refill but is satisfied by the refill of a previous miss, even if that previous refill is not complete at the time of the miss.
- A miss that does not generate a refill, such as a write through the cache.
- If ARMv8.4-PMU is not implemented, cache maintenance instructions.

If the cache is shared, only events Attributable to this PE are counted. If the cache is not shared, all events are counted. If ARMv8.4-PMU is implemented, it is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted by this event.

See also:

- *Attributability* on page D7-2707.
- *Meaningful ratios between common microarchitectural events* on page D7-2769.

`0x0004`, L1D_CACHE, Level 1 data cache access

If ARMv8.4-PMU is not implemented:

- The counter counts each Memory-read operation or Memory-write operation that causes a cache access to at least the Level 1 data or unified cache.
- The counter does not count cache maintenance instructions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If ARMv8.4-PMU is implemented:

- The counter counts each Memory-read operation or Memory-write operation that cause a cache access, involving a data transfer to or from the cache, to at least the Level 1 data or unified cache. This includes accesses that transfer data to the cache when the data is not ultimately allocated to the cache. This does not include accesses that update the cache status information for the cache entry without changing the content of the cache data.
- It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If the cache is shared, only events Attributable to this PE are counted. If the cache is not shared, all events are counted.

Each access to a cache line is counted including the multiple accesses of instructions, such as `LDM` or `STM`. Each access to other Level 1 data or unified memory structures, for example refill buffers, write buffers, and write-back buffers, is also counted.

See also *Attributability* on page D7-2707.

`0x0005`, L1D_TLB_REFILL, Attributable Level 1 data TLB refill

The counter counts each Attributable Memory-read operation or Attributable Memory-write operation that causes a TLB refill of at least the Level 1 data or unified TLB. It counts each read or write that causes a refill, in the form of a translation table walk or an access to another level of TLB caching. It is IMPLEMENTATION DEFINED whether the count increments when:

- A refill results in a Translation fault.
- A refill is not allocated in the TLB.

The counter does not count:

- A TLB miss that does not cause a refill but does generate a translation table walk.
- TLB maintenance instructions.

See also:

- *Attributability* on page D7-2707.
- *Meaningful ratios between common microarchitectural events* on page D7-2769.

`0x0010`, BR_MIS_PRED, Mispredicted or not predicted branch Speculatively executed

The counter counts each correction to the predicted program flow that occurs because of a misprediction from, or no prediction from, the branch prediction resources and that relates to instructions that the branch prediction resources are capable of predicting.

If no program-flow prediction resources are implemented, Arm recommends that the counter counts all branches that are not taken.

`0x0011`, CPU_CYCLES, Cycle

The counter increments on every cycle.

All counters are subject to changes in clock frequency, including when a `WFI` or `WFE` instruction stops the clock. This means that it is CONSTRAINED UNPREDICTABLE whether or not CPU_CYCLES continues to increment when the clocks are stopped by `WFI` and `WFE` instructions.

Note

Unlike PMCCNTR, this count is not affected by PMCR.DP, PMCR.D, PMCR.C, SDCR.SCCD or HDCR.HCCD:

- The counter is not incremented in prohibited regions, so is not affected by PMCR.DP.
- The counter increments on every cycle, regardless of the setting of PMCR.D.
- The counter is reset when event counters are reset by PMCR.P, never by PMCR.C.
- The counter is not disabled when ARMv8.5-PMU is implemented, EL3 is implemented, the PE is in Secure state, and SDCR.SCCD is set to 1.
- The counter is not disabled when ARMv8.5-PMU is implemented, EL2 is implemented, the PE is executing at EL2, and HDCR.HCCD is set to 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In a multithreaded implementation, CPU_CYCLES counts each cycle for the processor for which this PE thread was active and could issue an instruction. For more information, see *Cycle event counting on multithreaded implementations* on page D7-2768.

`0x0012`**, BR_PRED, Predictable branch Speculatively executed**

The counter counts every branch or other change in the program flow that the branch prediction resources are capable of predicting.

If all branches are subject to prediction, for example a BTB or BTAC, then all branches are predictable branches.

If branches are decoded before the predictor, so that the branch prediction logic dynamically predicts only some branches, for example conditional and indirect branches, then it is IMPLEMENTATION DEFINED whether other branches are counted as predictable branches. Arm recommends that all branches are counted.

An implementation might include other structures that predict branches, such as a loop buffer that predicts short backwards direct branches as taken. Each execution of such a branch is a predictable branch. Terminating the loop might generate a misprediction event that is counted by BR_MIS_PRED.

If no program-flow prediction resources are implemented, this event is optional, but Arm recommends that BR_PRED counts all branches.

`0x0013`**, MEM_ACCESS, Data memory access**

The counter counts Memory-read operations and Memory-write operations that the PE made. The counter increments whether the access results in an access to a Level 1 data or unified cache, a Level 2 data or unified cache, or neither of these.

The counter does not increment as a result of:

- Instruction memory accesses, see *Definition of terms* on page D7-2716.
- Translation table walks.
- Write-back from any cache.
- Refilling of any cache.

The number of accesses generated by each instruction is IMPLEMENTATION DEFINED.

If ARMv8.4-PMU is not implemented, the counter does not count cache maintenance instructions. If ARMv8.4-PMU is implemented, it is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

`0x0014`**, L1I_CACHE, Attributable Level 1 instruction cache access**

If ARMv8.4-PMU is not implemented, the counter counts Attributable instruction memory accesses that access at least the Level 1 instruction or unified cache.

If ARMv8.4-PMU is implemented, the counter counts Attributable instruction memory accesses, involving a data transfer to or from the cache, that access at least the Level 1 instruction or unified cache. This includes accesses that transfer data to the cache when the data is not ultimately allocated to the cache. This does not include accesses that update the cache status information for the cache entry without changing the content of the cache data.

Each access to other Level 1 instruction memory structures, such as refill buffers, is also counted.

See also *Attributability* on page D7-2707.

`0x0015`**, L1D_CACHE_WB, Attributable Level 1 data cache write-back**

The counter counts every write-back of data from the Level 1 data or unified cache. The counter counts each write-back that causes data to be written from the Level 1 cache to outside of the Level 1 cache. For example, the counter counts the following cases:

- A write-back that causes data to be written to a Level 2 cache or memory.
- A write-back of a recently fetched cache line that has not been allocated to the Level 1 cache.
- Transfer of data from the Level 1 cache to outside of this cache made as a result of a coherency request. The conditions determining which of these are counted for transfers to other Level 1 caches within the same multiprocessor cluster are IMPLEMENTATION DEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Each write-back is counted once, even if multiple accesses are required to complete the write-back.

Whether write-backs made as a result of cache maintenance instructions are counted is IMPLEMENTATION DEFINED.

The counter does not count:

- The invalidation of a cache line without any write-back to a Level 2 cache or memory.
- Writes from the PE that write through the Level 1 cache to outside of the Level 1 cache.

An Unattributable write-back event occurs when a requestor outside the PE makes a coherency request that results in write-back. If the cache is shared, then an Unattributable write-back event is not counted. If the cache is not shared, then the event is counted. See *Attributability* on page D7-2707.

It is IMPLEMENTATION DEFINED whether a write of a whole cache line that is not the result of the eviction of a line from the cache, is counted. For example, this applies when the PE determines streaming writes to memory and does not allocate lines to the cache, or by a `DC ZVA` operation.

See also *Attributability* on page D7-2707.

`0x0016`, L2D_CACHE, Level 2 data cache access

If ARMv8.4-PMU is not implemented:

- The counter counts each Memory-read operation or Memory-write operation that causes a cache access to at least the Level 2 data or unified cache.
- The counter does not count cache maintenance instructions.

If ARMv8.4-PMU is implemented:

- The counter counts each Memory-read operation or Memory-write operation that cause a cache access, involving a data transfer to or from the cache, to at least the Level 2 data or unified cache. This includes accesses that transfer data to the cache when the data is not ultimately allocated to the cache. This does not include accesses that update the cache status information for the cache entry without changing the content of the cache data.
- It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If the cache is shared, only events Attributable to this PE are counted. If the cache is not shared, all events are counted.

Each access to a cache line is counted including refills of and write-backs from the Level 1 data, instruction, or unified caches. Each access to other Level 2 data or unified memory structures, such as refill buffers, write buffers, and write-back buffers, is also counted.

See also *Attributability* on page D7-2707.

`0x0017`, L2D_CACHE_REFILL, Level 2 data cache refill

The counter counts each access counted by L2D_CACHE that causes a refill of a demand refill of any of the Level 1 or Level 2 caches from outside the Level 1 and Level 2 caches of the PE.

A refill includes any access that causes data to be fetched from outside the cache, even if the data is ultimately not allocated into the cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

For example, the counter counts:

- Accesses to the Level 2 cache that cause a refill that is satisfied by another Level 2 cache, a Level 3 cache, or memory.
- Refills of and write-backs from any Level 1 data, instruction or unified cache that cause a refill from outside the Level 1 and Level 2 caches.
- Accesses to the Level 2 cache that cause a refill of a Level 1 cache from outside of the Level 1 and Level 2 caches, even if there is no refill of the Level 2 cache.

The counter does not count, as events on this PE:

- A miss that does not cause a new refill but is satisfied by the refill of a previous miss, even if that previous refill is not complete at the time of the miss.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- A miss that does not generate a refill, such as a write through the cache.
- If ARMv8.4-PMU is not implemented, cache maintenance instructions.

If the cache is shared, only events Attributable to this PE are counted. If the cache is not shared, all events are counted. If ARMv8.4-PMU is implemented, it is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted by this event.

See also:

- *Attributability* on page D7-2707.
- *Meaningful ratios between common microarchitectural events* on page D7-2769.

`0x0018`**, L2D_CACHE_WB, Attributable Level 2 data cache write-back**

The counter counts every write-back of data from the Level 2 data or unified cache that occurs as a result of an operation by this PE. It counts each write-back that causes data to be written from the Level 2 cache to outside the Level 1 and Level 2 caches. For example, the counter counts:

- A write-back that causes data to be written to a Level 3 cache or memory.
- A write-back of a recently fetched cache line that has not been allocated to the Level 2 cache.

Each write-back is counted once, even if it requires multiple accesses to complete the write-back.

It is IMPLEMENTATION DEFINED whether the counter counts:

- A transfer of data from the Level 2 cache to outside the Level 1 and Level 2 cache made as a result of a coherency request.
- Write-backs made as a result of Cache maintenance instructions.

The counter does not count:

- The invalidation of a cache line without any write-back to a Level 3 cache or memory.
- Writes from the PE or Level 1 data or unified cache that write through the Level 2 cache to outside the Level 1 and Level 2 caches.
- Transfers of data from the Level 2 cache to a Level 1 cache, to satisfy a Level 1 cache refill.

An Unattributable write-back event occurs when a requestor outside the PE makes a coherency request that results in write-back. If the cache is shared, then an Unattributable write-back event is not counted. If the cache is not shared, then the event is counted.

It is IMPLEMENTATION DEFINED whether a write of a whole cache line that is not the result of the eviction of a line from the cache, is counted. For example, this applies when the PE determines streaming writes to memory and does not allocate lines to the cache, or by a `DC ZVA` operation.

See also *Attributability* on page D7-2707.

`0x0019`**, BUS_ACCESS, Attributable Bus access**

The counter counts Memory-read operations and Memory-write operations that access outside of the boundary of the PE and its closely-coupled caches. Where this boundary lies with respect to any implemented caches is IMPLEMENTATION DEFINED.

The definition of a bus access is IMPLEMENTATION DEFINED but physically is a single beat rather than a burst. That is, for each bus cycle for which the bus is active.

Bus accesses include refills of and write-backs from data, instruction, and unified caches. Whether bus accesses include operations that do use the bus but not explicitly transfer data is IMPLEMENTATION DEFINED.

An Unattributable bus access occurs when a requestor outside the PE makes a request that results in a bus access, for example, a coherency request. If the bus is shared, then an Unattributable bus access is not counted. If the bus is not shared, then the event is counted.

If the bus is shared, then only Attributable bus accesses are counted. If the bus is not shared, then all bus accesses are counted.

Where an implementation has multiple buses at this boundary, this event counts the sum of accesses across all buses.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If a bus supports multiple accesses per cycle, for example through multiple channels, the counter increments once for each channel that is active on a cycle, and so it might increment by more than one in any given cycle.

The maximum increment in any given cycle is implementation defined.

See also:

- *Attributability* on page D7-2707.
- *Meaningful ratios between common microarchitectural events* on page D7-2769.

`0x001A`, **MEMORY_ERROR, Local memory error**

The counter counts every occurrence of a memory error signaled by a memory closely coupled to this PE. The definition of local memories is IMPLEMENTATION DEFINED but includes caches, tightly-coupled memories, and TLB arrays.

Memory error refers to a physical error detected by the hardware, such as a parity or ECC error. It includes errors that are correctable and those that are not. It does not include errors as defined in the architecture, such as MMU faults.

`0x001B`, **INST_SPEC, Operation Speculatively executed**

The counter counts Speculatively executed operations. The definition of Speculatively executed is IMPLEMENTATION DEFINED.

`0x001D`, **BUS_CYCLES, Bus cycle**

The counter increments on every cycle of the interface at the boundary of the PE and its closely-coupled caches. Where this boundary lies with respect to any implemented caches is IMPLEMENTATION DEFINED.

Note

If the implementation clocks the external memory interface at the same rate as the processor hardware, the counter counts every cycle.

See also *Meaningful ratios between common microarchitectural events* on page D7-2769.

`0x001F`, **L1D_CACHE_ALLOCATE, Attributable Level 1 data cache allocation without refill**

The counter increments on every Attributable write that writes an entire line into the Level 1 cache without fetching from outside the Level 1 cache, for example:

- A write from a coalescing buffer of a full cache line.
- A `DC ZVA` operation.

See also *Attributability* on page D7-2707.

`0x0020`, **L2D_CACHE_ALLOCATE, Attributable Level 2 data cache allocation without refill**

The counter increments on every Attributable write that writes an entire line into the Level 2 cache without fetching from outside the Level 1 or Level 2 caches, for example:

- A write-back from a Level 1 to Level 2 cache.
- A write from a coalescing buffer of a full cache line.
- A `DC ZVA` operation.

See also *Attributability* on page D7-2707.

`0x0022`, **BR_MIS_PRED_RETIRED, Instruction architecturally executed, mispredicted branch**

The counter counts all instructions counted by BR_RETIRED that were not correctly predicted.

If no program-flow prediction resources are implemented, this event counts all retired not-taken branches.

`0x0023`, **STALL_FRONTEND, No operation issued due to the frontend**

The counter counts every cycle counted by the CPU_CYCLES event on which no operation was issued because there are no operations available to issue for this PE from the frontend.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The division between frontend and backend is IMPLEMENTATION DEFINED. STALL, STALL_FRONTEND, and STALL_BACKEND events must count at the same point in the pipeline.

Note

- For a simplified pipeline model of Fetch → Decode → Issue → Execute → Retire, Arm recommends that the events are counted when instructions are dispatched from Decode to Issue.
- On a given cycle, both events might be counted if the backend is unable to accept any operations and there are no operations available to issue from the frontend.

For more information, see *Cycle event counting on multithreaded implementations* on page D7-2768.

`0x0024`, **STALL_BACKEND, No operation issued due to the backend**

The counter counts every cycle counted by the CPU_CYCLES event on which no operation was issued because either:

- The backend is unable to accept any of the operations available for issue for this PE.
- The backend is unable to accept any operations.

For example, the back end might be unable to accept operations because of a resource conflict or non-availability.

The division between frontend and backend is IMPLEMENTATION DEFINED. STALL, STALL_FRONTEND, and STALL_BACKEND events must count at the same point in the pipeline. See STALL_FRONTEND for more information.

For more information, see *Cycle event counting on multithreaded implementations* on page D7-2768.

`0x0025`, **L1D_TLB, Attributable Level 1 data or unified TLB access**

The counter counts each Attributable Memory-read operation or Attributable Memory-write operation that causes a TLB access to at least the Level 1 data or unified TLB. Each access to a TLB record is counted including the multiple accesses of instructions, such as `LDM` or STM.

The counter does not count TLB maintenance instructions.

See also *Attributability* on page D7-2707.

`0x0026`, **L1I_TLB, Attributable Level 1 instruction TLB access**

The counter counts each Attributable Instruction memory access that causes a TLB access to at least the Level 1 instruction or unified TLB.

The counter does not count TLB maintenance instructions.

See also *Attributability* on page D7-2707.

`0x0027`, **L2I_CACHE, Attributable Level 2 instruction cache access**

If ARMv8.4-PMU is not implemented, the counter counts Attributable instruction memory accesses that access at least the Level 2 instruction or unified cache.

If ARMv8.4-PMU is implemented, the counter counts Attributable instruction memory accesses, involving a data transfer to or from the cache, that access at least the Level 2 instruction or unified cache. This includes accesses that transfer data to the cache when the data is not ultimately allocated to the cache. This does not include accesses that update the cache status information for the cache entry without changing the content of the cache data.

Each Attributable access to other Level 2 instruction memory structures, such as refill buffers, is also counted.

See also *Attributability* on page D7-2707.

`0x0028`, **L2I_CACHE_REFILL, Attributable Level 2 instruction cache refill**

The counter counts each access counted by L2I_CACHE that causes a demand refill of any of the Level 1 or 2 caches outside the Level 1 or 2 caches of this PE.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A refill includes any access that causes data to be fetched from outside the cache, even if the data is ultimately not allocated into the cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

If the cache is shared, only events Attributable to this PE are counted. If the cache is not shared, all events are counted. If ARMv8.4-PMU is not implemented, the counter does not count cache maintenance instructions. If ARMv8.4-PMU is implemented, it is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted by this event.

See also:

- *Attributability* on page D7-2707.
- *Meaningful ratios between common microarchitectural events* on page D7-2769.

`0x0029`, **L3D_CACHE_ALLOCATE, Attributable Level 3 data cache allocation without refill**

The counter increments on every Attributable write that writes an entire line into the Level 3 cache without fetching from outside the Level 1, Level 2, or Level 3 cache, for example:

- A write-back from a Level 2 to Level 3 cache.
- A write from a coalescing buffer of a full cache line.
- `A DC ZVA` operation.

See also *Attributability* on page D7-2707.

`0x002A`, **L3D_CACHE_REFILL, Attributable Level 3 data cache refill**

The counter counts each access counted by L3D_CACHE which causes a demand refill of any of the Level 1, Level 2, or Level 3 caches from outside the Level 1, Level 2, and Level 3 caches.

A refill includes any access that causes data to be fetched from outside the cache, even if the data is ultimately not allocated into the cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

The counter does not count as events on this PE:

- A miss that does not cause a new refill but is satisfied by the refill of a previous miss, even if that previous refill is not complete at the time of the miss.
- A miss that does not generate a refill, such as a write through the cache.
- If ARMv8.4-PMU is not implemented, cache maintenance instructions.

If the cache is shared, only events Attributable to this PE are counted. If the cache is not shared, all events are counted. If ARMv8.4-PMU is implemented, it is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted by this event.

See also:

- *Attributability* on page D7-2707.
- *Meaningful ratios between common microarchitectural events* on page D7-2769.

`0x002B`, **L3D_CACHE, Attributable Level 3 data cache access**

If ARMv8.4-PMU is not implemented:

- The counter counts each Memory-read operation or Memory-write operation that causes a cache access to at least the Level 3 data or unified cache.
- The counter does not count cache maintenance instructions.

If ARMv8.4-PMU is implemented:

- The counter counts each Memory-read operation or Memory-write operation that cause a cache access, involving a data transfer to or from the cache, to at least the Level 3 data or unified cache. This includes accesses that transfer data to the cache when the data is not ultimately allocated to the cache. This does not include accesses that update the cache status information for the cache entry without changing the content of the cache data.
- It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If the cache is shared, only events Attributable to this PE are counted. If the cache is not shared, all events are counted.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Each access to a cache line is counted including refills of and write-backs from the Level 1 or Level 2 data, instruction, or unified caches. Each access to other Level 3 data or unified memory structures, such as refill buffers, write buffers, and write-back buffers, is also counted.

See also *Attributability* on page D7-2707.

`0x002C`, L3D_CACHE_WB, Attributable Level 3 data cache write-back

The counter counts every write-back of data from the Level 3 data or unified cache that occurs as a result of an operation by this PE. It counts each write-back that causes data to be written from the Level 3 cache to outside of the Level 1, Level 2, and Level 3 caches. For example, the counter counts the following cases:

- A write-back that causes data to be written to a Level 4 cache, or to memory.
- A write-back of a recently fetched cache line that has not been allocated to the Level 3 cache.

Each write-back is counted once, even if multiple accesses are required to complete the write-back.

It is IMPLEMENTATION DEFINED whether the counter counts:

- A transfer of data from the Level 3 cache to outside the Level 1, Level 2, and Level 3 caches made as a result of a coherency request.
- A write-back made as a result of a Cache maintenance instruction.

The counter does not count:

- The invalidation of a cache line without any write-back to a Level 4 cache or memory.
- Writes from the PE, Level 1, or Level 2 data or unified cache, that write through the Level 3 cache to outside of the Level 3 cache.
- Transfers of data from the Level 3 cache to a Level 1 or Level 2 cache, to satisfy a Level 1 or Level 2 cache refill.

An Unattributable write-back event occurs when a requestor outside the PE makes a coherency request that results in write-back. If the cache is shared, then Unattributable write-back events are not counted. If the cache is not shared, then Unattributable write-back events are counted.

It is IMPLEMENTATION DEFINED whether a write of a whole cache line that is not the result of the eviction of a line from the cache, is counted. For example, this applies when the PE determines streaming writes to memory and does not allocate lines to the cache, or by a `DC ZVA` operation.

See also *Attributability* on page D7-2707.

`0x002D`, L2D_TLB_REFILL, Attributable Level 2 data TLB refill

The counter counts each Attributable Memory-read operation or Attributable Memory-write operation that causes a TLB refill of at least the Level 2 data or unified TLB. It counts each Attributable read or Attributable write that causes a refill, in the form of a translation table walk or an access to another level of TLB caching. It is IMPLEMENTATION DEFINED whether the count increments when:

- A refill results in a Translation fault.
- A refill is not allocated in the TLB.

The counter does not count:

- A TLB miss that does not cause a refill but does generate a translation table walk.
- TLB maintenance instructions.

See also:

- *Attributability* on page D7-2707.
- *Meaningful ratios between common microarchitectural events* on page D7-2769.

`0x002E`, L2I_TLB_REFILL, Attributable Level 2 instruction TLB refill

The counter counts Attributable instruction memory accesses that cause a TLB refill of at least the Level 2 instruction TLB. This includes each Attributable Instruction memory access that causes an access to a level of memory system due to a translation table walk or an access to another level of TLB caching. It is IMPLEMENTATION DEFINED whether the count increments when:

- A refill results in a Translation fault.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- A refill is not allocated in the TLB.

The counter does not count:

- A TLB miss that does not cause a refill but does generate a translation table walk.
- TLB maintenance instructions.

See also:

- *Attributability* on page D7-2707.
- *Meaningful ratios between common microarchitectural events* on page D7-2769.

`0x002F`, L2D_TLB, Attributable Level 2 data or unified TLB access

The counter counts each Attributable memory read operation or Attributable memory write operation that causes a TLB access to at least the Level 2 data or unified TLB. Each access to a TLB record is counted, including the multiple accesses of instructions such as `LDM` or `STM`.

The counter does not count TLB maintenance instructions.

See also *Attributability* on page D7-2707.

`0x0030`, L2I_TLB, Attributable Level 2 instruction TLB access

The counter counts each Attributable memory read operation or Attributable memory write operation that causes a TLB access to at least the Level 2 instruction or unified TLB.

The counter does not count TLB maintenance instructions.

See also *Attributability* on page D7-2707.

`0x0031`, REMOTE_ACCESS, Access to another socket in a multi-socket system

The counter counts each Attributable memory read operation or memory write operation that causes an access to another socket in a multi-socket system.

It is IMPLEMENTATION DEFINED whether an access that causes a snoop into another socket but does not return data from or pass data to the remote socket is counted.

See also *Attributability* on page D7-2707.

`0x0032`, LL_CACHE, Last Level cache access

The counter counts each Memory-read operation or Memory-write operation that causes a cache access to at least the Last Level data or unified cache. If the cache is shared, only events Attributable to this PE are counted. If the cache is not shared, all events are counted.

If ARMv8.4-PMU is not implemented, the counter does not count cache maintenance instructions. If ARMv8.4-PMU is implemented, it is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

See also *Attributability* on page D7-2707.

`0x0033`, LL_CACHE_MISS, Last Level cache miss

The Counter counts each Attributable Memory-read operation or Memory-write operation that causes a cache access to at least the Last Level data or unified cache, but is not completed by the Last Level cache. That is, either of the following:

- A memory read operation that does not return data from the Last Level cache.
- A memory write operation that does not update the Last Level cache.

The counter does not count operations that are completed by a cache above the Last Level cache.

If ARMv8.4-PMU is not implemented, the counter does not count cache maintenance instructions. If ARMv8.4-PMU is implemented, it is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

See also:

- *Attributability* on page D7-2707.
- *Meaningful ratios between common microarchitectural events* on page D7-2769.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

0x0034, DTLB_WALK, Access to data TLB causes a translation table walk

The counter counts each Attributable memory read or memory write operation that causes a refill of a data or unified TLB involving at least one translation table walk access. This includes each complete or partial translation table walk that causes an access to memory, including to data or translation table walk caches.

The counter does not count TLB maintenance instructions.

See also:

- *Attributability* on page D7-2707.
- *Meaningful ratios between common microarchitectural events* on page D7-2769.

0x0035, ITLB_WALK, Access to instruction TLB that causes a translation table walk

The counter counts each Attributable Instruction memory access that causes a refill of an instruction TLB, involving at least one translation table walk access. This includes each complete or partial translation table walk that causes an access to memory, including to data or translation table walk caches.

The counter does not count TLB maintenance instructions.

See also:

- *Attributability* on page D7-2707.
- *Meaningful ratios between common microarchitectural events* on page D7-2769.

0x0036, LL_CACHE_RD, Attributable Last level cache memory read

As LL_CACHE, but counts only memory read accesses.

If the cache is shared, only events Attributable to this PE are counted. If the cache is not shared, all events are counted. If ARMv8.4-PMU is not implemented, the counter does not count cache maintenance instructions. If ARMv8.4-PMU is implemented, it is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted by this event.

See also:

- *Attributability* on page D7-2707.
- *Meaningful ratios between common microarchitectural events* on page D7-2769.

0x0037, LL_CACHE_MISS_RD, Last level cache miss, read

As LL_CACHE_MISS, but counts only memory read operations.

If ARMv8.4-PMU is not implemented, the counter does not count cache maintenance instructions. If ARMv8.4-PMU is implemented, it is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

See also:

- *Attributability* on page D7-2707.
- *Meaningful ratios between common microarchitectural events* on page D7-2769.

0x0038, REMOTE_ACCESS_RD, Access to another socket in a multi-socket system, read

As REMOTE_ACCESS, but counts only memory read operations.

See also:

- *Attributability* on page D7-2707.
- *Meaningful ratios between common microarchitectural events* on page D7-2769.

0x0039, L1D_CACHE_LMISS_RD, Level 1 data cache long-latency read miss

The counter counts each memory read access counted by L1D_CACHE that incurs additional latency because it returns data from outside the Level 1 data or unified cache of this PE.

If the cache is shared, only events Attributable to this PE are counted. If the cache is not shared, all events are counted. It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The event indicates to software that the access missed in the Level 1 data or unified cache and might have a significant performance impact compared to the latency of an access that hits in the Level 1 data or unified cache.

This counter does not count:

- Access where the additional latency is unlikely to be significantly performance-impacting. For example, if the access hits in another cache in the same local cluster, and the additional latency is small when compared against a miss in all Level 1 caches that the access looks up in that results in an access being made to a Level 2 cache or elsewhere beyond the Level 1 data and unified cache.
- A miss that does not cause a new cache refill but is satisfied from a previous miss.

An implementation is not required to measure the latency nor to track the access to determine whether the additional latency had a performance impact. An implementation can extend the definition of this event with additional scenarios where a memory read access counted by L1D_CACHE might have a significant performance impact due to additional latency for the address.

See also *Attributability* on page D7-2707.

`0x003A`, **OP_RETIRED, Micro-operation architecturally executed**

The counter counts each operation counted by OP_SPEC that would be executed in a *Simple sequential execution* of the program.

`0x003B`, **OP_SPEC, Micro-operation Speculatively executed**

The counter counts the number of operations executed by the PE, including those that are executed speculatively and would not be executed in a *Simple sequential execution* of the program.

`0x003C`, **STALL, No operation sent for execution**

The counter counts every Attributable cycle on which no Attributable instruction or operation was sent for execution on this PE.

If the PMU supports multi-threading:

- When PMEVTYPER<n>_EL0.MT = `0b0`, the counter counts cycles for which only instructions or operations Attributable to other PEs are sent for execution when this PE is eligible to execute instructions or operations on that cycle. The counter does not count cycles when this PE of the multi-threaded operation is not eligible to execute instructions or operations.
- When PMEVTYPER<n>_EL0.MT = `0b1`, the counter counts all cycles when no instructions or operations for any PE of the multi-threaded operation are sent for execution.

The division between frontend and backend is IMPLEMENTATION DEFINED. STALL, STALL_FRONTEND, and STALL_BACKEND events must count at the same point in the pipeline. For more information, see STALL_FRONTEND.

See also:

- *Attributability* on page D7-2707.
- *Meaningful ratios between common microarchitectural events* on page D7-2769.

`0x003D`, **STALL_SLOT_BACKEND, No operation sent for execution on a Slot due to the backend**

Counts each Slot counted by STALL_SLOT where no Attributable instruction or operation was sent for execution because the backend is unable to accept one of:

- The instruction operation available for the PE on the Slot.
- Any operations on the Slot.

The division between frontend and backend is IMPLEMENTATION DEFINED. STALL_SLOT, STALL_SLOT_FRONTEND, and STALL_SLOT_BACKEND events must count at the same point in the pipeline. The maximum value that STALL_SLOT_FRONTEND and STALL_SLOT_BACKEND events can count in a single-cycle is IMPLEMENTATION DEFINED. For more information, see STALL_SLOT.

See also *Attributability* on page D7-2707.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0x003E`, STALL_SLOT_FRONTEND, No operation sent for execution on a Slot due to the frontend

Counts each Slot counted by STALL_SLOT where no Attributable instruction or operation was sent for execution because there was no Attributable instruction or operation available to issue from the PE from the frontend for the Slot.

The division between frontend and backend is IMPLEMENTATION DEFINED. STALL_SLOT, STALL_SLOT_FRONTEND, and STALL_SLOT_BACKEND events must count at the same point in the pipeline. The maximum value that STALL_SLOT_FRONTEND and STALL_SLOT_BACKEND events can count in a single-cycle is implementation defined. For more information, see STALL_SLOT.

Note

Arm recommends that STALL_SLOT_FRONTEND counts instructions that have been decoded and, if applicable, split into micro-operations.

See also *Attributability* on page D7-2707.

`0x003F`, STALL_SLOT, No operation sent for execution on a Slot

The counter counts on each Attributable cycle the number of instruction or operation Slots that were not occupied by an instruction or operation Attributable to the PE.

If the PMU supports multi-threading:

- When PMEVTYPER<n>_EL0.MT = `0b0`, the counter counts instruction or operation Slots for which those Slots are occupied by instructions or operations Attributable to other PEs of the multi-threaded implementation only when the PE was eligible to execute instruction or operations in that cycle. The counter does not count any instruction or operation Slots on cycles when this PE was not eligible to execute instructions or operations.
- When PMEVTYPER<n>_EL0.MT = `0b1`, for every cycle the counter counts all instruction or operation Slots not occupied by any instruction or operation for any PE of the multi-threaded implementation.

If ARMv8.4-PMU is implemented:

- If STALL_SLOT is not implemented, it is IMPLEMENTATION DEFINED whether the PMMIR System registers are implemented.
- If STALL_SLOT is implemented, then the PMMIR System registers are implemented.

See also *Attributability* on page D7-2707.

`0x4000`, SAMPLE_POP, Sample Population

The counter increments for each operation that might be sampled, whether or not the operation was sampled. Operations that are executed at an Exception level or Security state in which the Statistical Profiling Extension is disabled are not counted.

`0x4001`, SAMPLE_FEED, Sample Taken

The counter increments each time the sample interval counter reaches zero and is reloaded, and the sample does not collide with the previous sample. Samples that are removed by filtering, or discarded, and not written to the Profiling Buffer are counted.

`0x4002`, SAMPLE_FILTRATE, Sample taken and not removed by filtering

The counter increments each time that a completed sample record is checked against the filters and not removed. Sample records that are not removed by filtering, but are discarded before being written to the Profiling Buffer because of a Profiling Buffer management event, are counted.

`0x4003`, SAMPLE_COLLISION, Sample collided with a previous sample

The counter increments for each sample record that is taken when the previous sampled operation has not completed generating its sample record.

`0x4004`, CNT_CYCLES, Constant frequency cycles

The counter is defined identically to CNT_CYCLES in the AMUv1 architecture.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0x4005`, **STALL_BACKEND_MEM, Memory stall cycles**

The counter is defined identically to STALL_BACKEND_MEM in the AMUv1 architecture.

`0x4006`, **L1I_CACHE_LMISS, Level 1 instruction cache long-latency read miss**

The counter counts each access counted by L1I_CACHE that incurs additional latency because it returns instructions from outside the Level 1 instruction cache.

If the cache is shared, only events Attributable to this PE are counted. If the cache is not shared, all events are counted. It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

The event indicates to software that the access missed in the Level 1 instruction cache and might have a significant performance impact due to the additional latency, compared to the latency of an access that hits in the Level 1 instruction cache.

This counter does not count:

- Access where the additional latency is unlikely to be significantly performance-impacting. For example, if the access hits in another cache in the same local cluster, and the additional latency is small when compared against a miss in all Level 1 caches that the access looks up in that results in an access being made to a Level 2 cache or elsewhere beyond the Level 1 data and unified cache.
- A miss that does not cause a new cache refill but is satisfied from a previous miss.

An implementation is not required to measure the latency nor to track the access to determine whether the additional latency had a performance impact. An implementation can extend the definition of this event with additional scenarios where a memory read access counted by L1I_CACHE might have a significant performance impact due to additional latency for the address.

`0x4009`, **L2D_CACHE_LMISS_RD, Level 2 data cache long-latency read miss**

The counter counts each memory read access counted by L2D_CACHE that incurs additional latency because it returns data from outside the Level 2 data or unified cache of this PE.

If the cache is shared, only events Attributable to this PE are counted. If the cache is not shared, all events are counted. It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

The event indicates to software that the access missed in the Level 2 data or unified cache and might have a significant performance impact compared to the latency of an access that hits in the Level 2 data or unified cache.

This counter does not count:

- Access where the additional latency is unlikely to be significantly performance-impacting. For example, if the access hits in another cache in the same local cluster, and the additional latency is small when compared against a miss in all Level 2 caches that the access looks up in that results in an access being made to a Level 3 cache or elsewhere beyond the Level 2 data and unified cache. This might be counted as a Level 1 cache miss.
- A miss that does not cause a new cache refill but is satisfied from a previous miss.

An implementation is not required to measure the latency nor to track the access to determine whether the additional latency had a performance impact. An implementation can extend the definition of this event with additional scenarios where a memory read access counted by L2D_CACHE might have a significant performance impact due to additional latency for the address.

See also *Attributability* on page D7-2707.

`0x400A`, **L2I_CACHE_LMISS, Level 2 instruction cache long-latency read miss**

The counter counts each access counted by L2I_CACHE that incurs additional latency because it returns instructions from outside the Level 2 instruction cache.

If the cache is shared, only events Attributable to this PE are counted. If the cache is not shared, all events are counted. It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The event indicates to software that the access missed in the Level 2 instruction cache and might have a significant performance impact due to the additional latency, compared to the latency of an access that hits in the Level 2 instruction cache.

This counter does not count:

- Access where the additional latency is unlikely to be significantly performance-impacting. For example, if the access hits in another cache in the same local cluster, and the additional latency is small when compared against a miss in all Level 2 caches that the access looks up in that results in an access being made to a Level 3 cache or elsewhere beyond the Level 2 data and unified cache. This might be counted as a Level 1 cache miss.
- A miss that does not cause a new cache refill but is satisfied from a previous miss.

An implementation is not required to measure the latency nor to track the access to determine whether the additional latency had a performance impact. An implementation can extend the definition of this event with additional scenarios where a memory read access counted by L2I_CACHE might have a significant performance impact due to additional latency for the address.

`0x400B`, **L3D_CACHE_LMISS_RD, Level 3 data cache long-latency read miss**

The counter counts each memory read access counted by L3D_CACHE that incurs additional latency because it returns data from outside the Level 3 data or unified cache of this PE.

If the cache is shared, only events Attributable to this PE are counted. If the cache is not shared, all events are counted. It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

The event indicates to software that the access missed in the Level 3 data or unified cache and might have a significant performance impact compared to the latency of an access that hits in the Level 3 data or unified cache.

This counter does not count:

- Access where the additional latency is unlikely to be significantly performance-impacting. For example, if the access hits in another cache in the same local cluster, and the additional latency is small when compared against a miss in all Level 3 caches that the access looks up in that results in an access being made to a Level 4 cache or elsewhere beyond the Level 3 data and unified cache. This might be counted as a Level 2 cache miss.
- A miss that does not cause a new cache refill but is satisfied from a previous miss.

An implementation is not required to measure the latency nor to track the access to determine whether the additional latency had a performance impact. An implementation can extend the definition of this event with additional scenarios where a memory read access counted by L3D_CACHE might have a significant performance impact due to additional latency for the address.

See also *Attributability* on page D7-2707.

`0x4020`, **LDST_ALIGN_LAT, Access with additional latency from alignment**

The counter counts each access counted by MEM_ACCESS that, due to the alignment of the address and size of data being accessed, incurred additional latency.

`0x4021`, **LD_ALIGN_LAT, Load with additional latency from alignment**

The counter counts each memory-read access counted by LDST_ALIGN_LAT.

`0x4022`, **ST_ALIGN_LAT, Store with additional latency from alignment**

The counter counts each memory-write access counted by LDST_ALIGN_LAT.

`0x4024`, **MEM_ACCESS_CHECKED, Checked data memory access**

The counter counts each memory access counted by MEM_ACCESS that is Tag Checked by the Memory Tagging Extension. For more information see Chapter D6 *Armv8.5 Memory Tagging Extension*.

`0x4025`, **MEM_ACCESS_CHECKED_RD, Checked data memory access, read**

The counter counts each memory-read access counted by MEM_ACCESS_CHECKED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0x4026`, **MEM_ACCESS_CHECKED_WR, Checked data memory access, write**

The counter counts each memory-write access counted by MEM_ACCESS_CHECKED.

`0x8004`, **SIMD_INST_SPEC, SIMD Instructions, Operations speculatively executed**

The counter counts speculatively executed operations due to the following SIMD instructions:

- SVE instructions, but not non-SIMD SVE instructions.
- Advanced SIMD instructions, but not Advanced SIMD scalar instructions.

`0x8005`, **ASE_INST_SPEC, Advanced SIMD Instructions, Operations speculatively executed**

The counter counts speculatively executed operations due to Advanced SIMD instructions. It is IMPLEMENTATION DEFINED whether this event counts operations due to Advanced SIMD scalar instructions.

`0x8006`, **SVE_INST_SPEC, SVE Operations speculatively executed**

The counter counts speculatively executed operations due to SVE instructions. It is IMPLEMENTATION DEFINED whether this event counts operations due to non-SIMD SVE instructions

`0x8007`, **ASE_SVE_INST_SPEC, Advanced SIMD and SVE Operations speculatively executed**

The counter counts speculatively executed operations that would be counted by ASE_INST_SPEC or SVE_INST_SPEC.

`0x8008`, **UOP_SPEC, Microarchitectural operation, Operations speculatively executed**

The counter counts all speculatively executed microarchitectural operations, irrespective of the IMPLEMENTATION DEFINED interpretation of Operations speculatively executed.

`0x8009`, **ASE_UOP_SPEC, Advanced SIMD Microarchitectural operation, Operations speculatively executed**

The counter counts all speculatively executed microarchitectural operations due to Advanced SIMD instructions, irrespective of the IMPLEMENTATION DEFINED interpretation of Operations speculatively executed. It is IMPLEMENTATION DEFINED whether this event counts microarchitectural operations due to Advanced SIMD scalar instructions.

`0x800A`, **SVE_UOP_SPEC, SVE micro-operation, Speculatively executed**

The counter counts all speculatively executed microarchitectural operations due to SVE instructions, irrespective of the IMPLEMENTATION DEFINED interpretation of Operations speculatively executed. It is IMPLEMENTATION DEFINED whether this event counts microarchitectural operations due to non-SIMD SVE instructions.

`0x800B`, **ASE_SVE_UOP_SPEC, Advanced SIMD and SVE Microarchitectural operation, Operations speculatively executed**

The counter counts all speculatively executed microarchitectural operations that are counted by SVE_UOP_SPEC or ASE_UOP_SPEC.

`0x800C`, **SIMD_UOP_SPEC, SIMD micro-operation, Speculatively executed**

The counter counts the following speculatively executed microarchitectural operations, irrespective of the IMPLEMENTATION DEFINED interpretation of Operations speculatively executed, due to:

- SVE instructions, but not non-SIMD SVE instructions.
- Advanced SIMD instructions, but not Advanced SIMD scalar instructions.

`0x800E`, **SVE_MATH_SPEC, SVE Math accelerator Operations speculatively executed**

The counter counts speculatively executed math function operations due to the SVE `FTSMUL`, `FTMAD`, `FTSSEL`, and `FEXPA` instructions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0x8010`, **FP_SPEC, Floating-point Operations speculatively executed**

The counter counts speculatively executed operations due to scalar, Advanced SIMD, and SVE floating-point instructions. These instructions are in the floating point instructions category and optionally the floating-point conversions instructions category and the floating-point or integer instructions category listed in the *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*.

Note

The IMPLEMENTATION DEFINED event counter VFP_SPEC is similar to this event counter, but does not count SIMD operations.

`0x8011`, **ASE_FP_SPEC, Advanced SIMD floating-point Operations speculatively executed**

The counter counts speculatively executed operations due to Advanced SIMD floating-point instructions. These instructions are in the floating point instructions category and optionally the floating-point conversions instructions category and the floating-point or integer instructions category listed in the *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*.

`0x8012`, **SVE_FP_SPEC, SVE floating-point Operations speculatively executed**

The counter counts speculatively executed operations due to SVE floating-point instructions. These instructions are in the floating point instructions category and optionally the floating-point conversions instructions category and the floating-point or integer instructions category listed in the *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*.

`0x8013`, **ASE_SVE_FP_SPEC, Advanced SIMD and SVE floating-point Operations speculatively executed**

The counter counts speculatively executed operations due to Advanced SIMD and SVE floating-point instructions. These instructions are in the floating point instructions category and optionally the floating-point conversions instructions category and the floating-point or integer instructions category listed in the *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*.

`0x8014`, **FP_HP_SPEC, Half-precision floating-point Operations speculatively executed**

The counter counts speculatively executed operations due to scalar, Advanced SIMD, and SVE floating-point instructions, where the largest type is half-precision. These instructions are in the floating point instructions category and optionally the floating-point conversions instructions category and the floating-point or integer instructions category listed in the *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*

`0x8015`, **ASE_FP_HP_SPEC, Advanced SIMD, Half-precision floating-point Operations speculatively executed**

The counter counts speculatively executed operations due to Advanced SIMD floating-point instructions, where the largest type is half-precision. These instructions are in the floating point instructions category and optionally the floating-point conversions instructions category and the floating-point or integer instructions category listed in the *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*.

`0x8016`, **SVE_FP_HP_SPEC, SVE Half-precision floating-point Operations speculatively executed**

The counter counts speculatively executed operations due to SVE floating-point instructions, where the largest type is half-precision. These instructions are in the floating point instructions category and optionally the floating-point conversions instructions category and the floating-point or integer instructions category listed in the *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0x8017`, **ASE_SVE_FP_HP_SPEC, Advanced SIMD and SVE Half-precision floating-point Operations speculatively executed**

The counter counts speculatively executed operations due to Advanced SIMD and SVE instructions, where the largest type is half-precision. These instructions are in the floating point instructions category and optionally the floating-point conversions instructions category and the floating-point or integer instructions category listed in the *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*.

`0x8018`, **FP_SP_SPEC, Single-precision floating-point Operations speculatively executed**

The counter counts speculatively executed operations due to scalar, Advanced SIMD, and SVE floating-point instructions, where the largest type is single-precision. These instructions are in the floating point instructions category and optionally the floating-point conversions instructions category and the floating-point or integer instructions category listed in the *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*.

`0x8019`, **ASE FP_SP_SPEC, Advanced SIMD Single-precision floating-point Operations speculatively executed**

The counter counts speculatively executed operations due to Advanced SIMD floating-point instructions, where the largest type is single-precision. These instructions are in the floating point instructions category and optionally the floating-point conversions instructions category and the floating-point or integer instructions category listed in the *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*.

`0x801A`, **SVE FP_SP_SPEC, SVE Single-precision floating-point operations, Operations speculatively executed**

The counter counts speculatively executed operations due to SVE floating-point instructions, where the largest type is single-precision. These instructions are in the floating point instructions category and optionally the floating-point conversions instructions category and the floating-point or integer instructions category listed in the *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*.

`0x801B`, **ASE_SVE FP_SP_SPEC, Advanced SIMD and SVE Single-precision floating-point Operations speculatively executed**

The counter counts speculatively executed operations due to Advanced SIMD and SVE instructions., where the largest type is single-precision. These instructions are in the floating point instructions category and optionally the floating-point conversions instructions category and the floating-point or integer instructions category listed in the *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*.

`0x801C`, **FP_DP_SPEC, Double-precision floating-point Operations speculatively executed**

The counter counts speculatively executed operations due to scalar, Advanced SIMD, and SVE floating-point instructions, where the largest type is double-precision. These instructions are in the floating point instructions category and optionally the floating-point conversions instructions category and the floating-point or integer instructions category listed in the *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*

`0x801D`, **ASE_FP_DP_SPEC, Advanced SIMD Double-precision floating-point Operations speculatively executed**

The counter counts speculatively executed operations due to Advanced SIMD floating-point instructions, where the largest type is double-precision. These instructions are in the floating point instructions category and optionally the floating-point conversions instructions category and the floating-point or integer instructions category listed in the *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0x801E`, **SVE_FP_DP_SPEC, SVE Double-precision floating-point Operations speculatively executed**

The counter counts speculatively executed operations due to SVE floating-point instructions, where the largest type is double-precision. These instructions are in the floating point instructions category and optionally the floating-point conversions instructions category and the floating-point or integer instructions category listed in the *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*.

`0x801F`, **ASE_SVE_FP_DP_SPEC, Advanced SIMD and SVE Double-precision floating-point Operations speculatively executed**

The counter counts speculatively executed operations due to Advanced SIMD and SVE floating-point instructions, where the largest type is double-precision. These instructions are in the floating point instructions category and optionally the floating-point conversions instructions category and the floating-point or integer instructions category listed in the *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*.

`0x8020`, **FP_DIV_SPEC, Floating-point divide Operations speculatively executed**

The counter counts speculatively executed floating-point divide operations.

`0x8021`, **ASE_FP_DIV_SPEC, Advanced SIMD Floating-point divide Operations speculatively executed**

The counter counts speculatively executed Advanced SIMD floating point divide operations.

`0x8022`, **SVE_FP_DIV_SPEC, SVE Floating-point divide Operations speculatively executed**

The counter counts speculatively executed SVE floating-point divide operations.

`0x8023`, **ASE_SVE_FP_DIV_SPEC, Advanced SIMD and SVE Floating-point divide Operations speculatively executed**

The counter counts speculatively executed Advanced SIMD and SVE floating-point divide operations.

`0x8024`, **FP_SQRT_SPEC, Floating-point square-root Operations speculatively executed**

The counter counts speculatively executed floating-point square-root operations.

`0x8025`, **ASE_FP_SQRT_SPEC, Advanced SIMD Floating-point square-root Operations speculatively executed**

The counter counts speculatively executed Advanced SIMD floating-point square-root operations.

`0x8026`, **SVE_FP_SQRT_SPEC, SVE Floating-point square-root Operations speculatively executed**

The counter counts speculatively executed SVE floating-point square-root operations.

`0x8027`, **ASE_SVE_FP_SQRT_SPEC, Advanced SIMD and SVE Floating-point square root Operations speculatively executed**

The counter counts speculatively executed Advanced SIMD and SVE floating point square-root operations.

`0x8028`, **FP_FMA_SPEC, Floating-point FMA Operations speculatively executed**

The counter counts speculatively executed floating point fused multiply-add and multiply-subtract operations due to the following instructions:

- Scalar: `FMADD`, `FMSUB`, `FNMADD`, `FNMSUB`.
- Advanced SIMD: `FCMLA`, `FMLA`, `FMLS`.
- SVE: `FCMLA`, `FMAD`, `FMLA`, `FMLS`, `FMSB`, `FNMAD`, `FNMLA`, `FNMLS`, `FNMSB`, `FTMAD`.

`0x8029`, **ASE_FP_FMA_SPEC, Advanced SIMD Floating-point FMA Operations speculatively executed**

The counter counts speculatively executed floating point multiply-add and multiply-subtract operations due to the Advanced SIMD `FCMLA`, `FMLA`, and `FMLS` instructions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0x802A`, **SVE_FP_FMA_SPEC, SVE Floating-point FMA Operations speculatively executed**

The counter counts speculatively executed floating point multiply-add and multiply-subtract operations due to the SVE `FCMLA`, `FMAD`, `FMLA`, `FMLS`, `FMSB`, `FNMAD`, `FNMLA`, `FNMLS`, `FNMSB`, `FTMAD` instructions.

`0x802B`, **ASE_SVE_FP_FMA_SPEC, Advanced SIMD and SVE Floating-point FMA Operations speculatively executed**

The counter counts speculatively executed floating-point fused multiply-add and multiply-subtract operations due to the following instructions:

- Advanced SIMD: `FCMLA`, `FMLA`, `FMLS`.
- SVE: `FCMLA`, `FMAD`, `FMLA`, `FMLS`, `FMSB`, `FNMAD`, `FNMLA`, `FNMLS`, `FNMSB`, `FTMAD`.

`0x802C`, **FP_MUL_SPEC, Floating-point multiply Operations speculatively executed**

The counter counts speculatively executed floating-point multiply operations due to the scalar, Advanced SIMD, and SVE `FMUL` and `FMULX` instructions and the SVE `FTSMUL` instruction.

`0x802D`, **ASE_FP_MUL_SPEC, Advanced SIMD Floating-point multiply Operations speculatively executed**

The counter counts speculatively executed floating-point multiply operations due to the scalar Advanced SIMD instructions `FMUL` and `FMULX`.

`0x802E`, **SVE_FP_MUL_SPEC, SVE Floating-point multiply Operations speculatively executed**

The counter counts speculatively executed floating-point multiply operations due to SVE instructions `FMUL`,`FMULX`, and `FTSMUL`.

`0x802F`, **ASE_SVE_FP_MUL_SPEC, Advanced SIMD and SVE Floating-point multiply Operations speculatively executed**

The counter counts speculatively executed floating-point multiply operations due to the Advanced SIMD and SVE `FMUL` and `FMULX` instructions and the SVE `FTSMUL` instruction.

`0x8030`, **FP_ADDSUB_SPEC, floating-point add or subtract Operations speculatively executed**

The counter counts speculatively executed floating-point add and subtract operations due to the scalar, Advanced SIMD, and SVE `FADD` and `FSUB` instructions, and the Advanced SIMD and SVE FABD instructions.

`0x8031`, **ASE_FP_ADDSUB_SPEC, Advanced SIMD floating-point add or subtract Operations speculatively executed**

The counter counts speculatively executed floating-point add and subtract operations due to the Advanced SIMD `FABD`, `FADD`, and `FSUB` instructions.

`0x8032`, **SVE_FP_ADDSUB_SPEC, SVE floating-point add or subtract Operations speculatively executed**

The counter counts speculatively executed floating-point add and subtract operations due to the SVE `FABD`, `FADD`, and `FSUB` instructions.

`0x8033`, **ASE_SVE_FP_ADDSUB_SPEC, Advanced SIMD and SVE floating-point add and subtract Operations speculatively executed**

The counter counts speculatively executed floating-point add and subtract operations due to the Advanced SIMD and SVE `FABD`, `FADD`, and `FSUB` instructions.

`0x8034`, **FP_RECPE_SPEC, Floating-point reciprocal estimate Operations speculatively executed**

The counter counts speculatively executed floating-point reciprocal estimate operations due to the Advanced SIMD scalar, Advanced SIMD vector, and SVE `FRECPE` and `FRSQRTE` instructions.

`0x8035`, **ASE_FP_RECPE_SPEC, Advanced SIMD floating-point reciprocal estimate Operations speculatively executed**

The counter counts speculatively executed floating-point reciprocal estimate operations due to the Advanced SIMD vector `FRECPE` and `FRSQRTE` instructions.

`0x8036`, **SVE_FP_RECPE_SPEC, SVE floating-point reciprocal estimate Operations speculatively executed**

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The counter counts speculatively executed floating-point reciprocal estimate operations due to the SVE FRECPE and FRSQRTE instructions.

0x8037, ASE_SVE_FP_RECPE_SPEC, Advanced SIMD and SVE floating-point reciprocal estimate Operations speculatively executed

The counter counts speculatively executed floating-point reciprocal estimate operations due to Advanced SIMD vector and SVE FRECPE and FRSQRTE instructions.

0x8038, FP_CVT_SPEC, floating-point convert Operations speculatively executed

The counter counts speculatively executed floating-point convert operations due to the scalar, Advanced SIMD, and SVE floating-point conversion instructions. The instructions in the Floating-point conversions category are listed in the *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*.

0x8039, ASE_FP_CVT_SPEC, Advanced SIMD floating-point convert Operations speculatively executed

The counter counts speculatively executed floating-point convert operations due to the Advanced SIMD floating-point conversion instructions. The instructions in the Floating-point conversions category are listed in the *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*.

0x803A, SVE_FP_CVT_SPEC, SVE floating-point convert Operations speculatively executed

The counter counts speculatively executed floating-point convert operations due to the SVE floating-point conversion instructions. The instructions in the Floating-point conversions category are listed in the *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*.

0x803B, ASE_SVE_FP_CVT_SPEC, Advanced SIMD and SVE floating-point convert Operations speculatively executed

The counter counts speculatively executed floating-point convert operations due to the Advanced SIMD and SVE floating-point conversion instructions. The instructions in the Floating-point conversions category are listed in the *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*.

0x803C, SVE_FP_AREDUCE_SPEC, SVE floating-point accumulating reduction Operations speculatively executed

The counter counts speculatively executed floating-point accumulating reduction operations due to the SVE FADDA instruction.

0x803D, ASE_FP_PREDUCE_SPEC, Advanced SIMD floating-point pairwise add step Operations speculatively executed

The counter counts speculatively executed floating-point pairwise add operations due to the Advanced SIMD FADDP instruction.

0x803E, SVE_FP_VREDUCE_SPEC, SVE floating-point vector reduction Operations speculatively executed

The counter counts speculatively executed floating-point treewise reduction operations due to the SVE FADDV, FMAXNMV, FMAXV, FMINNMV, and FMINV instructions.

0x803F, ASE_SVE_FP_VREDUCE_SPEC, Advanced SIMD and SVE floating-point vector reduction Operations speculatively executed

The counter counts speculatively executed floating-point reduction operations due to the Advanced SIMD and SVE FMAXNMV, FMAXV, FMINNMV, and FMINV instructions, the Advanced SIMD FADDP instruction, and the SVE FADDV instruction.

0x8040, INT_SPEC, integer Operations speculatively executed

The counter counts speculatively executed integer arithmetic operations due to scalar, Advanced SIMD, and SVE data-processing instructions. These instructions are listed in the integer instructions category and optionally the floating-point conversions category and the floating-point or integer category in the *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0x8041`, **ASE_INT_SPEC, Advanced SIMD integer Operations speculatively executed**

The counter counts speculatively executed integer arithmetic operations due to Advanced SIMD data-processing instructions. These instructions are listed in the integer instructions category and optionally the floating-point conversions category and the floating-point or integer category in the *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*.

`0x8042`, **SVE_INT_SPEC, SVE integer Operations speculatively executed**

The counter counts speculatively executed integer arithmetic operations due to SVE data-processing instructions. These instructions are listed in the integer instructions category and optionally the floating-point conversions category and the floating-point or integer category in the *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*.

`0x8043`, **ASE_SVE_INT_SPEC, Advanced SIMD and SVE integer Operations speculatively executed**

The counter counts speculatively executed integer arithmetic operations due to Advanced SIMD and SVE data-processing instructions. These instructions are listed in the integer instructions category and optionally the floating-point conversions category and the floating-point or integer category in the *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*.

`0x8044`, **INT_DIV_SPEC, integer divide Operations speculatively executed**

The counter counts speculatively executed scalar and SVE integer divide operations due to the `SDIV` and `UDIV` instructions.

`0x8045`, **INT_DIV64_SPEC, 64-bit integer divide Operations speculatively executed**

The counter counts speculatively executed scalar and SVE integer divide operations due to the `SDIV` and `UDIV` instructions with 64-bit operands or vector elements.

`0x8046`, **SVE_INT_DIV_SPEC, SVE integer divide Operations speculatively executed**

The counter counts speculatively executed SVE integer divide operations due to the SVE `SDIV` and `UDIV` instructions.

`0x8047`, **SVE_INT_DIV64_SPEC, SVE 64-bit integer divide Operations speculatively executed**

The counter counts speculatively executed SVE integer divide operations due to the SVE `SDIV` and `UDIV` instructions with 64-bit vector elements.

`0x8048`, **INT_MUL_SPEC, integer multiply Operations speculatively executed**

The counter counts speculatively executed integer multiply operations due to the following instructions:

- Scalar: `MADD`, `MSUB`, `MUL`, `SMADDL`, `SMULH`, `UMADDL`, `UMULH`.
- Advanced SIMD: `MLA`, `MLS`, `MUL`, `PMUL`, `PMULL`, `SMLAL`, `SMLS`, `SMULL`, `SQMLAL`, `SQDMLSL`, `SQDMULH`, `SQDMULL`, `SQRDMLAH`, `SQRDMLSH`, `SQRDMULH`, `UMLAL`, `UMLSL`, `UMULL`.
- SVE: `MAD`, `MLA`, `MLS`, `MSB`, `MUL`, `SMULH`, `UMULH`.

`0x8049`, **ASE_INT_MUL_SPEC, Advanced SIMD integer multiply Operations speculatively executed**

The counter counts speculatively executed integer multiply operations due to the following Advanced SIMD instructions: `MLA`, `MLS`, `MUL`, `PMUL`, `PMULL`, `SMLAL`, `SMLSL`, `SMULL`, `SQDMLAL`, `SQDMLSL`, `SQDMULH`, `SQDMULL`, `SQRDMLAH`, `SQRDMLSH`, `SQRDMULH`, `UMLAL`, `UMLSL`, `UMULL`.

`0x804A`, **SVE_INT_MUL_SPEC, SVE integer multiply Operations speculatively executed**

The counter counts speculatively executed integer multiply operations due to the following SVE instructions: `MAD`, `MLA`, `MLS`, `MSB`, `MUL`, `SMULH`, `UMULH`.

`0x804B`, **ASE_SVE_INT_MUL_SPEC, Advanced SIMD and SVE integer multiply Operations speculatively executed**

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The counter counts speculatively executed integer multiply operations due to the following instructions:

- Advanced SIMD: `MLA`, `MLS`, `MUL`, `PMUL`, `PMULL`, `SMLAL`, `SMLSL`, `SMULL`, `SQDMLAL`, `SQDMLSL`, `SQDMULH`, `SQDMULL`, `SQRDMLAH`, `SQRDMLSH`, `SQRDMULH`, `UMLAL`, `UMLSL`, `UMULL`.
- SVE: `MAD`, `MLA`, `MLS`, `MSB`, `MUL`, `SMULH`, `UMULH`.

`0x804C`, **INT_MUL64_SPEC, integer 64x64 multiply Operations speculatively executed**

The counter counts speculatively executed integer multiply operations returning a 64-bit result for the following instructions:

- Scalar: `MADD`, `MSUB`, `MUL`, `SMADDL`, `SMULH`, `UMADDL`, `UMULH`.
- SVE: `MAD`, `MLS`, `MLA`, `MSB`, `MUL`, `SMULH`, `UMULH`.

`0x804D`, **SVE_INT_MUL64_SPEC, SVE integer 64-bit multiply Operations speculatively executed**

The counter counts speculatively executed integer multiply operations returning a 64-bit result for the following SVE instructions: `MAD`, `MLA`, `MLS`, `MSB`, `MUL`, `SMULH`, `UMULH`.

`0x804E`, **INT_MULH64_SPEC, integer 64-bit multiply returning high part Operations speculatively executed**

The counter counts speculatively executed widening integer multiply operations returning a 64-bit result for the scalar and SVE `SMULH` and `UMULH` instructions.

`0x804F`, **SVE_INT_MULH64_SPEC, SVE integer 64-bit multiply high part Operations speculatively executed**

The counter counts speculatively executed widening integer multiply operations returning a 64-bit result for the SVE `SMULH` and `UMULH` instructions.

`0x8058`, **NONFP_SPEC, Non floating-point Operations speculatively executed**

The counter counts speculatively executed operations due to the following instructions:

- Scalar instructions that would be counted by the DP_SPEC event.
- Advanced SIMD data processing instructions defined in *Data processing - SIMD and floating-point* on page C3-239 that would not be counted by FP_SPEC.
- SVE instructions with vector source or destination registers that would not be counted by FP_SPEC.

`0x8059`, **ASE_NONFP_SPEC, Advanced SIMD non-floating-point Operations speculatively executed**

The counter counts speculatively executed operations due to Advanced SIMD data processing instructions defined in the section titled *Data processing - SIMD and floating-point* on page C3-239 that would not be counted by ASE_FP_SPEC.

`0x805A`, **SVE_NONFP_SPEC, SVE non-floating-point Operations speculatively executed**

The counter counts speculatively executed operations due to SVE instructions with vector source or destination registers that would not be counted by SVE_FP_SPEC.

`0x805B`, **ASE_SVE_NONFP_SPEC, Advanced SIMD and SVE non-floating-point Operations speculatively executed**

The counter counts speculatively executed operations due to the following instructions:

- Advanced SIMD data-processing instructions defined in *Data processing - SIMD and floating-point* on page C3-239 that would not be counted by ASE_SVE_FP_SPEC.
- SVE instructions with vector source or destination registers that would not be counted by ASE_SVE_FP_SPEC.

`0x805D`, **ASE_INT_VREDUCE_SPEC, Advanced SIMD integer reduction Operations speculatively executed**

The counter counts speculatively executed across-vector and pairwise integer reduction operations due to the Advanced SIMD `SADDLP`, `SADDLV`, `SMAXP`, `SMAXV`, `SMINP`, `SMINV`, `UADDVL`, `UMAXV`, and `UMINV` instructions.

`0x805E`, **SVE_INT_VREDUCE_SPEC, SVE integer reduction Operations speculatively executed**

The counter counts speculatively executed across-vector integer reduction operations due to the following SVE instructions: `ANDV`, `EORV`, `ORV`, `SADDV`, `SMAXV`, `SMINV`, `UADDV`, `UMAXV`, and `UMINV` instructions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

0x805F, ASE_SVE_INT_VREDUCE_SPEC, Advanced SIMD and SVE integer reduction Operations speculatively executed

The counter counts speculatively executed across-vector and pairwise integer reduction operations due to the following instructions:

- Advanced SIMD: SADDLP, SADDLV, SMAXP, SMAXV, SMINP, SMINV, UADDLV, UMAXV, and UMINV.
- SVE: ANDV, EORV, ORV, SADDV, SMAXV, SMINV, UADDV, UMAXV, UMINV.

0x8060, SVE_PERM_SPEC, SVE permute Operations speculatively executed

The counter counts speculatively executed vector or predicate permute operations due to the following SVE instructions: CLASTA, CLASTB, CPY, COMPACT, DUP, EXT, INSR, LASTA, LASTB, PUNPKHI, PUNPKLO, REV, REV16, REV32, REV64, SPLICE, SUNPKHI, SUNPKLO, TBL, TRN1, TRN2, UUNPKHI, UUNPKLO, UZP1, UZP2, ZIP1, and ZIP2.

0x8061, SVE_PERM_IGRANULE_SPEC, SVE intra-granule permute Operations speculatively executed

The counter counts speculatively executed vector or predicate permute operations within a 128-bit vector granule or 16-bit predicate granule for the following SVE instructions: REV16, REV32, REV64, TRN1, TRN2.

0x8062, SVE_PERM_XGRANULE_SPEC, SVE cross-granule permute Operations speculatively executed

The counter counts speculatively executed vector or predicate permute operations that can cross between 128-bit vector granules or 16-bit predicate granules for the following SVE instructions: CLASTA, CLASTB, CPY, COMPACT, DUP, EXT, INSR, LASTA, LASTB, PUNPKHI, PUNPKLO, REV, SPLICE, SUNPKHI, SUNPKLO, TBL, UNPKHI, UNPKLO, UZP1, UZP2, ZIP1, and ZIP2.

0x8063, SVE_PERM_VARIABLE_SPEC, SVE programmable permute Operations speculatively executed

The counter counts speculatively executed variable vector permute operations due to the following SVE instructions: CLASTA, CLASTB, COMPACT, LASTA, LASTB, SPLICE, and TBL.

0x8064, SVE_XPIPE_SPEC, SVE cross-pipe Operations speculatively executed

The counter counts speculatively executed cross-pipeline transfer operations due to the following SVE instructions: CLASTA (scalar), CLASTB (scalar), CNTP, CPY (scalar), DECP (scalar), DUP (scalar), INCP (scalar), INDEX (immediate, scalar), INDEX (scalar, immediate), INDEX (scalar, scalar), INSR (scalar), LASTA (scalar), LASTB (scalar), SQDECP (scalar), SQINCP (scalar), UQDECP (scalar), UQDECP (scalar), WHILE<cc>.

0x8065, SVE_XPIPE_Z2R_SPEC, SVE vector to scalar cross-pipe Operations speculatively executed

The counter counts speculatively executed vector to general-purpose scalar cross-pipeline transfer operations due to the following SVE instructions: CLASTA (scalar), CLASTB (scalar), CNTP, DECP (scalar), INCP (scalar), LASTA (scalar), LASTB (scalar), SQDECP (scalar), SQINCP (scalar), UQDECP (scalar), UQDECP (scalar).

0x8066, SVE_XPIPE_R2Z_SPEC, SVE scalar to vector cross-pipe Operations speculatively executed

The counter counts speculatively executed general-purpose scalar to vector cross-pipeline transfer operations due to the following SVE instructions: CPY (scalar), DUP (scalar), INDEX (immediate, scalar), INDEX (scalar, immediate), INDEX (scalar, scalar), INSR (scalar), WHILE<cc>.

0x8067, SVE_PGEN_NVEC_SPEC, SVE predicate-only Operations speculatively executed

The counter counts speculatively executed predicate-generating operations that do not read vector registers due to the following SVE instructions: AND (predicates), ANDS, BIC (predicates), BICS, BRKA, BRKAS, BRKB, BRKBS, BRKN, BRKNS, BRKPA, BRKPAS, BRKPB, BRKPBS, EOR (predicates), EORS, NAND, NANDS, NOR, NORS, ORN (predicates), ORNS, ORR (predicates), ORRS, PFALSE, PFIRST, PNEXT, PTRUE, PTRUES, PUNPKHI, PUNPKLO, RDFFR, RDFFRS, REV (predicate), SEL (predicates), TRN1 (predicates), TRN2 (predicates), UZP1 (predicates), UZP2 (predicates), WHILE<cc>, ZIP1 (predicates), ZIP2 (predicates).

0x8068, SVE_PGEN_SPEC, SVE predicate generating Operations speculatively executed

The counter counts speculatively executed predicate-generating operations due to the following SVE instructions: AND (predicates), ANDS, BIC (predicates), BICS, BRKA, BRKAS, BRKB, BRKBS, BRKN, BRKNS, BRKPA, BRKPAS, BRKPB, BRKPBS, CMP<cc>, EOR (predicates), EORS, FAC<cc>, FCM<cc>, NAND, NANDS, NOR, NORS,

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ORN (predicates), ORNS, ORR (predicates), ORRS, PFALSE, PFIRST, PNEXT, PTRUE, PTRUES, PUNPKHI, PUNPKLO, RDFFR, RDFFRS, REV (predicate), SEL (predicates), TRN1 (predicates), TRN2 (predicates), UZP1 (predicates), UZP2 (predicates), WHILE<cc>, ZIP1 (predicates), ZIP2 (predicates).

0x8069, SVE_PGEN_FLG_SPEC, SVE predicate flag setting Operations speculatively executed

The counter counts speculatively executed predicate-generating operations that set condition flags, due to the following SVE instructions: ANDS, BICS, BRKAS, BRKBS, BRKNS, BRKPAS, BRKPBS, CMP<cc>, EORS, NANDS, NORS, ORNS, ORRS, PFIRST, PNEXT, PTRUES, RDFFRS, WHILE<cc>.

0x806A, SVE_PGEN_CMP_SPEC, SVE vector compare Operations speculatively executed

The counter counts speculatively executed vector compare operations due to the following SVE instructions: CMP<cc>, FAC<cc>, FCM<cc>.

0x806B, SVE_PGEN_FCM_SPEC, SVE floating-point vector compare Operations speculatively executed

The counter counts speculatively executed vector floating-point compare operations, due to the following SVE instructions: FAC<cc>, FCM<cc>.

0x806C, SVE_PGEN_LOGIC_SPEC, SVE predicate logical Operations speculatively executed

The counter counts speculatively executed predicate logical due to the following SVE instructions: AND (predicates), ANDS, BIC (predicates), BICS, EOR (predicates), EORS, NAND, NANDS, NOR, NORS, ORN (predicates), ORNS, ORR (predicates), ORRS.

0x806D, SVE_PPERM_SPEC, SVE predicate permute Operations speculatively executed

The counter counts speculatively executed predicate permute operations, due to the following SVE instructions: PUNPKHI, PUNPKLO, REV (predicate), TRN1 (predicates), TRN2 (predicates), UZP1 (predicates), UZP2 (predicates), ZIP1 (predicates), ZIP2 (predicates).

0x806E, SVE_PSCAN_SPEC, SVE predicate scan Operations speculatively executed

The counter counts speculatively executed predicate scanning and generation operations, due to the following SVE instructions: BRKA, BRKAS, BRKB, BRKBS, BRKN, BRKNS, BRKPA, BRKPAS, BRKPB, BRKPBS, PFIRST, PNEXT.

0x806F, SVE_PCNT_SPEC, SVE predicate count Operations speculatively executed

The counter counts speculatively executed predicate population count operations, due to the following SVE instructions: CNTP, DECP, INCP, SQDECP, SQINCP, UQDECP, UQINCP.

0x8070, SVE_PLOOP_WHILE_SPEC, SVE predicate loop while Operations speculatively executed

The counter counts speculatively executed counted predicate generation operations, due to the following SVE instructions: WHILELE, WHILELO, WHILELS, WHILELT.

0x8071, SVE_PLOOP_TEST_SPEC, SVE predicate loop test Operations speculatively executed

The counter counts speculatively executed loop predicate test operations, due to the following SVE instructions: BRKAS, BRKBS, BRKNS, BRKPAS, BRKPBS, WHILELE, WHILELO, WHILELS, WHILELT.

0x8072, SVE_PLOOP_ELTS_SPEC, SVE predicate loop elements Operations speculatively executed

The counter counts speculatively executed loop predicate generation operations, due to the following SVE instructions: WHILELE, WHILELO, WHILELS, WHILELT. This event increments the counter by (128 ÷ CSIZE).

Note

This counter must be multiplied by (VL ÷ 128) to determine the number of vector elements speculatively processed by while loops.

0x8073, SVE_PLOOP_TERM_SPEC, SVE predicate loop termination, Operations speculatively executed

The counter counts speculatively executed loop-terminating predicate generation operations due to the following SVE instructions:

- WHILELE, WHILELO, WHILELS, WHILELT, which set PSTATE.N to 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- `BRKAS`, `BRKBS`, `BRKNS`, `BRKPAS`, `BRKPBS`, which set PSTATE.C to 1.
- `CTERMEQ` and `CTERMNE`, which set PSTATE.N to 1 and PSTATE.V to 0.

`0x8074`, **SVE_PRED_SPEC, SVE predicated Operations speculatively executed**

The counter counts speculatively executed SIMD data-processing and load and store operations due to SVE instructions with a Governing predicate operand that determines the Active elements.

`0x8075`, **SVE_PRED_EMPTY_SPEC, SVE predicated operations with no active predicates, Operations speculatively executed**

The counter counts speculatively executed SIMD data-processing and load and store operations due to SVE instructions with a Governing predicate in which all elements are FALSE.

`0x8076`, **SVE_PRED_FULL_SPEC, SVE predicated operations with all active predicates, Operations speculatively executed**

The counter counts speculatively executed SIMD data-processing and load and store operations due to SVE instructions with a Governing predicate in which all elements are TRUE.

`0x8077`, **SVE_PRED_PARTIAL_SPEC, SVE predicated operations with partially active predicates, Operations speculatively executed**

The counter counts speculatively executed SIMD data-processing and load and store operations due to SVE instructions with a Governing predicate in which elements are neither all TRUE nor all FALSE.

`0x8078`, **SVE_UNPRED_SPEC, SVE unpredicated Operations speculatively executed**

The counter counts speculatively executed SIMD data-processing and load/store operations due to SVE instructions without a Governing predicate.

`0x8079`, **SVE_PRED_NOT_FULL_SPEC, SVE predicated operations with empty or partially active predicates, Operations speculatively executed**

The counter counts speculatively executed SIMD data-processing and load and store operations due to SVE instructions with a Governing predicate in which elements are not all TRUE, but may be all FALSE.

`0x807C`, **SVE_MOVPRFX_SPEC, SVE MOVPRFX Operations speculatively executed**

The counter counts speculatively executed operations due to `MOVPRFX` instructions, whether or not they were fused with the prefixed instruction.

`0x807D`, **SVE_MOVPRFX_Z_SPEC, SVE MOVPRFX zeroing predication Operations speculatively executed**

The counter counts speculatively executed operations due to `MOVPRFX` instructions using zeroing predication, whether or not they were fused with the prefixed instruction.

`0x807E`, **SVE_MOVPRFX_M_SPEC, SVE MOVPRFX merging predication Operations speculatively executed**

The counter counts speculatively executed operations due to `MOVPRFX` instructions using merging predication, whether or not they were fused with the prefixed instruction.

`0x807F`, **SVE_MOVPRFX_U_SPEC, SVE MOVPRFX unfused Operations speculatively executed**

The counter counts speculatively executed operations due to `MOVPRFX` instructions that were not fused with the prefixed instruction.

`0x8080`, **SVE_LDST_SPEC, SVE load, store, and prefetch Operations speculatively executed**

The counter counts speculatively executed operations that read from, write to, or prefetch memory due to SVE instructions.

`0x8081`, **SVE_LD_SPEC, SVE load Operations speculatively executed**

The counter counts speculatively executed operations that read from memory due to SVE load instructions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

0x8082, SVE_ST_SPEC, SVE store Operations speculatively executed

The counter counts speculatively executed operations that write to memory due to SVE store instructions.

0x8083, SVE_PRF_SPEC, SVE prefetch Operations speculatively executed

The counter counts speculatively executed operations that prefetch memory due to SVE prefetch instructions.

0x8084, ASE_SVE_LDST_SPEC, Advanced SIMD and SVE, load and store Operations speculatively executed

The counter counts speculatively executed operations that read from or write to memory due to SVE and Advanced SIMD instructions, or any instructions that prefetch memory.

0x8085, ASE_SVE_LD_SPEC, Advanced SIMD and SVE load Operations speculatively executed

The counter counts speculatively executed operations that read from memory due to SVE and Advanced SIMD load instructions.

0x8086, ASE_SVE_ST_SPEC, Advanced SIMD and SVE store Operations speculatively executed

The counter counts speculatively executed operations that write to memory due to SVE and Advanced SIMD store instructions.

0x8087, PRF_SPEC, Prefetch Operations speculatively executed

The counter counts speculatively executed prefetch operations due to scalar `PRFM` and SVE `PRF` instructions.

0x8088, BASE_LDST_REG_SPEC, General-purpose register load, store and prefetch Operations speculatively executed

The counter counts speculatively executed operations that read from memory to a general-purpose register, write a general-purpose register to memory, or prefetch memory due to the `PRFM` instruction. It is IMPLEMENTATION DEFINED whether operations due to the `DC ZVA` instruction are counted.

0x8089, BASE_LD_REG_SPEC, General-purpose register load Operations speculatively executed

The counter counts speculatively executed operations that read from memory due to an instruction that loads a general-purpose register.

0x808A, BASE_ST_REG_SPEC, General-purpose register store Operations speculatively executed

The counter counts speculatively executed operations that write to memory due to an instruction that stores a general-purpose register. It is IMPLEMENTATION DEFINED whether operations due to the `DC ZVA` instruction are counted.

0x808B, BASE_PRF_SPEC, General-purpose register prefetch Operations speculatively executed

The counter counts speculatively executed operations that prefetch memory due to the `PRFM` instruction.

0x808C, FPASE_LDST_REG_SPEC, Floating-point and Advanced SIMD register load and store Operations speculatively executed

The counter counts speculatively executed operations that read from or write to memory, due to scalar SIMD&FP `LDR`, `LDP`, `STR`, and `STP` instructions or Advanced SIMD `LD1`, `LD1R`, and `ST1` instructions.

0x808D, FPASE_LD_REG_SPEC, Floating-point and Advanced SIMD register load Operations speculatively executed

The counter counts speculatively executed operations that read from memory, due to scalar SIMD&FP `LDR` and `LDP` instructions or Advanced SIMD `LD1` and `LD1R` instructions.

0x808E, FPASE_ST_REG_SPEC, Floating-point and Advanced SIMD register store Operations speculatively executed

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The counter counts speculatively executed operations that write to memory, due to scalar SIMD&FP STR and STP instructions or Advanced SIMD ST1 instructions.

0x8090, **SVE_LDST_REG_SPEC, SVE unpredicated load and store register Operations speculatively executed**

The counter counts speculatively executed operations that read from memory or write to memory due to SVE LDR and STR instructions.

0x8091, **SVE_LDR_REG_SPEC, SVE unpredicated load register Operations speculatively executed**

The counter counts speculatively executed operations that read from memory due to an SVE LDR instruction.

0x8092, **SVE_STR_REG_SPEC, SVE unpredicated store register Operations speculatively executed**

The counter counts speculatively executed operations that write to memory due to an SVE STR instruction.

0x8094, **SVE_LDST_PREG_SPEC, SVE load and store predicate register Operations speculatively executed**

The counter counts speculatively executed operations that read from memory or write to memory due to SVE LDR (predicate) and STR (predicate) instructions.

0x8095, **SVE_LDR_PREG_SPEC, SVE load predicate register Operations speculatively executed**

The counter counts speculatively executed operations that read from memory due to an SVE LDR (predicate) instruction.

0x8096, **SVE_STR_PREG_SPEC, SVE store predicate register Operations speculatively executed**

The counter counts speculatively executed operations that write to memory due to an SVE STR (predicate) instruction.

0x8098, **SVE_LDST_ZREG_SPEC, SVE load and store vector register Operations speculatively executed**

The counter counts speculatively executed operations that read from memory or write to memory due to SVE LDR (vector) and STR (vector) instructions.

0x8099, **SVE_LDR_ZREG_SPEC, SVE load vector register Operations speculatively executed**

The counter counts speculatively executed operations that read from memory due to an SVE LDR (vector) instruction.

0x809A, **SVE_STR_ZREG_SPEC, SVE store vector register Operations speculatively executed**

The counter counts speculatively executed operations that write to memory due to an SVE STR (vector) instruction.

0x809C, **SVE_LDST_CONTIG_SPEC, SVE contiguous load, store, and prefetch element Operations speculatively executed**

The counter counts speculatively executed operations that read from, write to, or prefetch memory due to an SVE predicated single vector contiguous element load, store, or prefetch instruction. Operations due to SVE load and replicate LD1R and LD1RQ instructions are also counted.

0x809D, **SVE_LD_CONTIG_SPEC, SVE contiguous load element Operations speculatively executed**

The counter counts speculatively executed operations that read from memory due to SVE predicated single vector contiguous element load instructions. Operations due to SVE load and replicate LD1R and LD1RQ instructions are also counted.

0x809E, **SVE_ST_CONTIG_SPEC, SVE contiguous store element Operations speculatively executed**

The counter counts speculatively executed operations that write to memory due to SVE predicated single vector contiguous element store instructions.

0x809F, **SVE_PRF_CONTIG_SPEC, SVE contiguous prefetch element Operations speculatively executed**

The counter counts speculatively executed operations that prefetch memory due to an SVE predicated single contiguous element prefetch instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0x80A0`, **SVE_LDSTNT_CONTIG_SPEC, SVE non-temporal contiguous load and store element Operations speculatively executed**

The counter counts speculatively executed operations that read from memory or write to memory with a non-temporal hint due to an SVE non-temporal contiguous element load or store instruction.

`0x80A1`, **SVE_LDNT_CONTIG_SPEC, SVE non-temporal contiguous load element Operations speculatively executed**

The counter counts speculatively executed operations that read from memory with a non-temporal hint due to an SVE non-temporal contiguous element load instruction.

`0x80A2`, **SVE_STNT_CONTIG_SPEC, SVE non-temporal contiguous store element Operations speculatively executed**

The counter counts speculatively executed operations that write to memory with a non-temporal hint due to an SVE non-temporal contiguous element store instruction.

`0x80A4`, **ASE_SVE_LDST_MULTI_SPEC, Advanced SIMD and SVE contiguous load and store multiple vector Operations speculatively executed**

The counter counts speculatively executed operations that read from memory or write to memory due to an SVE or Advanced SIMD multiple vector contiguous structure load and store instruction.

`0x80A5`, **ASE_SVE_LD_MULTI_SPEC, Advanced SIMD and SVE contiguous load multiple vector Operations speculatively executed**

The counter counts speculatively executed operations that read from memory due to SVE and Advanced SIMD multiple vector contiguous structure load instructions.

`0x80A6`, **ASE_SVE_ST_MULTI_SPEC, Advanced SIMD and SVE contiguous store multiple vector Operations speculatively executed**

The counter counts speculatively executed operations that write to memory due to SVE and Advanced SIMD multiple vector contiguous structure store instructions.

`0x80A8`, **SVE_LDST_MULTI_SPEC, SVE contiguous load and store multiple vector Operations speculatively executed**

The counter counts speculatively executed operations that read from memory or write to memory due to SVE multiple vector contiguous structure load and store instructions.

`0x80A9`, **SVE_LD_MULTI_SPEC, SVE contiguous load multiple vector Operations speculatively executed**

The counter counts speculatively executed operations that read from memory due to SVE multiple vector contiguous structure load instructions.

`0x80AA`, **SVE_ST_MULTI_SPEC, SVE contiguous store multiple vector Operations speculatively executed**

The counter counts speculatively executed operations that write to memory due to SVE multiple vector contiguous structure store instructions.

`0x80AC`, **SVE_LDST_NONCONTIG_SPEC, SVE non-contiguous load, store, and prefetch Operations speculatively executed**

The counter counts speculatively executed operations that read from, write to, or prefetch memory due to SVE non-contiguous gather-load, scatter-store, and gather-prefetch instructions.

`0x80AD`, **SVE_LD_GATHER_SPEC, SVE gather-load Operations speculatively executed**

The counter counts speculatively executed operations that read from memory due to SVE non-contiguous gather-load instructions.

`0x80AE`, **SVE_ST_SCATTER_SPEC, SVE scatter-store Operations speculatively executed**

The counter counts speculatively executed operations that write to memory due to SVE non-contiguous scatter-store instructions.

`0x80AF`, **SVE_PRF_GATHER_SPEC, SVE gather-prefetch Operations speculatively executed**

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The counter counts speculatively executed operations that prefetch memory due to SVE non-contiguous gather-prefetch instructions.

`0x80B0`, **SVE_LDST64_NONCONTIG_SPEC, SVE 64-bit non-contiguous load, store, and prefetch Operations speculatively executed**

The counter counts speculatively executed operations that read from, write to, or prefetch memory due to SVE non-contiguous gather-load, scatter-store, and gather-prefetch instructions with 64-bit vector elements in the address.

`0x80B1`, **SVE_LD64_GATHER_SPEC, SVE 64-bit gather-load Operations speculatively executed**

The counter counts speculatively executed operations that read from memory due to SVE non-contiguous gather-load instructions with 64-bit vector elements in the address.

`0x80B2`, **SVE_ST64_SCATTER_SPEC, SVE 64-bit scatter-store Operations speculatively executed**

The counter counts speculatively executed operations that write to memory due to SVE non-contiguous scatter-store instructions with 64-bit vector elements in the address.

`0x80B3`, **SVE_PRF64_GATHER_SPEC, SVE 64-bit gather-prefetch Operations speculatively executed**

The counter counts speculatively executed operations that prefetch memory due to SVE non-contiguous gather-prefetch instructions with 64-bit vector elements in the address.

`0x80B4`, **ASE_SVE_UNALIGNED_LDST_SPEC, Advanced SIMD and SVE unaligned accesses**

The counter counts memory read and write accesses due to SVE and Advanced SIMD load and store instructions where:

- A contiguous vector address is not aligned to the minimum of the in-memory size of the vector and the cache line size, in bytes.
- A gather, scatter, or single element address is not aligned to the memory element access size, in bytes.

The counter also counts unaligned accesses if they are subsequently converted into multiple aligned accesses.

`0x80B5`, **ASE_SVE_UNALIGNED_LD_SPEC, Advanced SIMD and SVE unaligned read accesses**

The counter counts memory read and write accesses due to SVE and Advanced SIMD load instructions where:

- A contiguous vector address is not aligned to the minimum of the in-memory size of the vector and the cache line size, in bytes.
- A gather, scatter or single element address is not aligned to the memory element access size, in bytes.

The counter also counts unaligned accesses if they are subsequently converted into multiple aligned accesses.

`0x80B6`, **ASE_SVE_UNALIGNED_ST_SPEC, Advanced SIMD and SVE unaligned write accesses**

The counter counts memory read and write accesses due to SVE and Advanced SIMD store instructions where:

- A contiguous vector address is not aligned to the minimum of the in-memory size of the vector and the cache line size, in bytes.
- A gather. scatter or single element address is not aligned to the memory element access size, in bytes.

The counter also counts unaligned accesses if they are subsequently converted into multiple aligned accesses.

`0x80B8`, **ASE_SVE_UNALIGNED_CONTIG_LDST_SPEC, Advanced SIMD and SVE unaligned contiguous accesses**

The counter counts memory read and write accesses due to SVE and Advanced SIMD contiguous load and store instructions where the address is not aligned to the minimum of the in-memory size of the vector and the cache line size, in bytes.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The counter also counts unaligned accesses if they are subsequently converted into multiple aligned accesses.

`0x80B9`, **ASE_SVE_UNALIGNED_CONTIG_LD_SPEC, Advanced SIMD and SVE unaligned contiguous read accesses**

The counter counts memory read accesses due to SVE and Advanced SIMD contiguous load instructions where the address is not aligned to the minimum of the in-memory size of the vector and the cache line size, in bytes.

The counter also counts unaligned accesses if they are subsequently converted into multiple aligned accesses.

`0x80BA`, **ASE_SVE_UNALIGNED_CONTIG_ST_SPEC, Advanced SIMD and SVE unaligned contiguous write accesses**

The counter counts memory write accesses due to SVE and Advanced SIMD contiguous store instructions where the address is not aligned to the minimum of the in-memory size of the vector and the cache line size, in bytes.

The counter also counts unaligned accesses if they are subsequently converted into multiple aligned accesses.

`0x80BC`, **SVE_LDFF_SPEC, SVE First-fault load Operations speculatively executed**

The counter counts speculatively executed memory read operations due to SVE First-fault and Non-fault load instructions.

`0x80BD`, **SVE_LDFF_FAULT_SPEC, SVE First-fault load operations which set FFR bit to 0, Operations speculatively executed**

The counter counts speculatively executed memory read operations due to SVE First-fault and Non-fault load instructions that write 0 to at least one bit in FFR.

`0x80C0`, **FP_SCALE_OPS_SPEC, Scalable floating-point element Operations speculatively executed**

The counter counts speculatively executed operations that would be counted by SVE_FP_SPEC, except that it is IMPLEMENTATION DEFINED whether this includes operations due to instructions other than those listed in the Floating-point arithmetic (SVE) category in the *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*.
The counter is incremented by (128 ÷ CSIZE) and by twice that amount for operations that would also be counted by SVE_FP_FMA_SPEC. See *Operation counts for dot-product and multiply-accumulate operations* on page D7-2719 for information on counts for, dot product, matrix multiplication, and BFloat16 multiply-accumulate instructions.

`0x80C1`, **FP_FIXED_OPS_SPEC, Non-scalable floating-point element Operations speculatively executed**

The counter counts speculatively executed operations that would be counted by FP_SPEC but not by SVE_FP_SPEC, and it is IMPLEMENTATION DEFINED whether this includes operations due to instructions other than those listed in the Floating-point arithmetic (scalar) category and the Floating-point arithmetic (Advanced SIMD) category in *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*. The counter is incremented by the specified number of elements for Advanced SIMD operations or by 1 for scalar operations and by twice those amounts for operations that would also be counted by FP_FMA_SPEC. See *Operation counts for dot-product and multiply-accumulate operations* on page D7-2719 for information on counts for dot product, matrix multiplication, and BFloat16 multiply-accumulate instructions.

`0x80C2`, **FP_HP_SCALE_OPS_SPEC, Scalable half-precision floating-point element Operations speculatively executed**

The counter counts speculatively executed operations that would be counted by SVE_FP_HP_SPEC, except that is IMPLEMENTATION DEFINED whether this includes operations due to instructions other than those listed in the Floating-point arithmetic (SVE) category in *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*.
The counter is incremented by 8, or by 16 for operations that would also be counted by

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

SVE_FP_FMA_SPEC. See *Operation counts for dot-product and multiply-accumulate operations* on page D7-2719 for information on counts for dot product, matrix multiplication, and BFloat16 multiply-accumulate instructions.

`0x80C3`, **FP_HP_FIXED_OPS_SPEC, Non-scalable half-precision floating-point element Operations speculatively executed**

The counter counts speculatively executed operations that would be counted by FP_HP_SPEC but not by SVE_FP_HP_SPEC, and it is IMPLEMENTATION DEFINED whether this includes operations due to instructions other than those listed in the Floating-point arithmetic (scalar) category and the Floating-point arithmetic (Advanced SIMD) category in *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*. The counter is incremented by the number of 16-bit elements for Advanced SIMD operations, or by 1 for scalar operations, and by twice those amounts for operations that would also be counted by FP_FMA_SPEC. See *Operation counts for dot-product and multiply-accumulate operations* on page D7-2719 for information on counts for dot product, matrix multiplication, and BFloat16 multiply-accumulate instructions.

`0x80C4`, **FP_SP_SCALE_OPS_SPEC, Scalable single-precision floating-point element Operations speculatively executed**

The counter counts speculatively executed operations that would be counted by SVE FP_SP_SPEC, except that is IMPLEMENTATION DEFINED whether this includes operations other than those listed in the Floating-point arithmetic (SVE) category in *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*. The counter is incremented by 4, or by 8 for operations that would also be counted by SVE_FP_FMA_SPEC. See *Operation counts for dot-product and multiply-accumulate operations* on page D7-2719 for information on counts for dot product, matrix multiplication, and BFloat16 multiply-accumulate instructions.

`0x80C5`, **FP_SP_FIXED_OPS_SPEC, Non-scalable single-precision floating-point element Operations speculatively executed**

The counter counts speculatively executed operations that would be counted by FP_SP_SPEC but not by SVE FP_SP_SPEC, and it is IMPLEMENTATION DEFINED whether this includes operations due to instructions other than those listed in the Floating-point arithmetic (scalar) category and the Floating-point arithmetic (Advanced SIMD) category in *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*. The counter is incremented by the number of 32-bit elements for Advanced SIMD operations, or by 1 for scalar operations, and by twice those amounts for operations that would also be counted by FP_FMA_SPEC. See *Operation counts for dot-product and multiply-accumulate operations* on page D7-2719 for information on counts for dot product, matrix multiplication, and BFloat16 multiply-accumulate instructions.

`0x80C6`, **FP_DP_SCALE_OPS_SPEC, Scalable double-precision floating-point element Operations speculatively executed**

The counter counts speculatively executed operations that would be counted by SVE_FP_DP_SPEC, except that is IMPLEMENTATION DEFINED whether this includes operations due to instructions other than those listed in the Floating-point arithmetic (SVE) category in *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*. The counter is incremented by 2, or by 4 for operations that would also be counted by SVE_FP_FMA_SPEC. See *Operation counts for dot-product and multiply-accumulate operations* on page D7-2719 for information on counts for dot product, matrix multiplication, and BFloat16 multiply-accumulate instructions.

`0x80C7`, **FP_DP_FIXED_OPS_SPEC, Non-scalable double-precision floating-point element Operations speculatively executed**

The counter counts speculatively executed operations that would be counted by FP_DP_SPEC but not by SVE_FP_DP_SPEC, and it is IMPLEMENTATION DEFINED whether this includes operations due to instructions other than those listed in the Floating-point arithmetic (scalar) category in *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A*. The counter is incremented by 2 for Advanced SIMD operations, or by 1 for scalar operations, and by twice those amounts for operations that would also be counted by FP_FMA_SPEC. See

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation counts for dot-product and multiply-accumulate operations on page D7-2719 for information on counts for dot product, matrix multiplication, and BFloat16 multiply-accumulate instructions.

`0x80C8`, **INT_SCALE_OPS_SPEC, Scalable integer element Operations speculatively executed**

The counter counts speculatively executed operations that would be counted by SVE_INT_SPEC. The counter is incremented by (128 ÷ CSIZE), and by twice that amount for operations due to the SVE `MAD`, `MLA`, and `MSB` instructions. See *Operation counts for dot-product and multiply-accumulate operations* on page D7-2719 for information on counts for dot product, matrix multiplication, and BFloat16 multiply-accumulate instructions.

`0x80C9`, **INT_FIXED_OPS_SPEC, Non-scalable integer element Operations speculatively executed**

The counter counts speculatively executed operations that would be counted by INT_SPEC but not by SVE_INT_SPEC. The counter is incremented by the specified number of elements for Advanced SIMD operations, or by 1 for scalar operations, and by twice those amounts for operations due to:

- Scalar: `MADD`, `MSUB`, `SMADDL`, `SMSUBL`, `UMADD`, `UMSUBL`.
- Advanced SIMD: `MLA`, `MLS`, `SMLAL`, `SMLSL`, `SQDMLAL`, `SQDMLSL`, `SQRDMLAH`, `SQRDMLSH`, `UMLAL`, `UMLSL`.

See *Operation counts for dot-product and multiply-accumulate operations* on page D7-2719 for information on counts for dot product, matrix multiplication, and BFloat16 multiply-accumulate instructions.

`0x80CA`, **LDST_SCALE_OPS_SPEC, Scalable load and store element Operations speculatively executed**

The counter counts speculatively executed memory read and write operations, due to the SVE predicated vector load and store instructions, excluding the replicating `LD1R` and `LD1RQ` instructions. For each instruction, the counter is incremented by (128 ÷ CSIZE), multiplied by the number of transferred vector register. See *Operation counts for dot-product and multiply-accumulate operations* on page D7-2719 for information on counts for dot product, matrix multiplication, and BFloat16 multiply-accumulate instructions.

`0x80CB`, **LDST_FIXED_OPS_SPEC, Non-scalable load and store element Operations speculatively executed**

The counter counts speculatively executed memory read and write operations as follows:

- Loading or storing a single scalar register increments the counter by 1.
- Loading or storing a pair of scalar registers increments the counter by 2.
- An atomic store instruction increments the counter by 1.
- An atomic load instruction increments the counter by 2.
- SVE and Advanced SIMD `LD1R` instructions increment the counter by 1.
- SVE `LD1RQ` instructions increment the counter by (128 ÷ CSIZE).
- Advanced SIMD `LD[1-4]` and `ST[1-4]` instructions increment the counter by the number of elements transferred per vector multiplied by the number of transferred registers.

See *Operation counts for dot-product and multiply-accumulate operations* on page D7-2719 for information on counts for dot product, matrix multiplication, and BFloat16 multiply-accumulate instructions.

`0x80CC`, **LD_SCALE_OPS_SPEC, Scalable load element Operations speculatively executed**

The counter counts speculatively executed memory read operations, due to SVE predicated vector load instructions, excluding the replicating `LD1R` and `LD1RQ` instructions. For each instruction, the counter is incremented by (128 ÷ CSIZE), multiplied by the number of transferred registers. See *Operation counts for dot-product and multiply-accumulate operations* on page D7-2719 for information on counts for dot product, matrix multiplication, and BFloat16 multiply-accumulate instructions.

`0x80CD`, **LD_FIXED_OPS_SPEC, Non-scalable load element Operations speculatively executed**

The counter counts speculatively executed memory read operations as follows:

- Loading a single scalar register increments the counter by 1.
- Loading a pair of scalar registers increments the counter by 2.
- An atomic load instruction increments the counter by 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- SVE and Advanced SIMD `LD1R` instructions increment the counter by 1.
- SVE `LD1RQ` instructions increment the counter by (128 ÷ CSIZE).
- Advanced SIMD `LD[1-4]` instructions increment the counter by the number of elements transferred per vector multiplied by the number of transferred registers.

See *Operation counts for dot-product and multiply-accumulate operations* on page D7-2719 for information on counts for dot product, matrix multiplication, and BFloat16 multiply-accumulate instructions.

`0x80CE`, ST_SCALE_OPS_SPEC, Scalable store element Operations speculatively executed

The counter counts speculatively executed memory write operations, due to SVE predicated vector store instructions. For each instruction, the counter is incremented by (128 ÷ CSIZE), multiplied by the number of transferred registers. See *Operation counts for dot-product and multiply-accumulate operations* on page D7-2719 for information on counts for dot product, matrix multiplication, and BFloat16 multiply-accumulate instructions.

`0x80CF`, ST_FIXED_OPS_SPEC, Non-scalable store element Operations speculatively executed

The counter counts speculatively executed memory write operations as follows:

- Storing a single scalar register increments the counter by 1.
- Storing a pair of scalar registers increments the counter by 2.
- An atomic store instruction increments the counter by 1.
- Advanced SIMD `ST[1-4]` instructions increment the counter by the number of elements transferred per vector multiplied by the number of transferred registers.

See *Operation counts for dot-product and multiply-accumulate operations* on page D7-2719 for information on counts for dot product, matrix multiplication, and BFloat16 multiply-accumulate instructions.

`0x80DA`, LDST_SCALE_BYTES_SPEC, Scalable load and store bytes, Speculatively executed

The counter counts bytes speculatively read or written due to SVE vector load and store instructions, excluding the replicating `LD1R` and `LD1RQ` instructions. For each instruction, the counter is incremented by (16 ÷ (CSIZE ÷ MSIZE)), multiplied by the number of transferred vector registers.

`0x80DB`, LDST_FIXED_BYTES_SPEC, Non-scalable load and store bytes, Speculatively executed

The counter counts bytes speculatively read and written as follows:

- Non-SVE `LDR` and `STR` instructions increment the counter by (MSIZE ÷ 8).
- `LDP`, `LDNP`, `STP`, and `STNP` instructions increment the counter by 2 × (MSIZE ÷ 8).
- Atomic store instructions increment the counter by (MSIZE ÷ 8).
- Atomic load instructions increment the counter by 2 × (MSIZE ÷ 8).
- SVE and Advanced SIMD `LD1R` instructions increment the counter by (MSIZE ÷ 8).
- SVE `LD1RQ` instructions increment the counter by 16.
- Advanced SIMD `LD[1-4]` and `ST[1-4]` instructions increment the counter by the number of bytes being transferred per register multiplied by the number of registers transferred.

`0x80DC`, LD_SCALE_BYTES_SPEC, Scalable load bytes, Speculatively executed

The counter counts bytes speculatively read due to SVE vector load instructions, excluding the replicating `LD1R` and `LD1RQ` instructions. For each instruction, the counter is incremented by (16 ÷ (CSIZE ÷ MSIZE)), multiplied by the number of transferred vector registers.

`0x80DD`, LD_FIXED_BYTES_SPEC, Non-scalable load bytes, Speculatively executed

The counter counts bytes speculatively read as follows:

- Non-SVE `LDR` instructions increment the counter by (MSIZE ÷ 8).
- `LDP` and `LDNP` instructions increment the counter by (2 × (MSIZE ÷ 8)).
- Atomic load instructions increment the counter by (MSIZE ÷ 8).
- SVE and Advanced SIMD `LD1R` instructions increment the counter by (MSIZE ÷ 8).
- SVE `LD1RQ` instructions increment the counter by 16.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Advanced SIMD `LD[1-4]` instructions increment the counter by the number of bytes read per register multiplied by the number of registers transferred.

`0x80DE`, **ST_SCALE_BYTES_SPEC, Scalable store bytes, Speculatively executed**

The counter counts bytes speculatively written due to SVE vector store instructions. For each instruction, the counter is incremented by (16 ÷ (CSIZE ÷ MSIZE)), multiplied by the number of transferred vector registers.

`0x80DF`, **ST_FIXED_BYTES_SPEC, Non-scalable store bytes, Speculatively executed**

The counter counts bytes speculatively written as follows:

- Non-SVE `STR` instructions increment the counter by (MSIZE ÷ 8).
- `STP` and `STNP` instructions increment the counter by 2 × (MSIZE ÷ 8).
- Atomic load and store instructions increment the counter by (MSIZE ÷ 8).
- Advanced SIMD `ST[1-4]` instructions increment the counter by the number of bytes read per register multiplied by the number of registers transferred.

`0x80E1`, **ASE_INT8_SPEC, Advanced SIMD 8-bit integer Operations speculatively executed**

The counter counts each operation counted by ASE_SVE_INT8_SPEC where the operation is an Advanced SIMD operation.

`0x80E2`, **SVE_INT8_SPEC, SVE 8-bit integer Operations speculatively executed**

The counter counts each operation counted by ASE_SVE_INT8_SPEC where the operation is an SVE operation.

`0x80E3`, **ASE_SVE_INT8_SPEC, Advanced SIMD and SVE 8-bit integer Operations speculatively executed**

The counter counts each operation counted by ASE_SVE_INT_SPEC where the largest type is an 8-bit integer.

`0x80E5`, **ASE_INT16_SPEC, Advanced SIMD 16-bit integer Operations speculatively executed**

The counter counts each operation counted by ASE_SVE_INT16_SPEC where the operation is an Advanced SIMD operation.

`0x80E6`, **SVE_INT16_SPEC, SVE 16-bit integer Operations speculatively executed**

The counter counts each operation counted by ASE_SVE_INT16_SPEC where the operation is an SVE operation.

`0x80E7`, **ASE_SVE_INT16_SPEC, Advanced SIMD and SVE 16-bit integer Operations speculatively executed**

The counter counts each operation counted by ASE_SVE_INT_SPEC where the largest type is an 16-bit integer.

`0x80E9`, **ASE_INT32_SPEC, Advanced SIMD 32-bit integer Operations speculatively executed**

The counter counts each operation counted by ASE_SVE_INT32_SPEC where the operation is an Advanced SIMD operation.

`0x80EA`, **SVE_INT32_SPEC, SVE 32-bit integer Operations speculatively executed**

The counter counts each operation counted by ASE_SVE_INT32_SPEC where the operation is an SVE operation.

`0x80EB`, **ASE_SVE_INT32_SPEC, Advanced SIMD and SVE 32-bit integer Operations speculatively executed**

The counter counts each operation counted by ASE_SVE_INT_SPEC where the largest type is an 32-bit integer.

`0x80ED`, **ASE_INT64_SPEC, Advanced SIMD 64-bit integer Operations speculatively executed**

The counter counts each operation counted by ASE_SVE_INT64_SPEC where the operation is an Advanced SIMD operation.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0x80EE`, **SVE_INT64_SPEC, SVE 64-bit integer Operations speculatively executed**

The counter counts each operation counted by ASE_SVE_INT64_SPEC where the operation is an SVE operation.

`0x80EF`, **ASE_SVE_INT64_SPEC, Advanced SIMD and SVE 64-bit integer Operations speculatively executed**

The counter counts each operation counted by ASE_SVE_INT_SPEC where the largest type is an 64-bit integer.

`0x80F1`, **ASE_FP_DOT_SPEC, Advanced SIMD floating-point dot-product Operations speculatively executed**

The counter counts each operation counted by ASE_SVE_FP_DOT_SPEC where the operation is an Advanced SIMD dot-product operation.

`0x80F2`, **SVE_FP_DOT_SPEC, SVE floating-point dot-product Operations speculatively executed**

The counter counts each operation counted by ASE_SVE_FP_DOT_SPEC where the operation is an SVE dot-product operation.

`0x80F3`, **ASE_SVE_FP_DOT_SPEC, Advanced SIMD and SVE floating-point dot-product Operations speculatively executed**

The counter counts each operation counted by FP_SPEC where the operation is an Advanced SIMD or SVE dot-product operation.

`0x80F5`, **ASE_FP_MMLA_SPEC, Advanced SIMD floating-point matrix multiply Operations speculatively executed**

The counter counts each operation counted by ASE_SVE_FP_MMLA_SPEC where the operation is an Advanced SIMD matrix multiply operation.

`0x80F6`, **SVE_FP_MMLA_SPEC, SVE floating-point matrix multiply Operations speculatively executed**

The counter counts each operation counted by ASE_SVE_FP_MMLA_SPEC where the operation is an SVE matrix multiply operation.

`0x80F7`, **ASE_SVE_FP_MMLA_SPEC, Advanced SIMD and SVE floating-point matrix multiply Operations speculatively executed**

The counter counts each operation counted by FP_SPEC where the operation is an Advanced SIMD or SVE matrix multiply operation.

`0x80F9`, **ASE_INT_DOT_SPEC, Advanced SIMD integer dot-product Operations speculatively executed**

The counter counts each operation counted by ASE_SVE_INT_DOT_SPEC where the operation is an Advanced SIMD dot-product operation.

`0x80FA`, **SVE_INT_DOT_SPEC, SVE integer dot-product Operations speculatively executed**

The counter counts each operation counted by ASE_SVE_INT_DOT_SPEC where the operation is an SVE dot-product operation.

`0x80FB`, **ASE_SVE_INT_DOT_SPEC, Advanced SIMD and SVE integer dot-product Operations speculatively executed**

The counter counts each operation counted by INT_SPEC where the operation is an Advanced SIMD or SVE dot-product operation.

`0x80FD`, **ASE_INT_MMLA_SPEC, Advanced SIMD integer matrix multiply Operations speculatively executed**

The counter counts each operation counted by ASE_SVE_INT_MMLA_SPEC where the operation is an Advanced SIMD matrix multiply operation.

`0x80FE`, **SVE_INT_MMLA_SPEC, SVE integer matrix multiply Operations speculatively executed**

The counter counts each operation counted by ASE_SVE_INT_MMLA_SPEC where the operation is an SVE matrix multiply operation.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

0x80FF, ASE_SVE_INT_MMLA_SPEC, Advanced SIMD and SVE integer matrix multiply Operations speculatively executed

The counter counts each operation counted by INT_SPEC where the operation is an Advanced SIMD or SVE matrix multiply operation.

0x8110, BR_IMMED_PRED_RETIRED, Instruction architecturally executed, predicted immediate branch

The counter counts the instructions on the architecturally executed path counted by both BR_IMMED_RETIRED and BR_PRED_RETIRED. These are all immediate branch instructions where the branch was correctly predicted.

0x8111, BR_IMMED_MIS_PRED_RETIRED, Instruction architecturally executed, mispredicted immediate branch

The counter counts the instructions on the architecturally executed path, counted by both BR_IMMED_RETIRED and BR_MIS_PRED_RETIRED. These are all immediate branch instructions where the branch was mispredicted.

0x8112, BR_IND_PRED_RETIRED, Instruction architecturally executed, predicted indirect branch

The counter counts the instructions on the architecturally executed path counted by both BR_IND_RETIRED and BR_PRED_RETIRED. These are branch instructions where the branch was correctly predicted, but does not include immediate instructions.

0x8113, BR_IND_MIS_PRED_RETIRED, Instruction architecturally executed, mispredicted indirect branch

The counter counts the instructions on the architecturally executed path counted by both BR_IND_RETIRED and BR_MIS_PRED_RETIRED. These are branch instructions where the branch was mispredicted, but does not include immediate instructions.

0x8114, BR_RETURN_PRED_RETIRED, Instruction architecturally executed, predicted procedure return

The counter counts the instructions on the architecturally executed path counted by BR_IND_PRED_RETIRED where, if taken, the branch would be counted by BR_RETURN_RETIRED. These are branch return instructions, where the branch was correctly predicted.

0x8115, BR_RETURN_MIS_PRED_RETIRED, Instruction architecturally executed, mispredicted procedure return

The counter counts the instructions on the architecturally executed path counted by BR_IND_MIS_PRED_RETIRED where, if taken, the branch would also be counted by BR_RETURN_RETIRED. These are branch return instructions where the branch was mispredicted.

0x8116, BR_INDNR_PRED_RETIRED, Instruction architecturally executed, predicted indirect branch, excluding return

The counter counts the instructions on the architecturally executed path counted by BR_IND_PRED_RETIRED where, if taken, the branch would not be counted by BR_RETURN_RETIRED. These are branch instructions where the branch was correctly predicted, but does not include immediate or return instructions.

0x8117, BR_INDNR_MIS_PRED_RETIRED, Instruction architecturally executed, mispredicted indirect branch, excluding return

The counter counts the instructions on the architecturally executed path counted by BR_IND_MIS_PRED_RETIRED where, if taken, the branch would not be counted by BR_RETURN_RETIRED. These are branch instructions where the branch was mispredicted, but does not include immediate or return instructions.

0x8118, BR_TAKEN_PRED_RETIRED, Instruction architecturally executed, predicted branch, taken

The counter counts the instructions on the architecturally executed path counted by both PC_WRITE_RETIRED and BR_PRED_RETIRED. These are branch instructions, where the branch was correctly predicted and taken.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0x8119`, **BR_TAKEN_MIS_PRED_RETIRED, Instruction architecturally executed, mispredicted branch, taken**

The counter counts the instructions on the architecturally executed path counted by both PC_WRITE_RETIRED and BR_MIS_PRED_RETIRED. These are branch instructions where the branch was mispredicted and taken.

`0x811A`, **BR_SKIP_PRED_RETIRED, Instruction architecturally executed, predicted branch, not taken**

The counter counts the instructions on the architecturally executed path counted by both BR_SKIP_RETIRED and BR_PRED_RETIRED. These are branch instructions, where the branch was correctly predicted and not taken.

`0x811B`, **BR_SKIP_MIS_PRED_RETIRED, Instruction architecturally executed, mispredicted branch, not taken**

The counter counts the instructions on the architecturally executed path counted by both BR_SKIP_RETIRED and BR_MIS_PRED_RETIRED. These are branch instructions where the branch was mispredicted and not taken.

`0x811C`, **BR_PRED_RETIRED, Instruction architecturally executed, predicted branch**

The counter counts the instructions on the architecturally executed path counted by BR_RETIRED that are not counted by BR_MIS_PRED_RETIRED. These are branch instructions, where the branch was correctly predicted.

D7.11.4 Cycle event counting on multithreaded implementations

For most events, the event is only counted when it is Attributable to the counting PE or thread, see *Attributability* on page D7-2707.

Multithreaded implementations can have various forms, some examples of these are:

- *Simultaneous Multithreading* (SMT), where every PE thread is active on every cycle.
- *Fine-grained Multithreading* (FGMT), also known as a Barrel processor, where one PE thread is active on each cycle, and this changes regularly.
- *Switch on Event Multithreading* (SoEMT), also known as *Coarse-grained Multithreading* (CGMT), where high latency events cause the processor to switch the active PE thread.

In the above examples, active means that the PE might execute the instructions. A PE can be active but not executing instructions when no instruction is available or because of limited execution resources.

When the PMU implementation supports multithreading, and the *Effective value* of PMEVTYPER<n>_EL0.MT bit is 0, the CPU_CYCLES event only counts cycles on which the thread was active. For the example multithreaded implementations, this means that:

- For an SMT implementation, the CPU_CYCLES event counts every cycle.
- For a particular FGMT implementation, that alternates between two threads on each cycle, the CPU_CYCLES event counts every other cycle.
- For a particular SoEMT implementation, that is waiting for a long latency operation, the CPU_CYCLES event does not count cycles, as the PE thread is not active.

If the *Effective value* of PMEVTYPER<n>_EL0.MT bit is 1, the CPU_CYCLES event counts each processor cycle, and can only count a maximum of one cycle each cycle.

In addition, the STALL_FRONTEND and STALL_BACKEND events only count cycles that are counted by the CPU_CYCLES event, and so have the same limitation. For example, in an SMT implementation, if a PE thread cannot issue an instruction because of contention with other PE threads, these are counted as STALL_BACKEND cycles.

If the *Effective value* of PMEVTYPER<n>_EL0.MT bit is 1, the PE only counts cycles on which no operation is issued from any thread.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

The PMCCNTR register counts every processor cycle.

See *Multithreaded implementations* on page D7-2710, MDCR_EL3.MTPME, SDCR.MTPME, MDCR_EL2.MTPME, and HDCR.MTPME for more information about when the *Effective value* of PMEVTYPER<n>_EL0 is 0.

D7.11.5 Meaningful ratios between common microarchitectural events

The architecture highlights some meaningful ratios that can be derived from the common microarchitectural events. Table D7-8 lists the highlighted ratios.

Table D7-8 REFILL events and associated access events

Numerator	Denominator	Ratio
0x0001 L1I_CACHE_REFILL	0x0014 L1I_CACHE	Attributable Level 1 instruction cache refill rate
0x0002 L1I_TLB_REFILL	0x0026 L1I_TLB	Attributable Level 1 instruction TLB refill rate
0x0003 L1D_CACHE_REFILL	0x0004 L1D_CACHE	Attributable Level 1 data or unified cache refill rate
0x0005 L1D_TLB_REFILL	0x0025 L1D_TLB	Attributable Level 1 data or unified TLB refill rate
0x0017 L2D_CACHE_REFILL	0x0016 L2D_CACHE	Attributable Level 2 data or unified cache refill rate
0x0028 L2I_CACHE_REFILL	0x0027 L2I_CACHE	Attributable Level 2 instruction cache refill rate
0x002A L3D_CACHE_REFILL	0x002B L3D_CACHE	Attributable Level 3 data or unified cache refill rate
0x002D L2D_TLB_REFILL	0x002F L2D_TLB	Attributable Level 2 data or unified TLB refill rate
0x002E L2I_TLB_REFILL	0x0030 L2I_TLB	Attributable Level 2 instruction TLB refill rate
0x0019 BUS_ACCESS	0x001D BUS_CYCLES	Attributable Bus accesses per cycle
0x0033 LL_CACHE_MISS	0x0032 LL_CACHE	Attributable Last Level data or unified cache refill rate
0x0034 DTLB_WALK	0x0025 L1D_TLB	Attributable data TLB miss rate
0x0035 ITLB_WALK	0x0026 L1I_TLB	Attributable instruction TLB miss rate
0x0037 LL_CACHE_MISS_RD	0x0036 LL_CACHE_RD	Attributable memory read operation miss rate
0x0038 REMOTE_ACCESS_RD	0x0031 REMOTE_ACCESS	Attributable read accesses to another socket in a multi-socket system

D7.11.6 Required events

PMUv3 requires that an implementation includes the following common events:

- 0x0000, SW_INCR, Instruction architecturally executed, Condition code check pass, software increment.
- 0x0003, L1D_CACHE_REFILL, Level 1 data cache refill.

  Note

  Event 0x0003 is only required if the implementation includes a Level 1 data or unified cache.

- 0x0004, L1D_CACHE, Level 1 data cache access.

  Note

  Event 0x0004 is only required if the implementation includes a Level 1 data or unified cache.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- `0x0010`, BR_MIS_PRED, Mispredicted or not predicted branch Speculatively executed.

  Note

  Event `0x0010` is only required if the implementation includes program-flow prediction. However, Arm strongly recommends that the event is implemented as described in *Common microarchitectural events* on page D7-2730.

- `0x0011`, CPU_CYCLES, Cycle.
- `0x0012`, BR_PRED, Predictable branch Speculatively executed.

  Note

  Event `0x0012` is only required if the implementation includes program-flow prediction. However, Arm recommends that the event is implemented as described in *Common microarchitectural events* on page D7-2730.

- At least one of:
  - `0x0008`, INST_RETIRED, Instruction architecturally executed.
  - `0x001B`, INST_SPEC, Operation Speculatively executed.

  Note

  Arm strongly recommends that event `0x008` is implemented.

- When ARMv8.1-PMU is implemented:
  - `0x0023`, STALL_FRONTEND, No operation issued due to the frontend.
  - `0x0024`, STALL_BACKEND, No operation issued due to the backend.
- When The Scalable Vector Extension (SVE) is implemented, at least one of:
  - `0x8002`, SVE_INST_RETIRED, SVE instruction architecturally retired.
  - `0x8006`, SVE_INST_SPEC, SVE operation speculatively executed.
- When the Statistical Profiling Extension is implemented:
  - `0x4000`, SAMPLE_POP, Sample Population.
  - `0x4001`, SAMPLE_FEED, Sample Taken.
  - `0x4002`, SAMPLE_FILTRATE, Sample Filtered.
  - `0x4003`, SAMPLE_COLLISION, Sample Collision.
- When ARMv8.4-PMU is implemented:
  - `0x003C`, STALL, No operation sent for execution.
  - `0x0039`, L1D_CACHE_LMISS_RD, Level 1 data cache long-latency read miss.
  - `0x4006`, L1I_CACHE_LMISS, Level 1 instruction cache long-latency miss.
  - `0x0040`, L1D_CACHE_RD, Level 1 data cache read.

When any of the following common events are implemented, all three of them are implemented:

- `0x003D`, STALL_SLOT_BACKEND, No operation sent for execution on a Slot due to the backend,
- `0x003E`, STALL_SLOT_FRONTEND, No operation sent for execution on a Slot due to the frontend.
- `0x003F`, STALL_SLOT, No operation sent for execution on a Slot.

Arm strongly recommends that the following events are implemented:

- `0x0021`, BR_RETIRED.
- `0x0022`, BR_MIS_PRED_RETIRED.
- `0x003A`, OP_RETIRED.
- `0x003B`, OP_SPEC.
- `0x003D`, STALL_SLOT_BACKEND.
- `0x003E`, STALL_SLOT_FRONTEND.
- `0x003F`, STALL_SLOT.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D7.11.7 IMPLEMENTATION DEFINED event numbers

Arm recommends that implementers establish a standardized numbering scheme for their IMPLEMENTATION DEFINED events, with common definitions, and common event numbers, applied to all of their implementations. In general, the recommended approach is for standardization across implementations with common features. However, Arm recognizes that attempting to standardize the encoding of microarchitectural features across too wide a range of implementations is not productive.

Arm strongly recommends that at least the following classes of event are identified in the IMPLEMENTATION DEFINED events:

- Separating each of the STALL_FRONTEND and STALL_SLOT_FRONTEND events to count holes in instruction availability.
- Separating each of the STALL_BACKEND and STALL_SLOT_BACKEND events, to count, for example, cumulative duration of stalls, unavailability of execution resources, or missed superscalar issue opportunities.
- Miss rates for additional levels of caches and TLBs.
- Any external events passed to the PE through an IMPLEMENTATION DEFINED mechanism.
- Cumulative duration of a PSTATE.{A, I, F} interrupt mask set to 1.
- Cumulative occupancy for resource queues, such as data access queues, and entry/exit counts, so that average latencies can be determined, separating out counts for key resources that might exist. An implementation might also provide registers in the IMPLEMENTATION DEFINED space to further extend such counts, for example by specifying a minimum latency for an event to be counted.
- Any other microarchitectural features that the implementer considers are valuable to count.

The range of possible IMPLEMENTATION DEFINED event numbers is described in *The PMU event number space and common events* on page D7-2721. Appendix K3 *Recommendations for Performance Monitors Event Numbers for IMPLEMENTATION DEFINED Events* lists the Arm recommended standardized numbering scheme for these events.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D7.12 Performance Monitors Extension registers

Further information on the Performance Monitors Extension Registers can be found in the following sections:

- Table K14-2 on page K14-8040 lists the Performance Monitors register names for AArch32 and AArch64 states.
- *Performance monitors registers* on page K14-8061 summarizes the Performance Monitors Extension registers in AArch64 state.
- *Performance monitors registers* on page K14-8086 summarizes the Performance Monitors Extension registers in AArch32 state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter D8
The Activity Monitors Extension

This chapter describes the Armv8 implementation of version 1 of the *Activity Monitor Unit* (AMU) architecture, AMUv1, an optional non-invasive component. It contains the following sections:

- *About the Activity Monitors Extension* on page D8-2774.
- *Properties and behavior of the activity monitors* on page D8-2775.
- *AMU events and event numbers* on page D8-2777.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D8.1 About the Activity Monitors Extension

The Activity Monitors Extension is an optional extension to the Armv8.4 architecture.

The Activity Monitors Extension implements version 1 of the Activity Monitors architecture, AMUv1, and interfaces to the registers defined by AMUv1, the Activity Monitors registers.

Version 1 of the Activity Monitors architecture implements:

- A counter group of four architected 64-bit event counters. The events counted by the architected event counter are fixed and architecturally defined.

  Note

  The Activity Monitors architecture provides space for up to 16 architected event counters. Future versions of the Activity Monitors architecture may use this space to implement additional architected event counters.

- A counter group of up to 16 auxiliary 64-bit event counters. The event counted for each auxiliary event counter may be fixed or programmable, and whether it is fixed or programmable is IMPLEMENTATION DEFINED. When the event counted by an auxiliary event counter is fixed, this event is IMPLEMENTATION DEFINED.
- Controls for enabling and disabling counters.
- When the event counted by an auxiliary event counter is programmable, controls for assigning an event to the counter.
- Controls that determine whether the activity monitor counters continue to count while the PE is halted in Debug state.

The read-only registers AMCFGR and AMCGCR provide information about features supported by the Activity Monitors Extension, the number of counter groups implemented, the total number of counters implemented, the number of counters implemented within each group, and the size of the counters.

The Activity Monitors Extension provides:

- A mandatory System register interface to the Activity Monitors registers, for both AArch64 and AArch32 states.

  Base system registers on page K14-8069 lists the AArch64 Activity Monitors registers, and *Base system registers* on page K14-8093 lists the AArch32 Activity Monitors registers. Table K14-3 on page K14-8041 shows the relationship between the AArch64 and the AArch32 Activity Monitors register.

- Controls that allow software to enable or disable access by software running at lower Exception levels to the Activity Monitors registers.
- When ARMv8.6-AMU is implemented, and the hypervisor is using AArch64, offset registers that support virtualization of the Activity Monitor event counters.
- An optional external interface providing read-only memory-mapped access to the Activity Monitors registers.

  Alphabetical index of memory-mapped registers on page K14-8096 lists the Activity Monitors memory-mapped registers. For more information on the recommended external interface, see Chapter I4 *Recommended External Interface to the Activity Monitors*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D8.2 Properties and behavior of the activity monitors

D8.2.1 Basic characteristics of the activity monitor event counters

Every activity monitor event counter is a 64-bit wrapping counter. When an activity monitor event counter wraps, the counter overflows.

Note

The Activity Monitor architecture does not provide support for overflow status indication or interrupts.

The state of the authentication signals do not affect counting.

Any change in clock frequency, including when a WFI and WFE instruction stops the clock, can affect any counter.

If ARMv8.6-AMU is implemented, for the architected event counters 0, 2 and 3, and each auxiliary event counter configured to use an offset, there is an offset register which is used to virtualize the count on a read from EL1 or EL0. At EL2, EL3 or from the memory-mapped view, permitted accesses to the counters use the physical view without any offset. See *Virtualization* on page D8-2776

D8.2.2 Counter configuration and controls

For each architected event counter AMEVCNTR0<n>, there is a corresponding event type register AMEVTYPER0<n> which provides information on the event counted by that counter. The event type registers AMEVTYPER0<n> are read-only.

For each auxiliary event counter AMEVCNTR1<n>, there is a corresponding event type register AMEVTYPER1<n> which provides information on the event counted by that counter. When the event counted by an auxiliary event counter is fixed, the corresponding event type register AMEVTYPER1<n> is read-only. When the event counted by an auxiliary event counter is programmable, the corresponding event type register AMEVTYPER1<n> is read/write.

For each counter group, there is a pair of separate controls to enable and disable the counters in that counter group. AMCNTENCLR0 and AMCNTENSET0 are used to disable and enable the architected event counters. AMCNTENCLR1 and AMCNTENSET1 are used to disable and enable the auxiliary event counters.

While the PE is halted in Debug state, AMCR.HDBG controls whether activity monitor counting is halted.

AMUSERENR.EN controls access from EL0 to the Activity Monitor Extension System registers. CPTR_EL2.TAM and HCPTR.TAM control access from EL0 and EL1 to the Activity Monitor Extension System registers. CPTR_EL3.TAM control access from EL0, EL1, and EL2 to the Activity Monitor Extension System registers.

Note

These controls obey the priority order described in *Synchronous exception prioritization for exceptions taken to AArch64 state* on page D1-2355 and *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-5707.

AMUSERENR.EN is configurable at EL1, EL2, and EL3. All other controls, as well as the value of the counters, are configurable only at the highest implemented Exception level.

If ARMv8.6-AMU is implemented, AMCG1IDR_EL0 defines which auxiliary counters are implemented, and if virtual offsets are enabled, indicates which of the implemented counters have a virtual offset when read from EL0 and EL1.

If ARMv8.6-AMU is implemented, AMCR.CG1RZ controls whether the auxiliary event counters read as zero if they are accessed at an Exception level lower than the highest implemented Exception level.

D8.2.3 Power and reset domains

The power domain of the activity monitoring unit is IMPLEMENTATION DEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The reset domain of the activity monitoring unit is IMPLEMENTATION DEFINED.

When a Cold reset of the power domain of the activity monitoring unit occurs, the activity monitoring unit is reset and the counters are reset to zero. When the PE is not in reset, the activity monitoring unit is available.

D8.2.4 Accuracy and non-invasive behavior

The activity monitors are a non-invasive component which must provide broadly accurate and statistically useful count information.

The implementation of an architecturally required event might create a conflict between the requirement to be non-invasive and the requirement to present an accurate value of the count under normal operating conditions. An implementation might provide an IMPLEMENTATION DEFINED control that disables accurate count of the event to restore performance and document the impact on performance of accurate counting. The expectations for non-invasive behavior and the degree of inaccuracy of the activity monitors are otherwise as described for the Performance Monitors architecture.

Note

For information on the expectations for non-invasive behavior and the degree of inaccuracy of the Performance Monitors, see *Non-invasive behavior* on page D7-2703 and *A reasonable degree of inaccuracy* on page D7-2703.

D8.2.5 Virtualization

ARMv8.6-AMU supports virtualized access to the Activity Monitors event counters at EL1 and EL0.

The fields HCR_EL2.AMVOFFEN and SCR_EL3.AMVOFFEN enable and disable virtualization. When enabled, the architected event counters 0, 2 and 3 have counter offsets. Architected event counter 1 does not have an offset. The register AMCG1IDR_EL0 indicates which of the implemented auxiliary event counters has implemented counter offsets. An implemented event counter that does not have a defined offset has an effective offset of zero. The offset registers can be accessed only at EL2 or EL3, and affect views of the event counters at EL1 and EL0 from the System register interfaces only.

The AMEVCNTVOFF0<n>_EL2 registers hold the offsets for the implemented and enabled architected event counters.

The AMEVCNTVOFF1<n>_EL2 registers hold the offsets for the implemented and enabled auxiliary event counters.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D8.3 AMU events and event numbers

The Activity Monitors architecture uses the event number space defined by the Performance Monitors architecture to identify events.

The Activity Monitors architecture defines additional events and adds them to the event number space defined by the Performance Monitors architecture for common events.

If the event is counting an IMPLEMENTATION DEFINED event, it must use an event number from the IMPLEMENTATION DEFINED event space.

When an implementation supports monitoring of an event that is assigned a common architectural or microarchitectural event number, Arm strongly recommends that it uses that number for the event.

When a common event is available to both the Performance Monitors architecture and the Activity Monitors architecture within one implementation, both architectures use the same event number.

D8.3.1 Architected event counters

Version 1 of the Activity Monitors architecture, AMUv1, requires four events to be counted by the architected activity monitor event counters.

The events required to be counted are:

`0x0011`**, CPU_CYCLES, Processor frequency cycles**

This event is defined identically to CPU_CYCLES in the PMUv3 architecture.

When the PE is in WFI or WFE, this counter does not increment. When in a multithreaded implementation, regardless of which PE is currently active, this counter continues to count for all PEs not in WFI or WFE.

This event is counted by AMEVCNTR0<n>, where n is 0.

`0x4004`**, CNT_CYCLES, Constant frequency cycles**

The constant frequency cycles counter increments at a constant frequency equal to the rate of increment of the System counter, CNTPCT_EL0.

When the PE is in WFI or WFE, this counter does not increment. When in a multithreaded implementation, regardless of which PE is currently active, this counter continues to count for all PEs not in WFI or WFE.

This event is counted by AMEVCNTR0<n>, where n is 1.

`0x0008`**, INST_RETIRED, Instructions retired**

This event is defined identically to INST_RETIRED in the PMUv3 architecture.

This event is counted by AMEVCNTR0<n>, where n is 2.

`0x4005`**, STALL_BACKEND_MEM, Memory stall cycles**

The counter counts cycles in which the PE is unable to dispatch instructions from the frontend to the backend of the PE due to a backend stall caused by a miss in the last level of cache within the PE clock domain.

This event is counted by AMEVCNTR0<n>, where n is 3.

D8.3.2 Auxiliary event counters

Auxiliary event counters can count events defined by the Performance Monitors architecture and IMPLEMENTATION DEFINED events defined specifically for activity monitoring.

Implementations must not re-use an IMPLEMENTATION DEFINED event number for different hardware events across the Performance Monitors architecture and the Activity Monitors architecture.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter D9
The Statistical Profiling Extension

This chapter describes the Statistical Profiling Extension. It contains the following sections:

- *About the Statistical Profiling Extension* on page D9-2780.
- *Defining the sample population* on page D9-2782.
- *Controlling when an operation is sampled* on page D9-2784.
- *Enabling profiling* on page D9-2787.
- *Filtering sample records* on page D9-2789.
- *The profiling data* on page D9-2790.
- *The Profiling Buffer* on page D9-2799.
- *Profiling Buffer management* on page D9-2803.
- *Synchronization and Statistical Profiling* on page D9-2807.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D9.1 About the Statistical Profiling Extension

Statistical profiling is a four stage process.

1. An operation is chosen from a sample population, that can be restricted by Exception level, at a programmable interval that might have some random, or pseudorandom, perturbation.
2. A trace of the sampled operation is taken. This includes the PC, events, timings, and data addresses, related to the sampled operation. This is the profiling operation.
3. Before a sample record is created, it is possible to filter out potential sample records generated by the profiling operation by reference to any or all of the following:
   a. The type of operation.
   b. Events.
   c. Latency.
4. A sample record is created that contains the traced information. Sample records that meet the criteria of the filter are written to and stored in a memory buffer. These sample records can be processed by software when the memory buffer is full.

D9.1.1 Non-invasive behavior

Statistical Profiling is a non-invasive debug operation:

- While profiling is enabled, the operation and performance of the processing element (PE) must not be significantly impacted between sampled operations, that is, other than for writing out sample records and processing Profiling Buffer management interrupts.
- The performance of the sampled operation and the performance of the PE in general must not be significantly impacted. The sample records are not written to memory until after the sampled operation has completed. However, this does not apply if the sample records are physical addresses for data access operations. In this case, the impact is IMPLEMENTATION DEFINED.
- The profiling operation to write sample records must not be excessively impactful on the performance of the sampled operation or the performance of the PE generally.

D9.1.2 PMU extensions

If the Statistical Profiling and Performance Monitors Extensions are implemented, then the following PMU events must be implemented:

- SAMPLE_POP.
- SAMPLE_FEED.
- SAMPLE_FILTRATE.
- SAMPLE_COLLISION.

Note

These events are discoverable through a read of PMCEID0_EL0[35:32].

D9.1.3 Multithreaded implementations

In a multithreaded implementation:

- Statistical Profiling is implemented per-thread.
- The sample interval counter counts only operations for the thread that is being profiled.
- Latency and other cycle counters count each cycle for the PE for which the thread was active and could issue an operation.

The architecture does not define features for inter-thread profiling and does not support sharing the Profiling Buffer between threads.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

An implementation is described as multithreaded when the lowest level of affinity consists of logical processors that are implemented using a multi-threading type approach. That is, the performance of processors at the lowest affinity level is very interdependent.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D9.2 Defining the sample population

All samples are taken from a *population of operations*. The population is dynamic rather than static. That is, if a program executes the same operation multiple times (for example, because of loops and subroutines) then that operation appears multiple times in the population.

The operations are an IMPLEMENTATION DEFINED choice between:

- Architecture instructions.
- IMPLEMENTATION DEFINED microarchitectural operations (micro-ops).

Architecture instruction means a single instruction that is defined by the Armv8 instruction set architecture in AArch64 state.

An architecture instruction might create one or more micro-ops at any point in the execution pipeline. The definition of a micro-op is implementation specific. An architecture instruction might create more than one micro-op for each instruction. A micro-op might also be removed or merged with another micro-op in the execution stream, so an architecture instruction might create no micro-ops for an instruction.

Any arbitrary translation of architecture instructions to an equivalent sequence of micro-ops is permitted. In some implementations, the relationship between architecture instructions and micro-ops might vary over time.

Note

Sampling from architecture instructions does not require that the instruction is architecturally executed.

D9.2.1 Operations that might be excluded from the sample population

It is IMPLEMENTATION DEFINED whether each of the following operations is part of the sample population:

- Operations on misspeculated paths.
- Operations (specifically micro-ops) that do not relate to any architecture instruction.
- Operations that generate non-architectural exceptions.

If the operation is not part of the sample population, the operation does not cause the sample interval counter to decrement, is not counted by the SAMPLE_POP event and therefore is never sampled.

If the operation is part of the sample population, the operation causes the sample interval counter to decrement, is counted by the SAMPLE_POP event, and might be sampled and counted by the SAMPLE_FEED event.

D9.2.2 Misspeculation and non-architectural operations

It is IMPLEMENTATION DEFINED whether the sample record for a sampled misspeculated operation or a sampled operation generating a non-architectural exception is written to the Profiling Buffer:

- If a sample record of a misspeculated operation or an operation generating a non-architectural exception is written to the Profiling Buffer, then neither event 0 (generated exception) nor event 1 (architecturally retired) are set in the record Events packet.
- If a sample record of a misspeculated operation or an operation generating a non-architectural exception is not captured into the Profiling Buffer, then no event packets are output and the sample is not counted by the SAMPLE_FILTRATE event.

Note

It is IMPLEMENTATION DEFINED whether such operations can be sampled. See *Operations that might be excluded from the sample population*.

If such an operation is not part of the sample population

The operation does not cause the sample counter to decrement, is not counted by the SAMPLE_POP event, and hence is never sampled.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If such an operation is part of the sample population

The operation causes the sample counter to decrement, is counted by the SAMPLE_POP event, and might be sampled and counted by the SAMPLE_FEED event.

It is IMPLEMENTATION DEFINED whether the sample record for such an operation, if sampled, is captured in the Profiling Buffer:

- If such a sample record is captured in the Profiling Buffer then some information for the operation might not be present. However, the Events packet and either the End packet or theTimestamp packet is always output. Neither event 0 (generated exception) nor event 1 (architecturally retired) will be set in the Events packet.
- If such a sample record is not captured into the Profiling Buffer then no packets are output and the sample is not counted by the SAMPLE_FILTRATE event.

If the sample record for an operation on a misspeculated path is captured into the Profiling Buffer, then the record must not contain information that cannot be accessed by privileged software of the owning Exception level.

Note

If the Owning Exception level passes this data to less privileged software for processing, it can set PMSFCR_EL1.FE to 1 and PMSEVFR_EL1[1] to 1 to prevent speculative instructions from being recorded in the Profiling Buffer.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D9.3 Controlling when an operation is sampled

The sample interval counter, PMSICR_EL1.COUNT controls when an operation is selected for sampling. In some implementations, a secondary sample interval counter, PMSICR_EL1.ECOUNT, is also used.

The following sections describe the operation of the sample interval counters.

Details of the random or pseudorandom number generator used when PMSIRR_EL1.RND is set to 1 are IMPLEMENTATION DEFINED. See *Generating random numbers for sampling*.

D9.3.1 Operation sampling

A sample operation is as follows:

1. A sampling interval is written to PMSICR_EL1.COUNT by software. The interval is measured in operations.
2. The sample interval counter is decremented by hardware for each operation when sampling is enabled.
3. When the sample interval counter reaches zero, then:
   a. If random perturbation is enabled, the PE continues to count for a random number of further operations while sampling is enabled.
   b. An operation is chosen for profiling.

      Note

      The choice of operation around the sampling point is arbitrary. The chosen operation might be the operation for which the sample interval counter reached 0, or the next operation. The choice must be applied consistently so as not to introduce sampling bias.

4. The sample interval counter is reloaded and the process loops to step 2. It is IMPLEMENTATION DEFINED whether the sample interval counter is reloaded before step 3.a) or at step 3.b). That is, before or after counting the random number of further operations.
5. The chosen operation is marked as the sampled operation. The PE collects information about the sampled operation as it executes by a profiling operation.
6. When the sampled operation is completed, the sample record is created.

D9.3.2 Generating random numbers for sampling

The random number generator is IMPLEMENTATION DEFINED. Implementations might use a pseudorandom number. The random number generator must be reset into a useable state. An implementation might include IMPLEMENTATION DEFINED registers to further configure the random number generator.

It is IMPLEMENTATION DEFINED whether the PE adds the random number to the sample interval counter prior to counting down the interval, or after the counter reaches zero and the counter has been reloaded.

D9.3.3 Initializing the one or more sample interval counters

When the PE moves from a state where profiling is disabled to a state where profiling is enabled:

- If PMSICR_EL1 is nonzero, then sampling restarts from the current values in PMSICR_EL1.
- If PMSICR_EL1 is zero, then it is loaded with an initial value. The behavior depends on PMSIRR_EL1.RND and an IMPLEMENTATION DEFINED choice discoverable by a read of PMSIDR_EL1.ERnd.
- If PMSIRR_EL1.RND is 0:
  — PMSICR_EL1.COUNT[31:8] is set to PMSIRR_EL1.INTERVAL.
  — PMSICR_EL1.COUNT[7:0] is set to `0x00`.
- If PMSIRR_EL1.RND is 1 and PMSIDR_EL1.ERnd is 0:
  — PMSICR_EL1.COUNT[31:8] is set to PMSIRR_EL1.INTERVAL.
  — PMSICR_EL1.COUNT[7:0] is set to a random or pseudorandom value in the range `0x00` to `0xFF`.
- If PMSIRR_EL1.RND is 1 and PMSIDR_EL1.ERnd is 1:
  — PMSICR_EL1.COUNT[[31:8] is set to PMSIRR_EL1.INTERVAL.
  — PMSICR_EL1.COUNT[7:0] is set to a random or pseudorandom value in the range `0x00` to `0xFF`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If PMSIRR_EL1.RND is 1 and PMSIDR_EL1.ERnd is 1:
  - PMSICR_EL1.COUNT[31:8] is set to PMSIRR_EL1.INTERVAL.
  - PMSICR_EL1.COUNT[7:0] is set to `0x00`.

D9.3.4 Behavior of the sample interval counter while profiling is enabled

While profiling is enabled, the counters control when an operation is selected for sampling. The behavior depends on PMSIRR_EL1.RND and an IMPLEMENTATION DEFINED choice discoverable in PMSIDR_EL1.ERnd.

If PMSIRR_EL1.RND is cleared to 0:

While nonzero, the sample interval counter decrements by 1 for each member of the sample population. When the counter reaches zero:

- A member of the sampling population is selected for sampling.
- The counter is set as follows:
  - PMSICR_EL1.COUNT[31:8] is set to PMSIRR_EL1.INTERVAL.
  - PMSICR_EL1.COUNT[7:0] is set to `0x00`.

Note

Because the counter counts down to zero, when PMSIRR_EL1.RND is cleared to 0 the interval between operations being selected for sampling is (*INTERVAL*×256+1).

If PMSIRR_EL1.RND is 1 and PMSIDR_EL1.ERnd is 0

While nonzero, the sample interval counter decrements by 1 for each member of the sample population. When the counter reaches zero:

- A member of the sampling population is selected for sampling.
- The counter is set as follows:
  - PMSICR_EL1.COUNT[31:8] is set to PMSIRR_EL1.INTERVAL.
  - PMSICR_EL1.COUNT[7:0] is set to a random or pseudorandom value in the range `0x00` to `0xFF`.

Note

When PMSIRR_EL1.RND is set to 0 and PMSIDR_EL1.ERnd is 1, the mean interval between operations being selected for sampling is (*INTERVAL*×256+128), if the random number generator is uniform.

If PMSIRR_EL1.RND is 1 and PMSIDR_EL1.ERnd is 1

While nonzero, the primary sample interval counter decrements by 1 for each member of the sample population. When the primary counter reaches zero:

- The primary sample interval counter is reloaded.
- The secondary sample interval counter, PMSICR_EL1.ECOUNT, is set to a random or pseudorandom value in the range `0x00` to `0xFF`.

While the secondary sample interval counter is nonzero, the secondary sample interval counter decrements by 1 for each member of the sample population. The primary sample interval counter also continues to decrement because it is also nonzero.

When the secondary sample interval counter reaches zero, an operation is selected for sampling.

Note

When PMSIRR_EL1.RND is set to 1 and PMSIDR_EL1.ERnd is 1, the mean interval between operations being selected for sampling is (*INTERVAL*×256+1), if the random number generator is uniform.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D9.3.5 Behavior of the sample interval counter while profiling is disabled

When profiling is disabled:

- No operations are selected for sampling.
- No sample records are collected.
- The sample interval counters retain their values and do not decrement.

D9.3.6 Where operations are sampled

The exact point in the sampled lifespan of operations at which operations are chosen for profiling is IMPLEMENTATION DEFINED.

Note

Arm recommends that the point at which operations are sampled is linked to the definition of the Performance Monitors Extension (PMU) STALL_FRONTEND and STALL_BACKEND events, so that sampling records information for STALL_BACKEND stalls.

D9.3.7 Sample collisions

The maximum number of sampled operations that a PE can support simultaneously is IMPLEMENTATION DEFINED. If the maximum number of simultaneous sampled operations has been reached at the point when a new operation must be sampled, the new sample is said to have collided with a previous sampled operation.

The PE records the fact that a sampled operation has collided with another sampled operation. Software can also count the number of collisions and gauge the impact of the collisions.

On a sample collision:

- The PMU event SAMPLE_COLLISION is generated.
- PMBSR_EL1.COLL is set to 1.
- The new operation is not sampled.

Following a context synchronization event an indirect write to PMBSR_EL1.COLL is guaranteed to be visible to instructions in program order after the sampled operation that collided. There is no guarantee of visibility without a context synchronization event. For more information see *Synchronization and Statistical Profiling* on page D9-2807.

Note

This means that following a context synchronization event PMBSR_EL1.COLL will not change on entry to a state where profiling is disabled.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D9.4 Enabling profiling

Profiling is disabled if the Profiling Buffer is disabled, including when:

- PMBLIMITR_EL1.E is cleared to 0 or PMBSR_EL1.S is set to 1.
- Executing at a higher Exception level than the Profiling Buffer owning Exception level.
- Executing in the Security state that is not the Security state of the owning Exception level.
- The PE is in Debug state.

Note

The owning Exception level is controlled by MDCR_EL3.NSPB and MDCR_EL2.E2PB.

PMSCR_EL1.{E1SPE, E0SPE} and PMSCR_EL2.{E2SPE, E0HSPE} enable sampling by Exception level:

- In a guest operating system or Secure state, PMSCR_EL1.E1SPE enables profiling at EL1 and PMSCR_EL1.E0SPE at EL0.
- In a hypervisor or host operating system, PMSCR_EL2.E2SPE enables profiling at EL2 and PMSCR_EL2.E0HSPE at EL0.
- Sampling is always disabled at EL3.

Table D9-1 defines the valid combinations of the Effective values of SCR_EL3.NS, SCR_EL3.EEL2, MDCR_EL3.NSPB, MDCR_EL2.E2PB, and HCR_EL2.TGE that define when sampling is enabled.

In Table D9-1:

D Disabled.

E2SPE Enabled if PMSCR_EL2.E2SPE == 1, disabled otherwise.

E1SPE Enabled if PMSCR_EL1.E1SPE == 1, disabled otherwise.

E0HSPE Enabled if PMSCR_EL2.E0HSPE == 1, disabled otherwise.

E0SPE Enabled if PMSCR_EL1.E0SPE == 1, disabled otherwise.

Table D9-1 Enabling by Exception level and Security state (for all Exception levels using AArch64 state)

Controls					Sampling enabled at			
NS	NSPB	E2PB	EEL2	TGE	EL3	EL2	EL1	EL0
1	`0b0X`	X	X	X	D	D	D	D
	`0b1X`	`0b1X`	X	0	D	D	E1SPE	E0SPE
		`0b1X`	X	1	D	D	n/a	D
		`0b00`	X	0	D	E2SPE	E1SPE	E0SPE
		`0b00`	X	1	D	E2SPE	n/a	E0HSPE
0	`0b1X`	X	X	X	D	D	D	D
	`0b0X`	X	0	X	D	n/a	E1SPE	E0SPE
		`0b1X`	1	0	D	D	E1SPE	E0SPE
		`0b1X`	1	1	D	D	n/a	D
		`0b00`	1	0	D	E2SPE	E1SPE	E0SPE
		`0b00`	1	1	D	E2SPE	n/a	E0HSPE

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This is described in the pseudocode function `StatisticalProfilingEnabled()`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D9.5 Filtering sample records

PMSFCR_EL1.FT enables filtering by operation type. When enabled PMSFCR_EL1.{ST, LD, B} define the collected types:

- ST enables collection of store sampled operations, including all atomic operations.
- LD enables collection of load sampled operations, including atomic operations that return a value to a register.
- B enables collection of branch sampled operations, including direct and indirect branches and exception returns.

Note

When micro-op sampling is implemented, filtering is based on the micro-op type.

Table D9-2 summarizes the controls for filtering by operation type. In this table:

Load Atomic Refers to atomic operations which return a value to a general-purpose register. Other atomic operations are classed as Store.

D Indicates that the operation is discarded.

C Indicates that the operation is collected.

Table D9-2 Filtering by Operation type

PMSFCR_EL1 field				Operation type				
FT	LD	ST	B	Load	Load Atomic	Store	Branch	Other
0	X	X	X	C	C	C	C	C
1	0	0	0	D	D	D	D	D
			1	D	D	D	C	D
		1	0	D	C	C	D	D
			1	D	C	C	C	D
	1	0	0	C	C	D	D	D
			1	C	C	D	C	D
		1	0	C	C	C	D	D
			1	C	C	C	C	D

PMSFCR_EL1.FE enables filtering by a set of events that are defined by PMSEVFR_EL1. When enabled, only sampled operations with all the events in the filter set are recorded and written to the Profiling Buffer.

PMSFCR_EL1.FL enables filtering by total latency. PMSLATFR_EL1.MINLAT defines the minimum latency. When enabled, only sampled operations with a total latency greater than or equal to the minimum latency are recorded and written to the Profiling Buffer.

These controls combine together as a logical AND.

This is described in the pseudocode function `CollectRecord()`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D9.6 The profiling data

Unless otherwise stated all sample records that are generated by a profiling operation contain:

- A timestamp, if enabled. This is one of:
  - The physical counter, CNTPCT_EL0.
  - The offset physical counter, CNTPCT_EL0 - CNTPOFF_EL2. When any of the following are true, the *Effective value* of CNTPOFF_EL2 is 0 for all trace and SPE purposes:
    - EL3 is using AArch32.
    - EL2 is not implemented.
    - ARMv8.6-ECV is not implemented.
    - EL2 is enabled in the current Security state and is using AArch32.
    - CNTHCTL_EL2.ECV is 0.
    - SCR_EL3.ECVEn is 0.
  - The virtual counter, CNTVCT_EL0.

  It is IMPLEMENTATION DEFINED how this timestamp relates to the sampled operation. It might be the time when the sampled operation was taken or any later time during the lifetime of the sampled operation, that is, up to the time when the sampled operation is completed.

  If the Generic Timer system counter is disabled and timestamps are enabled, then it is IMPLEMENTATION DEFINED whether:
  - The Statistical Profiling Extension behaves as if timestamps are disabled.
  - The timestamp that is collected in the sample record is UNKNOWN.

  Note

  This behavior describes when CNTEN.EN is cleared to 0. This behavior does not apply when the Generic Timer system counter is enabled but not accessible at the current Exception level.

- The context, if enabled, which is one or more of:
  - CONTEXTIDR_EL1.
  - CONTEXTIDR_EL2.
  - The Exception level.
  - The Security state.
- Information about whether the sampled operation generated an exception:
  - The target address for an exception generating operation is not collected.
- Information about whether the sampled operation completed execution.

If the sampled operation completes execution and does not generate an exception, the sample record also contains:

- The PC virtual address for the sampled operation.
- Information about whether the sampled operation is a branch, a load, a load atomic, a store, or other.
- Information about whether the sampled operation is conditional, conditional select, or not.
- The total latency, a cycle count of the lifetime of the sampled operation. For more information see *Lifespan of a sampled operation* on page D9-2797.
- The issue latency, a cycle count from the start of the sampled operation up to the point when at least one part of the sampled operation starts executing. A sampled operation might be delayed, for example, because the input operands were not available.

If the sampled operation does not complete execution or generates an exception it is UNPREDICTABLE whether the record contains all or any of this information and the other information about the operation listed in this section and the following subsections. For information on exceptions being taken in sampled operations see *Exceptions* on page D9-2797.

The architecture defines a set of additional data that is collected in the sample record for each sampled operation. This is described in the following subsections, and comprises:

- Events, which are required to be implemented consistently with PMU Events, for more information see Chapter D10 *Statistical Profiling Extension Sample Record Specification* and Chapter D7 *The Performance Monitors Extension*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Cycle counters. Cycle count values as described in this architecture, which, for a particular implementation, are fixed with an IMPLEMENTATION DEFINED value, might be omitted from the sample record.
- Addresses.

In addition, the architecture permits IMPLEMENTATION DEFINED events, counters, and addresses to be collected.

D9.6.1 Information collected for micro-ops

Because architectural instructions might create zero, one, or more micro-ops, micro-ops might have different characteristics from the architectural instructions they are created from. The data collected for each micro-op is IMPLEMENTATION DEFINED. Implementations should collect the subset of data appropriate to the micro-op.

Example D9-1 Collection of sampled operations

If PMSFCR_EL1.FE is set to 1, PMSFCR_EL1.FT is set to 1, and PMSFCR_EL1.FL is set to 1, then only sampled operations that meet all of the following criteria are recorded and written to the Profiling Buffer:

- The sampled operation is one of the selected operation types.
- The operation has all of the events in the filter set.
- The total latency is equal to or greater than the minimum latency.

D9.6.2 Additional information for each profiled branch or exception return

For a completed branch or exception return sampled operation, the profiling operation must record:

- The sampled operation type as an unconditional branch or a conditional branch. Sampled exception returns are treated as unconditional branches by the Statistical Profiling Extension.
- If the branch is taken, the target virtual address of the branch. The target virtual address of the branch includes the Exception level and Security state of the target. The target virtual address includes the Exception level and Security state of the target. If the sampled operation is an illegal exception return, it is CONSTRAINED UNPREDICTABLE whether the context information recorded in the target virtual address is the actual target context, or the target context that is described by the SPSR.
- If the PE implements branch prediction, whether the branch was correctly predicted or mispredicted.
- Whether the branch was taken or not taken.
- Whether the branch was direct or indirect.
- If the branch is not taken, a target virtual address might be recorded. Software must treat this value if present as UNKNOWN.

Note

A sampled operation that generates an exception is not treated as a branch.

D9.6.3 Additional information for each profiled memory access operation

For a completed load, store, or atomic sampled operation that does not generate an exception, the profiling operation must record:

- The data virtual and, if enabled, physical addresses being accessed.
  - If the applicable Top Byte Ignore (TBI) bit is set to one, the virtual address includes any top-byte tag.
  - The physical address is the address the PE accesses in the physical address space, and so includes the Secure Address Space Identifier.
- The sampled operation type, which includes:
  - Whether the sampled operation is a load, store, or atomic.
  - Whether the sampled operation is Load-Exclusive, Store-Exclusive or Load-acquire, Store-release.
  - Whether the sampled operation accesses the general-purpose or SIMD&FP registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The translation latency. This is defined as an IMPLEMENTATION DEFINED choice between:
  - The count of cycles for which at least one part or a chosen part of a load or store operation is waiting for the MMU to complete an address translation, and no part of the operation is accessing memory.
  - The count of cycles for which at least one part or a chosen part of a load or store operation is waiting for the MMU to complete an address translation.
- Whether the sampled operation accessed the Level 1 data cache and the result.
- Whether the sampled operation accessed the data TLB and the result.
- An optional, IMPLEMENTATION DEFINED, record of whether the sampled operation accessed Last Level data cache and the result.
- An optional, IMPLEMENTATION DEFINED, record of whether the sampled operation accessed *another socket* in a multi-socket system.
- An optional, IMPLEMENTATION DEFINED, indicator of the data source for a load.
- If ARMv8.3-SPE is implemented, an optional indication that the sampled memory operation is non-optimal for the access size. For more information see *Data Alignment Flag*.

For each of the Last level cache and another socket indicators, it is IMPLEMENTATION DEFINED whether this information is present only for load accesses, only for store accesses, for neither, or for both.

For more information see *Events packet* on page D10-2824.

Note

A store might be marked as not accessing a cache or another socket because it completed before doing so. For example, the write was held in a write buffer. This behavior is IMPLEMENTATION DEFINED, and such events must be interpreted with care.

If architecture instructions are sampled, for a sampled load or store operation that is not single-copy atomic, the data addresses are the lowest address that is accessed by the sampled operation regardless of whether architecture instructions are sampled or not.

Otherwise the information is for the micro-op that is sampled.

Example D9-2 Sampling of micro-ops

If an architectural load instruction is split into an address generation micro-op and a load micro-op, then when generating the sample record and filtering based on operation type:

- If the address generation micro-op is sampled, the sampled operation is treated as *other*.
- If the load micro-op is sampled, the sampled operation is treated as a load.

The sampled data physical address is always the address generated from translating the sampled data virtual address. The sampled data physical address packet is not output if any of the following are true:

- The PE does not translate the address, for example because it does not perform the access or the address translation generates a Translation fault.
- The sampled data virtual address packet is not output.
- Prohibited by System register controls.

If ARMv8.5-MemTag is implemented, a PE will generate an Unchecked access for each access to the Profiling Buffer as part of writing a sample record.

For more information on ARMv8.5-MemTag, see Chapter D6 *Armv8.5 Memory Tagging Extension*.

Data Alignment Flag

If ARMv8.3-SPE is implemented Events packet.E[11] is set to 1 for a sampled memory operation if the address alignment is non-optimal for the access size.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Address alignment is defined as *non-optimal* if that access incurs an additional performance penalty only because of the address alignment, and is unrelated to whether the access is architecturally misaligned for the access size.

Example D9-3 Data Alignment Flag operation

- A 32-bit word access that is not word aligned is architecturally misaligned, but (if Alignment faults are disabled) might not incur an additional penalty because of this alignment unless the word also happens to span a cache-line boundary.
- A contiguous load operation that loads a vector that is the length of two cache lines is optimally aligned if it has cache-line alignment, even though the operation makes two cache line accesses.
- A non-contiguous SVE load operation that makes a sequence of access is optimal only if all of the access are optimal.

The definition of non-optimal is IMPLEMENTATION DEFINED and support for the Alignment Flag is OPTIONAL.

D9.6.4 Additional information for each profiled conditional instruction

For a completed conditional select, conditional move, or conditional increment sampled operation, the profiling operation must record:

- That the sampled operation was conditional.
- Whether the condition passed or failed.

For conditional branches, see *Additional information for each profiled branch or exception return* on page D9-2791.

D9.6.5 Additional Information for each profiled Scalable Vector Extension operation

When ARMv8.3-SPE and The Scalable Vector Extension (SVE) are implemented, SVE operations are sampled as described in this section.

In this section the following terms are used:

Maximum implemented vector length

Means the implemented width of the vector registers. This value is IMPLEMENTATION DEFINED.

Accessible vector length

Means the accessible width of the SVE vector registers at the current Exception level, as constrained by the ZCR_EL1, ZCR_EL2 or ZCR_EL3 System registers. The Accessible vector length is always less-than-or-equal-to the Maximum implemented vector length.

Sampled SVE operation

Means an instruction or micro-operation defined by the *Arm Architecture Reference Manual Supplement: The Scalable Vector Extension (SVE), for Armv8-A* and sampled by the Statistical Profiling Extension that has a vector or a predicate as an input or output. This includes instructions with scalar outputs, but excludes the Non-SIMD SVE instructions.

Sampled operation vector

Means the portion of the accessible vector operated on by the Sampled SVE operation.

Effective vector length

Is the length of the Sampled operation vector. The Effective vector length is always less-than-or-equal-to the Accessible vector length.

Note

The Accessible vector length is always quantized into multiples of 128-bits. However, the Sampled operation vector can be any size down to the element size of the operation.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Sampled predicated SVE operation

Means a Sampled SVE operation that is one of:

- An SVE operation that writes to a vector destination register under a Governing predicate using either zeroing or merging predication.
- A predicated store of a vector register or registers.

For an implementation that samples micro-operations, an SVE instruction might be split up into one or more micro-operations, some of which are predicated and some of which are not predicated.

Note

Sampled predicated SVE operation excludes operations that do not write a vector register, or do so but not using zeroing or merging predication, and applies to machine instructions rather than aliases. For example, the following instructions are not predicated SVE instructions under this definition:

- `CNTP`, `LASTA`, and `PTRUE` do not write to vector registers.
- `FADDV`, and `SMAX` write scalar values to SIMD&FP registers.
- `COMPACT` and `SEL (vectors)` write to vector registers, and have a predicate operand, but do not use that predicate as a Governing predicate for zeroing or merging predication.
- `MOV (vector, predicated)` appears to be a predicated SVE instruction because it specifies merging predication through the `<PG>/M` operand, but it is actually an alias for the `SEL (vectors)` instruction.

If an implementation samples micro-operations, it is IMPLEMENTATION DEFINED whether individual elements, or groups of elements, are treated as single micro-operations.

The division of instructions into micro-operations must be fixed prior to sampling to guarantee consistently accurate statistical sampling.

Example D9-4 Vector length

For example, to support a vector length of 1024 bits, an implementation might split all instructions into four micro-operations on 256-bit vector paths. The implementation must, however, implement 1024-bit wide vector registers.

This behavior might vary based on operation type. For example, an implementation that has a full-width data-path for most operations might choose to break certain complex operations, such as non-contiguous load or stores, into shorter vectors.

Example D9-5 Accessible vector length less-than the Maximum implemented

To support an Accessible vector length less-than the Maximum implemented vector length, an implementation might choose to do all operations at the Maximum implemented vector length and discard the results above the Accessible vector length. Discarded results, arising from difference between Maximum implemented vector length and Accessible vector length, do not form part of the sampled operation and the Effective vector length must not include any discarded portions of the vector.

Results discarded because of predication are part of the sampled operation.

For a sampled SVE cache prefetch operation:

- The profiling operation captures an IMPLEMENTATION DEFINED subset of the information captured for an SVE load instruction.
- The profiling operation treats the operation type as Other when generating the sample records and filtering based on operation.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- It is IMPLEMENTATION DEFINED whether the operation is treated as a Sampled SVE operation:
  - If treated as a Sampled SVE operation, the Operation Type packet payload format is the *Operation Type packet* on page D10-2829.
  - If not treated as a Sampled SVE operation, the Operation Type packet format is the *Operation Type packet payload (Other)* on page D10-2829.

For a Sampled SVE operation the Operation Type packet is one of:

- The SVE operation format.
- The SVE load or store format.

For a Sampled SVE operation the Operation Type packet.EVL field records an upper bound on the Effective vector length. The value recorded in the Operation Type packet.EVL field is the Effective vector length rounded up to a power-of-two value.

For a Sampled SVE operation that is a Sampled predicated SVE operation;

- Operation Type packet.PRED, Predicated SVE operation, is set to 1.
- If any elements in the Sampled operation vector are Inactive elements, then Events packet.E[17], Partial predicate, is set to 1.
- If all elements in the Sampled operation vector are Inactive elements, then Events packet.E[18],Empty predicate, is set to 1 and Events packet.E[17] (Partial predicate) is set to 1.
- If all elements in the Sampled operation vector are Active elements then Events packet.E[18:17] is set to `0b00`.

For a Sampled SVE operation, that is not a Sampled predicated SVE operation:

- Operation Type packet.PRED, Predicated SVE operation, is set to 0.
- Events packet.E[18:17] is set to `0b00`.

For a sampled non-contiguous SVE load or store operation that makes multiple memory accesses, the sampled data virtual address is the address accessed by a random one of the load or store operations chosen from the Sampled operation vector. If the chosen load or store operation is for an Inactive element, the data virtual address packet is not output.

For more information on memory access operations see *Additional information for each profiled memory access operation* on page D9-2791.

For a sampled contiguous SVE load or store operation that makes multiple memory accesses, the sampled data virtual address is an IMPLEMENTATION DEFINED choice of:

- The address accessed for the lowest element in the Sampled operation vector.
- The address used for the access containing the lowest Active element in the Sampled operation vector.

If the corresponding element is an Inactive element, it is IMPLEMENTATION DEFINED whether the data virtual address packet is output.

D9.6.6 Additional information for other operations

For cache maintenance operations by virtual address, cache prefetch, other than SVE cache prefetch, or address translation instructions, the profiling operation:

- Captures an IMPLEMENTATION DEFINED subset of the information captured for a load instruction.
- Treats the operation type as *other* when generating the sample record and filtering based on operation type.

See *Filtering sample records* on page D9-2789, Operation Type packet and *Additional Information for each profiled Scalable Vector Extension operation* on page D9-2793.

D9.6.7 Controlling the data that is collected

Certain data in sample records is only collected if permitted by one or both of EL1 and EL2. This is to restrict exposure of data to a lower Exception level or to Non-secure state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CONTEXTIDR_EL1 is collected only if PMSCR_EL1.CX is set to 1, the PE is executing at EL1 or EL0 and any of the following are true when an operation is sampled:

- EL2 is not implemented.
- ARMv8.4-SecEL2 is implemented and EL2 is disabled for the current Security state.
- The Effective value of HCR_EL2.TGE is 0.

CONTEXTIDR_EL2 is collected only if the Effective value of PMSCR_EL2.CX is 1 and EL2 is implemented and enabled for the current Security state.

This is described in the pseudocode functions `CollectContextIDR1()` and `CollectContextIDR2()`.

Timestamps are collected only if one of the following is true:

- PMSCR_EL1.TS is set to 1 and the Profiling Buffer is owned by EL1.
- PMSCR_EL2.TS is set to 1 and the Profiling Buffer is owned by EL2.

The timestamp is a choice between:

- Physical time, which is defined by the value of CNTPCT_EL0.
- If ARMv8.6-ECVis implemented and enabled, offset physical time, as defined by the value of (CNTPCT_EL0 - CNTPOFF_EL2). That is, the physical time minus the physical offset, CNTPOFF_EL2.
- Virtual time, as defined by the value of CNTVCT_EL0. That is, the physical time minus the virtual offset, CNTVOFF_EL2. However, the virtual offset is treated as zero if a read of CNTVCT_EL0 at the current Exception level would treat the virtual offset as zero.

Table D9-3 summarizes the choice of value for the Timestamp packet when ARMv8.6-ECV is implemented and `StatisticalProfilingEnabled()` is TRUE. In Table D9-3:

Owning EL This is the Exception level that owns the Profiling Buffer. This is returned by the function `ProfilingBufferOwner()`. If EL2 is disabled in the current Security state, this is always EL1.

EL2 enabled This is TRUE when EL2 is enabled in the current Security state. When EL2 is disabled in the current Security state, this is FALSE.

Virtual This means the timestamp is offset physical time, as returned by a direct read of CNTVCT_EL0 at the Exception level the sampled operation is executed at.

Physical This means the timestamp is physical time, given by the value of CNTPCT_EL0 at the Exception level the sampled operation is executed at.

Offset physical This means the timestamp is offset physical time, given by the value (CNTPCT_EL0 - CNTPOFF_EL2) at the Exception level the sampled operation is executed at. That is, the physical time minus the physical offset. When any of the following are true, the *Effective value* of CNTPOFF_EL2 is 0 for all trace and SPE purposes:

- EL3 is using AArch32.
- EL2 is not implemented.
- ARMv8.6-ECV is not implemented.
- EL2 is enabled in the current Security state and is using AArch32.
- CNTHCTL_EL2.ECV is 0.
- SCR_EL3.ECVEn is 0.

Table D9-3 Recorded timestamp when ARMv8.6-ECV is implemented

EL2 enabled	Owning EL	PMSCR_EL2		PMSCR_EL1		Recorded timestamp
		PCT[1:0]	TS	PCT[1:0]	TS	
x	EL1	xx	x	xx	0	None
				0b00	1	Virtual
FALSE	EL1	xx	x	0b01	1	Physical
				0b11	1	Offset physical

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D9-3 Recorded timestamp when ARMv8.6-ECV is implemented (continued)

EL2 enabled	Owning EL	PMSCR_EL2		PMSCR_EL1		Recorded timestamp
		PCT[1:0]	TS	PCT[1:0]	TS	
TRUE	EL1	0b00	x	xx	1	Virtual
		0b01	x	0b01	1	Physical
				0b11	1	Offset physical
		0b11	x	0b01	1	Offset physical
				0b11	1	Offset physical
	EL2	xx	0	xx	x	None
		0b00	1	xx	x	Virtual
		0b01	1	xx	x	Physical
		0b11	1	xx	x	Offset physical

If EL2 is not implemented, see the register descriptions of PMSCR_EL1.PCT and PMSCR_EL2.PCT for details of their behavior. This behavior is described by the pseudocode function `CollectTimeStamp()`.

Physical data addresses are collected only if one of the following is true:

- PMSCR_EL1.PA is set to 1 and the Profiling Buffer is owned by Secure EL1, and Secure EL2 is disabled or is not implemented.
- PMSCR_EL2.PA is set to 1 and the Profiling Buffer is owned by Secure or Non-secure EL2.
- PMSCR_EL1.PA is set to 1 and PMSCR_EL2.PA is set to 1 and either the Profiling Buffer is owned by Non-secure EL1, or the Profiling Buffer is owned by Secure EL1 and Secure EL2 is implemented and enabled.

If EL2 is not implemented or is disabled for the current Security state, the PE behaves as if PMSCR_EL2.PA is set to 1, other than for a direct read of the register.

Enabling collection of the physical data addresses has an IMPLEMENTATION DEFINED impact on the sampled operation. This is described by the pseudocode function `CollectPhysicalAddress()`.

D9.6.8 Lifespan of a sampled operation

The sampled lifespan of an operation:

- Starts with the decoded architectural instruction or micro-op is dispatched for issue.
- Completes when one of:
  - The instruction or micro-op is committed to the architectural state of the PE (including completion by generating a synchronous exception).
  - The speculative instruction or micro-op is discarded because of misspeculation. For more information see *Misspeculation and non-architectural operations* on page D9-2782.
  - The instruction or micro-op is replayed, on some microarchitectures. For more information see *Non-architectural exceptions* on page D9-2798.

D9.6.9 Exceptions

All sample records written to the Profiling Buffer contain the Events packet and either the End packet or the Timestamp packet.

If the sampled operation generates an exception, it is UNPREDICTABLE whether the sample record contains any other information.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Where a sampled operation generates an exception and the type of exception means that a particular item is not computed by the sampled operation, that information is not collected by the profiling operation. For more information see *Synchronization and Statistical Profiling* on page D9-2807.

Example D9-6 Translation Faults

If a sampled operation generates a Translation Fault, the physical address for the sampled operation was not generated by the MMU and cannot be recorded.

Non-architectural exceptions

An implementation might include exceptions that are not architectural exceptions. That is, in executing an operation, the PE performs some exceptional behavior that does not take an exception as defined by the architecture. These non-architectural exceptions are usually not visible to software.

The use on non-architectural exceptions are IMPLEMENTATION DEFINED.

If a sampled operations generates a non-architectural exception, the sample might include handling of the non-architectural exception. If the sample does not include handling of the non-architectural exception, then the sampled operation does not complete because of the non-architectural exception and it is recorded using E[1] == 0 (operation did not retire) in the Events packet. Bit E[0] (operation generated an exception) might be used to indicate the operation did not complete because of the non-architectural exception.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D9.7 The Profiling Buffer

The profile data is collected in a memory *Profiling Buffer*. The Profiling Buffer is defined by:

- PMBPTR_EL1, the current write pointer.
- PMBLIMITR_EL1, the write limit pointer.

The Profiling Buffer starts at the current write pointer and extends to the current limit pointer minus one. The write limit pointer must be aligned to the smallest implemented translation granule size. The alignment of the current write pointer is IMPLEMENTATION DEFINED.

PMBLIMITR_EL1 and PMBPTR_EL1 are virtual addresses in the stage 1 translation regime of the owning Exception level. This is called the *owning translation regime*.

Note

The translation of virtual addresses to physical addresses is identical to that for any other virtual address in the owning Exception level. For example, PMBPTR_EL1[63:56] are ignored by address translation if the respective TBI bit is set to 1.

D9.7.1 Restrictions on the current write pointer

This section describes the software rules on setting the current write pointer, PMBPTR_EL1. If these rules are not followed, the value returned for a direct read of PMBPTR_EL1 is UNKNOWN, the behavior is UNPREDICTABLE, and the PE might do any of the following at any point after profiling is enabled:

- Write sample records to any writeable address in memory that is writable at the owning Exception level.
- Generate a Profiling Buffer management event, with or without indicating data loss, for one of the following reasons:
  - The Profiling Buffer is full.
  - Any MMU Fault.

When profiling becomes enabled, all the following must be true:

- The current write pointer must be at least one sample record below the write limit pointer. That is:
  `UInt`(PMBPTR_EL1.PTR) <= `UInt`(PMBLIMITR_EL1.LIMIT :`Zeros`(12)) - $2^{\text{PMSIDR_EL1.MaxSize}}$.
- PMBPTR_EL1.PTR[63:56] must equal PMBLIMITR_EL1.LIMIT[63:56].

When the Profiling Buffer is first configured, PMBPTR_EL1.PTR must be aligned to PMBIDR_EL1.Align. That is, if PMBIDR_EL1.Align is nonzero, PMBPTR_EL1.PTR [`UInt`(PMBIDR_EL1.Align)-1:0] must be all zeros.

However, the current write pointer can usually be restored to the saved write pointer value it had when profiling was disabled, providing a `PSB CSYNC` and a context synchronization event were executed before reading PMBPTR_EL1:

- If no Profiling Buffer management event was signaled then profiling can be restarted from the saved write pointer. In this case, the saved write pointer points within one sample record of the write limit pointer.
- If a Profiling Buffer management event was signaled then:
  - If PMBSR_EL1.S is restored to 1, then profiling is not being enabled, and there are no constraints on the value written to PMBPTR_EL1.
  - If PMBSR_EL1.S is restored to 0, and the Profiling Buffer management event was caused by an MMU fault, profiling can be restarted from the saved write pointer; if PMBSR_EL1.{EA, DL} did not also indicate an external abort or data loss, and the saved write pointer is at least one sample record below the write limit pointer.

    Note

    If a signaled MMU fault has not been corrected, the Statistical Profiling Extension generates a new MMU fault Profiling Buffer management event when it next tries to write a sample record.

  - If PMBSR_EL1.S is restored to 0, and the Profiling Buffer management event was caused by a buffer full event, the Profiling Buffer can be extended and profiling restarted from the saved write pointer; if PMBSR_EL1.{EA, DL} did not also indicate an external abort or data loss and the saved write pointer is at least one sample record below the extended write limit pointer.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The current write pointer must not be restored from the saved write pointer following a Profiling Buffer management event if PMBSR_EL1.DL was set to 1.

The saved write pointer might not be aligned to $2^{\text{PMBIDR_EL1.Align}}$ and might point to within one sample record of the write limit pointer.

For more information see *Synchronization and Statistical Profiling* on page D9-2807.

D9.7.2 The owning Exception level

The owning Exception level is:

- Non-secure EL1, if all of the following are true:
  - Either EL3 is not implemented and the PE is executing in Non-Secure state, or MDCR_EL3.NSPB is set to either `0b10` or `0b11`.
  - Either EL2 is not implemented, or MDCR_EL2.E2PB is set to either `0b10` or `0b11`.
- Non-secure EL2, if all of the following are true:
  - EL2 is implemented.
  - Either EL3 is not implemented and the PE is executing in Non-secure state, or MDCR_EL3.NSPB is set to either `0b10` or `0b11`.
  - MDCR_EL2.E2PB is set to `0b00`.
- Secure EL1, if all of the following are true:
  - Either EL3 is not implemented and the PE is executing in Secure state, or MDCR_EL3.NSPB is set to either `0b00` or `0b01`.
  - Either Secure EL2 is not implemented or is disabled, or MDCR_EL2.E2PB is set to either `0b10` or `0b11`.
- Secure EL2, if all of the following are true:
  - Secure EL2 is implemented and enabled.
  - Either EL3 is not implemented and the PE is executing in Secure state, or MDCR_EL3.NSPB is set to either `0b00` or `0b01`.
  - MDCR_EL2.E2PB is set to `0b00`.

When the owning Exception level is Non-secure EL1

The Profiling Buffer addresses are in the Non-secure EL1&0 translation regime using the current ASID from TTBRx_EL1. This is a two-stage translation using the current VMID if EL2 is implemented and HCR_EL2.VM is set to 1.

If EL3 is implemented, then the Profiling Buffer is disabled in Secure state.

If EL2 is implemented, then profiling is disabled at EL2 and at Non-secure EL0 when HCR_EL2.TGE is set to 1.

When the owning Exception level is Non-secure EL2

The Profiling Buffer addresses are in the Non-secure EL2 translation regime. If both HCR_EL2.E2H is set to 1 and HCR_EL2.TGE is set to 1, this is an EL2&0 translation regime using the current EL2&0 translation regime ASID from TTBRx_EL2.

If EL3 is implemented, then the Profiling Buffer is disabled in Secure state.

Note

If either HCR_EL2.E2H is cleared to 0 or HCR_EL2.TGE is cleared to 0, and the PE is executing at EL1 or EL0, the EL2 translation regime is not the current stage 1 translation regime because the current stage 1 translation regime is EL1&0.

When the owning Exception level is Secure EL1

The Profiling Buffer addresses are in the Secure EL1&0 translation regime using the current ASID from TTBRx_EL1. This is a two-stage translation using the current VMID if Secure EL2 is implemented and enabled and HCR_EL2.VM is set to 1.

If EL3 is implemented, then the Profiling Buffer is disabled in Non-secure state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If Secure EL2 is implemented and enabled, then profiling is disabled at EL2 and at Secure EL0 when HCR_EL2.TGE is set to 1.

When the owning Exception level is Secure EL2

The Profiling Buffer addresses are in the Secure EL2 translation regime. If both HCR_EL2.E2H is set to 1 and HCR_EL2.TGE is set to 1, this is an EL2&0 translation regime using the current EL2&0 translation regime ASID from TTBRx_EL2.

If EL3 is implemented, then the Profiling Buffer is disabled in Non-secure state.

Summary of the owning translation regime

The Profiling Buffer is disabled if any of the following are true:

- The owning Exception level is using AArch32 state.
- PMBLIMITR_EL1.E is cleared to 0.

Table D9-4 summarizes the owning translation regime.

Table D9-4 Summary of owning translation regime (for all Exception levels using AArch64 state)

PMBLIMITR_EL1.E	SCR_EL3.NS	SCR_EL3.EEL2	MDCR_EL3.NSPB	MDCR_EL2.E2PB	Owning translation regime
0	X	X	X	X	Disabled
1	1	0	`0b1x`	`0b1x`	Non-secure EL1&0
				`0b00`	Non-secure EL2 or EL2&0[a]
			`0b0x`	X	Disabled
	0	0	`0b0x`	X	Secure EL1&0
	0	1	`0b0x`	`0b1x`	Secure EL1&0
				`0b00`	Secure EL2 or EL2&0[a]
		X	`0b1x`	X	Disabled

a. Depending on the values of HCR_EL2.{E2H,TGE}.

D9.7.3 Memory access types and coherency

The Statistical Profiling Extension acts as a separate observer in the system and is subject to the rules regarding coherency.

Writes to any Device memory type by the Statistical Profiling Extension occur once.

The memory type and attributes that are used for a write by the Statistical Profiling Extension to the Profiling Buffer is taken from the translation table entries for the virtual address being written to. That is:

- The writes are treated as coming from an observer that is coherent with all observers in the Shareability domain that is defined by the translation tables.
- There is no requirement to manage coherency for observers in the same Shareability domain but coherency for other observers in the system might require explicit management.

For more information see *Synchronization and Statistical Profiling* on page D9-2807.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If ARMv8.5-MemTag is implemented, an instruction which loads or stores an Allocation tag will be treated as a load or store if profiling is enabled. For more information on ARMv8.5-MemTag, see Chapter D6 *Armv8.5 Memory Tagging Extension*.

Writes to the Profiling Buffer are made as privileged writes within the owning translation regime.

This means that if ARMv8.5-E0PD is implemented, the values of TCR_ELx.E0PDy, where ELx is the owning Exception level, do not apply to accesses to the Profiling Buffer made by the Statistical Profiling Extension.

D9.7.4 Cache and TLB operations

TLB maintenance operations that affect the TLB of the PE also affect any TLB caching translations for the Statistical Profiling Extension of that PE.

Cache maintenance operations that affect the caches of the PE also affect data caching by the Statistical Profiling Extension of that PE.

This means that the completion of any cache or TLB maintenance instruction includes its completion on all Statistical Profiling Extensions for PEs that are affected by both the instruction and the `DSB` operation that is required to guarantee visibility of the maintenance instruction. See *Completion and endpoint ordering* on page B2-130.

Although the Statistical Profiling Extension acts as another observer in the system, for determining the Shareability domain of this `DSB`, or cache, or TLB maintenance operation, the writes of sample records are treated as coming from the PE that is being profiled.

D9.7.5 Effect on the exclusive monitors

If an operation between Load-Exclusive and Store-Exclusive instructions is sampled, then the Store-Exclusive must be guaranteed not to fail, even though the sample record is written to an unrelated address.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D9.8 Profiling Buffer management

A Profiling Buffer management event occurs:

- On a fault, see *Faults and watchpoints* on page D9-2804.
- On an external abort, see *External aborts* on page D9-2806.
- When the Profiling Buffer fills, see *Buffer full event* on page D9-2804.

On a Profiling Buffer management event:

- The service bit, PMBSR_EL1.S, is set to 1.
- The data loss bit, PMBSR_EL1.DL, is set as described in the event description.
- The Profiling Buffer management interrupt request signal, **PMBIRQ**, is asserted:
  - **PMBIRQ** is a level-sensitive interrupt request driven by PMBSR_EL1.S. This means that a direct write that sets PMBSR_EL1.S to 1 causes the interrupt to be asserted, and **PMBIRQ** remains asserted until software clears PMBSR_EL1.S to 0.
  - If a Generic Interrupt Controller (GIC) is implemented, **PMBIRQ** must be configured as a Private Peripheral Interrupt (PPI) in a multiprocessor system. **PMBIRQ** is signaled by the PE that implements the Statistical Profiling Extension.

    Note

    A standard PPI number is allocated by the *ARM®Server Base System Architecture* (SBSA).

- Additional syndrome for the event is written to PMBSR_EL1.MSS. Unless otherwise stated in the event description, other PMBSR_EL1 fields are unchanged.

While PMBSR_EL1.S is set to 1:

- The buffer is disabled and profiling is disabled.
- All remaining buffered sample records are discarded.
- The values in PMBPTR_EL1 are retained and PMSICR_EL1 does not decrement.

Buffer full events and MMU fault Profiling Buffer management events are reported synchronously.

Note

Reported synchronously means that profiling is disabled before the SPE samples further operations. The interrupt exception resulting from asserting the Profiling Buffer interrupt request is an asynchronous exception.

It is IMPLEMENTATION DEFINED whether external aborts are reported to the Statistical Profiling Extension synchronously or asynchronously. If external aborts are reported as asynchronous:

- The external abort might not be received until after a first Profiling Buffer management event has set PMBSR_EL1.S to 1.
- Writes to the buffer might generate a second Profiling Buffer management event after the external abort has set PMBSR_EL1.S to 1.

The architecture does not require that a sample record is written sequentially by the Statistical Profiling Extension, only that:

- The Statistical Profiling Extension never writes past the PMBLIMITR_EL1 limit pointer.
- On a Profiling Buffer management interrupt, PMBSR_EL1.DL indicates whether PMBPTR_EL1 points to the first byte after the last complete sample record.
- On an MMU fault or synchronous external abort, PMBPTR_EL1 serves as a Fault Address Register.

Note

- This means that it must not be assumed that:
  - There is ever any valid data beyond the current PMBPTR_EL1 write pointer.
  - The PE has not written a valid sample record between the current PMBPTR_EL1 write pointer and the PMBLIMITR_EL1 limit pointer.
  - If PMBSR_EL1.DL is set to 1 on a Profiling Buffer management interrupt, that there is any valid data between the end of the last complete sample record and the current PMBPTR_EL1 write pointer.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

  — *Any* valid data has been written to the Profiling Buffer if an external abort is reported asynchronously to the Statistical Profiling Extension.

- The last complete sample record must end at most $2^{(PMSIDR_EL1.MaxSize)}$ bytes below PMBPTR_EL1.

D9.8.1 Prioritization of Profiling Buffer management events

Where multiple synchronous Profiling Buffer management events occur on writing a sample record, the PE prioritizes them as follows (from highest to lowest priority):

1. Synchronous fault.
2. Synchronous external abort.
3. Buffer full event.

Asynchronous external aborts are not prioritized with respect to other events.

Note

Prioritization of Profiling Buffer management interrupt requests is managed by the interrupt controller. Profiling Buffer management events are prioritized internally by the PE.

D9.8.2 Buffer full event

If, after writing a sample record, there is not sufficient space in the Profiling Buffer for a sample record of the size indicated by PMSIDR_EL1.MaxSize, and PMBSR_EL1.S is 0, a Profiling Buffer management event is generated:

- PMBSR_EL1.EC is set to `0b000000`, other buffer management event.
- The BSC field of PMBSR_EL1.MSS is set as follows:
  — PMBSR_EL1.BSC is set to `0b000001`, buffer filled.
- PMBPTR_EL1 is set to the first byte after the last complete sample record. PMBSR_EL1.DL is unchanged.
- The other PMBSR_EL1 fields are unchanged.

That is, the Profiling Buffer management event is generated when the PE writes past the write limit pointer minus $2^{(PMSIDR_EL1.MaxSize)}$. The Statistical Profiling Extension never writes beyond the write limit pointer.

For more information see *Restrictions on the current write pointer* on page D9-2799.

D9.8.3 Faults and watchpoints

Table D9-5 lists the faults that might be generated by a write to the Profiling Buffer by the Statistical Profiling Extension.

Writes to the Profiling Buffer never generate watchpoints.

Table D9-5 Faults

Fault	Conditions
Translation	The translation of a virtual address to a physical address might generate a Translation fault.
Address Size	The translation of a virtual address to a physical address might generate an Address Size fault.
Alignment	If PMBPTR_EL1 is not aligned to an IMPLEMENTATION DEFINED minimum alignment, the behavior is UNPREDICTABLE and a write to the Profiling Buffer by the Statistical Profiling Extension might generate an Alignment fault. For more information see *Restrictions on the current write pointer* on page D9-2799.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D9-5 Faults (continued)

Fault	Conditions
Permission	Writes to the Profiling Buffer are made as privileged writes. If the write does not have write permission, a Permission fault is generated. The value of PSTATE.PAN is ignored and treated as zero. If the Statistical Profiling Extension does not manage the dirty state in translation tables, then accesses ignore the Dirty Bit Modifier bit in Page and Block descriptors and an access might as a result generate a Permission fault. For more information see *The dirty state* on page D5-2618.[a]
Access Flag	If the Statistical Profiling Extension does not manage the Access Flag in translation tables or hardware management of the Access Flag state is disabled for the owning translation regime, then any access to a Page or Block with the Access Flag bit set to 0 in a descriptor will generate an Access Flag fault. For more information see *The Access flag* on page D5-2618.[a]
TLB Conflict fault	IMPLEMENTATION DEFINED.
Unsupported atomic hardware update fault	If hardware update of the translation tables is not guaranteed atomic in regard to other agents that access the memory, the translation of a virtual address to a physical address might generate an Unsupported atomic hardware update fault.

a. PMBIDR_EL1.F defines whether the Statistical Profiling Extension manages the Access Flag and dirty state in the translation tables.

If a write to the Profiling Buffer generates a fault and PMBSR_EL1.S is 0, then a Profiling Buffer management event is generated:

- PMBSR_EL1.S is set to 1.
- PMBSR_EL1.EC is set to one of:
  - `0b100100`, stage 1 Data Abort on write to the Profiling Buffer.
  - `0b100101`, stage 2 Data Abort on write to the Profiling Buffer.
- The FSC field of PMBSR_EL1.MSS is set as follows:
  - PMBSR_EL1.FSC is set to indicate the type of the fault.
- PMBPTR_EL1 is set to the address that generated the fault.
- If PMBPTR_EL1 is not the address of the first byte after the last complete sample record written by the Statistical Profiling Extension, then PMBSR_EL1.DL is set to 1. Otherwise, PMBSR_EL1.DL is unchanged.
- The other PMBSR_EL1 fields are unchanged.

Note

Each of these faults gives rise to a Profiling Buffer management interrupt, not an actual MMU fault exception. The ESR and FAR registers are unchanged.

For more information see *The MMU fault-checking sequence* on page D5-2655.

Hardware management of dirty state and the Access flag by the Statistical Profiling Extension

It is IMPLEMENTATION DEFINED whether address translations performed by the Statistical Profiling Extension manage dirty state and the Access flag. This is discoverable by software using PMBIDR_EL1.F. See *Hardware management of dirty state* on page D5-2620 and *Hardware management of the Access flag* on page D5-2619.

If hardware management of dirty state by the Statistical Profiling Extension is implemented, and hardware management of dirty state is enabled for the owning translation regime, then the Statistical Profiling Extension can speculatively update the translation table descriptor for any Page or Block in the Statistical Profiling buffer before writing data to it, if the write is otherwise permitted. This includes the case where a buffer management event means the Statistical Profiling Extension stops writing data before the page or block is written to.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D9.8.4 External aborts

A write to the Profiling Buffer might generate an external abort, including an external abort on a translation table walk or translation table update. It is an IMPLEMENTATION DEFINED choice whether such an external abort:

- Is reported to the Statistical Profiling Extension and treated as a Profiling Buffer management event.
- Generates an SError interrupt exception.

If a write to the Profiling Buffer generates an external abort that is reported to the Statistical Profiling Extension:

- The external abort bit, PMBSR_EL1.EA, is set to 1.
- The Statistical Profiling Extension stops writing sample records to the Profiling Buffer. It is implementation defined whether an external abort on a write to the Profiling Buffer is reported as synchronous or asynchronous:
  - The external abort is reported as *synchronous* if PMBPTR_EL1 is set to the address that was externally aborted.
  - The external abort is reported as *asynchronous* if PMBPTR_EL1 is not guaranteed to be set to the address that was externally aborted.
- If the external abort is reported as asynchronous or PMBPTR_EL1 is not the address of the first byte of the sample record being written by the Statistical Profiling Extension, then PMBSR_EL1.DL is set to 1. Otherwise PMBSR_EL1.DL is unchanged.

  Note

  Following an external abort reported asynchronously to the Statistical Profiling Extension, software must not assume that any valid data has been written to the Profiling Buffer.

- The other PMBSR_EL1 fields are unchanged.

If a write to the Profiling Buffer generates an external abort that is taken as an SError interrupt exception, the PE takes the SError interrupt exception as normal, and PMBSR_EL1 fields are unchanged.

Note

Treating the external abort as a Profiling Buffer management event:

- Sets PMBSR_EL1.S to 1 and so disables the Statistical Profiling Extension.
- Allows error recovery software to isolate the event to the actions of the Statistical Profiling Extension.

Taking an SError interrupt:

- Means that the Statistical Profiling Extension will only be disabled if the SError interrupt is taken to an Exception level where the Statistical Profiling Extension is disabled.
- Might not allow error recovery software to isolate the event and error containment.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D9.9 Synchronization and Statistical Profiling

The profiling operation of the Statistical Profiling Extension:

- Makes indirect reads and indirect writes of System registers.
- Writes to memory.
- Makes further indirect writes to PMBPTR_EL1 as a result of an external abort on a write to memory.

The indirect reads of the PMSCR_EL1.{E1SPE, E0SPE} and PMSCR_EL2.{E2SPE, E0HSPE} controls when determining whether to select an operation for profiling are treated as indirect reads made by the instruction being executed, and subject to the standard requirements for synchronization.

Otherwise, although the profiling operation is generated by a sampled operation, the profiling operation executes independently of the instructions that are executed on the PE, and acts as a separate memory observer from the PE in the system.

A DSB instruction guarantees that all memory transactions that are made by the PE are observable by writes made by a profiling operation relating to a sampled operation in program order after the DSB instruction.

A Context synchronization event guarantees that a direct write to a System register made by the PE in program order before the context synchronization event are observable by indirect reads and indirect writes of the same System register made by a profiling operation relating to a sampled operation in program order after the context synchronization event.

To synchronize previous profiling operations, software must execute a PSB CSYNC Buffer Synchronization instruction.

Note

The PSB CSYNC instruction is not defined in the AArch32 instruction set architecture.

Following a context synchronization event, a PSB CSYNC instruction is guaranteed to synchronize the profiling operations for all instructions that are executed in program order before the context synchronization event.

Synchronized by the PSB CSYNC instruction means:

- A direct read of a System register in program order following a PSB CSYNC instruction requires explicit synchronization to observe an indirect write to the same System register made by a profiling operation synchronized by the PSB CSYNC instruction.
- An indirect write to a System register made by a profiling operation synchronized by a PSB CSYNC instruction does not affect a direct write to the same System register made in program order following the PSB CSYNC instruction.
- A direct write to a System register in program order following a PSB CSYNC instruction is not allowed to affect an indirect read of the same System register made by a profiling operation synchronized by the PSB CSYNC instruction.
- A DSB instruction in program order following a PSB CSYNC instruction does not complete before the writes to the Profiling Buffer of sample records for profiling operations synchronized by the PSB CSYNC instruction have completed. The DSB instruction must apply to both loads and stores.

For the indirect write to PMBSR_EL1 that is made as a result of an external abort on a write of a sample record to memory, the synchronization rules apply only after the write has completed.

Although the Statistical Profiling Extension acts as another observer in the system, for determining the Shareability domain of the DSB instructions, the writes of sample records are treated as coming from the PE that is being profiled.

Note

If the Statistical Profiling Extension is not disabled when the context synchronization event occurs, further profiling operations might be generated that are not guaranteed to be synchronized by the PSB CSYNC instruction.

If the PE takes an exception to an Exception level where the Statistical Profiling Extension is disabled, no new operations are selected for sampling. The Statistical Profiling Extension is always disabled if the owning Exception level is a lower Exception level than the current Exception level.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In the absence of a context synchronization event, a `PSB CSYNC` instruction is not required to execute in program order with respect to sampled operations.

D9.9.1 UNPREDICTABLE behavior

In the absence of correct context synchronization events, it is UNPREDICTABLE whether an indirect read of a System register made by a profiling operation will return the old or the new values.

If the indirect reads mean that `ProfilingBufferEnabled()` returns FALSE when a sample record or records are about to be written to the physical address, then it is further unpredictable whether the sample record or records:

- Are written to memory.
- Are silently discarded and not written to memory.
- Are discarded and not written to memory, and a Profiling Buffer management event is generated:
  - PMBSR_EL1.DL is set to 1.
  - PMBSR_EL1.EC is set to `0x00`.
  - PMBSR_EL1.BSC is set to `0x00` to indicate that the buffer is not full.

This means that software must execute a `PSB CSYNC` instruction to force any sample records to be written to the Profiling Buffer before changing context.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter D10
Statistical Profiling Extension Sample Record Specification

This chapter describes the sample records generated by the Statistical Profiling Extension. It contains the following sections:

- *About the Statistical Profiling Extension Sample Records* on page D10-2810.
- *Alphabetical list of Statistical Profiling Extension packets* on page D10-2813.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D10.1 About the Statistical Profiling Extension Sample Records

The sample record format is self-describing and extensible. This format allows software to parse profile data even when that profile data contains extended information.

The Statistical Profiling Extension writes a series of sample records to memory, each record consisting of a sequence of packets, and each packet consisting of:

- One or two header bytes.
- Zero, 1, 2, 4 or 8 payload bytes.

D10.1.1 Headers

The first header byte encodes the number of payload bytes:

`0x00-0x1F` Single byte header, no payload.

`0x20-0x3F` First byte of extended header. Second byte encodes the payload length.

`0x40-0x4F`, `0x80-0x8F`, `0xC0-0xCF`
Header with an 8-bit payload.

`0x50-0x5F`, `0x90-0x9F`, `0xD0-0xDF`
Header with a 16-bit payload.

`0x60-0x6F`, `0xA0-0xAF`, `0xE0-0xEF`
Header with a 32-bit payload.

`0x70-0x7F`, `0xB0-0xBF`, `0xF0-0xFF`
Header with a 64-bit payload.

D10.1.2 Records

A record consists of multiple packets. A record comprises, in ascending address order:

- A sequence of headers, each followed by their payload byte or bytes.
- Either:
  - An End packet header.
  - A Timestamp packet.

Figures in this chapter show each packet as a sequence of bytes. Figure D10-1 shows how bytes are stored in memory in increasing addresses from left to right.

<table>
<tr><td>First byte</td><td>1</td><td>2</td><td>3</td><td>4</td><td>Last Byte</td></tr>
<tr><td rowspan="2">Header
(16-bit data)</td><td colspan="2">Data</td><td rowspan="2">Header
(8-bit data)</td><td rowspan="2">Data</td><td rowspan="2">0x01
End Packet</td></tr>
<tr><td>LSB</td><td>MSB</td></tr>
</table>

<table>
<tr><td>First Byte</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>...</td><td>12</td><td>Last Byte</td></tr>
<tr><td rowspan="2">Header
(16-bit data)</td><td colspan="2">Data</td><td rowspan="2">Header
(8-bit data)</td><td rowspan="2">Data</td><td rowspan="2">0x71
Timestamp
Packet</td><td rowspan="2">TS [7:0]</td><td rowspan="2">...</td><td rowspan="2">TS [55:48]</td><td rowspan="2">TS [63:56]</td></tr>
<tr><td>LSB</td><td>MSB</td></tr>
</table>

Figure D10-1 Convention for packet descriptions

In some sections, the figures are split into separate figures for the header byte and payload bytes. For instance, where the number of payload bytes varies according to a field in the header.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D10.1.3 Byte order

Header bytes and payload bytes are written in ascending address order. Within a payload value, values are written in little-endian byte order.

The size of the access granule for writes to the Profiling Buffer by the Statistical Profiling Extension is IMPLEMENTATION DEFINED, up to a maximum of 2KB. The size of the access granule can vary from time to time.

Note

This means that if the memory type accessed is non-Gathering Device, the architecture does not require a specific access granule size at the end device.

D10.1.4 Protocol framing packets and forwards compatibility

The padding header, alignment command, timestamp packet, and end packet are protocol framing packets that frame the records created by the Statistical Profiling Extension. Only padding headers and alignment commands are permitted between records.

Note

PMBIDR_EL1.Align defines a minimum alignment for records. However, implementations must nevertheless create a valid protocol stream that can be parsed without knowledge of the minimum alignment.

The packet types are described in the following sections. Software must ignore unknown packets, using the size field encoded in the header. This includes packets containing reserved values in fields.

The following sections give an overview of the Statistical Profiling Extension packets output to a memory-mapped Profiling Buffer or Device memory:

- *Statistical Profiling Extension protocol packet headers*

D10.1.5 Statistical Profiling Extension protocol packet headers

8-bit headers

For Address packets and Counter packets, the 8-bit header format is described as the *short format*.

Table D10-1 8-bit header encodings

[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]	Description
0	0	0	0	0	0	0	0	*Padding* on page D10-2834
0	0	0	0	0	0	0	1	*End packet* on page D10-2823
0	1	1	1	0	0	0	1	*Timestamp packet* on page D10-2835
0	1	x	x	0	0	1	0	*Events packet* on page D10-2824
0	1	x	x	0	0	1	1	*Data Source packet* on page D10-2822
0	1	1	0	0	1	x	x	*Context packet* on page D10-2818
0	1	0	0	1	0	x	x	*Operation Type packet* on page D10-2829
1	0	1	1	0	x	x	x	*Address packet* on page D10-2813 (Short format)
1	0	0	1	1	x	x	x	*Counter packet* on page D10-2819 (Short format)

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

16-bit headers

For Address packets and Counter packets, the 16-bit header format is described as the *extended format*.

Table D10-2 16-bit header encodings

Byte 0								Byte 1								Description
[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]	
0	0	1	0	0	0	x	x	1	0	1	1	0	x	x	x	*Address packet* on page D10-2813
0	0	1	0	0	0	x	x	1	0	0	1	1	x	x	x	*Counter packet* on page D10-2819

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D10.2 Alphabetical list of Statistical Profiling Extension packets

D10.2.1 Address packet

The Address packet characteristics are:

Purpose Provides an address value for the record. Addresses are always 64 bits.

Attributes Multi-part packet comprising:

- 8 or 16-bit header.
- 64-bit payload.

Address packet header

When Extended format is used, the Address packet header bit assignments are:

	7	6	5	4	3	2	1	0	
	0	0	1	0	0	0	INDEX[4:3]		Byte 0
	1	0	SZ 1	1	0	INDEX[2:0]			Byte 1

When Short format is used, the Address packet header bit assignments are:

	7	6	5	4	3	2	1	0	
	1	0	SZ 1	1	0	INDEX			Byte 0

Byte 1 bits [7:6], when Extended format, Byte 0 bits [7:6], when Short format

This field reads as `0b10`.

SZ, byte 1 bits [5:4], when Extended format, SZ, byte 0 bits [5:4], when Short format

Payload size. The defined values of this field are:

`0b11` Doubleword.

This field reads as `0b11`.

Byte 1 bit [3], when Extended format, Byte 0 bit [3], when Short format

This bit reads as `0b0`.

Byte 0 bits [7:5], when Extended format

This field reads as `0b001`.

Byte 0 bits [4:2], when Extended format

This field reads-as-zero.

INDEX, byte 0 bits [1:0], byte 1 bits [2:0], when Extended format, INDEX, byte 0 bits [2:0], when Short format

The defined values of this field are:

`0b00000` Issued instruction virtual address (PC). Included for all operations.

`0b00001` Branch target address:

- It is IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether this address is included for an Exception Return to an Exception level where profiling is disabled.
- Included for all other branch and exception return instructions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b00010` Data access virtual address. Included for all load, store and atomic operations.

`0b00011` Data access physical address:

- It is IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether this address included for accesses that generate Permission or Access Flag faults.
- Not included for all other accesses that generate an abort, or if disabled by CollectPhysicalAddress.
- Included for all other load, store and atomic operations.

`0b0011x` IMPLEMENTATION DEFINED address.

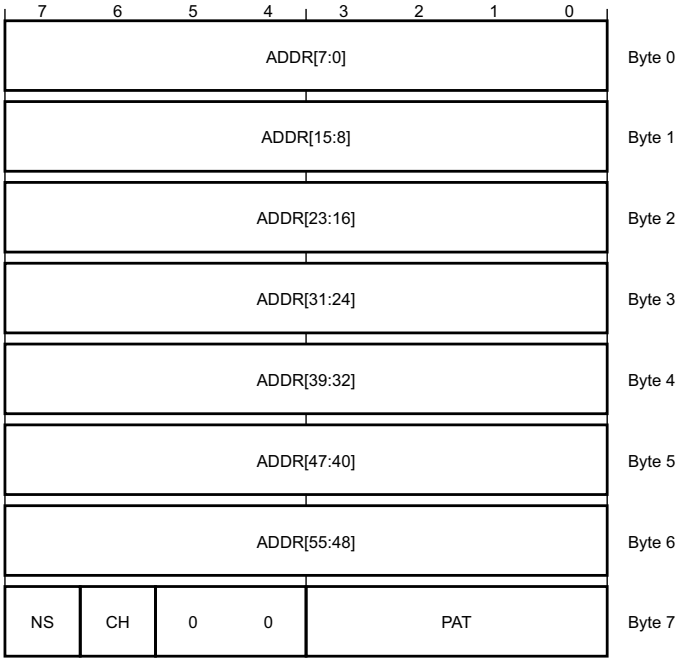
`0b1xxxx` IMPLEMENTATION DEFINED address.

All other values are reserved.

In the Short format header, bits [4:3] are zero.

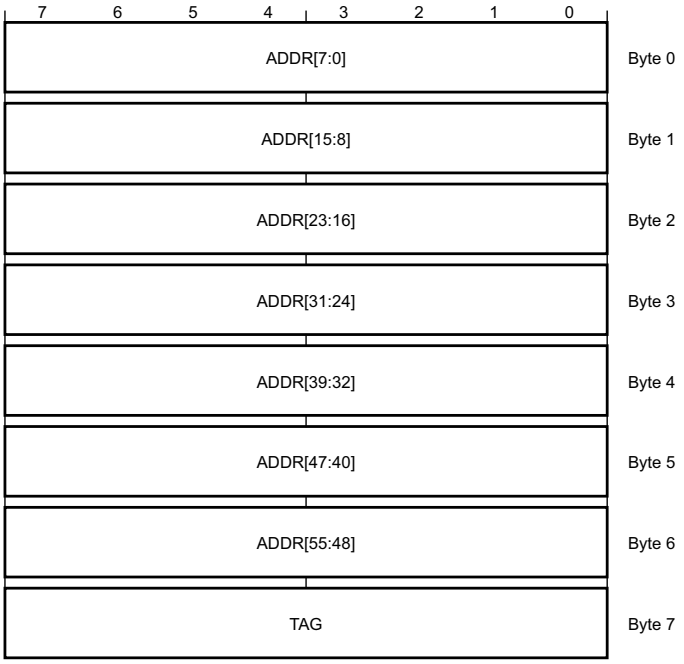
Address packet payload

When Data access physical address, the Address packet payload bit assignments are:

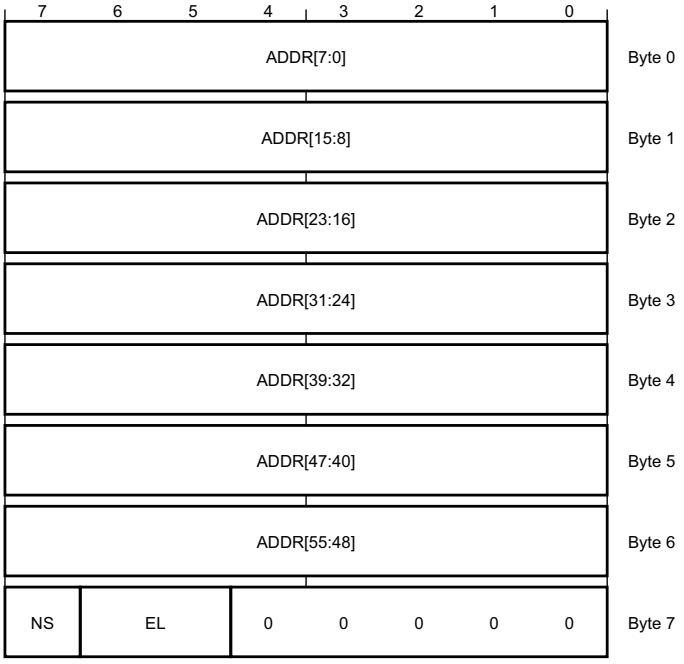


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When Data access virtual address, the Address packet payload bit assignments are:



When Instruction virtual address, the Address packet payload bit assignments are:



TAG byte <7>, when Data access virtual address

Top-byte tag.

If the value of the applicable TBI bit is one, a data access virtual address includes the top-byte tag. If the applicable TBI bit is zero, it is IMPLEMENTATION DEFINED whether this field reads as zero or holds the address tag of the applicable address.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

NS, byte 7 bit [7], when Instruction virtual address

Non-secure state. The Security state associated with the address. For an issued instruction virtual address (PC) this is the Security state the instruction was executed in. For a branch target address, this is the Security state at the target of the branch. The defined values of this bit are:

`0` Secure state.

`1` Non-secure state.

Note

For an Exception Return, the Security state at the target of the branch might be different to the Security state the instruction was executed in.

NS, byte 7 bit [7], when Data access physical address

Physical address space identifier. The Security attribute for the physical address. The defined values of this bit are:

`0` Secure physical address space.

`1` Non-secure physical address space.

CH, byte 7 bit[6], when Data access physical address

When ARMv8.5-MemTag is implemented, Checked access identifier. Checked or Unchecked access. The defined values of this bit are:

`0` Unchecked access.

`1` Checked access.

For more information see Chapter D6 *Armv8.5 Memory Tagging Extension*.

If ARMv8.5-MemTag is not implemented this bit is RAZ.

EL, byte 7 bits [6:5], when Instruction virtual address

Exception level. The Exception level associated with the address. For an issued instruction virtual address (PC) this is the Exception level the instruction was executed in. For a branch target address, this is the Exception level at the target of the branch. The defined values of this field are:

`0b00` EL0.

`0b01` EL1.

`0b10` EL2.

`0b11` EL3.

Note

For an Exception Return, the Exception level at the target of the branch might be different to the Exception level the instruction was executed in.

Byte 7 bits [5:4], when Data access physical address

This field reads as `0b00`.

PAT, Byte 7 bits [3:0], when Data access physical address

When ARMv8.5-MemTag is implemented, this field provides the Physical Address Tag for a Checked access. If the access is Unchecked this field reads as an IMPLEMENTATION DEFINED choice between `0b0000` and the Physical Address Tag used to perform the access.

For more information see Chapter D6 *Armv8.5 Memory Tagging Extension*.

If ARMv8.5-MemTag is not implemented this field is RAZ.

Byte 7 bits [4:0], when Instruction virtual address

This field reads as `0b00000`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ADDR, bytes <6:0>

Address. Bits [55:0] of the address.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D10.2.2 Context packet

The Context packet characteristics are:

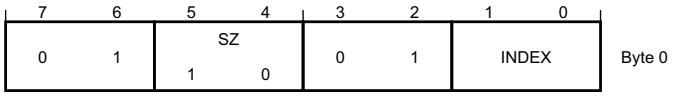
Purpose Provides context information for the record.

Attributes Multi-part packet comprising:

- 8-bit header.
- 32-bit payload.

Context packet header

The Context packet header bit assignments are:



Byte 0 bits [7:6]

This field reads as `0b01`.

SZ, byte 0 bits [5:4]

Payload size. The defined values of this field are:

`0b10` Word.

This field reads as `0b10`.

Byte 0 bits [3:2]

This field reads as `0b01`.

INDEX, byte 0 bits [1:0]

Identifies the context value. The defined values of this field are:

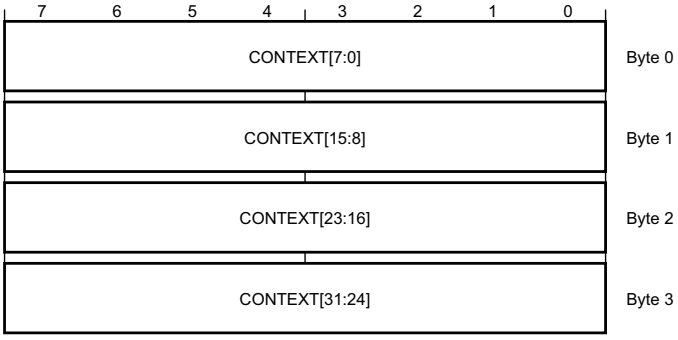
`0b00` CONTEXTIDR_EL1. Included for all operations if enabled by CollectContextIDR1.

`0b01` CONTEXTIDR_EL2. Included for all operations if enabled by CollectContextIDR2.

All other values are reserved.

Context packet payload

The Context packet payload bit assignments are:



CONTEXT, bytes <3:0>

The context value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D10.2.3 Counter packet

The Counter packet characteristics are:

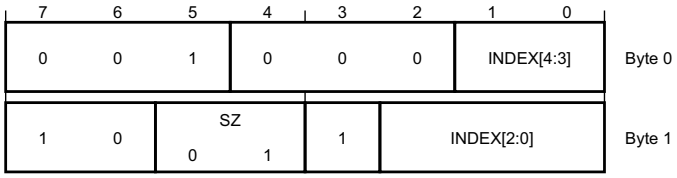
Purpose Count of cycles the operation spent performing all or part of its behavior. The counter value occupies the least significant bits of the payload. The remaining bits are set to zero.

Attributes Multi-part packet comprising:

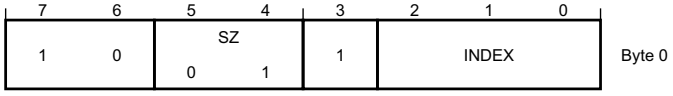
- 8 or 16-bit header.
- 16-bit payload.

Counter packet header

When Extended format, the Counter packet header bit assignments are:



When Short format, the Counter packet header bit assignments are:



Byte 1 bits [7:6], when Extended format, Byte 0 bits [7:6], when Short format

This field reads as `0b10`.

SZ, byte 1 bits [5:4], when Extended format, SZ, byte 0 bits [5:4], when Short format

Payload size. The defined values of this field are:

`0b01` Halfword.

This field reads as `0b01`.

Byte 1 bit [3], when Extended format, Byte 0 bit [3], when Short format

This bit reads as `0b1`.

Byte 0 bits [7:5], when Extended format

This field reads as `0b001`.

Byte 0 bits [4:2], when Extended format

This field reads-as-zero.

INDEX, byte 0 bits [1:0], byte 1 bits [2:0], when Extended format, INDEX, byte 0 bits [2:0], when Short format

The defined values of this field are:

`0b00000` Total latency. Cycle count from the operation being dispatched for issue to the operation being complete. Included for all operations.

`0b00001` Issue latency. Cycle count from the operation being dispatched for issue to the operation being issued for execution. This counts any delay in waiting the operation being ready to issue. Included for all operations.

`0b00010` Translation latency. Cycle count from a virtual address being passed to the MMU for translation to the result of the translation being available. Included for all load, store and atomic operations.

`0b0011x` IMPLEMENTATION DEFINED counter value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1xxxx` IMPLEMENTATION DEFINED counter value.

All other values are reserved.

In the Short format header, bits [4:3] are zero.

Dispatched for issue means:

- The operation has been decoded.
- The operation might not be ready to start execution because it is waiting for input values. The operation might be put into a queue.

Issued for execution means the operation is ready to start executing:

- For example, for a memory operation, this should be indicative of the cycle count from memory operation being dispatched for issue to access being initiated (virtual address).

Complete means:

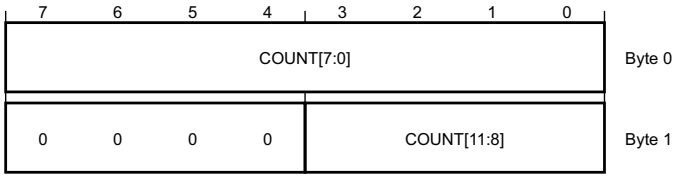
- The operation has completed execution and is no longer capable of stalling any instruction that consumes its output.
- It is IMPLEMENTATION DEFINED whether the operation has committed its results to the architectural state of the PE.
- For example:
  - For an arithmetic, floating-point, or SIMD operation with variable timing, such as divide, the results of the operation are available.
  - For load and atomic operations that return data, all data have been returned from memory.
  - For store and atomic operations that do not return data, it is not required that the store has reached its end point for it to be complete.
  - For branch operations, the branch has been resolved as taken or not taken.
  - For barrier operations, the barrier has completed.

For WFE and WFI operations, it is IMPLEMENTATION DEFINED whether:

- The instruction is complete before the PE enters a low-power state or when the PE wakes from the low-power state.
- Counters count in the low power state.
- Sampling an operation is itself a wake-up event.

Counter packet payload

The Counter packet payload bit assignments are:



Byte 1 bits [7:4]

This field reads-as-zero.

COUNT, byte 1 bits [3:0], byte <0>, when a 12-bit counter is implemented

The counter value occupies the least significant bits of the payload. The remaining bits are set to zero. The counters are:

- Unsigned numbers.
- 12 bits.
- Saturating.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The value `0xFFF` indicates the count has saturated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D10.2.4 Data Source packet

The Data Source packet characteristics are:

Purpose

If the implementation includes support for indicating the loaded data source, the Data Source packet indicates where the data returned for a load operation was sourced. It might also include other information, such as the state of the data at the source. It is IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether this is included for load and atomic operations that generate an external abort. It is IMPLEMENTATION DEFINED whether this is included for atomic operations that do not return data to a PE register. Included for all other load and atomic operations.

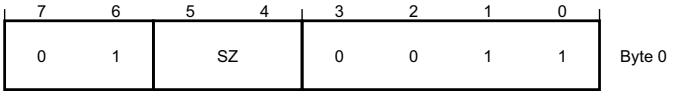
Attributes

Multi-part packet comprising:

- 8-bit header.
- 8 or 16-bit payload.

Data Source packet header

The Data Source packet header bit assignments are:



Byte 0 bits [7:6]

This field reads as `0b01`.

SZ, byte 0 bits [5:4]

Payload size. The defined values of this field are:

`0b00` Byte.

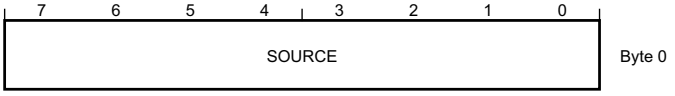
`0b01` Halfword.

Byte 0 bits [3:0]

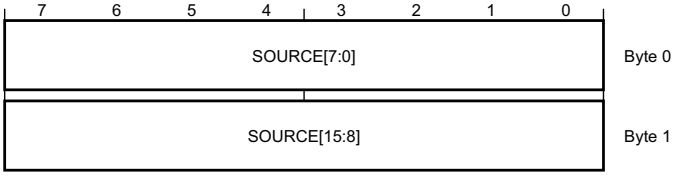
This field reads as `0b0011`.

Data Source packet payload

When SZ == `0b00`, the Data Source packet payload bit assignments are:



When SZ == `0b01`, the Data Source packet payload bit assignments are:



SOURCE, byte <0>, when SZ == `0b00`, SOURCE, bytes <1:0>, when SZ == `0b01`

Because the list of data sources varies from system to system, the definition of this field is IMPLEMENTATION DEFINED. If a sampled operation generated multiple data accesses, it is IMPLEMENTATION DEFINED how the data source information is combined.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D10.2.5 End packet

The End packet characteristics are:

Purpose Defines the end of a record if a Timestamp packet is not present.

Attributes 8-bit packet.

Field descriptions

The End packet bit assignments are:

7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	1	Byte 0

Byte <0>

This field reads as `0b00000001`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D10.2.6 Events packet

The Events packet characteristics are:

Purpose

Indicates up to 64 events generated by the sampled operation. If ARMv8.1-PMU is implemented and an event counter is configured to count PMU events, then a sampled operation that causes the event counter to be incremented has the event recorded as one, and conversely a sampled operation that does not cause the counter to be incremented is recorded as zero.

Note

Arm recommends that the Performance Monitors Extension implements the Events.

Attributes

Multi-part packet comprising:

- 8-bit header.
- 8, 16, 32, or 64-bit payload.

Events packet header

The Events packet header bit assignments are:

7	6	5	4	3	2	1	0	
0	1	SZ		0	0	1	0	Byte 0

Byte 0 bits [7:6]

This field reads as `0b01`.

SZ, byte 0 bits [5:4]

Payload size. The defined values of this field are:

`0b00` Byte.

`0b01` Halfword.

`0b10` Word.

`0b11` Doubleword.

Software must treat bits that are not output as zero.

Byte 0 bits [3:0]

This field reads as `0b0010`.

Events packet payload

When SZ == `0b00`, the Events packet payload bit assignments are:

7	6	5	4	3	2	1	0	
E[7]	E[6]	E[5]	E[4]	E[3]	E[2]	E[1]	E[0]	Byte 0

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When SZ == `0b01`, the Events packet payload bit assignments are:

7	6	5	4	3	2	1	0	
E[7]	E[6]	E[5]	E[4]	E[3]	E[2]	E[1]	E[0]	Byte 0
E[15:12]				E[11]	E[10]	E[9]	E[8]	Byte 1

When SZ == `0b10`, the Events packet payload bit assignments are:

7	6	5	4	3	2	1	0	
E[7]	E[6]	E[5]	E[4]	E[3]	E[2]	E[1]	E[0]	Byte 0
E[15:12]				E[11]	E[10]	E[9]	E[8]	Byte 1
0	0	0	0	0	E[18]	E[17]	E[16]	Byte 2
E[31:24]								Byte 3

When SZ == `0b11`, the Events packet payload bit assignments are:

7	6	5	4	3	2	1	0	
E[7]	E[6]	E[5]	E[4]	E[3]	E[2]	E[1]	E[0]	Byte 0
E[15:12]				E[11]	E[10]	E[9]	E[8]	Byte 1
0	0	0	0	0	E[18]	E[17]	E[16]	Byte 2
E[31:24]								Byte 3
0	0	0	0	0	0	0	0	Byte 4
0	0	0	0	0	0	0	0	Byte 5
E[55:48]								Byte 6
E[63:56]								Byte 7

E[63:48], bytes <7:6>, when SZ == `0b11`

Events 63 to 48. IMPLEMENTATION DEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bytes <5:4,2>, byte 1 bit [3], when SZ == `0b11`

This field reads-as-zero.

E[31:24], byte <3>, when SZ == `0b10`, or when SZ == `0b11`

Events 31 to 24. IMPLEMENTATION DEFINED.

E[18], byte 2 bit [18], when SZ == `0b10`, or SZ == `0b11`

Empty predicate.

When The Scalable Vector Extension (SVE) and ARMv8.3-SPE are implemented the defined values of this bit are:

`0` Operation was not an SVE operation, was unpredicated or executed with all elements Active.

`1` SVE operation executed with all elements Inactive.

Otherwise this bit reads-as-zero.

If PMUv3 and The Scalable Vector Extension (SVE) are implemented this Event is required to be implemented consistently with SVE_PRED_EMPTY_SPEC in the *Arm Architecture Reference Manual Supplement, the Scalable Vector Extension, for v8-A*.

E[17], byte 2 bit [17], when SZ == `0b10`, or SZ == `0b11`

Partial predicate.

When The Scalable Vector Extension (SVE) and ARMv8.3-SPE are implemented the defined values of this bit are:

`0` Operation was not an SVE operation, was unpredicated, or executed with all elements Active.

`1` Predicated SVE operation executed with at least one Inactive element.

Otherwise this bit reads-as-zero.

If PMUv3 and The Scalable Vector Extension (SVE) are implemented this Event is required to be implemented consistently with SVE_PRED_EMPTY_SPEC and SVE_PRED_PARTIAL_SPEC in the *Arm Architecture Reference Manual Supplement, the Scalable Vector Extension, for v8-A*.

E[15:12], byte 1 bits [7:4], when SZ == `0b01`, when SZ == `0b10`, or when SZ == `0b11`

Events 15 to 12. IMPLEMENTATION DEFINED.

E[11], byte 1, bit [11], when SZ == `0b10`, or SZ == `0b11`

Alignment.

When ARMv8.3-SPE is implemented the defined values of this bit are:

`0` Load/store operation that was optimally aligned for the size of data being accessed.

`1` Load/store operation that, due to the alignment of the address and size of data being accessed, incurred additional latency.

Otherwise this bit reads-as-zero.

If PMUv3 is implemented this Event is required to be implemented consistently with LDST_ALIGN_LAT.

Byte 1 bit [3], when SZ == `0b01`

This bit reads-as-zero.

E[10], byte 1 bit [2], when SZ == `0b01`, when SZ == `0b10`, or when SZ == `0b11`

Remote access. The defined values of this bit are:

`0` Did not cause access to another socket.

`1` Load/store operation caused an access to another socket in a multi-socket system. This includes each data memory access that accesses another socket in a multi-socket system, including those that do not return data.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If PMUv3 is implemented this Event is required to be implemented consistently with REMOTE_ACCESS.

E[9], byte 1 bit [1], when SZ == `0b01`, when SZ == `0b10`, or when SZ == `0b11`

Last Level cache miss. The defined values of this bit are:

`0` Did not miss Last Level cache.

`1` Load/store operation caused an access to at least the Last Level cache but is not completed by the Last Level cache. That is, each:

- Load operation that does not return data from the Last Level cache.
- Store operation that does not update the Last Level cache.

The event is not set for operations that are completed by a cache above the Last Level cache.

If PMUv3 is implemented this Event is required to be implemented consistently with LL_CACHE_MISS.

E[8], byte 1 bit [0], when SZ == `0b01`, when SZ == `0b10`, or when SZ == `0b11`

Last Level cache access. The defined values of this bit are:

`0` Did not access Last Level data or unified cache.

`1` Load/store operation caused a cache access to at least the Last Level data or unified cache.

Note

The architecture does not define the Last Level cache. The Last Level cache is typically the largest cache on this device shared by all PEs in the inner or outer Shareable domain of this PE. In a multi-socket system, it is IMPLEMENTATION DEFINED whether this includes caches on other sockets.

If PMUv3 is implemented this Event is required to be implemented consistently with LL_CACHE.

E[7], byte 0 bit [7]

Mispredicted. The defined values of this bit are:

`0` Did not cause correction to the predicted program flow.

`1` A branch that caused a correction to the predicted program flow.

If PMUv3 is implemented this Event is required to be implemented consistently with either BR_MIS_PRED or BR_MIS_PRED_RETIRED.

E[6], byte 0 bit [6]

Not taken. The defined values of this bit are:

`0` Did not fail condition code check.

`1` A conditional instruction that failed its condition code check. This includes conditional branches, compare-and-branch, conditional select, and conditional compares:

- For a conditional branch or compare-and-branch instruction, this means the branch was not taken.
- For a conditional select, this means the second operand was written to the result.
- For a condition compare, this means the condition flags were set to the immediate value and not the result of the compare.

Note

This Event includes branches, selects, `CCMP (register)`, and `CCMP (immediate)`.

E[5], byte 0 bit [5]

TLB walk. The defined values of this bit are:

`0` Did not generate TLB walk.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`1` Load/store operation that causes a refill of a data or unified TLB, involving at least one translation table walk access. This includes each complete or partial translation table walk that causes an access to memory, including to data or translation table walk caches.

If PMUv3 is implemented this Event is required to be implemented consistently with DTLB_WALK.

E[4], byte 0 bit [4]

TLB access. The defined values of this bit are:

`0` Did not access TLB.

`1` Load/store operation caused an access to at least the first level of data or unified TLB.

If PMUv3 is implemented this Event is required to be implemented consistently with L1D_TLB.

E[3], byte 0 bit [3]

Level 1 Data cache refill. The defined values of this bit are:

`0` Did not cause level 1 data cache refill.

`1` Load/store operation caused a refill of at least the first level of data or unified cache. This includes each data memory access that causes a refill from outside the cache. It excludes accesses that do not cause a new cache refill but are satisfied from refilling data of a previous miss.

If PMUv3 is implemented this Event is required to be implemented consistently with L1D_CACHE_REFILL.

E[2], byte 0 bit [2]

Level 1 Data cache access. The defined values of this bit are:

`0` Did not access level 1 data cache.

`1` Load/store operation caused a cache access to at least the first level of data or unified cache.

If PMUv3 is implemented this Event is required to be implemented consistently with L1D_CACHE.

E[1], byte 0 bit [1]

Architecturally retired. The defined values of this bit are:

`0` Did not retire.

`1` Committed its results to the architectural state of the PE, or completed with a synchronous architectural exception.

Note

A conditional instruction can retire even if it fails its condition code check.

If PMUv3 is implemented this Event is required to be implemented consistently with INST_RETIRED.

E[0], byte 0 bit [0]

Generated exception. The defined values of this bit are:

`0` Did not generate an exception.

`1` Completed with a synchronous exception.

If E[1] in the same Events packet is set to 0, then the meaning of this bit is IMPLEMENTATION DEFINED.

If PMUv3 is implemented this Event is required to be implemented consistently with EXC_TAKEN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D10.2.7 Operation Type packet

The Operation Type packet characteristics are:

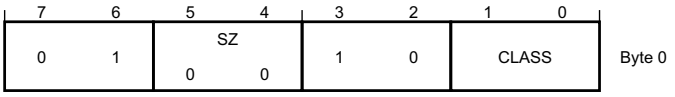
Purpose Defines the type of operation sampled. Included for all operations.

Attributes Multi-part packet comprising:

- 8-bit header.
- 8-bit payload.

Operation Type packet header

The Operation Type packet header bit assignments are:



Byte 0 bits [7:6]

This field reads as `0b01`.

SZ, byte 0 bits [5:4]

Payload size. The defined values of this field are:

`0b00` Byte.

This field reads as `0b00`.

Byte 0 bits [3:2]

This field reads as `0b10`.

CLASS, byte 0 bits [1:0]

Top-level instruction class. The defined values of this field are:

`0b00` Other.

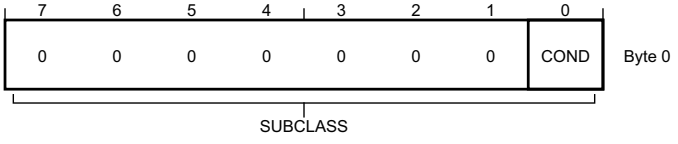
`0b01` Load, store, or atomic.

`0b10` Branch or exception return.

All other values are reserved.

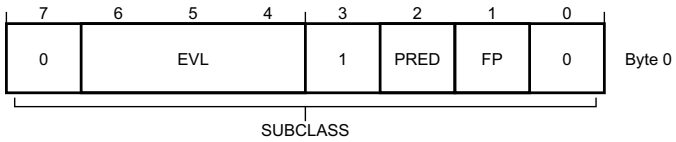
Operation Type packet payload (Other)

When Other operation, the Operation Type packet payload (Other) bit assignments are:



Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When an SVE operation, the Operation Type packet payload (Other) bit assignments are:



SUBCLASS, byte<0>

Second-level instruction class. Defines the type of instruction. The defined values of this field are:

`0b0000000x` Other operation.

`0b0xxx1xx0` SVE operation. If SVE is implemented, and if ARMV8.3-SPE is implemented, bits [6:4:2:1] are further defined as the EVL, PRED, and FP fields.

Otherwise this value is reserved.

EVL, byte 0 bits [6:4], when SVE operation

Effective Vector Length. Defines the sampled operation vector length, rounded up to a power of two. That is, the length of vector operated on by the sampled operation. The defined values of this field are:

`0b000` 32 bits.

`0b001` 64 bits.

`0b010` 128 bits.

`0b011` 256 bits.

`0b100` 512 bits.

`0b101` 1024 bits.

`0b110` 2048 bits.

All other values reserved.

The accessible vector length is always quantized into multiples of 128 bits. However, the effective vector length can be any size down to the smallest element size.

If the effective vector length is not a power of two, or is less than 32 bits, the value is rounded up before it is encoded in this field.

PRED, byte 0 bit[2], when SVE operation

Predicated SVE operation. The defined values of this bit are:

`0` Not predicated.

`1` Predicated SVE operation. The operation is an SVE operation that writes to a vector destination register under a Governing predicate using either zeroing or merging predication.

FP, byte 0 bits [6:4], when SVE operation

Floating-point operation. The defined values of this bit are:

`0` Integer.

`1` Floating-point.

COND, byte 0 bit [0], when Other operation

Conditional. The defined values of this bit are:

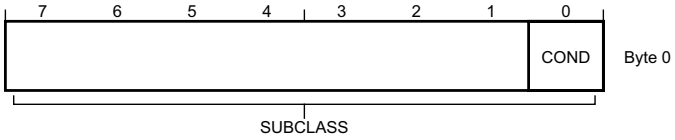
`0` Unconditional operation.

`1` Conditional operation or select.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation Type packet payload (Branch)

The Operation Type packet payload (Branch) bit assignments are:



SUBCLASS, byte <0>

Second-level instruction class. Describes the branch type. The defined values of this field are:

`0b0000000x` Direct branch.

`0b0000001x` Indirect branch.

All other values are reserved.

COND, byte 0 bit [0]

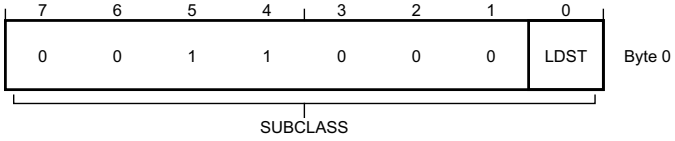
Conditional. The defined values of this bit are:

`0` Unconditional branch.

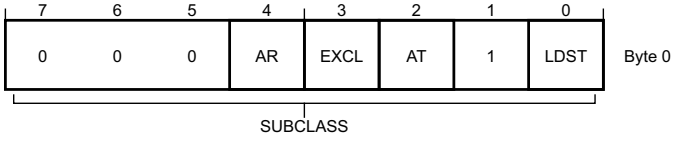
`1` Conditional branch.

Operation Type packet payload (Load/store)

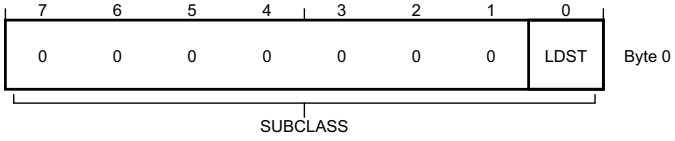
When the ARMv8.4-NV transformed System register access, the Operation Type packet payload (Load/store) bit assignments are:



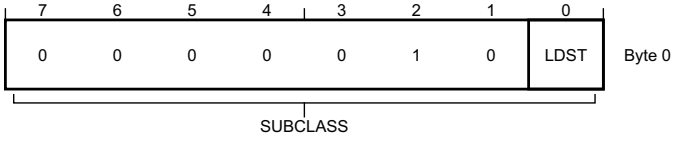
When Extended load/store, the Operation Type packet payload (Load/store) bit assignments are:



When General-purpose load/store, the Operation Type packet payload (Load/store) bit assignments are:

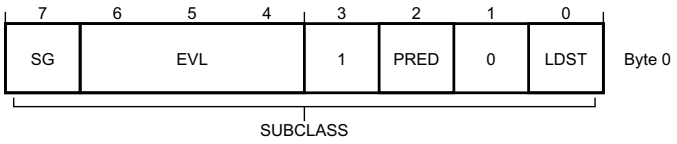


When SIMD&FP load/store, the Operation Type packet payload (Load/store) bit assignments are:

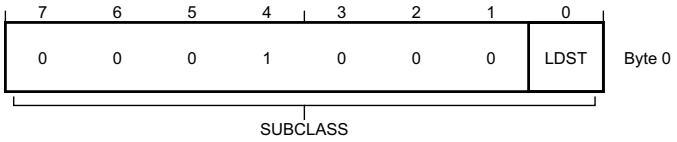


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When SVE load/store, the Operation Type packet payload (Load/store) bit assignments are:



When Unspecified load/store the Operation type packet payload (Load/store) bit assignments are:



SUBCLASS, byte <0>

Second-level instruction class. Indicates the load/store type. The defined values of this field are:

`0b0000000x` A load/store targeting the general-purpose registers, other than an atomic operation, load-acquire, store-release or exclusive.

`0b000xxx1x` An atomic operation, load-acquire, store-release or exclusive. Bits [4:2] are further subdivided as described by the AR, EXCL and AT fields.

`0b0000010x` A load/store targeting the SIMD&FP registers.

`0bxxxx1x0x` A load/store targeting the SVE registers. Bits [7:4,2] are further defined as SG, EVL and PRED fields.

This value is defined only if both The Scalable Vector Extension (SVE) and ARMv8.3-SPE are implemented.

This value is reserved otherwise.

`0b0001000x` A load/store targeting unspecified registers.

This value is defined only if ARMv8.3-SPE is implemented

This value is reserved otherwise.

`0b0011000x` An `MRS` or `MSR` operation at EL1 transformed to a load/store when HCR_EL2.NV2 is 1.

This value is defined only if ARMv8.4-NV is implemented and reserved otherwise.

All other values are reserved.

SG, byte 0 bit [7], when SVE load/store

Gather/scatter load/store. The defined values of this bit are:

`0` Not gather load or scatter store.

`1` Gather load or scatter store.

EVL, byte 0 bits [6:4], when SVE load/store

Effective Vector Length. Defines the sampled operation vector length, rounded up to a power of two. That is, the length of vector operated on by the sampled operation. The defined values of this field are:

`0b000` 32 bits.

`0b001` 64 bits.

`0b010` 128 bits.

`0b011` 256 bits.

`0b100` 512 bits.

`0b101` 1024 bits.

`0b110` 2048 bits.

All other values reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The accessible vector length is always quantized into multiples of 128 bits. However, the effective vector length can be any size down to the smallest element size.

If the effective vector length is not a power of two, or is less than 32 bits, the value is rounded up before it is encoded in this field.

AR, byte 0 bit [4], when Extended load/store

Acquire/Release. The defined values of this bit are:

`0` Load/store/atomic without Acquire or Release semantics.

`1` Load/store/atomic with Acquire or Release semantics.

EXCL, byte 0 bit [3], when Extended load/store

Exclusive. The defined values of this bit are:

`0` Load/store/atomic without Exclusive.

`1` Load/store with Exclusive.

This bit is RES0 if AT == 1.

PRED, byte 0 bit[2], when SVE load/store

Predicated SVE operation. The defined values of this bit are:

`0` Not predicated.

`1` Predicated SVE operation. The operation is an SVE operation that writes to a vector destination register under a Governing predicate using either zeroing or merging predication.

AT, byte 0 bit [2], when Extended load/store

Atomic load/store. The defined values of this bit are:

`0` Not atomic.

`1` Atomic.

LDST, byte 0 bit [0]

Store not load. The defined values of this bit are:

`0` Load or swap.

`1` Store.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D10.2.8 Padding

The Padding characteristics are:

Purpose Allows the PE to create alignment in the protocol buffer.

Attributes 8-bit packet.

Field descriptions

The Padding bit assignments are:

7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	0	Byte 0

Byte <0>

This field reads as `0b00000000`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D10.2.9 Timestamp packet

The Timestamp packet characteristics are:

Purpose The 64-bit timestamp value when the operation was sampled. The Timestamp packet must come at the end of the record. If the Timestamp packet is not present, an End packet must come at the end of the record.

Attributes Multi-part packet comprising:

- 8-bit header.
- 64-bit payload.

Timestamp packet header

The Timestamp packet header bit assignments are:

7	6	5	4	3	2	1	0	
0	1	SZ		0	0	0	1	Byte 0
		1	1					

Byte 0 bits [7:6]

This field reads as `0b01`.

SZ, byte 0 bits [5:4]

Payload size. The defined values of this field are:

`0b11` Doubleword.

This field reads as `0b11`.

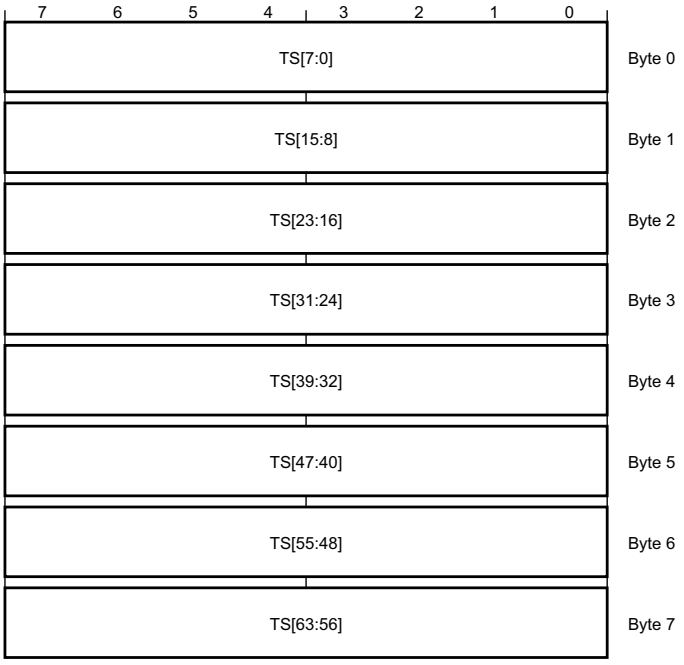
Byte 0 bits [3:0]

This field reads as `0b0001`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Timestamp packet payload

The Timestamp packet payload bit assignments are:



TS, bytes <7:0>

Timestamp value when the operation was sampled. The value depends on the result of `CollectTimeStamp()`:

- If TimeStamp_Virtual, this is the virtual timestamp, CNTVCT_EL0.
- If TimeStamp_Physical, this is the physical timestamp, CNTPCT_EL0.
- If TimeStamp_OffsetPhysical, this is the offset physical timestamp, CNTPCT_EL0 - CNTPOFF_EL2.
- If TimeStamp_None, the timestamp packet is not included and an End packet must come at the end of the record.

However, if the Generic Timer System counter is disabled and `CollectTimeStamp()` returns a value other than TimeStamp_None, then it is IMPLEMENTATION DEFINED whether:

- The Statistical Profiling Extension behaves as if `CollectTimeStamp()` returns the value TimeStamp_None.
- The value of this field in the record is UNKNOWN.

Note

This relaxation refers to when the actual System counter is disabled, that is, CNTEN.EN == 0. It does not apply when the System counter is enabled but not accessible at the current Exception level.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter D11
The Generic Timer in AArch64 state

This chapter describes the implementation of the Arm Generic Timer. It includes an overview of the AArch64 System register interface to an Arm Generic Timer.

It contains the following sections:

- *About the Generic Timer* on page D11-2838.
- *The AArch64 view of the Generic Timer* on page D11-2842.

Chapter G6 *The Generic Timer in AArch32 state* describes the AArch32 view of the Generic Timer, and Chapter I2 *System Level Implementation of the Generic Timer* describes the system level implementation of the Generic Timer.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D11.1 About the Generic Timer

Figure D11-1 shows an example system-on-chip that uses the Generic Timer as a system timer. In this figure:

- This manual defines the architecture of the individual PEs in the multiprocessor blocks.
- The *ARM Generic Interrupt Controller Architecture Specification* defines a possible architecture for the interrupt controllers.
- Generic Timer functionality is distributed across multiple components.

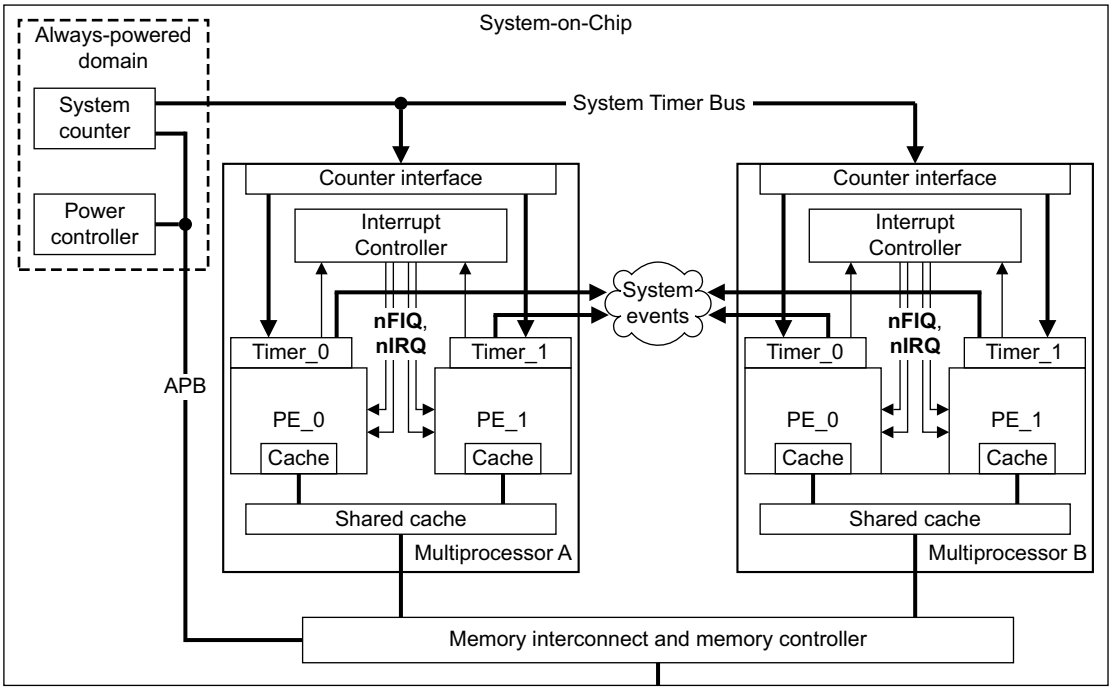


Figure D11-1 Generic Timer example

The Generic Timer:

- Provides a system counter, that measures the passing of time in real-time.

  Note

  The Generic Timer can also provide other components at a system level, but Figure D11-1 does not show any such components.

- Supports *virtual counters* that measure the passing of virtual-time. That is, a virtual counter can measure the passing of time on a particular virtual machine.
- Timers, that can trigger events after a period of time has passed. The timers:
  - Can be used as count-up or as count-down timers.
  - Can operate in real-time or in virtual-time.

This chapter describes an instance of the Generic Timer component that Figure D11-1 shows as Timer_0 or Timer_1 within the Multiprocessor A or Multiprocessor B block. This component can be accessed from AArch64 state or AArch32 state, and this chapter describes access from AArch64 state. Chapter G6 *The Generic Timer in AArch32 state* describes access to this component from AArch32 state.

A Generic Timer implementation must also include a memory-mapped system component. This component:

- Must provide the System counter shown in Figure D11-1
- Optionally, can provide timer components for use at a system level.

Chapter I2 *System Level Implementation of the Generic Timer* describes this memory-mapped component.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D11.1.1 The full set of Generic Timer components

Within a system that might include multiple PEs, a full set of Generic Timer components is as follows:

The system counter

This provides a uniform view of system time, see *The system counter* on page D11-2840. Because this must be implemented at the system level, it is accessed through *The system level memory-mapped implementation of the Generic Timer*. However, during initialization, a status register in each implemented timer in the system must be programmed with the frequency of the system counter, so that software can read this frequency.

PE implementations of the Generic Timer

Each PE implementation of the Generic Timer provides the following components:

- A physical counter, that gives access to the count value of the system counter. When ARMv8.6-ECV is implemented, the CNTPOFF_EL2 register allows offsetting of physical timers and counters.
- A virtual counter, that gives access to virtual time. In AArch64 state, the CNTVOFF_EL2 register defines the offset between physical time, as defined by the value of the system counter, and virtual time.
- A number of timers. In an implementation where all Exception levels are implemented and can use AArch64 state, the timers that are accessible from AArch64 state are:
  - An EL1 physical timer.
  - A Non-secure EL2 physical timer.
  - An EL3 physical timer.
  - An EL1 virtual timer.
  - A Non-secure EL2 virtual timer.
  - A Secure EL2 virtual timer.
  - A Secure EL2 physical timer.

  The Non-secure EL2 virtual timer is available only when ARMv8.1-VHE is implemented.

  The Secure EL2 timers are available only when ARMv8.4-SecEL2 is implemented.

The AArch64 view of the Generic Timer on page D11-2842 describes these components.

The system level memory-mapped implementation of the Generic Timer

The memory-mapped registers that control the components of the system level implementation of the Generic Timer are grouped into *frames*. The Generic Timer architecture defines the offset of each register within its frame, but the base address of each frame is IMPLEMENTATION DEFINED, and defined by the system.

Each system level component has one or two register frames. The possible system level components are:

The memory-mapped counter module, required

This module controls the system counter. It has two frames:

- A control frame, CNTControlBase.
- A status frame, CNTReadBase.

The memory-mapped timer control module, required

The system level implementation of the Generic Timer can provide up to eight timers, and the memory-mapped timer control module identifies:

- Which timers are implemented.
- The features of each implemented timer.

This module has a single frame, CNTCTLBase.

Memory-mapped timers, optional

An implemented memory-mapped timer:

- Must provide a privileged view of the timer, in the CNTBase*N* frame.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Optionally. provides an unprivileged view of the timer in the CNTEL0Base*N* frame.

N is the timer number, and the corresponding frame number, in the range 0-7.

Chapter I2 *System Level Implementation of the Generic Timer* describes these components.

D11.1.2 The system counter

The Generic Timer provides a system counter with the following specification:

Width From Armv8.0 to Armv8.5 inclusive, at least 56 bits wide. The value returned by any 64-bit read of the counter is zero-extended to 64 bits.

From Armv8.6, must be 64 bits wide.

Frequency From Armv8.0 to Armv8.5 inclusive, increments at a fixed frequency, typically in the range 1-50MHz. It can support one or more alternative operating modes in which it increments by larger amounts at a lower frequency, typically for power-saving.

From Armv8.6, increments at a fixed frequency of 1GHz.

Roll-over Roll-over time of not less than 40 years.

Accuracy Arm does not specify a required accuracy, but recommends that the counter does not gain or lose more than ten seconds in a 24-hour period.

Use of lower-frequency modes must not affect the implemented accuracy.

Start-up Starts operating from zero.

The system counter, once configured and running, must provide a uniform view of system time. More precisely, it must be impossible for the following sequence of events to show system time going backwards:

1. Device A reads the time from the system counter.
2. Device A communicates with another agent in the system, Device B.
3. After recognizing the communication from Device A, Device B reads the time from the system counter.

The system counter must be implemented in an always-on power domain.

To support lower-power operating modes in architectures from Armv8.0 to Armv8.5, the counter can increment by larger amounts at a lower frequency. For example, a 10MHz system counter might either increment:

- By 1 at 10MHz.
- By 500 at 20kHz, when the system lowers the clock frequency, to reduce power consumption.

In this case, the counter must support transitions between high-frequency, high-precision operation, and lower-frequency, lower-precision operation, without any impact on the required accuracy of the counter.

From Armv8.6 the counter operates at a higher fixed frequency of 1GHz. This implies a resolution of 1ns.

Note

Though each unit of the counter is set to 1ns, this does not require that the counter is incremented every 1ns. A step in the counter might be more than a single bit increment. It is recommended that the count is not incremented at a rate that is less than 50MHz in normal running operation.

The CNTFRQ_EL0 register is intended to hold a copy of the current clock frequency to allow fast reference to this frequency by software running on the PE. For more information, see *Initializing and reading the system counter frequency* on page D11-2841.

The mechanism by which the count from the system counter is distributed to system components is IMPLEMENTATION DEFINED, but each PE with a System register interface to the system counter must have a counter input that can capture each increment of the counter.

Note

So that the system counter can be clocked independently from the PE hardware, the count value might be distributed using a Gray code sequence. *Gray-count scheme for timer distribution scheme* on page K5-7916 gives more information about this possibility.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Initializing and reading the system counter frequency

The CNTFRQ_EL0 register must be programmed to the clock frequency of the system counter. Typically, this is done only during the system boot process, by using the System register interface to write the system counter frequency to the CNTFRQ_EL0 register. Only software executing at the highest implemented Exception level can write to CNTFRQ_EL0.

Note

The CNTFRQ_EL0 register is UNKNOWN at reset, and therefore the counter frequency must be set as part of the system boot process.

Software can read the CNTFRQ_EL0 register, to determine the current system counter frequency, in the following states:

- Secure and Non-secure EL2.
- Secure and Non-secure EL1.
- When CNTKCTL_EL1.EL0PCTEN is set to 1, Secure and Non-secure EL0.

Memory-mapped controls of the system counter

Some system counter controls are accessible only through the memory-mapped interface to the system counter. These controls are:

- Enabling and disabling the counter.
- Setting the counter value.
- Changing the operating mode, to change the update frequency and increment value.
- Enabling Halt-on-debug, that a debugger can then use to suspend counting.

For descriptions of these controls, see Chapter I2 *System Level Implementation of the Generic Timer*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D11.2 The AArch64 view of the Generic Timer

The following sections describe the components and features of a PE implementation of the Generic Timer, as seen from AArch64 state:

- *The physical counter*.
- *The virtual counter* on page D11-2843.
- *Event streams* on page D11-2845.
- *Timers* on page D11-2846.

D11.2.1 The physical counter

The PE includes a physical counter that contains the count value of the system counter. The CNTPCT_EL0 register holds the current physical counter value. When ARMv8.6-ECV is implemented, the CNTPOFF_EL2 register holds the optional physical offset that can be applied at EL0 and EL1 whether EL0 and EL1 are using AArch64 state or AArch32 state. For more information, see *The physical offset register* on page D11-2843.

Reads of CNTPCT_EL0 can occur speculatively and out of order relative to other instructions executed on the same PE.

The self-synchronized view of the physical counter

When ARMv8.6-ECV is implemented, an alternative way to read the physical counter is supported. The CNTPCTSS_EL0 register is a non-speculative view of the physical counter, as seen from the Exception level that CNTPCTSS_EL0 is read from.

Accesses to the CNTPCTSS_EL0 are subject to the same traps as accesses to the CNTPCT_EL0.

Reads of CNTPCT_EL0 occur in program order relative to reads of CNTPCT_EL0 or CNTPCTSS_EL0.

Reads of CNTPCTSS_EL0 occur in program order relative to reads of CNTPCT_EL0 or CNTPCTSS_EL0.

Example D11-1 Ensuring reads of the physical counter occur after signal reads from memory

If a read from memory is used to obtain a signal from another agent that indicates that CNTPCT_EL0 must be read, an ISB is used to ensure that the read of CNTPCT_EL0 occurs after the signal has been read from memory, as shown in the following code sequence:

```
loop                ; polling for some communication to indicate a requirement to read the timer
    LDR X1, [X2]
    CMP X1, #1      ; has had the value 1 written to it
    B.NE loop
    ISB             ; without this, the CNTPCT_EL0 could be read before the memory location in [X2]
    MRS X1, CNTPCT_EL0
```

When ARMv8.6-ECV is implemented, an access to CNTPCTSS_EL0 can be used in place of the CNTPCT_EL0, which because it cannot be accessed speculatively, allows the ISB to be removed. Therefore the following code sequence can be used:

```
loop                        ; polling for some communication to indicate a requirement to read the timer
    LDR X1, [X2]
    CMP X1, #1              ; has had the value 1 written to it
    B.NE loop
    MRS X1, CNTPCTSS_EL0
```

Similarly where a read of the physical counter is required to take place after the completion of all loads and stores appearing in program order before the read of the counter, then the following code sequences can be used:

```
    ...                 ; earlier loads and stores
    DSB                 ; completes the earlier loads and stores
    ISB                 ; without this the CNTPCT_EL0 could be read before the completion of the earlier
                        ; loads and stores
    MRS X1, CNTPCT_EL0
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Or, if ARMv8.6-ECV is implemented:

```
...                     ; earlier loads and stores
DSB                     ; completes earlier loads and stores
MRS X1, CNTPCTSS_EL0
```

Neither view of the physical counter ensures that:

- Context changes occurring in program order before the read of the counter have been synchronised.
- Accesses to memory appearing in program order after the read of the counter are executed before the counter has been read.

Example D11-2 Ensuring reads of the physical counter occur after previous memory accesses

---

To ensure that all previous memory accesses have completed and all previous context changes have been synchronised before the read of the counter, the following sequence should be used:

```
DSB
ISB
MRS Xn, CNTPCT{SS}_EL0  ; either view of the physical counter has the same effect in this example
```

To ensure that a memory access only occurs after a read of the counter, the following sequence should be used:

```
MRS Xn, CNTPCT{SS}_EL0  ; either view of the physical counter has the same effect in this example
ISB
LDR Xa, [Xb]            ; this load will be executed after the timer has been read
```

The physical offset register

When ARMv8.6-ECV is implemented, the CNTPOFF_EL2 register allows an offset to be applied to the physical counter, as viewed from EL1 and EL0, and to the EL1 physical timer. The CNTPOFF_EL2 register holds the optional offset that applies to the physical counter. The functionality of this 64-bit register is affected by CNTHCTL_EL2.ECV.

When CNTHCTL_EL2.ECV is 1, an MRS to CNTPCT_EL0 from either EL0 or EL1 that is not trapped will return the value `(PCount<63:0> - CNTPOFF_EL2<63:0>)`. For information on how the EL1 physical timer interrupt is triggered when CNTHCTL_EL2.ECV is 1, see *Operation of the CompareValue views of the timers* on page D11-2847.

When EL2 is not enabled for the current Security state, or when CNTHCTL_EL2.ECV is 0, then the behavior of the counters and timers is as described for Armv8.5 and the optional physical offset is not used.

When SCR_EL3.ECVEn is 0, all values of CNTPOFF_EL2 are treated as 0 for all purposes other than direct reads or writes to the register from EL3.

D11.2.2 The virtual counter

An implementation of the Generic Timer always includes a virtual counter, that indicates virtual time.

The virtual counter contains the value of the physical counter minus a 64-bit virtual offset. When executing at EL1 or EL0, the virtual offset value relates to the current virtual machine.

The CNTVOFF_EL2 register contains the virtual offset, see *The virtual offset register* on page D11-2845.

The CNTVCT_EL0 register holds the current virtual counter value.

Reads of CNTVCT_EL0 can occur speculatively and out of order relative to other instructions executed on the same PE.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The self-synchronized view of the virtual counter

When ARMv8.6-ECV is implemented, an alternative way to read the physical counter is supported. The CNTVCTSS_EL0 register is a non-speculative view of the physical counter, as seen from the Exception level that CNTVCTSS_EL0 is read from.

Accesses to the CNTVCTSS_EL0 are subject to the same traps as accesses to the CNTVCT_EL0.

Reads of CNTVCT_EL0 occur in program order relative to reads of CNTVCT_EL0 or CNTVCTSS_EL0.

Reads of CNTVCTSS_EL0 occur in program order relative to reads of CNTVCT_EL0 or CNTVCTSS_EL0.

Example D11-3 Ensuring reads of the virtual counter occur after signal read from memory

If a read from memory is used to obtain a signal from another agent that indicates that CNTVCT_EL0 must be read, an ISB is used to ensure that the read of CNTVCT_EL0 occurs after the signal has been read from memory, as shown in the following code sequence:

```
loop                ; polling for some communication to indicate a requirement to read the timer
    LDR X1, [X2]
    CMP X1, #1      ; has had the value 1 written to it
    B.NE loop
    ISB             ; without this, the CNTVCT_EL0 could be read before the memory location in [X2]
    MRS X1, CNTVCT_EL0
```

When ARMv8.6-ECV is implemented, an access to CNTVCTSS_EL0 can be used in place of the CNTVCT_EL0, which because it cannot be accessed speculatively, allows the ISB to be removed. Therefore the following code sequence can be used:

```
loop                        ; polling for some communication to indicate a requirement to read the timer
    LDR X1, [X2]
    CMP X1, #1              ; has had the value 1 written to it
    B.NE loop
    MRS X1, CNTVCTSS_EL0
```

Similarly where a read of the virtual counter is required to take place after the completion of all loads and stores appearing in program order before the read of the counter, then the following code sequences can be used:

```
    ...                 ; earlier loads and stores
    DSB                 ; completes the earlier loads and stores
    ISB                 ; without this the CNTPCT_EL0 could be read before the completion of the earlier
                        ; loads and stores
    MRS X1, CNTVCT_EL0
```

Or, if ARMv8.6-ECV is implemented:

```
    ...                     ; earlier loads and stores
    DSB                     ; completes earlier loads and stores
    MRS X1, CNTVCTSS_EL0
```

Neither view of the virtual counter ensures that:

- Context changes occurring in program order before the read of the counter have been synchronised.
- Accesses to memory appearing in program order after the read of the counter are executed before the counter has been read.

Example D11-4 Ensuring reads of the virtual counter occur after previous memory accesses

To ensure that all previous memory accesses have completed and all previous context changes have been synchronised before the read of the counter, the following sequence should be used:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
DSB
ISB
MRS Xn, CNTVCT{SS}_EL0  ; either view of the virtual counter has the same effect in this example
```

To ensure that a memory access only occurs after a read of the counter, the following sequence should be used:

```
MRS Xn, CNTVCT{SS}_EL0  ; either view of the virtual counter has the same effect in this example
ISB
LDR Xa, [Xb]            ; this load will be executed after the timer has been read
```

The virtual offset register

The virtual counter is a counter that has a virtual offset relative to the physical counter as viewed from EL2 and EL3. This virtual offset is held in the register CNTVOFF_EL2. The virtual counter value is the count compared by the EL1 virtual timer.

If EL2 is not implemented and enabled, then the virtual counter uses a fixed offset of zero.

D11.2.3 Event streams

An implementation that includes the Generic Timer can use the system counter to generate one or more *event streams*, to generate periodic wake-up events as part of the mechanism described in *Wait for Event mechanism and Send event* on page D1-2397.

Note

An event stream might be used:

- To impose a time-out on a Wait For Event polling loop.
- To safeguard against any programming error that means an expected event is not generated.

The CNTKCTL_EL1.{EVNTEN, EVNTDIR, EVNTI, EVNTIS} fields define an event stream that is generated from the virtual counter.

In all implementations, the CNTHCTL_EL2.{EVNTEN, EVNTDIR, EVNTI, EVNTIS} fields define an event stream that is generated from the physical counter.

The event stream is configured as follows:

- EVNTI selects the counter bit that triggers the event.
- If ARMv8.6-ECV is not implemented, EVNTI selects between bits[0:15].
- If ARMv8.6-ECV is implemented, EVNTIS selects whether ENVTI selects between bits[0:15] or bits[8:23].
- EVNTDIR selects whether the event is generated on each 0 to 1 transition, or each 1 to 0 transition, of the selected counter bit.

The operation of an event stream is as follows:

- The pseudocode variables `PreviousCNTVCT` and `PreviousCNTPCT` are initialized as:

  ```
  // Variables used for generation of the timer event stream.
  bits(64) PreviousCNTVCT = bits(64) UNKNOWN;
  bits(64) PreviousCNTPCT = bits(64) UNKNOWN;
  ```

- The pseudocode functions `TestEventCNTV()` and `TestEventCNTP()` are called on each cycle of the PE clock.
- The `TestEventCNTx()` pseudocode template defines the functions `TestEventCNTV()` and `TestEventCNTP()`:

  ```
  // TestEventCNTx()
  // ===============

  // Template for the TestEventCNTV() and TestEventCNTP() functions
  // Describes operation when all Exception Levels are using AArch64:
  //   CNTxCT_EL0         is  CNTVCT_EL0          or  CNTPCT_EL0          64-bit count value
  ```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
//   CNTxCTL_ELx        is  CNTKCTL_EL1         or  CNTHCTL_EL2         Control register
//   PreviousCNTxCT_EL0 is  PreviousCNTVCT_EL0  or  PreviousCNTPCT_EL0

TestEventCNTx()
    if CNTxCTL_ELx.EVNTEN == '1' then
        n = UInt(CNTxCTL_ELx.EVNTI);

        if CNTxCTL_ELx.EVNTIS == '1' then
            n = n + 8;

        SampleBit   = CNTxCT_EL0<n>;
        PreviousBit = PreviousCNTxCT_ELx<n>;

        if CNTxCTL_ELx.EVNTDIR == '0' then
            if PreviousBit == '0' && SampleBit == '1' then EventRegisterSet();
        else
            if PreviousBit == '1' && SampleBit == '0' then EventRegisterSet();

    PreviousCNTxCT_EL0 = CNTxCT_EL0;

    return;
```

D11.2.4 Timers

In an implementation of the Generic Timer that includes EL3, if EL3 can use AArch64, the following timers are implemented:

- An EL1 physical timer, that:
  - In Secure state, can be accessed from EL1.
  - In Non-secure state, can be accessed from EL1 unless those accesses are trapped to EL2.

  When this timer can be accessed from EL1, an EL1 control determines whether it can be accessed from EL0.
- A Non-secure EL2 physical timer.
- A Secure EL3 physical timer. An EL3 control determines whether this register is accessible from Secure EL1.
- An EL1 virtual timer.
- A Non-secure EL2 virtual timer, when ARMv8.1-VHE is implemented.
- A Secure EL2 physical timer, when ARMv8.4-SecEL2 is implemented, that can only be accessed in Secure EL2 and EL3.
- A Secure EL2 Virtual timer, when ARMv8.4-SecEL2 is implemented, that can only be accessed in Secure EL2 and EL3.

The output of each implemented timer:

- Provides an output signal to the system.
- If the PE interfaces to a *Generic Interrupt Controller* (GIC), signals a *Private Peripheral Interrupt* (PPI) to that GIC. In a multiprocessor implementation, each PE must use the same interrupt number for each timer.

Each timer:

- Is based around a 64-bit CompareValue that provides a 64-bit unsigned upcounter.
- Provides an alternative view of the CompareValue, called the TimerValue, that appears to operate as a 32-bit downcounter.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Has, in addition, a 32-bit Control register.

Table D11-1 Physical Timer registers summary for the Generic Timer

Timer[a] Register	EL1 Physical Timer	EL2 Physical Timer	Secure EL2 Physical TImer[b]	EL3 Physical Timer
CV	CNTP_CVAL_EL0	CNTHP_CVAL_EL2	CNTHPS_CVAL_EL2	CNTPS_CVAL_EL1
TV	CNTP_TVAL_EL0	CNTHP_TVAL_EL2	CNTHPS_TVAL_EL2	CNTPS_TVAL_EL1
Control	CNTP_CTL_EL0	CNTHP_CTL_EL2	CNTHPS_CTL_EL2	CNTPS_CTL_EL1

a. In this column, CV indicates the CompareValue register, and TV indicates the TimerValue register.
b. Only present when the implementation includes ARMv8.4-SecEL2.

Table D11-2 Virtual Timer register summary for the Generic Timer

Timer[a] Register	EL1 Virtual Timer	EL2 Virtual Timer[b]	Secure EL2 Virtual Timer[c]
CV	CNTV_CVAL_EL0	CNTHV_CVAL_EL2	CNTHVS_CVAL_EL2
TV	CNTV_TVAL_EL0	CNTHV_TVAL_EL2	CNTHVS_TVAL_EL2
Control	CNTV_CTL_EL0	CNTHV_CTL_EL2	CNTHVS_CTL_EL2

a. In this column, CV indicates the CompareValue register, and TV indicates the TimerValue register.
b. Only when the implementation includes ARMv8.1-VHE.
c. Only present when the implementation includes ARMv8.4-SecEL2.

Operation of the CompareValue views of the timers

The CompareValue view of a timer operates as a 64-bit upcounter. The timer condition is met when the appropriate counter reaches the value programmed into its CompareValue register. When the timer condition is met, an interrupt is generated if the interrupt is not masked in the corresponding timer control register, CNTP_CTL_EL0, CNTHP_CTL_EL2, CNTHPS_CTL_EL2, CNTPS_CTL_EL1, CNTV_CTL_EL0, or CNTHVS_CTL_EL2. For CNTP_CTL_EL0, the asserted interrupt is the same as the interrupt asserted by the Non-secure instance of the AArch32 register CNTP_CTL.

The operation of this view of a timer is:

```
TimerConditionMet = (((Counter[63:0] – Offset[63:0])[63:0] - CompareValue[63:0]) >= 0)
```

Where:

`TimerConditionMet` Is TRUE if the timer condition for this counter is met, and FALSE otherwise.

`Counter` The physical counter value, that can be read from the CNTPCT_EL0 register.

`Offset` For the EL1 physical timer, if ARMv8.6-ECV is implemented and CNTHCTL_EL2.ECV is 1, this value is held in the CNTPOFF_EL2 register. Otherwise the value is zero.

For the EL1 virtual timer, this value is held in the CNTVOFF_EL2 register.

For the Non-secure EL2 and Secure EL2 virtual timer, this value is zero.

`CompareValue` The value of the appropriate CompareValue register, CNTP_CVAL_EL0, CNTHP_CVAL_EL2, CNTPS_CVAL_EL1, CNTV_CVAL_EL0, or CNTHV_CVAL_EL2.

In this view of a timer, `Counter`, `Offset`, and `CompareValue` are all 64-bit unsigned values.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

This means that a timer with a `CompareValue` of, or close to, `0xFFFF_FFFF_FFFF_FFFF` might never meet its timer condition. However, there is no practical requirement to use values close to the counter wrap value.

Software can observe the counter value by the offset in some situations by reading CNTVCT_EL0. For more information see CNTVCT_EL0.

Operation of the TimerValue views of the timers

The TimerValue view of a timer appears to operate as a signed 32-bit downcounter. A TimerValue register is programmed with a count value. This value decrements on each increment of the appropriate counter, and the timer condition is met when the value reaches zero. When the timer condition is met, an interrupt is generated if the interrupt is not masked in the corresponding timer control register, CNTP_CTL_EL0, CNTHP_CTL_EL2, CNTHPS_CTL_EL2, CNTPS_CTL_EL1, or CNTV_CTL_EL0, CNTHVS_CTL_EL2.

This view of a timer depends on the following behavior of accesses to TimerValue registers:

Reads `TimerValue = (CompareValue - (Counter - Offset))[31:0]`

Writes `CompareValue = ((Counter - Offset)[63:0] + SignExtend(TimerValue))[63:0]`

Where the arguments other than `TimerValue` have the definitions used in *Operation of the CompareValue views of the timers* on page D11-2847, and in addition:

`TimerValue` The value of a TimerValue register, CNTP_TVAL_EL0, CNTHP_TVAL_EL2, CNTHPS_TVAL_EL2, CNTPS_TVAL_EL1, CNTV_TVAL_EL0, or CNTHV_TVAL_EL2, CNTHVS_TVAL_EL2.

In this view of a timer, values are signed in standard two's complement form.

A read of a TimerValue register after the timer condition has been met indicates the time since the timer condition was met.

Note

- *Operation of the CompareValue views of the timers* on page D11-2847 gives a strict definition of `TimerConditionMet`. However, provided that the TimerValue is not expected to wrap as a 32-bit signed value when decremented from `0x80000000`, the TimerValue view can be used as giving an effect equivalent to:

  `TimerConditionMet = (TimerValue ≤ 0)`

- Programming TimerValue to a negative number with magnitude greater than (Counter−Offset) can lead to an arithmetic overflow that causes the CompareValue to be an extremely large positive value. This potentially delays meeting the timer condition for an extremely long period of time.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter D12
AArch64 System Register Encoding

This chapter describes the AArch64 System register encoding space. It contains the following sections:

- *The System register encoding space* on page D12-2850.
- *op0==0b10, Moves to and from debug and trace System registers* on page D12-2851.
- *op0==0b11, Moves to and from non-debug System registers, Special-purpose registers* on page D12-2853.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D12.1 The System register encoding space

The A64 instruction set includes instructions that access the System register encoding space. These instructions provide:

- Access to *System registers*, including the debug registers, that provide system control, and system status information.
- Access to Special-purpose registers such as SPSR_ELx, ELR_ELx, and the equivalent fields of the Process State.
- The cache and TLB maintenance instructions and address translation instructions.
- Barriers and the `CLREX` instruction.
- Architectural hint instructions.

This section describes the parts of the System register encoding space that provides access to the System registers described in Chapter D13 *AArch64 System Register Descriptions*.

Note

- See *Fixed values in AArch64 instruction and System register descriptions* on page C2-197 for information about abbreviations used in the System instruction descriptions.
- In AArch32 state much of this functionality is provided through the System register interface described in *The AArch32 System register interface* on page G1-5769. In AArch64 state, the parameters used to characterize the System register encoding space are {`op0`, `op1`, `CRn`, `CRm`, `op2`}. These are based on the parameters that characterize the AArch32 System register encoding space, which reflect the original implementation of these registers, as described in *Background to the System register interface* on page G1-5770. In Armv8, there is no particular significance to the naming of these parameters, and no functional distinction between the `op`*n* parameters and the `CR`*x* parameters.

Principles of the System instruction class encoding on page C5-376 describes some general properties of these encodings. *System instruction class encoding overview* on page C5-377 then describes the top-level encoding of these instructions, identifying that:

- Entries in the encoding space are characterized by the parameter set {`op0`, `op1`, `CRn`, `CRm`, `op2`}.
- `op0` is the most significant parameter for determining allocations in this space.

Much of this encoding space is used for System instructions, as described in Chapter C5 *The A64 System Instruction Class*. This chapter describes only the part of the encoding space that is used for System registers, in the following sections:

- *op0==0b10, Moves to and from debug and trace System registers* on page D12-2851.
- *op0==0b11, Moves to and from non-debug System registers, Special-purpose registers* on page D12-2853.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D12.2 op0==0b10, Moves to and from debug and trace System registers

The instructions that move data to and from the debug, Execution environment, and trace System registers are encoded with op0==0b10. This means the encoding of these instructions is:

31 30 29 28	27 26 25 24	23 22	21	20 19	18 16	15 12	11 8	7 5	4 0
1 1 0 1	0 1 0 1	0 0	L	1 0	op1	CRn	CRm	op2	Rt
				op0					

Note

- The section describes the use of all of the op0==0b10 region of the System register encoding space.
- These encodings access the registers that are equivalent to the AArch32 System registers in the (coproc==0b1110) encoding space.

The value of op1 provides the next level of decode of these instructions, as follows:

op1 == **{0, 3, 4}**

Debug. See *Instructions for accessing debug System registers*

Note

The standard encoding of debug registers is op0==0b10, op1==$\{0, 3, 4\}$. The registers in the op0==0b11 encoding space that are classified as debug registers are DLR_EL0, DSPSR_EL0, MDCR_EL2, MDCR_EL3, and SDER32_EL3. See *Instructions for accessing non-debug System registers* on page D12-2853 for the encodings of these registers.

op1 == **1** Trace. See the appropriate trace architecture specification.

D12.2.1 Instructions for accessing debug System registers

The instructions for accessing debug System registers are:

```
MSR <System register>, Xt    ; Write to System register
MRS Xt, <System register>    ; Read from System register
```

Where <System_register> is the register name, for example MDCCSR_EL0.

This section includes only the System register access encodings for which both:

- op0 is 0b10.
- The value of op1 is one of $\{0, 3, 4\}$.

Note

These encodings access the registers that are equivalent to the AArch32 System registers in the (coproc==0b1110) encoding space.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D12-1 shows the mapping of the System register encodings for debug System register access.

Table D12-1 System instruction encodings for debug System register access

Register	Access instruction encoding					Permitted accesses
	op0	op1	CRn	CRm	op2	
OSDTRRX_EL1	2	0	0	0	2	RW
MDCCINT_EL1				2	0	RW
MDSCR_EL1					2	RW
OSDTRTX_EL1				3	2	RW
OSECCR_EL1				6	2	RW
DBGBVR<n>_EL1				0-15[a]	4	RW
DBGBCR<n>_EL1				0-15[a]	5	RW
DBGWVR<n>_EL1				0-15[a]	6	RW
DBGWCR<n>_EL1				0-15[a]	7	RW
MDRAR_EL1	2	0	1	0	0	RO
OSLAR_EL1					4	WO
OSLSR_EL1				1	4	RO
OSDLR_EL1				3	4	RW
DBGPRCR_EL1				4	4	RW
DBGCLAIMSET_EL1			7	8	6	RW
DBGCLAIMCLR_EL1				9	6	RW
DBGAUTHSTATUS_EL1				14	6	RO
MDCCSR_EL0		3	0	1	0	RO
DBGDTR_EL0				4	0	RW
DBGDTRRX_EL0				5	0	RO
DBGDTRTX_EL0						WO
DBGVCR32_EL2		4	0	7	0	RW

a. Unimplemented breakpoint and watchpoint register access instructions are unallocated. CRm encodes <*n*>, the breakpoint or watchpoint number.

For more information see *Mapping of the System registers between the Execution states* on page D1-2409.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D12.3 op0==0b11, Moves to and from non-debug System registers, Special-purpose registers

The instructions that move data to and from non-debug System registers are encoded with op0==0b11, except that some of this encoding space is reserved for IMPLEMENTATION DEFINED functionality. The encoding of these instructions is:

31 30 29 28	27 26 25 24	23 22	21	20 19	18 16	15 12	11 8	7 5	4 0
1 1 0 1	0 1 0 1	0 0	L	1 1	op1	CRn	CRm	op2	Rt
				op0					

The value of CRn provides the next level of decode of these instructions, as follows:

CRn==**{0, 1, 2, 3, 5, 6, 7, 9, 10, 12, 13, 14}**
See *Instructions for accessing non-debug System registers*.

CRn==**4** See *Instructions for accessing Special-purpose registers* on page C5-387.

CRn==**{11, 15}** See *Reserved encodings for IMPLEMENTATION DEFINED registers* on page D12-2866.

D12.3.1 Instructions for accessing non-debug System registers

The A64 instructions for accessing System registers are:

```
MSR <System register>, Xt    ; Write to System register
MRS Xt, <System register>    ; Read from System register
```

Where <System_register> is the register name, for example MIDR_EL1.

This section includes only the System register access encodings for which both:

- op0 is 0b11.
- The value of CRn is one of {0, 1, 2, 3, 5, 6, 7, 9, 10, 12, 13, 14}.

Note

- These encodings access the registers that are equivalent to the AArch32 System registers in the (coproc==0b1111) encoding space.
- While this group is described as accessing the non-debug System registers, its correct characterization is by the {op0, CRn} values given in this subsection, and the group includes the debug registers DLR_EL0, DSPSR_EL0, MDCR_EL2, MDCR_EL3, and SDER32_EL3, that are described in *Debug registers* on page D13-3560. These registers are exceptions to the standard encoding of debug registers, that has op0==0b10, see *Instructions for accessing debug System registers* on page D12-2851.

The instruction encoding for these accesses is:

31 30 29 28	27 26 25 24	23 22	21	20 19	18 16	15 12	11 8	7 5	4 0
1 1 0 1	0 1 0 1	0 0	L	1 1	op1	CRn	CRm	op2	Rt
				op0					

See text for permitted values of CRn

Table D12-2 on page D12-2854 shows the encodings of the register access instructions. In the *Notes* on page D12-2854 column of the table:

Config-RO Means it is configurable whether read accesses are permitted. Write accesses are UNDEFINED.

Config-WO Means it is configurable whether write accesses are permitted. Read accesses are UNDEFINED.

Config-RW Means it is configurable whether accesses are permitted. Either read and write accesses are permitted, or read and write accesses are UNDEFINED.

See the register descriptions for information about the control that determines whether these accesses are permitted.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D12-2 System instruction encodings for non-Debug System register accesses

Register accessed	Width (bits)	Access instruction encoding					Source	Notes
		op0	op1	CRn	CRm	op2		
MIDR_EL1	32	3	0	0	0	0	v8.0	RO.
MPIDR_EL1	64					5	v8.0	RO.
REVIDR_EL1	32					6	v8.0	RO.
ID_PFR0_EL1	32				1	0	v8.0	RO, but UNKNOWN if AArch32 is not implemented.
ID_PFR1_EL1	32					1	v8.0	
ID_DFR0_EL1	32					2	v8.0	
ID_AFR0_EL1	32					3	v8.0	
ID_MMFR0_EL1	32					4	v8.0	
ID_MMFR1_EL1	32					5	v8.0	
ID_MMFR2_EL1	32					6	v8.0	
ID_MMFR3_EL1	32					7	v8.0	
ID_ISAR0_EL1	32				2	0	v8.0	RO, but UNKNOWN if AArch32 is not implemented.
ID_ISAR1_EL1	32					1	v8.0	
ID_ISAR2_EL1	32					2	v8.0	
ID_ISAR3_EL1	32					3	v8.0	
ID_ISAR4_EL1	32					4	v8.0	
ID_ISAR5_EL1	32					5	v8.0	
ID_MMFR4_EL1	32					6	v8.0	
Reserved, RAZ	-					*n*	-	RO, for *n*=7.
MVFR0_EL1	32				3	0	v8.0	RO, but UNKNOWN if AArch32 is not implemented.
MVFR1_EL1	32					1	v8.0	
MVFR2_EL1	32					2	v8.0	
ID_PFR2_EL1	64					4	v8.0	RO.
ID_MMFR5_EL1	32					6	v8.0	RO, but UNKNOWN if AArch32 is not implemented.
Reserved, RAZ	-					*n*	-	RO, for *n*={3, 5, 7}.
ID_AA64PFR0_EL1	64				4	0	v8.0	RO.
ID_AA64PFR1_EL1	64					1	v8.0	RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D12-2 System instruction encodings for non-Debug System register accesses (continued)

Register accessed	Width (bits)	Access instruction encoding					Source	Notes
		op0	op1	CRn	CRm	op2		
ID_AA64ZFR0_EL1	64	3	0	0	4	4	SVE[a]	RO, but RAZ if SVE is not implemented.
Reserved, RAZ	-					*n*	-	RO, for *n*={2, 3, 5, 6, 7}.
ID_AA64DFR0_EL1	64				5	0	v8.0	RO.
ID_AA64DFR1_EL1	64					1	v8.0	RO.
ID_AA64AFR0_EL1	64					4	v8.0	RO.
ID_AA64AFR1_EL1	64					5	v8.0	RO.
Reserved, RAZ	-					*n*	-	RO, for *n*={2, 3, 6, 7}.
ID_AA64ISAR0_EL1	64				6	0	v8.0	RO.
ID_AA64ISAR1_EL1	64					1	v8.0	RO.
Reserved, RAZ	-					*n*	-	RO, for *n*=2-7.
ID_AA64MMFR0_EL1	64				7	0	v8.0	RO.
ID_AA64MMFR1_EL1	64					1	v8.0	RO.
ID_AA64MMFR2_EL1	64					2	v8.2	RO.
Reserved, RAZ	-					*n*	-	RO, for *n*=3-7.
SCTLR_EL1	32			1	0	0	v8.0	RW.
ACTLR_EL1	64					1	v8.0	RW, contents IMPLEMENTATION DEFINED.
CPACR_EL1	32					2	v8.0	RW.
ZCR_EL1	64				2	0	SVE	RW.[a]
TRFCR_EL1	64					1	v8.4	RW.
TTBR0_EL1	64			2	0	0	v8.0	RW.
TTBR1_EL1	64					1	v8.0	RW.
TCR_EL1	64					2	v8.0	RW.
APIAKeyLo_EL1	64				1	0	v8.3	RW.
APIAKeyHi_EL1	64					1	v8.3	RW.
APIBKeyLo_EL1	64					2	v8.3	RW.
APIBKeyHi_EL1	64					3	v8.3	RW.
APDAKeyLo_EL1	64				2	0	v8.3	RW.
APDAKeyHi_EL1	64					1	v8.3	RW.
APDBKeyLo_EL1	64					2	v8.3	RW.
APDBKeyHi_EL1	64					3	v8.3	RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D12-2 System instruction encodings for non-Debug System register accesses (continued)

Register accessed	Width (bits)	Access instruction encoding					Source	Notes
		op0	op1	CRn	CRm	op2		
APGAKeyLo_EL1	64	3	0	2	3	0	v8.3	RW.
APGAKeyHi_EL1	64					1	v8.3	RW.
ICC_PMR_EL1 ICV_PMR_EL1	32			4	6	0	GIC[b]	RW.
AFSR0_EL1	32			5	1	0	v8.0	RW, contents IMPLEMENTATION DEFINED.
AFSR1_EL1	32					1	v8.0	
ESR_EL1	32				2	0	v8.0	RW.
ERRIDR_EL1	64				3	0	RAS[c]	RO.
ERRSELR_EL1	64					1	RAS[c]	RW.
ERXFR_EL1	64				4	0	RAS[c]	RO.
ERXCTLR_EL1	64					1	RAS[c]	RW.
ERXSTATUS_EL1	64					2	RAS[c]	RW.
ERXADDR_EL1	64					3	RAS[c]	RW.
ERXPFGF_EL1	64					4	RAS[c]	RO,
ERXPFGCTL_EL1	64					5	RAS[c]	RW.
ERXPFGCDN_EL1	64					6	RAS[c]	RW.
ERXMISC0_EL1	64				5	0	RAS[c]	RW.
ERXMISC1_EL1	64					1	RAS[c]	RW.
ERXMISC2_EL1	64					2	RAS[c]	RW.
ERXMISC3_EL1	64					3	RAS[c]	RW.
FAR_EL1	64			6	0	0	v8.0	RW.
PAR_EL1	64			7	4	0	v8.0	RW.
PMSCR_EL1	64			9	9	0	SPE[d]	RW.
PMSICR_EL1	64					2	SPE[d]	RW.
PMSIRR_EL1	64					3	SPE[d]	RW.
PMSFCR_EL1	64					4	SPE[d]	RW.
PMSEVFR_EL1	64					5	SPE[d]	RW.
PMSLATFR_EL1	64					6	SPE[d]	RW.
PMSIDR_EL1	64					7	SPE[d]	RO.
PMBLIMITR_EL1	64				10	0	SPE[d]	RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D12-2 System instruction encodings for non-Debug System register accesses (continued)

Register accessed	Width (bits)	Access instruction encoding					Source	Notes
		op0	op1	CRn	CRm	op2		
PMBPTR_EL1	64	3	0	9	10	1	SPE[d]	RW.
PMBSR_EL1	64					3	SPE[d]	RW.
PMBIDR_EL1	64					7	SPE[d]	RO.
PMINTENSET_EL1	32				14	1	v8.0	RW.[e]
PMINTENCLR_EL1	32					2	v8.0	RW.[e]
PMMIR_EL1	64					6	v8.4	RO.
MAIR_EL1	64			10	2	0	v8.0	RW.
AMAIR_EL1	64				3	0	v8.0	RW, contents IMPLEMENTATION DEFINED.
LORSA_EL1	64				4	0	v8.1	RW.
LOREA_EL1	64					1	v8.1	RW.
LORN_EL1	64					2	v8.1	RW.
LORC_EL1	64					3	v8.1	RW.
LORID_EL1	64					7	v8.1	RO.
VBAR_EL1	64			12	0	0	v8.0	RW.
RVBAR_EL1	64					1	v8.0	RO. Implemented only if EL2 and EL3 are not implemented.
RMR_EL1	64					2	v8.0	RW. Implemented only if EL2 and EL3 are not implemented.[f]
ISR_EL1	32				1	0	v8.0	RO.
DISR_EL1	64					1	RAS[c]	RW.
ICC_IAR0_EL1 ICV_IAR0_EL1	32				8	0	GIC[b]	RO.[b]
ICC_EOIR0_EL1 ICV_EOIR0_EL1	32					1	GIC[b]	WO.[b]
ICC_HPPIR0_EL1 ICV_HPPIR0_EL1	32					2	GIC[b]	RO.[b]
ICC_BPR0_EL1 ICV_BPR0_EL1	32					3	GIC[b]	RW.[b]
ICC_AP0R<n>_EL1 ICV_AP0R<n>_EL1	32					{4-7}	GIC[b]	RW, <n> = `op2-4`.[b]
ICC_AP1R<n>_EL1 ICV_AP1R<n>_EL1	32				9	{0-3}	GIC[b]	RW, <n> = `op2`.[b]

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D12-2 System instruction encodings for non-Debug System register accesses (continued)

Register accessed	Width (bits)	Access instruction encoding					Source	Notes
		op0	op1	CRn	CRm	op2		
ICC_DIR_EL1 ICV_DIR_EL1	32	3	0	12	11	1	GIC[b]	WO.[b]
ICC_RPR_EL1 ICV_RPR_EL1	32					3	GIC[b]	RO.[b]
ICC_SGI1R_EL1	64					5	GIC[b]	WO.[b]
ICC_ASGI1R_EL1	64					6	GIC[b]	WO.[b]
ICC_SGI0R_EL1	64					7	GIC[b]	WO.[b]
ICC_IAR1_EL1 ICV_IAR1_EL1	32				12	0	GIC[b]	RO.[b]
ICC_EOIR1_EL1 ICV_EOIR1_EL1	32					1	GIC[b]	WO.[b]
ICC_HPPIR1_EL1 ICV_HPPIR1_EL1	32					2	GIC[b]	RO.[b]
ICC_BPR1_EL1 ICV_BPR1_EL1	32					3	GIC[b]	RW.[b]
ICC_CTLR_EL1 ICV_CTLR_EL1	32					4	GIC[b]	RW.[b]
ICC_SRE_EL1	32					5	GIC[b]	RW.[b]
ICC_IGRPEN0_EL1 ICV_IGRPEN0_EL1	32					6	GIC[b]	RW.[b]
ICC_IGRPEN1_EL1 ICV_IGRPEN1_EL1	32					7	GIC[b]	RW.[b]
CONTEXTIDR_EL1	32			13	0	1	v8.0	RW.
TPIDR_EL1	64					4	v8.0	RW.
CNTKCTL_EL1	32			14	1	0	v8.0[g]	RW
CCSIDR_EL1	32		1	0	0	0	v8.0[h]	RO.
	64						v8.3[h]	
CLIDR_EL1	64					1	v8.0	RO.
CCSIDR2_EL1	32					2	v8.3[i]	RO, but IMPLEMENTATION DEFINED[j] if AArch32 is not implemented.
AIDR_EL1	32					7	v8.0	RO.
CSSELR_EL1	32		2	0	0	0	v8.0	RW.
CTR_EL0	32		3	0	0	1	v8.0	Config-RO at EL0, otherwise RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D12-2 System instruction encodings for non-Debug System register accesses (continued)

Register accessed	Width (bits)	Access instruction encoding					Source	Notes
		op0	op1	CRn	CRm	op2		
DCZID_EL0	32	3	3	0	0	7	v8.0	RO.
PMCR_EL0	32			9	12	0	v8.0[e]	Config-RW at EL0, otherwise RW.
PMCNTENSET_EL0	32					1	v8.0[e]	
PMCNTENCLR_EL0	32					2	v8.0[e]	
PMOVSCLR_EL0	32					3	v8.0[e]	
PMSWINC_EL0	32					4	v8.0[e]	Config-WO at EL0, otherwise WO.
PMSELR_EL0	32					5	v8.0[e]	Config-RW at EL0, otherwise RW.
PMCEID0_EL0	32					6	v8.0[e]	Config-RO at EL0, otherwise RO.
PMCEID1_EL0	32					7	v8.0[e]	
PMCCNTR_EL0	64				13	0	v8.0[e]	Config-RW at EL0, otherwise RW.
PMXEVTYPER_EL0	32					1	v8.0[e]	
PMXEVCNTR_EL0	32					2	v8.0[e]	
PMUSERENR_EL0	32				14	0	v8.0[e]	RO at EL0, otherwise RW.
PMOVSSET_EL0	32					3	v8.0[e]	Config-RW at EL0, otherwise RW.
TPIDR_EL0	64			13	0	2	v8.0	RW.
TPIDRRO_EL0	64					3	v8.0	RW.
AMCR_EL0	32				2	0	AMU[k]	Config-RO at EL0, RW at the highest implemented Exception level, otherwise RO.
AMCFGR_EL0	32					1	AMU[k]	Config-RO at EL0, otherwise RO.
AMCGCR_EL0	32					2	AMU[k]	Config-RO at EL0, otherwise RO.
AMUSERENR_EL0	32					3	AMU[k]	RO at EL0, otherwise RW.
AMCNTENCLR0_EL0	32					4	AMU[k]	Config-RO at EL0, RW at the highest implemented Exception level, otherwise RO.
AMCNTENSET0_EL0	32					5	AMU[k]	Config-RO at EL0, RW at the highest implemented Exception level, otherwise RO.
AMCG1IDR_EL0	64					6	v8.6	RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D12-2 System instruction encodings for non-Debug System register accesses (continued)

Register accessed	Width (bits)	Access instruction encoding					Source	Notes
		op0	op1	CRn	CRm	op2		
AMCNTENCLR1_EL0	32	3	3	13	3	0	AMU[k]	Config-RO at EL0, RW at the highest implemented Exception level, otherwise RO.
AMCNTENSET1_EL0	32					1	AMU[k]	Config-RO at EL0, RW at the highest implemented Exception level, otherwise RO.
AMEVCNTR0<n>_EL0	64				{4-5}	{0-7}	AMU[k]	Config-RO at EL0, RW at the highest implemented Exception level, otherwise RO. `CRm` and `op2` encode <n>, the counter number: • For `CRm`==4, <n>=`op2`. • For `CRm`==5, <n>=`op2`+8.
AMEVTYPER0<n>_EL0	32				{6-7}	{0-7}	AMU[k]	Config-RO at EL0, otherwise RO. `CRm` and `op2` encode <n>, the counter number: • For `CRm`==6, <n>=`op2`. • For `CRm`==7, <n>=`op2`+8.
AMEVCNTR1<n>_EL0	64				{12-13}	{0-7}	AMU[k]	Config-RO at EL0, RW at the highest implemented Exception level, otherwise RO. `CRm` and `op2` encode <n>, the counter number: • For `CRm`==12, <n>=`op2`. • For `CRm`==13, <n>=`op2`+8.
AMEVTYPER1<n>_EL0	32				{14-15}	{0-7}	AMU[k]	Config-RO at EL0, RW at the highest implemented Exception level, otherwise RO. `CRm` and `op2` encode <n>, the counter number: • For `CRm`==14, <n>=`op2`. • For `CRm`==15, <n>=`op2`+8.
CNTFRQ_EL0	32			14	0	0	v8.0[g]	Config-RO at EL0, RW at the highest implemented Exception level, otherwise RO.
CNTPCT_EL0	64					1	v8.0[g]	Config-RO at EL0, otherwise RO.
CNTVCT_EL0	64					2	v8.0[g]	
CNTPCTSS_EL0	64					5	v8.6	RO.
CNTVCTSS_EL0	64					6	v8.6	RO.
CNTP_TVAL_EL0	32				2	0	v8.0[g]	Config-RW at EL0 and Non-secure EL1, otherwise RW.
CNTP_CTL_EL0	32					1	v8.0[g]	

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D12-2 System instruction encodings for non-Debug System register accesses (continued)

Register accessed	Width (bits)	Access instruction encoding					Source	Notes
		op0	op1	CRn	CRm	op2		
CNTP_CVAL_EL0	64	3	3	14	2	2	v8.0[g]	
CNTV_TVAL_EL0	32				3	0	v8.0[g]	Config-RW at EL0, otherwise RW.
CNTV_CTL_EL0	32					1	v8.0[g]	
CNTV_CVAL_EL0	64					2	v8.0[g]	
PMEVCNTR<n>_EL0	32				{8-10}	{0-7}	v8.0[e]	Config-RW at EL0, otherwise RW. `CRm` and `op2` encode <n>, the counter number: • For `CRm`=={8, 12}, <n>=`op2`. • For `CRm`=={9, 13}, <n>=`op2`+8. • For `CRm`=={10, 14}, <n>=`op2`+16. • For `CRm`=={11, 15}, <n>=`op2`+24.
					11	{0-6}	v8.0[e]	
PMEVTYPER<n>_EL0	32				{12-14}	{0-7}	v8.0[e]	
					15	{0-6}	v8.0[e]	
PMCCFILTR_EL0	32					7	v8.0[e]	Config-RW at EL0, otherwise RW.
VPIDR_EL2	32		4	0	0	0	v8.0	RW.
VMPIDR_EL2	64					5	v8.0	RW.
SCTLR_EL2	32			1	0	0	v8.0	RW.
ACTLR_EL2	64					1	v8.0	RW, contents IMPLEMENTATION DEFINED.
HCR_EL2	64				1	0	v8.0	RW.
MDCR_EL2	32					1	v8.0	RW.[l]
CPTR_EL2	32					2	v8.0	RW.
HSTR_EL2	32					3	v8.0	RW.
HFGRTR_EL2	64					4	v8.6	RW.
HFGWTR_EL2	64					5	v8.6	RW.
HFGITR_EL2	64					6	v8.6	RW.
HACR_EL2	32					7	v8.0	RW, contents IMPLEMENTATION DEFINED.
ZCR_EL2	64				2	0	SVE[a]	RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D12-2 System instruction encodings for non-Debug System register accesses (continued)

Register accessed	Width (bits)	Access instruction encoding					Source	Notes
		op0	op1	CRn	CRm	op2		
TRFCR_EL2	64	3	4	1	2	1	v8.4	RW.
SDER32_EL2	32				3	1	v8.4	RW.
TTBR0_EL2	64			2	0	0	v8.0	RW.
TCR_EL2	32					2	v8.0	RW.
VTTBR_EL2	64				1	0	v8.0	RW.
VTCR_EL2	32					2	v8.0	RW.
VNCR_EL2	64				2	0	v8.4	RW.
VSTTBR_EL2	64				6	0	v8.4	RW.
VSTCR_EL2	32					2	v8.4	RW.
DACR32_EL2	32			3	0	0	v8.0	RW if EL1 can use AArch32, otherwise UNDEFINED.[m]
HDFGRTR_EL2	64				1	4	v8.6	RW.
HDFGWTR_EL2	64					5	v8.6	RW.
HAFGRTR_EL2	64					6	v8.6 and AMU[k]	RW.
IFSR32_EL2	32			5	0	1	v8.0	RW if EL1 can use AArch32, otherwise UNDEFINED.[m]
AFSR0_EL2	32				1	0	v8.0	RW, contents IMPLEMENTATION DEFINED.
AFSR1_EL2	32					1	v8.0	RW, contents IMPLEMENTATION DEFINED.
ESR_EL2	32				2	0	v8.0	RW.
VSESR_EL2	64					3	RAS[c]	RW.
FPEXC32_EL2	32				3	0	v8.0	RW if EL1 can use AArch32, otherwise UNDEFINED.[m]
FAR_EL2	64			6	0	0	v8.0	RW.
HPFAR_EL2	64					4	v8.0	RW.
PMSCR_EL2	64			9	9	0	SPE[d]	RW.
MAIR_EL2	64			10	2	0	v8.0	RW.
AMAIR_EL2	64				3	0	v8.0	RW, contents IMPLEMENTATION DEFINED.
VBAR_EL2	64			12	0	0	v8.0	RW.
RVBAR_EL2	64					1	v8.0	RO. Implemented only if EL3 is not implemented.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D12-2 System instruction encodings for non-Debug System register accesses (continued)

Register accessed	Width (bits)	Access instruction encoding					Source	Notes
		op0	op1	CRn	CRm	op2		
RMR_EL2	64	3	4	12	0	2	v8.0	RW. Implemented only if EL2 is implemented and EL3 is not implemented.[f]
VDISR_EL2	64				1	1	RAS[c]	RW.
ICH_AP0R<n>_EL2	32				8	{0-3}	GIC[b]	RW, <n>=op2.
ICH_AP1R<n>_EL2	32				9	{0-3}	GIC[b]	RW, <n>=op2.
ICC_SRE_EL2	32					5	GIC[b]	RW.
ICH_HCR_EL2	32				11	0	GIC[b]	RW.
ICH_VTR_EL2	32					1	GIC[b]	RO.
ICH_MISR_EL2	32					2	GIC[b]	RO.
ICH_EISR_EL2	32					3	GIC[b]	RO.
ICH_ELRSR_EL2	32					5	GIC[b]	RO.
ICH_VMCR_EL2	32					7	GIC[b]	RW.
ICH_LR<n>_EL2	64				{12,13}	{0-7}	GIC[b]	RW: • For CRm==12, <n>=op2. • For CRm==13, <n>=op2+8.
CONTEXTIDR_EL2	32			13	0	1	v8.1	RW.
TPIDR_EL2	64					2	v8.0	RW.
AMEVCNTVOFF0<n>_EL2	64				{8-9}	{0-7}	v8.6	RW.
AMEVCNTVOFF1<n>_EL2	64				{10-11}	{0-7}	v8.6	RW.
CNTVOFF_EL2	64			14	0	3	v8.0[g]	RW.
CNTPOFF_EL2	64					6	v8.6	RW.
CNTHCTL_EL2	32				1	0	v8.0[g]	RW.
CNTHP_TVAL_EL2	32				2	0	v8.0[g]	RW.
CNTHP_CTL_EL2	32					1	v8.0[g]	RW.
CNTHP_CVAL_EL2	64					2	v8.0[g]	RW.
CNTHV_TVAL_EL2	32				3	0	v8.1	RW.
CNTHV_CTL_EL2	32					1	v8.1	RW.
CNTHV_CVAL_EL2	64					2	v8.1	RW.
CNTHVS_TVAL_EL2	32				4	0	v8.4	RW.
CNTHVS_CTL_EL2	32					1	v8.4	RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D12-2 System instruction encodings for non-Debug System register accesses (continued)

Register accessed	Width (bits)	Access instruction encoding					Source	Notes
		op0	op1	CRn	CRm	op2		
CNTHVS_CVAL_EL2	64	3	4	14	4	2	v8.4	RW.
CNTHPS_TVAL_EL2	32				5	0	v8.4	RW.
CNTHPS_CTL_EL2	32					1	v8.4	RW.
CNTHPS_CVAL_EL2	64					2	v8.4	RW.
*_EL02 *_EL12	64		5	{0- 15}	{0-15}	{0-7}	v8.1	Reserved for EL2 aliases of EL0 and EL1 registers, see Table D5-47 on page D5-2642 and Table D5-48 on page D5-2643.
SCTLR_EL3	32		6	1	0	0	v8.0	RW.
ACTLR_EL3[63:0]	64					1	v8.0	RW, contents IMPLEMENTATION DEFINED.
SCR_EL3	32				1	0	v8.0	RW.
SDER32_EL3	32					1	v8.0	RW if EL1 can use AArch32, otherwise UNDEFINED.[l, m]
CPTR_EL3	32					2	v8.0	RW.
ZCR_EL3	64				2	0	SVE[a]	RW.
MDCR_EL3	32				3	1	v8.0	RW.[l]
TTBR0_EL3	64			2	0	0	v8.0	RW.
TCR_EL3	32					2	v8.0	RW.
AFSR0_EL3	32			5	1	0	v8.0	RW, contents IMPLEMENTATION DEFINED.
AFSR1_EL3	32					1	v8.0	RW, contents IMPLEMENTATION DEFINED.
ESR_EL3	32				2	0	v8.0	RW.
FAR_EL3	64			6	0	0	v8.0	RW.
MAIR_EL3	64			10	2	0	v8.0	RW.
AMAIR_EL3	64				3	0	v8.0	RW, contents IMPLEMENTATION DEFINED.
VBAR_EL3	64			12	0	0	v8.0	RW.
RVBAR_EL3	64					1	v8.0	RO.
RMR_EL3	64					2	v8.0	RW. Implemented only if EL3 is implemented[f].
ICC_CTLR_EL3	32				12	4	GIC[b]	RW.
ICC_SRE_EL3	32					5	GIC[b]	RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D12-2 System instruction encodings for non-Debug System register accesses (continued)

Register accessed	Width (bits)	Access instruction encoding					Source	Notes
		op0	op1	CRn	CRm	op2		
ICC_IGRPEN1_EL3	32	3	6	12	12	7	GIC[b]	RW.
TPIDR_EL3	64			13	0	2	v8.0	RW.
CNTPS_TVAL_EL1	32		7	14	2	0	v8.0[g]	RW at EL3, Config-RW at Secure EL1.
CNTPS_CTL_EL1	32					1	v8.0[g]	
CNTPS_CVAL_EL1	64					2	v8.0[g]	

a. Scalable Vector Extension System register, see *The Scalable Vector Extension (SVE)* on page A2-99.
b. GIC System register, see *About the GIC System registers* As that subsection describes, each ICV_* register uses the same encoding as the corresponding ICC_* register.
c. RAS Extension System registers, see *The Reliability, Availability, and Serviceability Extension (RAS Extension)* on page A2-97.
d. Statistical Profiling Extension System registers, see *The Statistical Profiling Extension (SPE)* on page A2-98.
e. Performance Monitors Extension System register, see *Performance Monitors registers* on page D13-3663.
f. Required if the highest implemented Exception level can use both AArch32 and AArch64. If the highest implemented Exception level can use only AArch64 then it is IMPLEMENTATION DEFINED whether this register is implemented.
g. Generic Timer System register, see *Generic Timer registers* on page D13-3839.
h. When ARMv8.3-CCIDX is implemented, CCSIDR_EL1 is a 64-bit register. Otherwise, it is a 32-bit register.
i. CCSIDR2_EL1 is implemented only when ARMv8.3-CCIDX is implemented.
j. When AArch32 is not implemented, it is IMPLEMENTATION DEFINED whether CCSIDR2_EL1 is UNDEFINED or UNKNOWN.
k. Activity Monitors System register, see *Activity Monitors registers* on page D13-3724.
l. Debug register in the `op0`==3 encoding space, see *Debug registers* on page D13-3560.
m. Defined to allow access from AArch64 state to registers that are only used in AArch32 state.

About the GIC System registers

From version 3.0 of the GIC architecture specification, the specification defines three groups of System registers, identified by the prefix of the register name:

ICC_ GIC physical CPU interface System registers.
ICH_ GIC virtual interface control System registers.
ICV_ GIC Virtual CPU interface System registers.

Note

These registers are in addition to the GIC memory-mapped register groups GICC_, GICD_, GICH_, GICR_, GICV_, and GITS_.

When implemented, the GIC System registers form part of an Arm processor implementation, and therefore these registers are included in the register summaries. However, the registers are defined only in the GIC Architecture Specification.

As Table D12-2 on page D12-2854 shows, the ICV_* registers have the same {`op0`, `op1`, `CRn`, `CRm`, `op2`} encodings as the corresponding ICC_* registers. For these encodings, GIC register configuration fields determine which register is accessed.

For more information see the *ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0 and version 4.0* (ARM IHI 0069).

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D12.3.2 Reserved encodings for IMPLEMENTATION DEFINED registers

The System register encoding space with `op0==0b11` reserves the following encodings for IMPLEMENTATION DEFINED registers:

31 30 29 28	27 26 25 24	23 22	21	20 19	18 16	15 12	11 8	7 5	4 0
1 1 0 1	0 1 0 1	0 0	L	1 1	op1	1 x 1 1	CRm	op2	Rt
				op0		CRn			

The value of `L` defines the access type and the use of `Rt` as follows:

0 Write the value in `Rt` to the IMPLEMENTATION DEFINED register.

1 Read the value of the IMPLEMENTATION DEFINED register to `Rt`.

For more information about these encodings see *S3_<op1>_<Cn>_<Cm>_<op2>, IMPLEMENTATION DEFINED registers* on page D13-3380. As that section describes, any IMPLEMENTATION DEFINED registers are accessed in a similar way to architecturally-defined System registers, using `MRS` and `MSR` instructions, see:

- *MRS* on page C6-1128.
- *MSR (immediate)* on page C6-1129.
- *MSR (register)* on page C6-1132.

The Arm architecture guarantees not to define any register name prefixed with *IMP_* as part of the standard Arm architecture.

Note

Arm strongly recommends that any register names created in the IMPLEMENTATION DEFINED register spaces be prefixed with *IMP_* and postfixed with *_ELx*, where appropriate.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter D13
AArch64 System Register Descriptions

This chapter defines the AArch64 System registers. It contains the following sections:

- *About the AArch64 System registers* on page D13-2868.
- *General system control registers* on page D13-2876.
- *Debug registers* on page D13-3560.
- *Performance Monitors registers* on page D13-3663.
- *Activity Monitors registers* on page D13-3724.
- *Statistical Profiling Extension registers* on page D13-3755.
- *RAS registers* on page D13-3801.
- *Generic Timer registers* on page D13-3839.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.1 About the AArch64 System registers

The following sections describe common features of the AArch64 registers:

- *Fixed values in the System register descriptions*.
- *General behavior of accesses to the AArch64 System registers*.
- *Principles of the ID scheme for fields in ID registers* on page D13-2873.

D13.1.1 Fixed values in the System register descriptions

See *Fixed values in AArch64 instruction and System register descriptions* on page C2-197. This section defines how the glossary terms RAZ, RES0, RAO, and RES1 can be represented in the System register descriptions.

D13.1.2 General behavior of accesses to the AArch64 System registers

The following subsections give general information about the behavior of accesses to the System registers:

- *Reset behavior of AArch64 System registers*.
- *Synchronization requirements for AArch64 System registers* on page D13-2869.

Reset behavior of AArch64 System registers

Reset values apply only to RW registers and fields, however:

- Some RO registers or fields, including feature ID registers and some status registers or register fields, always return a known value.
- Some RW and RO registers or register fields return status information about the PE. Unless the register description indicates that the value is UNKNOWN on reset, a read of the register immediately after a reset returns valid information.
- Some RW and RO registers and fields are aliases of other registers or fields. In these cases, the reset behavior of the aliased register or field determines the value returned by a read of the register immediately after a reset.
- WO registers that only have an effect on writes do not have meaningful reset values. However, an access to a WO register might affect underlying state, and that state might have a defined reset value.
- IMPLEMENTATION DEFINED registers have IMPLEMENTATION DEFINED reset behavior.

After a reset, only a limited subset of the PE state is guaranteed to be set to defined values. Also, for debug and trace System registers, reset requirements must take account of different levels of reset. For more information about the reset behavior of System registers when the PE resets into an Exception level that is using AArch64, see:

- *PE state on reset to AArch64 state* on page D1-2336.
- The appropriate Trace architecture specification, for the Trace System registers.

For a PE reset into an Exception level that is using AArch64, the architecture defines which AArch64 System registers have a defined reset value, and when that defined reset value applies. The register descriptions include this information, and *PE state on reset to AArch64 state* on page D1-2336 summarizes these architectural requirements. Otherwise, RW registers that have a meaningful reset value reset to an architecturally UNKNOWN value.

Note

When the PE resets into an Exception level that is using AArch32, no PE state that relates to execution in AArch64 state is accessible until another reset causes the Execution state to change to AArch64. Therefore, on a reset into AArch32 state, PE state that relates only to execution in AArch64 state cannot have a meaningful reset value.

Pseudocode description of resetting System registers

The `AArch64.ResetSystemRegisters()` pseudocode function resets all System registers, and register fields, that have defined reset values, as described in this section and *PE state on reset to AArch64 state* on page D1-2336.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

For debug and trace System registers, this function resets registers as defined for the appropriate level of reset.

Synchronization requirements for AArch64 System registers

Reads of the System registers can occur out of order with respect to earlier instructions executed on the same PE, provided that both:

- Any data dependencies between the instructions, including read-after-read dependencies, are respected.
- The reads to the register do not occur earlier than the most recent *Context synchronization event* to its architectural position in the instruction stream.

Note

In particular, the values read from System registers that hold self-incrementing counts, such as the Performance Monitors counters or the Generic Timer counter or timers, could be accessed from any time after the previous *Context synchronization event*. For example, where a memory access is used to communicate a read of such a counter, an `ISB` must be inserted between the read of the memory location that is known to have returned its data, either as a result of a condition on that data or of the read having completed, and the read of the counter, if it is necessary that the counter returns a count value after the memory communication.

Direct writes using the instructions in Table D12-2 on page D12-2854 require synchronization before software can rely on the effects of changes to the System registers to affect instructions appearing in program order after the direct write to the System register. Direct writes to these registers are not allowed to affect any instructions appearing in program order before the direct write. The only exceptions are:

- All direct writes to the same register, that use the same encoding for that register, are guaranteed to occur in program order relative to each other
- All direct writes to a register occur in program order with respect to all direct reads to the same register using the same encoding.
- Any System register access that an Arm *Architecture Specification* or equivalent specification defines as not requiring synchronization.

Explicit synchronization occurs as a result of a *Context synchronization event*, which is one of the following events:

- Execution of an `ISB` instruction.
- Exception entry, if ARMv8.5-CSEH is not implemented, or if ARMv8.5-CSEH is implemented and defines that exception entries to this Exception level are context synchronization events.
- Exception return, if ARMv8.5-CSEH is not implemented, or if ARMv8.5-CSEH is implemented and defines that exception returns from this Exception level are context synchronization events.
- Execution of a `DCPS` instruction in Debug state.
- Execution of a `DRPS` instruction in Debug state.
- Exit from Debug state.

Note

The `ISB` and exception entry events are applicable both in Debug state and in Non-debug state.

Conceptually, explicit synchronization occurs as the first step of each of these events, so that if the event uses state that has previously been changed but was not synchronized by the time of the event, the event is guaranteed to use the state as if it had been synchronized.

Note

This explicit synchronization applies as the first step of the execution of the events, and does not apply to any effect of System registers that apply to the fetch and decode of the instructions that cause these events, such as breakpoints or changes to the translation table.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In addition, any system instructions that cause a write to a System register must be synchronized before the result is guaranteed to be visible to subsequent direct reads of that System register.

Direct reads to any one of the following registers, using the same encoding, occur in program order relative to each other:

- ISR_EL1.
- The Generic Timer registers, that is, CNTPCT_EL0 and CNTVCT_EL0, and the Counter registers CNTP_TVAL_EL0, CNTV_TVAL_EL0, CNTHP_TVAL_EL2, and CNTPS_TVAL_EL1.
- DBGCLAIMCLR_EL1.
- The PMU Counters, that is, PMCCNTR_EL0, PMEVCNTR<n>_EL0, PMXEVCNTR_EL0, PMOVSCLR_EL0, and PMOVSSET_EL0.
- The Debug Communications Channel registers, that is, DBGDTRRX_EL0, DBGDTR_EL0, and MDCCSR_EL0.

All other direct reads of System registers can occur in any order if synchronization has not been performed.

Table D13-1 describes the synchronization requirements between two successive read or write accesses to the same register, where the ordering of the read or write accesses is:

1. Program order, in the event that both the reads or writes are caused by an instruction executed on this PE, other than one caused by a memory access by this PE.
2. The order of arrival of asynchronous reads and writes at the PE relative to the execution of instructions that cause reads or writes.
3. The order of arrival of asynchronous reads and writes at the PE relative to each other.

Table D13-1 Synchronization requirements

First read-write	Second read-write	Synchronization requirement
Direct read	Direct read	None
	Direct write	None
	Indirect read	None
	Indirect write	None, see *Notes* on page D13-2871
Direct write	Direct read	None
	Direct write	None
	Indirect read	Required
	Indirect write	None, see *Notes* on page D13-2871
Indirect read	Direct read	None
	Direct write	None
	Indirect read	None
	Indirect write	None

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D13-1 Synchronization requirements (continued)

First read-write	Second read-write	Synchronization requirement
Indirect write	Direct read	Required, see *Notes*
	Direct write	None, see *Notes*
	Indirect read	Required, see *Notes*
	Indirect write	None, see *Notes*

Notes

The terms Direct read, Direct write, Indirect read, and Indirect write, as used in Table D13-1 on page D13-2870, are defined as follows:

Direct read Where software uses a System register access instruction to read the register, see both:

- *Instructions for accessing debug System registers* on page D12-2851.
- *Instructions for accessing non-debug System registers* on page D12-2853.

Where a direct read of a register has a side-effect that changes the contents of a register, the effect of a direct read on that register is defined to be an indirect write. In this case, the indirect write is only guaranteed to have occurred, and be visible to subsequent direct or indirect reads or writes, if synchronization is performed after the direct read.

Direct write Where software uses a System register access instruction to write to the register, see both:

- *Instructions for accessing debug System registers* on page D12-2851.
- *Instructions for accessing non-debug System registers* on page D12-2853.

Where a direct write to a register has an effect on the register that means that the value in the register is not always the last value that is written (as is the case with set and clear registers), the effect of a direct write on that register is defined to be an indirect write. In this case, the indirect write is only guaranteed to be visible to subsequent direct or indirect reads or writes if synchronization is performed after the direct write and before the subsequent direct or indirect reads or writes.

Indirect read Where an instruction uses a System register to establish operating conditions for the instruction, for example, the TTBR_ELx address or whether memory accesses are forced to be Non-cacheable. This includes situations where the contents of one System register selects what value is read or written using a different register. Indirect reads also include reads of the System register by external agents such as debuggers. Where an indirect read of a register has a side-effect that changes the contents of that register, that is defined to be an indirect write.

Indirect write Where a System register is written as the consequence of some other instruction, exception, operation, or by the asynchronous operation of an external agent, including the passage of time as seen in counters, timers, or performance counters, the assertion of interrupts, or writes from an external debugger.

Note

Since an exception is context synchronizing, registers such as the Exception Syndrome registers that are indirectly written as part of exception entry do not require additional synchronization.

Where a direct read or write to a register is followed by an indirect write caused by an external agent, autonomous asynchronous event, or as a result of memory mapped write, synchronization is required to guarantee the order of those two accesses.

Where an indirect write caused by a direct write is followed by an indirect write caused by an external agent, autonomous asynchronous event, or as a result of memory mapped write, synchronization is required to guarantee the order of those two indirect accesses.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Where a direct read to one register causes a bit or field in a different register (or the same register using a different encoding) to be updated, the change to the different register (or same register using a different encoding) is defined to be an indirect write. In this case, the indirect write is only guaranteed to be visible to subsequent direct or indirect reads or writes if synchronization is performed after the direct read and before the subsequent direct or indirect reads or writes.

Where a direct write to one register causes a bit or field in a different register (or the same register using a different encoding) to be updated as a side-effect of that direct write (as opposed to simply being a direct write to the different encoding), the change to the different register (or same register using a different encoding) is defined to be an indirect write. In this case, the indirect write is only guaranteed to be visible to subsequent direct or indirect reads or writes if synchronization is performed after the direct write and before the subsequent direct or indirect reads or writes.

Where indirect writes are caused by the actions of external agents such as debuggers, or by memory-mapped reads or writes by the PE, then an indirect write by that agent and mechanism to a register, followed by an indirect read by that agent and mechanism to the same register using the same address, does not require synchronization.

Where an indirect write occurs as a side-effect of an access, this happens atomically with the access, meaning no other accesses are allowed between the register access and its side-effect.

Indirect writes caused by external agents, autonomous asynchronous events, or as a result of memory-mapped writes, to the registers shown in Table D13-2, are required to be observable to:

- Direct reads in finite time without explicit synchronization.
- Subsequent indirect reads without explicit synchronization.

Without explicit synchronization to guarantee the order of the accesses, where the same register is accessed by two or more of a System register access instruction, and external agent, and autonomous asynchronous event, or as a result of a memory-mapped access, the behavior must be as if the accesses occurred atomically and in any order. This applies even if the accesses occur simultaneously.

Table D13-2 Registers with a guarantee of observability, VMSAv8-64

Registers	Notes
ISR_EL1	Interrupt Status Register
DBGCLAIMCLR_EL1, DBGCLAIMSET_EL1	Debug CLAIM registers
CNTPCT_EL0, CNTVCT_EL0, CNTP_TVAL_EL0, CNTV_TVAL_EL0, CNTHP_TVAL_EL2, CNTPS_TVAL_EL1	Generic Timer registers
PMCCNTR_EL0, PMEVCNTR<n>_EL0, PMXEVCNTR_EL0, PMOVSCLR_EL0, PMOVSSET_EL0	PMU Counters
DBGDTRTX_EL0, DBGDTRRX_EL0, DBGDTR_EL0, and the DCC flags in MDCCSR_EL0 and EDSCR	Debug Communication Channel registers
EDSCR.PipeAdv	External Debug Status and Control Register PipeAdv field

In addition to the requirements shown in Table D13-2:

- Indirect writes to the following registers as a result of memory-mapped writes, including accesses by external agents, are required to be observable to the indirect read made in determining the response to a subsequent memory-mapped access without explicit synchronization:
  - OSLAR_EL1. OSLAR_EL1 is indirectly read to determine whether the subsequent access is permitted.
  - EDLAR, if implemented. EDLAR is indirectly read to determine whether a subsequent write or side-effect of an access is ignored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

This requirement is stricter than the general requirement for the observability of indirect writes.

- The requirement that an indirect write to the registers in Table D13-2 on page D13-2872 is observable to direct reads in finite time does not imply that all observers will observe the indirect write at the same time.

  For example, an increment of the system counter is an autonomous asynchronous event that performs an indirect write to the counter. This asynchronous event might generate a timer interrupt request, resulting in a *Context synchronization event*. When a GIC is used, the timer interrupt might arrive at the GIC after the PE has taken an interrupt request from another source, but before software reads the current interrupt ID from the GIC. This means that the GIC might identify the timer interrupt as the current interrupt. Software must not assume that a subsequent direct read of the counter register is guaranteed to observe the updated value of that register.

  Although this example uses the counter-timer registers, it applies equally to other registers that might be linked to interrupt requests, including the PMU and Statistical Profiling status registers.

- When the PE is in Debug state, there are synchronization requirements for the Debug Communication Channel and Instruction Transfer registers. See *DCC and ITR access in Debug state* on page H4-7006.

Note

- The provision of explicit synchronization requirements to System registers is provided to allow the direct access to these registers to be implemented in a small number of cycles, and that updates to multiple registers can be performed quickly with the synchronization penalty being paid only when the updates have occurred.
- Since toolkits might use registers such as the thread-local storage registers within compiled code, it is recommended that access to these registers is implemented to take a small number of cycles.
- While no synchronization is required between a direct write and a direct read, or between a direct read and an indirect write, this does not imply that a direct read causes synchronization of a previous direct write. That is, the sequence direct write → direct read → indirect read, with no intervening context synchronization, does not guarantee that the indirect read observes the result of the direct write.

D13.1.3 Principles of the ID scheme for fields in ID registers

The Arm architecture specifies a number of *ID registers* that are characterized as comprising a set of 4-bit *ID fields,* Each ID field identifies the presence, and possibly the level of support for, a particular feature in an implementation of the architecture. These fields follow an architectural model that aids their use by software and provides future compatibility. This section describes that model. *ID registers to which this scheme applies* on page D13-2875 identifies the set of ID registers.

A small number of ID fields do not follow the scheme described in this section. In these cases, the field description states that it does not follow this scheme.

Note

- The ID fields described here are distinct from register fields that enumerate the number of resources, such as the number of breakpoints, watchpoints, or performance monitors, or the amount of memory.
- ID fields that do not follow this scheme include the ID_AA64DFR0_EL1.PMUVer, ID_DFR0_EL1.PerfMon, ID_DFR0.PerfMon and EDDFR.PMUVer fields, see *Alternative ID scheme used for the Performance Monitors Extension version* on page D13-2875.
- The presence of an ID field for a feature does not imply that the feature is optional.

To provide forward compatibility, software can rely on the features of these fields that are described in this section.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The ID fields, which are either signed or unsigned, use increasing numerical values to indicate increases in functionality. Therefore, if a value of `0x1` indicates the presence of some instructions, then the value `0x2` will indicate the presence of those instructions plus some additional instructions or functionality. This means software can be written in the form:

```
if (value >= number) {    // do something that relies on the value of the feature}
```

For ID fields where the value `0x0` defines that a feature is not present, the field holds an unsigned value. This covers the vast majority of such fields.

In a few cases, the architecture has been changed to permit implementations to exclude a feature that has previously been required and for which no ID field has been defined. In these cases, a new ID field is defined and:

- The field holds a signed value.
- The field value `0xF` indicates that the feature is not implemented.
- The field value `0x0` indicates that the feature is implemented.
- Software that depends on the feature can use the test:

```
    if value >= 0 {        // Software features that depend on the presence of the hardware
feature    }
```

In some cases, it has been decided retrospectively that the increase in functionality between two consecutive numerical values is too great, and it is desirable to permit an intermediate degree of functionality, and the means to discover this. This is done by the introduction of a *fractional* field that both:

- Is referred to in the definition of the original field.
- Applies only when the original field is at the lower value of the step.

In principle, a fractional field can be used for two different fractional steps, with different meanings associated with each of these steps. For this reason, a fractional field must be interpreted in the context of the field to which it relates and the value of that field. Example D13-1 shows the use of such a field.

Example D13-1 Example of the use of a fractional field

For a field describing some class of functionality:

- The value `0x1` was defined as indicating that item A is present.
- The value `0x2` was defined as indicating that items B and C are present, in addition to item A.

Subsequently, it might be necessary to introduce a second ID field to indicate that A and B only are present. This new field is a fractional field, and might be defined as having the value `0x1` when A and B only are present. This fractional field is valid only when the original ID field has the value `0x1`.

This approach means that:

- Software that depends on the test if `(value >= 0x2)` can rely on features A, B, and C being present,
- Software that depends on the test `if (value >= 0x1)` can rely on feature A being present.
- If new software needs to check only that features A and B are present, then it can test:

```
    if (value >= 0x2 || (value == 0x1 && fractional_value >= 0x1)) {        // Software features
that depend on A and B only    }
```

A fractional field uses the same approach of increasing numerical values indicating increasing functionality, and the fractional approach can also be applied recursively to fractional fields.

Unused ID fields, and fractional fields that are not applicable, are RES0 to allow their future use when features, or fractional implementation options, are added.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ID registers to which this scheme applies

This scheme applies to the following registers:

AArch64 System registers

- The AArch64 views of the AArch32 feature ID registers given by:
  - The AArch32 Auxiliary Feature register ID_AFR0_EL1.
  - The AArch32 Processor Feature registers ID_PFR0_EL1 and ID_PFR1_EL1.
  - The AArch32 Debug Feature register ID_DFR0_EL1.
  - The AArch32 Memory Model Feature registers ID_MMFR0_EL1, ID_MMFR1_EL1, ID_MMFR2_EL1, ID_MMFR3_EL1, and ID_MMFR4_EL1.
  - The AArch32 Instruction Set Attribute registers ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, ID_ISAR4_EL1, and ID_ISAR5_EL1.
  - The AArch32 Media and VFP Feature registers MVFR0_EL1, MVFR1_EL1, and MVFR2_EL1.
- The AArch64 Auxiliary Feature registers ID_AA64AFR0_EL1 and ID_AA64AFR1_EL1.
- The AArch64 Processor Feature registers ID_AA64PFR0_EL1 and ID_AA64PFR1_EL1.
- The AArch64 Debug Feature registers ID_AA64DFR0_EL1 and ID_AA64DFR1_EL1.
- The AArch64 Memory Model Feature registers ID_AA64MMFR0_EL1, ID_AA64MMFR1_EL1, and ID_AA64MMFR2_EL1.
- The AArch64 Instruction Set Attribute registers ID_AA64ISAR0_EL1 and ID_AA64ISAR1_EL1.

AArch32 System registers

- The AArch32 Auxiliary Feature register ID_AFR0.
- The AArch32 Processor Feature registers ID_PFR0 and ID_PFR1.
- The AArch32 Debug Feature register ID_DFR0.
- The AArch32 Memory Model Feature registers ID_MMFR0, ID_MMFR1, ID_MMFR2, ID_MMFR3, and ID_MMFR4.
- The AArch32 Instruction Set Attribute registers ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR3, ID_ISAR4, and ID_ISAR5.
- The AArch32 Media and FP Feature registers MVFR0, MVFR1, and MVFR2.

Memory-mapped registers

- The External Debug Processor Feature register EDPFR.
- The External Debug Feature register EDDFR.

Alternative ID scheme used for the Performance Monitors Extension version

The ID_AA64DFR0_EL1.PMUVer, ID_DFR0_EL1.PerfMon, ID_DFR0.PerfMon and EDDFR.PMUVer fields, that identify the version of the Performance Monitors Extension, do not follow the standard ID scheme. Software must treat these fields as follows:

- The value `0xF` indicates that the Arm-architected Performance Monitors Extension is not implemented.
- If the field value is not `0xF` the field is treated as an unsigned value, as described for the standard ID scheme.

This means that software that depends on the implementation of a particular version of the Arm Performance Monitors Extension must be written in the form:

```
    if (value != 0xF and value >= number) {        // do something that relies on version 'number' of the
feature    }
```

For these fields, Arm deprecates use of the value `0xF` in new implementations.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2 General system control registers

This section lists the System registers in AArch64 that are not part of one of the other listed groups.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.1 ACTLR_EL1, Auxiliary Control Register (EL1)

The ACTLR_EL1 characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED configuration and control options for execution at EL1 and EL0.

Note

Arm recommends the contents of this register have no effect on the PE when HCR_EL2.{E2H, TGE} is {1, 1}, and instead the configuration and control fields are provided by the ACTLR_EL2 register. This avoids the need for software to manage the contents of these register when switching between a Guest OS and a Host OS.

Configurations

AArch64 System register ACTLR_EL1[31:0] is architecturally mapped to AArch32 System register ACTLR[31:0].

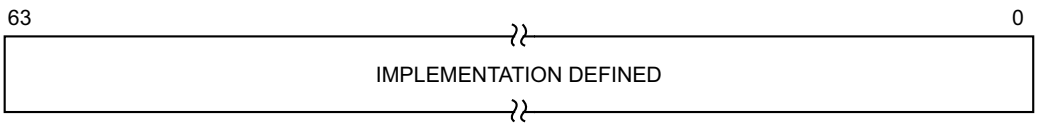
AArch64 System register ACTLR_EL1[63:32] is architecturally mapped to AArch32 System register ACTLR2[31:0].

Attributes

ACTLR_EL1 is a 64-bit register.

Field descriptions

The ACTLR_EL1 bit assignments are:



IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

Accessing the ACTLR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ACTLR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TACR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '1x1' then
        return NVMem[0x118];
    else
        return ACTLR_EL1;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    return ACTLR_EL1;
elsif PSTATE.EL == EL3 then
    return ACTLR_EL1;
```

MSR ACTLR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TACR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '1x1' then
        NVMem[0x118] = X[t];
    else
        ACTLR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    ACTLR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    ACTLR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.2 ACTLR_EL2, Auxiliary Control Register (EL2)

The ACTLR_EL2 characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED configuration and control options for EL2.

Note

Arm recommends the contents of this register are updated to apply to EL0 when HCR_EL2.{E2H, TGE} is {1, 1}, gaining configuration and control fields from the ACTLR_EL1. This avoids the need for software to manage the contents of these register when switching between a Guest OS and a Host OS.

Configurations

AArch64 System register ACTLR_EL2[31:0] is architecturally mapped to AArch32 System register HACTLR[31:0].

AArch64 System register ACTLR_EL2[63:32] is architecturally mapped to AArch32 System register HACTLR2[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

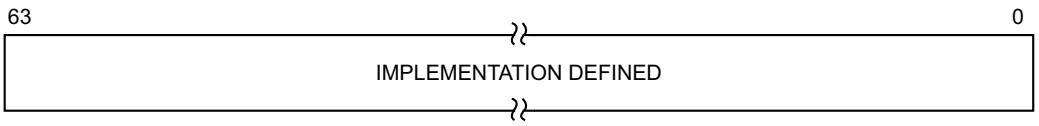
This register has no effect if EL2 is not enabled in the current Security state.

Attributes

ACTLR_EL2 is a 64-bit register.

Field descriptions

The ACTLR_EL2 bit assignments are:



IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

Accessing the ACTLR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ACTLR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    return ACTLR_EL2;
elsif PSTATE.EL == EL3 then
    return ACTLR_EL2;
```

MSR ACTLR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    ACTLR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    ACTLR_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.3 ACTLR_EL3, Auxiliary Control Register (EL3)

The ACTLR_EL3 characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED configuration and control options for EL3.

Configurations

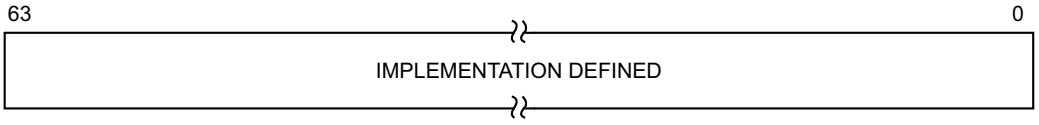
This register is present only when EL3 is implemented. Otherwise, direct accesses to ACTLR_EL3 are UNDEFINED.

Attributes

ACTLR_EL3 is a 64-bit register.

Field descriptions

The ACTLR_EL3 bit assignments are:



IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

Accessing the ACTLR_EL3

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ACTLR_EL3

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    return ACTLR_EL3;
```

MSR ACTLR_EL3, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    ACTLR_EL3 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.4 AFSR0_EL1, Auxiliary Fault Status Register 0 (EL1)

The AFSR0_EL1 characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED fault status information for exceptions taken to EL1.

Configurations

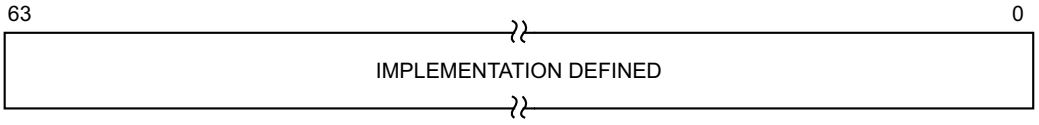
AArch64 System register AFSR0_EL1[31:0] is architecturally mapped to AArch32 System register ADFSR[31:0].

Attributes

AFSR0_EL1 is a 64-bit register.

Field descriptions

The AFSR0_EL1 bit assignments are:



IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

Accessing the AFSR0_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic AFSR0_EL1 or AFSR0_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, AFSR0_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.AFSR0_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x128];
    else
        return AFSR0_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return AFSR0_EL2;
    else
        return AFSR0_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    return AFSR0_EL1;
```

MSR AFSR0_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.AFSR0_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x128] = X[t];
    else
        AFSR0_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        AFSR0_EL2 = X[t];
    else
        AFSR0_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    AFSR0_EL1 = X[t];
```

MRS <Xt>, AFSR0_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0101	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        return NVMem[0x128];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return AFSR0_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        return AFSR0_EL1;
    else
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR AFSR0_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0101	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        NVMem[0x128] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        AFSR0_EL1 = X[t];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        AFSR0_EL1 = X[t];
    else
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.5 AFSR0_EL2, Auxiliary Fault Status Register 0 (EL2)

The AFSR0_EL2 characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED fault status information for exceptions taken to EL2.

Configurations

AArch64 System register AFSR0_EL2[31:0] is architecturally mapped to AArch32 System register HADFSR[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

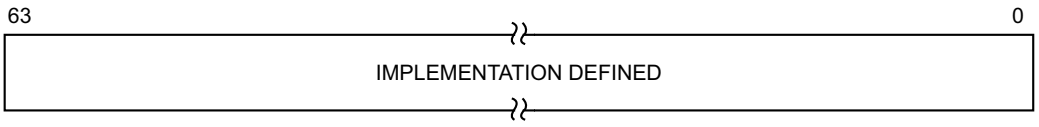
This register has no effect if EL2 is not enabled in the current Security state.

Attributes

AFSR0_EL2 is a 64-bit register.

Field descriptions

The AFSR0_EL2 bit assignments are:



IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

Accessing the AFSR0_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic AFSR0_EL2 or AFSR0_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, AFSR0_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return AFSR0_EL2;
elsif PSTATE.EL == EL3 then
    return AFSR0_EL2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR AFSR0_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AFSR0_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    AFSR0_EL2 = X[t];
```

MRS <Xt>, AFSR0_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.AFSR0_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x128];
    else
        return AFSR0_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return AFSR0_EL2;
    else
        return AFSR0_EL1;
elsif PSTATE.EL == EL3 then
    return AFSR0_EL1;
```

MSR AFSR0_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.AFSR0_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x128] = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        AFSR0_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        AFSR0_EL2 = X[t];
    else
        AFSR0_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    AFSR0_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.6 AFSR0_EL3, Auxiliary Fault Status Register 0 (EL3)

The AFSR0_EL3 characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED fault status information for exceptions taken to EL3.

Configurations

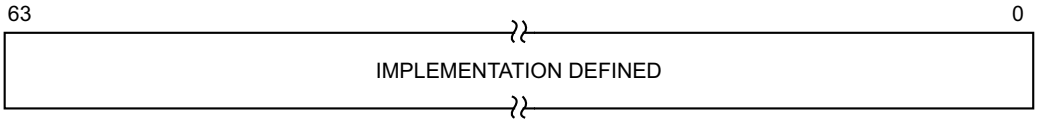
This register is present only when EL3 is implemented. Otherwise, direct accesses to AFSR0_EL3 are UNDEFINED.

Attributes

AFSR0_EL3 is a 64-bit register.

Field descriptions

The AFSR0_EL3 bit assignments are:



IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

Accessing the AFSR0_EL3

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, AFSR0_EL3

op0	op1	CRn	CRm	op2
0b11	0b110	0b0101	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    return AFSR0_EL3;
```

MSR AFSR0_EL3, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b110	0b0101	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    AFSR0_EL3 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.7 AFSR1_EL1, Auxiliary Fault Status Register 1 (EL1)

The AFSR1_EL1 characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED fault status information for exceptions taken to EL1.

Configurations

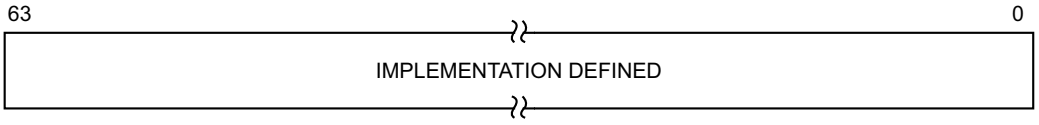
AArch64 System register AFSR1_EL1[31:0] is architecturally mapped to AArch32 System register AIFSR[31:0].

Attributes

AFSR1_EL1 is a 64-bit register.

Field descriptions

The AFSR1_EL1 bit assignments are:



IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

Accessing the AFSR1_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic AFSR1_EL1 or AFSR1_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, AFSR1_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.AFSR1_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x130];
    else
        return AFSR1_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return AFSR1_EL2;
    else
        return AFSR1_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    return AFSR1_EL1;
```

MSR AFSR1_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.AFSR1_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x130] = X[t];
    else
        AFSR1_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        AFSR1_EL2 = X[t];
    else
        AFSR1_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    AFSR1_EL1 = X[t];
```

MRS <Xt>, AFSR1_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0101	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        return NVMem[0x130];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return AFSR1_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        return AFSR1_EL1;
    else
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR AFSR1_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0101	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        NVMem[0x130] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        AFSR1_EL1 = X[t];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        AFSR1_EL1 = X[t];
    else
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.8 AFSR1_EL2, Auxiliary Fault Status Register 1 (EL2)

The AFSR1_EL2 characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED fault status information for exceptions taken to EL2.

Configurations

AArch64 System register AFSR1_EL2[31:0] is architecturally mapped to AArch32 System register HAIFSR[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

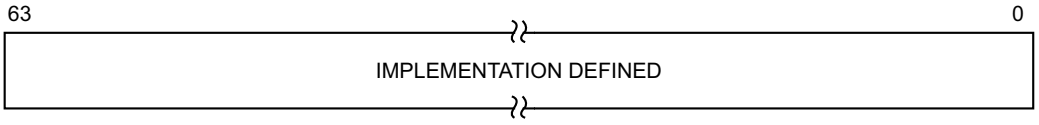
This register has no effect if EL2 is not enabled in the current Security state.

Attributes

AFSR1_EL2 is a 64-bit register.

Field descriptions

The AFSR1_EL2 bit assignments are:



IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

Accessing the AFSR1_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic AFSR1_EL2 or AFSR1_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, AFSR1_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return AFSR1_EL2;
elsif PSTATE.EL == EL3 then
    return AFSR1_EL2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR AFSR1_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AFSR1_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    AFSR1_EL2 = X[t];
```

MRS <Xt>, AFSR1_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.AFSR1_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x130];
    else
        return AFSR1_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return AFSR1_EL2;
    else
        return AFSR1_EL1;
elsif PSTATE.EL == EL3 then
    return AFSR1_EL1;
```

MSR AFSR1_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.AFSR1_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x130] = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        AFSR1_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        AFSR1_EL2 = X[t];
    else
        AFSR1_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    AFSR1_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.9 AFSR1_EL3, Auxiliary Fault Status Register 1 (EL3)

The AFSR1_EL3 characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED fault status information for exceptions taken to EL3.

Configurations

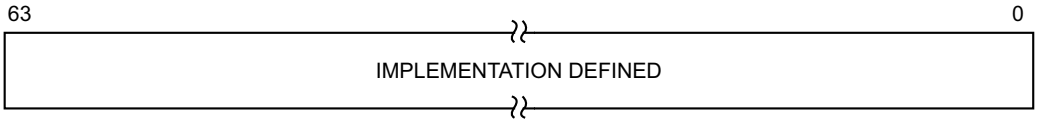
This register is present only when EL3 is implemented. Otherwise, direct accesses to AFSR1_EL3 are UNDEFINED.

Attributes

AFSR1_EL3 is a 64-bit register.

Field descriptions

The AFSR1_EL3 bit assignments are:



IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

Accessing the AFSR1_EL3

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, AFSR1_EL3

op0	op1	CRn	CRm	op2
0b11	0b110	0b0101	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    return AFSR1_EL3;
```

MSR AFSR1_EL3, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b110	0b0101	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    AFSR1_EL3 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.10 AIDR_EL1, Auxiliary ID Register

The AIDR_EL1 characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED identification information.

The value of this register must be interpreted in conjunction with the value of MIDR_EL1.

Configurations

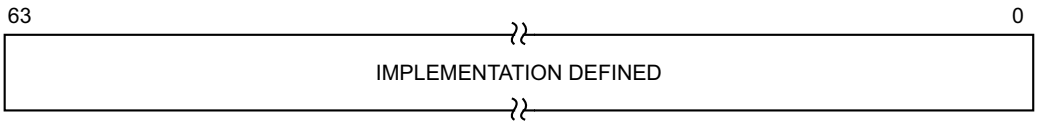
AArch64 System register AIDR_EL1[31:0] is architecturally mapped to AArch32 System register AIDR[31:0].

Attributes

AIDR_EL1 is a 64-bit register.

Field descriptions

The AIDR_EL1 bit assignments are:



IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

Accessing the AIDR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, AIDR_EL1

op0	op1	CRn	CRm	op2
0b11	0b001	0b0000	0b0000	0b111

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.AIDR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return AIDR_EL1;
elsif PSTATE.EL == EL2 then
    return AIDR_EL1;
elsif PSTATE.EL == EL3 then
    return AIDR_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.11 AMAIR_EL1, Auxiliary Memory Attribute Indirection Register (EL1)

The AMAIR_EL1 characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED memory attributes for the memory regions specified by MAIR_EL1.

Configurations

AArch64 System register AMAIR_EL1[31:0] is architecturally mapped to AArch32 System register AMAIR0[31:0].

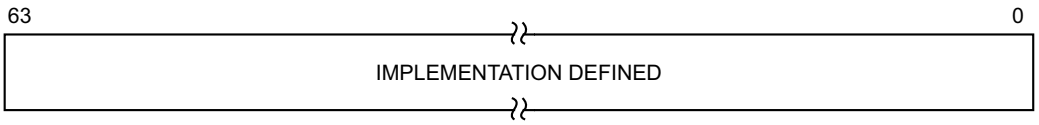
AArch64 System register AMAIR_EL1[63:32] is architecturally mapped to AArch32 System register AMAIR1[31:0].

Attributes

AMAIR_EL1 is a 64-bit register.

Field descriptions

The AMAIR_EL1 bit assignments are:



AMAIR_EL1 is permitted to be cached in a TLB.

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

Accessing the AMAIR_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic AMAIR_EL1 or AMAIR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, AMAIR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.AMAIR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x148];
    else
        return AMAIR_EL1;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if HCR_EL2.E2H == '1' then
        return AMAIR_EL2;
    else
        return AMAIR_EL1;
elsif PSTATE.EL == EL3 then
    return AMAIR_EL1;
```

MSR AMAIR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.AMAIR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x148] = X[t];
    else
        AMAIR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        AMAIR_EL2 = X[t];
    else
        AMAIR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    AMAIR_EL1 = X[t];
```

MRS <Xt>, AMAIR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b1010	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        return NVMem[0x148];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return AMAIR_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        return AMAIR_EL1;
    else
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR AMAIR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b1010	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        NVMem[0x148] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        AMAIR_EL1 = X[t];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        AMAIR_EL1 = X[t];
    else
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.12 AMAIR_EL2, Auxiliary Memory Attribute Indirection Register (EL2)

The AMAIR_EL2 characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED memory attributes for the memory regions specified by MAIR_EL2.

Configurations

AArch64 System register AMAIR_EL2[31:0] is architecturally mapped to AArch32 System register HAMAIR0[31:0].

AArch64 System register AMAIR_EL2[63:32] is architecturally mapped to AArch32 System register HAMAIR1[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

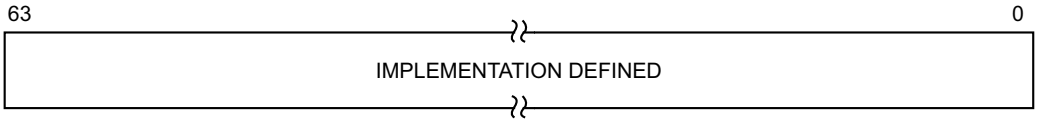
This register has no effect if EL2 is not enabled in the current Security state.

Attributes

AMAIR_EL2 is a 64-bit register.

Field descriptions

The AMAIR_EL2 bit assignments are:



AMAIR_EL2 is permitted to be cached in a TLB.

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

Accessing the AMAIR_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic AMAIR_EL2 or AMAIR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, AMAIR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return AMAIR_EL2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    return AMAIR_EL2;
```

MSR AMAIR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AMAIR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    AMAIR_EL2 = X[t];
```

MRS <Xt>, AMAIR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.AMAIR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x148];
    else
        return AMAIR_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return AMAIR_EL2;
    else
        return AMAIR_EL1;
elsif PSTATE.EL == EL3 then
    return AMAIR_EL1;
```

MSR AMAIR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.AMAIR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x148] = X[t];
    else
        AMAIR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        AMAIR_EL2 = X[t];
    else
        AMAIR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    AMAIR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.13 AMAIR_EL3, Auxiliary Memory Attribute Indirection Register (EL3)

The AMAIR_EL3 characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED memory attributes for the memory regions specified by MAIR_EL3.

Configurations

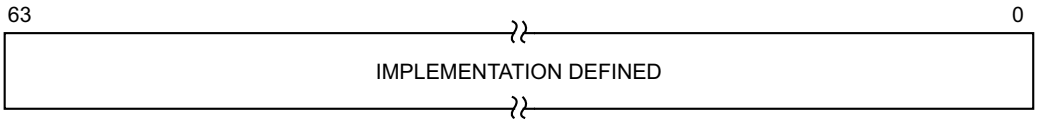
This register is present only when EL3 is implemented. Otherwise, direct accesses to AMAIR_EL3 are UNDEFINED.

Attributes

AMAIR_EL3 is a 64-bit register.

Field descriptions

The AMAIR_EL3 bit assignments are:



AMAIR_EL3 is permitted to be cached in a TLB.

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

Accessing the AMAIR_EL3

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, AMAIR_EL3

op0	op1	CRn	CRm	op2
0b11	0b110	0b1010	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    return AMAIR_EL3;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR AMAIR_EL3, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b110	0b1010	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    AMAIR_EL3 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.14 APDAKeyHi_EL1, Pointer Authentication Key A for Data (bits[127:64])

The APDAKeyHi_EL1 characteristics are:

Purpose

Holds bits[127:64] of key A used for authentication of data pointer values.

Note

The term APDAKey_EL1 is used to describe the concatenation of APDAKeyHi_EL1: APDAKeyLo_EL1.

Configurations

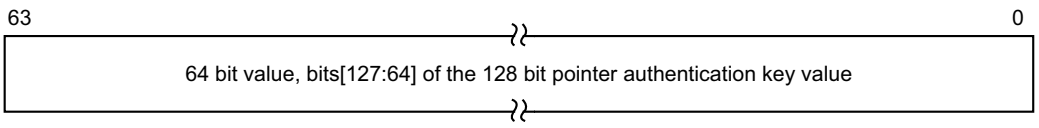
This register is present only when ARMv8.3-PAuth is implemented. Otherwise, direct accesses to APDAKeyHi_EL1 are UNDEFINED.

Attributes

APDAKeyHi_EL1 is a 64-bit register.

Field descriptions

The APDAKeyHi_EL1 bit assignments are:



Bits [63:0]

64 bit value, bits[127:64] of the 128 bit pointer authentication key value.

This field resets to an architecturally UNKNOWN value.

Accessing the APDAKeyHi_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, APDAKeyHi_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.APDAKey == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return APDAKeyHi_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        return APDAKeyHi_EL1;
elsif PSTATE.EL == EL3 then
    return APDAKeyHi_EL1;
```

MSR APDAKeyHi_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.APDAKey == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        APDAKeyHi_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        APDAKeyHi_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    APDAKeyHi_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.15 APDAKeyLo_EL1, Pointer Authentication Key A for Data (bits[63:0])

The APDAKeyLo_EL1 characteristics are:

Purpose

Holds bits[63:0] of key A used for authentication of data pointer values.

Note

The term APDAKey_EL1 is used to describe the concatenation of APDAKeyHi_EL1: APDAKeyLo_EL1.

Configurations

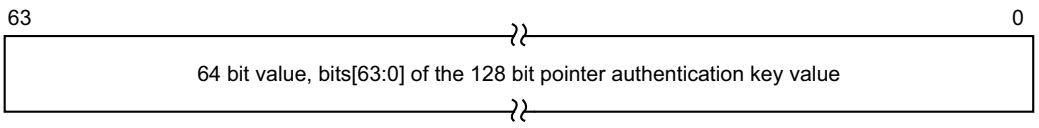
This register is present only when ARMv8.3-PAuth is implemented. Otherwise, direct accesses to APDAKeyLo_EL1 are UNDEFINED.

Attributes

APDAKeyLo_EL1 is a 64-bit register.

Field descriptions

The APDAKeyLo_EL1 bit assignments are:



Bits [63:0]

64 bit value, bits[63:0] of the 128 bit pointer authentication key value.

This field resets to an architecturally UNKNOWN value.

Accessing the APDAKeyLo_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, APDAKeyLo_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.APDAKey == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return APDAKeyLo_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        return APDAKeyLo_EL1;
elsif PSTATE.EL == EL3 then
    return APDAKeyLo_EL1;
```

MSR APDAKeyLo_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.APDAKey == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        APDAKeyLo_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        APDAKeyLo_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    APDAKeyLo_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.16 APDBKeyHi_EL1, Pointer Authentication Key B for Data (bits[127:64])

The APDBKeyHi_EL1 characteristics are:

Purpose

Holds bits[127:64] of key B used for authentication of data pointer values.

Note

The term APDBKey_EL1 is used to describe the concatenation of APDBKeyHi_EL1: APDBKeyLo_EL1.

Configurations

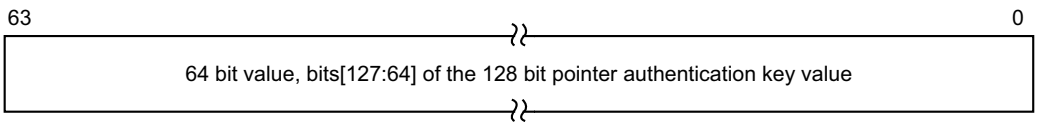
This register is present only when ARMv8.3-PAuth is implemented. Otherwise, direct accesses to APDBKeyHi_EL1 are UNDEFINED.

Attributes

APDBKeyHi_EL1 is a 64-bit register.

Field descriptions

The APDBKeyHi_EL1 bit assignments are:



Bits [63:0]

64 bit value, bits[127:64] of the 128 bit pointer authentication key value.

This field resets to an architecturally UNKNOWN value.

Accessing the APDBKeyHi_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, APDBKeyHi_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0010	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.APDBKey == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return APDBKeyHi_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        return APDBKeyHi_EL1;
elsif PSTATE.EL == EL3 then
    return APDBKeyHi_EL1;
```

MSR APDBKeyHi_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0010	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.APDBKey == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        APDBKeyHi_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        APDBKeyHi_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    APDBKeyHi_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.17 APDBKeyLo_EL1, Pointer Authentication Key B for Data (bits[63:0])

The APDBKeyLo_EL1 characteristics are:

Purpose

Holds bits[63:0] of key B used for authentication of data pointer values.

Note

The term APDBKey_EL1 is used to describe the concatenation of APDBKeyHi_EL1: APDBKeyLo_EL1.

Configurations

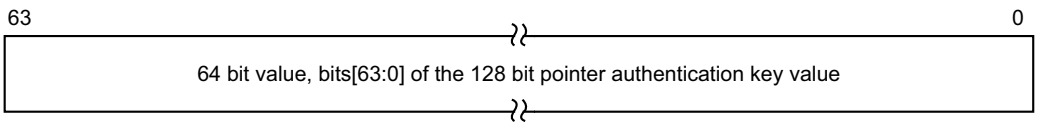
This register is present only when ARMv8.3-PAuth is implemented. Otherwise, direct accesses to APDBKeyLo_EL1 are UNDEFINED.

Attributes

APDBKeyLo_EL1 is a 64-bit register.

Field descriptions

The APDBKeyLo_EL1 bit assignments are:



Bits [63:0]

64 bit value, bits[63:0] of the 128 bit pointer authentication key value.

This field resets to an architecturally UNKNOWN value.

Accessing the APDBKeyLo_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, APDBKeyLo_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0010	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.APDBKey == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return APDBKeyLo_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        return APDBKeyLo_EL1;
elsif PSTATE.EL == EL3 then
    return APDBKeyLo_EL1;
```

MSR APDBKeyLo_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0010	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.APDBKey == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        APDBKeyLo_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        APDBKeyLo_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    APDBKeyLo_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.18 APGAKeyHi_EL1, Pointer Authentication Key A for Code (bits[127:64])

The APGAKeyHi_EL1 characteristics are:

Purpose

Holds bits[127:64] of key used for generic pointer authentication code.

Note

The term APGAKey_EL1 is used to describe the concatenation of APGAKeyHi_EL1: APGAKeyLo_EL1.

Configurations

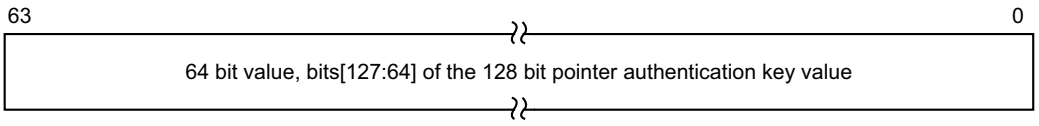
This register is present only when ARMv8.3-PAuth is implemented. Otherwise, direct accesses to APGAKeyHi_EL1 are UNDEFINED.

Attributes

APGAKeyHi_EL1 is a 64-bit register.

Field descriptions

The APGAKeyHi_EL1 bit assignments are:



Bits [63:0]

64 bit value, bits[127:64] of the 128 bit pointer authentication key value.

This field resets to an architecturally UNKNOWN value.

Accessing the APGAKeyHi_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, APGAKeyHi_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0011	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.APGAKey == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return APGAKeyHi_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        return APGAKeyHi_EL1;
elsif PSTATE.EL == EL3 then
    return APGAKeyHi_EL1;
```

MSR APGAKeyHi_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0011	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.APGAKey == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        APGAKeyHi_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        APGAKeyHi_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    APGAKeyHi_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.19 APGAKeyLo_EL1, Pointer Authentication Key A for Code (bits[63:0])

The APGAKeyLo_EL1 characteristics are:

Purpose

Holds bits[63:0] of key used for generic pointer authentication code.

Note

The term APGAKey_EL1 is used to describe the concatenation of APGAKeyHi_EL1: APGAKeyLo_EL1.

Configurations

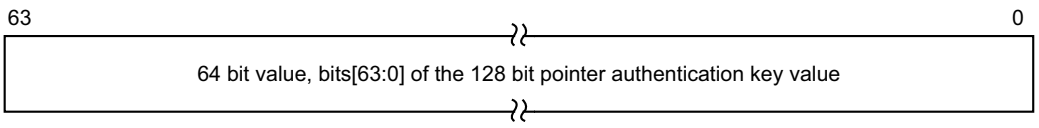
This register is present only when ARMv8.3-PAuth is implemented. Otherwise, direct accesses to APGAKeyLo_EL1 are UNDEFINED.

Attributes

APGAKeyLo_EL1 is a 64-bit register.

Field descriptions

The APGAKeyLo_EL1 bit assignments are:



Bits [63:0]

64 bit value, bits[63:0] of the 128 bit pointer authentication key value.

This field resets to an architecturally UNKNOWN value.

Accessing the APGAKeyLo_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, APGAKeyLo_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.APGAKey == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return APGAKeyLo_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        return APGAKeyLo_EL1;
elsif PSTATE.EL == EL3 then
    return APGAKeyLo_EL1;
```

MSR APGAKeyLo_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.APGAKey == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        APGAKeyLo_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        APGAKeyLo_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    APGAKeyLo_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.20 APIAKeyHi_EL1, Pointer Authentication Key A for Instruction (bits[127:64])

The APIAKeyHi_EL1 characteristics are:

Purpose

Holds bits[127:64] of key A used for authentication of instruction pointer values.

Note

The term APIAKey_EL1 is used to describe the concatenation of APIAKeyHi_EL1: APIAKeyLo_EL1.

Configurations

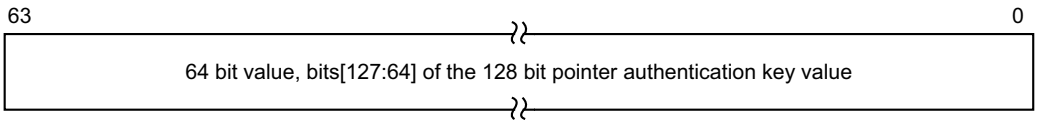
This register is present only when ARMv8.3-PAuth is implemented. Otherwise, direct accesses to APIAKeyHi_EL1 are UNDEFINED.

Attributes

APIAKeyHi_EL1 is a 64-bit register.

Field descriptions

The APIAKeyHi_EL1 bit assignments are:



Bits [63:0]

64 bit value, bits[127:64] of the 128 bit pointer authentication key value.

This field resets to an architecturally UNKNOWN value.

Accessing the APIAKeyHi_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, APIAKeyHi_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.APIAKey == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return APIAKeyHi_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        return APIAKeyHi_EL1;
elsif PSTATE.EL == EL3 then
    return APIAKeyHi_EL1;
```

MSR APIAKeyHi_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.APIAKey == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        APIAKeyHi_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        APIAKeyHi_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    APIAKeyHi_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.21 APIAKeyLo_EL1, Pointer Authentication Key A for Instruction (bits[63:0])

The APIAKeyLo_EL1 characteristics are:

Purpose

Holds bits[63:0] of key A used for authentication of instruction pointer values.

Note

The term APIAKey_EL1 is used to describe the concatenation of APIAKeyHi_EL1: APIAKeyLo_EL1.

Configurations

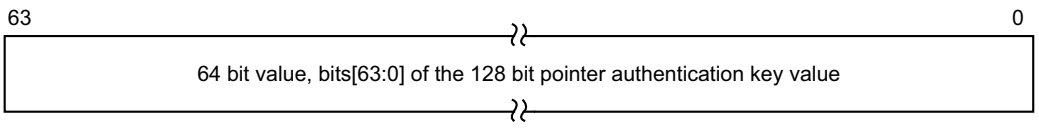
This register is present only when ARMv8.3-PAuth is implemented. Otherwise, direct accesses to APIAKeyLo_EL1 are UNDEFINED.

Attributes

APIAKeyLo_EL1 is a 64-bit register.

Field descriptions

The APIAKeyLo_EL1 bit assignments are:



Bits [63:0]

64 bit value, bits[63:0] of the 128 bit pointer authentication key value.

This field resets to an architecturally UNKNOWN value.

Accessing the APIAKeyLo_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, APIAKeyLo_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.APIAKey == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return APIAKeyLo_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        return APIAKeyLo_EL1;
elsif PSTATE.EL == EL3 then
    return APIAKeyLo_EL1;
```

MSR APIAKeyLo_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.APIAKey == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        APIAKeyLo_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        APIAKeyLo_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    APIAKeyLo_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.22 APIBKeyHi_EL1, Pointer Authentication Key B for Instruction (bits[127:64])

The APIBKeyHi_EL1 characteristics are:

Purpose

Holds bits[127:64] of key B used for authentication of instruction pointer values.

Note

The term APIBKey_EL1 is used to describe the concatenation of APIBKeyHi_EL1: APIBKeyLo_EL1.

Configurations

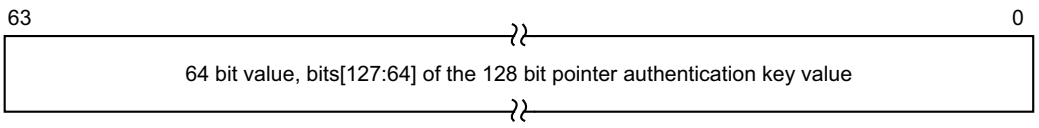
This register is present only when ARMv8.3-PAuth is implemented. Otherwise, direct accesses to APIBKeyHi_EL1 are UNDEFINED.

Attributes

APIBKeyHi_EL1 is a 64-bit register.

Field descriptions

The APIBKeyHi_EL1 bit assignments are:



Bits [63:0]

64 bit value, bits[127:64] of the 128 bit pointer authentication key value.

This field resets to an architecturally UNKNOWN value.

Accessing the APIBKeyHi_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, APIBKeyHi_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0001	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.APIBKey == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return APIBKeyHi_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        return APIBKeyHi_EL1;
elsif PSTATE.EL == EL3 then
    return APIBKeyHi_EL1;
```

MSR APIBKeyHi_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0001	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.APIBKey == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        APIBKeyHi_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        APIBKeyHi_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    APIBKeyHi_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.23 APIBKeyLo_EL1, Pointer Authentication Key B for Instruction (bits[63:0])

The APIBKeyLo_EL1 characteristics are:

Purpose

Holds bits[63:0] of key B used for authentication of instruction pointer values.

Note

The term APIBKey_EL1 is used to describe the concatenation of APIBKeyHi_EL1: APIBKeyLo_EL1.

Configurations

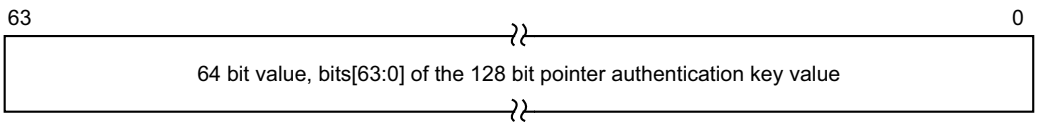
This register is present only when ARMv8.3-PAuth is implemented. Otherwise, direct accesses to APIBKeyLo_EL1 are UNDEFINED.

Attributes

APIBKeyLo_EL1 is a 64-bit register.

Field descriptions

The APIBKeyLo_EL1 bit assignments are:



Bits [63:0]

64 bit value, bits[63:0] of the 128 bit pointer authentication key value.

This field resets to an architecturally UNKNOWN value.

Accessing the APIBKeyLo_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, APIBKeyLo_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0001	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.APIBKey == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return APIBKeyLo_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        return APIBKeyLo_EL1;
elsif PSTATE.EL == EL3 then
    return APIBKeyLo_EL1;
```

MSR APIBKeyLo_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0001	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.APIBKey == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        APIBKeyLo_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.APK == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        APIBKeyLo_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    APIBKeyLo_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.24 CCSIDR2_EL1, Current Cache Size ID Register 2

The CCSIDR2_EL1 characteristics are:

Purpose

When ARMv8.3-CCIDX is implemented, provides the information about the architecture of the currently selected cache from bits[63:32] of CCSIDR_EL1.

When ARMv8.3-CCIDX is not implemented, this register is not implemented.

Configurations

AArch64 System register CCSIDR2_EL1[31:0] is architecturally mapped to AArch32 System register CCSIDR2[31:0].

This register is present only when ARMv8.3-CCIDX is implemented. Otherwise, direct accesses to CCSIDR2_EL1 are UNDEFINED.

In an AArch64 only implementation, it is IMPLEMENTATION DEFINED whether reading this register gives an UNKNOWN value or is UNDEFINED.

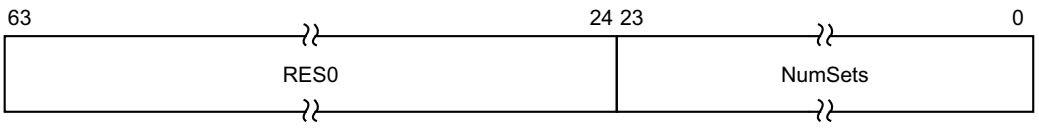
The implementation includes one CCSIDR2_EL1 for each cache that it can access. CSSELR_EL1 selects which Cache Size ID Register is accessible.

Attributes

CCSIDR2_EL1 is a 64-bit register.

Field descriptions

The CCSIDR2_EL1 bit assignments are:



Bits [63:24]

Reserved, RES0.

NumSets, bits [23:0]

(Number of sets in cache) - 1, therefore a value of 0 indicates 1 set in the cache. The number of sets does not have to be a power of 2.

Accessing the CCSIDR2_EL1

If CSSELR_EL1.Level is programmed to a cache level that is not implemented, then on a read of the CCSIDR2_EL1 the behavior is CONSTRAINED UNPREDICTABLE, and can be one of the following:

- The CCSIDR2_EL1 read is treated as NOP.
- The CCSIDR2_EL1 read is UNDEFINED.
- The CCSIDR2_EL1 read returns an UNKNOWN value.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, CCSIDR2_EL1

op0	op1	CRn	CRm	op2
0b11	0b001	0b0000	0b0000	0b010

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID2 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID4 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return CCSIDR2_EL1;
elsif PSTATE.EL == EL2 then
    return CCSIDR2_EL1;
elsif PSTATE.EL == EL3 then
    return CCSIDR2_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.25 CCSIDR_EL1, Current Cache Size ID Register

The CCSIDR_EL1 characteristics are:

Purpose

Provides information about the architecture of the currently selected cache.

Configurations

AArch64 System register CCSIDR_EL1[31:0] is architecturally mapped to AArch32 System register CCSIDR[31:0].

AArch64 System register CCSIDR_EL1[63:32] is architecturally mapped to AArch32 System register CCSIDR2[31:0].

The implementation includes one CCSIDR_EL1 for each cache that it can access. CSSELR_EL1 selects which Cache Size ID Register is accessible.

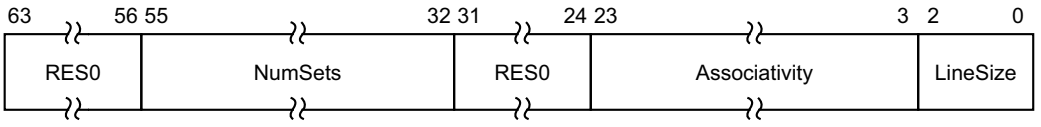
Attributes

CCSIDR_EL1 is a 64-bit register.

Field descriptions

The CCSIDR_EL1 bit assignments are:

When ARMv8.3-CCIDX is implemented:



Note

The parameters NumSets, Associativity, and LineSize in these registers define the architecturally visible parameters that are required for the cache maintenance by Set/Way instructions. They are not guaranteed to represent the actual microarchitectural features of a design. You cannot make any inference about the actual sizes of caches based on these parameters.

Bits [63:56]

Reserved, RES0.

NumSets, bits [55:32]

(Number of sets in cache) - 1, therefore a value of 0 indicates 1 set in the cache. The number of sets does not have to be a power of 2.

Bits [31:24]

Reserved, RES0.

Associativity, bits [23:3]

(Associativity of cache) - 1, therefore a value of 0 indicates an associativity of 1. The associativity does not have to be a power of 2.

LineSize, bits [2:0]

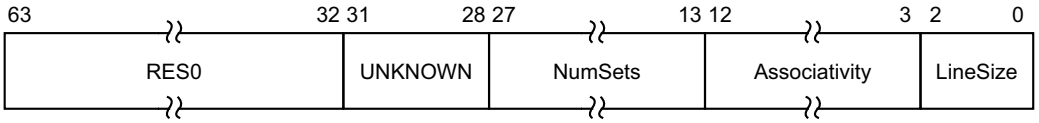
(Log_2(Number of bytes in cache line)) - 4. For example:

- For a line length of 16 bytes: $Log_2(16) = 4$, LineSize entry = 0. This is the minimum line length.
- For a line length of 32 bytes: $Log_2(32) = 5$, LineSize entry = 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When ARMv8.5-MemTag is implemented and enabled, where a cache only holds Allocation tags, this field is RES0.

Otherwise:



Note

The parameters NumSets, Associativity, and LineSize in these registers define the architecturally visible parameters that are required for the cache maintenance by Set/Way instructions. They are not guaranteed to represent the actual microarchitectural features of a design. You cannot make any inference about the actual sizes of caches based on these parameters.

Bits [63:32]

Reserved, RES0.

UNKNOWN, bits [31:28]

Reserved, UNKNOWN.

NumSets, bits [27:13]

(Number of sets in cache) - 1, therefore a value of 0 indicates 1 set in the cache. The number of sets does not have to be a power of 2.

Associativity, bits [12:3]

(Associativity of cache) - 1, therefore a value of 0 indicates an associativity of 1. The associativity does not have to be a power of 2.

LineSize, bits [2:0]

(Log_2(Number of bytes in cache line)) - 4. For example:

- For a line length of 16 bytes: $Log_2(16) = 4$, LineSize entry = 0. This is the minimum line length.
- For a line length of 32 bytes: $Log_2(32) = 5$, LineSize entry = 1.

Accessing the CCSIDR_EL1

If CSSELR_EL1.Level is programmed to a cache level that is not implemented, then on a read of the CCSIDR_EL1 the behavior is CONSTRAINED UNPREDICTABLE, and can be one of the following:

- The CCSIDR_EL1 read is treated as NOP.
- The CCSIDR_EL1 read is UNDEFINED.
- The CCSIDR_EL1 read returns an UNKNOWN value.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, CCSIDR_EL1

op0	op1	CRn	CRm	op2
0b11	0b001	0b0000	0b0000	0b000

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID2 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID4 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.CCSIDR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return CCSIDR_EL1;
elsif PSTATE.EL == EL2 then
    return CCSIDR_EL1;
elsif PSTATE.EL == EL3 then
    return CCSIDR_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.26 CLIDR_EL1, Cache Level ID Register

The CLIDR_EL1 characteristics are:

Purpose

Identifies the type of cache, or caches, that are implemented at each level and can be managed using the architected cache maintenance instructions that operate by set/way, up to a maximum of seven levels. Also identifies the Level of Coherence (LoC) and Level of Unification (LoU) for the cache hierarchy.

Configurations

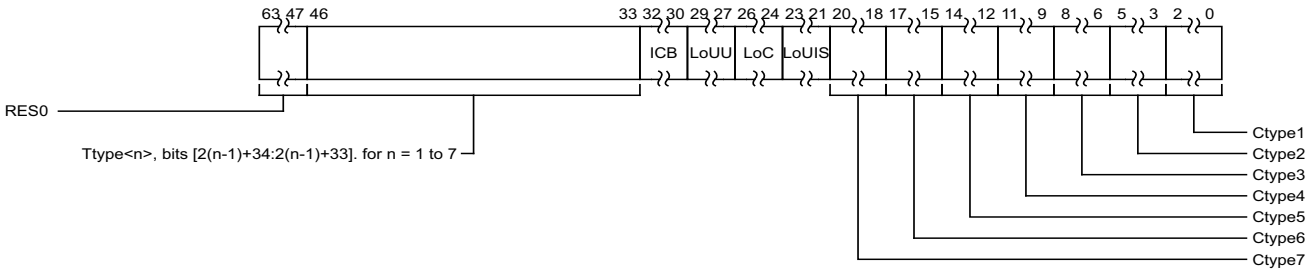
AArch64 System register CLIDR_EL1[31:0] is architecturally mapped to AArch32 System register CLIDR[31:0].

Attributes

CLIDR_EL1 is a 64-bit register.

Field descriptions

The CLIDR_EL1 bit assignments are:



Bits [63:47]

Reserved, RES0.

Ttype<n>, bits [2(n-1)+34:2(n-1)+33], for n = 1 to 7

Tag cache type. Indicate the type of cache that is implemented and can be managed using the architected cache maintenance instructions that operate by set/way at each level, from Level 1 up to a maximum of seven levels of cache hierarchy.

`0b00` No Tag Cache.

`0b01` Separate Allocation Tag Cache.

`0b10` Unified Allocation Tag and Data cache, Allocation Tags and Data in unified lines.

`0b11` Unified Allocation Tag and Data cache, Allocation Tags and Data in separate lines.

ICB, bits [32:30]

Inner cache boundary. This field indicates the boundary for caching Inner Cacheable memory regions.

The possible values are:

`0b000` Not disclosed by this mechanism.

`0b001` L1 cache is the highest Inner Cacheable level.

`0b010` L2 cache is the highest Inner Cacheable level.

`0b011` L3 cache is the highest Inner Cacheable level.

`0b100` L4 cache is the highest Inner Cacheable level.

`0b101` L5 cache is the highest Inner Cacheable level.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b110` L6 cache is the highest Inner Cacheable level.

`0b111` L7 cache is the highest Inner Cacheable level.

LoUU, bits [29:27]

Level of Unification Uniprocessor for the cache hierarchy.

LoC, bits [26:24]

Level of Coherence for the cache hierarchy.

LoUIS, bits [23:21]

Level of Unification Inner Shareable for the cache hierarchy.

Ctype<n>, bits [3(n-1)+2:3(n-1)], for n = 1 to 7

Cache Type fields. Indicate the type of cache that is implemented and can be managed using the architected cache maintenance instructions that operate by set/way at each level, from Level 1 up to a maximum of seven levels of cache hierarchy. Possible values of each field are:

`0b000` No cache.

`0b001` Instruction cache only.

`0b010` Data cache only.

`0b011` Separate instruction and data caches.

`0b100` Unified cache.

All other values are reserved.

If software reads the Cache Type fields from Ctype1 upwards, once it has seen a value of 000, no caches that can be managed using the architected cache maintenance instructions that operate by set/way exist at further-out levels of the hierarchy. So, for example, if Ctype3 is the first Cache Type field with a value of 000, the values of Ctype4 to Ctype7 must be ignored.

Accessing the CLIDR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, CLIDR_EL1

op0	op1	CRn	CRm	op2
0b11	0b001	0b0000	0b0000	0b001

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID2 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID4 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.CLIDR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return CLIDR_EL1;
elsif PSTATE.EL == EL2 then
    return CLIDR_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    return CLIDR_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.27 CONTEXTIDR_EL1, Context ID Register (EL1)

The CONTEXTIDR_EL1 characteristics are:

Purpose

Identifies the current Process Identifier.

The value of the whole of this register is called the Context ID and is used by:

- The debug logic, for Linked and Unlinked Context ID matching.
- The trace logic, to identify the current process.

The significance of this register is for debug and trace use only.

This register is used when ARMv8.1-VHE is not implemented, or when ARMv8.1-VHE is implemented and HCR_EL2.E2H is set to 0.

Note

When ARMv8.1-VHE is implemented and HCR_EL2.E2H is set to 1, CONTEXTIDR_EL2 is used.

Configurations

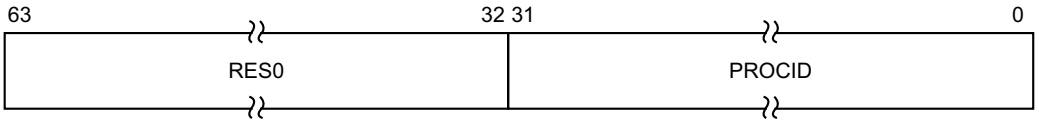
AArch64 System register CONTEXTIDR_EL1[31:0] is architecturally mapped to AArch32 System register CONTEXTIDR[31:0].

Attributes

CONTEXTIDR_EL1 is a 64-bit register.

Field descriptions

The CONTEXTIDR_EL1 bit assignments are:



Bits [63:32]

Reserved, RES0.

PROCID, bits [31:0]

Process Identifier. This field must be programmed with a unique value that identifies the current process.

Note

In AArch32 state, when TTBCR.EAE is set to 0, CONTEXTIDR.ASID holds the ASID.

In AArch64 state, CONTEXTIDR_EL1 is independent of the ASID, and for the EL1&0 translation regime either TTBR0_EL1 or TTBR1_EL1 holds the ASID.

This field resets to an architecturally UNKNOWN value.

Accessing the CONTEXTIDR_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic CONTEXTIDR_EL1 or CONTEXTIDR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, CONTEXTIDR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.CONTEXTIDR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x108];
    else
        return CONTEXTIDR_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return CONTEXTIDR_EL2;
    else
        return CONTEXTIDR_EL1;
elsif PSTATE.EL == EL3 then
    return CONTEXTIDR_EL1;
```

MSR CONTEXTIDR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.CONTEXTIDR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x108] = X[t];
    else
        CONTEXTIDR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        CONTEXTIDR_EL2 = X[t];
    else
        CONTEXTIDR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    CONTEXTIDR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, CONTEXTIDR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b1101	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        return NVMem[0x108];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return CONTEXTIDR_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        return CONTEXTIDR_EL1;
    else
        UNDEFINED;
```

MSR CONTEXTIDR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b1101	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        NVMem[0x108] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        CONTEXTIDR_EL1 = X[t];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        CONTEXTIDR_EL1 = X[t];
    else
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.28 CONTEXTIDR_EL2, Context ID Register (EL2)

The CONTEXTIDR_EL2 characteristics are:

Purpose

Provides a current Process Identifier for EL2.

The value of the whole of this register is called the Context ID and is used by:

- The debug logic, for Linked and Unlinked Context ID matching.
- The trace logic, to identify the current process.

The significance of this register is for debug and trace use only.

Configurations

This register is present only when ARMv8.1-VHE is implemented or ARMv8.2-Debug is implemented. Otherwise, direct accesses to CONTEXTIDR_EL2 are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

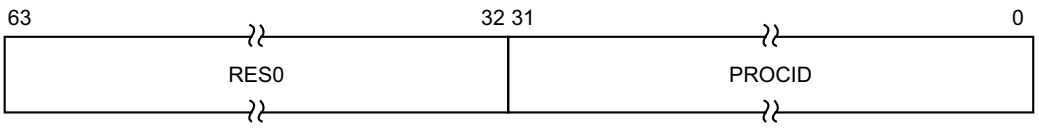
This register has no effect if EL2 is not enabled in the current Security state.

Attributes

CONTEXTIDR_EL2 is a 64-bit register.

Field descriptions

The CONTEXTIDR_EL2 bit assignments are:



Bits [63:32]

Reserved, RES0.

PROCID, bits [31:0]

Process Identifier. This field must be programmed with a unique value that identifies the current process.

Note

In AArch32 state, when TTBCR.EAE is set to 0, CONTEXTIDR.ASID holds the ASID.

In AArch64 state, CONTEXTIDR_EL2 is independent of the ASID, and for the EL2&0 translation regime either TTBR0_EL2 or TTBR1_EL2 holds the ASID.

This field resets to an architecturally UNKNOWN value.

Accessing the CONTEXTIDR_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic CONTEXTIDR_EL2 or CONTEXTIDR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, CONTEXTIDR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1101	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return CONTEXTIDR_EL2;
elsif PSTATE.EL == EL3 then
    return CONTEXTIDR_EL2;
```

MSR CONTEXTIDR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1101	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    CONTEXTIDR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    CONTEXTIDR_EL2 = X[t];
```

MRS <Xt>, CONTEXTIDR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.CONTEXTIDR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x108];
    else
        return CONTEXTIDR_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return CONTEXTIDR_EL2;
    else
        return CONTEXTIDR_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    return CONTEXTIDR_EL1;
```

MSR CONTEXTIDR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.CONTEXTIDR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x108] = X[t];
    else
        CONTEXTIDR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        CONTEXTIDR_EL2 = X[t];
    else
        CONTEXTIDR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    CONTEXTIDR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.29 CPACR_EL1, Architectural Feature Access Control Register

The CPACR_EL1 characteristics are:

Purpose

Controls access to trace, SVE, Advanced SIMD and floating-point functionality.

Configurations

AArch64 System register CPACR_EL1[31:0] is architecturally mapped to AArch32 System register CPACR[31:0].

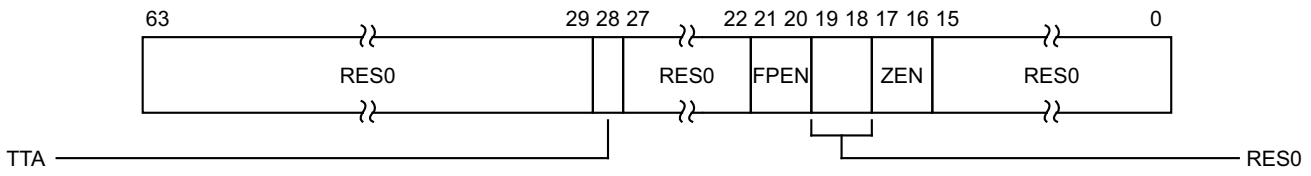
When HCR_EL2.{E2H, TGE} == {1, 1}, the fields in this register have no effect on execution at EL0 and EL1. In this case, the controls provided by CPTR_EL2 are used.

Attributes

CPACR_EL1 is a 64-bit register.

Field descriptions

The CPACR_EL1 bit assignments are:



Bits [63:29]

Reserved, RES0.

TTA, bit [28]

Traps EL0 and EL1 System register accesses to all implemented trace registers to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from both Execution states as follows:

- In AArch64 state, accesses to trace registers are trapped, reported using EC syndrome value `0x18`.
- In AArch32 state, MRC and MCR accesses to trace registers are trapped, reported using EC syndrome value `0x05`.
- In AArch32 state, MRRC and MCRR accesses to trace registers are trapped, reported using EC syndrome value `0x0C`.

`0b0` This control does not cause any instructions to be trapped.

`0b1` This control causes EL0 and EL1 System register accesses to all implemented trace registers to be trapped.

Note

- The ETMv4 architecture does not permit EL0 to access the trace registers. If the Armv8-A architecture is implemented with an ETMv4 implementation, EL0 accesses to the trace registers are UNDEFINED, and any resulting exception is higher priority than an exception that would be generated because the value of CPACR_EL1.TTA is 1.
- The Armv8-A architecture does not provide traps on trace register accesses through the optional memory-mapped interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If System register access to the trace functionality is not implemented, this bit is RES0.

This field resets to an architecturally UNKNOWN value.

Bits [27:22]

Reserved, RES0.

FPEN, bits [21:20]

Traps EL0 and EL1 accesses to the SVE, Advanced SIMD, and floating-point registers to EL1, reported using EC syndrome value `0x07`, or to EL2 reported using EC syndrome value `0x00`, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from both Execution states as follows:

- In AArch64 state, accesses to FPCR, FPSR, any of the SIMD and floating-point registers V0-V31, including their views as D0-D31 registers or S0-31 registers. See *The SIMD and floating-point registers, V0-V31* on page D1-2324.
- FPSCR, and any of the SIMD and floating-point registers Q0-15, including their views as D0-D31 registers or S0-31 registers. See *Advanced SIMD and floating-point System registers* on page G1-5774.

`0b00` This control causes any instructions at EL0 or EL1 that use the registers associated with SVE, Advanced SIMD and floating-point execution to be trapped, unless they are trapped by CPACR_EL1.ZEN.

`0b01` This control causes any instructions at EL0 that use the registers associated with SVE, Advanced SIMD and floating-point execution to be trapped, unless they are trapped by CPACR_EL1.ZEN, but does not cause any instruction at EL1 to be trapped.

`0b10` This control causes any instructions at EL0 or EL1 that use the registers associated with SVE, Advanced SIMD and floating-point execution to be trapped, unless they are trapped by CPACR_EL1.ZEN.

`0b11` This control does not cause any instructions to be trapped.

Writes to MVFR0, MVFR1 and MVFR2 from EL1 or higher are CONSTRAINED UNPREDICTABLE and whether these accesses can be trapped by this control depends on implemented CONSTRAINED UNPREDICTABLE behavior.

Note

- Attempts to write to the FPSID count as use of the registers for accesses from EL1 or higher.
- Accesses from EL0 to FPSID, MVFR0, MVFR1, MVFR2, and FPEXC are UNDEFINED, and any resulting exception is higher priority than an exception that would be generated because the value of CPACR_EL1.FPEN is not `0b11`.

This field resets to an architecturally UNKNOWN value.

Bits [19:18]

Reserved, RES0.

ZEN, bits [17:16]

When SVE is implemented:

Traps SVE instructions and instructions that access SVE System registers at EL0 and EL1 to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1.

`0b00` This control causes these instructions executed at EL0 or EL1 to be trapped.

`0b01` This control causes these instructions executed at EL0 to be trapped, but does not cause any instruction at EL1 to be trapped.

`0b10` This control causes these instructions executed at EL0 or EL1 to be trapped.

`0b11` This control does not cause any instruction to be trapped.

If SVEis not implemented, this field is RES0.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

Reserved, RES0.

Bits [15:0]

Reserved, RES0.

Accessing the CPACR_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic CPACR_EL1 or CPACR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, CPACR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.CPACR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x100];
    else
        return CPACR_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        return CPTR_EL2;
    else
        return CPACR_EL1;
elsif PSTATE.EL == EL3 then
    return CPACR_EL1;
```

MSR CPACR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.CPACR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x100] = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        CPACR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        CPTR_EL2 = X[t];
    else
        CPACR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    CPACR_EL1 = X[t];
```

MRS <Xt>, CPACR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        return NVMem[0x100];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            return CPACR_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        return CPACR_EL1;
    else
        UNDEFINED;
```

MSR CPACR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        NVMem[0x100] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            CPACR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        else
            UNDEFINED;
    elsif PSTATE.EL == EL3 then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
            CPACR_EL1 = X[t];
        else
            UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.30 CPTR_EL2, Architectural Feature Trap Register (EL2)

The CPTR_EL2 characteristics are:

Purpose

Controls:

- Trapping to EL2 of access to CPACR, CPACR_EL1, trace functionality, and to SVE, Advanced SIMD and floating-point functionality.
- EL2 access to trace functionality, and to SVE, Advanced SIMD and floating-point functionality.

Configurations

AArch64 System register CPTR_EL2[31:0] is architecturally mapped to AArch32 System register HCPTR[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

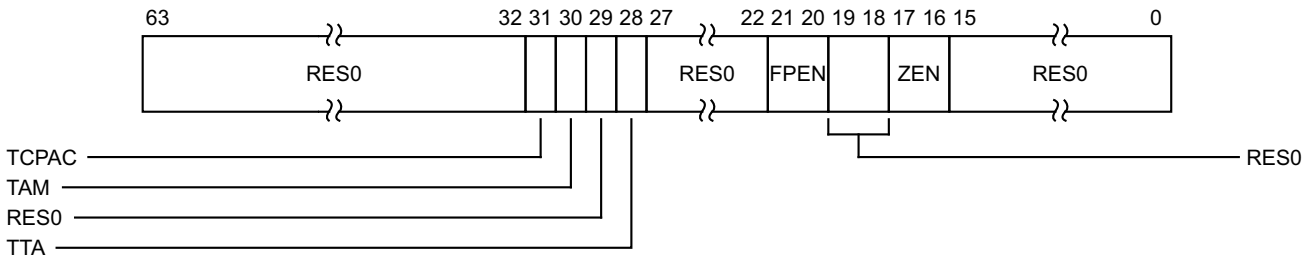
Attributes

CPTR_EL2 is a 64-bit register.

Field descriptions

The CPTR_EL2 bit assignments are:

When ARMv8.1-VHE is implemented and HCR_EL2.E2H == 1:



Bits [63:32]

Reserved, RES0.

TCPAC, bit [31]

When HCR_EL2.TGE is 0, traps EL1 accesses to CPACR_EL1 reported using EC syndrome value `0x18`, and accesses to CPACR reported using EC syndrome value `0x03`, to EL2 when EL2 is enabled in the current Security state.

`0b0` This control does not cause any instructions to be trapped.

`0b1` EL1 accesses to CPACR_EL1 and CPACR are trapped to EL2 when EL2 is enabled in the current Security state.

When HCR_EL2.TGE is 1, this control does not cause any instructions to be trapped.

Note

CPACR_EL1 and CPACR are not accessible at EL0.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TAM, bit [30]

When AMUv1 is implemented:

Trap Activity Monitor access. Traps EL1 and EL0 accesses to all Activity Monitor registers to EL2, as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value `0x18`:
  - AMUSERENR_EL0, AMCFGR_EL0, AMCGCR_EL0, AMCNTENCLR0_EL0, AMCNTENCLR1_EL0, AMCNTENSET0_EL0, AMCNTENSET1_EL0, AMCR_EL0, AMEVCNTR0<n>_EL0, AMEVCNTR1<n>_EL0, AMEVTYPER0<n>_EL0, and AMEVTYPER1<n>_EL0.
- In AArch32 state, MRC or MCR accesses to the following registers are trapped to EL2 and reported using EC syndrome value `0x03`:
  - AMUSERENR, AMCFGR, AMCGCR, AMCNTENCLR0, AMCNTENCLR1, AMCNTENSET0, AMCNTENSET1, AMCR, AMEVTYPER0<n>, and AMEVTYPER1<n>.
- In AArch32 state, MRRC or MCRR accesses to AMEVCNTR0<n> and AMEVCNTR1<n>, are trapped to EL2, reported using EC syndrome value `0x04`.

`0b0` Accesses from EL1 and EL0 to Activity Monitor registers are not trapped.

`0b1` Accesses from EL1 and EL0 to Activity Monitor registers are trapped to EL2, when EL2 is enabled in the current Security state.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [29]

Reserved, RES0.

TTA, bit [28]

Traps System register accesses to all implemented trace registers to EL2 when EL2 is enabled in the current Security state, from both Execution states, as follows:

- In AArch64 state, accesses to trace registers with op0=2, op1=1 are trapped to EL2, reported using EC syndrome value `0x18`.
- In AArch32 state, MRC or MCR accesses to trace registers with cpnum=14, opc1=1, are trapped to EL2, reported using EC syndrome value `0x05`.
- In AArch32 state, MRRC or MCRR accesses to trace registers with cpnum=14, opc1=1, are trapped to EL2, reported using EC syndrome value `0x0C`.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Any attempt at EL0, EL1 or EL2, to execute a System register access to an implemented trace register is trapped to EL2 when EL2 is enabled in the current Security state, unless HCR_EL2.TGE is 0 and it is trapped by CPACR.NSTRCDIS or CPACR_EL1.TTA.

When HCR_EL2.TGE is 1, any attempt at EL0 or EL2 to execute a System register access to an implemented trace register is trapped to EL2 when EL2 is enabled in the current Security state.

Note

- The ETMv4 architecture does not permit EL0 to access the trace registers. If the Armv8-A architecture is implemented with an ETMv4 implementation, EL0 accesses to the trace registers are UNDEFINED, and any resulting exception is higher priority than an exception that would be generated because the value of CPTR_EL2.TTA is 1.
- EL2 does not provide traps on trace register accesses through the optional memory-mapped interface.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

System register accesses to the trace registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

If System register access to the trace functionality is not supported, this bit is RES0.

This field resets to an architecturally UNKNOWN value.

Bits [27:22]

Reserved, RES0.

FPEN, bits [21:20]

Traps EL0, EL2 and, when HCR_EL2.TGE is 0, EL1 accesses to the SVE, Advanced SIMD and floating-point registers to EL2 when EL2 is enabled in the current Security state, from both Execution states.

`0b00` This control causes any instructions at EL0, EL1, or EL2 that use the registers associated with SVE, Advanced SIMD and floating-point execution to be trapped, subject to the exception prioritization rules, unless they are trapped by CPTR_EL2.ZEN.

`0b01` When HCR_EL2.TGE is 0, this control does not cause any instructions to be trapped.

When HCR_EL2.TGE is 1, this control causes instructions at EL0 that use the registers associated with SVE, Advanced SIMD and floating-point execution to be trapped, unless they are trapped by CPTR_EL2.ZEN, but does not cause any instruction at EL2 to be trapped.

`0b10` This control causes any instructions at EL0, EL1, or EL2 that use the registers associated with SVE, Advanced SIMD and floating-point execution to be trapped, subject to the exception prioritization rules, unless they are trapped by CPTR_EL2.ZEN.

`0b11` This control does not cause any instructions to be trapped.

Writes to MVFR0, MVFR1, and MVFR2 from EL1 or higher are CONSTRAINED UNPREDICTABLE and whether these accesses can be trapped by this control depends on implemented CONSTRAINED UNPREDICTABLE behavior.

Note

- Attempts to write to the FPSID count as use of the registers for accesses from EL1 or higher.
- Accesses from EL0 to FPSID, MVFR0, MVFR1, MVFR2, and FPEXC are UNDEFINED, and any resulting exception is higher priority than an exception that would be generated because the value of CPTR_EL2.FPEN is not `0b11`.

This field resets to an architecturally UNKNOWN value.

Bits [19:18]

Reserved, RES0.

ZEN, bits [17:16]

When SVE is implemented:

Traps execution at EL2, EL1, and EL0 of SVE instructions or instructions that access SVE System registers to EL2 when EL2 is enabled in the current Security state.

`0b00` This control causes execution at EL2, EL1, and EL0 of these instructions to be trapped, subject to the exception prioritization rules.

`0b01` When HCR_EL2.TGE is 0, this control does not cause any instruction to be trapped.

When HCR_EL2.TGE is 1, this control causes these instructions executed at EL0 to be trapped, but does not cause any instruction at EL2 to be trapped.

`0b10` This control causes execution at EL2, EL1, and EL0 of these instructions to be trapped, subject to the exception prioritization rules.

`0b11` This control does not cause any instruction to be trapped.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

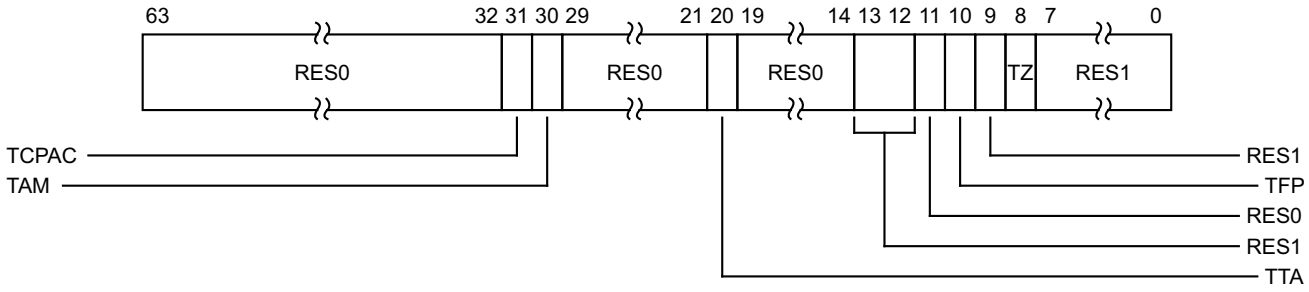
Otherwise:

Reserved, RES0.

Bits [15:0]

Reserved, RES0.

Otherwise:



This format applies in all Armv8.0 implementations.

Bits [63:32]

Reserved, RES0.

TCPAC, bit [31]

Traps EL1 accesses to CPACR_EL1, reported using EC syndrome value `0x18` and accesses to CPACR, reported using EC syndrome value `0x03`, to EL2 when EL2 is enabled in the current Security state.

`0b0` This control does not cause any instructions to be trapped.

`0b1` EL1 accesses to CPACR_EL1 and CPACR are trapped to EL2 when EL2 is enabled in the current Security state.

When HCR_EL2.TGE is 1, this control does not cause any instructions to be trapped.

Note

CPACR_EL1 and CPACR are not accessible at EL0.

This field resets to an architecturally UNKNOWN value.

TAM, bit [30]

Trap Activity Monitor access. Traps EL1 and EL0 accesses to all Activity Monitor registers to EL2, as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value `0x18`:
  - AMUSERENR_EL0, AMCFGR_EL0, AMCGCR_EL0, AMCNTENCLR0_EL0, AMCNTENCLR1_EL0, AMCNTENSET0_EL0, AMCNTENSET1_EL0, AMCR_EL0, AMEVCNTR0<n>_EL0, AMEVCNTR1<n>_EL0, AMEVTYPER0<n>_EL0, and AMEVTYPER1<n>_EL0.
- In AArch32 state, MCR or MRC accesses to the following registers are trapped to EL2 and reported using EC syndrome value `0x03`:
  - AMUSERENR, AMCFGR, AMCGCR, AMCNTENCLR0, AMCNTENCLR1, AMCNTENSET0, AMCNTENSET1, AMCR, AMEVTYPER0<n>, and AMEVTYPER1<n>.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- In AArch32 state, MCRR or MRRC accesses to AMEVCNTR0<n> and AMEVCNTR1<n>, are trapped to EL2, reported using EC syndrome value `0x04`.

`0b0` Accesses from EL1 and EL0 to Activity Monitor registers are not trapped.

`0b1` Accesses from EL1 and EL0 to Activity Monitor registers are trapped to EL2, when EL2 is enabled in the current Security state.

This field resets to an architecturally UNKNOWN value.

Bits [29:21]

Reserved, RES0.

TTA, bit [20]

Traps System register accesses to all implemented trace registers to EL2 when EL2 is enabled in the current Security state, from both Execution states as follows:

- In AArch64 state, accesses to trace registers with op0=2, op1=1 are trapped to EL2, reported using EC syndrome value `0x18`.
- In AArch32 state, MRC or MCR accesses to trace registers with cpnum=14, opc1=1 are trapped to EL2, reported using EC syndrome value `0x05`.
- In AArch32 state, MRRC or MCRR accesses to trace registers with cpnum=14, opc1=1 are trapped to EL2, reported using EC syndrome value `0x0C`.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Any attempt at EL0, EL1, or EL2, to execute a System register access to an implemented trace register is trapped to EL2 when EL2 is enabled in the current Security state, unless it is trapped by CPACR.TRCDIS or CPACR_EL1.TTA.

Note

- The ETMv4 architecture does not permit EL0 to access the trace registers. If the Armv8-A architecture is implemented with an ETMv4 implementation, EL0 accesses to the trace registers are UNDEFINED, and any resulting exception is higher priority than an exception that would be generated because the value of CPTR_EL2.TTA is 1.
- EL2 does not provide traps on trace register accesses through the optional memory-mapped interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

If System register access to the trace functionality is not supported, this bit is RES0.

This field resets to an architecturally UNKNOWN value.

Bits [19:14]

Reserved, RES0.

Bits [13:12]

Reserved, RES1.

Bit [11]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TFP, bit [10]

Traps accesses to SVE, Advanced SIMD and floating-point functionality to EL2 when EL2 is enabled in the current Security state, from both Execution states, as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value `0x07`:
  - FPCR, FPSR, FPEXC32_EL2, any of the SIMD and floating-point registers V0-V31, including their views as D0-D31 registers or S0-31 registers. See *The SIMD and floating-point registers, V0-V31* on page D1-2324.
- In AArch32 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value `0x07`:
  - MVFR0, MVFR1, MVFR2, FPSCR, FPEXC, and any of the SIMD and floating-point registers Q0-15, including their views as D0-D31 registers or S0-31 registers. See *Advanced SIMD and floating-point System registers* on page G1-5774. For the purposes of this trap, the architecture defines a VMSR access to FPSID from EL1 or higher as an access to a SIMD and floating point register. Otherwise, permitted VMSR accesses to FPSID are ignored.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Any attempt at EL0, EL1 or EL2, to execute an instruction that uses the registers associated with SVE, Advanced SIMD and floating-point execution is trapped to EL2 when EL2 is enabled in the current Security state, subject to the exception prioritization rules, unless it is trapped by CPTR_EL2.TZ.

Note

FPEXC32_EL2 is not accessible from EL0 using AArch64.

FPSID, MVFR0, MVFR1, and FPEXC are not accessible from EL0 using AArch32.

This field resets to an architecturally UNKNOWN value.

Bit [9]

Reserved, RES1.

TZ, bit [8]

Traps execution at EL2, EL1, or EL0 of SVE instructions and instructions that access SVE System registers to EL2 when EL2 is enabled in the current Security state.

`0b0` This control does not cause any instruction to be trapped.

`0b1` This control causes these instructions to be trapped, subject to the exception prioritization rules.

This field resets to an architecturally UNKNOWN value.

Bits [7:0]

Reserved, RES1.

Accessing the CPTR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, CPTR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return CPTR_EL2;
elsif PSTATE.EL == EL3 then
    return CPTR_EL2;
```

MSR CPTR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        CPTR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    CPTR_EL2 = X[t];
```

MRS <Xt>, CPACR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.CPACR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x100];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        return CPACR_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        return CPTR_EL2;
    else
        return CPACR_EL1;
elsif PSTATE.EL == EL3 then
    return CPACR_EL1;
```

MSR CPACR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.CPACR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x100] = X[t];
    else
        CPACR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        CPTR_EL2 = X[t];
    else
        CPACR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    CPACR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.31 CPTR_EL3, Architectural Feature Trap Register (EL3)

The CPTR_EL3 characteristics are:

Purpose

Controls trapping to EL3 of access to CPACR_EL1, CPTR_EL2, trace functionality and registers associated with SVE, Advanced SIMD and floating-point execution. Also controls EL3 access to trace functionality and registers associated with SVE, Advanced SIMD and floating-point execution.

Configurations

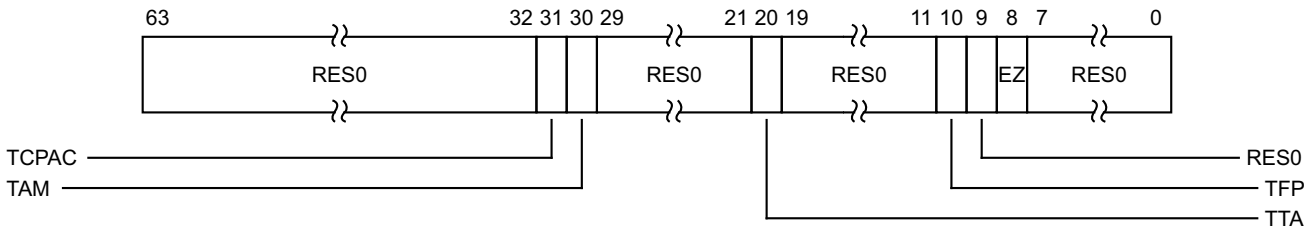
This register is present only when EL3 is implemented. Otherwise, direct accesses to CPTR_EL3 are UNDEFINED.

Attributes

CPTR_EL3 is a 64-bit register.

Field descriptions

The CPTR_EL3 bit assignments are:



Bits [63:32]

Reserved, RES0.

TCPAC, bit [31]

Traps all of the following to EL3, from both Security states and both Execution states.

- EL2 accesses to CPTR_EL2, reported using EC syndrome value `0x18`, or HCPTR, reported using EC syndrome value `0x03`.
- EL2 and EL1 accesses to CPACR_EL1 reported using EC syndrome value `0x18`, or CPACR reported using EC syndrome value `0x03`.

When CPTR_EL3.TCPAC is:

`0b0` This control does not cause any instructions to be trapped.

`0b1` EL2 accesses to the CPTR_EL2 or HCPTR, and EL2 and EL1 accesses to the CPACR_EL1 or CPACR, are trapped to EL3, unless they are trapped by CPTR_EL2.TCPAC.

This field resets to an architecturally UNKNOWN value.

TAM, bit [30]

When AMUv1 is implemented:

Trap Activity Monitor access. Traps EL2, EL1 and EL0 accesses to all Activity Monitor registers to EL3.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accesses to the Activity Monitors registers are trapped as follows:

- In AArch64 state, the following registers are trapped to EL3 and reported with EC syndrome value `0x18`:
  - AMUSERENR_EL0, AMCFGR_EL0, AMCGCR_EL0, AMCNTENCLR0_EL0, AMCNTENCLR1_EL0, AMCNTENSET0_EL0, AMCNTENSET1_EL0, AMCR_EL0, AMEVCNTR0<n>_EL0, AMEVCNTR1<n>_EL0, AMEVTYPER0<n>_EL0, and AMEVTYPER1<n>_EL0.
- In AArch32 state, accesses with MRC or MCR to the following registers reported with EC syndrome value `0x03`:
  - AMUSERENR, AMCFGR, AMCGCR, AMCNTENCLR0, AMCNTENCLR1, AMCNTENSET0, AMCNTENSET1, AMCR, AMEVTYPER0<n>, and AMEVTYPER1<n>.
- In AArch32 state, accesses with MRRC or MCRR to the following registers, reported with EC syndrome value `0x04`:
  - AMEVCNTR0<n>, AMEVCNTR1<n>.

`0b0` Accesses from EL2, EL1, and EL0 to Activity Monitor registers are not trapped.

`0b1` Accesses from EL2, EL1, and EL0 to Activity Monitor registers are trapped to EL3.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [29:21]

Reserved, RES0.

TTA, bit [20]

Traps System register accesses. Accesses to the trace registers, from all Exception levels, both Security states, and both Execution states are trapped to EL3 as follows:

- In AArch64 state, Trace registers with op0=2, op1=1, are trapped to EL3 and reported using EC syndrome value `0x18`.
- In AArch32 state, accesses using MCR or MRC to the Trace registers with cpnum=14 and opc1=1 are reported using EC syndrome value `0x05`.
- In AArch32 state, accesses using MCRR or MRRC to the Trace registers with cpnum=14 and opc1=1 are reported using EC syndrome value `0x0C`.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Any System register access to the trace registers is trapped to EL3, subject to the exception prioritization rules, unless it is trapped by CPACR.TRCDIS, CPACR_EL1.TTA or CPTR_EL2.TTA.

If System register access to trace functionality is not supported, this bit is RES0.

Note

The ETMv4 architecture does not permit EL0 to access the trace registers. If the Armv8-A architecture is implemented with an ETMv4 implementation, EL0 accesses to the trace registers are UNDEFINED, and any resulting exception is higher priority than this trap exception.

EL3 does not provide traps on trace register accesses through the Memory-mapped interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, no side-effects occur before the exception is taken, see *Traps on instructions* on page D1-2374.

This field resets to an architecturally UNKNOWN value.

Bits [19:11]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TFP, bit [10]

Traps all accesses to SVE, Advanced SIMD and floating-point functionality, from all Exception levels, both Security states, and both Execution states, to EL3. Defined values are:

This includes the following registers, all reported using EC syndrome value `0x07`:

- FPCR, FPSR, FPEXC32_EL2, any of the SIMD and floating-point registers V0-V31, including their views as D0-D31 registers or S0-31 registers. See *The SIMD and floating-point registers, V0-V31* on page D1-2324.
- MVFR0, MVFR1, MVFR2, FPSCR, FPEXC, and any of the SIMD and floating-point registers Q0-15, including their views as D0-D31 registers or S0-31 registers. See *Advanced SIMD and floating-point System registers* on page G1-5774.

Permitted VMSR accesses to FPSID are ignored, but for the purposes of this trap the architecture define a VMSR access to the FPSID from EL1 or higher as an access to a SIMD and floating-point register.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Any attempt at any Exception level to execute an instruction that uses the registers associated with SVE, Advanced SIMD and floating-point is trapped to EL3, subject to the exception prioritization rules, unless it is trapped by CPTR_EL3.EZ.

Note

FPEXC32_EL2 is not accessible from EL0 using AArch64.

FPSID, MVFR0, MVFR1, and FPEXC are not accessible from EL0 using AArch32.

This field resets to an architecturally UNKNOWN value.

Bit [9]

Reserved, RES0.

EZ, bit [8]

When SVE is implemented:

Traps all accesses to SVE functionality and registers from all Exception levels, and both Security states, to EL3.

`0b0` This control causes these instructions executed at any Exception level to be trapped, subject to the exception prioritization rules.

`0b1` This control does not cause any instruction to be trapped.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [7:0]

Reserved, RES0.

Accessing the CPTR_EL3

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, CPTR_EL3

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0001	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    return CPTR_EL3;
```

MSR CPTR_EL3, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0001	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    CPTR_EL3 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.32 CSSELR_EL1, Cache Size Selection Register

The CSSELR_EL1 characteristics are:

Purpose

Selects the current Cache Size ID Register, CCSIDR_EL1, by specifying the required cache level and the cache type (either instruction or data cache).

Configurations

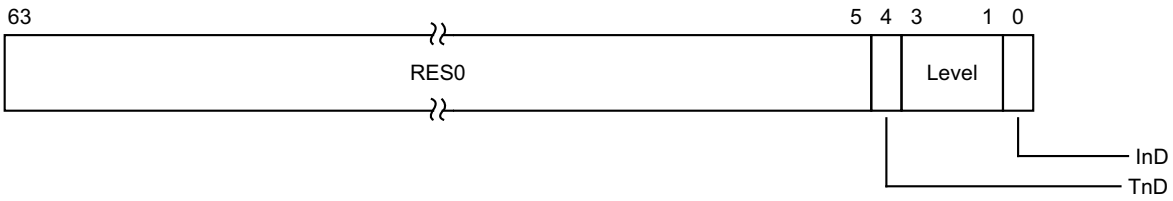
AArch64 System register CSSELR_EL1[31:0] is architecturally mapped to AArch32 System register CSSELR[31:0].

Attributes

CSSELR_EL1 is a 64-bit register.

Field descriptions

The CSSELR_EL1 bit assignments are:



Bits [63:5]

Reserved, RES0.

TnD, bit [4]

Allocation Tag not Data bit.

`0b0` Data, Instruction or Unified cache.

`0b1` Separate Allocation Tag cache.

When CSSELR_EL1.InD == 1, this bit is RES0.

If CSSELR_EL1.Level is programmed to a cache level that is not implemented, then the value for this field on a read of CSSELR_EL1 is UNKNOWN.

This field resets to an architecturally UNKNOWN value.

Level, bits [3:1]

Cache level of required cache.

`0b000` Level 1 cache.

`0b001` Level 2 cache.

`0b010` Level 3 cache.

`0b011` Level 4 cache.

`0b100` Level 5 cache.

`0b101` Level 6 cache.

`0b110` Level 7 cache.

All other values are reserved.

If CSSELR_EL1.Level is programmed to a cache level that is not implemented, then the value for this field on a read of CSSELR_EL1 is UNKNOWN.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

InD, bit [0]

Instruction not Data bit.

`0b0` Data or unified cache.

`0b1` Instruction cache.

If CSSELR_EL1.Level is programmed to a cache level that is not implemented, then a read of CSSELR_EL1 is CONSTRAINED UNPREDICTABLE, and returns UNKNOWN values for CSSELR_EL1.{Level, InD}.

This field resets to an architecturally UNKNOWN value.

Accessing the CSSELR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, CSSELR_EL1

op0	op1	CRn	CRm	op2
0b11	0b010	0b0000	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID2 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID4 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.CSSELR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return CSSELR_EL1;
elsif PSTATE.EL == EL2 then
    return CSSELR_EL1;
elsif PSTATE.EL == EL3 then
    return CSSELR_EL1;
```

MSR CSSELR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b010	0b0000	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID2 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID4 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.CSSELR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        CSSELR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    CSSELR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    CSSELR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.33 CTR_EL0, Cache Type Register

The CTR_EL0 characteristics are:

Purpose

Provides information about the architecture of the caches.

Configurations

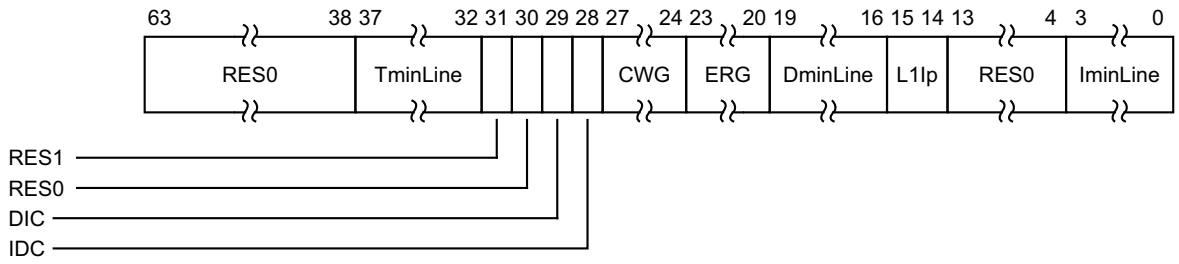
AArch64 System register CTR_EL0[31:0] is architecturally mapped to AArch32 System register CTR[31:0].

Attributes

CTR_EL0 is a 64-bit register.

Field descriptions

The CTR_EL0 bit assignments are:



Bits [63:38]

Reserved, RES0.

TminLine, bits [37:32]

Tag minimum Line. Log_2 of the number of words covered by Allocation Tags in the smallest cache line of all caches which can contain Allocation tags that are controlled by the PE.

Note

- For an implementation with cache lines containing 64 bytes of data and 4 Allocation Tags, this will be $log_2(64/4) = 4$.
- For an implementation with Allocations Tags in separate cache lines of 128 Allocation Tags per line, this will be $log_2(128*16/4) = 9$.

Bit [31]

Reserved, RES1.

Bit [30]

Reserved, RES0.

DIC, bit [29]

Instruction cache invalidation requirements for data to instruction coherence.

`0b0` Instruction cache invalidation to the Point of Unification is required for data to instruction coherence.

`0b1` Instruction cache invalidation to the Point of Unification is not required for data to instruction coherence.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

IDC, bit [28]

Data cache clean requirements for instruction to data coherence. The meaning of this bit is:

`0b0` Data cache clean to the Point of Unification is required for instruction to data coherence, unless CLIDR_EL1.LoC == `0b000` or (CLIDR_EL1.LoUIS == `0b000` && CLIDR_EL1.LoUU == `0b000`).

`0b1` Data cache clean to the Point of Unification is not required for instruction to data coherence.

CWG, bits [27:24]

Cache writeback granule. Log_2 of the number of words of the maximum size of memory that can be overwritten as a result of the eviction of a cache entry that has had a memory location in it modified.

A value of `0b0000` indicates that this register does not provide Cache writeback granule information and either:

- The architectural maximum of 512 words (2KB) must be assumed.
- The Cache writeback granule can be determined from maximum cache line size encoded in the Cache Size ID Registers.

Values greater than `0b1001` are reserved.

Arm recommends that an implementation that does not support cache write-back implements this field as `0b0001`. This applies, for example, to an implementation that supports only write-through caches.

ERG, bits [23:20]

Exclusives reservation granule. Log_2 of the number of words of the maximum size of the reservation granule that has been implemented for the Load-Exclusive and Store-Exclusive instructions.

The use of the value `0b0000` is deprecated.

The value `0b0001` and values greater than `0b1001` are reserved.

DminLine, bits [19:16]

Log_2 of the number of words in the smallest cache line of all the data caches and unified caches that are controlled by the PE.

L1Ip, bits [15:14]

Level 1 instruction cache policy. Indicates the indexing and tagging policy for the L1 instruction cache. Possible values of this field are:

`0b00` VMID aware Physical Index, Physical tag (VPIPT)

`0b01` ASID-tagged Virtual Index, Virtual Tag (AIVIVT)

`0b10` Virtual Index, Physical Tag (VIPT)

`0b11` Physical Index, Physical Tag (PIPT)

The value `0b01` is reserved in Armv8.

The value `0b00` is permitted only in an implementation that includes ARMv8.2-VPIPT, otherwise the value is reserved.

Bits [13:4]

Reserved, RES0.

IminLine, bits [3:0]

Log_2 of the number of words in the smallest cache line of all the instruction caches that are controlled by the PE.

Accessing the CTR_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, CTR_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b0000	0b0000	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.UCT == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TID2 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HFGRTR_EL2.CTR_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.UCT == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return CTR_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID2 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.CTR_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return CTR_EL0;
elsif PSTATE.EL == EL2 then
    return CTR_EL0;
elsif PSTATE.EL == EL3 then
    return CTR_EL0;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.34 DACR32_EL2, Domain Access Control Register

The DACR32_EL2 characteristics are:

Purpose

Allows access to the AArch32 DACR register from AArch64 state only. Its value has no effect on execution in AArch64 state.

Configurations

AArch64 System register DACR32_EL2[31:0] is architecturally mapped to AArch32 System register DACR[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DACR32_EL2 are UNDEFINED.

If EL2 is not implemented but EL3 is implemented, and EL1 is capable of using AArch32, then this register is not RES0.

Attributes

DACR32_EL2 is a 64-bit register.

Field descriptions

The DACR32_EL2 bit assignments are:

63 … 32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RES0	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

Bits [63:32]

Reserved, RES0.

D<n>, bits [2n+1:2n], for n = 0 to 15

Domain n access permission, where n = 0 to 15. Permitted values are:

`0b00` No access. Any access to the domain generates a Domain fault.

`0b01` Client. Accesses are checked against the permission bits in the translation tables.

`0b11` Manager. Accesses are not checked against the permission bits in the translation tables.

The value `0b10` is reserved.

This field resets to an architecturally UNKNOWN value.

Accessing the DACR32_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, DACR32_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0011	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return DACR32_EL2;
elsif PSTATE.EL == EL3 then
    return DACR32_EL2;
```

MSR DACR32_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0011	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    DACR32_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    DACR32_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.35 DCZID_EL0, Data Cache Zero ID register

The DCZID_EL0 characteristics are:

Purpose

Indicates the block size that is written with byte values of 0 by the DC ZVA (Data Cache Zero by Address) System instruction.

If ARMv8.5-MemTag is implemented, this register also indicates the granularity at which the DC GVA and DC GZVA instructions write.

Configurations

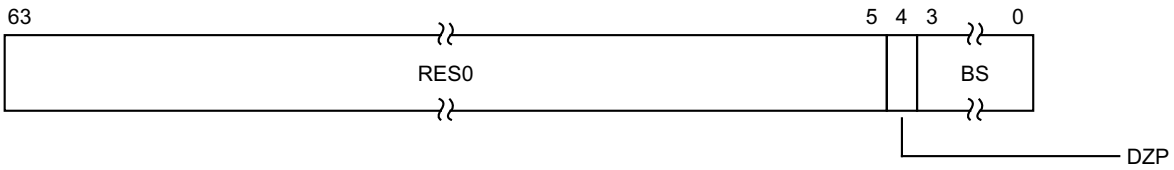
There are no configuration notes.

Attributes

DCZID_EL0 is a 64-bit register.

Field descriptions

The DCZID_EL0 bit assignments are:



Bits [63:5]

Reserved, RES0.

DZP, bit [4]

Data Zero Prohibited. This field indicates whether use of DC ZVA instructions is permitted or prohibited.

If ARMv8.5-MemTag is implemented, this field also indicates whether use of the DC GVA and DC GZVA instructions are permitted or prohibited.

`0b0` Instructions are permitted.

`0b1` Instructions are prohibited.

The value read from this field is governed by the access state and the values of the HCR_EL2.TDZ and SCTLR_EL1.DZE bits.

BS, bits [3:0]

Log_2 of the block size in words. The maximum size supported is 2KB (value == 9).

Accessing the DCZID_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, DCZID_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b0000	0b0000	0b111

```
if PSTATE.EL == EL0 then
    if EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HFGRTR_EL2.DCZID_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return DCZID_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.DCZID_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return DCZID_EL0;
elsif PSTATE.EL == EL2 then
    return DCZID_EL0;
elsif PSTATE.EL == EL3 then
    return DCZID_EL0;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.36 ESR_EL1, Exception Syndrome Register (EL1)

The ESR_EL1 characteristics are:

Purpose

Holds syndrome information for an exception taken to EL1.

Configurations

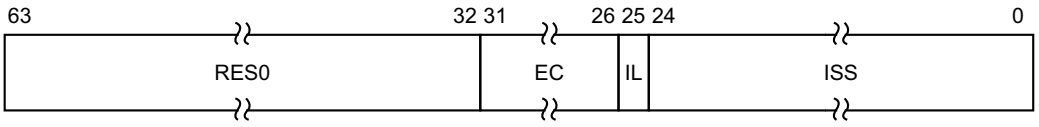
AArch64 System register ESR_EL1[31:0] is architecturally mapped to AArch32 System register DFSR[31:0].

Attributes

ESR_EL1 is a 64-bit register.

Field descriptions

The ESR_EL1 bit assignments are:



ESR_EL1 is made UNKNOWN as a result of an exception return from EL1.

When an UNPREDICTABLE instruction is treated as UNDEFINED, and the exception is taken to EL1, the value of ESR_EL1 is UNKNOWN. The value written to ESR_EL1 must be consistent with a value that could be created as a result of an exception from the same Exception level that generated the exception as a result of a situation that is not UNPREDICTABLE at that Exception level, in order to avoid the possibility of a privilege violation.

Bits [63:32]

Reserved, RES0.

EC, bits [31:26]

Exception Class. Indicates the reason for the exception that this register holds information about.

For each EC value, the table references a subsection that gives information about:

- The cause of the exception, for example the configuration required to enable the trap.
- The encoding of the associated ISS.

Possible values of the EC field are:

EC == 0b000000

Unknown reason.

See ISS encoding for exceptions with an unknown reason.

EC == 0b000001

Trapped WFI or WFE instruction execution.

Conditional WFE and WFI instructions that fail their condition code check do not cause an exception.

See ISS encoding for an exception from a WFI or WFE instruction.

EC == 0b000011 When AArch32 is supported at any Exception level

Trapped MCR or MRC access with (coproc==`0b1111`) that is not reported using EC `0b000000`.

See ISS encoding for an exception from an MCR or MRC access.

EC == 0b000100 When AArch32 is supported at any Exception level

Trapped MCRR or MRRC access with (coproc==`0b1111`) that is not reported using EC `0b000000`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

See ISS encoding for an exception from an MCRR or MRRC access.

EC == 0b000101 When AArch32 is supported at any Exception level

Trapped MCR or MRC access with (coproc==`0b1110`).

See ISS encoding for an exception from an MCR or MRC access.

EC == 0b000110 When AArch32 is supported at any Exception level

Trapped LDC or STC access.

The only architected uses of these instruction are:

- An STC to write data to memory from DBGDTRRXint.
- An LDC to read data from memory to DBGDTRTXint.

See ISS encoding for an exception from an LDC or STC instruction.

EC == 0b000111

Access to SVE, Advanced SIMD, or floating-point functionality trapped by CPACR_EL1.FPEN, CPTR_EL2.FPEN, CPTR_EL2.TFP, or CPTR_EL3.TFP control.

Excludes exceptions resulting from CPACR_EL1 when the value of HCR_EL2.TGE is 1, or because SVE or Advanced SIMD and floating-point are not implemented. These are reported with EC value `0b000000` as described in *The EC used to report an exception routed to EL2 because HCR_EL2.TGE is 1* on page D1-2347.

See ISS encoding for an exception from an access to SVE, Advanced SIMD or floating-point functionality, resulting from CPACR_EL1.FPEN, CPTR_EL2.FPEN or CPTR_ELx.TFP.

EC == 0b001100 When AArch32 is supported at any Exception level

Trapped MRRC access with (coproc==`0b1110`).

See ISS encoding for an exception from an MCRR or MRRC access.

EC == 0b001101 When ARMv8.5-BTI is implemented

Branch Target Exception.

See ISS encoding for an exception from Branch Target Identification instruction.

EC == 0b001110

Illegal Execution state.

See ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault.

EC == 0b010001 When AArch32 is supported at any Exception level

SVC instruction execution in AArch32 state.

This is reported in ESR_EL2 only when the exception is generated because the value of HCR_EL2.TGE is 1.

See ISS encoding for an exception from HVC or SVC instruction execution.

EC == 0b010101 When AArch64 is supported at any Exception level

SVC instruction execution in AArch64 state.

See ISS encoding for an exception from HVC or SVC instruction execution.

EC == 0b011000 When AArch64 is supported at any Exception level

Trapped MSR, MRS or System instruction execution in AArch64 state, that is not reported using EC `0b000000`, `0b000001` or `0b000111`.

This includes all instructions that cause exceptions that are part of the encoding space defined in *System instruction class encoding overview* on page C5-377, except for those exceptions reported using EC values `0b000000`, `0b000001`, or `0b000111`.

See ISS encoding for an exception from MSR, MRS, or System instruction execution in AArch64 state.

EC == 0b011001 When SVE is implemented

Access to SVE functionality trapped as a result of CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ, that is not reported using EC `0b000000`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

See ISS encoding for an exception from an access to SVE functionality, resulting from CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ.

EC == 0b011100 When ARMv8.3-FPAC is implemented

Exception from a Pointer Authentication instruction authentication failure

See ISS encoding for an exception from a Pointer Authentication instruction authentication failure.

EC == 0b100000

Instruction Abort from a lower Exception level.

Used for MMU faults generated by instruction accesses and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug related exceptions.

See ISS encoding for an exception from an Instruction Abort.

EC == 0b100001

Instruction Abort taken without a change in Exception level.

Used for MMU faults generated by instruction accesses and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug related exceptions.

See ISS encoding for an exception from an Instruction Abort.

EC == 0b100010

PC alignment fault exception.

See ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault.

EC == 0b100100

Data Abort from a lower Exception level.

Used for MMU faults generated by data accesses, alignment faults other than those caused by Stack Pointer misalignment, and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug related exceptions.

See ISS encoding for an exception from a Data Abort.

EC == 0b100101

Data Abort taken without a change in Exception level.

Used for MMU faults generated by data accesses, alignment faults other than those caused by Stack Pointer misalignment, and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug related exceptions.

See ISS encoding for an exception from a Data Abort.

EC == 0b100110

SP alignment fault exception.

See ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault.

EC == 0b101000 When AArch32 is supported at any Exception level

Trapped floating-point exception taken from AArch32 state.

This EC value is valid if the implementation supports trapping of floating-point exceptions, otherwise it is reserved. Whether a floating-point implementation supports trapping of floating-point exceptions is IMPLEMENTATION DEFINED.

See ISS encoding for an exception from a trapped floating-point exception.

EC == 0b101100 When AArch64 is supported at any Exception level

Trapped floating-point exception taken from AArch64 state.

This EC value is valid if the implementation supports trapping of floating-point exceptions, otherwise it is reserved. Whether a floating-point implementation supports trapping of floating-point exceptions is IMPLEMENTATION DEFINED.

See ISS encoding for an exception from a trapped floating-point exception.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

EC == 0b101111

SError interrupt.

See ISS encoding for an SError interrupt.

EC == 0b110000

Breakpoint exception from a lower Exception level.

See ISS encoding for an exception from a Breakpoint or Vector Catch debug exception.

EC == 0b110001

Breakpoint exception taken without a change in Exception level.

See ISS encoding for an exception from a Breakpoint or Vector Catch debug exception.

EC == 0b110010

Software Step exception from a lower Exception level.

See ISS encoding for an exception from a Software Step exception.

EC == 0b110011

Software Step exception taken without a change in Exception level.

See ISS encoding for an exception from a Software Step exception.

EC == 0b110100

Watchpoint exception from a lower Exception level.

See ISS encoding for an exception from a Watchpoint exception.

EC == 0b110101

Watchpoint exception taken without a change in Exception level.

See ISS encoding for an exception from a Watchpoint exception.

EC == 0b111000 When AArch32 is supported at any Exception level

BKPT instruction execution in AArch32 state.

See ISS encoding for an exception from execution of a Breakpoint instruction.

EC == 0b111100 When AArch64 is supported at any Exception level

BRK instruction execution in AArch64 state.

This is reported in ESR_EL3 only if a BRK instruction is executed.

See ISS encoding for an exception from execution of a Breakpoint instruction.

All other EC values are reserved by Arm, and:

- Unused values in the range `0b000000` - `0b101100` (`0x00` - `0x2C`) are reserved for future use for synchronous exceptions.
- Unused values in the range `0b101101` - `0b111111` (`0x2D` - `0x3F`) are reserved for future use, and might be used for synchronous or asynchronous exceptions.

The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-7853.

This field resets to an architecturally UNKNOWN value.

IL, bit [25]

Instruction Length for synchronous exceptions. Possible values of this bit are:

`0b0` 16-bit instruction trapped.

`0b1` 32-bit instruction trapped. This value is also used when the exception is one of the following:

- An SError interrupt.
- An Instruction Abort exception.
- A PC alignment fault exception.
- An SP alignment fault exception.
- A Data Abort exception for which the value of the ISV bit is 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- An Illegal Execution state exception.
- Any debug exception except for Breakpoint instruction exceptions. For Breakpoint instruction exceptions, this bit has its standard meaning:
  - `0b0`: 16-bit T32 BKPT instruction.
  - `0b1`: 32-bit A32 BKPT instruction or A64 BRK instruction.
- An exception reported using EC value `0b000000`.

This field resets to an architecturally UNKNOWN value.

ISS, bits [24:0]

Instruction Specific Syndrome. Architecturally, this field can be defined independently for each defined Exception class. However, in practice, some ISS encodings are used for more than one Exception class.

Typically, an ISS encoding has a number of subfields. When an ISS subfield holds a register number, the value returned in that field is the AArch64 view of the register number.

For an exception taken from AArch32 state, *Mapping of the general-purpose registers between the Execution states* on page D1-2407.

If the AArch32 register descriptor is `0b1111`, then:

- If the instruction that generated the exception was not UNPREDICTABLE, the field takes the value `0b11111`.
- If the instruction that generated the exception was UNPREDICTABLE, the field takes an UNKNOWN value that must be either:
  - The AArch64 view of the register number of a register that might have been used at the Exception level from which the exception was taken.
  - The value `0b11111`.

When the EC field is `0b000000`, indicating an exception with an unknown reason, the ISS field is not valid, RES0.

The following subsections describe each ISS format.

ISS encoding for exceptions with an unknown reason

24 … 0
RES0

Bits [24:0]

Reserved, RES0.

When an exception is reported using this EC code the IL field is set to 1.

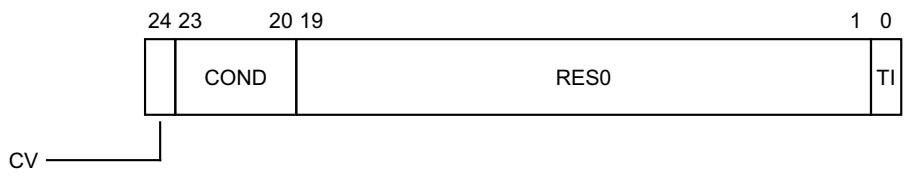
This EC code is used for all exceptions that are not covered by any other EC value. This includes exceptions that are generated in the following situations:

- The attempted execution of an instruction bit pattern that has no allocated instruction or that is not accessible at the current Exception level and Security state, including:
  - A read access using a System register pattern that is not allocated for reads or that does not permit reads at the current Exception level and Security state.
  - A write access using a System register pattern that is not allocated for writes or that does not permit writes at the current Exception level and Security state.
  - Instruction encodings that are unallocated.
  - Instruction encodings for instructions or System registers that are not implemented in the implementation.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- In Debug state, the attempted execution of an instruction bit pattern that is not accessible in Debug state.
- In Non-debug state, the attempted execution of an instruction bit pattern that is not accessible in Non-debug state.
- In AArch32 state, attempted execution of a short vector floating-point instruction.
- In an implementation that does not include Advanced SIMD and floating-point functionality, an attempted access to Advanced SIMD or floating-point functionality under conditions where that access would be permitted if that functionality was present. This includes the attempted execution of an Advanced SIMD or floating-point instruction, and attempted accesses to Advanced SIMD and floating-point System registers.
- An exception generated because of the value of one of the SCTLR_EL1.{ITD, SED, CP15BEN} control bits.
- Attempted execution of:
  - An HVC instruction when disabled by HCR_EL2.HCD or SCR_EL3.HCE.
  - An SMC instruction when disabled by SCR_EL3.SMD.
  - An HLT instruction when disabled by EDSCR.HDE.
- Attempted execution of an MSR or MRS instruction to access SP_EL0 when the value of SPSel.SP is 0.
- Attempted execution, in Debug state, of:
  - A DCPS1 instruction when the value of HCR_EL2.TGE is 1 and EL2 is disabled or not implemented in the current Security state.
  - A DCPS2 instruction from EL1 or EL0 when EL2 is disabled or not implemented in the current Security state.
  - A DCPS3 instruction when the value of EDSCR.SDD is 1, or when EL3 is not implemented.
- When EL3 is using AArch64, attempted execution from Secure EL1 of an SRS instruction using R13_mon. See *Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32* on page D1-2391.
- In Debug state when the value of EDSCR.SDD is 1, the attempted execution at EL2, EL1, or EL0 of an instruction that is configured to trap to EL3.
- In AArch32 state, the attempted execution of an MRS (banked register) or an MSR (banked register) instruction to SPSR_mon, SP_mon, or LR_mon.
- An exception that is taken to EL2 because the value of HCR_EL2.TGE is 1 that, if the value of HCR_EL2.TGE was 0 would have been reported with an ESR_ELx.EC value of `0b000111`.

ISS encoding for an exception from a WFI or WFE instruction



CV, bit [24]

Condition code valid. Possible values of this bit are:

`0b0` The COND field is not valid.

`0b1` The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to `0b1110`.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to `0b1110`.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to `0b1110`, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

This field resets to an architecturally UNKNOWN value.

Bits [19:1]

Reserved, RES0.

TI, bit [0]

Trapped instruction. Possible values of this bit are:

`0b0` WFI trapped.

`0b1` WFE trapped.

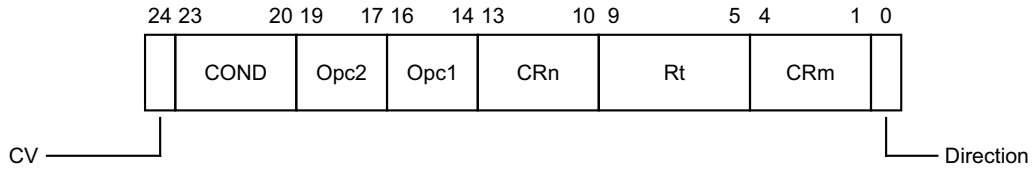
This field resets to an architecturally UNKNOWN value.

The following fields describe configuration settings for generating this exception:

- SCTLR_EL1.{nTWE, nTWI}.
- HCR_EL2.{TWE, TWI}.
- SCR_EL3.{TWE, TWI}.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ISS encoding for an exception from an MCR or MRC access



CV, bit [24]

Condition code valid. Possible values of this bit are:

`0b0` The COND field is not valid.

`0b1` The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to `0b1110`.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - — If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - — If the instruction is unconditional, COND is set to `0b1110`.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - — With COND set to `0b1110`, the value for unconditional.
  - — With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - — CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - — CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

This field resets to an architecturally UNKNOWN value.

Opc2, bits [19:17]

The Opc2 value from the issued instruction.

For a trapped VMRS access, holds the value `0b000`.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Opc1, bits [16:14]

The Opc1 value from the issued instruction.

For a trapped VMRS access, holds the value `0b111`.

This field resets to an architecturally UNKNOWN value.

CRn, bits [13:10]

The CRn value from the issued instruction.

For a trapped VMRS access, holds the reg field from the VMRS instruction encoding.

This field resets to an architecturally UNKNOWN value.

Rt, bits [9:5]

The Rt value from the issued instruction, the general-purpose register used for the transfer. The reported value gives the AArch64 view of the register. See *Mapping of the general-purpose registers between the Execution states* on page D1-2407.

This field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

For a trapped VMRS access, holds the value `0b0000`.

This field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction. The possible values of this bit are:

`0b0` Write to System register space. MCR instruction.

`0b1` Read from System register space. MRC or VMRS instruction.

This field resets to an architecturally UNKNOWN value.

The following fields describe configuration settings for generating exceptions that are reported using EC value `0b000011`:

- CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN}, for accesses to the Generic Timer Registers from EL0 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL1 or EL2.
- PMUSERENR_EL0.{ER, CR, SW, EN}, for accesses to Performance Monitor registers from EL0 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL1 or EL2.
- AMUSERENR_EL0.EN, for accesses to Activity Monitors registers from EL0 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL1 or EL2.
- HCR_EL2.{TRVM, TVM}, for accesses to virtual memory control registers from EL1 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL2.
- HCR_EL2.TTLB, for execution of TLB maintenance instructions at EL1 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL2.
- HCR_EL2.{TSW, TPC, TPU} for execution of cache maintenance instructions at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL2.
- HCR_EL2.TACR, for accesses to the Auxiliary Control Register at EL1 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL2.
- HCR_EL2.TIDCP, for accesses to lockdown, DMA, and TCM operations at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL2.
- HCR_EL2.{TID1, TID2, TID3}, for accesses to ID registers at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- CPTR_EL2.TCPAC, for accesses to CPACR_EL1 or CPACR using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL2.
- HSTR_EL2.T<n>, for accesses to System registers using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL2.
- CNTHCTL_EL2.EL1PCEN, for accesses to the Generic Timer registers from EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL2.
- MDCR_EL2.{TPM, TPMCR}, for accesses to Performance Monitor registers from EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL2.
- CPTR_EL2.TAM, for accesses to Activity Monitors registers from EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL2.
- CPTR_EL3.TCPAC, for accesses to CPACR from EL1 and EL2, and accesses to HCPTR from EL2 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL3.
- MDCR_EL3.TPM, for accesses to Performance Monitor registers from EL0, EL1 and EL2 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL3.
- CPTR_EL3.TAM, for accesses to Activity Monitors registers from EL0, EL1 and EL2 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL3.
- See *Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32* on page D1-2391 for information on other traps using EC value `0b000011`.
- If ARMv8.6-FGT is implemented, MCR or MRC access to some registers at EL0, trapped to EL2.

The following fields describe configuration settings for generating exceptions that are reported using EC value `0b000101`:

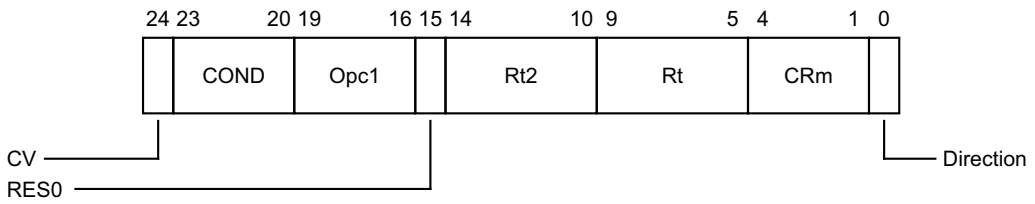
- CPACR_EL1.TTA for accesses to trace registers, MCR or MRC access (coproc == `0b1110`) trapped to EL1 or EL2.
- MDSCR_EL1.TDCC, for accesses to the Debug Communications Channel (DCC) registers at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == `0b1110`) trapped to EL1 or EL2.
- If ARMv8.6-FGT is implemented, MDCR_EL2.TDCC and MDCR_EL3.TDCC, for accesses to the DCC registers at EL0 and EL1, trapped to EL2.
- HCR_EL2.TID0, for accesses to the JIDR register in the ID group 0 at EL0 and EL1 using AArch32, MRC access (coproc == `0b1110`) trapped to EL2.
- CPTR_EL2.TTA, for accesses to trace registers using AArch32, MCR or MRC access (coproc == `0b1110`) trapped to EL2.
- MDCR_EL2.TDRA, for accesses to Debug ROM registers DBGDRAR and AArch-DBGDSAR using AArch32, MCR or MRC access (coproc == `0b1110`) trapped to EL2.
- MDCR_EL2.TDOSA, for accesses to powerdown debug registers, using AArch32 state, MCR or MRC access (coproc == `0b1110`) trapped to EL2.
- MDCR_EL2.TDA, for accesses to other debug registers, using AArch32 state, MCR or MRC access (coproc == `0b1110`) trapped to EL2.
- CPTR_EL3.TTA, for accesses to trace registers using AArch32, MCR or MRC access (coproc == `0b1110`) trapped to EL3.
- MDCR_EL3.TDOSA, for accesses to powerdown debug registers using AArch32, MCR or MRC access (coproc == `0b1110`) trapped to EL3.
- MDCR_EL3.TDA, for accesses to other debug registers, using AArch32, MCR or MRC access (coproc == `0b1110`) trapped to EL3.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following fields describe configuration settings for generating exceptions that are reported using EC value `0b001000`:

- HCR_EL2.TID0, for accesses to the FPSID register in ID group 0 at EL1 using AArch32 state, VMRS access trapped to EL2.
- HCR_EL2.TID3, for accesses to registers in ID group 3 including MVFR0, MVFR1 and MVFR2, VMRS access trapped to EL2.

ISS encoding for an exception from an MCRR or MRRC access



CV, bit [24]

Condition code valid. Possible values of this bit are:

`0b0` The COND field is not valid.

`0b1` The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to `0b1110`.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to `0b1110`.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to `0b1110`, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

Opc1, bits [19:16]

The Opc1 value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Bit [15]

Reserved, RES0.

Rt2, bits [14:10]

The Rt2 value from the issued instruction, the second general-purpose register used for the transfer. The reported value gives the AArch64 view of the register. See *Mapping of the general-purpose registers between the Execution states* on page D1-2407.

This field resets to an architecturally UNKNOWN value.

Rt, bits [9:5]

The Rt value from the issued instruction, the first general-purpose register used for the transfer. The reported value gives the AArch64 view of the register. See *Mapping of the general-purpose registers between the Execution states* on page D1-2407.

This field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction. The possible values of this bit are:

`0b0` Write to System register space. MCRR instruction.

`0b1` Read from System register space. MRRC instruction.

This field resets to an architecturally UNKNOWN value.

The following fields describe configuration settings for generating exceptions that are reported using EC value `0b000100`:

- CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN}, for accesses to the Generic Timer Registers from EL0 using AArch32 state, MCRR or MRRC access (coproc == `0b1111`) trapped to EL1 or EL2.
- PMUSERENR_EL0.{CR, EN}, for accesses to Performance Monitor registers from EL0 using AArch32 state, MCRR or MRRC access (coproc == `0b1111`) trapped to EL1 or EL2.
- AMUSERENR_EL0.{EN}, for accesses to Activity Monitors registers AMEVCNTR0<n> and AMEVCNTR1<n> from EL0 using AArch32 state, MCRR or MRRC access (coproc == `0b1111`) trapped to EL1 or EL2.
- HCR_EL2.{TRVM, TVM}, for accesses to virtual memory control registers from EL1 using AArch32 state, MCRR or MRRC access (coproc == `0b1111`) trapped to EL2.
- HSTR_EL2.T<n>, for accesses to System registers using AArch32 state, MCRR or MRRC access (coproc == `0b1111`) trapped to EL2.
- CNTHCTL_EL2.{EL1PCEN, EL1PCTEN}, for accesses to the Generic Timer registers from EL0 and EL1 using AArch32 state, MCRR or MRRC access (coproc == `0b1111`) trapped to EL2.
- MDCR_EL2.{TPM, TPMCR}, for accesses to Performance Monitor registers from EL0 and EL1 using AArch32 state, MCRR or MRRC access (coproc == `0b1111`) trapped to EL2.
- CPTR_EL2.TAM, for accesses to Activity Monitors registers AMEVCNTR0<n> and AMEVCNTR1<n> from EL0 and EL1 using AArch32 state, MCRR or MRRC access (coproc == `0b1111`) trapped to EL2.

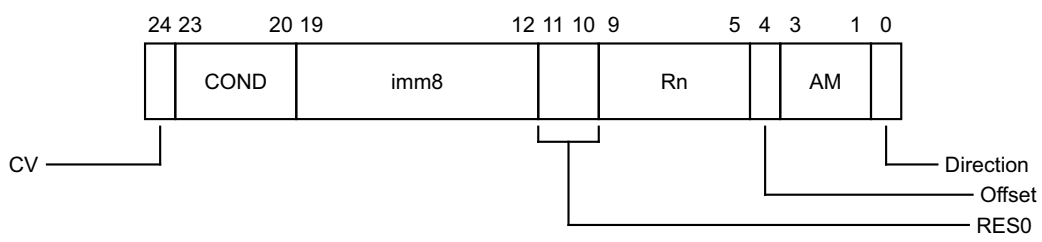
Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- MDCR_EL3.TPM, for accesses to Performance Monitor registers from EL0, EL1 and EL2 using AArch32 state, MCRR or MRRC access (coproc == `0b1111`) trapped to EL3.
- CPTR_EL3.TAM, for accesses to Activity Monitors registers from EL0, EL1 and EL2 using AArch32 state, MCRR or MRRC access (coproc == `0b1111`) trapped to EL3.
- If ARMv8.6-FGT is implemented, HDFGRTR_EL2.PMCCNTR_EL0 for MRRC access and HDFGWTR_EL2.PMCCNTR_EL0 for MCRR access to PMCCNTR at EL0, trapped to EL2.

The following sections describe configuration settings for generating exceptions that are reported using EC value `0b001100`:

- CPACR_EL1.TTA for accesses to trace registers using MCR or MRC instructions, MCRR or MRRC access (coproc == `0b1110`) trapped to EL1 or EL2.
- MDSCR_EL1.TDCC, for accesses to the Debug Communications Channel (DCC) registers DBGDSAR and DBGDRAR at EL0 using AArch32 state, MCRR or MRRC access (coproc == `0b1110`) trapped to EL1 or EL2.
- CPTR_EL2.TTA, for accesses to trace registers using AArch32, MCRR or MRRC access (coproc == `0b1110`) trapped to EL2.
- MDCR_EL2.TDRA, for accesses to Debug ROM registers DBGDRAR and AArch-DBGDSAR using AArch32, MCRR or MRRC access (coproc == `0b1110`) trapped to EL2.
- CPTR_EL3.TTA, for accesses to trace registers using AArch32, MCRR or MRRC access (coproc == `0b1110`) trapped to EL3.
- MDCR_EL3.TDOSA, for traps to powerdown debug registers using AArch32, MCRR or MRRC access (coproc == `0b1110`) trapped to EL3.
- MDCR_EL3.TDA, for accesses to other debug registers, using AArch32, MCRR or MRRC access (coproc == `0b1110`) trapped to EL3.

ISS encoding for an exception from an LDC or STC instruction



CV, bit [24]

Condition code valid. Possible values of this bit are:

`0b0` The COND field is not valid.

`0b1` The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to `0b1110`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to `0b1110`.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to `0b1110`, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

This field resets to an architecturally UNKNOWN value.

imm8, bits [19:12]

The immediate value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Bits [11:10]

Reserved, RES0.

Rn, bits [9:5]

The Rn value from the issued instruction, the general-purpose register used for the transfer. The reported value gives the AArch64 view of the register. See *Mapping of the general-purpose registers between the Execution states* on page D1-2407.

This field is valid only when AM[2] is 0, indicating an immediate form of the LDC or STC instruction. When AM[2] is 1, indicating a literal form of the LDC or STC instruction, this field is UNKNOWN.

This field resets to an architecturally UNKNOWN value.

Offset, bit [4]

Indicates whether the offset is added or subtracted:

`0b0` Subtract offset.

`0b1` Add offset.

This bit corresponds to the U bit in the instruction encoding.

This field resets to an architecturally UNKNOWN value.

AM, bits [3:1]

Addressing mode. The permitted values of this field are:

`0b000` Immediate unindexed.

`0b001` Immediate post-indexed.

`0b010` Immediate offset.

`0b011` Immediate pre-indexed.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b100` For a trapped STC instruction or a trapped T32 LDC instruction this encoding is reserved.

`0b110` For a trapped STC instruction, this encoding is reserved.

The values `0b101` and `0b111` are reserved. The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-7871.

Bit [2] in this subfield indicates the instruction form, immediate or literal.

Bits [1:0] in this subfield correspond to the bits {P, W} in the instruction encoding.

This field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction. The possible values of this bit are:

`0b0` Write to memory. STC instruction.

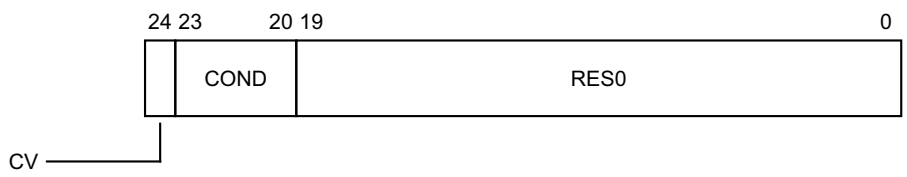
`0b1` Read from memory. LDC instruction.

This field resets to an architecturally UNKNOWN value.

The following fields describe the configuration settings for the traps that are reported using EC value `0b000110`:

- MDSCR_EL1.TDCC, for accesses using AArch32 state, LDC access to DBGDTRTXint or STC access to DBGDTRRXint trapped to EL1 or EL2.
- MDCR_EL2.TDA, for accesses using AArch32 state, LDC access to DBGDTRTXint or STC access to DBGDTRRXint MCR or MRC access trapped to EL2.
- MDCR_EL3.TDA, for accesses using AArch32 state, LDC access to DBGDTRTXint or STC access to DBGDTRRXint MCR or MRC access trapped to EL3.
- If ARMv8.6-FGT is implemented, MDCR_EL2.TDCC and MDCR_EL3.TDCC, for accesses to the DCC registers at EL0 and EL1, trapped to EL2.

ISS encoding for an exception from an access to SVE, Advanced SIMD or floating-point functionality, resulting from CPACR_EL1.FPEN, CPTR_EL2.FPEN or CPTR_ELx.TFP



The accesses covered by this trap include:

- Execution of SVE or Advanced SIMD and floating-point instructions.
- Accesses to the Advanced SIMD and floating-point System registers.

For an implementation that does not include either SVE or support for floating-point and Advanced SIMD, the exception is reported using the EC value `0b000000`.

CV, bit [24]

Condition code valid. Possible values of this bit are:

`0b0` The COND field is not valid.

`0b1` The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to `0b1110`.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to `0b1110`.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to `0b1110`, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

This field resets to an architecturally UNKNOWN value.

Bits [19:0]

Reserved, RES0.

The following sections describe the configuration settings for the traps that are reported using EC value `0b000111`:

- CPACR_EL1.FPEN, for accesses to SIMD and floating-point registers trapped to EL1.
- CPTR_EL2.TFP, for accesses to SIMD and floating-point registers trapped to EL2.
- CPTR_EL2.TFP, for accesses to SIMD and floating-point registers trapped to EL3.

ISS encoding for an exception from an access to SVE functionality, resulting from CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ

24 … 0
RES0

Bits [24:0]

When SVE is implemented:

Reserved, RES0.

The accesses covered by this trap include:

- Execution of SVE instructions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Accesses to the SVE system registers, ZCR_ELx and ID_AA64ZFR0_EL1.

For an implementation that does not include SVE, the exception is reported using the EC value `0b000000`.

ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault

24 … 0
RES0

Bits [24:0]

Reserved, RES0.

There are no configuration settings for generating Illegal Execution state exceptions and PC alignment fault exceptions. For more information about these exceptions see *The Illegal Execution state exception* on page D1-2353 and *PC alignment checking* on page D1-2333.

SP alignment checking on page D1-2333 describes the configuration settings for generating SP alignment fault exceptions.

ISS encoding for an exception from HVC or SVC instruction execution

24 … 16	15 … 0
RES0	imm16

Bits [24:16]

Reserved, RES0.

imm16, bits [15:0]

The value of the immediate field from the HVC or SVC instruction.

For an HVC instruction, and for an A64 SVC instruction, this is the value of the imm16 field of the issued instruction.

For an A32 or T32 SVC instruction:

- If the instruction is unconditional, then:
  - For the T32 instruction, this field is zero-extended from the imm8 field of the instruction.
  - For the A32 instruction, this field is the bottom 16 bits of the imm24 field of the instruction.
- If the instruction is conditional, this field is UNKNOWN.

This field resets to an architecturally UNKNOWN value.

In AArch32 state, the HVC instruction is unconditional, and a conditional SVC instruction generates an exception only if it passes its condition code check. Therefore, the syndrome information for these exceptions does not require conditionality information.

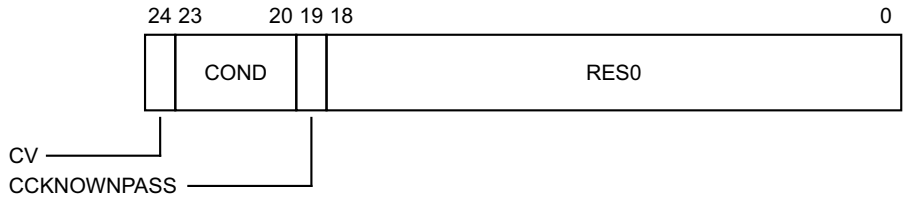
For T32 and A32 instructions, see SVC and HVC.

For A64 instructions, see SVC and HVC.

If ARMv8.6-FGT is implemented, HFGITR_EL2.{SVC_EL1, SVC_EL0} control fine-grained traps on SVC execution.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ISS encoding for an exception from SMC instruction execution in AArch32 state



For an SMC instruction that completes normally and generates an exception that is taken to EL3, the ISS encoding is RES0.

For an SMC instruction that is trapped to EL2 from EL1 because HCR_EL2.TSC is 1, the ISS encoding is as shown in the diagram.

CV, bit [24]

Condition code valid. Possible values of this bit are:

`0b0` The COND field is not valid.

`0b1` The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

This field is only valid if CCKNOWNPASS is 1, otherwise it is RES0.

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to `0b1110`.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - — If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - — If the instruction is unconditional, COND is set to `0b1110`.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - — With COND set to `0b1110`, the value for unconditional.
  - — With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - — CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - — CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

This field is only valid if CCKNOWNPASS is 1, otherwise it is RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

CCKNOWNPASS, bit [19]

Indicates whether the instruction might have failed its condition code check.

`0b0` The instruction was unconditional, or was conditional and passed its condition code check.

`0b1` The instruction was conditional, and might have failed its condition code check.

Note

In an implementation in which an SMC instruction that fails it code check is not trapped, this field can always return the value 0.

This field resets to an architecturally UNKNOWN value.

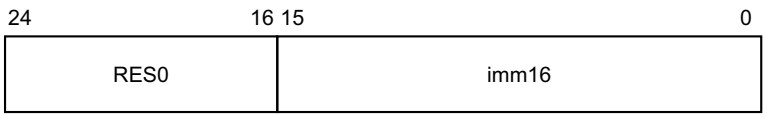
Bits [18:0]

Reserved, RES0.

HCR_EL2.TSC describes the configuration settings for trapping SMC instructions to EL2.

See *System calls* on page D1-2396 describes the case where these exceptions are trapped to EL3.

ISS encoding for an exception from SMC instruction execution in AArch64 state



Bits [24:16]

Reserved, RES0.

imm16, bits [15:0]

The value of the immediate field from the issued SMC instruction.

This field resets to an architecturally UNKNOWN value.

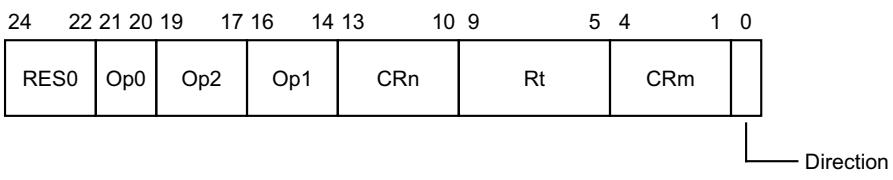
The value of ISS[24:0] described here is used both:

- When an SMC instruction is trapped from EL1 modes.
- When an SMC instruction is not trapped, so completes normally and generates an exception that is taken to EL3.

HCR_EL2.TSC describes the configuration settings for trapping SMC from EL1 modes.

System calls on page D1-2396 describes the case where these exceptions are trapped to EL3.

ISS encoding for an exception from MSR, MRS, or System instruction execution in AArch64 state



Bits [24:22]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Op0, bits [21:20]

The Op0 value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Op2, bits [19:17]

The Op2 value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Op1, bits [16:14]

The Op1 value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

CRn, bits [13:10]

The CRn value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Rt, bits [9:5]

The Rt value from the issued instruction, the general-purpose register used for the transfer.

This field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction. The possible values of this bit are:

`0b0` Write access, including MSR instructions.

`0b1` Read access, including MRS instructions.

This field resets to an architecturally UNKNOWN value.

For exceptions caused by System instructions, see System instructions for the encoding values returned by an instruction.

The following fields describe configuration settings for generating the exception that is reported using EC value `0b011000`:

- SCTLR_EL1.UCI, for execution of cache maintenance instructions using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- SCTLR_EL1.UCT, for accesses to CTR_EL0 using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- SCTLR_EL1.DZE, for execution of DC ZVA instructions using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- SCTLR_EL1.UMA, for accesses to the PSTATE interrupt masks using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- CPACR_EL1.TTA, for accesses to the trace registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- MDSCR_EL1.TDCC, for accesses to the Debug Communications Channel (DCC) registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- If ARMv8.6-FGT is implemented, MDCR_EL2.TDCC and MDCR_EL3.TDCC, for accesses to the DCC registers at EL0 and EL1, trapped to EL2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN} accesses to the Generic Timer registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- PMUSERENR_EL0.{ER, CR, SW, EN}, for accesses to the Performance Monitor registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- AMUSERENR_EL0.EN, for accesses to Activity Monitors registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- HCR_EL2.{TRVM, TVM}, for accesses to virtual memory control registers using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.TDZ, for execution of DC ZVA instructions using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.TTLB, for execution of TLB maintenance instructions using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.{TSW, TPC, TPU}, for execution of cache maintenance instructions using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.TACR, for accesses to the Auxiliary Control Register, ACTLR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.TIDCP, for accesses to lockdown, DMA, and TCM operations using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.{TID1, TID2, TID3}, for accesses to ID group 1, ID group 2 or ID group 3 registers, using AArch64 state, MSR or MRS access trapped to EL2.
- CPTR_EL2.TCPAC, for accesses to CPACR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.
- CPTR_EL2.TTA, for accesses to the trace registers, using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TTRF, for accesses to the trace filter register, TRFCR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TDRA, for accesses to Debug ROM registers, using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TDOSA, for accesses to powerdown debug registers using AArch64 state, MSR or MRS access trapped to EL2.
- CNTHCTL_EL2.{EL1PCEN, EL1PCTEN}, for accesses to the Generic Timer registers using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TDA, for accesses to debug registers using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.{TPM, TPMCR}, for accesses to Performance Monitor registers, using AArch64 state, MSR or MRS access trapped to EL2.
- CPTR_EL2.TAM, for accesses to Activity Monitors registers, using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.APK, for accesses to Pointer authentication key registers. using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.{NV, NV1}, for Nested virtualization register access, using AArch64 state, MSR or MRS access, trapped to EL2.
- HCR_EL2.AT, for execution of AT S1E* instructions, using AArch64 state, MSR or MRS access, trapped to EL2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- HCR_EL2.{TERR, FIEN}, for accesses to RAS registers, using AArch64 state, MSR or MRS access, trapped to EL2.
- SCR_EL3.APK, for accesses to Pointer authentication key registers, using AArch64 state, MSR or MRS access trapped to EL3.
- SCR_EL3.ST, for accesses to the Counter-timer Physical Secure timer registers, using AArch64 state, MSR or MRS access trapped to EL3.
- SCR_EL3.{TERR, FIEN}, for accesses to RAS registers, using AArch64 state, MSR or MRS access trapped to EL3.
- CPTR_EL3.TCPAC, for accesses to CPTR_EL2 and CPACR_EL1 using AArch64 state, MSR or MRS access trapped to EL3.
- CPTR_EL3.TTA, for accesses to the trace registers, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TTRF, for accesses to the filter trace control registers, TRFCR_EL1 and TRFCR_EL2, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TDA, for accesses to debug registers, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TDOSA, for accesses to powerdown debug registers, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TPM, for accesses to Performance Monitor registers, using AArch64 state, MSR or MRS access trapped to EL3.
- CPTR_EL3.TAM, for accesses to Activity Monitors registers, using AArch64 state, MSR or MRS access, trapped to EL3.
- If ARMv8.2-EVT is implemented the following registers control traps for EL1 and EL0 Cache controls that use this EC value:
  - HCR_EL2.{TTLBOS, TTLBIS, TICAB, TOCU, TID4}.
  - HCR2.{TTLBIS, TICAB, TOCU, TID4}.
- If ARMv8.6-FGT is implemented:
  - SCR_EL3.FGTEn, for accesses to the fine-grained trap registers, MSR or MRS access at EL2 trapped to EL3.
  - HFGRTR_EL2 for reads and HFGWTR_EL2 for writes of registers, using AArch64 state, MSR or MRS access at EL0 and EL1 trapped to EL2.
  - HFGITR_EL2 for execution of system instructions, MSR or MRS access trapped to EL2
  - HDFGRTR_EL2 for reads and HDFGWTR_EL2 for writes of registers, using AArch64 state, MSR or MRS access at EL0 and EL1 state trapped to EL2.
  - HAFGRTR_EL2 for reads of Activity Monitor counters, using AArch64 state, MRS access at EL0 and EL1 trapped to EL2.

ISS encoding for an IMPLEMENTATION DEFINED exception to EL3

24 … 0
IMPLEMENTATION DEFINED

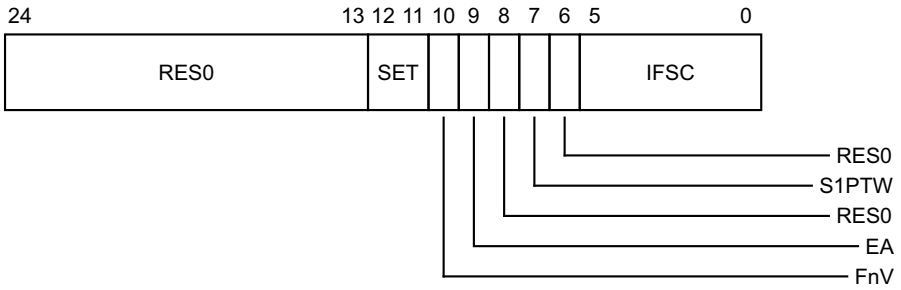
IMPLEMENTATION DEFINED, bits [24:0]

IMPLEMENTATION DEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

ISS encoding for an exception from an Instruction Abort



Bits [24:13]

Reserved, RES0.

SET, bits [12:11]

Synchronous Error Type. When the RAS Extension is implemented and IFSC is `0b010000`, describes the state of the PE after taking the Instruction Abort exception. The possible values of this field are:

`0b00` Recoverable error (UER).

`0b10` Uncontainable error (UC).

`0b11` Restartable error (UEO) or Corrected error (CE).

All other values are reserved.

Note

Software can use this information to determine what recovery might be possible. Taking a synchronous External Abort exception might result in an unrecoverable PE state.

This field is RES0 if either:

- The RAS Extension is not implemented.
- The value returned in the IFSC field is not `0b010000`.

This field resets to an architecturally UNKNOWN value.

FnV, bit [10]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

`0b0` FAR is valid.

`0b1` FAR is not valid, and holds an UNKNOWN value.

This field is only valid if the IFSC code is `0b010000`. It is RES0 for all other aborts.

This field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

This field resets to an architecturally UNKNOWN value.

Bit [8]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

S1PTW, bit [7]

For a stage 2 fault, indicates whether the fault was a stage 2 fault on an access made for a stage 1 translation table walk:

`0b0` Fault not on a stage 2 translation for a stage 1 translation table walk.

`0b1` Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

This field resets to an architecturally UNKNOWN value.

Bit [6]

Reserved, RES0.

IFSC, bits [5:0]

Instruction Fault Status Code.

`0b000000` Address size fault, level 0 of translation or translation table base register.

`0b000001` Address size fault, level 1.

`0b000010` Address size fault, level 2.

`0b000011` Address size fault, level 3.

`0b000100` Translation fault, level 0.

`0b000101` Translation fault, level 1.

`0b000110` Translation fault, level 2.

`0b000111` Translation fault, level 3.

`0b001001` Access flag fault, level 1.

`0b001010` Access flag fault, level 2.

`0b001011` Access flag fault, level 3.

`0b001101` Permission fault, level 1.

`0b001110` Permission fault, level 2.

`0b001111` Permission fault, level 3.

`0b010000` Synchronous External abort, not on translation table walk or hardware update of translation table.

`0b010100` Synchronous External abort, on translation table walk or hardware update of translation table, level 0.

`0b010101` Synchronous External abort, on translation table walk or hardware update of translation table, level 1.

`0b010110` Synchronous External abort, on translation table walk or hardware update of translation table, level 2.

`0b010111` Synchronous External abort, on translation table walk or hardware update of translation table, level 3.

`0b011000` **When RAS is not implemented** Synchronous parity or ECC error on memory access, not on translation table walk.

`0b011100` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 0.

`0b011101` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 1.

`0b011110` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 2.

`0b011111` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 3.

`0b110000` TLB conflict abort.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b110001` **When ARMv8.1-TTHM is implemented** Unsupported atomic hardware update fault.

All other values are reserved.

For more information about the lookup level associated with a fault, see *The level associated with MMU faults* on page D5-2658.

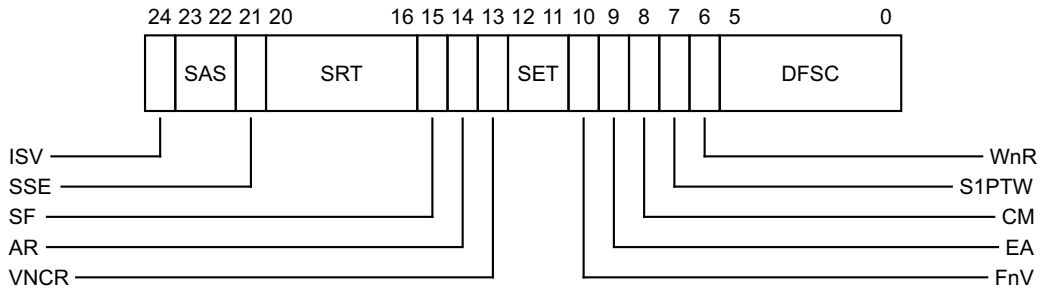
Note

Because Access flag faults and Permission faults can only result from a Block or Page translation table descriptor, they cannot occur at level 0.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.

This field resets to an architecturally UNKNOWN value.

ISS encoding for an exception from a Data Abort



ISV, bit [24]

Instruction Syndrome Valid. Indicates whether the syndrome information in ISS[23:14] is valid.

`0b0` No valid instruction syndrome. ISS[23:14] are RES0.

`0b1` ISS[23:14] hold a valid instruction syndrome.

This bit is 0 for all faults reported in ESR_EL2 except the following stage 2 aborts:

- AArch64 loads and stores of a single general-purpose register (including the register specified with `0b11111`, including those with Acquire/Release semantics, but excluding Load Exclusive or Store Exclusive and excluding those with writeback.
- AArch32 instructions where the instruction:
  - Is an LDR, LDA, LDRT, LDRSH, LDRSHT, LDRH, LDAH, LDRHT, LDRSB, LDRSBT, LDRB, LDAB, LDRBT, STR, STL, STRT, STRH, STLH, STRHT, STRB, STLB, or STRBT instruction.
  - Is not performing register writeback.
  - Is not using R15 as a source or destination register.

For these cases, ISV is UNKNOWN if the exception was generated in Debug state in memory access mode, and otherwise indicates whether ISS[23:14] hold a valid syndrome.

ISV is 0 for all faults reported in ESR_EL1 or ESR_EL3.

When the RAS Extension is implemented, ISV is 0 for any synchronous External abort.

For ISS reporting, a stage 2 abort on a stage 1 translation table walk does not return a valid instruction syndrome, and therefore ISV is 0 for these aborts.

When the RAS Extension is not implemented, the value of ISV on a synchronous External abort on a stage 2 translation table walk is IMPLEMENTATION DEFINED.

When ARMv8.5-MemTag is implemented, for a synchronous Tag Check Fault abort taken to ELx, ESR_ELx.FNV is 0 and FAR_ELx is valid.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

SAS, bits [23:22]

Syndrome Access Size. When ISV is 1, indicates the size of the access attempted by the faulting operation.

`0b00` Byte

`0b01` Halfword

`0b10` Word

`0b11` Doubleword

This field is UNKNOWN when the value of ISV is UNKNOWN.

This field is RES0 when the value of ISV is 0.

This field resets to an architecturally UNKNOWN value.

SSE, bit [21]

Syndrome Sign Extend. When ISV is 1, for a byte, halfword, or word load operation, indicates whether the data item must be sign extended. For these cases, the possible values of this bit are:

`0b0` Sign-extension not required.

`0b1` Data item must be sign-extended.

For all other operations this bit is 0.

This field is UNKNOWN when the value of ISV is UNKNOWN.

This field is RES0 when the value of ISV is 0.

This field resets to an architecturally UNKNOWN value.

SRT, bits [20:16]

Syndrome Register Transfer. When ISV is 1, the register number of the Rt operand of the faulting instruction.

If the exception was taken from an Exception level that is using AArch32 then this is the AArch64 view of the register. See *Mapping of the general-purpose registers between the Execution states* on page D1-2407.

This field is UNKNOWN when the value of ISV is UNKNOWN.

This field is RES0 when the value of ISV is 0.

This field resets to an architecturally UNKNOWN value.

SF, bit [15]

Width of the register accessed by the instruction is Sixty-Four. When ISV is 1, the possible values of this bit are:

`0b0` Instruction loads/stores a 32-bit wide register.

`0b1` Instruction loads/stores a 64-bit wide register.

Note

This field specifies the register width identified by the instruction, not the Execution state.

This field is UNKNOWN when the value of ISV is UNKNOWN.

This field is RES0 when the value of ISV is 0.

This field resets to an architecturally UNKNOWN value.

AR, bit [14]

Acquire/Release. When ISV is 1, the possible values of this bit are:

`0b0` Instruction did not have acquire/release semantics.

`0b1` Instruction did have acquire/release semantics.

This field is UNKNOWN when the value of ISV is UNKNOWN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field is RES0 when the value of ISV is 0.

This field resets to an architecturally UNKNOWN value.

VNCR, bit [13]

Indicates that the fault came from use of VNCR_EL2 register by EL1 code.

`0b0` The fault was not generated by the use of VNCR_EL2, by an MRS or MSR instruction executed at EL1.

`0b1` The fault was generated by the use of VNCR_EL2, by an MRS or MSR instruction executed at EL1.

This field is 0 in ESR_EL1.

This field resets to an architecturally UNKNOWN value.

SET, bits [12:11]

Synchronous Error Type. When the RAS Extension is implemented and DFSC is `0b010000`, describes the state of the PE after taking the Data Abort exception. The possible values of this field are:

`0b00` Recoverable error (UER).

`0b10` Uncontainable error (UC).

`0b11` Restartable error (UEO) or Corrected error (CE).

All other values are reserved.

Note

Software can use this information to determine what recovery might be possible. Taking a synchronous External Abort exception might result in an unrecoverable PE state.

This field is RES0 if either:

- The RAS Extension is not implemented.
- The value returned in the DFSC field is not `0b010000`.

This field resets to an architecturally UNKNOWN value.

FnV, bit [10]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

`0b0` FAR is valid.

`0b1` FAR is not valid, and holds an UNKNOWN value.

This field is valid only if the DFSC code is `0b010000`. It is RES0 for all other aborts.

This field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

This field resets to an architecturally UNKNOWN value.

CM, bit [8]

Cache maintenance. Indicates whether the Data Abort came from a cache maintenance or address translation instruction:

`0b0` The Data Abort was not generated by the execution of one of the System instructions identified in the description of value 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` The Data Abort was generated by either the execution of a cache maintenance instruction or by a synchronous fault on the execution of an address translation instruction. The DC ZVA instruction is not classified as a cache maintenance instruction, and therefore its execution cannot cause this field to be set to 1.

This field resets to an architecturally UNKNOWN value.

S1PTW, bit [7]

For a stage 2 fault, indicates whether the fault was a stage 2 fault on an access made for a stage 1 translation table walk:

`0b0` Fault not on a stage 2 translation for a stage 1 translation table walk.

`0b1` Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

This field resets to an architecturally UNKNOWN value.

WnR, bit [6]

Write not Read. Indicates whether a synchronous abort was caused by an instruction writing to a memory location, or by an instruction reading from a memory location. The possible values of this bit are:

`0b0` Abort caused by an instruction reading from a memory location.

`0b1` Abort caused by an instruction writing to a memory location.

For faults on cache maintenance and address translation instructions, this bit always returns a value of 1.

For faults from an atomic instruction that both reads and writes from a memory location, this bit is set to 0 if a read of the address specified by the instruction would have generated the fault which is being reported, otherwise it is set to 1. The architecture permits, but does not require, a relaxation of this requirement such that for all stage 2 aborts on stage 1 translation table walks for atomic instructions, the WnR bit is always 0.

This field is UNKNOWN for:

- An External abort on an Atomic access.
- A fault reported using a DFSC value of `0b110101` or `0b110001`, indicating an unsupported Exclusive or atomic access.

This field resets to an architecturally UNKNOWN value.

DFSC, bits [5:0]

Data Fault Status Code.

`0b000000` Address size fault, level 0 of translation or translation table base register.

`0b000001` Address size fault, level 1.

`0b000010` Address size fault, level 2.

`0b000011` Address size fault, level 3.

`0b000100` Translation fault, level 0.

`0b000101` Translation fault, level 1.

`0b000110` Translation fault, level 2.

`0b000111` Translation fault, level 3.

`0b001001` Access flag fault, level 1.

`0b001010` Access flag fault, level 2.

`0b001011` Access flag fault, level 3.

`0b001101` Permission fault, level 1.

`0b001110` Permission fault, level 2.

`0b001111` Permission fault, level 3.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b010000` Synchronous External abort, not on translation table walk or hardware update of translation table.

`0b010001` **When ARMv8.5-MemTag is implemented** Synchronous Tag Check Fault.

`0b010100` Synchronous External abort, on translation table walk or hardware update of translation table, level 0.

`0b010101` Synchronous External abort, on translation table walk or hardware update of translation table, level 1.

`0b010110` Synchronous External abort, on translation table walk or hardware update of translation table, level 2.

`0b010111` Synchronous External abort, on translation table walk or hardware update of translation table, level 3.

`0b011000` **When RAS is not implemented** Synchronous parity or ECC error on memory access, not on translation table walk.

`0b011100` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 0.

`0b011101` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 1.

`0b011110` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 2.

`0b011111` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 3.

`0b100001` Alignment fault.

`0b110000` TLB conflict abort.

`0b110001` **When ARMv8.1-TTHM is implemented** Unsupported atomic hardware update fault.

`0b110100` IMPLEMENTATION DEFINED fault (Lockdown).

`0b110101` IMPLEMENTATION DEFINED fault (Unsupported Exclusive or Atomic access).

All other values are reserved.

For more information about the lookup level associated with a fault, see *The level associated with MMU faults* on page D5-2658.

Note

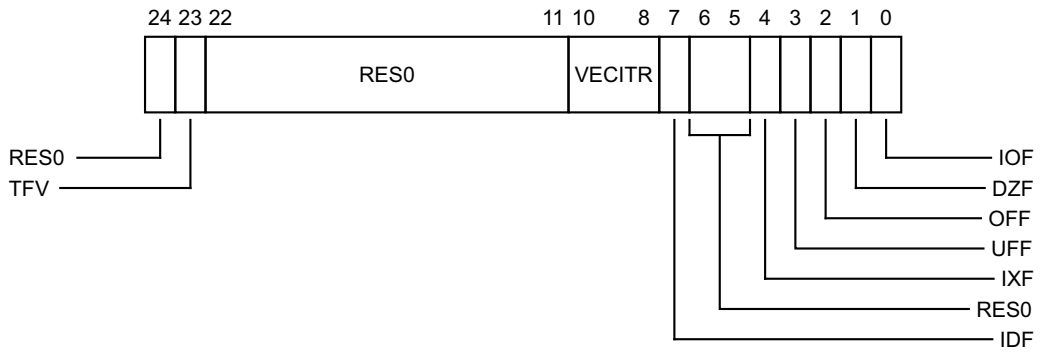
Because Access flag faults and Permission faults can only result from a Block or Page translation table descriptor, they cannot occur at level 0.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ISS encoding for an exception from a trapped floating-point exception



Bit [24]

Reserved, RES0.

TFV, bit [23]

Trapped Fault Valid bit. Indicates whether the IDF, IXF, UFF, OFF, DZF, and IOF bits hold valid information about trapped floating-point exceptions. The possible values of this bit are:

`0b0` The IDF, IXF, UFF, OFF, DZF, and IOF bits do not hold valid information about trapped floating-point exceptions and are UNKNOWN.

`0b1` One or more floating-point exceptions occurred during an operation performed while executing the reported instruction. The IDF, IXF, UFF, OFF, DZF, and IOF bits indicate trapped floating-point exceptions that occurred. For more information see *Floating-point exceptions and exception traps* on page D1-2360.

It is IMPLEMENTATION DEFINED whether this field is set to 0 on an exception generated by a trapped floating point exception from a vector instruction.

Note

This is not a requirement. Implementations can set this field to 1 on a trapped floating-point exception from a vector instruction and return valid information in the {IDF, IXF, UFF, OFF, DZF, IOF} fields.

This field resets to an architecturally UNKNOWN value.

Bits [22:11]

Reserved, RES0.

VECITR, bits [10:8]

For a trapped floating-point exception from an instruction executed in AArch32 state this field is RES1.

For a trapped floating-point exception from an instruction executed in AArch64 state this field is UNKNOWN.

This field resets to an architecturally UNKNOWN value.

IDF, bit [7]

Input Denormal floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

`0b0` Input denormal floating-point exception has not occurred.

`0b1` Input denormal floating-point exception occurred during execution of the reported instruction.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bits [6:5]

Reserved, RES0.

IXF, bit [4]

Inexact floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

`0b0` Inexact floating-point exception has not occurred.

`0b1` Inexact floating-point exception occurred during execution of the reported instruction.

This field resets to an architecturally UNKNOWN value.

UFF, bit [3]

Underflow floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

`0b0` Underflow floating-point exception has not occurred.

`0b1` Underflow floating-point exception occurred during execution of the reported instruction.

This field resets to an architecturally UNKNOWN value.

OFF, bit [2]

Overflow floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

`0b0` Overflow floating-point exception has not occurred.

`0b1` Overflow floating-point exception occurred during execution of the reported instruction.

This field resets to an architecturally UNKNOWN value.

DZF, bit [1]

Divide by Zero floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

`0b0` Divide by Zero floating-point exception has not occurred.

`0b1` Divide by Zero floating-point exception occurred during execution of the reported instruction.

This field resets to an architecturally UNKNOWN value.

IOF, bit [0]

Invalid Operation floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

`0b0` Invalid Operation floating-point exception has not occurred.

`0b1` Invalid Operation floating-point exception occurred during execution of the reported instruction.

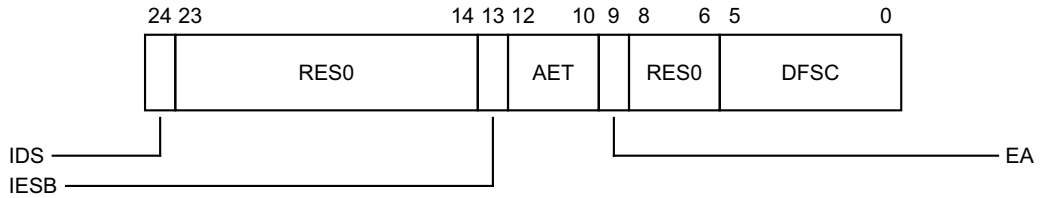
This field resets to an architecturally UNKNOWN value.

In an implementation that supports the trapping of floating-point exceptions:

- From an Exception level using AArch64, the FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} bits enable each of the floating-point exception traps.
- From an Exception level using AArch32, the FPSCR.{IDE, IXE, UFE, OFE, DZE, IOE} bits enable each of the floating-point exception traps.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ISS encoding for an SError interrupt



IDS, bit [24]

IMPLEMENTATION DEFINED syndrome. Possible values of this bit are:

`0b0` Bits[23:0] of the ISS field holds the fields described in this encoding.

Note

If the RAS Extension is not implemented, this means that bits[23:0] of the ISS field are RES0.

`0b1` Bits[23:0] of the ISS field holds IMPLEMENTATION DEFINED syndrome information that can be used to provide additional information about the SError interrupt.

Note

This field was previously called ISV.

This field resets to an architecturally UNKNOWN value.

Bits [23:14]

Reserved, RES0.

IESB, bit [13]

Implicit error synchronization event.

`0b0` The SError interrupt was either not synchronized by the implicit error synchronization event or not taken immediately.

`0b1` The SError interrupt was synchronized by the implicit error synchronization event and taken immediately.

This field is RES0 if the value returned in the DFSC field is not `0b010001`.

This field resets to an architecturally UNKNOWN value.

AET, bits [12:10]

Asynchronous Error Type.

When the RAS Extension is implemented and DFSC is `0b010001`, describes the state of the PE after taking the SError interrupt exception. The possible values of this field are:

`0b000` Uncontainable error (UC).

`0b001` Unrecoverable error (UEU).

`0b010` Restartable error (UEO).

`0b011` Recoverable error (UER).

`0b110` Corrected error (CE).

All other values are reserved.

If multiple errors are taken as a single SError interrupt exception, the overall state of the PE is reported. For example, if both a Recoverable and Unrecoverable error occurred, the state is Unrecoverable.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

Software can use this information to determine what recovery might be possible. The recovery software must also examine any implemented fault records to determine the location and extent of the error.

This field is RES0 if either:

- The RAS Extension is not implemented.
- The value returned in the DFSC field is not `0b010001`.

This field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort type. When the RAS Extension is implemented, this bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

This field is RES0 if either:

- The RAS Extension is not implemented.
- The value returned in the DFSC field is not `0b010001`.

This field resets to an architecturally UNKNOWN value.

Bits [8:6]

Reserved, RES0.

DFSC, bits [5:0]

When RAS is implemented:

Data Fault Status Code.

`0b000000` Uncategorized.

`0b010001` Asynchronous SError interrupt.

All other values are reserved.

This field resets to an architecturally UNKNOWN value.

ISS encoding for an exception from a Breakpoint or Vector Catch debug exception

24 … 6	5 … 0
RES0	IFSC

Bits [24:6]

Reserved, RES0.

IFSC, bits [5:0]

Instruction Fault Status Code.

`0b100010` Debug exception.

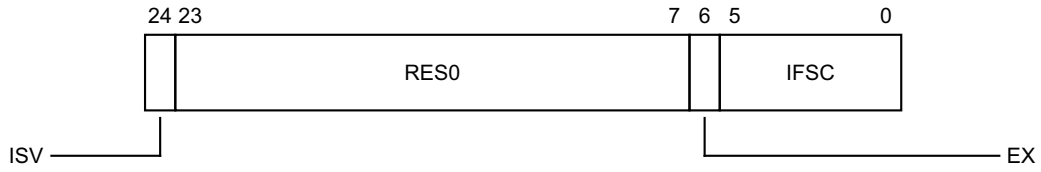
This field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions:

- For exceptions from AArch64, see *Breakpoint exceptions* on page D2-2439.
- For exceptions from AArch32, see *Breakpoint exceptions* on page G2-5830 and *Vector Catch exceptions* on page G2-5869.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ISS encoding for an exception from a Software Step exception



ISV, bit [24]

Instruction syndrome valid. Indicates whether the EX bit, ISS[6], is valid, as follows:

`0b0` EX bit is RES0.

`0b1` EX bit is valid.

See the EX bit description for more information.

This field resets to an architecturally UNKNOWN value.

Bits [23:7]

Reserved, RES0.

EX, bit [6]

Exclusive operation. If the ISV bit is set to 1, this bit indicates whether a Load-Exclusive instruction was stepped.

`0b0` An instruction other than a Load-Exclusive instruction was stepped.

`0b1` A Load-Exclusive instruction was stepped.

If the ISV bit is set to 0, this bit is RES0, indicating no syndrome data is available.

This field resets to an architecturally UNKNOWN value.

IFSC, bits [5:0]

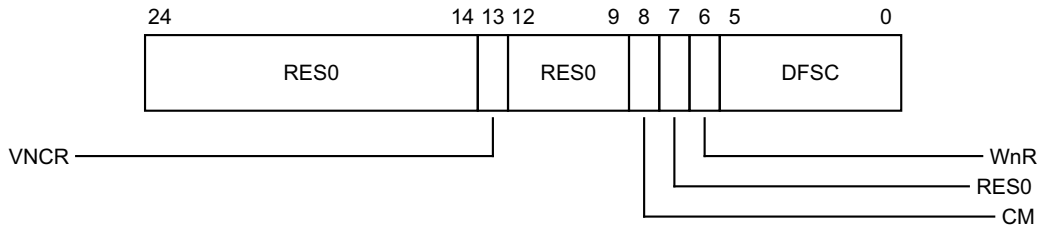
Instruction Fault Status Code.

`0b100010` Debug exception.

This field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions, see *Software Step exceptions* on page D2-2472.

ISS encoding for an exception from a Watchpoint exception



Bits [24:14]

Reserved, RES0.

VNCR, bit [13]

Indicates that the watchpoint came from use of VNCR_EL2 register by EL1 code.

`0b0` The watchpoint was not generated by the use of VNCR_EL2 by EL1 code.

`0b1` The watchpoint was generated by the use of VNCR_EL2 by EL1 code.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field is 0 in ESR_EL1.

This field resets to an architecturally UNKNOWN value.

Bits [12:9]

Reserved, RES0.

CM, bit [8]

Cache maintenance. Indicates whether the Watchpoint exception came from a cache maintenance or address translation instruction:

`0b0` The Watchpoint exception was not generated by the execution of one of the System instructions identified in the description of value 1.

`0b1` The Watchpoint exception was generated by either the execution of a cache maintenance instruction or by a synchronous Watchpoint exception on the execution of an address translation instruction. The DC ZVA instruction is not classified as a cache maintenance instruction, and therefore its execution cannot cause this field to be set to 1.

This field resets to an architecturally UNKNOWN value.

Bit [7]

Reserved, RES0.

WnR, bit [6]

Write not Read. Indicates whether the Watchpoint exception was caused by an instruction writing to a memory location, or by an instruction reading from a memory location. The possible values of this bit are:

`0b0` Watchpoint exception caused by an instruction reading from a memory location.

`0b1` Watchpoint exception caused by an instruction writing to a memory location.

For Watchpoint exceptions on cache maintenance and address translation instructions, this bit always returns a value of 1.

For Watchpoint exceptions from an atomic instruction, this field is set to 0 if a read of the location would have generated the Watchpoint exception, otherwise it is set to 1.

If multiple watchpoints match on the same access, it is UNPREDICTABLE which watchpoint generates the Watchpoint exception.

This field resets to an architecturally UNKNOWN value.

DFSC, bits [5:0]

Data Fault Status Code.

`0b100010` Debug exception.

This field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions, see *Watchpoint exceptions* on page D2-2457.

ISS encoding for an exception from execution of a Breakpoint instruction

24 … 16	15 … 0
RES0	Comment

Bits [24:16]

Reserved, RES0.

Comment, bits [15:0]

Set to the instruction comment field value, zero extended as necessary.

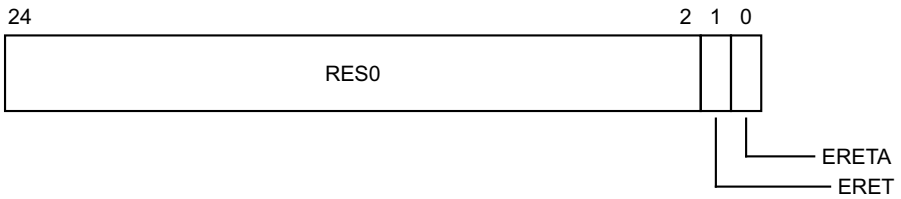
Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the AArch32 BKPT instructions, the comment field is described as the immediate field.

This field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions, see *Breakpoint Instruction exceptions* on page D2-2437.

ISS encoding for an exception from ERET, ERETAA or ERETAB instruction



This EC value applies when ARMv8.6-FGT is implemented, or when HCR_EL2.NV is 1.

Bits [24:2]

Reserved, RES0.

ERET, bit [1]

Indicates whether an ERET or ERETA* instruction was trapped to EL2. Possible values are:

`0b0` ERET instruction trapped to EL2.

`0b1` ERETAA or ERETAB instruction trapped to EL2.

If this bit is 0, the ERETA field is RES0.

This field resets to an architecturally UNKNOWN value.

ERETA, bit [0]

Indicates whether an ERETAA or ERETAB instruction was trapped to EL2. Possible values are:

`0b0` ERETAA instruction trapped to EL2.

`0b1` ERETAB instruction trapped to EL2.

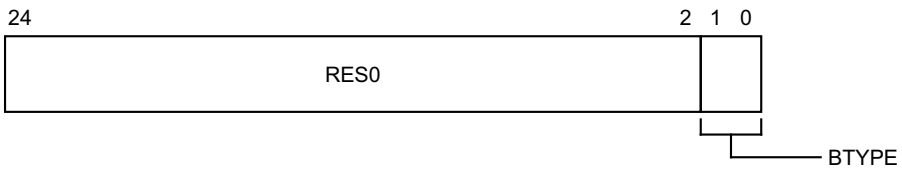
When the ERET field is 0, this bit is RES0.

This field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions, see HCR_EL2.NV.

If ARMv8.6-FGT is implemented, HFGITR_EL2.ERET controls fine-grained trap exceptions from ERET, ERETAA and ERETAB execution.

ISS encoding for an exception from Branch Target Identification instruction



Bits [24:2]

Reserved, RES0.

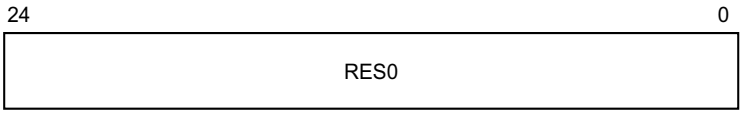
BTYPE, bits [1:0]

This field is set to the PSTATE.BTYPE value that generated the Branch Target Exception.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For more information about generating these exceptions, see Chapter B1 *The AArch64 Application Level Programmers' Model*.

ISS encoding for an exception from a Pointer Authentication instruction when HCR_EL2.API == 0 || SCR_EL3.API == 0



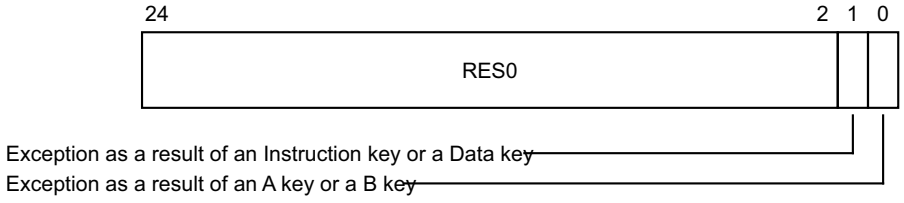
Bits [24:0]

Reserved, RES0.

For more information about generating these exceptions, see:

- HCR_EL2.API, for exceptions from Pointer authentication instructions, using AArch64 state, trapped to EL2.
- SCR_EL3.API, for exceptions from Pointer authentication instructions, using AArch64 state, trapped to EL3.

ISS encoding for an exception from a Pointer Authentication instruction authentication failure



Bits [24:2]

Reserved, RES0.

Bit [1]

This field indicates whether the exception is as a result of an Instruction key or a Data key.

`0b0` Instruction Key.

`0b1` Data Key.

This field resets to an architecturally UNKNOWN value.

Bit [0]

This field indicates whether the exception is as a result of an A key or a B key.

`0b0` A key.

`0b1` B key.

This field resets to an architecturally UNKNOWN value.

The following instructions generate an exception when the Pointer Authentication Code (PAC) is incorrect:

- `AUTIASP`, `AUTIAZ`, `AUTIA1716`.
- `AUTIBSP`, `AUTIBZ`, `AUTIB1716`.
- `AUTIA`, `AUTDA`, `AUTIB`, `AUTDB`.
- `AUTIZA`, `AUTIZB`, `AUTDZA`, `AUTDZB`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

It is IMPLEMENTATION DEFINED whether the following instructions generate an exception directly from the authorization failure, rather than changing the address in a way that will generate a translation fault when the address is accessed:

- RETAA, RETAB.
- BRAA, BRAB, BLRAA, BLRAB.
- BRAAZ, BRABZ, BLRAAZ, BLRABZ.
- ERETAA, ERETAB.
- LDRAA, LDRAB, whether the authenticated address is written back to the base register or not.

Accessing the ESR_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic ESR_EL1 or ESR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ESR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.ESR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x138];
    else
        return ESR_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return ESR_EL2;
    else
        return ESR_EL1;
elsif PSTATE.EL == EL3 then
    return ESR_EL1;
```

MSR ESR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.ESR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        NVMem[0x138] = X[t];
    else
        ESR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        ESR_EL2 = X[t];
    else
        ESR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    ESR_EL1 = X[t];
```

MRS <Xt>, ESR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0101	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        return NVMem[0x138];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return ESR_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        return ESR_EL1;
    else
        UNDEFINED;
```

MSR ESR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0101	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        NVMem[0x138] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        ESR_EL1 = X[t];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        ESR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        UNDEFINED;
```

MRS <Xt>, ESR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        return ESR_EL1;
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return ESR_EL2;
elsif PSTATE.EL == EL3 then
    return ESR_EL2;
```

MSR ESR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        ESR_EL1 = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    ESR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    ESR_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.37 ESR_EL2, Exception Syndrome Register (EL2)

The ESR_EL2 characteristics are:

Purpose

Holds syndrome information for an exception taken to EL2.

Configurations

AArch64 System register ESR_EL2[31:0] is architecturally mapped to AArch32 System register HSR[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

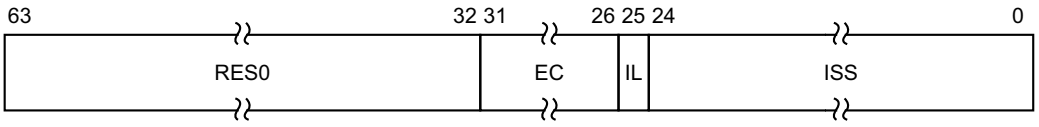
This register has no effect if EL2 is not enabled in the current Security state.

Attributes

ESR_EL2 is a 64-bit register.

Field descriptions

The ESR_EL2 bit assignments are:



ESR_EL2 is made UNKNOWN as a result of an exception return from EL2.

When an UNPREDICTABLE instruction is treated as UNDEFINED, and the exception is taken to EL2, the value of ESR_EL2 is UNKNOWN. The value written to ESR_EL2 must be consistent with a value that could be created as a result of an exception from the same Exception level that generated the exception as a result of a situation that is not UNPREDICTABLE at that Exception level, in order to avoid the possibility of a privilege violation.

Bits [63:32]

Reserved, RES0.

EC, bits [31:26]

Exception Class. Indicates the reason for the exception that this register holds information about.

For each EC value, the table references a subsection that gives information about:

- The cause of the exception, for example the configuration required to enable the trap.
- The encoding of the associated ISS.

Possible values of the EC field are:

EC == 0b000000

Unknown reason.

See ISS encoding for exceptions with an unknown reason.

EC == 0b000001

Trapped WFI or WFE instruction execution.

Conditional WFE and WFI instructions that fail their condition code check do not cause an exception.

See ISS encoding for an exception from a WFI or WFE instruction.

EC == 0b000011 When AArch32 is supported at any Exception level

Trapped MCR or MRC access with (coproc==`0b1111`) that is not reported using EC `0b000000`.

See ISS encoding for an exception from an MCR or MRC access.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

EC == 0b000100 When AArch32 is supported at any Exception level

Trapped MCRR or MRRC access with (coproc==`0b1111`) that is not reported using EC `0b000000`.

See ISS encoding for an exception from an MCRR or MRRC access.

EC == 0b000101 When AArch32 is supported at any Exception level

Trapped MCR or MRC access with (coproc==`0b1110`).

See ISS encoding for an exception from an MCR or MRC access.

EC == 0b000110 When AArch32 is supported at any Exception level

Trapped LDC or STC access.

The only architected uses of these instruction are:

- An STC to write data to memory from DBGDTRRXint.
- An LDC to read data from memory to DBGDTRTXint.

See ISS encoding for an exception from an LDC or STC instruction.

EC == 0b000111

Access to SVE, Advanced SIMD, or floating-point functionality trapped by CPACR_EL1.FPEN, CPTR_EL2.FPEN, CPTR_EL2.TFP, or CPTR_EL3.TFP control.

Excludes exceptions resulting from CPACR_EL1 when the value of HCR_EL2.TGE is 1, or because SVE or Advanced SIMD and floating-point are not implemented. These are reported with EC value `0b000000` as described in *The EC used to report an exception routed to EL2 because HCR_EL2.TGE is 1* on page D1-2347.

See ISS encoding for an exception from an access to SVE, Advanced SIMD or floating-point functionality, resulting from CPACR_EL1.FPEN, CPTR_EL2.FPEN or CPTR_ELx.TFP.

EC == 0b001000 When AArch32 is supported at any Exception level

Trapped VMRS access, from ID group trap, that is not reported using EC `0b000111`.

See ISS encoding for an exception from an MCR or MRC access.

EC == 0b001001 When ARMv8.3-PAuth is implemented

Trapped use of a Pointer authentication instruction because HCR_EL2.API == 0 || SCR_EL3.API == 0.

See *ISS encoding for an exception from a Pointer Authentication instruction when HCR_EL2.API == 0 || SCR_EL3.API == 0* on page D13-3045.

EC == 0b001100 When AArch32 is supported at any Exception level

Trapped MRRC access with (coproc==`0b1110`).

See ISS encoding for an exception from an MCRR or MRRC access.

EC == 0b001101 When ARMv8.5-BTI is implemented

Branch Target Exception.

See ISS encoding for an exception from Branch Target Identification instruction.

EC == 0b001110

Illegal Execution state.

See ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault.

EC == 0b010001 When AArch32 is supported at any Exception level

SVC instruction execution in AArch32 state.

This is reported in ESR_EL2 only when the exception is generated because the value of HCR_EL2.TGE is 1.

See ISS encoding for an exception from HVC or SVC instruction execution.

EC == 0b010010 When AArch32 is supported at any Exception level

HVC instruction execution in AArch32 state, when HVC is not disabled.

See ISS encoding for an exception from HVC or SVC instruction execution.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

EC == 0b010011 When AArch32 is supported at any Exception level

SMC instruction execution in AArch32 state, when SMC is not disabled.

This is reported in ESR_EL2 only when the exception is generated because the value of HCR_EL2.TSC is 1.

See ISS encoding for an exception from SMC instruction execution in AArch32 state.

EC == 0b010101 When AArch64 is supported at any Exception level

SVC instruction execution in AArch64 state.

See ISS encoding for an exception from HVC or SVC instruction execution.

EC == 0b010110 When AArch64 is supported at any Exception level

HVC instruction execution in AArch64 state, when HVC is not disabled.

See ISS encoding for an exception from HVC or SVC instruction execution.

EC == 0b010111 When AArch64 is supported at any Exception level

SMC instruction execution in AArch64 state, when SMC is not disabled.

This is reported in ESR_EL2 only when the exception is generated because the value of HCR_EL2.TSC is 1.

See ISS encoding for an exception from SMC instruction execution in AArch64 state.

EC == 0b011000 When AArch64 is supported at any Exception level

Trapped MSR, MRS or System instruction execution in AArch64 state, that is not reported using EC `0b000000`, `0b000001` or `0b000111`.

This includes all instructions that cause exceptions that are part of the encoding space defined in *System instruction class encoding overview* on page C5-377, except for those exceptions reported using EC values `0b000000`, `0b000001`, or `0b000111`.

See ISS encoding for an exception from MSR, MRS, or System instruction execution in AArch64 state.

EC == 0b011001 When SVE is implemented

Access to SVE functionality trapped as a result of CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ, that is not reported using EC `0b000000`.

See ISS encoding for an exception from an access to SVE functionality, resulting from CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ.

EC == 0b011010 When ARMv8.3-PAuth is implemented and ARMv8.3-NV is implemented

Trapped ERET, ERETAA, or ERETAB instruction execution.

See ISS encoding for an exception from ERET, ERETAA or ERETAB instruction.

EC == 0b011100 When ARMv8.3-FPAC is implemented

Exception from a Pointer Authentication instruction authentication failure

See ISS encoding for an exception from a Pointer Authentication instruction authentication failure.

EC == 0b100000

Instruction Abort from a lower Exception level.

Used for MMU faults generated by instruction accesses and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug related exceptions.

See ISS encoding for an exception from an Instruction Abort.

EC == 0b100001

Instruction Abort taken without a change in Exception level.

Used for MMU faults generated by instruction accesses and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug related exceptions.

See ISS encoding for an exception from an Instruction Abort.

EC == 0b100010

PC alignment fault exception.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

See ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault.

EC == 0b100100

Data Abort from a lower Exception level, excluding Data Aborts taken to EL2 as a result of accesses generated associated with VNCR_EL2 as part of nested virtualization support.

These Data Aborts might be generated from Exception levels in any Execution state.

Used for MMU faults generated by data accesses, alignment faults other than those caused by Stack Pointer misalignment, and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug related exceptions.

See ISS encoding for an exception from a Data Abort.

EC == 0b100101

Data Abort without a change in Exception level, or Data Aborts taken to EL2 as a result of accesses generated associated with VNCR_EL2 as part of nested virtualization support.

Used for MMU faults generated by data accesses, alignment faults other than those caused by Stack Pointer misalignment, and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug related exceptions.

See ISS encoding for an exception from a Data Abort.

EC == 0b100110

SP alignment fault exception.

See ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault.

EC == 0b101000 When AArch32 is supported at any Exception level

Trapped floating-point exception taken from AArch32 state.

This EC value is valid if the implementation supports trapping of floating-point exceptions, otherwise it is reserved. Whether a floating-point implementation supports trapping of floating-point exceptions is IMPLEMENTATION DEFINED.

See ISS encoding for an exception from a trapped floating-point exception.

EC == 0b101100 When AArch64 is supported at any Exception level

Trapped floating-point exception taken from AArch64 state.

This EC value is valid if the implementation supports trapping of floating-point exceptions, otherwise it is reserved. Whether a floating-point implementation supports trapping of floating-point exceptions is IMPLEMENTATION DEFINED.

See ISS encoding for an exception from a trapped floating-point exception.

EC == 0b101111

SError interrupt.

See ISS encoding for an SError interrupt.

EC == 0b110000

Breakpoint exception from a lower Exception level.

See ISS encoding for an exception from a Breakpoint or Vector Catch debug exception.

EC == 0b110001

Breakpoint exception taken without a change in Exception level.

See ISS encoding for an exception from a Breakpoint or Vector Catch debug exception.

EC == 0b110010

Software Step exception from a lower Exception level.

See ISS encoding for an exception from a Software Step exception.

EC == 0b110011

Software Step exception taken without a change in Exception level.

See ISS encoding for an exception from a Software Step exception.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

EC == 0b110100

Watchpoint from a lower Exception level, excluding Watchpoint Exceptions taken to EL2 as a result of accesses generated associated with VNCR_EL2 as part of nested virtualization support.

These Watchpoint Exceptions might be generated from Exception levels using any Execution state.

See ISS encoding for an exception from a Watchpoint exception.

EC == 0b110101

Watchpoint exceptions without a change in Exception level, or Watchpoint exceptions taken to EL2 as a result of accesses generated associated with VNCR_EL2 as part of nested virtualization support.

See ISS encoding for an exception from a Watchpoint exception.

EC == 0b111000 When AArch32 is supported at any Exception level

BKPT instruction execution in AArch32 state.

See ISS encoding for an exception from execution of a Breakpoint instruction.

EC == 0b111010 When AArch32 is supported at any Exception level

Vector Catch exception from AArch32 state.

The only case where a Vector Catch exception is taken to an Exception level that is using AArch64 is when the exception is routed to EL2 and EL2 is using AArch64.

See ISS encoding for an exception from a Breakpoint or Vector Catch debug exception.

EC == 0b111100 When AArch64 is supported at any Exception level

BRK instruction execution in AArch64 state.

This is reported in ESR_EL3 only if a BRK instruction is executed.

See ISS encoding for an exception from execution of a Breakpoint instruction.

All other EC values are reserved by Arm, and:

- Unused values in the range `0b000000` - `0b101100` (`0x00` - `0x2C`) are reserved for future use for synchronous exceptions.
- Unused values in the range `0b101101` - `0b111111` (`0x2D` - `0x3F`) are reserved for future use, and might be used for synchronous or asynchronous exceptions.

The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-7853.

This field resets to an architecturally UNKNOWN value.

IL, bit [25]

Instruction Length for synchronous exceptions. Possible values of this bit are:

`0b0` 16-bit instruction trapped.

`0b1` 32-bit instruction trapped. This value is also used when the exception is one of the following:

- An SError interrupt.
- An Instruction Abort exception.
- A PC alignment fault exception.
- An SP alignment fault exception.
- A Data Abort exception for which the value of the ISV bit is 0.
- An Illegal Execution state exception.
- Any debug exception except for Breakpoint instruction exceptions. For Breakpoint instruction exceptions, this bit has its standard meaning:
  - `0b0`: 16-bit T32 BKPT instruction.
  - `0b1`: 32-bit A32 BKPT instruction or A64 BRK instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- An exception reported using EC value `0b000000`.

This field resets to an architecturally UNKNOWN value.

ISS, bits [24:0]

Instruction Specific Syndrome. Architecturally, this field can be defined independently for each defined Exception class. However, in practice, some ISS encodings are used for more than one Exception class.

Typically, an ISS encoding has a number of subfields. When an ISS subfield holds a register number, the value returned in that field is the AArch64 view of the register number.

For an exception taken from AArch32 state, *Mapping of the general-purpose registers between the Execution states* on page D1-2407.

If the AArch32 register descriptor is `0b1111`, then:

- If the instruction that generated the exception was not UNPREDICTABLE, the field takes the value `0b11111`.
- If the instruction that generated the exception was UNPREDICTABLE, the field takes an UNKNOWN value that must be either:
  - The AArch64 view of the register number of a register that might have been used at the Exception level from which the exception was taken.
  - The value `0b11111`.

When the EC field is `0b000000`, indicating an exception with an unknown reason, the ISS field is not valid, RES0.

The following subsections describe each ISS format.

ISS encoding for exceptions with an unknown reason

24 … 0
RES0

Bits [24:0]

Reserved, RES0.

When an exception is reported using this EC code the IL field is set to 1.

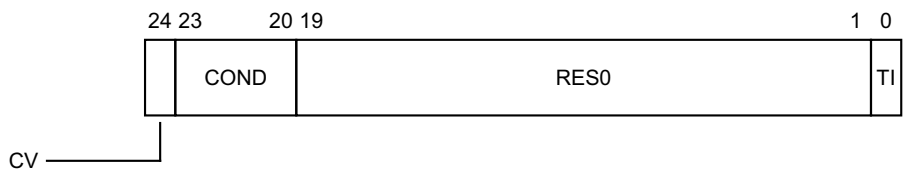
This EC code is used for all exceptions that are not covered by any other EC value. This includes exceptions that are generated in the following situations:

- The attempted execution of an instruction bit pattern that has no allocated instruction or that is not accessible at the current Exception level and Security state, including:
  - A read access using a System register pattern that is not allocated for reads or that does not permit reads at the current Exception level and Security state.
  - A write access using a System register pattern that is not allocated for writes or that does not permit writes at the current Exception level and Security state.
  - Instruction encodings that are unallocated.
  - Instruction encodings for instructions or System registers that are not implemented in the implementation.
- In Debug state, the attempted execution of an instruction bit pattern that is not accessible in Debug state.
- In Non-debug state, the attempted execution of an instruction bit pattern that is not accessible in Non-debug state.
- In AArch32 state, attempted execution of a short vector floating-point instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- In an implementation that does not include Advanced SIMD and floating-point functionality, an attempted access to Advanced SIMD or floating-point functionality under conditions where that access would be permitted if that functionality was present. This includes the attempted execution of an Advanced SIMD or floating-point instruction, and attempted accesses to Advanced SIMD and floating-point System registers.
- An exception generated because of the value of one of the SCTLR_EL1.{ITD, SED, CP15BEN} control bits.
- Attempted execution of:
  - An HVC instruction when disabled by HCR_EL2.HCD or SCR_EL3.HCE.
  - An SMC instruction when disabled by SCR_EL3.SMD.
  - An HLT instruction when disabled by EDSCR.HDE.
- Attempted execution of an MSR or MRS instruction to access SP_EL0 when the value of SPSel.SP is 0.
- Attempted execution, in Debug state, of:
  - A DCPS1 instruction when the value of HCR_EL2.TGE is 1 and EL2 is disabled or not implemented in the current Security state.
  - A DCPS2 instruction from EL1 or EL0 when EL2 is disabled or not implemented in the current Security state.
  - A DCPS3 instruction when the value of EDSCR.SDD is 1, or when EL3 is not implemented.
- When EL3 is using AArch64, attempted execution from Secure EL1 of an SRS instruction using R13_mon. See *Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32* on page D1-2391.
- In Debug state when the value of EDSCR.SDD is 1, the attempted execution at EL2, EL1, or EL0 of an instruction that is configured to trap to EL3.
- In AArch32 state, the attempted execution of an MRS (banked register) or an MSR (banked register) instruction to SPSR_mon, SP_mon, or LR_mon.
- An exception that is taken to EL2 because the value of HCR_EL2.TGE is 1 that, if the value of HCR_EL2.TGE was 0 would have been reported with an ESR_ELx.EC value of `0b000111`.

ISS encoding for an exception from a WFI or WFE instruction



CV, bit [24]

Condition code valid. Possible values of this bit are:

`0b0` The COND field is not valid.

`0b1` The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to `0b1110`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - — If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - — If the instruction is unconditional, COND is set to `0b1110`.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - — With COND set to `0b1110`, the value for unconditional.
  - — With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - — CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - — CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

This field resets to an architecturally UNKNOWN value.

Bits [19:1]

Reserved, RES0.

TI, bit [0]

Trapped instruction. Possible values of this bit are:

`0b0` WFI trapped.

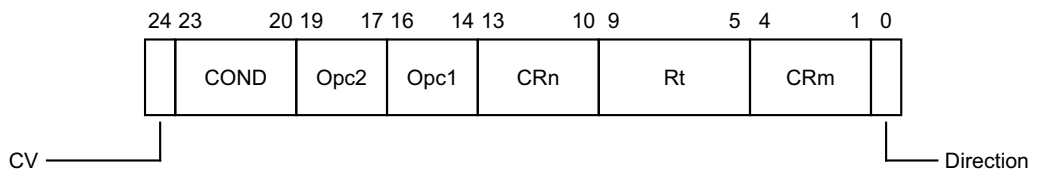
`0b1` WFE trapped.

This field resets to an architecturally UNKNOWN value.

The following fields describe configuration settings for generating this exception:

- SCTLR_EL1.{nTWE, nTWI}.
- HCR_EL2.{TWE, TWI}.
- SCR_EL3.{TWE, TWI}.

ISS encoding for an exception from an MCR or MRC access



CV, bit [24]

Condition code valid. Possible values of this bit are:

`0b0` The COND field is not valid.

`0b1` The COND field is valid.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to `0b1110`.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - — If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - — If the instruction is unconditional, COND is set to `0b1110`.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - — With COND set to `0b1110`, the value for unconditional.
  - — With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - — CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - — CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

This field resets to an architecturally UNKNOWN value.

Opc2, bits [19:17]

The Opc2 value from the issued instruction.

For a trapped VMRS access, holds the value `0b000`.

This field resets to an architecturally UNKNOWN value.

Opc1, bits [16:14]

The Opc1 value from the issued instruction.

For a trapped VMRS access, holds the value `0b111`.

This field resets to an architecturally UNKNOWN value.

CRn, bits [13:10]

The CRn value from the issued instruction.

For a trapped VMRS access, holds the reg field from the VMRS instruction encoding.

This field resets to an architecturally UNKNOWN value.

Rt, bits [9:5]

The Rt value from the issued instruction, the general-purpose register used for the transfer. The reported value gives the AArch64 view of the register. See *Mapping of the general-purpose registers between the Execution states* on page D1-2407.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

For a trapped VMRS access, holds the value `0b0000`.

This field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction. The possible values of this bit are:

`0b0` Write to System register space. MCR instruction.

`0b1` Read from System register space. MRC or VMRS instruction.

This field resets to an architecturally UNKNOWN value.

The following fields describe configuration settings for generating exceptions that are reported using EC value `0b000011`:

- CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN}, for accesses to the Generic Timer Registers from EL0 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL1 or EL2.
- PMUSERENR_EL0.{ER, CR, SW, EN}, for accesses to Performance Monitor registers from EL0 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL1 or EL2.
- AMUSERENR_EL0.EN, for accesses to Activity Monitors registers from EL0 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL1 or EL2.
- HCR_EL2.{TRVM, TVM}, for accesses to virtual memory control registers from EL1 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL2.
- HCR_EL2.TTLB, for execution of TLB maintenance instructions at EL1 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL2.
- HCR_EL2.{TSW, TPC, TPU} for execution of cache maintenance instructions at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL2.
- HCR_EL2.TACR, for accesses to the Auxiliary Control Register at EL1 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL2.
- HCR_EL2.TIDCP, for accesses to lockdown, DMA, and TCM operations at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL2.
- HCR_EL2.{TID1, TID2, TID3}, for accesses to ID registers at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL2.
- CPTR_EL2.TCPAC, for accesses to CPACR_EL1 or CPACR using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL2.
- HSTR_EL2.T<n>, for accesses to System registers using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL2.
- CNTHCTL_EL2.EL1PCEN, for accesses to the Generic Timer registers from EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL2.
- MDCR_EL2.{TPM, TPMCR}, for accesses to Performance Monitor registers from EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL2.
- CPTR_EL2.TAM, for accesses to Activity Monitors registers from EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL2.
- CPTR_EL3.TCPAC, for accesses to CPACR from EL1 and EL2, and accesses to HCPTR from EL2 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL3.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- MDCR_EL3.TPM, for accesses to Performance Monitor registers from EL0, EL1 and EL2 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL3.
- CPTR_EL3.TAM, for accesses to Activity Monitors registers from EL0, EL1 and EL2 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL3.
- See *Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32* on page D1-2391 for information on other traps using EC value `0b000011`.
- If ARMv8.6-FGT is implemented, MCR or MRC access to some registers at EL0, trapped to EL2.

The following fields describe configuration settings for generating exceptions that are reported using EC value `0b000101`:

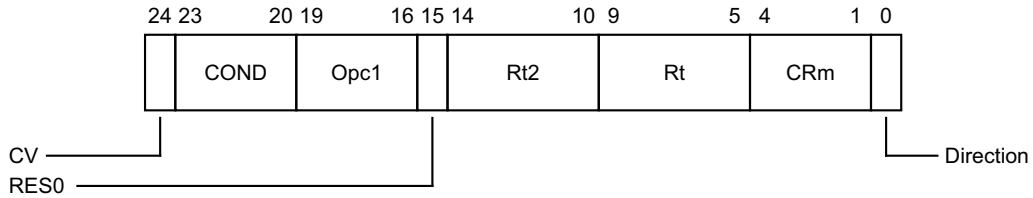
- CPACR_EL1.TTA for accesses to trace registers, MCR or MRC access (coproc == `0b1110`) trapped to EL1 or EL2.
- MDSCR_EL1.TDCC, for accesses to the Debug Communications Channel (DCC) registers at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == `0b1110`) trapped to EL1 or EL2.
- If ARMv8.6-FGT is implemented, MDCR_EL2.TDCC and MDCR_EL3.TDCC, for accesses to the DCC registers at EL0 and EL1, trapped to EL2.
- HCR_EL2.TID0, for accesses to the JIDR register in the ID group 0 at EL0 and EL1 using AArch32, MRC access (coproc == `0b1110`) trapped to EL2.
- CPTR_EL2.TTA, for accesses to trace registers using AArch32, MCR or MRC access (coproc == `0b1110`) trapped to EL2.
- MDCR_EL2.TDRA, for accesses to Debug ROM registers DBGDRAR and AArch-DBGDSAR using AArch32, MCR or MRC access (coproc == `0b1110`) trapped to EL2.
- MDCR_EL2.TDOSA, for accesses to powerdown debug registers, using AArch32 state, MCR or MRC access (coproc == `0b1110`) trapped to EL2.
- MDCR_EL2.TDA, for accesses to other debug registers, using AArch32 state, MCR or MRC access (coproc == `0b1110`) trapped to EL2.
- CPTR_EL3.TTA, for accesses to trace registers using AArch32, MCR or MRC access (coproc == `0b1110`) trapped to EL3.
- MDCR_EL3.TDOSA, for accesses to powerdown debug registers using AArch32, MCR or MRC access (coproc == `0b1110`) trapped to EL3.
- MDCR_EL3.TDA, for accesses to other debug registers, using AArch32, MCR or MRC access (coproc == `0b1110`) trapped to EL3.

The following fields describe configuration settings for generating exceptions that are reported using EC value `0b001000`:

- HCR_EL2.TID0, for accesses to the FPSID register in ID group 0 at EL1 using AArch32 state, VMRS access trapped to EL2.
- HCR_EL2.TID3, for accesses to registers in ID group 3 including MVFR0, MVFR1 and MVFR2, VMRS access trapped to EL2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ISS encoding for an exception from an MCRR or MRRC access



CV, bit [24]

Condition code valid. Possible values of this bit are:

`0b0` The COND field is not valid.

`0b1` The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to `0b1110`.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - — If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - — If the instruction is unconditional, COND is set to `0b1110`.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - — With COND set to `0b1110`, the value for unconditional.
  - — With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - — CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - — CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

This field resets to an architecturally UNKNOWN value.

Opc1, bits [19:16]

The Opc1 value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bit [15]

Reserved, RES0.

Rt2, bits [14:10]

The Rt2 value from the issued instruction, the second general-purpose register used for the transfer. The reported value gives the AArch64 view of the register. See *Mapping of the general-purpose registers between the Execution states* on page D1-2407.

This field resets to an architecturally UNKNOWN value.

Rt, bits [9:5]

The Rt value from the issued instruction, the first general-purpose register used for the transfer. The reported value gives the AArch64 view of the register. See *Mapping of the general-purpose registers between the Execution states* on page D1-2407.

This field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction. The possible values of this bit are:

`0b0` Write to System register space. MCRR instruction.

`0b1` Read from System register space. MRRC instruction.

This field resets to an architecturally UNKNOWN value.

The following fields describe configuration settings for generating exceptions that are reported using EC value `0b000100`:

- CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN}, for accesses to the Generic Timer Registers from EL0 using AArch32 state, MCRR or MRRC access (coproc == `0b1111`) trapped to EL1 or EL2.
- PMUSERENR_EL0.{CR, EN}, for accesses to Performance Monitor registers from EL0 using AArch32 state, MCRR or MRRC access (coproc == `0b1111`) trapped to EL1 or EL2.
- AMUSERENR_EL0.{EN}, for accesses to Activity Monitors registers AMEVCNTR0<n> and AMEVCNTR1<n> from EL0 using AArch32 state, MCRR or MRRC access (coproc == `0b1111`) trapped to EL1 or EL2.
- HCR_EL2.{TRVM, TVM}, for accesses to virtual memory control registers from EL1 using AArch32 state, MCRR or MRRC access (coproc == `0b1111`) trapped to EL2.
- HSTR_EL2.T<n>, for accesses to System registers using AArch32 state, MCRR or MRRC access (coproc == `0b1111`) trapped to EL2.
- CNTHCTL_EL2.{EL1PCEN, EL1PCTEN}, for accesses to the Generic Timer registers from EL0 and EL1 using AArch32 state, MCRR or MRRC access (coproc == `0b1111`) trapped to EL2.
- MDCR_EL2.{TPM, TPMCR}, for accesses to Performance Monitor registers from EL0 and EL1 using AArch32 state, MCRR or MRRC access (coproc == `0b1111`) trapped to EL2.
- CPTR_EL2.TAM, for accesses to Activity Monitors registers AMEVCNTR0<n> and AMEVCNTR1<n> from EL0 and EL1 using AArch32 state, MCRR or MRRC access (coproc == `0b1111`) trapped to EL2.
- MDCR_EL3.TPM, for accesses to Performance Monitor registers from EL0, EL1 and EL2 using AArch32 state, MCRR or MRRC access (coproc == `0b1111`) trapped to EL3.
- CPTR_EL3.TAM, for accesses to Activity Monitors registers from EL0, EL1 and EL2 using AArch32 state, MCRR or MRRC access (coproc == `0b1111`) trapped to EL3.

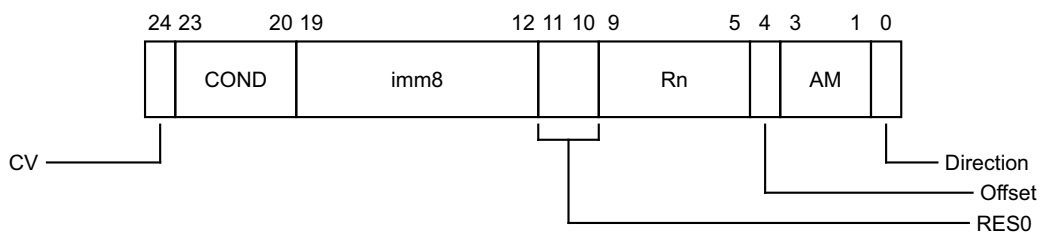
Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If ARMv8.6-FGT is implemented, HDFGRTR_EL2.PMCCNTR_EL0 for MRRC access and HDFGWTR_EL2.PMCCNTR_EL0 for MCRR access to PMCCNTR at EL0, trapped to EL2.

The following sections describe configuration settings for generating exceptions that are reported using EC value `0b001100`:

- CPACR_EL1.TTA for accesses to trace registers using MCR or MRC instructions, MCRR or MRRC access (coproc == `0b1110`) trapped to EL1 or EL2.
- MDSCR_EL1.TDCC, for accesses to the Debug Communications Channel (DCC) registers DBGDSAR and DBGDRAR at EL0 using AArch32 state, MCRR or MRRC access (coproc == `0b1110`) trapped to EL1 or EL2.
- CPTR_EL2.TTA, for accesses to trace registers using AArch32, MCRR or MRRC access (coproc == `0b1110`) trapped to EL2.
- MDCR_EL2.TDRA, for accesses to Debug ROM registers DBGDRAR and AArch-DBGDSAR using AArch32, MCRR or MRRC access (coproc == `0b1110`) trapped to EL2.
- CPTR_EL3.TTA, for accesses to trace registers using AArch32, MCRR or MRRC access (coproc == `0b1110`) trapped to EL3.
- MDCR_EL3.TDOSA, for traps to powerdown debug registers using AArch32, MCRR or MRRC access (coproc == `0b1110`) trapped to EL3.
- MDCR_EL3.TDA, for accesses to other debug registers, using AArch32, MCRR or MRRC access (coproc == `0b1110`) trapped to EL3.

ISS encoding for an exception from an LDC or STC instruction



CV, bit [24]

Condition code valid. Possible values of this bit are:

`0b0` The COND field is not valid.

`0b1` The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to `0b1110`.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to `0b1110`.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to `0b1110`, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

This field resets to an architecturally UNKNOWN value.

imm8, bits [19:12]

The immediate value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Bits [11:10]

Reserved, RES0.

Rn, bits [9:5]

The Rn value from the issued instruction, the general-purpose register used for the transfer. The reported value gives the AArch64 view of the register. See *Mapping of the general-purpose registers between the Execution states* on page D1-2407.

This field is valid only when AM[2] is 0, indicating an immediate form of the LDC or STC instruction. When AM[2] is 1, indicating a literal form of the LDC or STC instruction, this field is UNKNOWN.

This field resets to an architecturally UNKNOWN value.

Offset, bit [4]

Indicates whether the offset is added or subtracted:

`0b0` Subtract offset.

`0b1` Add offset.

This bit corresponds to the U bit in the instruction encoding.

This field resets to an architecturally UNKNOWN value.

AM, bits [3:1]

Addressing mode. The permitted values of this field are:

`0b000` Immediate unindexed.

`0b001` Immediate post-indexed.

`0b010` Immediate offset.

`0b011` Immediate pre-indexed.

`0b100` For a trapped STC instruction or a trapped T32 LDC instruction this encoding is reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b110` For a trapped STC instruction, this encoding is reserved.

The values `0b101` and `0b111` are reserved. The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-7871.

Bit [2] in this subfield indicates the instruction form, immediate or literal.

Bits [1:0] in this subfield correspond to the bits {P, W} in the instruction encoding.

This field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction. The possible values of this bit are:

`0b0` Write to memory. STC instruction.

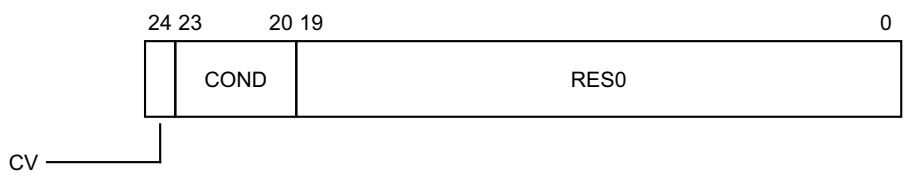
`0b1` Read from memory. LDC instruction.

This field resets to an architecturally UNKNOWN value.

The following fields describe the configuration settings for the traps that are reported using EC value `0b000110`:

- MDSCR_EL1.TDCC, for accesses using AArch32 state, LDC access to DBGDTRTXint or STC access to DBGDTRRXint trapped to EL1 or EL2.
- MDCR_EL2.TDA, for accesses using AArch32 state, LDC access to DBGDTRTXint or STC access to DBGDTRRXint MCR or MRC access trapped to EL2.
- MDCR_EL3.TDA, for accesses using AArch32 state, LDC access to DBGDTRTXint or STC access to DBGDTRRXint MCR or MRC access trapped to EL3.
- If ARMv8.6-FGT is implemented, MDCR_EL2.TDCC and MDCR_EL3.TDCC, for accesses to the DCC registers at EL0 and EL1, trapped to EL2.

ISS encoding for an exception from an access to SVE, Advanced SIMD or floating-point functionality, resulting from CPACR_EL1.FPEN, CPTR_EL2.FPEN or CPTR_ELx.TFP



The accesses covered by this trap include:

- Execution of SVE or Advanced SIMD and floating-point instructions.
- Accesses to the Advanced SIMD and floating-point System registers.

For an implementation that does not include either SVE or support for floating-point and Advanced SIMD, the exception is reported using the EC value `0b000000`.

CV, bit [24]

Condition code valid. Possible values of this bit are:

`0b0` The COND field is not valid.

`0b1` The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to `0b1110`.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to `0b1110`.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to `0b1110`, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

This field resets to an architecturally UNKNOWN value.

Bits [19:0]

Reserved, RES0.

The following sections describe the configuration settings for the traps that are reported using EC value `0b000111`:

- CPACR_EL1.FPEN, for accesses to SIMD and floating-point registers trapped to EL1.
- CPTR_EL2.TFP, for accesses to SIMD and floating-point registers trapped to EL2.
- CPTR_EL2.TFP, for accesses to SIMD and floating-point registers trapped to EL3.

ISS encoding for an exception from an access to SVE functionality, resulting from CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ

24 … 0
RES0

Bits [24:0]

When SVE is implemented:

Reserved, RES0.

The accesses covered by this trap include:

- Execution of SVE instructions.
- Accesses to the SVE system registers, ZCR_ELx and ID_AA64ZFR0_EL1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For an implementation that does not include SVE, the exception is reported using the EC value `0b000000`.

ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault

24 … 0
RES0

Bits [24:0]

Reserved, RES0.

There are no configuration settings for generating Illegal Execution state exceptions and PC alignment fault exceptions. For more information about these exceptions see *The Illegal Execution state exception* on page D1-2353 and *PC alignment checking* on page D1-2333.

SP alignment checking on page D1-2333 describes the configuration settings for generating SP alignment fault exceptions.

ISS encoding for an exception from HVC or SVC instruction execution

24 … 16	15 … 0
RES0	imm16

Bits [24:16]

Reserved, RES0.

imm16, bits [15:0]

The value of the immediate field from the HVC or SVC instruction.

For an HVC instruction, and for an A64 SVC instruction, this is the value of the imm16 field of the issued instruction.

For an A32 or T32 SVC instruction:

- If the instruction is unconditional, then:
  - For the T32 instruction, this field is zero-extended from the imm8 field of the instruction.
  - For the A32 instruction, this field is the bottom 16 bits of the imm24 field of the instruction.
- If the instruction is conditional, this field is UNKNOWN.

This field resets to an architecturally UNKNOWN value.

In AArch32 state, the HVC instruction is unconditional, and a conditional SVC instruction generates an exception only if it passes its condition code check. Therefore, the syndrome information for these exceptions does not require conditionality information.

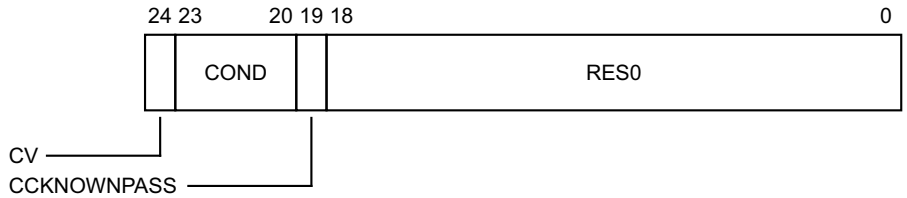
For T32 and A32 instructions, see SVC and HVC.

For A64 instructions, see SVC and HVC.

If ARMv8.6-FGT is implemented, HFGITR_EL2.{SVC_EL1, SVC_EL0} control fine-grained traps on SVC execution.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ISS encoding for an exception from SMC instruction execution in AArch32 state



For an SMC instruction that completes normally and generates an exception that is taken to EL3, the ISS encoding is RES0.

For an SMC instruction that is trapped to EL2 from EL1 because HCR_EL2.TSC is 1, the ISS encoding is as shown in the diagram.

CV, bit [24]

Condition code valid. Possible values of this bit are:

`0b0` The COND field is not valid.

`0b1` The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

This field is only valid if CCKNOWNPASS is 1, otherwise it is RES0.

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to `0b1110`.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - — If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - — If the instruction is unconditional, COND is set to `0b1110`.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - — With COND set to `0b1110`, the value for unconditional.
  - — With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - — CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - — CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

This field is only valid if CCKNOWNPASS is 1, otherwise it is RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

CCKNOWNPASS, bit [19]

Indicates whether the instruction might have failed its condition code check.

`0b0` The instruction was unconditional, or was conditional and passed its condition code check.

`0b1` The instruction was conditional, and might have failed its condition code check.

Note

In an implementation in which an SMC instruction that fails it code check is not trapped, this field can always return the value 0.

This field resets to an architecturally UNKNOWN value.

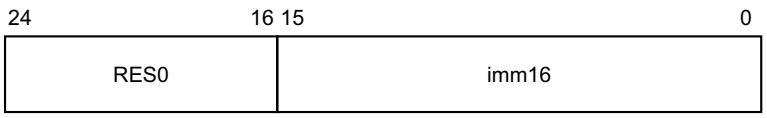
Bits [18:0]

Reserved, RES0.

HCR_EL2.TSC describes the configuration settings for trapping SMC instructions to EL2.

See *System calls* on page D1-2396 describes the case where these exceptions are trapped to EL3.

ISS encoding for an exception from SMC instruction execution in AArch64 state



Bits [24:16]

Reserved, RES0.

imm16, bits [15:0]

The value of the immediate field from the issued SMC instruction.

This field resets to an architecturally UNKNOWN value.

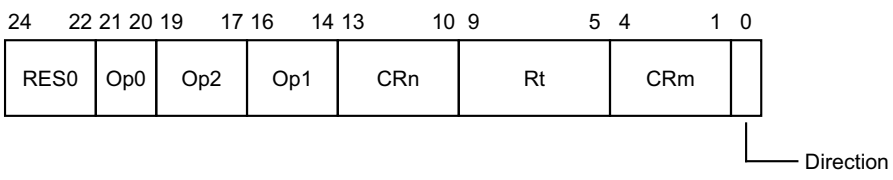
The value of ISS[24:0] described here is used both:

- When an SMC instruction is trapped from EL1 modes.
- When an SMC instruction is not trapped, so completes normally and generates an exception that is taken to EL3.

HCR_EL2.TSC describes the configuration settings for trapping SMC from EL1 modes.

System calls on page D1-2396 describes the case where these exceptions are trapped to EL3.

ISS encoding for an exception from MSR, MRS, or System instruction execution in AArch64 state



Bits [24:22]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Op0, bits [21:20]

The Op0 value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Op2, bits [19:17]

The Op2 value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Op1, bits [16:14]

The Op1 value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

CRn, bits [13:10]

The CRn value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Rt, bits [9:5]

The Rt value from the issued instruction, the general-purpose register used for the transfer.

This field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction. The possible values of this bit are:

`0b0` Write access, including MSR instructions.

`0b1` Read access, including MRS instructions.

This field resets to an architecturally UNKNOWN value.

For exceptions caused by System instructions, see System instructions for the encoding values returned by an instruction.

The following fields describe configuration settings for generating the exception that is reported using EC value `0b011000`:

- SCTLR_EL1.UCI, for execution of cache maintenance instructions using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- SCTLR_EL1.UCT, for accesses to CTR_EL0 using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- SCTLR_EL1.DZE, for execution of DC ZVA instructions using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- SCTLR_EL1.UMA, for accesses to the PSTATE interrupt masks using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- CPACR_EL1.TTA, for accesses to the trace registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- MDSCR_EL1.TDCC, for accesses to the Debug Communications Channel (DCC) registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- If ARMv8.6-FGT is implemented, MDCR_EL2.TDCC and MDCR_EL3.TDCC, for accesses to the DCC registers at EL0 and EL1, trapped to EL2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN} accesses to the Generic Timer registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- PMUSERENR_EL0.{ER, CR, SW, EN}, for accesses to the Performance Monitor registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- AMUSERENR_EL0.EN, for accesses to Activity Monitors registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- HCR_EL2.{TRVM, TVM}, for accesses to virtual memory control registers using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.TDZ, for execution of DC ZVA instructions using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.TTLB, for execution of TLB maintenance instructions using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.{TSW, TPC, TPU}, for execution of cache maintenance instructions using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.TACR, for accesses to the Auxiliary Control Register, ACTLR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.TIDCP, for accesses to lockdown, DMA, and TCM operations using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.{TID1, TID2, TID3}, for accesses to ID group 1, ID group 2 or ID group 3 registers, using AArch64 state, MSR or MRS access trapped to EL2.
- CPTR_EL2.TCPAC, for accesses to CPACR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.
- CPTR_EL2.TTA, for accesses to the trace registers, using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TTRF, for accesses to the trace filter register, TRFCR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TDRA, for accesses to Debug ROM registers, using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TDOSA, for accesses to powerdown debug registers using AArch64 state, MSR or MRS access trapped to EL2.
- CNTHCTL_EL2.{EL1PCEN, EL1PCTEN}, for accesses to the Generic Timer registers using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TDA, for accesses to debug registers using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.{TPM, TPMCR}, for accesses to Performance Monitor registers, using AArch64 state, MSR or MRS access trapped to EL2.
- CPTR_EL2.TAM, for accesses to Activity Monitors registers, using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.APK, for accesses to Pointer authentication key registers. using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.{NV, NV1}, for Nested virtualization register access, using AArch64 state, MSR or MRS access, trapped to EL2.
- HCR_EL2.AT, for execution of AT S1E* instructions, using AArch64 state, MSR or MRS access, trapped to EL2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- HCR_EL2.{TERR, FIEN}, for accesses to RAS registers, using AArch64 state, MSR or MRS access, trapped to EL2.
- SCR_EL3.APK, for accesses to Pointer authentication key registers, using AArch64 state, MSR or MRS access trapped to EL3.
- SCR_EL3.ST, for accesses to the Counter-timer Physical Secure timer registers, using AArch64 state, MSR or MRS access trapped to EL3.
- SCR_EL3.{TERR, FIEN}, for accesses to RAS registers, using AArch64 state, MSR or MRS access trapped to EL3.
- CPTR_EL3.TCPAC, for accesses to CPTR_EL2 and CPACR_EL1 using AArch64 state, MSR or MRS access trapped to EL3.
- CPTR_EL3.TTA, for accesses to the trace registers, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TTRF, for accesses to the filter trace control registers, TRFCR_EL1 and TRFCR_EL2, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TDA, for accesses to debug registers, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TDOSA, for accesses to powerdown debug registers, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TPM, for accesses to Performance Monitor registers, using AArch64 state, MSR or MRS access trapped to EL3.
- CPTR_EL3.TAM, for accesses to Activity Monitors registers, using AArch64 state, MSR or MRS access, trapped to EL3.
- If ARMv8.2-EVT is implemented the following registers control traps for EL1 and EL0 Cache controls that use this EC value:
  - HCR_EL2.{TTLBOS, TTLBIS, TICAB, TOCU, TID4}.
  - HCR2.{TTLBIS, TICAB, TOCU, TID4}.
- If ARMv8.6-FGT is implemented:
  - SCR_EL3.FGTEn, for accesses to the fine-grained trap registers, MSR or MRS access at EL2 trapped to EL3.
  - HFGRTR_EL2 for reads and HFGWTR_EL2 for writes of registers, using AArch64 state, MSR or MRS access at EL0 and EL1 trapped to EL2.
  - HFGITR_EL2 for execution of system instructions, MSR or MRS access trapped to EL2
  - HDFGRTR_EL2 for reads and HDFGWTR_EL2 for writes of registers, using AArch64 state, MSR or MRS access at EL0 and EL1 state trapped to EL2.
  - HAFGRTR_EL2 for reads of Activity Monitor counters, using AArch64 state, MRS access at EL0 and EL1 trapped to EL2.

ISS encoding for an IMPLEMENTATION DEFINED exception to EL3

24 … 0
IMPLEMENTATION DEFINED

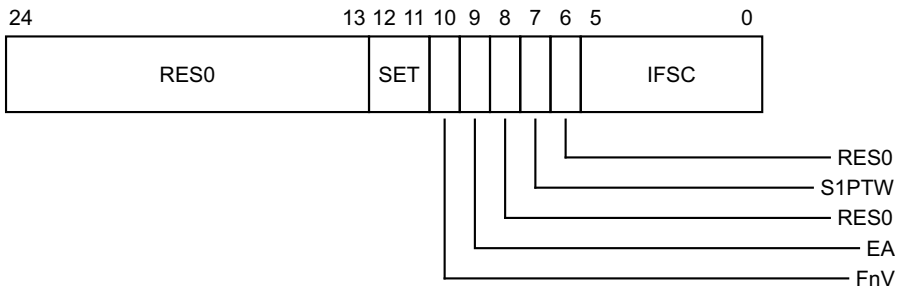
IMPLEMENTATION DEFINED, bits [24:0]

IMPLEMENTATION DEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

ISS encoding for an exception from an Instruction Abort



Bits [24:13]

Reserved, RES0.

SET, bits [12:11]

Synchronous Error Type. When the RAS Extension is implemented and IFSC is `0b010000`, describes the state of the PE after taking the Instruction Abort exception. The possible values of this field are:

`0b00` Recoverable error (UER).

`0b10` Uncontainable error (UC).

`0b11` Restartable error (UEO) or Corrected error (CE).

All other values are reserved.

Note

Software can use this information to determine what recovery might be possible. Taking a synchronous External Abort exception might result in an unrecoverable PE state.

This field is RES0 if either:

- The RAS Extension is not implemented.
- The value returned in the IFSC field is not `0b010000`.

This field resets to an architecturally UNKNOWN value.

FnV, bit [10]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

`0b0` FAR is valid.

`0b1` FAR is not valid, and holds an UNKNOWN value.

This field is only valid if the IFSC code is `0b010000`. It is RES0 for all other aborts.

This field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

This field resets to an architecturally UNKNOWN value.

Bit [8]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

S1PTW, bit [7]

For a stage 2 fault, indicates whether the fault was a stage 2 fault on an access made for a stage 1 translation table walk:

`0b0` Fault not on a stage 2 translation for a stage 1 translation table walk.

`0b1` Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

This field resets to an architecturally UNKNOWN value.

Bit [6]

Reserved, RES0.

IFSC, bits [5:0]

Instruction Fault Status Code.

`0b000000` Address size fault, level 0 of translation or translation table base register.

`0b000001` Address size fault, level 1.

`0b000010` Address size fault, level 2.

`0b000011` Address size fault, level 3.

`0b000100` Translation fault, level 0.

`0b000101` Translation fault, level 1.

`0b000110` Translation fault, level 2.

`0b000111` Translation fault, level 3.

`0b001001` Access flag fault, level 1.

`0b001010` Access flag fault, level 2.

`0b001011` Access flag fault, level 3.

`0b001101` Permission fault, level 1.

`0b001110` Permission fault, level 2.

`0b001111` Permission fault, level 3.

`0b010000` Synchronous External abort, not on translation table walk or hardware update of translation table.

`0b010100` Synchronous External abort, on translation table walk or hardware update of translation table, level 0.

`0b010101` Synchronous External abort, on translation table walk or hardware update of translation table, level 1.

`0b010110` Synchronous External abort, on translation table walk or hardware update of translation table, level 2.

`0b010111` Synchronous External abort, on translation table walk or hardware update of translation table, level 3.

`0b011000` **When RAS is not implemented** Synchronous parity or ECC error on memory access, not on translation table walk.

`0b011100` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 0.

`0b011101` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 1.

`0b011110` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 2.

`0b011111` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 3.

`0b110000` TLB conflict abort.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b110001` **When ARMv8.1-TTHM is implemented** Unsupported atomic hardware update fault.

All other values are reserved.

For more information about the lookup level associated with a fault, see *The level associated with MMU faults* on page D5-2658.

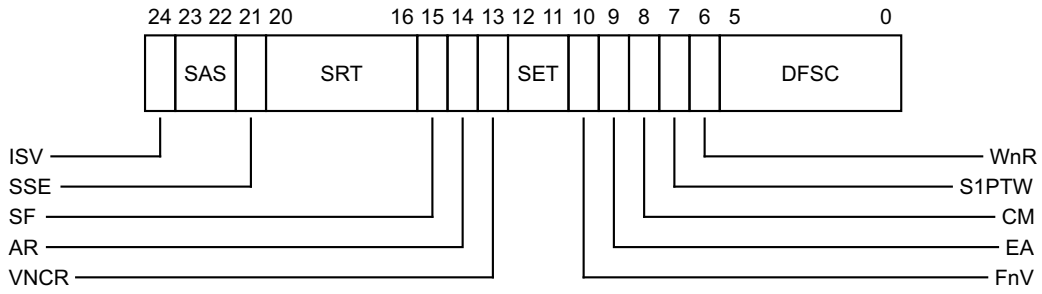
Note

Because Access flag faults and Permission faults can only result from a Block or Page translation table descriptor, they cannot occur at level 0.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.

This field resets to an architecturally UNKNOWN value.

ISS encoding for an exception from a Data Abort



ISV, bit [24]

Instruction Syndrome Valid. Indicates whether the syndrome information in ISS[23:14] is valid.

`0b0` No valid instruction syndrome. ISS[23:14] are RES0.

`0b1` ISS[23:14] hold a valid instruction syndrome.

This bit is 0 for all faults reported in ESR_EL2 except the following stage 2 aborts:

- AArch64 loads and stores of a single general-purpose register (including the register specified with `0b11111`, including those with Acquire/Release semantics, but excluding Load Exclusive or Store Exclusive and excluding those with writeback.
- AArch32 instructions where the instruction:
  - Is an LDR, LDA, LDRT, LDRSH, LDRSHT, LDRH, LDAH, LDRHT, LDRSB, LDRSBT, LDRB, LDAB, LDRBT, STR, STL, STRT, STRH, STLH, STRHT, STRB, STLB, or STRBT instruction.
  - Is not performing register writeback.
  - Is not using R15 as a source or destination register.

For these cases, ISV is UNKNOWN if the exception was generated in Debug state in memory access mode, and otherwise indicates whether ISS[23:14] hold a valid syndrome.

ISV is 0 for all faults reported in ESR_EL1 or ESR_EL3.

When the RAS Extension is implemented, ISV is 0 for any synchronous External abort.

For ISS reporting, a stage 2 abort on a stage 1 translation table walk does not return a valid instruction syndrome, and therefore ISV is 0 for these aborts.

When the RAS Extension is not implemented, the value of ISV on a synchronous External abort on a stage 2 translation table walk is IMPLEMENTATION DEFINED.

When ARMv8.5-MemTag is implemented, for a synchronous Tag Check Fault abort taken to ELx, ESR_ELx.FNV is 0 and FAR_ELx is valid.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

SAS, bits [23:22]

Syndrome Access Size. When ISV is 1, indicates the size of the access attempted by the faulting operation.

`0b00` Byte

`0b01` Halfword

`0b10` Word

`0b11` Doubleword

This field is UNKNOWN when the value of ISV is UNKNOWN.

This field is RES0 when the value of ISV is 0.

This field resets to an architecturally UNKNOWN value.

SSE, bit [21]

Syndrome Sign Extend. When ISV is 1, for a byte, halfword, or word load operation, indicates whether the data item must be sign extended. For these cases, the possible values of this bit are:

`0b0` Sign-extension not required.

`0b1` Data item must be sign-extended.

For all other operations this bit is 0.

This field is UNKNOWN when the value of ISV is UNKNOWN.

This field is RES0 when the value of ISV is 0.

This field resets to an architecturally UNKNOWN value.

SRT, bits [20:16]

Syndrome Register Transfer. When ISV is 1, the register number of the Rt operand of the faulting instruction.

If the exception was taken from an Exception level that is using AArch32 then this is the AArch64 view of the register. See *Mapping of the general-purpose registers between the Execution states* on page D1-2407.

This field is UNKNOWN when the value of ISV is UNKNOWN.

This field is RES0 when the value of ISV is 0.

This field resets to an architecturally UNKNOWN value.

SF, bit [15]

Width of the register accessed by the instruction is Sixty-Four. When ISV is 1, the possible values of this bit are:

`0b0` Instruction loads/stores a 32-bit wide register.

`0b1` Instruction loads/stores a 64-bit wide register.

Note

This field specifies the register width identified by the instruction, not the Execution state.

This field is UNKNOWN when the value of ISV is UNKNOWN.

This field is RES0 when the value of ISV is 0.

This field resets to an architecturally UNKNOWN value.

AR, bit [14]

Acquire/Release. When ISV is 1, the possible values of this bit are:

`0b0` Instruction did not have acquire/release semantics.

`0b1` Instruction did have acquire/release semantics.

This field is UNKNOWN when the value of ISV is UNKNOWN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field is RES0 when the value of ISV is 0.

This field resets to an architecturally UNKNOWN value.

VNCR, bit [13]

Indicates that the fault came from use of VNCR_EL2 register by EL1 code.

`0b0` The fault was not generated by the use of VNCR_EL2, by an MRS or MSR instruction executed at EL1.

`0b1` The fault was generated by the use of VNCR_EL2, by an MRS or MSR instruction executed at EL1.

This field is 0 in ESR_EL1.

This field resets to an architecturally UNKNOWN value.

SET, bits [12:11]

Synchronous Error Type. When the RAS Extension is implemented and DFSC is `0b010000`, describes the state of the PE after taking the Data Abort exception. The possible values of this field are:

`0b00` Recoverable error (UER).

`0b10` Uncontainable error (UC).

`0b11` Restartable error (UEO) or Corrected error (CE).

All other values are reserved.

Note

Software can use this information to determine what recovery might be possible. Taking a synchronous External Abort exception might result in an unrecoverable PE state.

This field is RES0 if either:

- The RAS Extension is not implemented.
- The value returned in the DFSC field is not `0b010000`.

This field resets to an architecturally UNKNOWN value.

FnV, bit [10]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

`0b0` FAR is valid.

`0b1` FAR is not valid, and holds an UNKNOWN value.

This field is valid only if the DFSC code is `0b010000`. It is RES0 for all other aborts.

This field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

This field resets to an architecturally UNKNOWN value.

CM, bit [8]

Cache maintenance. Indicates whether the Data Abort came from a cache maintenance or address translation instruction:

`0b0` The Data Abort was not generated by the execution of one of the System instructions identified in the description of value 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` The Data Abort was generated by either the execution of a cache maintenance instruction or by a synchronous fault on the execution of an address translation instruction. The DC ZVA instruction is not classified as a cache maintenance instruction, and therefore its execution cannot cause this field to be set to 1.

This field resets to an architecturally UNKNOWN value.

S1PTW, bit [7]

For a stage 2 fault, indicates whether the fault was a stage 2 fault on an access made for a stage 1 translation table walk:

`0b0` Fault not on a stage 2 translation for a stage 1 translation table walk.

`0b1` Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

This field resets to an architecturally UNKNOWN value.

WnR, bit [6]

Write not Read. Indicates whether a synchronous abort was caused by an instruction writing to a memory location, or by an instruction reading from a memory location. The possible values of this bit are:

`0b0` Abort caused by an instruction reading from a memory location.

`0b1` Abort caused by an instruction writing to a memory location.

For faults on cache maintenance and address translation instructions, this bit always returns a value of 1.

For faults from an atomic instruction that both reads and writes from a memory location, this bit is set to 0 if a read of the address specified by the instruction would have generated the fault which is being reported, otherwise it is set to 1. The architecture permits, but does not require, a relaxation of this requirement such that for all stage 2 aborts on stage 1 translation table walks for atomic instructions, the WnR bit is always 0.

This field is UNKNOWN for:

- An External abort on an Atomic access.
- A fault reported using a DFSC value of `0b110101` or `0b110001`, indicating an unsupported Exclusive or atomic access.

This field resets to an architecturally UNKNOWN value.

DFSC, bits [5:0]

Data Fault Status Code.

`0b000000` Address size fault, level 0 of translation or translation table base register.

`0b000001` Address size fault, level 1.

`0b000010` Address size fault, level 2.

`0b000011` Address size fault, level 3.

`0b000100` Translation fault, level 0.

`0b000101` Translation fault, level 1.

`0b000110` Translation fault, level 2.

`0b000111` Translation fault, level 3.

`0b001001` Access flag fault, level 1.

`0b001010` Access flag fault, level 2.

`0b001011` Access flag fault, level 3.

`0b001101` Permission fault, level 1.

`0b001110` Permission fault, level 2.

`0b001111` Permission fault, level 3.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b010000` Synchronous External abort, not on translation table walk or hardware update of translation table.

`0b010001` **When ARMv8.5-MemTag is implemented** Synchronous Tag Check Fault.

`0b010100` Synchronous External abort, on translation table walk or hardware update of translation table, level 0.

`0b010101` Synchronous External abort, on translation table walk or hardware update of translation table, level 1.

`0b010110` Synchronous External abort, on translation table walk or hardware update of translation table, level 2.

`0b010111` Synchronous External abort, on translation table walk or hardware update of translation table, level 3.

`0b011000` **When RAS is not implemented** Synchronous parity or ECC error on memory access, not on translation table walk.

`0b011100` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 0.

`0b011101` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 1.

`0b011110` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 2.

`0b011111` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 3.

`0b100001` Alignment fault.

`0b110000` TLB conflict abort.

`0b110001` **When ARMv8.1-TTHM is implemented** Unsupported atomic hardware update fault.

`0b110100` IMPLEMENTATION DEFINED fault (Lockdown).

`0b110101` IMPLEMENTATION DEFINED fault (Unsupported Exclusive or Atomic access).

All other values are reserved.

For more information about the lookup level associated with a fault, see *The level associated with MMU faults* on page D5-2658.

Note

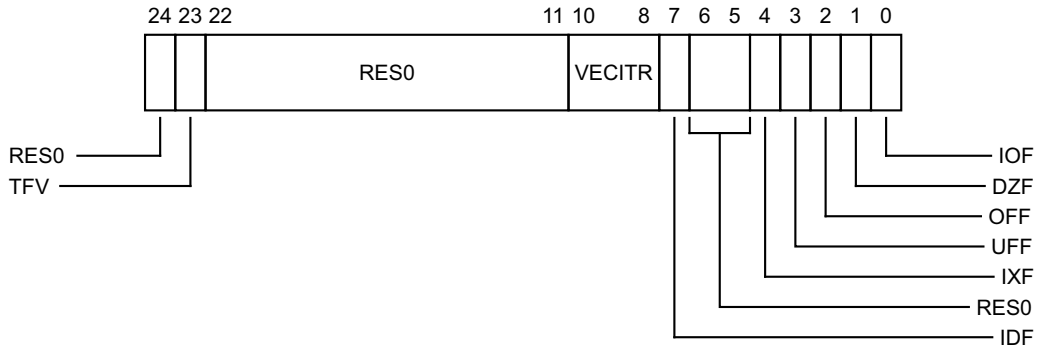
Because Access flag faults and Permission faults can only result from a Block or Page translation table descriptor, they cannot occur at level 0.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ISS encoding for an exception from a trapped floating-point exception



Bit [24]

Reserved, RES0.

TFV, bit [23]

Trapped Fault Valid bit. Indicates whether the IDF, IXF, UFF, OFF, DZF, and IOF bits hold valid information about trapped floating-point exceptions. The possible values of this bit are:

`0b0` The IDF, IXF, UFF, OFF, DZF, and IOF bits do not hold valid information about trapped floating-point exceptions and are UNKNOWN.

`0b1` One or more floating-point exceptions occurred during an operation performed while executing the reported instruction. The IDF, IXF, UFF, OFF, DZF, and IOF bits indicate trapped floating-point exceptions that occurred. For more information see *Floating-point exceptions and exception traps* on page D1-2360.

It is IMPLEMENTATION DEFINED whether this field is set to 0 on an exception generated by a trapped floating point exception from a vector instruction.

Note

This is not a requirement. Implementations can set this field to 1 on a trapped floating-point exception from a vector instruction and return valid information in the {IDF, IXF, UFF, OFF, DZF, IOF} fields.

This field resets to an architecturally UNKNOWN value.

Bits [22:11]

Reserved, RES0.

VECITR, bits [10:8]

For a trapped floating-point exception from an instruction executed in AArch32 state this field is RES1.

For a trapped floating-point exception from an instruction executed in AArch64 state this field is UNKNOWN.

This field resets to an architecturally UNKNOWN value.

IDF, bit [7]

Input Denormal floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

`0b0` Input denormal floating-point exception has not occurred.

`0b1` Input denormal floating-point exception occurred during execution of the reported instruction.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bits [6:5]

Reserved, RES0.

IXF, bit [4]

Inexact floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

`0b0` Inexact floating-point exception has not occurred.

`0b1` Inexact floating-point exception occurred during execution of the reported instruction.

This field resets to an architecturally UNKNOWN value.

UFF, bit [3]

Underflow floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

`0b0` Underflow floating-point exception has not occurred.

`0b1` Underflow floating-point exception occurred during execution of the reported instruction.

This field resets to an architecturally UNKNOWN value.

OFF, bit [2]

Overflow floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

`0b0` Overflow floating-point exception has not occurred.

`0b1` Overflow floating-point exception occurred during execution of the reported instruction.

This field resets to an architecturally UNKNOWN value.

DZF, bit [1]

Divide by Zero floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

`0b0` Divide by Zero floating-point exception has not occurred.

`0b1` Divide by Zero floating-point exception occurred during execution of the reported instruction.

This field resets to an architecturally UNKNOWN value.

IOF, bit [0]

Invalid Operation floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

`0b0` Invalid Operation floating-point exception has not occurred.

`0b1` Invalid Operation floating-point exception occurred during execution of the reported instruction.

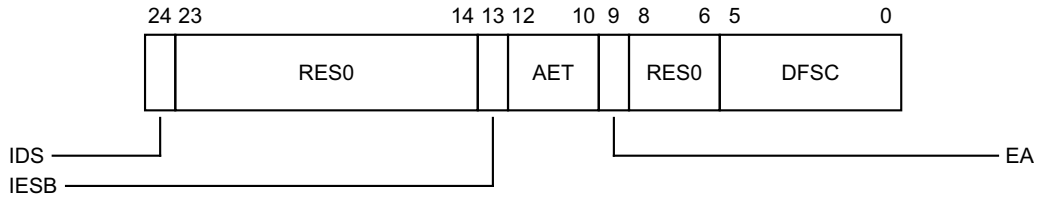
This field resets to an architecturally UNKNOWN value.

In an implementation that supports the trapping of floating-point exceptions:

- From an Exception level using AArch64, the FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} bits enable each of the floating-point exception traps.
- From an Exception level using AArch32, the FPSCR.{IDE, IXE, UFE, OFE, DZE, IOE} bits enable each of the floating-point exception traps.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ISS encoding for an SError interrupt



IDS, bit [24]

IMPLEMENTATION DEFINED syndrome. Possible values of this bit are:

`0b0` Bits[23:0] of the ISS field holds the fields described in this encoding.

Note

If the RAS Extension is not implemented, this means that bits[23:0] of the ISS field are RES0.

`0b1` Bits[23:0] of the ISS field holds IMPLEMENTATION DEFINED syndrome information that can be used to provide additional information about the SError interrupt.

Note

This field was previously called ISV.

This field resets to an architecturally UNKNOWN value.

Bits [23:14]

Reserved, RES0.

IESB, bit [13]

Implicit error synchronization event.

`0b0` The SError interrupt was either not synchronized by the implicit error synchronization event or not taken immediately.

`0b1` The SError interrupt was synchronized by the implicit error synchronization event and taken immediately.

This field is RES0 if the value returned in the DFSC field is not `0b010001`.

This field resets to an architecturally UNKNOWN value.

AET, bits [12:10]

Asynchronous Error Type.

When the RAS Extension is implemented and DFSC is `0b010001`, describes the state of the PE after taking the SError interrupt exception. The possible values of this field are:

`0b000` Uncontainable error (UC).

`0b001` Unrecoverable error (UEU).

`0b010` Restartable error (UEO).

`0b011` Recoverable error (UER).

`0b110` Corrected error (CE).

All other values are reserved.

If multiple errors are taken as a single SError interrupt exception, the overall state of the PE is reported. For example, if both a Recoverable and Unrecoverable error occurred, the state is Unrecoverable.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

Software can use this information to determine what recovery might be possible. The recovery software must also examine any implemented fault records to determine the location and extent of the error.

This field is RES0 if either:

- The RAS Extension is not implemented.
- The value returned in the DFSC field is not `0b010001`.

This field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort type. When the RAS Extension is implemented, this bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

This field is RES0 if either:

- The RAS Extension is not implemented.
- The value returned in the DFSC field is not `0b010001`.

This field resets to an architecturally UNKNOWN value.

Bits [8:6]

Reserved, RES0.

DFSC, bits [5:0]

When RAS is implemented:

Data Fault Status Code.

`0b000000` Uncategorized.

`0b010001` Asynchronous SError interrupt.

All other values are reserved.

This field resets to an architecturally UNKNOWN value.

ISS encoding for an exception from a Breakpoint or Vector Catch debug exception

24 … 6	5 … 0
RES0	IFSC

Bits [24:6]

Reserved, RES0.

IFSC, bits [5:0]

Instruction Fault Status Code.

`0b100010` Debug exception.

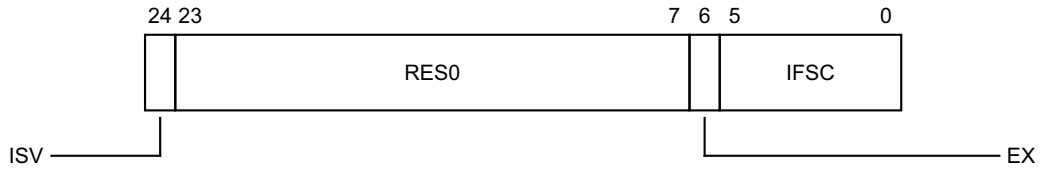
This field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions:

- For exceptions from AArch64, see *Breakpoint exceptions* on page D2-2439.
- For exceptions from AArch32, see *Breakpoint exceptions* on page G2-5830 and *Vector Catch exceptions* on page G2-5869.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ISS encoding for an exception from a Software Step exception



ISV, bit [24]

Instruction syndrome valid. Indicates whether the EX bit, ISS[6], is valid, as follows:

`0b0` EX bit is RES0.

`0b1` EX bit is valid.

See the EX bit description for more information.

This field resets to an architecturally UNKNOWN value.

Bits [23:7]

Reserved, RES0.

EX, bit [6]

Exclusive operation. If the ISV bit is set to 1, this bit indicates whether a Load-Exclusive instruction was stepped.

`0b0` An instruction other than a Load-Exclusive instruction was stepped.

`0b1` A Load-Exclusive instruction was stepped.

If the ISV bit is set to 0, this bit is RES0, indicating no syndrome data is available.

This field resets to an architecturally UNKNOWN value.

IFSC, bits [5:0]

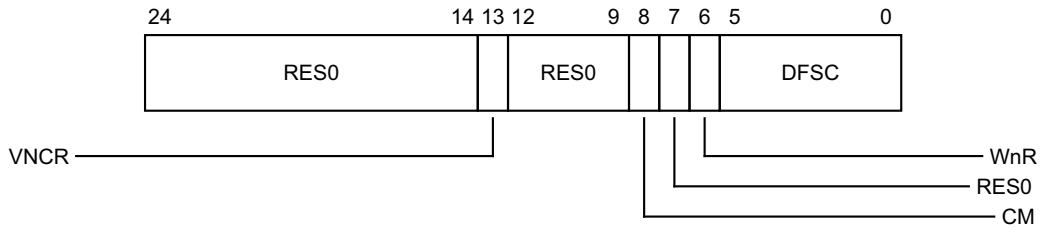
Instruction Fault Status Code.

`0b100010` Debug exception.

This field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions, see *Software Step exceptions* on page D2-2472.

ISS encoding for an exception from a Watchpoint exception



Bits [24:14]

Reserved, RES0.

VNCR, bit [13]

Indicates that the watchpoint came from use of VNCR_EL2 register by EL1 code.

`0b0` The watchpoint was not generated by the use of VNCR_EL2 by EL1 code.

`0b1` The watchpoint was generated by the use of VNCR_EL2 by EL1 code.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field is 0 in ESR_EL1.

This field resets to an architecturally UNKNOWN value.

Bits [12:9]

Reserved, RES0.

CM, bit [8]

Cache maintenance. Indicates whether the Watchpoint exception came from a cache maintenance or address translation instruction:

`0b0` The Watchpoint exception was not generated by the execution of one of the System instructions identified in the description of value 1.

`0b1` The Watchpoint exception was generated by either the execution of a cache maintenance instruction or by a synchronous Watchpoint exception on the execution of an address translation instruction. The DC ZVA instruction is not classified as a cache maintenance instruction, and therefore its execution cannot cause this field to be set to 1.

This field resets to an architecturally UNKNOWN value.

Bit [7]

Reserved, RES0.

WnR, bit [6]

Write not Read. Indicates whether the Watchpoint exception was caused by an instruction writing to a memory location, or by an instruction reading from a memory location. The possible values of this bit are:

`0b0` Watchpoint exception caused by an instruction reading from a memory location.

`0b1` Watchpoint exception caused by an instruction writing to a memory location.

For Watchpoint exceptions on cache maintenance and address translation instructions, this bit always returns a value of 1.

For Watchpoint exceptions from an atomic instruction, this field is set to 0 if a read of the location would have generated the Watchpoint exception, otherwise it is set to 1.

If multiple watchpoints match on the same access, it is UNPREDICTABLE which watchpoint generates the Watchpoint exception.

This field resets to an architecturally UNKNOWN value.

DFSC, bits [5:0]

Data Fault Status Code.

`0b100010` Debug exception.

This field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions, see *Watchpoint exceptions* on page D2-2457.

ISS encoding for an exception from execution of a Breakpoint instruction

24 … 16	15 … 0
RES0	Comment

Bits [24:16]

Reserved, RES0.

Comment, bits [15:0]

Set to the instruction comment field value, zero extended as necessary.

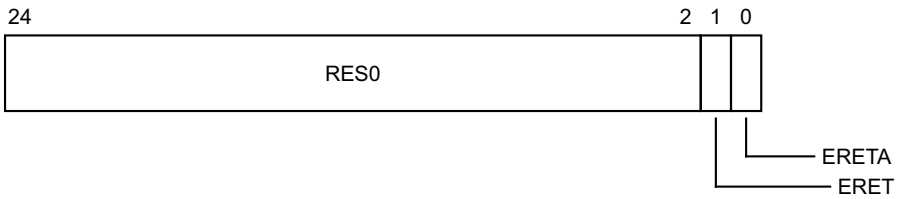
Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the AArch32 BKPT instructions, the comment field is described as the immediate field.

This field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions, see *Breakpoint Instruction exceptions* on page D2-2437.

ISS encoding for an exception from ERET, ERETAA or ERETAB instruction



This EC value applies when ARMv8.6-FGT is implemented, or when HCR_EL2.NV is 1.

Bits [24:2]

Reserved, RES0.

ERET, bit [1]

Indicates whether an ERET or ERETA* instruction was trapped to EL2. Possible values are:

`0b0` ERET instruction trapped to EL2.

`0b1` ERETAA or ERETAB instruction trapped to EL2.

If this bit is 0, the ERETA field is RES0.

This field resets to an architecturally UNKNOWN value.

ERETA, bit [0]

Indicates whether an ERETAA or ERETAB instruction was trapped to EL2. Possible values are:

`0b0` ERETAA instruction trapped to EL2.

`0b1` ERETAB instruction trapped to EL2.

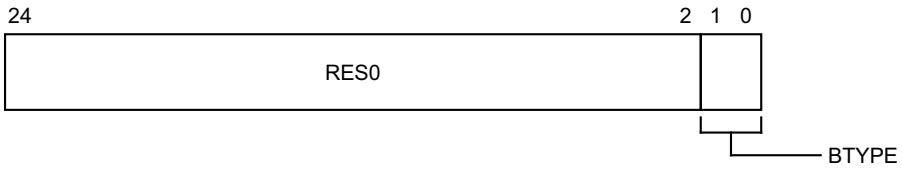
When the ERET field is 0, this bit is RES0.

This field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions, see HCR_EL2.NV.

If ARMv8.6-FGT is implemented, HFGITR_EL2.ERET controls fine-grained trap exceptions from ERET, ERETAA and ERETAB execution.

ISS encoding for an exception from Branch Target Identification instruction



Bits [24:2]

Reserved, RES0.

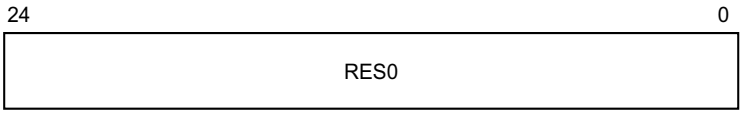
BTYPE, bits [1:0]

This field is set to the PSTATE.BTYPE value that generated the Branch Target Exception.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For more information about generating these exceptions, see Chapter B1 *The AArch64 Application Level Programmers' Model*.

ISS encoding for an exception from a Pointer Authentication instruction when HCR_EL2.API == 0 || SCR_EL3.API == 0



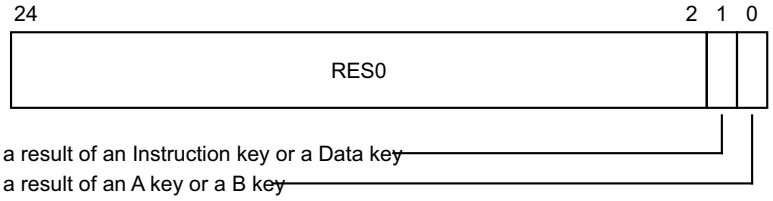
Bits [24:0]

Reserved, RES0.

For more information about generating these exceptions, see:

- HCR_EL2.API, for exceptions from Pointer authentication instructions, using AArch64 state, trapped to EL2.
- SCR_EL3.API, for exceptions from Pointer authentication instructions, using AArch64 state, trapped to EL3.

ISS encoding for an exception from a Pointer Authentication instruction authentication failure



Bits [24:2]

Reserved, RES0.

Bit [1]

This field indicates whether the exception is as a result of an Instruction key or a Data key.

`0b0` Instruction Key.

`0b1` Data Key.

This field resets to an architecturally UNKNOWN value.

Bit [0]

This field indicates whether the exception is as a result of an A key or a B key.

`0b0` A key.

`0b1` B key.

This field resets to an architecturally UNKNOWN value.

The following instructions generate an exception when the Pointer Authentication Code (PAC) is incorrect:

- `AUTIASP`, `AUTIAZ`, `AUTIA1716`.
- `AUTIBSP`, `AUTIBZ`, `AUTIB1716`.
- `AUTIA`, `AUTDA`, `AUTIB`, `AUTDB`.
- `AUTIZA`, `AUTIZB`, `AUTDZA`, `AUTDZB`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

It is IMPLEMENTATION DEFINED whether the following instructions generate an exception directly from the authorization failure, rather than changing the address in a way that will generate a translation fault when the address is accessed:

- RETAA, RETAB.
- BRAA, BRAB, BLRAA, BLRAB.
- BRAAZ, BRABZ, BLRAAZ, BLRABZ.
- ERETAA, ERETAB.
- LDRAA, LDRAB, whether the authenticated address is written back to the base register or not.

Accessing the ESR_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic ESR_EL2 or ESR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ESR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        return ESR_EL1;
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return ESR_EL2;
elsif PSTATE.EL == EL3 then
    return ESR_EL2;
```

MSR ESR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        ESR_EL1 = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    ESR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    ESR_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, ESR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.ESR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x138];
    else
        return ESR_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return ESR_EL2;
    else
        return ESR_EL1;
elsif PSTATE.EL == EL3 then
    return ESR_EL1;
```

MSR ESR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.ESR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x138] = X[t];
    else
        ESR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        ESR_EL2 = X[t];
    else
        ESR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    ESR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.38 ESR_EL3, Exception Syndrome Register (EL3)

The ESR_EL3 characteristics are:

Purpose

Holds syndrome information for an exception taken to EL3.

Configurations

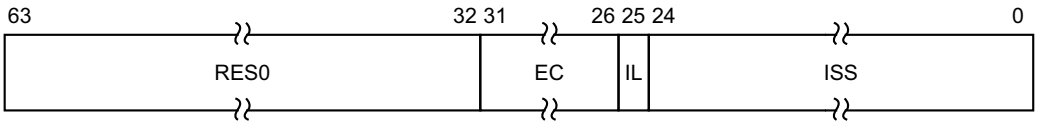
This register is present only when EL3 is implemented. Otherwise, direct accesses to ESR_EL3 are UNDEFINED.

Attributes

ESR_EL3 is a 64-bit register.

Field descriptions

The ESR_EL3 bit assignments are:



ESR_EL3 is made UNKNOWN as a result of an exception return from EL3.

When an UNPREDICTABLE instruction is treated as UNDEFINED, and the exception is taken to EL3, the value of ESR_EL3 is UNKNOWN. The value written to ESR_EL3 must be consistent with a value that could be created as a result of an exception from the same Exception level that generated the exception as a result of a situation that is not UNPREDICTABLE at that Exception level, in order to avoid the possibility of a privilege violation.

Bits [63:32]

Reserved, RES0.

EC, bits [31:26]

Exception Class. Indicates the reason for the exception that this register holds information about.

For each EC value, the table references a subsection that gives information about:

- The cause of the exception, for example the configuration required to enable the trap.
- The encoding of the associated ISS.

Possible values of the EC field are:

EC == 0b000000

Unknown reason.

See ISS encoding for exceptions with an unknown reason.

EC == 0b000001

Trapped WFI or WFE instruction execution.

Conditional WFE and WFI instructions that fail their condition code check do not cause an exception.

See ISS encoding for an exception from a WFI or WFE instruction.

EC == 0b000011 When AArch32 is supported at any Exception level

Trapped MCR or MRC access with (coproc==`0b1111`) that is not reported using EC `0b000000`.

See ISS encoding for an exception from an MCR or MRC access.

EC == 0b000100 When AArch32 is supported at any Exception level

Trapped MCRR or MRRC access with (coproc==`0b1111`) that is not reported using EC `0b000000`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

See ISS encoding for an exception from an MCRR or MRRC access.

EC == 0b000101 When AArch32 is supported at any Exception level

Trapped MCR or MRC access with (coproc==`0b1110`).

See ISS encoding for an exception from an MCR or MRC access.

EC == 0b000110 When AArch32 is supported at any Exception level

Trapped LDC or STC access.

The only architected uses of these instruction are:

- An STC to write data to memory from DBGDTRRXint.
- An LDC to read data from memory to DBGDTRTXint.

See ISS encoding for an exception from an LDC or STC instruction.

EC == 0b000111

Access to SVE, Advanced SIMD, or floating-point functionality trapped by CPACR_EL1.FPEN, CPTR_EL2.FPEN, CPTR_EL2.TFP, or CPTR_EL3.TFP control.

Excludes exceptions resulting from CPACR_EL1 when the value of HCR_EL2.TGE is 1, or because SVE or Advanced SIMD and floating-point are not implemented. These are reported with EC value `0b000000` as described in *The EC used to report an exception routed to EL2 because HCR_EL2.TGE is 1* on page D1-2347.

See ISS encoding for an exception from an access to SVE, Advanced SIMD or floating-point functionality, resulting from CPACR_EL1.FPEN, CPTR_EL2.FPEN or CPTR_ELx.TFP.

EC == 0b001001 When ARMv8.3-PAuth is implemented

Trapped use of a Pointer authentication instruction because HCR_EL2.API == 0 || SCR_EL3.API == 0.

See *ISS encoding for an exception from a Pointer Authentication instruction when HCR_EL2.API == 0 || SCR_EL3.API == 0* on page D13-3084.

EC == 0b001100 When AArch32 is supported at any Exception level

Trapped MRRC access with (coproc==`0b1110`).

See ISS encoding for an exception from an MCRR or MRRC access.

EC == 0b001101 When ARMv8.5-BTI is implemented

Branch Target Exception.

See ISS encoding for an exception from Branch Target Identification instruction.

EC == 0b001110

Illegal Execution state.

See ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault.

EC == 0b010011 When AArch32 is supported at any Exception level

SMC instruction execution in AArch32 state, when SMC is not disabled.

This is reported in ESR_EL2 only when the exception is generated because the value of HCR_EL2.TSC is 1.

See ISS encoding for an exception from SMC instruction execution in AArch32 state.

EC == 0b010101 When AArch64 is supported at any Exception level

SVC instruction execution in AArch64 state.

See ISS encoding for an exception from HVC or SVC instruction execution.

EC == 0b010110 When AArch64 is supported at any Exception level

HVC instruction execution in AArch64 state, when HVC is not disabled.

See ISS encoding for an exception from HVC or SVC instruction execution.

EC == 0b010111 When AArch64 is supported at any Exception level

SMC instruction execution in AArch64 state, when SMC is not disabled.

This is reported in ESR_EL2 only when the exception is generated because the value of HCR_EL2.TSC is 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

See ISS encoding for an exception from SMC instruction execution in AArch64 state.

EC == 0b011000 When AArch64 is supported at any Exception level

Trapped MSR, MRS or System instruction execution in AArch64 state, that is not reported using EC `0b000000`, `0b000001` or `0b000111`.

This includes all instructions that cause exceptions that are part of the encoding space defined in *System instruction class encoding overview* on page C5-377, except for those exceptions reported using EC values `0b000000`, `0b000001`, or `0b000111`.

See ISS encoding for an exception from MSR, MRS, or System instruction execution in AArch64 state.

EC == 0b011001 When SVE is implemented

Access to SVE functionality trapped as a result of CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ, that is not reported using EC `0b000000`.

See ISS encoding for an exception from an access to SVE functionality, resulting from CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ.

EC == 0b011100 When ARMv8.3-FPAC is implemented

Exception from a Pointer Authentication instruction authentication failure

See ISS encoding for an exception from a Pointer Authentication instruction authentication failure.

EC == 0b011111

IMPLEMENTATION DEFINED exception to EL3.

See ISS encoding for an IMPLEMENTATION DEFINED exception to EL3.

EC == 0b100000

Instruction Abort from a lower Exception level.

Used for MMU faults generated by instruction accesses and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug related exceptions.

See ISS encoding for an exception from an Instruction Abort.

EC == 0b100001

Instruction Abort taken without a change in Exception level.

Used for MMU faults generated by instruction accesses and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug related exceptions.

See ISS encoding for an exception from an Instruction Abort.

EC == 0b100010

PC alignment fault exception.

See ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault.

EC == 0b100100

Data Abort from a lower Exception level.

Used for MMU faults generated by data accesses, alignment faults other than those caused by Stack Pointer misalignment, and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug related exceptions.

See ISS encoding for an exception from a Data Abort.

EC == 0b100101

Data Abort taken without a change in Exception level.

Used for MMU faults generated by data accesses, alignment faults other than those caused by Stack Pointer misalignment, and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug related exceptions.

See ISS encoding for an exception from a Data Abort.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

EC == 0b100110

SP alignment fault exception.

See ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault.

EC == 0b101100 When AArch64 is supported at any Exception level

Trapped floating-point exception taken from AArch64 state.

This EC value is valid if the implementation supports trapping of floating-point exceptions, otherwise it is reserved. Whether a floating-point implementation supports trapping of floating-point exceptions is IMPLEMENTATION DEFINED.

See ISS encoding for an exception from a trapped floating-point exception.

EC == 0b101111

SError interrupt.

See ISS encoding for an SError interrupt.

EC == 0b111100 When AArch64 is supported at any Exception level

BRK instruction execution in AArch64 state.

This is reported in ESR_EL3 only if a BRK instruction is executed.

See ISS encoding for an exception from execution of a Breakpoint instruction.

All other EC values are reserved by Arm, and:

- Unused values in the range `0b000000` - `0b101100` (`0x00` - `0x2C`) are reserved for future use for synchronous exceptions.
- Unused values in the range `0b101101` - `0b111111` (`0x2D` - `0x3F`) are reserved for future use, and might be used for synchronous or asynchronous exceptions.

The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-7853.

This field resets to an architecturally UNKNOWN value.

IL, bit [25]

Instruction Length for synchronous exceptions. Possible values of this bit are:

`0b0` 16-bit instruction trapped.

`0b1` 32-bit instruction trapped. This value is also used when the exception is one of the following:

- An SError interrupt.
- An Instruction Abort exception.
- A PC alignment fault exception.
- An SP alignment fault exception.
- A Data Abort exception for which the value of the ISV bit is 0.
- An Illegal Execution state exception.
- Any debug exception except for Breakpoint instruction exceptions. For Breakpoint instruction exceptions, this bit has its standard meaning:
  - `0b0`: 16-bit T32 BKPT instruction.
  - `0b1`: 32-bit A32 BKPT instruction or A64 BRK instruction.
- An exception reported using EC value `0b000000`.

This field resets to an architecturally UNKNOWN value.

ISS, bits [24:0]

Instruction Specific Syndrome. Architecturally, this field can be defined independently for each defined Exception class. However, in practice, some ISS encodings are used for more than one Exception class.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Typically, an ISS encoding has a number of subfields. When an ISS subfield holds a register number, the value returned in that field is the AArch64 view of the register number.

For an exception taken from AArch32 state, *Mapping of the general-purpose registers between the Execution states* on page D1-2407.

If the AArch32 register descriptor is `0b1111`, then:

- If the instruction that generated the exception was not UNPREDICTABLE, the field takes the value `0b11111`.
- If the instruction that generated the exception was UNPREDICTABLE, the field takes an UNKNOWN value that must be either:
  - The AArch64 view of the register number of a register that might have been used at the Exception level from which the exception was taken.
  - The value `0b11111`.

When the EC field is `0b000000`, indicating an exception with an unknown reason, the ISS field is not valid, RES0.

The following subsections describe each ISS format.

ISS encoding for exceptions with an unknown reason

24 … 0
RES0

Bits [24:0]

Reserved, RES0.

When an exception is reported using this EC code the IL field is set to 1.

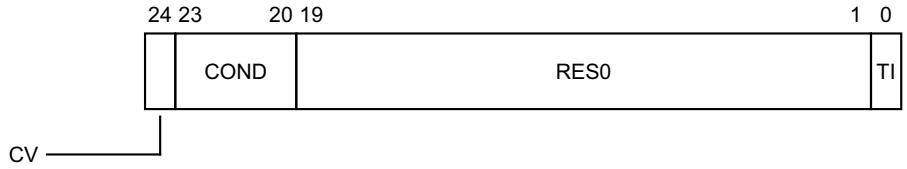
This EC code is used for all exceptions that are not covered by any other EC value. This includes exceptions that are generated in the following situations:

- The attempted execution of an instruction bit pattern that has no allocated instruction or that is not accessible at the current Exception level and Security state, including:
  - A read access using a System register pattern that is not allocated for reads or that does not permit reads at the current Exception level and Security state.
  - A write access using a System register pattern that is not allocated for writes or that does not permit writes at the current Exception level and Security state.
  - Instruction encodings that are unallocated.
  - Instruction encodings for instructions or System registers that are not implemented in the implementation.
- In Debug state, the attempted execution of an instruction bit pattern that is not accessible in Debug state.
- In Non-debug state, the attempted execution of an instruction bit pattern that is not accessible in Non-debug state.
- In AArch32 state, attempted execution of a short vector floating-point instruction.
- In an implementation that does not include Advanced SIMD and floating-point functionality, an attempted access to Advanced SIMD or floating-point functionality under conditions where that access would be permitted if that functionality was present. This includes the attempted execution of an Advanced SIMD or floating-point instruction, and attempted accesses to Advanced SIMD and floating-point System registers.
- An exception generated because of the value of one of the SCTLR_EL1.{ITD, SED, CP15BEN} control bits.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Attempted execution of:
  - An HVC instruction when disabled by HCR_EL2.HCD or SCR_EL3.HCE.
  - An SMC instruction when disabled by SCR_EL3.SMD.
  - An HLT instruction when disabled by EDSCR.HDE.
- Attempted execution of an MSR or MRS instruction to access SP_EL0 when the value of SPSel.SP is 0.
- Attempted execution, in Debug state, of:
  - A DCPS1 instruction when the value of HCR_EL2.TGE is 1 and EL2 is disabled or not implemented in the current Security state.
  - A DCPS2 instruction from EL1 or EL0 when EL2 is disabled or not implemented in the current Security state.
  - A DCPS3 instruction when the value of EDSCR.SDD is 1, or when EL3 is not implemented.
- When EL3 is using AArch64, attempted execution from Secure EL1 of an SRS instruction using R13_mon. See *Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32* on page D1-2391.
- In Debug state when the value of EDSCR.SDD is 1, the attempted execution at EL2, EL1, or EL0 of an instruction that is configured to trap to EL3.
- In AArch32 state, the attempted execution of an MRS (banked register) or an MSR (banked register) instruction to SPSR_mon, SP_mon, or LR_mon.
- An exception that is taken to EL2 because the value of HCR_EL2.TGE is 1 that, if the value of HCR_EL2.TGE was 0 would have been reported with an ESR_ELx.EC value of `0b000111`.

ISS encoding for an exception from a WFI or WFE instruction



CV, bit [24]

Condition code valid. Possible values of this bit are:

`0b0` The COND field is not valid.

`0b1` The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to `0b1110`.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - — If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - — If the instruction is unconditional, COND is set to `0b1110`.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - — With COND set to `0b1110`, the value for unconditional.
  - — With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - — CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - — CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

This field resets to an architecturally UNKNOWN value.

Bits [19:1]

Reserved, RES0.

TI, bit [0]

Trapped instruction. Possible values of this bit are:

`0b0` WFI trapped.

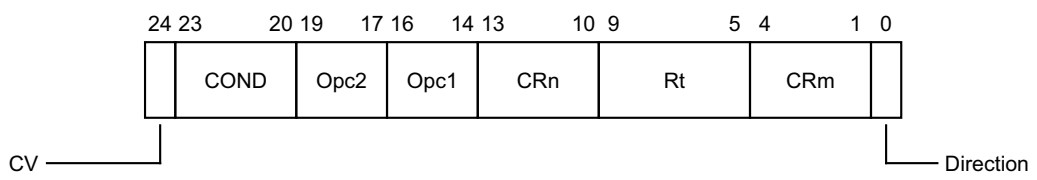
`0b1` WFE trapped.

This field resets to an architecturally UNKNOWN value.

The following fields describe configuration settings for generating this exception:

- SCTLR_EL1.{nTWE, nTWI}.
- HCR_EL2.{TWE, TWI}.
- SCR_EL3.{TWE, TWI}.

ISS encoding for an exception from an MCR or MRC access



CV, bit [24]

Condition code valid. Possible values of this bit are:

`0b0` The COND field is not valid.

`0b1` The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to `0b1110`.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to `0b1110`.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to `0b1110`, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

This field resets to an architecturally UNKNOWN value.

Opc2, bits [19:17]

The Opc2 value from the issued instruction.

For a trapped VMRS access, holds the value `0b000`.

This field resets to an architecturally UNKNOWN value.

Opc1, bits [16:14]

The Opc1 value from the issued instruction.

For a trapped VMRS access, holds the value `0b111`.

This field resets to an architecturally UNKNOWN value.

CRn, bits [13:10]

The CRn value from the issued instruction.

For a trapped VMRS access, holds the reg field from the VMRS instruction encoding.

This field resets to an architecturally UNKNOWN value.

Rt, bits [9:5]

The Rt value from the issued instruction, the general-purpose register used for the transfer. The reported value gives the AArch64 view of the register. See *Mapping of the general-purpose registers between the Execution states* on page D1-2407.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CRm, bits [4:1]

The CRm value from the issued instruction.

For a trapped VMRS access, holds the value `0b0000`.

This field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction. The possible values of this bit are:

`0b0` Write to System register space. MCR instruction.

`0b1` Read from System register space. MRC or VMRS instruction.

This field resets to an architecturally UNKNOWN value.

The following fields describe configuration settings for generating exceptions that are reported using EC value `0b000011`:

- CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN}, for accesses to the Generic Timer Registers from EL0 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL1 or EL2.
- PMUSERENR_EL0.{ER, CR, SW, EN}, for accesses to Performance Monitor registers from EL0 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL1 or EL2.
- AMUSERENR_EL0.EN, for accesses to Activity Monitors registers from EL0 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL1 or EL2.
- HCR_EL2.{TRVM, TVM}, for accesses to virtual memory control registers from EL1 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL2.
- HCR_EL2.TTLB, for execution of TLB maintenance instructions at EL1 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL2.
- HCR_EL2.{TSW, TPC, TPU} for execution of cache maintenance instructions at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL2.
- HCR_EL2.TACR, for accesses to the Auxiliary Control Register at EL1 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL2.
- HCR_EL2.TIDCP, for accesses to lockdown, DMA, and TCM operations at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL2.
- HCR_EL2.{TID1, TID2, TID3}, for accesses to ID registers at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL2.
- CPTR_EL2.TCPAC, for accesses to CPACR_EL1 or CPACR using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL2.
- HSTR_EL2.T<n>, for accesses to System registers using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL2.
- CNTHCTL_EL2.EL1PCEN, for accesses to the Generic Timer registers from EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL2.
- MDCR_EL2.{TPM, TPMCR}, for accesses to Performance Monitor registers from EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL2.
- CPTR_EL2.TAM, for accesses to Activity Monitors registers from EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL2.
- CPTR_EL3.TCPAC, for accesses to CPACR from EL1 and EL2, and accesses to HCPTR from EL2 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL3.
- MDCR_EL3.TPM, for accesses to Performance Monitor registers from EL0, EL1 and EL2 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL3.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- CPTR_EL3.TAM, for accesses to Activity Monitors registers from EL0, EL1 and EL2 using AArch32 state, MCR or MRC access (coproc == `0b1111`) trapped to EL3.
- See *Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32* on page D1-2391 for information on other traps using EC value `0b000011`.
- If ARMv8.6-FGT is implemented, MCR or MRC access to some registers at EL0, trapped to EL2.

The following fields describe configuration settings for generating exceptions that are reported using EC value `0b000101`:

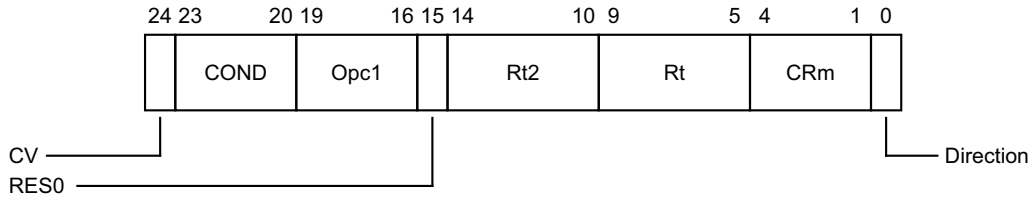
- CPACR_EL1.TTA for accesses to trace registers, MCR or MRC access (coproc == `0b1110`) trapped to EL1 or EL2.
- MDSCR_EL1.TDCC, for accesses to the Debug Communications Channel (DCC) registers at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == `0b1110`) trapped to EL1 or EL2.
- If ARMv8.6-FGT is implemented, MDCR_EL2.TDCC and MDCR_EL3.TDCC, for accesses to the DCC registers at EL0 and EL1, trapped to EL2.
- HCR_EL2.TID0, for accesses to the JIDR register in the ID group 0 at EL0 and EL1 using AArch32, MRC access (coproc == `0b1110`) trapped to EL2.
- CPTR_EL2.TTA, for accesses to trace registers using AArch32, MCR or MRC access (coproc == `0b1110`) trapped to EL2.
- MDCR_EL2.TDRA, for accesses to Debug ROM registers DBGDRAR and AArch-DBGDSAR using AArch32, MCR or MRC access (coproc == `0b1110`) trapped to EL2.
- MDCR_EL2.TDOSA, for accesses to powerdown debug registers, using AArch32 state, MCR or MRC access (coproc == `0b1110`) trapped to EL2.
- MDCR_EL2.TDA, for accesses to other debug registers, using AArch32 state, MCR or MRC access (coproc == `0b1110`) trapped to EL2.
- CPTR_EL3.TTA, for accesses to trace registers using AArch32, MCR or MRC access (coproc == `0b1110`) trapped to EL3.
- MDCR_EL3.TDOSA, for accesses to powerdown debug registers using AArch32, MCR or MRC access (coproc == `0b1110`) trapped to EL3.
- MDCR_EL3.TDA, for accesses to other debug registers, using AArch32, MCR or MRC access (coproc == `0b1110`) trapped to EL3.

The following fields describe configuration settings for generating exceptions that are reported using EC value `0b001000`:

- HCR_EL2.TID0, for accesses to the FPSID register in ID group 0 at EL1 using AArch32 state, VMRS access trapped to EL2.
- HCR_EL2.TID3, for accesses to registers in ID group 3 including MVFR0, MVFR1 and MVFR2, VMRS access trapped to EL2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ISS encoding for an exception from an MCRR or MRRC access



CV, bit [24]

Condition code valid. Possible values of this bit are:

`0b0` The COND field is not valid.

`0b1` The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to `0b1110`.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - — If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - — If the instruction is unconditional, COND is set to `0b1110`.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - — With COND set to `0b1110`, the value for unconditional.
  - — With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - — CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - — CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

This field resets to an architecturally UNKNOWN value.

Opc1, bits [19:16]

The Opc1 value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bit [15]

Reserved, RES0.

Rt2, bits [14:10]

The Rt2 value from the issued instruction, the second general-purpose register used for the transfer. The reported value gives the AArch64 view of the register. See *Mapping of the general-purpose registers between the Execution states* on page D1-2407.

This field resets to an architecturally UNKNOWN value.

Rt, bits [9:5]

The Rt value from the issued instruction, the first general-purpose register used for the transfer. The reported value gives the AArch64 view of the register. See *Mapping of the general-purpose registers between the Execution states* on page D1-2407.

This field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction. The possible values of this bit are:

`0b0` Write to System register space. MCRR instruction.

`0b1` Read from System register space. MRRC instruction.

This field resets to an architecturally UNKNOWN value.

The following fields describe configuration settings for generating exceptions that are reported using EC value `0b000100`:

- CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN}, for accesses to the Generic Timer Registers from EL0 using AArch32 state, MCRR or MRRC access (coproc == `0b1111`) trapped to EL1 or EL2.
- PMUSERENR_EL0.{CR, EN}, for accesses to Performance Monitor registers from EL0 using AArch32 state, MCRR or MRRC access (coproc == `0b1111`) trapped to EL1 or EL2.
- AMUSERENR_EL0.{EN}, for accesses to Activity Monitors registers AMEVCNTR0<n> and AMEVCNTR1<n> from EL0 using AArch32 state, MCRR or MRRC access (coproc == `0b1111`) trapped to EL1 or EL2.
- HCR_EL2.{TRVM, TVM}, for accesses to virtual memory control registers from EL1 using AArch32 state, MCRR or MRRC access (coproc == `0b1111`) trapped to EL2.
- HSTR_EL2.T<n>, for accesses to System registers using AArch32 state, MCRR or MRRC access (coproc == `0b1111`) trapped to EL2.
- CNTHCTL_EL2.{EL1PCEN, EL1PCTEN}, for accesses to the Generic Timer registers from EL0 and EL1 using AArch32 state, MCRR or MRRC access (coproc == `0b1111`) trapped to EL2.
- MDCR_EL2.{TPM, TPMCR}, for accesses to Performance Monitor registers from EL0 and EL1 using AArch32 state, MCRR or MRRC access (coproc == `0b1111`) trapped to EL2.
- CPTR_EL2.TAM, for accesses to Activity Monitors registers AMEVCNTR0<n> and AMEVCNTR1<n> from EL0 and EL1 using AArch32 state, MCRR or MRRC access (coproc == `0b1111`) trapped to EL2.
- MDCR_EL3.TPM, for accesses to Performance Monitor registers from EL0, EL1 and EL2 using AArch32 state, MCRR or MRRC access (coproc == `0b1111`) trapped to EL3.
- CPTR_EL3.TAM, for accesses to Activity Monitors registers from EL0, EL1 and EL2 using AArch32 state, MCRR or MRRC access (coproc == `0b1111`) trapped to EL3.

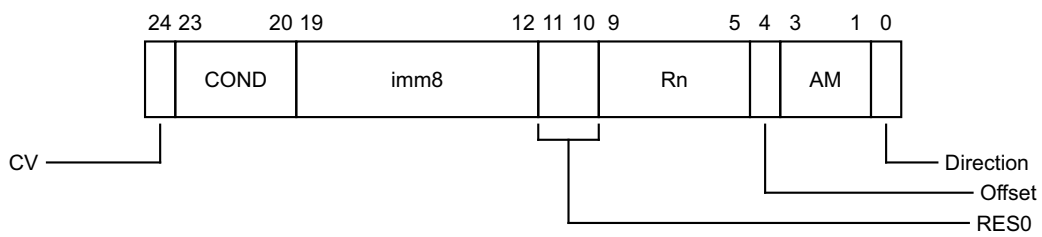
Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If ARMv8.6-FGT is implemented, HDFGRTR_EL2.PMCCNTR_EL0 for MRRC access and HDFGWTR_EL2.PMCCNTR_EL0 for MCRR access to PMCCNTR at EL0, trapped to EL2.

The following sections describe configuration settings for generating exceptions that are reported using EC value `0b001100`:

- CPACR_EL1.TTA for accesses to trace registers using MCR or MRC instructions, MCRR or MRRC access (coproc == `0b1110`) trapped to EL1 or EL2.
- MDSCR_EL1.TDCC, for accesses to the Debug Communications Channel (DCC) registers DBGDSAR and DBGDRAR at EL0 using AArch32 state, MCRR or MRRC access (coproc == `0b1110`) trapped to EL1 or EL2.
- CPTR_EL2.TTA, for accesses to trace registers using AArch32, MCRR or MRRC access (coproc == `0b1110`) trapped to EL2.
- MDCR_EL2.TDRA, for accesses to Debug ROM registers DBGDRAR and AArch-DBGDSAR using AArch32, MCRR or MRRC access (coproc == `0b1110`) trapped to EL2.
- CPTR_EL3.TTA, for accesses to trace registers using AArch32, MCRR or MRRC access (coproc == `0b1110`) trapped to EL3.
- MDCR_EL3.TDOSA, for traps to powerdown debug registers using AArch32, MCRR or MRRC access (coproc == `0b1110`) trapped to EL3.
- MDCR_EL3.TDA, for accesses to other debug registers, using AArch32, MCRR or MRRC access (coproc == `0b1110`) trapped to EL3.

ISS encoding for an exception from an LDC or STC instruction



CV, bit [24]

Condition code valid. Possible values of this bit are:

`0b0` The COND field is not valid.

`0b1` The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to `0b1110`.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to `0b1110`.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to `0b1110`, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

This field resets to an architecturally UNKNOWN value.

imm8, bits [19:12]

The immediate value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Bits [11:10]

Reserved, RES0.

Rn, bits [9:5]

The Rn value from the issued instruction, the general-purpose register used for the transfer. The reported value gives the AArch64 view of the register. See *Mapping of the general-purpose registers between the Execution states* on page D1-2407.

This field is valid only when AM[2] is 0, indicating an immediate form of the LDC or STC instruction. When AM[2] is 1, indicating a literal form of the LDC or STC instruction, this field is UNKNOWN.

This field resets to an architecturally UNKNOWN value.

Offset, bit [4]

Indicates whether the offset is added or subtracted:

`0b0` Subtract offset.

`0b1` Add offset.

This bit corresponds to the U bit in the instruction encoding.

This field resets to an architecturally UNKNOWN value.

AM, bits [3:1]

Addressing mode. The permitted values of this field are:

`0b000` Immediate unindexed.

`0b001` Immediate post-indexed.

`0b010` Immediate offset.

`0b011` Immediate pre-indexed.

`0b100` For a trapped STC instruction or a trapped T32 LDC instruction this encoding is reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b110` For a trapped STC instruction, this encoding is reserved.

The values `0b101` and `0b111` are reserved. The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-7871.

Bit [2] in this subfield indicates the instruction form, immediate or literal.

Bits [1:0] in this subfield correspond to the bits {P, W} in the instruction encoding.

This field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction. The possible values of this bit are:

`0b0` Write to memory. STC instruction.

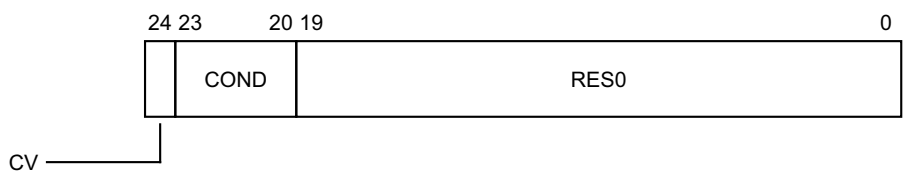
`0b1` Read from memory. LDC instruction.

This field resets to an architecturally UNKNOWN value.

The following fields describe the configuration settings for the traps that are reported using EC value `0b000110`:

- MDSCR_EL1.TDCC, for accesses using AArch32 state, LDC access to DBGDTRTXint or STC access to DBGDTRRXint trapped to EL1 or EL2.
- MDCR_EL2.TDA, for accesses using AArch32 state, LDC access to DBGDTRTXint or STC access to DBGDTRRXint MCR or MRC access trapped to EL2.
- MDCR_EL3.TDA, for accesses using AArch32 state, LDC access to DBGDTRTXint or STC access to DBGDTRRXint MCR or MRC access trapped to EL3.
- If ARMv8.6-FGT is implemented, MDCR_EL2.TDCC and MDCR_EL3.TDCC, for accesses to the DCC registers at EL0 and EL1, trapped to EL2.

ISS encoding for an exception from an access to SVE, Advanced SIMD or floating-point functionality, resulting from CPACR_EL1.FPEN, CPTR_EL2.FPEN or CPTR_ELx.TFP



The accesses covered by this trap include:

- Execution of SVE or Advanced SIMD and floating-point instructions.
- Accesses to the Advanced SIMD and floating-point System registers.

For an implementation that does not include either SVE or support for floating-point and Advanced SIMD, the exception is reported using the EC value `0b000000`.

CV, bit [24]

Condition code valid. Possible values of this bit are:

`0b0` The COND field is not valid.

`0b1` The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to `0b1110`.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to `0b1110`.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to `0b1110`, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

This field resets to an architecturally UNKNOWN value.

Bits [19:0]

Reserved, RES0.

The following sections describe the configuration settings for the traps that are reported using EC value `0b000111`:

- CPACR_EL1.FPEN, for accesses to SIMD and floating-point registers trapped to EL1.
- CPTR_EL2.TFP, for accesses to SIMD and floating-point registers trapped to EL2.
- CPTR_EL2.TFP, for accesses to SIMD and floating-point registers trapped to EL3.

ISS encoding for an exception from an access to SVE functionality, resulting from CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ

24 … 0
RES0

Bits [24:0]

When SVE is implemented:

Reserved, RES0.

The accesses covered by this trap include:

- Execution of SVE instructions.
- Accesses to the SVE system registers, ZCR_ELx and ID_AA64ZFR0_EL1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For an implementation that does not include SVE, the exception is reported using the EC value `0b000000`.

ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault

24 … 0
RES0

Bits [24:0]

Reserved, RES0.

There are no configuration settings for generating Illegal Execution state exceptions and PC alignment fault exceptions. For more information about these exceptions see *The Illegal Execution state exception* on page D1-2353 and *PC alignment checking* on page D1-2333.

SP alignment checking on page D1-2333 describes the configuration settings for generating SP alignment fault exceptions.

ISS encoding for an exception from HVC or SVC instruction execution

24 … 16	15 … 0
RES0	imm16

Bits [24:16]

Reserved, RES0.

imm16, bits [15:0]

The value of the immediate field from the HVC or SVC instruction.

For an HVC instruction, and for an A64 SVC instruction, this is the value of the imm16 field of the issued instruction.

For an A32 or T32 SVC instruction:

- If the instruction is unconditional, then:
  - For the T32 instruction, this field is zero-extended from the imm8 field of the instruction.
  - For the A32 instruction, this field is the bottom 16 bits of the imm24 field of the instruction.
- If the instruction is conditional, this field is UNKNOWN.

This field resets to an architecturally UNKNOWN value.

In AArch32 state, the HVC instruction is unconditional, and a conditional SVC instruction generates an exception only if it passes its condition code check. Therefore, the syndrome information for these exceptions does not require conditionality information.

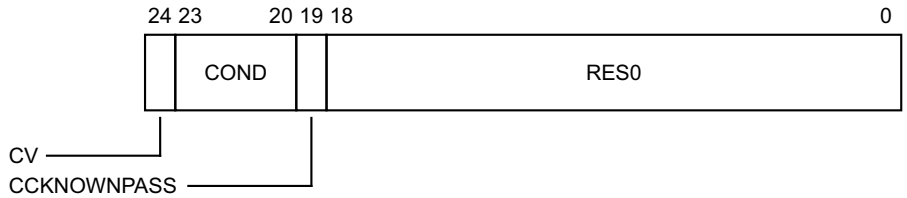
For T32 and A32 instructions, see SVC and HVC.

For A64 instructions, see SVC and HVC.

If ARMv8.6-FGT is implemented, HFGITR_EL2.{SVC_EL1, SVC_EL0} control fine-grained traps on SVC execution.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ISS encoding for an exception from SMC instruction execution in AArch32 state



For an SMC instruction that completes normally and generates an exception that is taken to EL3, the ISS encoding is RES0.

For an SMC instruction that is trapped to EL2 from EL1 because HCR_EL2.TSC is 1, the ISS encoding is as shown in the diagram.

CV, bit [24]

Condition code valid. Possible values of this bit are:

`0b0` The COND field is not valid.

`0b1` The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

This field is only valid if CCKNOWNPASS is 1, otherwise it is RES0.

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to `0b1110`.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - — If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - — If the instruction is unconditional, COND is set to `0b1110`.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - — With COND set to `0b1110`, the value for unconditional.
  - — With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - — CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - — CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

This field is only valid if CCKNOWNPASS is 1, otherwise it is RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

CCKNOWNPASS, bit [19]

Indicates whether the instruction might have failed its condition code check.

`0b0` The instruction was unconditional, or was conditional and passed its condition code check.

`0b1` The instruction was conditional, and might have failed its condition code check.

Note

In an implementation in which an SMC instruction that fails it code check is not trapped, this field can always return the value 0.

This field resets to an architecturally UNKNOWN value.

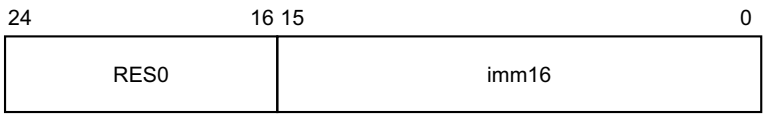
Bits [18:0]

Reserved, RES0.

HCR_EL2.TSC describes the configuration settings for trapping SMC instructions to EL2.

See *System calls* on page D1-2396 describes the case where these exceptions are trapped to EL3.

ISS encoding for an exception from SMC instruction execution in AArch64 state



Bits [24:16]

Reserved, RES0.

imm16, bits [15:0]

The value of the immediate field from the issued SMC instruction.

This field resets to an architecturally UNKNOWN value.

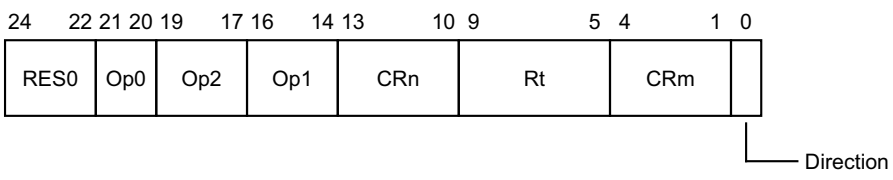
The value of ISS[24:0] described here is used both:

- When an SMC instruction is trapped from EL1 modes.
- When an SMC instruction is not trapped, so completes normally and generates an exception that is taken to EL3.

HCR_EL2.TSC describes the configuration settings for trapping SMC from EL1 modes.

System calls on page D1-2396 describes the case where these exceptions are trapped to EL3.

ISS encoding for an exception from MSR, MRS, or System instruction execution in AArch64 state



Bits [24:22]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Op0, bits [21:20]

The Op0 value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Op2, bits [19:17]

The Op2 value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Op1, bits [16:14]

The Op1 value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

CRn, bits [13:10]

The CRn value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Rt, bits [9:5]

The Rt value from the issued instruction, the general-purpose register used for the transfer.

This field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction. The possible values of this bit are:

`0b0` Write access, including MSR instructions.

`0b1` Read access, including MRS instructions.

This field resets to an architecturally UNKNOWN value.

For exceptions caused by System instructions, see System instructions for the encoding values returned by an instruction.

The following fields describe configuration settings for generating the exception that is reported using EC value `0b011000`:

- SCTLR_EL1.UCI, for execution of cache maintenance instructions using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- SCTLR_EL1.UCT, for accesses to CTR_EL0 using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- SCTLR_EL1.DZE, for execution of DC ZVA instructions using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- SCTLR_EL1.UMA, for accesses to the PSTATE interrupt masks using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- CPACR_EL1.TTA, for accesses to the trace registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- MDSCR_EL1.TDCC, for accesses to the Debug Communications Channel (DCC) registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- If ARMv8.6-FGT is implemented, MDCR_EL2.TDCC and MDCR_EL3.TDCC, for accesses to the DCC registers at EL0 and EL1, trapped to EL2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN} accesses to the Generic Timer registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- PMUSERENR_EL0.{ER, CR, SW, EN}, for accesses to the Performance Monitor registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- AMUSERENR_EL0.EN, for accesses to Activity Monitors registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- HCR_EL2.{TRVM, TVM}, for accesses to virtual memory control registers using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.TDZ, for execution of DC ZVA instructions using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.TTLB, for execution of TLB maintenance instructions using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.{TSW, TPC, TPU}, for execution of cache maintenance instructions using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.TACR, for accesses to the Auxiliary Control Register, ACTLR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.TIDCP, for accesses to lockdown, DMA, and TCM operations using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.{TID1, TID2, TID3}, for accesses to ID group 1, ID group 2 or ID group 3 registers, using AArch64 state, MSR or MRS access trapped to EL2.
- CPTR_EL2.TCPAC, for accesses to CPACR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.
- CPTR_EL2.TTA, for accesses to the trace registers, using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TTRF, for accesses to the trace filter register, TRFCR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TDRA, for accesses to Debug ROM registers, using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TDOSA, for accesses to powerdown debug registers using AArch64 state, MSR or MRS access trapped to EL2.
- CNTHCTL_EL2.{EL1PCEN, EL1PCTEN}, for accesses to the Generic Timer registers using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TDA, for accesses to debug registers using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.{TPM, TPMCR}, for accesses to Performance Monitor registers, using AArch64 state, MSR or MRS access trapped to EL2.
- CPTR_EL2.TAM, for accesses to Activity Monitors registers, using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.APK, for accesses to Pointer authentication key registers. using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.{NV, NV1}, for Nested virtualization register access, using AArch64 state, MSR or MRS access, trapped to EL2.
- HCR_EL2.AT, for execution of AT S1E* instructions, using AArch64 state, MSR or MRS access, trapped to EL2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- HCR_EL2.{TERR, FIEN}, for accesses to RAS registers, using AArch64 state, MSR or MRS access, trapped to EL2.
- SCR_EL3.APK, for accesses to Pointer authentication key registers, using AArch64 state, MSR or MRS access trapped to EL3.
- SCR_EL3.ST, for accesses to the Counter-timer Physical Secure timer registers, using AArch64 state, MSR or MRS access trapped to EL3.
- SCR_EL3.{TERR, FIEN}, for accesses to RAS registers, using AArch64 state, MSR or MRS access trapped to EL3.
- CPTR_EL3.TCPAC, for accesses to CPTR_EL2 and CPACR_EL1 using AArch64 state, MSR or MRS access trapped to EL3.
- CPTR_EL3.TTA, for accesses to the trace registers, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TTRF, for accesses to the filter trace control registers, TRFCR_EL1 and TRFCR_EL2, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TDA, for accesses to debug registers, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TDOSA, for accesses to powerdown debug registers, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TPM, for accesses to Performance Monitor registers, using AArch64 state, MSR or MRS access trapped to EL3.
- CPTR_EL3.TAM, for accesses to Activity Monitors registers, using AArch64 state, MSR or MRS access, trapped to EL3.
- If ARMv8.2-EVT is implemented the following registers control traps for EL1 and EL0 Cache controls that use this EC value:
  - HCR_EL2.{TTLBOS, TTLBIS, TICAB, TOCU, TID4}.
  - HCR2.{TTLBIS, TICAB, TOCU, TID4}.
- If ARMv8.6-FGT is implemented:
  - SCR_EL3.FGTEn, for accesses to the fine-grained trap registers, MSR or MRS access at EL2 trapped to EL3.
  - HFGRTR_EL2 for reads and HFGWTR_EL2 for writes of registers, using AArch64 state, MSR or MRS access at EL0 and EL1 trapped to EL2.
  - HFGITR_EL2 for execution of system instructions, MSR or MRS access trapped to EL2
  - HDFGRTR_EL2 for reads and HDFGWTR_EL2 for writes of registers, using AArch64 state, MSR or MRS access at EL0 and EL1 state trapped to EL2.
  - HAFGRTR_EL2 for reads of Activity Monitor counters, using AArch64 state, MRS access at EL0 and EL1 trapped to EL2.

ISS encoding for an IMPLEMENTATION DEFINED exception to EL3

24 … 0
IMPLEMENTATION DEFINED

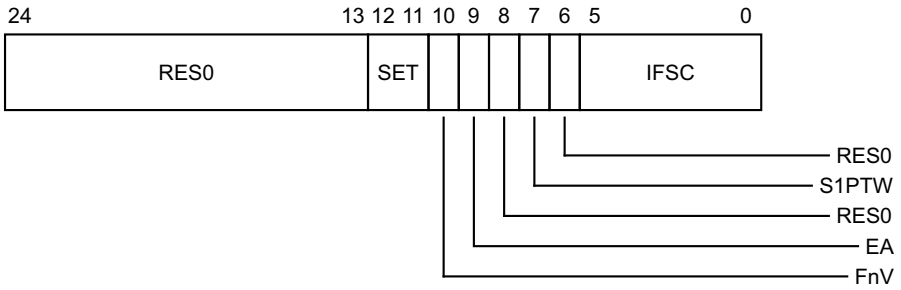
IMPLEMENTATION DEFINED, bits [24:0]

IMPLEMENTATION DEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

ISS encoding for an exception from an Instruction Abort



Bits [24:13]

Reserved, RES0.

SET, bits [12:11]

Synchronous Error Type. When the RAS Extension is implemented and IFSC is `0b010000`, describes the state of the PE after taking the Instruction Abort exception. The possible values of this field are:

`0b00` Recoverable error (UER).

`0b10` Uncontainable error (UC).

`0b11` Restartable error (UEO) or Corrected error (CE).

All other values are reserved.

Note

Software can use this information to determine what recovery might be possible. Taking a synchronous External Abort exception might result in an unrecoverable PE state.

This field is RES0 if either:

- The RAS Extension is not implemented.
- The value returned in the IFSC field is not `0b010000`.

This field resets to an architecturally UNKNOWN value.

FnV, bit [10]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

`0b0` FAR is valid.

`0b1` FAR is not valid, and holds an UNKNOWN value.

This field is only valid if the IFSC code is `0b010000`. It is RES0 for all other aborts.

This field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

This field resets to an architecturally UNKNOWN value.

Bit [8]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

S1PTW, bit [7]

For a stage 2 fault, indicates whether the fault was a stage 2 fault on an access made for a stage 1 translation table walk:

`0b0` Fault not on a stage 2 translation for a stage 1 translation table walk.

`0b1` Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

This field resets to an architecturally UNKNOWN value.

Bit [6]

Reserved, RES0.

IFSC, bits [5:0]

Instruction Fault Status Code.

`0b000000` Address size fault, level 0 of translation or translation table base register.

`0b000001` Address size fault, level 1.

`0b000010` Address size fault, level 2.

`0b000011` Address size fault, level 3.

`0b000100` Translation fault, level 0.

`0b000101` Translation fault, level 1.

`0b000110` Translation fault, level 2.

`0b000111` Translation fault, level 3.

`0b001001` Access flag fault, level 1.

`0b001010` Access flag fault, level 2.

`0b001011` Access flag fault, level 3.

`0b001101` Permission fault, level 1.

`0b001110` Permission fault, level 2.

`0b001111` Permission fault, level 3.

`0b010000` Synchronous External abort, not on translation table walk or hardware update of translation table.

`0b010100` Synchronous External abort, on translation table walk or hardware update of translation table, level 0.

`0b010101` Synchronous External abort, on translation table walk or hardware update of translation table, level 1.

`0b010110` Synchronous External abort, on translation table walk or hardware update of translation table, level 2.

`0b010111` Synchronous External abort, on translation table walk or hardware update of translation table, level 3.

`0b011000` **When RAS is not implemented** Synchronous parity or ECC error on memory access, not on translation table walk.

`0b011100` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 0.

`0b011101` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 1.

`0b011110` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 2.

`0b011111` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 3.

`0b110000` TLB conflict abort.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b110001` **When ARMv8.1-TTHM is implemented** Unsupported atomic hardware update fault.

All other values are reserved.

For more information about the lookup level associated with a fault, see *The level associated with MMU faults* on page D5-2658.

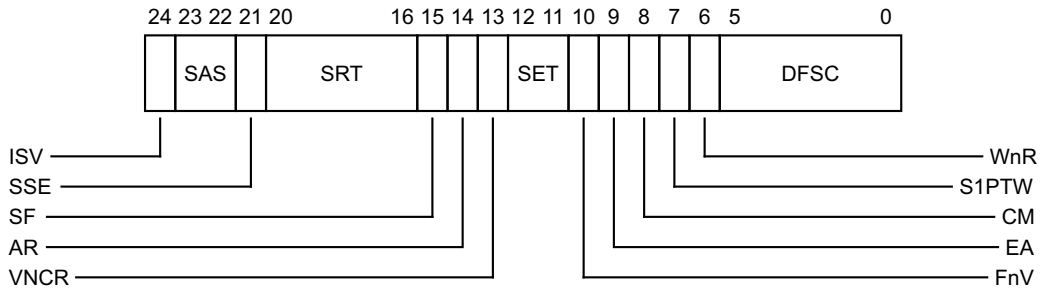
Note

Because Access flag faults and Permission faults can only result from a Block or Page translation table descriptor, they cannot occur at level 0.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.

This field resets to an architecturally UNKNOWN value.

ISS encoding for an exception from a Data Abort



ISV, bit [24]

Instruction Syndrome Valid. Indicates whether the syndrome information in ISS[23:14] is valid.

`0b0` No valid instruction syndrome. ISS[23:14] are RES0.

`0b1` ISS[23:14] hold a valid instruction syndrome.

This bit is 0 for all faults reported in ESR_EL2 except the following stage 2 aborts:

- AArch64 loads and stores of a single general-purpose register (including the register specified with `0b11111`, including those with Acquire/Release semantics, but excluding Load Exclusive or Store Exclusive and excluding those with writeback.
- AArch32 instructions where the instruction:
  - Is an LDR, LDA, LDRT, LDRSH, LDRSHT, LDRH, LDAH, LDRHT, LDRSB, LDRSBT, LDRB, LDAB, LDRBT, STR, STL, STRT, STRH, STLH, STRHT, STRB, STLB, or STRBT instruction.
  - Is not performing register writeback.
  - Is not using R15 as a source or destination register.

For these cases, ISV is UNKNOWN if the exception was generated in Debug state in memory access mode, and otherwise indicates whether ISS[23:14] hold a valid syndrome.

ISV is 0 for all faults reported in ESR_EL1 or ESR_EL3.

When the RAS Extension is implemented, ISV is 0 for any synchronous External abort.

For ISS reporting, a stage 2 abort on a stage 1 translation table walk does not return a valid instruction syndrome, and therefore ISV is 0 for these aborts.

When the RAS Extension is not implemented, the value of ISV on a synchronous External abort on a stage 2 translation table walk is IMPLEMENTATION DEFINED.

When ARMv8.5-MemTag is implemented, for a synchronous Tag Check Fault abort taken to ELx, ESR_ELx.FNV is 0 and FAR_ELx is valid.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

SAS, bits [23:22]

Syndrome Access Size. When ISV is 1, indicates the size of the access attempted by the faulting operation.

`0b00` Byte

`0b01` Halfword

`0b10` Word

`0b11` Doubleword

This field is UNKNOWN when the value of ISV is UNKNOWN.

This field is RES0 when the value of ISV is 0.

This field resets to an architecturally UNKNOWN value.

SSE, bit [21]

Syndrome Sign Extend. When ISV is 1, for a byte, halfword, or word load operation, indicates whether the data item must be sign extended. For these cases, the possible values of this bit are:

`0b0` Sign-extension not required.

`0b1` Data item must be sign-extended.

For all other operations this bit is 0.

This field is UNKNOWN when the value of ISV is UNKNOWN.

This field is RES0 when the value of ISV is 0.

This field resets to an architecturally UNKNOWN value.

SRT, bits [20:16]

Syndrome Register Transfer. When ISV is 1, the register number of the Rt operand of the faulting instruction.

If the exception was taken from an Exception level that is using AArch32 then this is the AArch64 view of the register. See *Mapping of the general-purpose registers between the Execution states* on page D1-2407.

This field is UNKNOWN when the value of ISV is UNKNOWN.

This field is RES0 when the value of ISV is 0.

This field resets to an architecturally UNKNOWN value.

SF, bit [15]

Width of the register accessed by the instruction is Sixty-Four. When ISV is 1, the possible values of this bit are:

`0b0` Instruction loads/stores a 32-bit wide register.

`0b1` Instruction loads/stores a 64-bit wide register.

Note

This field specifies the register width identified by the instruction, not the Execution state.

This field is UNKNOWN when the value of ISV is UNKNOWN.

This field is RES0 when the value of ISV is 0.

This field resets to an architecturally UNKNOWN value.

AR, bit [14]

Acquire/Release. When ISV is 1, the possible values of this bit are:

`0b0` Instruction did not have acquire/release semantics.

`0b1` Instruction did have acquire/release semantics.

This field is UNKNOWN when the value of ISV is UNKNOWN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field is RES0 when the value of ISV is 0.

This field resets to an architecturally UNKNOWN value.

VNCR, bit [13]

Indicates that the fault came from use of VNCR_EL2 register by EL1 code.

`0b0` The fault was not generated by the use of VNCR_EL2, by an MRS or MSR instruction executed at EL1.

`0b1` The fault was generated by the use of VNCR_EL2, by an MRS or MSR instruction executed at EL1.

This field is 0 in ESR_EL1.

This field resets to an architecturally UNKNOWN value.

SET, bits [12:11]

Synchronous Error Type. When the RAS Extension is implemented and DFSC is `0b010000`, describes the state of the PE after taking the Data Abort exception. The possible values of this field are:

`0b00` Recoverable error (UER).

`0b10` Uncontainable error (UC).

`0b11` Restartable error (UEO) or Corrected error (CE).

All other values are reserved.

Note

Software can use this information to determine what recovery might be possible. Taking a synchronous External Abort exception might result in an unrecoverable PE state.

This field is RES0 if either:

- The RAS Extension is not implemented.
- The value returned in the DFSC field is not `0b010000`.

This field resets to an architecturally UNKNOWN value.

FnV, bit [10]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

`0b0` FAR is valid.

`0b1` FAR is not valid, and holds an UNKNOWN value.

This field is valid only if the DFSC code is `0b010000`. It is RES0 for all other aborts.

This field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

This field resets to an architecturally UNKNOWN value.

CM, bit [8]

Cache maintenance. Indicates whether the Data Abort came from a cache maintenance or address translation instruction:

`0b0` The Data Abort was not generated by the execution of one of the System instructions identified in the description of value 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` The Data Abort was generated by either the execution of a cache maintenance instruction or by a synchronous fault on the execution of an address translation instruction. The DC ZVA instruction is not classified as a cache maintenance instruction, and therefore its execution cannot cause this field to be set to 1.

This field resets to an architecturally UNKNOWN value.

S1PTW, bit [7]

For a stage 2 fault, indicates whether the fault was a stage 2 fault on an access made for a stage 1 translation table walk:

`0b0` Fault not on a stage 2 translation for a stage 1 translation table walk.

`0b1` Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

This field resets to an architecturally UNKNOWN value.

WnR, bit [6]

Write not Read. Indicates whether a synchronous abort was caused by an instruction writing to a memory location, or by an instruction reading from a memory location. The possible values of this bit are:

`0b0` Abort caused by an instruction reading from a memory location.

`0b1` Abort caused by an instruction writing to a memory location.

For faults on cache maintenance and address translation instructions, this bit always returns a value of 1.

For faults from an atomic instruction that both reads and writes from a memory location, this bit is set to 0 if a read of the address specified by the instruction would have generated the fault which is being reported, otherwise it is set to 1. The architecture permits, but does not require, a relaxation of this requirement such that for all stage 2 aborts on stage 1 translation table walks for atomic instructions, the WnR bit is always 0.

This field is UNKNOWN for:

- An External abort on an Atomic access.
- A fault reported using a DFSC value of `0b110101` or `0b110001`, indicating an unsupported Exclusive or atomic access.

This field resets to an architecturally UNKNOWN value.

DFSC, bits [5:0]

Data Fault Status Code.

`0b000000` Address size fault, level 0 of translation or translation table base register.

`0b000001` Address size fault, level 1.

`0b000010` Address size fault, level 2.

`0b000011` Address size fault, level 3.

`0b000100` Translation fault, level 0.

`0b000101` Translation fault, level 1.

`0b000110` Translation fault, level 2.

`0b000111` Translation fault, level 3.

`0b001001` Access flag fault, level 1.

`0b001010` Access flag fault, level 2.

`0b001011` Access flag fault, level 3.

`0b001101` Permission fault, level 1.

`0b001110` Permission fault, level 2.

`0b001111` Permission fault, level 3.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b010000` Synchronous External abort, not on translation table walk or hardware update of translation table.

`0b010001` **When ARMv8.5-MemTag is implemented** Synchronous Tag Check Fault.

`0b010100` Synchronous External abort, on translation table walk or hardware update of translation table, level 0.

`0b010101` Synchronous External abort, on translation table walk or hardware update of translation table, level 1.

`0b010110` Synchronous External abort, on translation table walk or hardware update of translation table, level 2.

`0b010111` Synchronous External abort, on translation table walk or hardware update of translation table, level 3.

`0b011000` **When RAS is not implemented** Synchronous parity or ECC error on memory access, not on translation table walk.

`0b011100` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 0.

`0b011101` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 1.

`0b011110` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 2.

`0b011111` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 3.

`0b100001` Alignment fault.

`0b110000` TLB conflict abort.

`0b110001` **When ARMv8.1-TTHM is implemented** Unsupported atomic hardware update fault.

`0b110100` IMPLEMENTATION DEFINED fault (Lockdown).

`0b110101` IMPLEMENTATION DEFINED fault (Unsupported Exclusive or Atomic access).

All other values are reserved.

For more information about the lookup level associated with a fault, see *The level associated with MMU faults* on page D5-2658.

Note

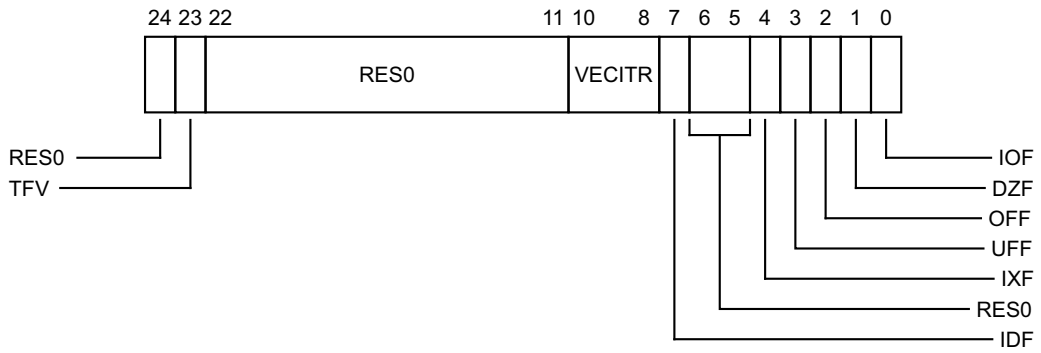
Because Access flag faults and Permission faults can only result from a Block or Page translation table descriptor, they cannot occur at level 0.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ISS encoding for an exception from a trapped floating-point exception



Bit [24]

Reserved, RES0.

TFV, bit [23]

Trapped Fault Valid bit. Indicates whether the IDF, IXF, UFF, OFF, DZF, and IOF bits hold valid information about trapped floating-point exceptions. The possible values of this bit are:

`0b0` The IDF, IXF, UFF, OFF, DZF, and IOF bits do not hold valid information about trapped floating-point exceptions and are UNKNOWN.

`0b1` One or more floating-point exceptions occurred during an operation performed while executing the reported instruction. The IDF, IXF, UFF, OFF, DZF, and IOF bits indicate trapped floating-point exceptions that occurred. For more information see *Floating-point exceptions and exception traps* on page D1-2360.

It is IMPLEMENTATION DEFINED whether this field is set to 0 on an exception generated by a trapped floating point exception from a vector instruction.

Note

This is not a requirement. Implementations can set this field to 1 on a trapped floating-point exception from a vector instruction and return valid information in the {IDF, IXF, UFF, OFF, DZF, IOF} fields.

This field resets to an architecturally UNKNOWN value.

Bits [22:11]

Reserved, RES0.

VECITR, bits [10:8]

For a trapped floating-point exception from an instruction executed in AArch32 state this field is RES1.

For a trapped floating-point exception from an instruction executed in AArch64 state this field is UNKNOWN.

This field resets to an architecturally UNKNOWN value.

IDF, bit [7]

Input Denormal floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

`0b0` Input denormal floating-point exception has not occurred.

`0b1` Input denormal floating-point exception occurred during execution of the reported instruction.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bits [6:5]

Reserved, RES0.

IXF, bit [4]

Inexact floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

`0b0` Inexact floating-point exception has not occurred.

`0b1` Inexact floating-point exception occurred during execution of the reported instruction.

This field resets to an architecturally UNKNOWN value.

UFF, bit [3]

Underflow floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

`0b0` Underflow floating-point exception has not occurred.

`0b1` Underflow floating-point exception occurred during execution of the reported instruction.

This field resets to an architecturally UNKNOWN value.

OFF, bit [2]

Overflow floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

`0b0` Overflow floating-point exception has not occurred.

`0b1` Overflow floating-point exception occurred during execution of the reported instruction.

This field resets to an architecturally UNKNOWN value.

DZF, bit [1]

Divide by Zero floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

`0b0` Divide by Zero floating-point exception has not occurred.

`0b1` Divide by Zero floating-point exception occurred during execution of the reported instruction.

This field resets to an architecturally UNKNOWN value.

IOF, bit [0]

Invalid Operation floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

`0b0` Invalid Operation floating-point exception has not occurred.

`0b1` Invalid Operation floating-point exception occurred during execution of the reported instruction.

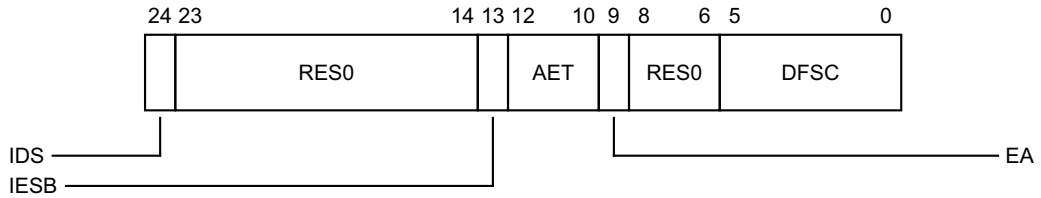
This field resets to an architecturally UNKNOWN value.

In an implementation that supports the trapping of floating-point exceptions:

- From an Exception level using AArch64, the FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} bits enable each of the floating-point exception traps.
- From an Exception level using AArch32, the FPSCR.{IDE, IXE, UFE, OFE, DZE, IOE} bits enable each of the floating-point exception traps.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ISS encoding for an SError interrupt



IDS, bit [24]

IMPLEMENTATION DEFINED syndrome. Possible values of this bit are:

`0b0` Bits[23:0] of the ISS field holds the fields described in this encoding.

Note

If the RAS Extension is not implemented, this means that bits[23:0] of the ISS field are RES0.

`0b1` Bits[23:0] of the ISS field holds IMPLEMENTATION DEFINED syndrome information that can be used to provide additional information about the SError interrupt.

Note

This field was previously called ISV.

This field resets to an architecturally UNKNOWN value.

Bits [23:14]

Reserved, RES0.

IESB, bit [13]

Implicit error synchronization event.

`0b0` The SError interrupt was either not synchronized by the implicit error synchronization event or not taken immediately.

`0b1` The SError interrupt was synchronized by the implicit error synchronization event and taken immediately.

This field is RES0 if the value returned in the DFSC field is not `0b010001`.

This field resets to an architecturally UNKNOWN value.

AET, bits [12:10]

Asynchronous Error Type.

When the RAS Extension is implemented and DFSC is `0b010001`, describes the state of the PE after taking the SError interrupt exception. The possible values of this field are:

`0b000` Uncontainable error (UC).

`0b001` Unrecoverable error (UEU).

`0b010` Restartable error (UEO).

`0b011` Recoverable error (UER).

`0b110` Corrected error (CE).

All other values are reserved.

If multiple errors are taken as a single SError interrupt exception, the overall state of the PE is reported. For example, if both a Recoverable and Unrecoverable error occurred, the state is Unrecoverable.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

Software can use this information to determine what recovery might be possible. The recovery software must also examine any implemented fault records to determine the location and extent of the error.

This field is RES0 if either:

- The RAS Extension is not implemented.
- The value returned in the DFSC field is not `0b010001`.

This field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort type. When the RAS Extension is implemented, this bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

This field is RES0 if either:

- The RAS Extension is not implemented.
- The value returned in the DFSC field is not `0b010001`.

This field resets to an architecturally UNKNOWN value.

Bits [8:6]

Reserved, RES0.

DFSC, bits [5:0]

When RAS is implemented:

Data Fault Status Code.

`0b000000` Uncategorized.

`0b010001` Asynchronous SError interrupt.

All other values are reserved.

This field resets to an architecturally UNKNOWN value.

ISS encoding for an exception from a Breakpoint or Vector Catch debug exception

24 … 6	5 … 0
RES0	IFSC

Bits [24:6]

Reserved, RES0.

IFSC, bits [5:0]

Instruction Fault Status Code.

`0b100010` Debug exception.

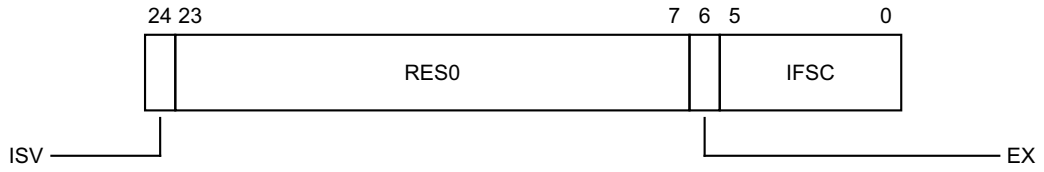
This field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions:

- For exceptions from AArch64, see *Breakpoint exceptions* on page D2-2439.
- For exceptions from AArch32, see *Breakpoint exceptions* on page G2-5830 and *Vector Catch exceptions* on page G2-5869.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ISS encoding for an exception from a Software Step exception



ISV, bit [24]

Instruction syndrome valid. Indicates whether the EX bit, ISS[6], is valid, as follows:

`0b0` EX bit is RES0.

`0b1` EX bit is valid.

See the EX bit description for more information.

This field resets to an architecturally UNKNOWN value.

Bits [23:7]

Reserved, RES0.

EX, bit [6]

Exclusive operation. If the ISV bit is set to 1, this bit indicates whether a Load-Exclusive instruction was stepped.

`0b0` An instruction other than a Load-Exclusive instruction was stepped.

`0b1` A Load-Exclusive instruction was stepped.

If the ISV bit is set to 0, this bit is RES0, indicating no syndrome data is available.

This field resets to an architecturally UNKNOWN value.

IFSC, bits [5:0]

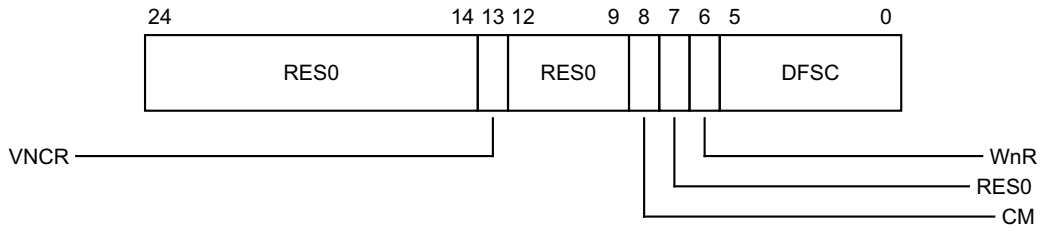
Instruction Fault Status Code.

`0b100010` Debug exception.

This field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions, see *Software Step exceptions* on page D2-2472.

ISS encoding for an exception from a Watchpoint exception



Bits [24:14]

Reserved, RES0.

VNCR, bit [13]

Indicates that the watchpoint came from use of VNCR_EL2 register by EL1 code.

`0b0` The watchpoint was not generated by the use of VNCR_EL2 by EL1 code.

`0b1` The watchpoint was generated by the use of VNCR_EL2 by EL1 code.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field is 0 in ESR_EL1.

This field resets to an architecturally UNKNOWN value.

Bits [12:9]

Reserved, RES0.

CM, bit [8]

Cache maintenance. Indicates whether the Watchpoint exception came from a cache maintenance or address translation instruction:

`0b0` The Watchpoint exception was not generated by the execution of one of the System instructions identified in the description of value 1.

`0b1` The Watchpoint exception was generated by either the execution of a cache maintenance instruction or by a synchronous Watchpoint exception on the execution of an address translation instruction. The DC ZVA instruction is not classified as a cache maintenance instruction, and therefore its execution cannot cause this field to be set to 1.

This field resets to an architecturally UNKNOWN value.

Bit [7]

Reserved, RES0.

WnR, bit [6]

Write not Read. Indicates whether the Watchpoint exception was caused by an instruction writing to a memory location, or by an instruction reading from a memory location. The possible values of this bit are:

`0b0` Watchpoint exception caused by an instruction reading from a memory location.

`0b1` Watchpoint exception caused by an instruction writing to a memory location.

For Watchpoint exceptions on cache maintenance and address translation instructions, this bit always returns a value of 1.

For Watchpoint exceptions from an atomic instruction, this field is set to 0 if a read of the location would have generated the Watchpoint exception, otherwise it is set to 1.

If multiple watchpoints match on the same access, it is UNPREDICTABLE which watchpoint generates the Watchpoint exception.

This field resets to an architecturally UNKNOWN value.

DFSC, bits [5:0]

Data Fault Status Code.

`0b100010` Debug exception.

This field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions, see *Watchpoint exceptions* on page D2-2457.

ISS encoding for an exception from execution of a Breakpoint instruction

24 … 16	15 … 0
RES0	Comment

Bits [24:16]

Reserved, RES0.

Comment, bits [15:0]

Set to the instruction comment field value, zero extended as necessary.

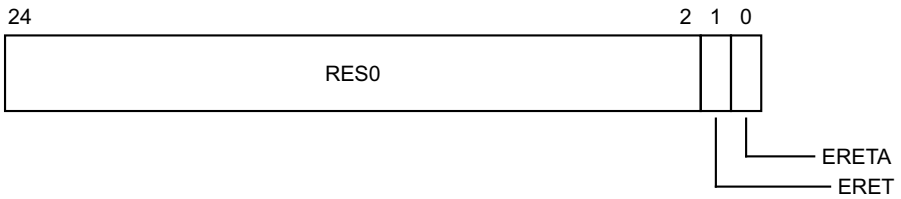
Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the AArch32 BKPT instructions, the comment field is described as the immediate field.

This field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions, see *Breakpoint Instruction exceptions* on page D2-2437.

ISS encoding for an exception from ERET, ERETAA or ERETAB instruction



This EC value applies when ARMv8.6-FGT is implemented, or when HCR_EL2.NV is 1.

Bits [24:2]

Reserved, RES0.

ERET, bit [1]

Indicates whether an ERET or ERETA* instruction was trapped to EL2. Possible values are:

`0b0` ERET instruction trapped to EL2.

`0b1` ERETAA or ERETAB instruction trapped to EL2.

If this bit is 0, the ERETA field is RES0.

This field resets to an architecturally UNKNOWN value.

ERETA, bit [0]

Indicates whether an ERETAA or ERETAB instruction was trapped to EL2. Possible values are:

`0b0` ERETAA instruction trapped to EL2.

`0b1` ERETAB instruction trapped to EL2.

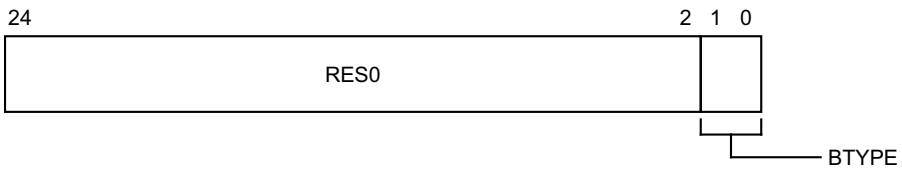
When the ERET field is 0, this bit is RES0.

This field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions, see HCR_EL2.NV.

If ARMv8.6-FGT is implemented, HFGITR_EL2.ERET controls fine-grained trap exceptions from ERET, ERETAA and ERETAB execution.

ISS encoding for an exception from Branch Target Identification instruction



Bits [24:2]

Reserved, RES0.

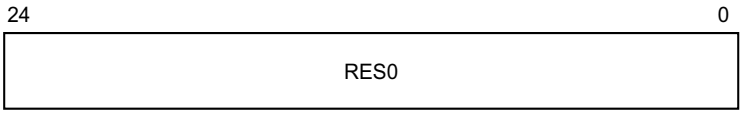
BTYPE, bits [1:0]

This field is set to the PSTATE.BTYPE value that generated the Branch Target Exception.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For more information about generating these exceptions, see Chapter B1 *The AArch64 Application Level Programmers' Model*.

ISS encoding for an exception from a Pointer Authentication instruction when HCR_EL2.API == 0 || SCR_EL3.API == 0



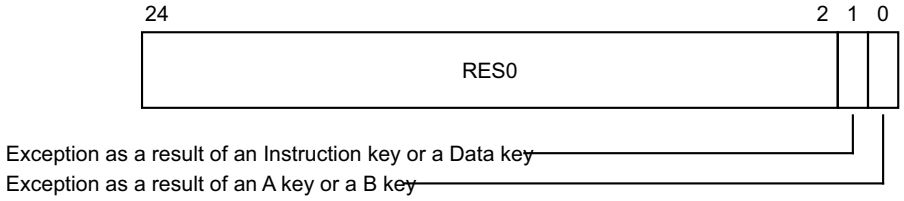
Bits [24:0]

Reserved, RES0.

For more information about generating these exceptions, see:

- HCR_EL2.API, for exceptions from Pointer authentication instructions, using AArch64 state, trapped to EL2.
- SCR_EL3.API, for exceptions from Pointer authentication instructions, using AArch64 state, trapped to EL3.

ISS encoding for an exception from a Pointer Authentication instruction authentication failure



Bits [24:2]

Reserved, RES0.

Bit [1]

This field indicates whether the exception is as a result of an Instruction key or a Data key.

`0b0` Instruction Key.

`0b1` Data Key.

This field resets to an architecturally UNKNOWN value.

Bit [0]

This field indicates whether the exception is as a result of an A key or a B key.

`0b0` A key.

`0b1` B key.

This field resets to an architecturally UNKNOWN value.

The following instructions generate an exception when the Pointer Authentication Code (PAC) is incorrect:

- `AUTIASP`, `AUTIAZ`, `AUTIA1716`.
- `AUTIBSP`, `AUTIBZ`, `AUTIB1716`.
- `AUTIA`, `AUTDA`, `AUTIB`, `AUTDB`.
- `AUTIZA`, `AUTIZB`, `AUTDZA`, `AUTDZB`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

It is IMPLEMENTATION DEFINED whether the following instructions generate an exception directly from the authorization failure, rather than changing the address in a way that will generate a translation fault when the address is accessed:

- RETAA, RETAB.
- BRAA, BRAB, BLRAA, BLRAB.
- BRAAZ, BRABZ, BLRAAZ, BLRABZ.
- ERETAA, ERETAB.
- LDRAA, LDRAB, whether the authenticated address is written back to the base register or not.

Accessing the ESR_EL3

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ESR_EL3

op0	op1	CRn	CRm	op2
0b11	0b110	0b0101	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    return ESR_EL3;
```

MSR ESR_EL3, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b110	0b0101	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    ESR_EL3 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.39 FAR_EL1, Fault Address Register (EL1)

The FAR_EL1 characteristics are:

Purpose

Holds the faulting Virtual Address for all synchronous Instruction or Data Abort, PC alignment fault and Watchpoint exceptions that are taken to EL1.

Configurations

AArch64 System register FAR_EL1[31:0] is architecturally mapped to AArch32 System register DFAR[31:0] (NS).

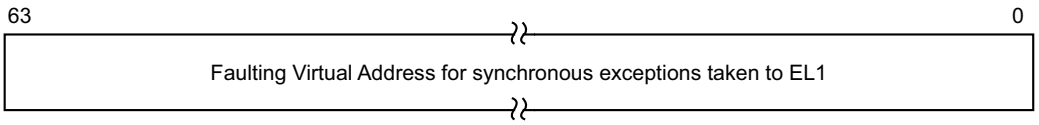
AArch64 System register FAR_EL1[63:32] is architecturally mapped to AArch32 System register IFAR[31:0] (NS).

Attributes

FAR_EL1 is a 64-bit register.

Field descriptions

The FAR_EL1 bit assignments are:



Bits [63:0]

Faulting Virtual Address for synchronous exceptions taken to EL1. Exceptions that set the FAR_EL1 are Instruction Aborts (EC `0x20` or `0x21`), Data Aborts (EC `0x24` or `0x25`), PC alignment faults (EC `0x22`), and Watchpoints (EC `0x34` or `0x35`). ESR_EL1.EC holds the EC value for the exception.

For a synchronous External abort, if the VA that generated the abort was from an address range for which TCR_ELx.TBI{<0|1>} == 1 for the translation regime in use when the abort was generated, then the top eight bits of FAR_EL1 are UNKNOWN.

For a synchronous External abort other than a synchronous External abort on a translation table walk, this field is valid only if ESR_EL1.FnV is 0, and the FAR_EL1 is UNKNOWN if ESR_EL1.FnV is 1.

For all other exceptions taken to EL1, the FAR_EL1 is UNKNOWN.

If a memory fault that sets FAR_EL1 is generated from a data cache maintenance or other DC instruction, this field holds the address specified in the register argument of the instruction.

If the exception that updates FAR_EL1 is taken from an Exception level that is using AArch32, the top 32 bits are all zero, unless both of the following apply, in which case the top 32 bits of FAR_ELx are `0x00000001`:

- The faulting address was generated by a load or store instruction that sequentially incremented from address `0xFFFFFFFF`. Such a load or store is CONSTRAINED UNPREDICTABLE. See *Out of range VA* on page K1-7840.
- The implementation treats such incrementing as setting bit[32] of the virtual address to 1.

For a Data Abort or Watchpoint exception, if address tagging is enabled for the address accessed by the data access that caused the exception, then this field includes the tag. For more information about address tagging, see *Address tagging in AArch64 state* on page D5-2534.

For a synchronous Tag Check Fault abort, bits[63:60] are UNKNOWN.

Execution at EL0 makes FAR_EL1 become UNKNOWN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

The address held in this field is an address accessed by the instruction fetch or data access that caused the exception that gave rise to the instruction or data abort. It is the lower address that gave rise to the fault. Where different faults from different addresses arise from the same instruction, such as for an instruction that loads or stores a mis-aligned address that crosses a page boundary, the architecture does not prioritize between those different faults.

FAR_EL1 is made UNKNOWN on an exception return from EL1.

This field resets to an architecturally UNKNOWN value.

Accessing the FAR_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic FAR_EL1 or FAR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, FAR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.FAR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x220];
    else
        return FAR_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return FAR_EL2;
    else
        return FAR_EL1;
elsif PSTATE.EL == EL3 then
    return FAR_EL1;
```

MSR FAR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.FAR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x220] = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        FAR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        FAR_EL2 = X[t];
    else
        FAR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    FAR_EL1 = X[t];
```

MRS <Xt>, FAR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        return NVMem[0x220];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return FAR_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        return FAR_EL1;
    else
        UNDEFINED;
```

MSR FAR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        NVMem[0x220] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        FAR_EL1 = X[t];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        FAR_EL1 = X[t];
    else
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, FAR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        return FAR_EL1;
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return FAR_EL2;
elsif PSTATE.EL == EL3 then
    return FAR_EL2;
```

MSR FAR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        FAR_EL1 = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    FAR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    FAR_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.40 FAR_EL2, Fault Address Register (EL2)

The FAR_EL2 characteristics are:

Purpose

Holds the faulting Virtual Address for all synchronous Instruction or Data Abort, PC alignment fault and Watchpoint exceptions that are taken to EL2.

Configurations

AArch64 System register FAR_EL2[31:0] is architecturally mapped to AArch32 System register HDFAR[31:0].

AArch64 System register FAR_EL2[63:32] is architecturally mapped to AArch32 System register HIFAR[31:0].

AArch64 System register FAR_EL2[31:0] is architecturally mapped to AArch32 System register DFAR[31:0] (S) when EL2 is implemented.

AArch64 System register FAR_EL2[63:32] is architecturally mapped to AArch32 System register IFAR[31:0] (S) when EL2 is implemented.

If EL2 is not implemented, this register is RES0 from EL3.

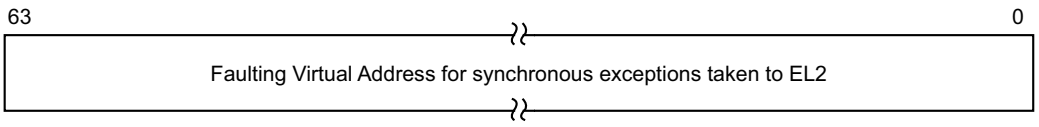
This register has no effect if EL2 is not enabled in the current Security state.

Attributes

FAR_EL2 is a 64-bit register.

Field descriptions

The FAR_EL2 bit assignments are:



Bits [63:0]

Faulting Virtual Address for synchronous exceptions taken to EL2. Exceptions that set the FAR_EL2 are Instruction Aborts (EC `0x20` or `0x21`), Data Aborts (EC `0x24` or `0x25`), PC alignment faults (EC `0x22`), and Watchpoints (EC `0x34` or `0x35`). ESR_EL2.EC holds the EC value for the exception.

For a synchronous External abort, if the VA that generated the abort was from an address range for which TCR_ELx.TBI{<0|1>} == 1 for the translation regime in use when the abort was generated, then the top eight bits of FAR_EL2 are UNKNOWN.

For a synchronous External abort other than a synchronous External abort on a translation table walk, this field is valid only if ESR_EL2.FnV is 0, and the FAR_EL2 is UNKNOWN if ESR_EL2.FnV is 1.

For all other exceptions taken to EL2, the FAR_EL2 is UNKNOWN.

If a memory fault that sets FAR_EL2 is generated from a data cache maintenance or other DC instruction, this field holds the address specified in the register argument of the instruction.

If the exception that updates FAR_EL2 is taken from an Exception level that is using AArch32, the top 32 bits are all zero, unless both of the following apply, in which case the top 32 bits of FAR_ELx are `0x00000001`:

- The faulting address was generated by a load or store instruction that sequentially incremented from address `0xFFFFFFFF`. Such a load or store instruction is CONSTRAINED UNPREDICTABLE. See *Out of range VA* on page K1-7840.
- The implementation treats such incrementing as setting bit[32] of the virtual address to 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For a Data Abort or Watchpoint exception, if address tagging is enabled for the address accessed by the data access that caused the exception, then this field includes the tag. For more information about address tagging, see *Address tagging in AArch64 state* on page D5-2534.

For a synchronous Tag Check Fault abort, bits[63:60] are UNKNOWN.

Execution at EL1 or EL0 makes FAR_EL2 become UNKNOWN.

Note

The address held in this field is an address accessed by the instruction fetch or data access that caused the exception that gave rise to the instruction or data abort. It is the lower address that gave rise to the fault. Where different faults from different addresses arise from the same instruction, such as for an instruction that loads or stores a mis-aligned address that crosses a page boundary, the architecture does not prioritize between those different faults.

FAR_EL2 is made UNKNOWN on an exception return from EL2.

This field resets to an architecturally UNKNOWN value.

Accessing the FAR_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic FAR_EL2 or FAR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, FAR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        return FAR_EL1;
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return FAR_EL2;
elsif PSTATE.EL == EL3 then
    return FAR_EL2;
```

MSR FAR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        FAR_EL1 = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL2 then
    FAR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    FAR_EL2 = X[t];
```

MRS <Xt>, FAR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.FAR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x220];
    else
        return FAR_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return FAR_EL2;
    else
        return FAR_EL1;
elsif PSTATE.EL == EL3 then
    return FAR_EL1;
```

MSR FAR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.FAR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x220] = X[t];
    else
        FAR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        FAR_EL2 = X[t];
    else
        FAR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    FAR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.41 FAR_EL3, Fault Address Register (EL3)

The FAR_EL3 characteristics are:

Purpose

Holds the faulting Virtual Address for all synchronous Instruction or Data Abort and PC alignment fault exceptions that are taken to EL3.

Configurations

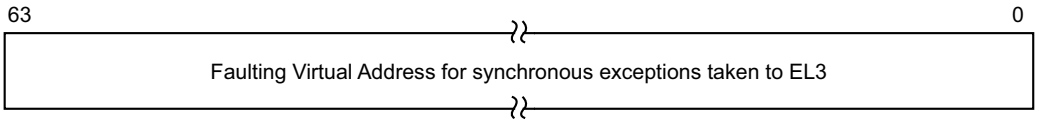
This register is present only when EL3 is implemented. Otherwise, direct accesses to FAR_EL3 are UNDEFINED.

Attributes

FAR_EL3 is a 64-bit register.

Field descriptions

The FAR_EL3 bit assignments are:



Bits [63:0]

Faulting Virtual Address for synchronous exceptions taken to EL3. Exceptions that set the FAR_EL3 are Instruction Aborts (EC `0x20` or `0x21`), Data Aborts (EC `0x24` or `0x25`), and PC alignment faults (EC `0x22`). ESR_EL3.EC holds the EC value for the exception.

For a synchronous External abort, if the VA that generated the abort was from an address range for which TCR_ELx.TBI{<0|1>} == 1 for the translation regime in use when the abort was generated, then the top eight bits of FAR_EL3 are UNKNOWN.

For a synchronous External abort other than a synchronous External abort on a translation table walk, this field is valid only if ESR_EL3.FnV is 0, and the FAR_EL3 is UNKNOWN if ESR_EL3.FnV is 1.

For all other exceptions taken to EL3, the FAR_EL3 is UNKNOWN.

If a memory fault that sets FAR_EL3 is generated from a data cache maintenance or other DC instruction, this field holds the address specified in the register argument of the instruction.

If the exception that updates FAR_EL3 is taken from an Exception Level using AArch32, the top 32 bits are all zero, unless both of the following apply, in which case the top 32 bits of FAR_ELx are `0x00000001`:

- The faulting address was generated by a load or store instruction that sequentially incremented from address `0xFFFFFFFF`. Such a load or store instruction is CONSTRAINED UNPREDICTABLE. See *Out of range VA* on page K1-7840.
- The implementation treats such incrementing as setting bit[32] of the virtual address to 1.

For a Data Abort or Watchpoint exception, if address tagging is enabled for the address accessed by the data access that caused the exception, then this field includes the tag. For more information about address tagging, see *Address tagging in AArch64 state* on page D5-2534.

For a synchronous Tag Check Fault abort, bits[63:60] are UNKNOWN.

Execution at EL2, EL1 or EL0 makes FAR_EL3 become UNKNOWN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

The address held in this register is an address accessed by the instruction fetch or data access that caused the exception that actually gave rise to the instruction or data abort. It is the lowest address that gave rise to the fault. Where different faults from different addresses arise from the same instruction, such as for an instruction that loads or stores a mis-aligned address that crosses a page boundary, the architecture does not prioritize between those different faults.

FAR_EL3 is made UNKNOWN on an exception return from EL3.

This field resets to an architecturally UNKNOWN value.

Accessing the FAR_EL3

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, FAR_EL3

op0	op1	CRn	CRm	op2
0b11	0b110	0b0110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    return FAR_EL3;
```

MSR FAR_EL3, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b110	0b0110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    FAR_EL3 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.42 FPEXC32_EL2, Floating-Point Exception Control register

The FPEXC32_EL2 characteristics are:

Purpose

Allows access to the AArch32 register FPEXC from AArch64 state only. Its value has no effect on execution in AArch64 state.

Configurations

AArch64 System register FPEXC32_EL2[31:0] is architecturally mapped to AArch32 System register FPEXC[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to FPEXC32_EL2 are UNDEFINED.

If EL2 is not implemented but EL3 is implemented, and EL1 is capable of using AArch32, then this register is not RES0.

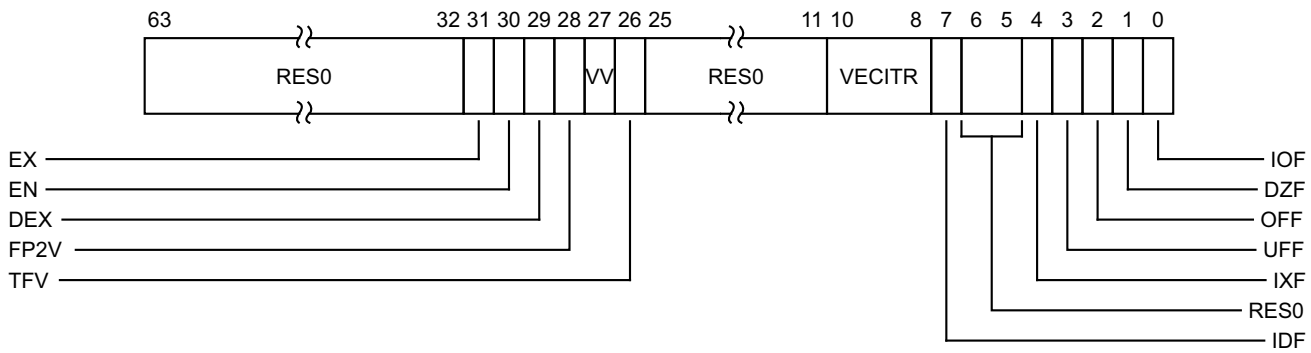
Implemented only if the implementation includes the Advanced SIMD and floating-point functionality.

Attributes

FPEXC32_EL2 is a 64-bit register.

Field descriptions

The FPEXC32_EL2 bit assignments are:



Bits [63:32]

Reserved, RES0.

EX, bit [31]

Exception bit. From Armv8, this bit is RAZ/WI.

This field resets to an architecturally UNKNOWN value.

EN, bit [30]

Enables access to the Advanced SIMD and floating-point functionality from all Exception levels, except that setting this field to 0 does not disable the following:

- VMSR accesses to the FPEXC or FPSID.
- VMRS accesses from the FPEXC, FPSID, MVFR0, MVFR1, or MVFR2.

`0b0` Accesses to the FPSCR, and any of the SIMD and floating-point registers Q0-Q15, including their views as D0-D31 registers or S0-S31 registers, are UNDEFINED at all Exception levels.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` This control permits access to the Advanced SIMD and floating-point functionality at all Exception levels.

Execution of floating-point and Advanced SIMD instructions in AArch32 state can be disabled or trapped by the following controls:

- CPACR.cp10, or, if executing at EL0, CPACR_EL1.FPEN.
- FPEXC.EN.
- If executing in Non-secure state:
  - HCPTR.TCP10, or if EL2 is using AArch64, CPTR_EL2.TFP.
  - NSACR.cp10, or if EL3 is using AArch64, CPTR_EL3.TFP.
- For Advanced SIMD instructions only:
  - CPACR.ASEDIS.
  - If executing in Non-secure state, HCPTR.TASE and NSACR.NSTRCDIS.

See the descriptions of the controls for more information.

Note

When executing at EL0 using AArch32:

- If EL1 is using AArch64 then behavior is as if the value of FPEXC.EN is 1.
- If EL2 is using AArch64 and enabled in the current Security state, and the value of HCR_EL2.{RW, TGE} is {1, 1} then behavior is as if the value of FPEXC.EN is 1.
- If EL2 is using AArch64 and enabled in the current Security state, and the value of HCR_EL2.{RW, TGE} is {0, 1} then it is IMPLEMENTATION DEFINED whether the behavior is:
  - As if the value of FPEXC.EN is 1.
  - Determined by the value of FPEXC32_EL2.EN, as described in this field description. However, Arm deprecates using the value of FPEXC32_EL2.EN to determine behavior.

This field resets to an architecturally UNKNOWN value.

DEX, bit [29]

Defined synchronous exception on floating-point execution.

This field identifies whether a synchronous exception generated by the attempted execution of an instruction was generated by an unallocated encoding. The instruction must be in the encoding space that is identified by the pseudocode function ExecutingCP10or11Instr() returning TRUE. This field also indicates whether the FPEXC32_EL2.TFV field is valid.

The meaning of this bit is:

`0b0` The exception was generated by the attempted execution of an unallocated instruction in the encoding space that is identified by the pseudocode function ExecutingCP10or11Instr(). If FPEXC32_EL2.TFV is RW then it is invalid and UNKNOWN. If FPEXC32_EL2.{IDF, IXF, UFF, OFF, DZF, IOF} are RW then they are invalid and UNKNOWN.

`0b1` The exception was generated during the execution of an unallocated encoding. FPEXC32_EL2.TFV is valid and indicates the cause of the exception.

On an exception that sets this bit to 1 the exception-handling routine must clear this bit to 0.

On an implementation that both does not support trapping of floating-point exceptions and implements the AArch32 FPSCR.{Stride, Len} fields as RAZ, this bit is RES0.

This field resets to an architecturally UNKNOWN value.

FP2V, bit [28]

FPINST2 instruction valid bit. From Armv8, this bit is RES0.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

VV, bit [27]

VECITR valid bit. From Armv8, this bit is RES0.

This field resets to an architecturally UNKNOWN value.

TFV, bit [26]

Trapped Fault Valid bit. Valid only when the value of FPEXC.DEX is 1. When valid, it indicates the cause of the exception and therefore whether the FPEXC.{IDF, IXF, UFF, OFF, DZF, IOF} bits are valid.

`0b0` The exception was caused by the execution of a floating-point VABS, VADD, VDIV, VFMA, VFMS, VFNMA, VFNMS, VMLA, VMLS, VMOV, VMUL, VNEG, VNMLA, VNMLS, VNMUL, VSQRT, or VSUB instruction when one or both of FPSCR.{Stride, Len} was non-zero. If the FPEXC.{IDF, IXF, UFF, OFF, DZF, IOF} bits are RW then they are invalid and UNKNOWN.

`0b1` FPEXC.{IDF, IXF, UFF, OFF, DZF, IOF} indicate the presence of trapped floating-point exceptions that had occurred at the time of the exception. Bits are set for all trapped exceptions that had occurred at the time of the exception.

This bit returns a status value and ignores writes.

When the value of FPEXC.DEX is 0 and this bit is RW, this bit is invalid and UNKNOWN.

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

On an implementation that supports the trapping of floating-point exceptions and implements FPSCR.{Stride, Len} as RAZ, this bit is RAO/WI.

This field resets to an architecturally UNKNOWN value.

Bits [25:11]

Reserved, RES0.

VECITR, bits [10:8]

Vector iteration count. From Armv8, this field is RES1.

This field resets to an architecturally UNKNOWN value.

IDF, bit [7]

Input Denormal trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Input Denormal exception occurred while FPSCR.IDE was 1:

`0b0` Input Denormal exception has not occurred.

`0b1` Input Denormal exception has occurred.

Input Denormal exceptions can occur only when FPSCR.FZ is 1.

Note

A half-precision floating-point value that is flushed to zero because the value of FPSCR.FZ16 is 1 does not generate an Input Denormal exception.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC32_EL2.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

This field resets to an architecturally UNKNOWN value.

Bits [6:5]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

IXF, bit [4]

Inexact trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Inexact exception occurred while FPSCR.IXE was 1:

`0b0` Inexact exception has not occurred.

`0b1` Inexact exception has occurred.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

This field resets to an architecturally UNKNOWN value.

UFF, bit [3]

Underflow trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Underflow exception occurred while FPSCR.UFE was 1:

`0b0` Underflow exception has not occurred.

`0b1` Underflow exception has occurred.

Underflow trapped exceptions can occur:

- On half-precision data-processing instructions only when FPSCR.FZ16 is 0.
- Otherwise only when FPSCR.FZ is 0.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC32_EL2.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

This field resets to an architecturally UNKNOWN value.

OFF, bit [2]

Overflow trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Overflow exception occurred while FPSCR.OFE was 1:

`0b0` Overflow exception has not occurred.

`0b1` Overflow exception has occurred.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

This field resets to an architecturally UNKNOWN value.

DZF, bit [1]

Divide by Zero trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether a Divide by Zero exception occurred while FPSCR.DZE was 1:

`0b0` Divide by Zero exception has not occurred.

`0b1` Divide by Zero exception has occurred.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

IOF, bit [0]

Invalid Operation trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Invalid Operation exception occurred while FPSCR.IOE was 1:

`0b0` Invalid Operation exception has not occurred.

`0b1` Invalid Operation exception has occurred.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

This field resets to an architecturally UNKNOWN value.

Accessing the FPEXC32_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, FPEXC32_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
        AArch64.SystemAccessTrap(EL3, 0x07);
    else
        return FPEXC32_EL2;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TFP == '1' then
        AArch64.SystemAccessTrap(EL3, 0x07);
    else
        return FPEXC32_EL2;
```

MSR FPEXC32_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TFP == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            AArch64.SystemAccessTrap(EL2, 0x07);
        elsif HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
            AArch64.SystemAccessTrap(EL2, 0x07);
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
            AArch64.SystemAccessTrap(EL3, 0x07);
        else
            FPEXC32_EL2 = X[t];
    elsif PSTATE.EL == EL3 then
        if CPTR_EL3.TFP == '1' then
            AArch64.SystemAccessTrap(EL3, 0x07);
        else
            FPEXC32_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.43 GCR_EL1, Tag Control Register.

The GCR_EL1 characteristics are:

Purpose

Tag Control Register.

Configurations

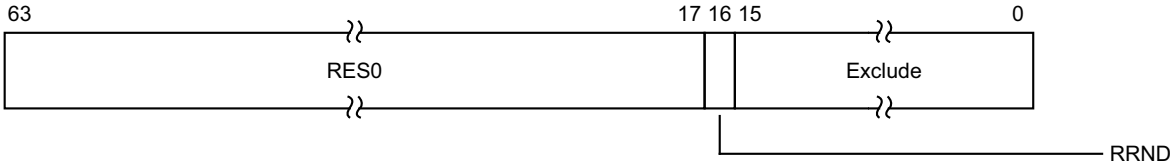
This register is present only when ARMv8.5-MemTag is implemented and ID_AA64PFR1_EL1.MTE != 0b0001. Otherwise, direct accesses to GCR_EL1 are UNDEFINED.

Attributes

GCR_EL1 is a 64-bit register.

Field descriptions

The GCR_EL1 bit assignments are:



Bits [63:17]

Reserved, RES0.

RRND, bit [16]

Controls generation of tag values by the IRG instruction.

0b0 IRG generates a tag value as defined by RandomTag().

0b1 IRG generates an implementation-specific tag value with a distribution of tag values no worse than generated with GCR_EL1.RRND == 0.

This field resets to an architecturally UNKNOWN value.

Exclude, bits [15:0]

Allocation Tag values excluded from selection by ChooseNonExcludedTag().

This field resets to an architecturally UNKNOWN value.

Accessing the GCR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, GCR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.ATA == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.ATA == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        return GCR_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.ATA == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return GCR_EL1;
elsif PSTATE.EL == EL3 then
    return GCR_EL1;
```

MSR GCR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.ATA == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.ATA == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        GCR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.ATA == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        GCR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    GCR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.44 GMID_EL1, Multiple tag transfer ID register

The GMID_EL1 characteristics are:

Purpose

Indicates the block size that is accessed by the LDGM and STGM System instructions.

Configurations

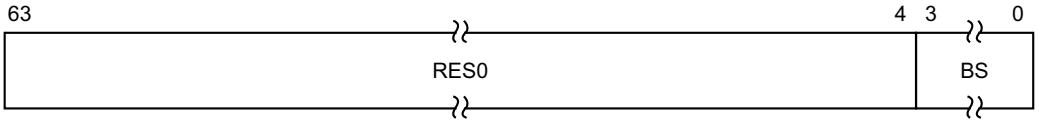
This register is present only when ARMv8.5-MemTag is implemented and ID_AA64PFR1_EL1.MTE != 0b0001. Otherwise, direct accesses to GMID_EL1 are UNDEFINED.

Attributes

GMID_EL1 is a 64-bit register.

Field descriptions

The GMID_EL1 bit assignments are:



Bits [63:4]

Reserved, RES0.

BS, bits [3:0]

Log_2 of the block size in words. The minimum supported size is 16B (value == 2) and the maximum is 256B (value == 6).

Accessing the GMID_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, GMID_EL1

CRn	op0	op1	op2	CRm
0b0000	0b11	0b001	0b100	0b0000

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID5 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return GMID_EL1;
elsif PSTATE.EL == EL2 then
    return GMID_EL1;
elsif PSTATE.EL == EL3 then
    return GMID_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.45 HACR_EL2, Hypervisor Auxiliary Control Register

The HACR_EL2 characteristics are:

Purpose

Controls trapping to EL2 of IMPLEMENTATION DEFINED aspects of EL1 or EL0 operation.

Note

Arm recommends that the values in this register do not cause unnecessary traps to EL2 when HCR_EL2.{E2H, TGE} == {1, 1}.

Configurations

AArch64 System register HACR_EL2[31:0] is architecturally mapped to AArch32 System register HACR[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

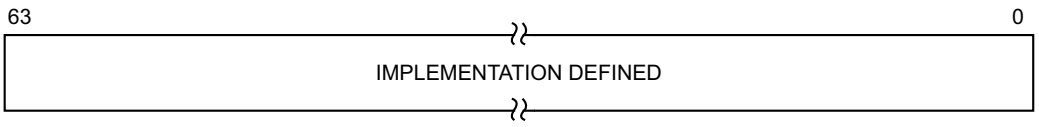
This register has no effect if EL2 is not enabled in the current Security state.

Attributes

HACR_EL2 is a 64-bit register.

Field descriptions

The HACR_EL2 bit assignments are:



IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

Accessing the HACR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, HACR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return HACR_EL2;
elsif PSTATE.EL == EL3 then
    return HACR_EL2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR HACR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    HACR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    HACR_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.46 HAFGRTR_EL2, Hypervisor Activity Monitors Fine-Grained Read Trap Register

The HAFGRTR_EL2 characteristics are:

Purpose

Provides controls for traps of `MRS` reads of Activity Monitors System registers.

Configurations

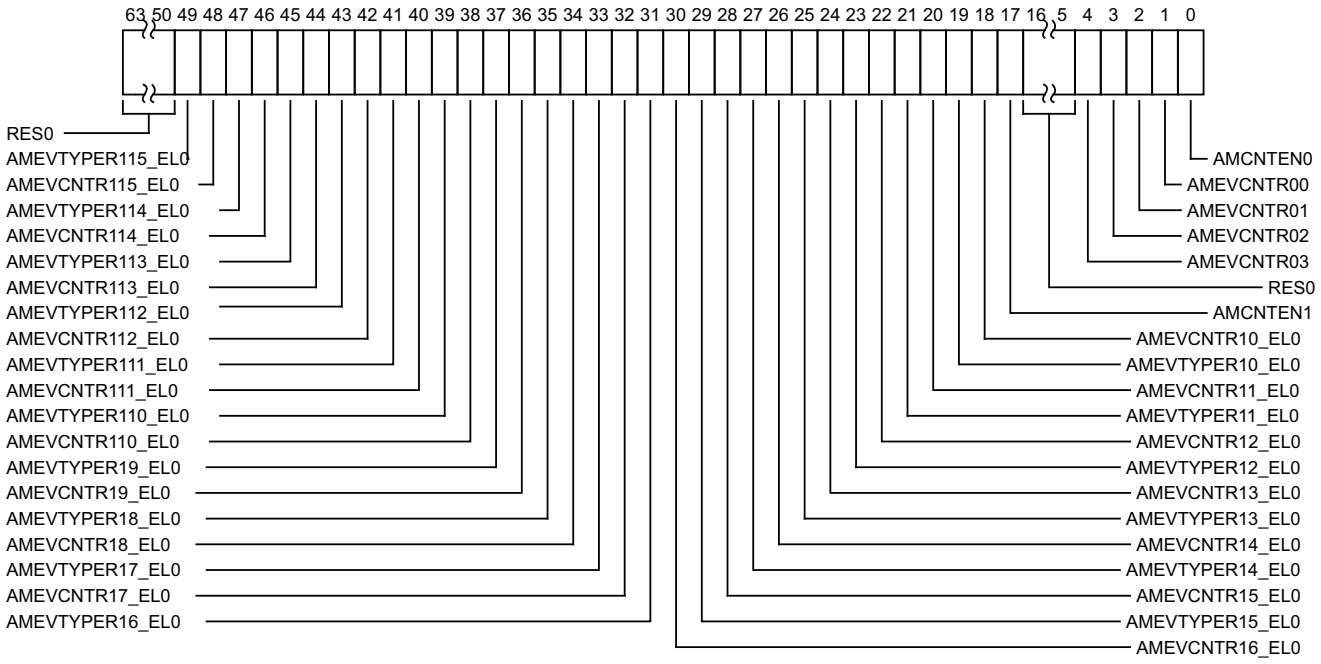
This register is present only when AMUv1 is implemented and ARMv8.6-FGT is implemented. Otherwise, direct accesses to HAFGRTR_EL2 are UNDEFINED.

Attributes

HAFGRTR_EL2 is a 64-bit register.

Field descriptions

The HAFGRTR_EL2 bit assignments are:



Bits [63:50]

Reserved, RES0.

AMEVTYPER115_EL0, bit [49]

Trap `MRS` reads of AMEVTYPER115_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVTYPER115 at EL0 using AArch32 when EL1 is using AArch64 to EL2.

`0b0` `MRS` reads of AMEVTYPER115_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVTYPER115 at EL0 using AArch32 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- `MRS` reads of AMEVTYPER115_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`.
- `MRC` reads of AMEVTYPER115 at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value `0x03`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In a system where the PE resets into EL2, this field resets to `0`.

AMEVCNTR115_EL0, bit [48]

Trap `MRS` reads of AMEVCNTR115_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVCNTR115 at EL0 using AArch32 when EL1 is using AArch64 to EL2.

`0b0` `MRS` reads of AMEVCNTR115_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVCNTR115 at EL0 using AArch32 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- `MRS` reads of AMEVCNTR115_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`.
- `MRC` reads of AMEVCNTR115 at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value `0x03`.

In a system where the PE resets into EL2, this field resets to `0`.

AMEVTYPER114_EL0, bit [47]

Trap `MRS` reads of AMEVTYPER114_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVTYPER114 at EL0 using AArch32 when EL1 is using AArch64 to EL2.

`0b0` `MRS` reads of AMEVTYPER114_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVTYPER114 at EL0 using AArch32 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- `MRS` reads of AMEVTYPER114_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`.
- `MRC` reads of AMEVTYPER114 at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value `0x03`.

In a system where the PE resets into EL2, this field resets to `0`.

AMEVCNTR114_EL0, bit [46]

Trap `MRS` reads of AMEVCNTR114_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVCNTR114 at EL0 using AArch32 when EL1 is using AArch64 to EL2.

`0b0` `MRS` reads of AMEVCNTR114_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVCNTR114 at EL0 using AArch32 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- `MRS` reads of AMEVCNTR114_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`.
- `MRC` reads of AMEVCNTR114 at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value `0x03`.

In a system where the PE resets into EL2, this field resets to `0`.

AMEVTYPER113_EL0, bit [45]

Trap `MRS` reads of AMEVTYPER113_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVTYPER113 at EL0 using AArch32 when EL1 is using AArch64 to EL2.

`0b0` `MRS` reads of AMEVTYPER113_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVTYPER113 at EL0 using AArch32 are not affected by this bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- `MRS` reads of AMEVTYPER113_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`.
- `MRC` reads of AMEVTYPER113 at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value `0x03`.

In a system where the PE resets into EL2, this field resets to `0`.

AMEVCNTR113_EL0, bit [44]

Trap `MRS` reads of AMEVCNTR113_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVCNTR113 at EL0 using AArch32 when EL1 is using AArch64 to EL2.

`0b0` `MRS` reads of AMEVCNTR113_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVCNTR113 at EL0 using AArch32 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- `MRS` reads of AMEVCNTR113_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`.
- `MRC` reads of AMEVCNTR113 at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value `0x03`.

In a system where the PE resets into EL2, this field resets to `0`.

AMEVTYPER112_EL0, bit [43]

Trap `MRS` reads of AMEVTYPER112_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVTYPER112 at EL0 using AArch32 when EL1 is using AArch64 to EL2.

`0b0` `MRS` reads of AMEVTYPER112_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVTYPER112 at EL0 using AArch32 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- `MRS` reads of AMEVTYPER112_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`.
- `MRC` reads of AMEVTYPER112 at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value `0x03`.

In a system where the PE resets into EL2, this field resets to `0`.

AMEVCNTR112_EL0, bit [42]

Trap `MRS` reads of AMEVCNTR112_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVCNTR112 at EL0 using AArch32 when EL1 is using AArch64 to EL2.

`0b0` `MRS` reads of AMEVCNTR112_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVCNTR112 at EL0 using AArch32 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- `MRS` reads of AMEVCNTR112_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`.
- `MRC` reads of AMEVCNTR112 at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value `0x03`.

In a system where the PE resets into EL2, this field resets to `0`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

AMEVTYPER111_EL0, bit [41]

Trap MRS reads of AMEVTYPER111_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVTYPER111 at EL0 using AArch32 when EL1 is using AArch64 to EL2.

0b0 MRS reads of AMEVTYPER111_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVTYPER111 at EL0 using AArch32 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- MRS reads of AMEVTYPER111_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.
- MRC reads of AMEVTYPER111 at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

In a system where the PE resets into EL2, this field resets to 0.

AMEVCNTR111_EL0, bit [40]

Trap MRS reads of AMEVCNTR111_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVCNTR111 at EL0 using AArch32 when EL1 is using AArch64 to EL2.

0b0 MRS reads of AMEVCNTR111_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVCNTR111 at EL0 using AArch32 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- MRS reads of AMEVCNTR111_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.
- MRC reads of AMEVCNTR111 at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

In a system where the PE resets into EL2, this field resets to 0.

AMEVTYPER110_EL0, bit [39]

Trap MRS reads of AMEVTYPER110_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVTYPER110 at EL0 using AArch32 when EL1 is using AArch64 to EL2.

0b0 MRS reads of AMEVTYPER110_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVTYPER110 at EL0 using AArch32 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- MRS reads of AMEVTYPER110_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.
- MRC reads of AMEVTYPER110 at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

In a system where the PE resets into EL2, this field resets to 0.

AMEVCNTR110_EL0, bit [38]

Trap MRS reads of AMEVCNTR110_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVCNTR110 at EL0 using AArch32 when EL1 is using AArch64 to EL2.

0b0 MRS reads of AMEVCNTR110_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVCNTR110 at EL0 using AArch32 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- MRS reads of AMEVCNTR110_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- MRC reads of AMEVCNTR110 at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

In a system where the PE resets into EL2, this field resets to 0.

AMEVTYPER19_EL0, bit [37]

Trap MRS reads of AMEVTYPER19_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVTYPER19 at EL0 using AArch32 when EL1 is using AArch64 to EL2.

0b0 MRS reads of AMEVTYPER19_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVTYPER19 at EL0 using AArch32 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- MRS reads of AMEVTYPER19_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.
- MRC reads of AMEVTYPER19 at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

In a system where the PE resets into EL2, this field resets to 0.

AMEVCNTR19_EL0, bit [36]

Trap MRS reads of AMEVCNTR19_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVCNTR19 at EL0 using AArch32 when EL1 is using AArch64 to EL2.

0b0 MRS reads of AMEVCNTR19_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVCNTR19 at EL0 using AArch32 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- MRS reads of AMEVCNTR19_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.
- MRC reads of AMEVCNTR19 at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

In a system where the PE resets into EL2, this field resets to 0.

AMEVTYPER18_EL0, bit [35]

Trap MRS reads of AMEVTYPER18_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVTYPER18 at EL0 using AArch32 when EL1 is using AArch64 to EL2.

0b0 MRS reads of AMEVTYPER18_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVTYPER18 at EL0 using AArch32 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- MRS reads of AMEVTYPER18_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.
- MRC reads of AMEVTYPER18 at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

In a system where the PE resets into EL2, this field resets to 0.

AMEVCNTR18_EL0, bit [34]

Trap MRS reads of AMEVCNTR18_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVCNTR18 at EL0 using AArch32 when EL1 is using AArch64 to EL2.

0b0 MRS reads of AMEVCNTR18_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVCNTR18 at EL0 using AArch32 are not affected by this bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- `MRS` reads of AMEVCNTR18_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`.
- `MRC` reads of AMEVCNTR18 at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value `0x03`.

In a system where the PE resets into EL2, this field resets to `0`.

AMEVTYPER17_EL0, bit [33]

Trap `MRS` reads of AMEVTYPER17_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVTYPER17 at EL0 using AArch32 when EL1 is using AArch64 to EL2.

`0b0` `MRS` reads of AMEVTYPER17_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVTYPER17 at EL0 using AArch32 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- `MRS` reads of AMEVTYPER17_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`.
- `MRC` reads of AMEVTYPER17 at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value `0x03`.

In a system where the PE resets into EL2, this field resets to `0`.

AMEVCNTR17_EL0, bit [32]

Trap `MRS` reads of AMEVCNTR17_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVCNTR17 at EL0 using AArch32 when EL1 is using AArch64 to EL2.

`0b0` `MRS` reads of AMEVCNTR17_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVCNTR17 at EL0 using AArch32 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- `MRS` reads of AMEVCNTR17_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`.
- `MRC` reads of AMEVCNTR17 at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value `0x03`.

In a system where the PE resets into EL2, this field resets to `0`.

AMEVTYPER16_EL0, bit [31]

Trap `MRS` reads of AMEVTYPER16_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVTYPER16 at EL0 using AArch32 when EL1 is using AArch64 to EL2.

`0b0` `MRS` reads of AMEVTYPER16_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVTYPER16 at EL0 using AArch32 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- `MRS` reads of AMEVTYPER16_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`.
- `MRC` reads of AMEVTYPER16 at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value `0x03`.

In a system where the PE resets into EL2, this field resets to `0`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

AMEVCNTR16_EL0, bit [30]

Trap MRS reads of AMEVCNTR16_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVCNTR16 at EL0 using AArch32 when EL1 is using AArch64 to EL2.

0b0 MRS reads of AMEVCNTR16_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVCNTR16 at EL0 using AArch32 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- MRS reads of AMEVCNTR16_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.
- MRC reads of AMEVCNTR16 at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

In a system where the PE resets into EL2, this field resets to 0.

AMEVTYPER15_EL0, bit [29]

Trap MRS reads of AMEVTYPER15_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVTYPER15 at EL0 using AArch32 when EL1 is using AArch64 to EL2.

0b0 MRS reads of AMEVTYPER15_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVTYPER15 at EL0 using AArch32 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- MRS reads of AMEVTYPER15_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.
- MRC reads of AMEVTYPER15 at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

In a system where the PE resets into EL2, this field resets to 0.

AMEVCNTR15_EL0, bit [28]

Trap MRS reads of AMEVCNTR15_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVCNTR15 at EL0 using AArch32 when EL1 is using AArch64 to EL2.

0b0 MRS reads of AMEVCNTR15_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVCNTR15 at EL0 using AArch32 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- MRS reads of AMEVCNTR15_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.
- MRC reads of AMEVCNTR15 at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

In a system where the PE resets into EL2, this field resets to 0.

AMEVTYPER14_EL0, bit [27]

Trap MRS reads of AMEVTYPER14_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVTYPER14 at EL0 using AArch32 when EL1 is using AArch64 to EL2.

0b0 MRS reads of AMEVTYPER14_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVTYPER14 at EL0 using AArch32 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- MRS reads of AMEVTYPER14_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- `MRC` reads of AMEVTYPER14 at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value `0x03`.

In a system where the PE resets into EL2, this field resets to `0`.

AMEVCNTR14_EL0, bit [26]

Trap `MRS` reads of AMEVCNTR14_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVCNTR14 at EL0 using AArch32 when EL1 is using AArch64 to EL2.

`0b0` `MRS` reads of AMEVCNTR14_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVCNTR14 at EL0 using AArch32 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- `MRS` reads of AMEVCNTR14_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`.
- `MRC` reads of AMEVCNTR14 at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value `0x03`.

In a system where the PE resets into EL2, this field resets to `0`.

AMEVTYPER13_EL0, bit [25]

Trap `MRS` reads of AMEVTYPER13_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVTYPER13 at EL0 using AArch32 when EL1 is using AArch64 to EL2.

`0b0` `MRS` reads of AMEVTYPER13_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVTYPER13 at EL0 using AArch32 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- `MRS` reads of AMEVTYPER13_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`.
- `MRC` reads of AMEVTYPER13 at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value `0x03`.

In a system where the PE resets into EL2, this field resets to `0`.

AMEVCNTR13_EL0, bit [24]

Trap `MRS` reads of AMEVCNTR13_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVCNTR13 at EL0 using AArch32 when EL1 is using AArch64 to EL2.

`0b0` `MRS` reads of AMEVCNTR13_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVCNTR13 at EL0 using AArch32 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- `MRS` reads of AMEVCNTR13_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`.
- `MRC` reads of AMEVCNTR13 at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value `0x03`.

In a system where the PE resets into EL2, this field resets to `0`.

AMEVTYPER12_EL0, bit [23]

Trap `MRS` reads of AMEVTYPER12_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVTYPER12 at EL0 using AArch32 when EL1 is using AArch64 to EL2.

`0b0` `MRS` reads of AMEVTYPER12_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVTYPER12 at EL0 using AArch32 are not affected by this bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- `MRS` reads of AMEVTYPER12_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`.
- `MRC` reads of AMEVTYPER12 at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value `0x03`.

In a system where the PE resets into EL2, this field resets to `0`.

AMEVCNTR12_EL0, bit [22]

Trap `MRS` reads of AMEVCNTR12_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVCNTR12 at EL0 using AArch32 when EL1 is using AArch64 to EL2.

`0b0` `MRS` reads of AMEVCNTR12_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVCNTR12 at EL0 using AArch32 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- `MRS` reads of AMEVCNTR12_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`.
- `MRC` reads of AMEVCNTR12 at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value `0x03`.

In a system where the PE resets into EL2, this field resets to `0`.

AMEVTYPER11_EL0, bit [21]

Trap `MRS` reads of AMEVTYPER11_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVTYPER11 at EL0 using AArch32 when EL1 is using AArch64 to EL2.

`0b0` `MRS` reads of AMEVTYPER11_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVTYPER11 at EL0 using AArch32 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- `MRS` reads of AMEVTYPER11_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`.
- `MRC` reads of AMEVTYPER11 at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value `0x03`.

In a system where the PE resets into EL2, this field resets to `0`.

AMEVCNTR11_EL0, bit [20]

Trap `MRS` reads of AMEVCNTR11_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVCNTR11 at EL0 using AArch32 when EL1 is using AArch64 to EL2.

`0b0` `MRS` reads of AMEVCNTR11_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of AMEVCNTR11 at EL0 using AArch32 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- `MRS` reads of AMEVCNTR11_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`.
- `MRC` reads of AMEVCNTR11 at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value `0x03`.

In a system where the PE resets into EL2, this field resets to `0`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

AMEVTYPER10_EL0, bit [19]

Trap MRS reads of AMEVTYPER10_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVTYPER10 at EL0 using AArch32 when EL1 is using AArch64 to EL2.

0b0 MRS reads of AMEVTYPER10_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVTYPER10 at EL0 using AArch32 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- MRS reads of AMEVTYPER10_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.
- MRC reads of AMEVTYPER10 at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

In a system where the PE resets into EL2, this field resets to 0.

AMEVCNTR10_EL0, bit [18]

Trap MRS reads of AMEVCNTR10_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVCNTR10 at EL0 using AArch32 when EL1 is using AArch64 to EL2.

0b0 MRS reads of AMEVCNTR10_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVCNTR10 at EL0 using AArch32 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- MRS reads of AMEVCNTR10_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.
- MRC reads of AMEVCNTR10 at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

In a system where the PE resets into EL2, this field resets to 0.

AMCNTEN1, bit [17]

Trap MRS reads and MRC reads of multiple System registers.

Enables a trap to EL2 the following operations:

- At EL1 and EL0 using AArch64: MRS reads of AMCNTENCLR1_EL0 and AMCNTENSET1_EL0.
- At EL0 using Arch32 when EL1 is using AArch64: MRC reads of AMCNTENCLR1 and AMCNTENSET1.

0b0 The operations listed above are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- MRS reads at EL1 and EL0 using AArch64 of AMCNTENCLR1_EL0 and AMCNTENSET1_EL0 are trapped to EL2 and reported with EC syndrome value 0x18.
- MRC reads at EL0 using AArch32 of AMCNTENCLR1 and AMCNTENSET1 are trapped to EL2 and reported with EC syndrome value 0x03, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Bits [16:5]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

AMEVCNTR0<x>_EL0, bit [x+1], for x = 0 to 3

Trap MRS reads of AMEVCNTR0<x>_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVCNTR0<x> at EL0 using AArch32 when EL1 is using AArch64 to EL2.

0b0 MRS reads of AMEVCNTR0<x>_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVCNTR0<x> at EL0 using AArch32 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- MRS reads of AMEVCNTR0<x>_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.
- MRC reads of AMEVCNTR0<x> at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

In a system where the PE resets into EL2, this field resets to 0.

AMCNTEN0, bit [0]

Trap MRS reads and MRC reads of multiple System registers.

Enables a trap to EL2 the following operations:

- At EL1 and EL0 using AArch64: MRS reads of AMCNTENCLR0_EL0 and AMCNTENSET0_EL0.
- At EL0 using Arch32 when EL1 is using AArch64: MRC reads of AMCNTENCLR0 and AMCNTENSET0.

0b0 The operations listed above are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- MRS reads at EL1 and EL0 using AArch64 of AMCNTENCLR0_EL0 and AMCNTENSET0_EL0 are trapped to EL2 and reported with EC syndrome value 0x18.
- MRC reads at EL0 using AArch32 of AMCNTENCLR0 and AMCNTENSET0 are trapped to EL2 and reported with EC syndrome value 0x03, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Accessing the HAFGRTR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, HAFGRTR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0011	0b0001	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        return NVMem[0x1E8];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.FGTEn == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        return HAFGRTR_EL2;
elsif PSTATE.EL == EL3 then
    return HAFGRTR_EL2;
```

MSR HAFGRTR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0011	0b0001	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NVMem[0x1E8] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.FGTEn == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        HAFGRTR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    HAFGRTR_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.47 HCR_EL2, Hypervisor Configuration Register

The HCR_EL2 characteristics are:

Purpose

Provides configuration controls for virtualization, including defining whether various operations are trapped to EL2.

Configurations

AArch64 System register HCR_EL2[31:0] is architecturally mapped to AArch32 System register HCR[31:0].

AArch64 System register HCR_EL2[63:32] is architecturally mapped to AArch32 System register HCR2[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

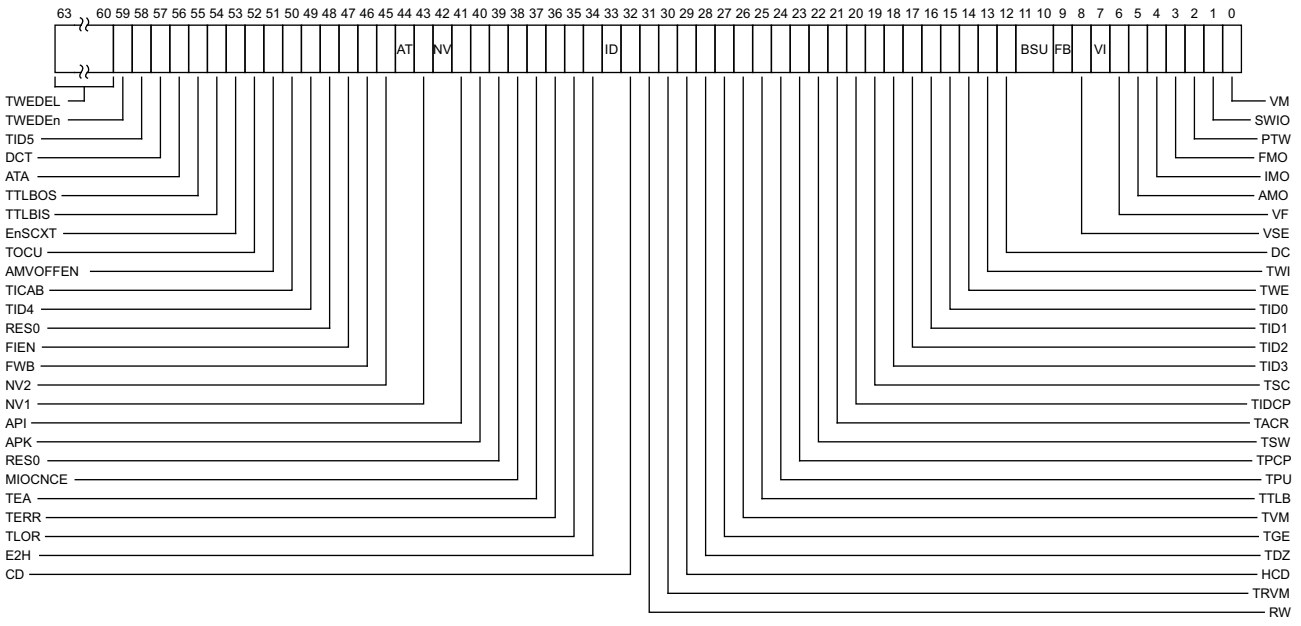
The bits in this register behave as if they are 0 for all purposes other than direct reads of the register if EL2 is not enabled in the current Security state.

Attributes

HCR_EL2 is a 64-bit register.

Field descriptions

The HCR_EL2 bit assignments are:



TWEDEL, bits [63:60]

When ARMv8.6-TWED is implemented:

TWE Delay. A 4-bit unsigned number that, when HCR_EL2.TWEDEn is 1, encodes the minimum delay in taking a trap of WFE caused by HCR_EL2.TWE as 2^(TWEDEL + 8) cycles.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TWEDEn, bit [59]

When ARMv8.6-TWED is implemented:

TWE Delay Enable. Enables a configurable delayed trap of the WFE instruction caused by HCR_EL2.TWE.

`0b0` The delay for taking a WFE trap is IMPLEMENTATION DEFINED.

`0b1` The delay for taking a WFE trap is at least the number of cycles defined in HCR_EL2.TWEDEL.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TID5, bit [58]

When ARMv8.5-MemTag is implemented:

Trap ID group 5. Traps the following register accesses to EL2, when EL2 is enabled in the current Security state:

AArch64:

- GMID_EL1.

`0b0` This control does not cause any instructions to be trapped.

`0b1` The specified EL1 and EL0 accesses to ID group 5 registers are trapped to EL2.

When the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field has an Effective value of 0 for all purposes other than a direct read of the value of this bit.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DCT, bit [57]

When ARMv8.5-MemTag is implemented:

Default Cacheability Tagging. When HCR_EL2.DC is in effect, controls whether stage 1 translations are treated as Tagged or Untagged.

`0b0` Stage 1 translations are treated as Untagged.

`0b1` Stage 1 translations are treated as Tagged.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ATA, bit [56]

When ARMv8.5-MemTag is implemented:

Allocation Tag Access. When SCR_EL3.ATA=1 and HCR_EL2.{E2H,TGE} != {1,1}, controls EL1 and EL0 access to Allocation Tags.

When access is prevented:

- Instructions which Load or Store data are Unchecked.
- Instructions which Load or Store Allocation Tags treat the Allocation Tag as RAZ/WI.
- Instructions which insert Logical Address Tags into addresses treat the Allocation Tag used to generate the Logical Address Tag as 0.
- Cache maintenance instructions which invalidate Allocation Tags from caches behave as the equivalent Clean and Invalidate operation on Allocation Tags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- MRS and MSR instructions at EL1 using GCR_EL1, RGSR_EL1, TFSR_EL1, TFSR_EL2, or TFSRE0_EL1 that are not UNDEFINED are trapped to EL2.

`0b0` Access is prevented.

`0b1` Access is not prevented.

This field is permitted to be cached in a TLB.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TTLBOS, bit [55]

When ARMv8.2-EVT is implemented:

Trap TLB maintenance instructions that operate on the Outer Shareable domain. Traps execution of those TLB maintenance instructions at EL1 to EL2, when EL2 is enabled in the current Security state. This applies to the following instructions:

TLBI VMALLE1OS, TLBI VAE1OS, TLBI ASIDE1OS,TLBI VAAE1OS, TLBI VALE1OS, TLBI VAALE1OS,TLBI RVAE1OS, TLBI RVAAE1OS,TLBI RVALE1OS, and TLBI RVAALE1OS.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Execution of the specified instructions are trapped to EL2.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TTLBIS, bit [54]

When ARMv8.2-EVT is implemented:

Trap TLB maintenance instructions that operate on the Inner Shareable domain. Traps execution of those TLB maintenance instructions at EL1 to EL2, when EL2 is enabled in the current Security state. This applies to the following instructions:

- When EL1 is using AArch64, TLBI VMALLE1IS, TLBI VAE1IS, TLBI ASIDE1IS,TLBI VAAE1IS, TLBI VALE1IS, TLBI VAALE1IS,TLBI RVAE1IS, TLBI RVAAE1IS,TLBI RVALE1IS, and TLBI RVAALE1IS.
- When EL1 is using AArch32, TLBIALLIS, TLBIMVAIS, TLBIASIDIS, TLBIMVAAIS, TLBIMVALIS, and TLBIMVAALIS.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Execution of the specified instructions are trapped to EL2.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnSCXT, bit [53]

When ARMv8.0-CSV2 is implemented:

Enable Access to the SCXTNUM_EL1 and SCXTNUM_EL0 registers. The defined values are:

`0b0` When (HCR_EL2.TGE==0 or HCR_EL2.E2H==0) and EL2 is enabled in the current Security state, EL1 and EL0 access to SCXTNUM_EL0 and EL1 access to SCXTNUM_EL1 is disabled by this mechanism, causing an exception to EL2, and the values of these registers to be treated as 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When ((HCR_EL2.TGE==1 and HCR_EL2.E2H==1) and EL2 is enabled in the current Security state, EL0 access to SCXTNUM_EL0 is disabled by this mechanism, causing an exception to EL2, and the value of this register to be treated as 0.

`0b1` This control does not cause accesses to SCXTNUM_EL0 or SCXTNUM_EL1 to be trapped.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1,1}, this bit has no effect on execution at EL0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TOCU, bit [52]

When ARMv8.2-EVT is implemented:

Trap cache maintenance instructions that operate to the Point of Unification. Traps execution of those cache maintenance instructions to EL2, when EL2 is enabled in the current Security state. This applies to the following instructions:

- When SCTLR_EL1.UCI is 1, HCR_EL2.{TGE, E2H} is not {1, 1}, and EL0 is using AArch64, IC IVAU, DC CVAU.
- When EL1 is using AArch64, IC IVAU, IC IALLU, DC CVAU.
- When EL1 is using AArch32, ICIMVAU, ICIALLU, DCCMVAU.

Note

An exception generated because an instruction is UNDEFINED at EL0 is higher priority than this trap to EL2. In addition:

- IC IALLUIS and IC IALLU are always UNDEFINED at EL0 using AArch64.
- ICIMVAU, ICIALLU, ICIALLUIS, and DCCMVAU are always UNDEFINED at EL0 using AArch32.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Execution of the specified instructions are trapped to EL2.

If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean by VA to the Point of Unification instruction can be trapped when the value of this control is 1.

If the Point of Unification is before any level of instruction cache, it is IMPLEMENTATION DEFINED whether the execution of any instruction cache invalidate to the Point of Unification instruction can be trapped when the value of this control is 1.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

AMVOFFEN, bit [51]

When ARMv8.6-AMU is implemented:

Activity Monitors Virtual Offsets Enable.

`0b0` Virtualization of the Activity Monitors is disabled. Indirect reads of the virtual offset registers are zero.

`0b1` Virtualization of the Activity Monitors is enabled.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

Reserved, RES0.

TICAB, bit [50]

When ARMv8.2-EVT is implemented:

Trap ICIALLUIS/IC IALLUIS cache maintenance instructions. Traps execution of those cache maintenance instructions at EL1 to EL2, when EL2 is enabled in the current Security state. This applies to the following instructions:

- When EL1 is using AArch64, IC IALLUIS.
- When EL1 is using AArch32, ICIALLUIS.

`0b0` This control does not cause any instructions to be trapped.

`0b1` EL1 execution of the specified instructions is trapped to EL2.

If the Point of Unification is before any level of instruction cache, it is IMPLEMENTATION DEFINED whether the execution of any instruction cache invalidate to the Point of Unification instruction can be trapped when the value of this control is 1.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TID4, bit [49]

When ARMv8.2-EVT is implemented:

Trap ID group 4. Traps the following register accesses to EL2, when EL2 is enabled in the current Security state:

AArch64:

- EL1 reads of CCSIDR_EL1, CCSIDR2_EL1, CLIDR_EL1, and CSSELR_EL1.
- EL1 writes to CSSELR_EL1.

AArch32:

- EL1 reads of CCSIDR, CCSIDR2, CLIDR, and CSSELR.
- EL1 writes to CSSELR.

`0b0` This control does not cause any instructions to be trapped.

`0b1` The specified EL1 and EL0 accesses to ID group 4 registers are trapped to EL2.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [48]

Reserved, RES0.

FIEN, bit [47]

When ARMv8.4-RAS is implemented:

Fault Injection Enable. Unless this bit is set to 1, accesses to the ERXPFGCDN_EL1, ERXPFGCTL_EL1, and ERXPFGF_EL1 registers from EL1 generate a Trap exception to EL2, when EL2 is enabled in the current Security state, reported using EC syndrome value `0x18`.

`0b0` Accesses to the specified registers from EL1 are trapped to EL2, when EL2 is enabled in the current Security state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` This control does not cause any instructions to be trapped.

If EL2 is disabled in the current Security state, the Effective value of HCR_EL2.FIEN is `0b1`.

If ERRIDR_EL1.NUM is zero, meaning no error records are implemented, or no error record accessible using System registers is owned by a node that implements the RAS Common Fault Injection Model Extension, then this bit might be RES0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FWB, bit [46]

When ARMv8.4-S2FWB is implemented:

Defines the combined cacheability attributes in a 2 stage translation regime.

`0b0` When this bit is 0, then:

- The combination of stage 1 and stage 2 translations on memory type and cacheability attributes are as described in the Armv8.0 architecture. For more information see *Combining the stage 1 and stage 2 attributes, EL1&0 translation regime* on page D5-2634
- The encoding of the stage 2 memory type and cacheability attributes in bits[5:2] of the stage 2 page or block descriptors are as described in the Armv8.0 architecture.

`0b1` When this bit is 1, then:

- Bit[5] of stage 2 page or block descriptor is RES0.
- When bit[4] of stage 2 page or block descriptor is 1 and when:
  - — Bits[3:2] of stage 2 page or block descriptor are `0b11`, the resultant memory type and inner or outer cacheability attribute is the same as the stage 1 memory type and inner or outer cacheability attribute.
  - — Bits[3:2] of stage 2 page or block descriptor are `0b10`, the resultant memory type and attribute is Normal Write-Back.
  - — Bits[3:2] of stage 2 page or block descriptor are `0b0x`, the resultant memory type will be Normal Non-cacheable except where the stage 1 memory type was Device->attr< the resultant memory type will be Device->attr<
- When bit[4] of stage 2 page or block descriptor is 0 the memory type is Device, and when:
  - — Bits[3:2] of stage 2 page or block descriptor are `0b00`, the stage 2 memory type is Device-nGnRnE.
  - — Bits[3:2] of stage 2 page or block descriptor are `0b01`, the stage 2 memory type is Device-nGnRE.
  - — Bits[3:2] of stage 2 page or block descriptor are `0b10`, the stage 2 memory type is Device-nGRE.
  - — Bits[3:2] of stage 2 page or block descriptor are `0b11`, the stage 2 memory type is Device-GRE.
- If the stage 1 translation specifies a cacheable memory type, then the stage 1 cache allocation hint is applied to the final cache allocation hint where the final memory type is cacheable.
- If the stage 1 translation does not specify a cacheable memory type, then if the final memory type is cacheable, it is treated as read allocate, write allocate.

The stage 1 and stage 2 memory types are combined in the manner described in *Combining the stage 1 and stage 2 attributes, EL1&0 translation regime* on page D5-2634

In Secure state, this bit applies to both the Secure stage 2 translation and the Non-secure stage 2 translation.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This bit is permitted to be cached in a TLB.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NV2, bit [45]

When ARMv8.4-NV is implemented:

Nested Virtualization. Changes the behaviors of HCR_EL2.{NV, NV1} to provide a mechanism for hardware to transform reads and writes from System registers into reads and writes from memory.

`0b0` This bit has no effect on the behavior of HCR_EL2.{NV, NV1}. The behavior of HCR_EL2.{NV, NV1} is as defined for ARMv8.3-NV.

`0b1` Redefines behavior of HCR_EL2{NV, NV1} to enable:

- Transformation of read/writes to registers into read/writes to memory.
- Redirection of EL2 registers to EL1 registers.

Any exception taken from EL1 and taken to EL1 causes SPSR_EL1.M[3:2] to be set to `0b10` and not `0b01`.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

AT, bit [44]

When ARMv8.3-NV is implemented:

Address Translation. EL1 execution of the following address translation instructions is trapped to EL2, when EL2 is enabled in the current Security state, reported using EC syndrome value `0x18`:

- AT S1E0R, AT S1E0W, AT S1E1R, AT S1E1W, AT S1E1RP, AT S1E1WP.

`0b0` This control does not cause any instructions to be trapped.

`0b1` EL1 execution of the specified instructions is trapped to EL2.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NV1, bit [43]

When ARMv8.4-NV is implemented:

Nested Virtualization.

`0b0` If HCR_EL2.{NV, NV2} are both 1, accesses executed from EL1 to implemented EL12, EL02, or EL2 registers are transformed to loads and stores.

If HCR_EL2.NV2 is 0 or HCR_EL2.{NV, NV2} == {0, 1}, this control does not cause any instructions to be trapped.

`0b1` If HCR_EL2.NV2 is 1, accesses executed from EL1 to implemented EL2 registers are transformed to loads and stores.

If HCR_EL2.NV2 is 0, EL1 accesses to VBAR_EL1, ELR_EL1, and SPSR_EL1, are trapped to EL2, when EL2 is enabled in the current Security state, reported using EC syndrome value `0x18`.

If HCR_EL2.NV2 is 1, the value of HCR_EL2.NV1 defines which EL1 register accesses are transformed to loads and stores. These transformed accesses have priority over the trapping of registers.

The trapping of EL1 registers caused by other control bits has priority over the transformation of these accesses.

If a register is specified that is not implemented by an implementation, then access to that register are UNDEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the list of registers affected, see *Enhanced support for nested virtualization* on page D5-2646.

If HCR_EL2.{NV, NV1, NV2} are {1, 0, 0},any exception taken from EL1, and taken to EL1, causes the SPSR_EL1.M[3:2] to be set to `0b10`, and not `0b01`.

If HCR_EL2.{NV, NV1, NV2} are {1, 1, 0}, then:

- The EL1 translation table Block and Page descriptors:
  - Bit[54] holds the PXN instead of the UXN.
  - Bit[53] is RES0.
  - Bit[6] is treated as 0 regardless of the actual value.
- If Hierarchical Permissions are enabled, the EL1 translation table Table descriptors are as follows:
  - Bit[61] is treated as 0 regardless of the actual value.
  - Bit[60] holds the PXNTable instead of the UXNTable.
  - Bit[59] is RES0.
- When executing at EL1, the PSTATE.PAN bit is treated as zero for all purposes except reading the value of the bit.
- When executing at EL1, the LDTR* instructions are treated as the equivalent LDR* instructions, and the STTR* instructions are treated as the equivalent STR* instructions.

If HCR_EL2.{NV, NV1, NV2} are {0, 1, 0}, then the behavior is a CONSTRAINED UNPREDICTABLE choice of:

- Behaving as if HCR_EL2.NV is 1 and HCR_EL2.NV1 is 1 for all purposes other than reading than reading back the value of the HCR_EL2.NV bit.
- Behaving as if HCR_EL2.NV is 0 and HCR_EL2.NV1 is 0 for all purposes other than reading than reading back the value of the HCR_EL2.NV1 bit.
- Behaving with regard to the HCR_EL2.NV and HCR_EL2.NV1 bits behavior as defined in the rest of this description.

This bit is permitted to be cached in a TLB.

This field resets to an architecturally UNKNOWN value.

When ARMv8.3-NV is implemented:

Nested Virtualization. EL1 accesses to certain registers are trapped to EL2, when EL2 is enabled in the current Security state.

`0b0` This control does not cause any instructions to be trapped.

`0b1` EL1 accesses to VBAR_EL1, ELR_EL1, SPSR_EL1 are trapped to EL2, when EL2 is enabled in the current Security state, reported using EC syndrome value `0x18`.

If HCR_EL2.NV is 1 and HCR_EL2.NV1 is 0 then the following effects also apply:

- Any exception taken from EL1, and taken to EL1, causes the SPSR_EL1.M[3:2] to be set to `0b10`, and not `0b01`.

If the bits HCR_EL2.NV and HCR_EL2.NV1 are both set to 1 then following effects also apply:

- The EL1 translation table Block and Page descriptors:
  - Bit[54] holds the PXN instead of the UXN.
  - Bit[53] is RES0.
  - Bit[6] is treated as 0 regardless of the actual value.
- If Hierarchical Permissions are enabled, the EL1 translation table Table descriptors are as follows:
  - Bit[61] is treated as 0 regardless of the actual value.
  - Bit[60] holds the PXNTable instead of the UXNTable.
  - Bit[59] is RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- When executing at EL1, the PSTATE.PAN bit is treated as zero for all purposes except reading the value of the bit.
- When executing at EL1, the LDTR* instructions are treated as the equivalent LDR* instructions, and the STTR* instructions are treated as the equivalent STR* instructions.

If HCR_EL2.NV is 0 and HCR_EL2.NV1 is 1 then the behavior is a CONSTRAINED UNPREDICTABLE choice of:

- Behaving as if HCR_EL2.NV is 1 and HCR_EL2.NV1 is 1 for all purposes other than reading than reading back the value of the HCR_EL2.NV bit.
- Behaving as if HCR_EL2.NV is 0 and HCR_EL2.NV1 is 0 for all purposes other than reading than reading back the value of the HCR_EL2.NV1 bit.
- Behaving with regard to the HCR_EL2.NV and HCR_EL2.NV1 bits behavior as defined in the rest of this description.

This bit is permitted to be cached in a TLB.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NV, bit [42]

When ARMv8.4-NV is implemented:

Nested Virtualization.

When HCR_EL2.NV2 is 1, redefines register accesses so that:

- Instructions accessing the Special purpose registers SPSR_EL2 and ELR_EL2 instead access SPSR_EL1 and ELR_EL1 respectively.
- Instructions accessing the System registers ESR_EL2 and FAR_EL2 instead access ESR_EL1 and FAR_EL1.

When HCR_EL2.NV2 is 0, or if ARMv8.4-NV is not implemented, traps functionality that is permitted at EL2 and would be UNDEFINED at EL1 if this field was 0, when EL2 is enabled in the current Security state. This applies to the following operations:

- EL1 accesses to Special-purpose registers that are not UNDEFINED at EL2.
- EL1 accesses to System registers that are not UNDEFINED at EL2.
- Execution of EL1 or EL2 translation regime address translation and TLB maintenance instructions for EL2 and above.

`0b0` When this bit is set to 0, HCR_EL2.NV2 == 0 for all purposes other than reading this register. This control does not cause any instructions to be trapped.

When HCR_EL2.NV2 is 1, no ARMv8.4-NV functionality is implemented.

`0b1` When HCR_EL2.NV2 is 0, or if ARMv8.4-NV is not implemented, EL1 accesses to the specified registers or the execution of the specified instructions are trapped to EL2, when EL2 is enabled in the current Security state. EL1 read accesses to the CurrentEL register return a value of `0x2`.

When HCR_EL2.NV2 is 1, this control redefines EL1 register accesses so that instructions accessing SPSR_EL2, ELR_EL2, ESR_EL2, and FAR_EL2 instead access SPSR_EL1, ELR_EL1, ESR_EL1, and FAR_EL1 respectively.

When HCR_EL2.NV2 is 0, or if ARMv8.4-NV is not implemented, then:

- The System or Special-purpose registers for which accesses are trapped and reported using EC syndrome value `0x18` are as follows:
  - Registers accessed using MRS or MSR with a name ending in _EL2, except SP_EL2.
  - Registers accessed using MRS or MSR with a name ending in _EL12.
  - Registers accessed using MRS or MSR with a name ending in _EL02.
  - Special-purpose registers SPSR_irq, SPSR_abt, SPSR_und and SPSR_fiq, accessed using MRS or MSR.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— Special-purpose register SP_EL1 accessed using the dedicated MRS or MSR instruction.

- The instructions for which the execution is trapped and reported using EC syndrome value `0x18` are as follows:
  — EL2 translation regime Address Translation instructions and TLB maintenance instructions.
  — EL1 translation regime Address Translation instructions and TLB maintenance instructions that are only accessible from EL2 and EL3.
- The instructions for which the execution is trapped as follows:
  — SMC in an implementation that does not include EL3 and when HCR_EL2.TSC is 1. HCR_EL2.TSC bit is not RES0 in this case. This is reported using EC syndrome value `0x17`.
  — The ERET, ERETAA, and ERETAB instructions, reported using EC syndrome value `0x1A`.

Note

The priority of this trap is higher than the priority of the HCR_EL2.API trap. If both of these bits are set so that EL1 execution of an ERETAA or ERETAB instruction is trapped to EL2, then the syndrome reported is `0x1A`.

This field resets to an architecturally UNKNOWN value.

When ARMv8.3-NV is implemented:

Nested Virtualization. Traps functionality that is permitted at EL2 and would be UNDEFINED at EL1 if this field was 0, when EL2 is enabled in the current Security state. This applies to the following operations:

- EL1 accesses to Special-purpose registers that are not UNDEFINED at EL2.
- EL1 accesses to System registers that are not UNDEFINED at EL2.
- Execution of EL1 or EL2 translation regime address translation and TLB maintenance instructions for EL2 and above.

The possible values are:

`0b0` This control does not cause any instructions to be trapped.

`0b1` EL1 accesses to the specified registers or the execution of the specified instructions are trapped to EL2, when EL2 is enabled in the current Security state. EL1 read accesses to the CurrentEL register return a value of `0x2`.

The System or Special-purpose registers for which accesses are trapped and reported using EC syndrome value `0x18` are as follows:

- Registers accessed using MRS or MSR with a name ending in _EL2, except SP_EL2.
- Registers accessed using MRS or MSR with a name ending in _EL12.
- Registers accessed using MRS or MSR with a name ending in _EL02.
- Special-purpose registers SPSR_irq, SPSR_abt, SPSR_und and SPSR_fiq, accessed using MRS or MSR.
- Special-purpose register SP_EL1 accessed using the dedicated MRS or MSR instruction.

The instructions for which the execution is trapped and reported using EC syndrome value `0x18` are as follows:

- EL2 translation regime Address Translation instructions and TLB maintenance instructions.
- EL1 translation regime Address Translation instructions and TLB maintenance instructions that are only accessible from EL2 and EL3.

The execution of the ERET, ERETAA, and ERETAB instructions are trapped and reported using EC syndrome value `0x1A`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

The priority of this trap is higher than the priority of the HCR_EL2.API trap. If both of these bits are set so that EL1 execution of an ERETAA or ERETAB instruction is trapped to EL2, then the syndrome reported is `0x1A`.

The execution of the SMC instructions in an implementation that does not include EL3 and when HCR_EL2.TSC is 1 are trapped and reported using EC syndrome value `0x17`. HCR_EL2.TSC bit is not RES0 in this case.

This bit is permitted to be cached in a TLB.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

API, bit [41]

When ARMv8.3-PAuth is implemented:

Controls the use of instructions related to Pointer Authentication:

- In EL0, when HCR_EL2.TGE==0 or HCR_EL2.E2H==0, and the associated SCTLR_EL1.En<N><M>==1.
- In EL1, the associated SCTLR_EL1.En<N><M>==1.

Traps are reported using EC syndrome value `0x09`. The Pointer Authentication instructions trapped are:

- AUTDA, AUTDB, AUTDZA, AUTDZB, AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZA, AUTIZB.
- PACGA, PACDA, PACDB, PACDZA, PACDZB, PACIA, PACIA1716, PACIASP, PACIAZ, PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZA, PACIZB.
- RETAA, RETAB, BRAA, BRAB, BLRAA, BLRAB, BRAAZ, BRABZ, BLRAAZ, BLRABZ.
- ERETAA, ERETAB, LDRAA and LDRAB.

`0b0` The instructions related to Pointer Authentication are trapped to EL2, when EL2 is enabled in the current Security state and the instructions are enabled for the EL1&0 translation regime, from:

- EL0 when HCR_EL2.TGE==0 or HCR_EL2.E2H==0.
- EL1.

If HCR_EL2.NV is 1, the HCR_EL2.NV trap takes precedence over the HCR_EL2.API trap for the ERETAA and ERETAB instructions.

`0b1` This control does not cause any instructions to be trapped.

If ARMv8.3-PAuth is implemented but EL2 is not implemented or disabled in the current Security state, the system behaves as if this bit is 1.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

APK, bit [40]

When ARMv8.3-PAuth is implemented:

Trap registers holding "key" values for Pointer Authentication. Traps accesses to the following registers from EL1 to EL2, when EL2 is enabled in the current Security state, reported using EC syndrome value `0x18`:

- APIAKeyLo_EL1, APIAKeyHi_EL1, APIBKeyLo_EL1, APIBKeyHi_EL1, APDAKeyLo_EL1, APDAKeyHi_EL1, APDBKeyLo_EL1, APDBKeyHi_EL1, APGAKeyLo_EL1, and APGAKeyHi_EL1.

`0b0` Access to the registers holding "key" values for pointer authentication from EL1 are trapped to EL2, when EL2 is enabled in the current Security state.

`0b1` This control does not cause any instructions to be trapped.

Note

If ARMv8.3-PAuth is implemented but EL2 is not implemented or is disabled in the current Security state, the system behaves as if this bit is 1.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [39]

Reserved, RES0.

MIOCNCE, bit [38]

Mismatched Inner/Outer Cacheable Non-Coherency Enable, for the EL1&0 translation regimes.

`0b0` For the EL1&0 translation regimes, for permitted accesses to a memory location that use a common definition of the Shareability and Cacheability of the location, there must be no loss of coherency if the Inner Cacheability attribute for those accesses differs from the Outer Cacheability attribute.

`0b1` For the EL1&0 translation regimes, for permitted accesses to a memory location that use a common definition of the Shareability and Cacheability of the location, there might be a loss of coherency if the Inner Cacheability attribute for those accesses differs from the Outer Cacheability attribute.

For more information see *Mismatched memory attributes* on page B2-164.

This field can be implemented as RAZ/WI.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, the PE ignores the value of this field for all purposes other than a direct read of this field.

This field resets to an architecturally UNKNOWN value.

TEA, bit [37]

Route synchronous External abort exceptions to EL2. If the RAS Extension is implemented, the possible values of this bit are:

`0b0` This control does not cause exceptions to be routed from EL0 and EL1 to EL2.

`0b1` Route synchronous External abort exceptions from EL0 and EL1 to EL2, when EL2 is enabled in the current Security state, if not routed to EL3.

When the RAS Extension is not implemented, this field is RES0.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TERR, bit [36]

When RAS is implemented:

Trap Error record accesses. Trap accesses to the RAS error registers from EL1 to EL2 as follows:

- If EL1 is using AArch64 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value `0x18`:
  - ERRIDR_EL1, ERRSELR_EL1, ERXADDR_EL1, ERXCTLR_EL1, ERXFR_EL1, ERXMISC0_EL1, ERXMISC1_EL1, and ERXSTATUS_EL1.
  - When ARMv8.4-RAS is implemented, ERXMISC2_EL1, and ERXMISC3_EL1.
- If EL1 is using AArch32 state, MCR or MRC accesses are trapped to EL2, reported using EC syndrome value `0x03`, MCRR or MRRC accesses are trapped to EL2, reported using EC syndrome value `0x04`:
  - ERRIDR, ERRSELR, ERXADDR, ERXADDR2, ERXCTLR, ERXCTLR2, ERXFR, ERXFR2, ERXMISC0, ERXMISC1, ERXMISC2, ERXMISC3, and ERXSTATUS.
  - When ARMv8.4-RAS is implemented, ERXMISC4, ERXMISC5, ERXMISC6, and ERXMISC7.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Accesses to the specified registers from EL1 generate a Trap exception to EL2, when EL2 is enabled in the current Security state.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TLOR, bit [35]

When ARMv8.1-LOR is implemented:

Trap LOR registers. Traps accesses to the LORSA_EL1, LOREA_EL1, LORN_EL1, LORC_EL1, and LORID_EL1 registers from EL1 to EL2, when EL2 is enabled in the current Security state.

`0b0` This control does not cause any instructions to be trapped.

`0b1` EL1 accesses to the LOR registers are trapped to EL2, when EL2 is enabled in the current Security state.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

E2H, bit [34]

When ARMv8.1-VHE is implemented:

EL2 Host. Enables a configuration where a Host Operating System is running in EL2, and the Host Operating System's applications are running in EL0.

`0b0` The facilities to support a Host Operating System at EL2 are disabled.

`0b1` The facilities to support a Host Operating System at EL2 are enabled.

For information on the behavior of this bit see *Behavior of HCR_EL2.E2H* on page D5-2638.

This bit is permitted to be cached in a TLB.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ID, bit [33]

Stage 2 Instruction access cacheability disable. For the EL1&0 translation regime, when EL2 is enabled in the current Security state and HCR_EL2.VM==1, this control forces all stage 2 translations for instruction accesses to Normal memory to be Non-cacheable.

`0b0` This control has no effect on stage 2 of the EL1&0 translation regime.

`0b1` Forces all stage 2 translations for instruction accesses to Normal memory to be Non-cacheable.

This bit has no effect on the EL2, EL2&0, or EL3 translation regimes.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, the PE ignores the value of this field for all purposes other than a direct read of this field.

This field resets to an architecturally UNKNOWN value.

CD, bit [32]

Stage 2 Data access cacheability disable. For the EL1&0 translation regime, when EL2 is enabled in the current Security state and HCR_EL2.VM==1, this control forces all stage 2 translations for data accesses and translation table walks to Normal memory to be Non-cacheable.

`0b0` This control has no effect on stage 2 of the EL1&0 translation regime for data accesses and translation table walks.

`0b1` Forces all stage 2 translations for data accesses and translation table walks to Normal memory to be Non-cacheable.

This bit has no effect on the EL2, EL2&0, or EL3 translation regimes.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, the PE ignores the value of this field for all purposes other than a direct read of this field.

This field resets to an architecturally UNKNOWN value.

RW, bit [31]

When AArch32 is supported at any Exception level:

Execution state control for lower Exception levels:

`0b0` Lower levels are all AArch32.

`0b1` The Execution state for EL1 is AArch64. The Execution state for EL0 is determined by the current value of PSTATE.nRW when executing at EL0.

If AArch32 state is not supported by the implementation at EL1, then this bit is RAO/WI.

In an implementation that includes EL3, when EL2 is not enabled in Secure state, the PE behaves as if this bit has the same value as the SCR_EL3.RW bit for all purposes other than a direct read or write access of HCR_EL2.

The RW bit is permitted to be cached in a TLB.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 1 for all purposes other than a direct read of the value of this bit.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAO/WI.

TRVM, bit [30]

Trap Reads of Virtual Memory controls. Traps EL1 reads of the virtual memory control registers to EL2, when EL2 is enabled in the current Security state, as follows:

- If EL1 is using AArch64 state, the following registers are trapped to EL2 and reported using EC syndrome value `0x18`.
  - SCTLR_EL1, TTBR0_EL1, TTBR1_EL1, TCR_EL1, ESR_EL1, FAR_EL1, AFSR0_EL1, AFSR1_EL1, MAIR_EL1, AMAIR_EL1, CONTEXTIDR_EL1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If EL1 is using AArch32 state, accesses using MRC to the following registers are trapped to EL2 and reported using EC syndrome value `0x03`, accesses using MRRC are trapped to EL2 and reported using EC syndrome value `0x04`:
  - SCTLR, TTBR0, TTBR1, TTBCR, TTBCR2, DACR, DFSR, IFSR, DFAR, IFAR, ADFSR, AIFSR, PRRR, NMRR, MAIR0, MAIR1, AMAIR0, AMAIR1, CONTEXTIDR.

`0b0` This control does not cause any instructions to be trapped.

`0b1` EL1 read accesses to the specified Virtual Memory controls are trapped to EL2, when EL2 is enabled in the current Security state.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

Note

EL2 provides a second stage of address translation, that a hypervisor can use to remap the address map defined by a Guest OS. In addition, a hypervisor can trap attempts by a Guest OS to write to the registers that control the memory system. A hypervisor might use this trap as part of its virtualization of memory management.

This field resets to an architecturally UNKNOWN value.

HCD, bit [29]

When EL3 is not implemented:

HVC instruction disable. Disables EL1 execution of HVC instructions, from both Execution states, when EL2 is enabled in the current Security state, reported using EC syndrome value `0x00`.

`0b0` HVC instruction execution is enabled at EL2 and EL1.

`0b1` HVC instructions are UNDEFINED at EL2 and EL1. Any resulting exception is taken to the Exception level at which the HVC instruction is executed.

Note

HVC instructions are always UNDEFINED at EL0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TDZ, bit [28]

Trap DC ZVA instructions. Traps EL0 and EL1 execution of DC ZVA instructions to EL2, when EL2 is enabled in the current Security state, from AArch64 state only, reported using EC syndrome value `0x18`.

If ARMv8.5-MemTag is implemented, this trap also applies to DC GVA and DC GZVA.

`0b0` This control does not cause any instructions to be trapped.

`0b1` In AArch64 state, any attempt to execute an instruction this trap applies to at EL1, or at EL0 when the instruction is not UNDEFINED at EL0, is trapped to EL2 when EL2 is enabled in the current Security state.

Reading the DCZID_EL0 returns a value that indicates that the instructions this trap applies to are not supported.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

This field resets to an architecturally UNKNOWN value.

TGE, bit [27]

Trap General Exceptions, from EL0.

`0b0` This control has no effect on execution at EL0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` When EL2 is not enabled in the current Security state, this control has no effect on execution at EL0.

When EL2 is enabled in the current Security state, in all cases:

- All exceptions that would be routed to EL1 are routed to EL2.
- If EL1 is using AArch64, the SCTLR_EL1.M field is treated as being 0 for all purposes other than returning the result of a direct read of SCTLR_EL1.
- If EL1 is using AArch32, the SCTLR.M field is treated as being 0 for all purposes other than returning the result of a direct read of SCTLR.
- All virtual interrupts are disabled.
- Any IMPLEMENTATION DEFINED mechanisms for signaling virtual interrupts are disabled.
- An exception return to EL1 is treated as an illegal exception return.
- The MDCR_EL2.{TDRA, TDOSA, TDA, TDE} fields are treated as being 1 for all purposes other than returning the result of a direct read of MDCR_EL2.

In addition, when EL2 is enabled in the current Security state, if:

- HCR_EL2.E2H is 0, the Effective values of the HCR_EL2.{FMO, IMO, AMO} fields are 1.
- HCR_EL2.E2H is 1, the Effective values of the HCR_EL2.{FMO, IMO, AMO} fields are 0.

For further information on the behavior of this bit when E2H is 1, see *Behavior of HCR_EL2.E2H* on page D5-2638.

HCR_EL2.TGE must not be cached in a TLB.

This field resets to an architecturally UNKNOWN value.

TVM, bit [26]

Trap Virtual Memory controls. Traps EL1 writes to the virtual memory control registers to EL2, when EL2 is enabled in the current Security state, as follows:

- If EL1 is using AArch64 state, the following registers are trapped to EL2 and reported using EC syndrome value `0x18`:
  - SCTLR_EL1, TTBR0_EL1, TTBR1_EL1, TCR_EL1, ESR_EL1, FAR_EL1, AFSR0_EL1, AFSR1_EL1, MAIR_EL1, AMAIR_EL1, CONTEXTIDR_EL1.
- If EL1 is using AArch32 state, accesses using MCR to the following registers are trapped to EL2 and reported using EC syndrome value `0x03`, accesses using MCRR are trapped to EL2 and reported using EC syndrome value `0x04`:
  - SCTLR, TTBR0, TTBR1, TTBCR, TTBCR2, DACR, DFSR, IFSR, DFAR, IFAR, ADFSR, AIFSR, PRRR, NMRR, MAIR0, MAIR1, AMAIR0, AMAIR1, CONTEXTIDR.

`0b0` This control does not cause any instructions to be trapped.

`0b1` EL1 write accesses to the specified EL1 virtual memory control registers are trapped to EL2, when EL2 is enabled in the current Security state.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TTLB, bit [25]

When ARMv8.4-TLBI is implemented:

Trap TLB maintenance instructions. Traps EL1 execution of TLB maintenance instructions to EL2, when EL2 is enabled in the current Security state, as follows:

- When EL1 is using AArch64 state, the following instructions are trapped to EL2 and reported using EC syndrome value `0x18`:
  - TLBI VMALLE1, TLBI VAE1, TLBI ASIDE1, TLBI VAAE1, TLBI VALE1, TLBI VAALE1.
  - TLBI VMALLE1IS, TLBI VAE1IS, TLBI ASIDE1IS, TLBI VAAE1IS, TLBI VALE1IS, TLBI VAALE1IS.
  - TLBI VMALLE1OS, TLBI VAE1OS, TLBI ASIDE1OS, TLBI VAAE1OS, TLBI VALE1OS, TLBI VAALE1OS.
  - TLBI RVAE1, TLBI RVAAE1, TLBI RVALE1, TLBI RVAALE1.
  - TLBI RVAE1IS, TLBI RVAAE1IS, TLBI RVALE1IS, TLBI RVAALE1IS.
  - TLBI RVAE1OS, TLBI RVAAE1OS, TLBI RVALE1OS, TLBI RVAALE1OS.
- When EL1 is using AArch32 state, the following instructions are trapped to EL2 and reported using EC syndrome value `0x03`:
  - TLBIALLIS, TLBIMVAIS, TLBIASIDIS, TLBIMVAAIS, TLBIMVALIS, TLBIMVAALIS.
  - TLBIALL, TLBIMVA, TLBIASID, TLBIMVAA, TLBIMVAL, TLBIMVAAL
  - ITLBIALL, ITLBIMVA, ITLBIASID.
  - DTLBIALL, DTLBIMVA, DTLBIASID.

`0b0` This control does not cause any instructions to be trapped.

`0b1` EL1 execution of the specified TLB maintenance instructions are trapped to EL2, when EL2 is enabled in the current Security state.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

Note

The TLB maintenance instructions are UNDEFINED at EL0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Trap TLB maintenance instructions. Traps EL1 execution of TLB maintenance instructions to EL2, when EL2 is enabled in the current Security state. This applies to the following instructions:

- When EL1 is using AArch64, TLBI VMALLE1IS, TLBI VAE1IS, TLBI ASIDE1IS, TLBI VAAE1IS, TLBI VALE1IS, TLBI VAALE1IS, TLBI VMALLE1, TLBI VAE1, TLBI ASIDE1, TLBI VAAE1, TLBI VALE1, TLBI VAALE1.
- When EL1 is using AArch32, TLBIALLIS, TLBIMVAIS, TLBIASIDIS, TLBIMVAAIS, TLBIMVALIS, TLBIMVAALIS, ITLBIALL, ITLBIMVA, ITLBIASID, DTLBIALL, DTLBIMVA, DTLBIASID, TLBIALL, TLBIMVA, TLBIASID, TLBIMVAA, TLBIMVAL, TLBIMVAAL

`0b0` This control does not cause any instructions to be trapped.

`0b1` EL1 execution of the specified TLB maintenance instructions are trapped to EL2, when EL2 is enabled in the current Security state.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TPU, bit [24]

Trap cache maintenance instructions that operate to the Point of Unification. Traps execution of those cache maintenance instructions to EL2, when EL2 is enabled in the current Security state as follows:

- If EL0 is using AArch64 state and the value of SCTLR_EL1.UCI is not 0, the following instructions are trapped to EL2 and reported with EC syndrome value `0x18`:
  - IC IVAU, DC CVAU. If the value of SCTLR_EL1.UCI is 0 these instructions are UNDEFINED at EL0 and any resulting exception is higher priority than this trap to EL2.
- If EL1 is using AArch64 state, the following instructions are trapped to EL2 and reported with EC syndrome value `0x18`:
  - IC IVAU, IC IALLU, IC IALLUIS, DC CVAU.
- If EL1 is using AArch32 state, the following instructions are trapped to EL2 and reported with EC syndrome value `0x18`:
  - ICIMVAU, ICIALLU, ICIALLUIS, DCCMVAU.

Note

An exception generated because an instruction is UNDEFINED at EL0 is higher priority than this trap to EL2. In addition:

- IC IALLUIS and IC IALLU are always UNDEFINED at EL0 using AArch64.
- ICIMVAU, ICIALLU, ICIALLUIS, and DCCMVAU are always UNDEFINED at EL0 using AArch32.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Execution of the specified instructions is trapped to EL2, when EL2 is enabled in the current Security state.

If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean by VA to the Point of Unification instruction can be trapped when the value of this control is 1.

If the Point of Unification is before any level of instruction cache, it is IMPLEMENTATION DEFINED whether the execution of any instruction cache invalidate to the Point of Unification instruction can be trapped when the value of this control is 1.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

This field resets to an architecturally UNKNOWN value.

TPCP, bit [23]

When ARMv8.2-DCPoP is implemented:

Trap data or unified cache maintenance instructions that operate to the Point of Coherency or Persistence. Traps execution of those cache maintenance instructions to EL2, when EL2 is enabled in the current Security state as follows:

- If EL0 is using AArch64 state and the value of SCTLR_EL1.UCI is not 0, the following instructions are trapped to EL2 and reported using EC syndrome value `0x18`:
  - DC CIVAC, DC CVAC, DC CVAP. If the value of SCTLR_EL1.UCI is 0 these instructions are UNDEFINED at EL0 and any resulting exception is higher priority than this trap to EL2.
- If EL1 is using AArch64 state, the following instructions are trapped to EL2 and reported using EC syndrome value `0x18`:
  - DC IVAC, DC CIVAC, DC CVAC, DC CVAP.
- If EL1 is using AArch32 state, the following instructions are trapped to EL2 and reported using EC syndrome value `0x03`:
  - DCIMVAC, DCCIMVAC, DCCMVAC.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If ARMv8.2-DCCVADP is implemented, this trap also applies to DC CVADP.

If ARMv8.5-MemTag is implemented, this trap also applies to DC CIGVAC, DC CIGDVAC, DC IGVAC, DC IGDVAC, DC CGVAC, DC CGDVAC, DC CGVAP and DC CGDVAP.

If ARMv8.2-DCCVADP and ARMv8.5-MemTag are implemented, this trap also applies to DC CGVADP and DC CGDVADP.

Note

- An exception generated because an instruction is UNDEFINED at EL0 is higher priority than this trap to EL2. In addition:
  - AArch64 instructions which invalidate by VA to the Point of Coherency are always UNDEFINED at EL0 using AArch64.
  - DCIMVAC, DCCIMVAC, and DCCMVAC are always UNDEFINED at EL0 using AArch32.
- In Armv8.0 and Armv8.1, this field is named TPC. From Armv8.2 it is named TPCP.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Execution of the specified instructions is trapped to EL2, when EL2 is enabled in the current Security state.

If the Point of Coherency is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean, invalidate, or clean and invalidate instruction that operates by VA to the point of coherency can be trapped when the value of this control is 1.

If HCR_EL2.{E2H, TGE} is set to {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Trap data or unified cache maintenance instructions that operate to the Point of Coherency. Traps execution of those cache maintenance instructions to EL2, when EL2 is enabled in the current Security state as follows:

- If EL0 is using AArch64 state and the value of SCTLR_EL1.UCI is not 0, accesses to the following registers are trapped and reported using EC syndrome value `0x18`:
  - DC CIVAC, DC CVAC. However, if the value of SCTLR_EL1.UCI is 0 these instructions are UNDEFINED at EL0 and any resulting exception is higher priority than this trap to EL2.
- If EL1 is using AArch64 state, accesses to DC IVAC, DC CIVAC, DC CVAC are trapped and reported using EC syndrome value `0x18`.
- When EL1 is using AArch32, accesses to DCIMVAC, DCCIMVAC, and DCCMVAC are trapped and reported using EC syndrome value `0x03`.

Note

- An exception generated because an instruction is UNDEFINED at EL0 is higher priority than this trap to EL2. In addition:
  - AArch64 instructions which invalidate by VA to the Point of Coherency are always UNDEFINED at EL0 using AArch64.
  - DCIMVAC, DCCIMVAC, and DCCMVAC are always UNDEFINED at EL0 using AArch32.
- In Armv8.0 and Armv8.1, this field is named TPC. From Armv8.2 it is named TPCP.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Execution of the specified instructions is trapped to EL2, when EL2 is enabled in the current Security state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the Point of Coherency is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean, invalidate, or clean and invalidate instruction that operates by VA to the point of coherency can be trapped when the value of this control is 1.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

This field resets to an architecturally UNKNOWN value.

TSW, bit [22]

Trap data or unified cache maintenance instructions that operate by Set/Way. Traps execution of those cache maintenance instructions at EL1 to EL2, when EL2 is enabled in the current Security state as follows:

- If EL1 is using AArch64 state, accesses to DC ISW, DC CSW, DC CISW are trapped to EL2, reported using EC syndrome value `0x18`.
- If EL1 is using AArch32 state, accesses to DCISW, DCCSW, DCCISW are trapped to EL2, reported using EC syndrome value `0x03`.

If ARMv8.5-MemTag is implemented, this trap also applies to DC IGSW, DC IGDSW, DC CGSW, DC CGDSW, DC CIGSW, and DC CIGDSW.

Note

An exception generated because an instruction is UNDEFINED at EL0 is higher priority than this trap to EL2, and these instructions are always UNDEFINED at EL0.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Execution of the specified instructions is trapped to EL2, when EL2 is enabled in the current Security state.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

This field resets to an architecturally UNKNOWN value.

TACR, bit [21]

Trap Auxiliary Control Registers. Traps EL1 accesses to the Auxiliary Control Registers to EL2, when EL2 is enabled in the current Security state, as follows:

- If EL1 is using AArch64 state, accesses to ACTLR_EL1 to EL2, are trapped to EL2 and reported using EC syndrome value `0x18`.
- If EL1 is using AArch32 state, accesses to ACTLR and, if implemented, ACTLR2 are trapped to EL2 and reported using EC syndrome value `0x03`.

`0b0` This control does not cause any instructions to be trapped.

`0b1` EL1 accesses to the specified registers are trapped to EL2, when EL2 is enabled in the current Security state.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

Note

ACTLR_EL1 is not accessible at EL0

ACTLR, and ACTLR2 are not accessible at EL0.

The Auxiliary Control Registers are IMPLEMENTATION DEFINED registers that might implement global control bits for the PE.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TIDCP, bit [20]

Trap IMPLEMENTATION DEFINED functionality. Traps EL1 accesses to the encodings reserved for IMPLEMENTATION DEFINED functionality to EL2, when EL2 is enabled in the current Security state as follows:

- In AArch64 state, access to any of the encodings in the following reserved encoding spaces are trapped and reported using EC syndrome `0x18`:
  - IMPLEMENTATION DEFINED System instructions, which are accessed using SYS and SYSL, with CRn == {11, 15}.
  - IMPLEMENTATION DEFINED System registers, which are accessed using MRS and MSR with the S3_<op1>_<Cn>_<Cm>_<op2> register name.
- In AArch32 state, MCR and MRC access to instructions with the following encodings are trapped and reported using EC syndrome `0x03`:
  - All coproc==p15, CRn==c9, opc1 == {0-7}, CRm == {c0-c2, c5-c8}, opc2 == {0-7}.
  - All coproc==p15, CRn==c10, opc1 ==={0-7}, CRm == {c0, c1, c4, c8}, opc2 == {0-7}.
  - All coproc==p15, CRn==c11, opc1==={0-7}, CRm == {c0-c8, c15}, opc2 == {0-7}.

When the value of HCR_EL2.TIDCP is 1, it is IMPLEMENTATION DEFINED whether any of this functionality accessed from EL0 is trapped to EL2. If it is not, then it is UNDEFINED, and any attempt to access it from EL0 generates an exception that is taken to EL1.

`0b0` This control does not cause any instructions to be trapped.

`0b1` EL1 accesses to or execution of the specified encodings reserved for IMPLEMENTATION DEFINED functionality are trapped to EL2, when EL2 is enabled in the current Security state.

An implementation can also include IMPLEMENTATION DEFINED registers that provide additional controls, to give finer-grained control of the trapping of IMPLEMENTATION DEFINED features.

Note

Arm expects the trapping of EL0 accesses to these functions to EL2 to be unusual, and used only when the hypervisor is virtualizing EL0 operation. Arm strongly recommends that unless the hypervisor must virtualize EL0 operation, an EL0 access to any of these functions is UNDEFINED, as it would be if the implementation did not include EL2. The PE then takes any resulting exception to EL1.

The trapping of accesses to these registers from EL1 is higher priority than an exception resulting from the register access being UNDEFINED.

This field resets to an architecturally UNKNOWN value.

TSC, bit [19]

Trap `SMC` instructions. Traps EL1 execution of `SMC` instructions to EL2, when EL2 is enabled in the current Security state.

If execution is in AArch64 state the trap is reported using EC syndrome value `0x17`.

If execution is in AArch32 state, the trap is reported using EC syndrome value `0x13`.

Note

HCR_EL2.TSC traps execution of the `SMC` instruction. It is not a routing control for the `SMC` exception. Trap exceptions and `SMC` exceptions have different preferred return addresses.

`0b0` This control does not cause any instructions to be trapped.

`0b1` If EL3 is implemented, then any attempt to execute an `SMC` instruction at EL1 is trapped to EL2, when EL2 is enabled in the current Security state, regardless of the value of SCR_EL3.SMD.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If EL3 is not implemented, ARMv8.3-NV is implemented, and HCR_EL2.NV is 1, then any attempt to execute an SMC instruction at EL1 using AArch64 is trapped to EL2, when EL2 is enabled in the current Security state.

If EL3 is not implemented, and either ARMv8.3-NV is not implemented or HCR_EL2.NV is 0, then it is IMPLEMENTATION DEFINED whether:

- Any attempt to execute an SMC instruction at EL1 is trapped to EL2, when EL2 is enabled in the current Security state.
- Any attempt to execute an SMC instruction is UNDEFINED.

In AArch32 state, the Armv8-A architecture permits, but does not require, this trap to apply to conditional `SMC` instructions that fail their condition code check, in the same way as with traps on other conditional instructions.

`SMC` instructions are UNDEFINED at EL0.

If EL3 is not implemented and HCR_EL2.NV is 0, it is IMPLEMENTATION DEFINED whether this bit is:

- RES0.
- Implemented with the functionality as described in HCR_EL2.TSC.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

This field resets to an architecturally UNKNOWN value.

TID3, bit [18]

Trap ID group 3. Traps EL1 reads of group 3 ID registers to EL2, when EL2 is enabled in the current Security state, as follows:

- In AArch64 state, reads of the following registers are trapped to EL2, reported using EC syndrome value `0x18`:
  - ID_PFR0_EL1, ID_PFR1_EL1, ID_DFR0_EL1, ID_AFR0_EL1, ID_MMFR0_EL1, ID_MMFR1_EL1, ID_MMFR2_EL1, ID_MMFR3_EL1, ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, ID_ISAR4_EL1, ID_ISAR5_EL1, ID_ISAR6_EL1, MVFR0_EL1, MVFR1_EL1, MVFR2_EL1.
  - ID_AA64PFR0_EL1, ID_AA64PFR1_EL1, ID_AA64DFR0_EL1, ID_AA64DFR1_EL1, ID_AA64ISAR0_EL1, ID_AA64ISAR1_EL1, ID_AA64MMFR0_EL1, ID_AA64MMFR1_EL1, ID_AA64MMFR2_EL1, ID_AA64AFR0_EL1, ID_AA64AFR1_EL1, ID_AA64ZFR0_EL1 (where SVE is implemented), and ID_MMFR4_EL1
- In AArch64 state, ID_MMFR4_EL1 and ID_MMFR5_EL1 are trapped to EL2, unless implemented as RAZ, when it is IMPLEMENTATION DEFINED whether accesses to ID_MMFR4_EL1 or ID_MMFR5_EL1 are trapped to EL2, reported using EC syndrome value `0x18`.
- In AArch64 state, ID_AA64MMFR2_EL1 and ID_ISAR6_EL1 are trapped to EL2, unless implemented as RAZ, when it is IMPLEMENTATION DEFINED whether accesses to ID_AA64MMFR2_EL1 or ID_ISAR6_EL1 are trapped to EL2, reported using EC syndrome value `0x18`.
- In AArch64 state, ID_DFR1_EL1 is trapped to EL2, unless implemented as RAZ, when it is IMPLEMENTATION DEFINED whether accesses to ID_DFR1_EL1 are trapped to EL2, reported using EC syndrome value `0x18`.
- In AArch64 state, ID_AA64ZFR0_EL1 is trapped to EL2, unless implemented as RAZ then it is IMPLEMENTATION DEFINED whether accesses to ID_AA64ZFR0_EL1 are trapped to EL2, reported using EC syndrome value `0x18`.
- In AArch64 state, if ARMv8.6-FGT is implemented, this field traps all MRS accesses to encodings in the following range that are not already mentioned in this field description, reported using EC syndrome value `0x18`:
  - Op0 == 3, op1 == 0, CRn == c0, CRm == {c1-c7}, op2 == {0-7}.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Otherwise, in AArch64 state, it is IMPLEMENTATION DEFINED whether this field traps MRS accesses to encodings in the following range that are not already mentioned in this field description, reported using EC syndrome value `0x18`:
  - Op0 == 3, op1 == 0, CRn == c0, CRm == {c2-c7}, op2 == {0-7}.
- In AArch32 state, MRC access to the following registers are trapped to EL2, reported using EC syndrome value `0x03`:
  - ID_PFR0, ID_PFR1, ID_DFR0, ID_AFR0, ID_MMFR0, ID_MMFR1, ID_MMFR2, ID_MMFR3, ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR3, ID_ISAR4, ID_ISAR5, MVFR0, MVFR1, MVFR2, and ID_MMFR4.
- In AArch32 state, VMRS access to MVFR0, MVFR1, and MVFR2, reported using EC syndrome value `0x08`.
- In AArch32 state, ID_MMFR4 and ID_MMFR5 are trapped to EL2, unless implemented as RAZ, when it is IMPLEMENTATION DEFINED whether accesses to ID_MMFR4 or ID_MMFR5 are trapped to EL2, reported using EC syndrome value `0x03`.
- In AArch32 state, ID_ISAR6 is trapped to EL2, unless implemented as RAZ, when it is IMPLEMENTATION DEFINED whether accesses to ID_ISAR6 are trapped to EL2, reported using EC syndrome value `0x03`.
- In AArch32 state, ID_DFR1 is trapped to EL2, unless implemented as RAZ, when it is IMPLEMENTATION DEFINED whether accesses to ID_DFR1 are trapped to EL2, reported using EC syndrome value `0x03`.
- In AArch32 state, if ARMv8.6-FGT is implemented, this field traps all MRC accesses to encodings in the following range that are not already mentioned in this field description, reported using EC syndrome value `0x03`:
  - coproc==p15, opc1 == 0, CRn == c0, CRm == {c1-c7}, opc2 == {0-7}.
- Otherwise, in AArch32 state, it is IMPLEMENTATION DEFINED whether this bit traps MRC accesses to encodings that are not already mentioned, with coproc==p15, opc1 == 0, CRn == c0, CRm == {c2-c7}, opc2 == {0-7}.

`0b0` This control does not cause any instructions to be trapped.

`0b1` The specified EL1 read accesses to ID group 3 registers are trapped to EL2, when EL2 is enabled in the current Security state.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

This field resets to an architecturally UNKNOWN value.

TID2, bit [17]

Trap ID group 2. Traps the following register accesses to EL2, when EL2 is enabled in the current Security state, as follows:

- If EL1 is using AArch64, reads of CTR_EL0, CCSIDR_EL1, CCSIDR2_EL1, CLIDR_EL1, and CSSELR_EL1 are trapped to EL2, reported using EC syndrome value `0x18`.
- If EL0 is using AArch64 and the value of SCTLR_EL1.UCT is not 0, reads of CTR_EL0 are trapped to EL2, reported using EC syndrome value `0x18`. If the value of SCTLR_EL1.UCT is 0 then EL0 reads of CTR_EL0 are UNDEFINED and any resulting exception takes precedence over this trap.
- If EL1 is using AArch64, writes to CSSELR_EL1 are trapped to EL2, reported using EC syndrome value `0x18`.
- If EL1 is using AArch32, reads of CTR, CCSIDR, CCSIDR2, CLIDR, and CSSELR are trapped to EL2, reported using EC syndrome value `0x03`.
- If EL1 is using AArch32, writes to CSSELR are trapped to EL2, reported using EC syndrome value `0x03`.

`0b0` This control does not cause any instructions to be trapped.

`0b1` The specified EL1 and EL0 accesses to ID group 2 registers are trapped to EL2, when EL2 is enabled in the current Security state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

This field resets to an architecturally UNKNOWN value.

TID1, bit [16]

Trap ID group 1. Traps EL1 reads of the following registers to EL2, when EL2 is enabled in the current Security state as follows:

- In AArch64 state, accesses of REVIDR_EL1, AIDR_EL1, reported using EC syndrome value `0x18`.
- In AArch32 state, accesses of TCMTR, TLBTR, REVIDR, AIDR, reported using EC syndrome value `0x03`.

`0b0` This control does not cause any instructions to be trapped.

`0b1` The specified EL1 read accesses to ID group 1 registers are trapped to EL2, when EL2 is enabled in the current Security state.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

This field resets to an architecturally UNKNOWN value.

TID0, bit [15]

When AArch32 is supported at any Exception level:

Trap ID group 0. Traps the following register accesses to EL2:

- EL1 reads of the JIDR, reported using EC syndrome value `0x05`.
- If the JIDR is RAZ from EL0, EL0 reads of the JIDR, reported using EC syndrome value `0x05`.
- EL1 accesses using VMRS of the FPSID, reported using EC syndrome value `0x08`.

Note

- It is IMPLEMENTATION DEFINED whether the JIDR is RAZ or UNDEFINED at EL0. If it is UNDEFINED at EL0 then any resulting exception takes precedence over this trap.
- The FPSID is not accessible at EL0 using AArch32.
- Writes to the FPSID are ignored, and not trapped by this control.

`0b0` This control does not cause any instructions to be trapped.

`0b1` The specified EL1 read accesses to ID group 0 registers are trapped to EL2, when EL2 is enabled in the current Security state.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TWE, bit [14]

Traps EL0 and EL1 execution of WFE instructions to EL2, when EL2 is enabled in the current Security state, from both Execution states, reported using EC syndrome value `0x01`.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Any attempt to execute a WFE instruction at EL0 or EL1 is trapped to EL2, when EL2 is enabled in the current Security state, if the instruction would otherwise have caused the PE to enter a low-power state and it is not trapped by SCTLR.nTWE or SCTLR_EL1.nTWE.

In AArch32 state, the attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

Since a WFE can complete at any time, even without a Wakeup event, the traps on WFE are not guaranteed to be taken, even if the WFE is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

For more information about when WFE instructions can cause the PE to enter a low-power state, see *Wait for Event mechanism and Send event* on page D1-2397.

This field resets to an architecturally UNKNOWN value.

TWI, bit [13]

Traps EL0 and EL1 execution of WFI instructions to EL2, when EL2 is enabled in the current Security state, from both Execution states, reported using EC syndrome value `0x01`.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Any attempt to execute a WFI instruction at EL0 or EL1 is trapped to EL2, when EL2 is enabled in the current Security state, if the instruction would otherwise have caused the PE to enter a low-power state and it is not trapped by SCTLR.nTWI or SCTLR_EL1.nTWI.

In AArch32 state, the attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

Note

Since a WFI can complete at any time, even without a Wakeup event, the traps on WFI are not guaranteed to be taken, even if the WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

For more information about when WFI instructions can cause the PE to enter a low-power state, see *Wait For Interrupt* on page D1-2400.

This field resets to an architecturally UNKNOWN value.

DC, bit [12]

Default Cacheability.

`0b0` This control has no effect on the EL1&0 translation regime.

`0b1` In both Security states:

- When EL1 is using AArch64, the PE behaves as if the value of the SCTLR_EL1.M field is 0 for all purposes other than returning the value of a direct read of SCTLR_EL1.
- When EL1 is using AArch32, the PE behaves as if the value of the SCTLR.M field is 0 for all purposes other than returning the value of a direct read of SCTLR.
- The PE behaves as if the value of the HCR_EL2.VM field is 1 for all purposes other than returning the value of a direct read of HCR_EL2.
- The memory type produced by stage 1 of the EL1&0 translation regime is Normal Non-Shareable, Inner Write-Back Read-Allocate Write-Allocate, Outer Write-Back Read-Allocate Write-Allocate.

This field has no effect on the EL2, EL2&0, and EL3 translation regimes.

This field is permitted to be cached in a TLB.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this field.

This field resets to an architecturally UNKNOWN value.

BSU, bits [11:10]

Barrier Shareability upgrade. This field determines the minimum shareability domain that is applied to any barrier instruction executed from EL1 or EL0:

`0b00` No effect.

`0b01` Inner Shareable.

`0b10` Outer Shareable.

`0b11` Full system.

This value is combined with the specified level of the barrier held in its instruction, using the same principles as combining the shareability attributes from two stages of address translation.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as `0b00` for all purposes other than a direct read of the value of this bit.

This field resets to an architecturally UNKNOWN value.

FB, bit [9]

Force broadcast. Causes the following instructions to be broadcast within the Inner Shareable domain when executed from EL1:

AArch32: BPIALL, TLBIALL, TLBIMVA, TLBIASID, DTLBIALL, DTLBIMVA, DTLBIASID, ITLBIALL, ITLBIMVA, ITLBIASID, TLBIMVAA, ICIALLU, TLBIMVAL, TLBIMVAAL.

AArch64: TLBI VMALLE1, TLBI VAE1, TLBI ASIDE1, TLBI VAAE1, TLBI VALE1, TLBI VAALE1, IC IALLU, TLBI RVAE1, TLBI RVAAE1, TLBI RVALE1, TLBI RVAALE1.

`0b0` This field has no effect on the operation of the specified instructions.

`0b1` When one of the specified instruction is executed at EL1, the instruction is broadcast within the Inner Shareable shareability domain.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

This field resets to an architecturally UNKNOWN value.

VSE, bit [8]

Virtual SError interrupt.

`0b0` This mechanism is not making a virtual SError interrupt pending.

`0b1` A virtual SError interrupt is pending because of this mechanism.

The virtual SError interrupt is only enabled when the value of HCR_EL2.{TGE, AMO} is {0, 1}.

This field resets to an architecturally UNKNOWN value.

VI, bit [7]

Virtual IRQ Interrupt.

`0b0` This mechanism is not making a virtual IRQ pending.

`0b1` A virtual IRQ is pending because of this mechanism.

The virtual IRQ is enabled only when the value of HCR_EL2.{TGE, IMO} is {0, 1}.

This field resets to an architecturally UNKNOWN value.

VF, bit [6]

Virtual FIQ Interrupt.

`0b0` This mechanism is not making a virtual FIQ pending.

`0b1` A virtual FIQ is pending because of this mechanism.

The virtual FIQ is enabled only when the value of HCR_EL2.{TGE, FMO} is {0, 1}.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

AMO, bit [5]

Physical SError interrupt routing.

`0b0` When executing at Exception levels below EL2, and EL2 is enabled in the current Security state:

- Physical SError interrupts are not taken to EL2.
- When the value of HCR_EL2.TGE is 0, if the PE is executing at EL2 using AArch64, physical SError interrupts are not taken unless they are routed to EL3 by the SCR_EL3.EA bit.
- Virtual SError interrupts are disabled.

`0b1` When executing at any Exception level, and EL2 is enabled in the current Security state:

- Physical SError interrupts are taken to EL2, unless they are routed to EL3.
- When the value of HCR_EL2.TGE is 0, then virtual SError interrupts are enabled.

If EL2 is enabled in the current Security state and the value of HCR_EL2.TGE is 1:

- Regardless of the value of the AMO bit physical asynchronous External aborts and SError interrupts target EL2 unless they are routed to EL3.
- When ARMv8.1-VHE is not implemented, or if HCR_EL2.E2H is 0, this field behaves as 1 for all purposes other than a direct read of the value of this bit.
- When ARMv8.1-VHE is implemented and HCR_EL2.E2H is 1, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

For more information, see *Asynchronous exception routing* on page D1-2364.

This field resets to an architecturally UNKNOWN value.

IMO, bit [4]

Physical IRQ Routing.

`0b0` When executing at Exception levels below EL2, and EL2 is enabled in the current Security state:

- Physical IRQ interrupts are not taken to EL2.
- When the value of HCR_EL2.TGE is 0, if the PE is executing at EL2 using AArch64, physical IRQ interrupts are not taken unless they are routed to EL3 by the SCR_EL3.IRQ bit.
- Virtual IRQ interrupts are disabled.

`0b1` When executing at any Exception level, and EL2 is enabled in the current Security state:

- Physical IRQ interrupts are taken to EL2, unless they are routed to EL3.
- When the value of HCR_EL2.TGE is 0, then Virtual IRQ interrupts are enabled.

If EL2 is enabled in the current Security state, and the value of HCR_EL2.TGE is 1:

- Regardless of the value of the IMO bit, physical IRQ Interrupts target EL2 unless they are routed to EL3.
- When ARMv8.1-VHE is not implemented, or if HCR_EL2.E2H is 0, this field behaves as 1 for all purposes other than a direct read of the value of this bit.
- When ARMv8.1-VHE is implemented and HCR_EL2.E2H is 1, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

For more information, see *Asynchronous exception routing* on page D1-2364.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

FMO, bit [3]

Physical FIQ Routing.

`0b0` When executing at Exception levels below EL2, and EL2 is enabled in the current Security state:

- Physical FIQ interrupts are not taken to EL2.
- When the value of HCR_EL2.TGE is 0, if the PE is executing at EL2 using AArch64, physical FIQ interrupts are not taken unless they are routed to EL3 by the SCR_EL3.FIQ bit.
- Virtual FIQ interrupts are disabled.

`0b1` When executing at any Exception level, and EL2 is enabled in the current Security state:

- Physical FIQ interrupts are taken to EL2, unless they are routed to EL3.
- When HCR_EL2.TGE is 0, then Virtual FIQ interrupts are enabled.

If EL2 is enabled in the current Security state and the value of HCR_EL2.TGE is 1:

- Regardless of the value of the FMO bit, physical FIQ Interrupts target EL2 unless they are routed to EL3.
- When ARMv8.1-VHE is not implemented, or if HCR_EL2.E2H is 0, this field behaves as 1 for all purposes other than a direct read of the value of this bit.
- When ARMv8.1-VHE is implemented and HCR_EL2.E2H is 1, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

For more information, see *Asynchronous exception routing* on page D1-2364.

This field resets to an architecturally UNKNOWN value.

PTW, bit [2]

Protected Table Walk. In the EL1&0 translation regime, a translation table access made as part of a stage 1 translation table walk is subject to a stage 2 translation. The combining of the memory type attributes from the two stages of translation means the access might be made to a type of Device memory. If this occurs, then the value of this bit determines the behavior:

`0b0` The translation table walk occurs as if it is to Normal Non-cacheable memory. This means it can be made speculatively.

`0b1` The memory access generates a stage 2 Permission fault.

This field is permitted to be cached in a TLB.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

This field resets to an architecturally UNKNOWN value.

SWIO, bit [1]

Set/Way Invalidation Override. Causes EL1 execution of the data cache invalidate by set/way instructions to perform a data cache clean and invalidate by set/way:

`0b0` This control has no effect on the operation of data cache invalidate by set/way instructions.

`0b1` Data cache invalidate by set/way instructions perform a data cache clean and invalidate by set/way.

When the value of this bit is 1:

AArch32: DCISW performs the same invalidation as a DCCISW instruction.

AArch64: DC ISW performs the same invalidation as a DC CISW instruction.

This bit can be implemented as RES1.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

VM, bit [0]

Virtualization enable. Enables stage 2 address translation for the EL1&0 translation regime, when EL2 is enabled in the current Security state.

`0b0` EL1&0 stage 2 address translation disabled.

`0b1` EL1&0 stage 2 address translation enabled.

When the value of this bit is 1, data cache invalidate instructions executed at EL1 perform a data cache clean and invalidate. For the invalidate by set/way instruction this behavior applies regardless of the value of the HCR_EL2.SWIO bit.

This bit is permitted to be cached in a TLB.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

This field resets to an architecturally UNKNOWN value.

Accessing the HCR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, HCR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        return NVMem[0x078];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return HCR_EL2;
elsif PSTATE.EL == EL3 then
    return HCR_EL2;
```

MSR HCR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NVMem[0x078] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    HCR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    HCR_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.48 HDFGRTR_EL2, Hypervisor Debug Fine-Grained Read Trap Register

The HDFGRTR_EL2 characteristics are:

Purpose

Provides controls for traps of `MRS` and `MRC` reads of debug, trace, PMU, and Statistical Profiling System registers.

Configurations

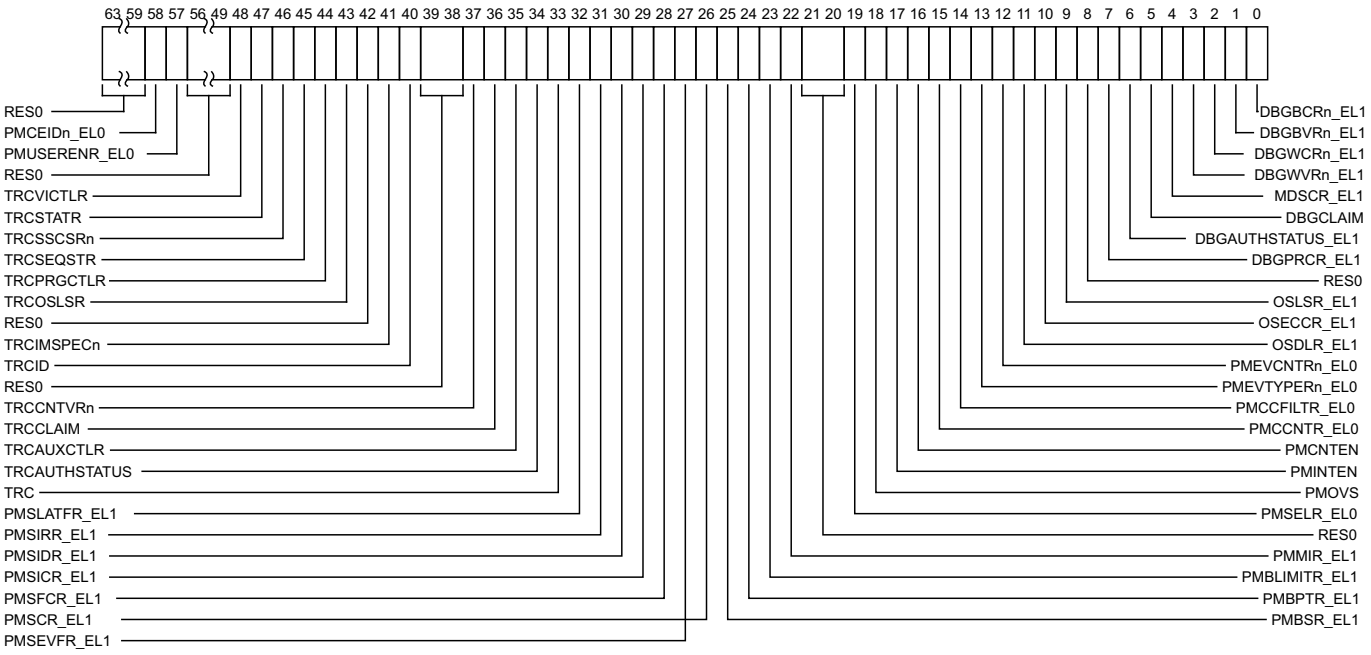
This register is present only when ARMv8.6-FGT is implemented. Otherwise, direct accesses to HDFGRTR_EL2 are UNDEFINED.

Attributes

HDFGRTR_EL2 is a 64-bit register.

Field descriptions

The HDFGRTR_EL2 bit assignments are:



Bits [63:59]

Reserved, RES0.

PMCEIDn_EL0, bit [58]

When PMUv3 is implemented:

Trap `MRS` reads of PMCEID<n>_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of PMCEID<n> at EL0 using AArch32 when EL1 is using AArch64 to EL2.

`0b0` `MRS` reads of PMCEID<n>_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of PMCEID<n> at EL0 using AArch32 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- `MRS` reads of PMCEID<n>_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- MRC reads of PMCEID<n> at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

PMUSERENR_EL0, bit [57]

When PMUv3 is implemented:

Trap MRS reads of PMUSERENR_EL0 at EL1 and EL0 using AArch64 and MRC reads of PMUSERENR at EL0 using AArch32 when EL1 is using AArch64 to EL2.

0b0 MRS reads of PMUSERENR_EL0 at EL1 and EL0 using AArch64 and MRC reads of PMUSERENR at EL0 using AArch32 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- MRS reads of PMUSERENR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.
- MRC reads of PMUSERENR at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

Bits [56:49]

Reserved, RES0.

TRCVICTLR, bit [48]

When the Trace Extension is implemented and System register access to the PE Trace Unit registers is implemented:

Trap MRS reads of TRCVICTLR at EL1 using AArch64 to EL2.

0b0 MRS reads of TRCVICTLR are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of TRCVICTLR at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

TRCSTATR, bit [47]

When the Trace Extension is implemented and System register access to the PE Trace Unit registers is implemented:

Trap MRS reads of TRCSTATR at EL1 using AArch64 to EL2.

0b0 MRS reads of TRCSTATR are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of TRCSTATR at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TRCSSCSRn, bit [46]

When the Trace Extension is implemented and System register access to the PE Trace Unit registers is implemented:

Trap MRS reads of TRCSSCSR<n> at EL1 using AArch64 to EL2.

0b0 MRS reads of TRCSSCSR<n> are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of TRCSSCSR<n> at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

If Single-shot Comparator n is not implemented, a read of TRCSSCSR<n> is UNDEFINED.

This bit is RES0 if TRCSSCSR<n> are not implemented.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

TRCSEQSTR, bit [45]

When the Trace Extension is implemented and System register access to the PE Trace Unit registers is implemented:

Trap MRS reads of TRCSEQSTR at EL1 using AArch64 to EL2.

0b0 MRS reads of TRCSEQSTR are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of TRCSEQSTR at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

This bit is RES0 if TRCSEQSTR is not implemented.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

TRCPRGCTLR, bit [44]

When the Trace Extension is implemented and System register access to the PE Trace Unit registers is implemented:

Trap MRS reads of TRCPRGCTLR at EL1 using AArch64 to EL2.

0b0 MRS reads of TRCPRGCTLR are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of TRCPRGCTLR at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

TRCOSLSR, bit [43]

When the Trace Extension is implemented and System register access to the PE Trace Unit registers is implemented:

Trap MRS reads of TRCOSLSR at EL1 using AArch64 to EL2.

0b0 MRS reads of TRCOSLSR are not affected by this bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads of TRCOSLSR at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

Bit [42]

Reserved, RES0.

TRCIMSPECn, bit [41]

When the Trace Extension is implemented and System register access to the PE Trace Unit registers is implemented:

Trap `MRS` reads of TRCIMSPEC<n> at EL1 using AArch64 to EL2.

`0b0` `MRS` reads of TRCIMSPEC<n> are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads of TRCIMSPEC<n> at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

TRCIMSPEC<1-7> are optional. If TRCIMSPEC<n> is not implemented, a read of TRCIMSPEC<n> is UNDEFINED.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

TRCID, bit [40]

When the Trace Extension is implemented and System register access to the PE Trace Unit registers is implemented:

Trap `MRS` reads of multiple System registers. Enables a trap on `MRS` reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- TRCDEVARCH.
- TRCDEVID.
- TRCIDR<n>.

`0b0` `MRS` reads of the System registers listed above are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

Bits [39:38]

Reserved, RES0.

TRCCNTVRn, bit [37]

When the Trace Extension is implemented and System register access to the PE Trace Unit registers is implemented:

Trap `MRS` reads of TRCCNTVR<n> at EL1 using AArch64 to EL2.

`0b0` `MRS` reads of TRCCNTVR<n> are not affected by this bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads of TRCCNTVR<n> at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

If Counter n is not implemented, a read of TRCCNTVR<n> is UNDEFINED.

This bit is RES0 if TRCCNTVR<n> are not implemented.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

TRCCLAIM, bit [36]

When the Trace Extension is implemented and System register access to the PE Trace Unit registers is implemented:

Trap `MRS` reads of multiple System registers. Enables a trap on `MRS` reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- TRCCLAIMCLR.
- TRCCLAIMSET.

`0b0` `MRS` reads of the System registers listed above are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

TRCAUXCTLR, bit [35]

When the Trace Extension is implemented and System register access to the PE Trace Unit registers is implemented:

Trap `MRS` reads of TRCAUXCTLR at EL1 using AArch64 to EL2.

`0b0` `MRS` reads of TRCAUXCTLR are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads of TRCAUXCTLR at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

TRCAUTHSTATUS, bit [34]

When the Trace Extension is implemented and System register access to the PE Trace Unit registers is implemented:

Trap `MRS` reads of TRCAUTHSTATUS at EL1 using AArch64 to EL2.

`0b0` `MRS` reads of TRCAUTHSTATUS are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads of TRCAUTHSTATUS at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

Reserved, RES0.

TRC, bit [33]

When the Trace Extension is implemented and System register access to the PE Trace Unit registers is implemented:

Trap MRS reads of multiple System registers. Enables a trap on MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- TRCACATR<n>.
- TRCACVR<n>.
- TRCBBCTLR.
- TRCCCCTLR.
- TRCCIDCCTLR<n>.
- TRCCIDCVR<n>.
- TRCCNTCTLR<n>.
- TRCCNTRLDVR<n>.
- TRCCONFIGR.
- TRCEVENTCTL0R.
- TRCEVENTCTL1R.
- TRCEXTINSELR.
- TRCQCTLR.
- TRCRSCTLR<n>.
- TRCSEQEVR<n>.
- TRCSEQRSTEVR.
- TRCSSCCR<n>.
- TRCSSPCICR<n>.
- TRCSTALLCTLR.
- TRCSYNCPR.
- TRCTRACEIDR.
- TRCTSCTLR.
- TRCVIIECTLR.
- TRCVIPCSSCTLR.
- TRCVISSCTLR.
- TRCVMIDCCTLR<n>.
- TRCVMIDCVR<n>.

`0b0` MRS reads of the System registers listed above are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

A read of an unimplemented register is UNDEFINED.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

PMSLATFR_EL1, bit [32]

When SPE is implemented:

Trap MRS reads of PMSLATFR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of PMSLATFR_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of PMSLATFR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

PMSIRR_EL1, bit [31]

When SPE is implemented:

Trap MRS reads of PMSIRR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of PMSIRR_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of PMSIRR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

PMSIDR_EL1, bit [30]

When SPE is implemented:

Trap MRS reads of PMSIDR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of PMSIDR_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of PMSIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

PMSICR_EL1, bit [29]

When SPE is implemented:

Trap MRS reads of PMSICR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of PMSICR_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of PMSICR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

PMSFCR_EL1, bit [28]

When SPE is implemented:

Trap MRS reads of PMSFCR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of PMSFCR_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of PMSFCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

PMSEVFR_EL1, bit [27]

When SPE is implemented:

Trap MRS reads of PMSEVFR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of PMSEVFR_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of PMSEVFR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

PMSCR_EL1, bit [26]

When SPE is implemented:

Trap MRS reads of PMSCR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of PMSCR_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of PMSCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

PMBSR_EL1, bit [25]

When SPE is implemented:

Trap MRS reads of PMBSR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of PMBSR_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of PMBSR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

PMBPTR_EL1, bit [24]

When SPE is implemented:

Trap MRS reads of PMBPTR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of PMBPTR_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of PMBPTR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

PMBLIMITR_EL1, bit [23]

When SPE is implemented:

Trap MRS reads of PMBLIMITR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of PMBLIMITR_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of PMBLIMITR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

PMMIR_EL1, bit [22]

When PMUv3 is implemented:

Trap MRS reads of PMMIR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of PMMIR_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of PMMIR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

Bits [21:20]

Reserved, RES0.

PMSELR_EL0, bit [19]

When PMUv3 is implemented:

Trap MRS reads of PMSELR_EL0 at EL1 and EL0 using AArch64 and MRC reads of PMSELR at EL0 using AArch32 when EL1 is using AArch64 to EL2.

0b0 MRS reads of PMSELR_EL0 at EL1 and EL0 using AArch64 and MRC reads of PMSELR at EL0 using AArch32 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- MRS reads of PMSELR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- `MRC` reads of PMSELR at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value `0x03`.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

PMOVS, bit [18]

When PMUv3 is implemented:

Trap `MRS` reads and `MRC` reads of multiple System registers.

Enables a trap to EL2 the following operations:

- At EL1 and EL0 using AArch64: `MRS` reads of PMOVSCLR_EL0 and PMOVSSET_EL0.
- At EL0 using Arch32 when EL1 is using AArch64: `MRC` reads of PMOVSR and PMOVSSET.

`0b0` The operations listed above are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- `MRS` reads at EL1 and EL0 using AArch64 of PMOVSCLR_EL0 and PMOVSSET_EL0 are trapped to EL2 and reported with EC syndrome value `0x18`.
- `MRC` reads at EL0 using AArch32 of PMOVSR and PMOVSSET are trapped to EL2 and reported with EC syndrome value `0x03`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

PMINTEN, bit [17]

When PMUv3 is implemented:

Trap `MRS` reads of multiple System registers. Enables a trap on `MRS` reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- PMINTENCLR_EL1.
- PMINTENSET_EL1.

`0b0` `MRS` reads of the System registers listed above are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

PMCNTEN, bit [16]

When PMUv3 is implemented:

Trap `MRS` reads and `MRC` reads of multiple System registers.

Enables a trap to EL2 the following operations:

- At EL1 and EL0 using AArch64: `MRS` reads of PMCNTENCLR_EL0 and PMCNTENSET_EL0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- At EL0 using Arch32 when EL1 is using AArch64: MRC reads of PMCNTENCLR and PMCNTENSET.

0b0 The operations listed above are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- MRS reads at EL1 and EL0 using AArch64 of PMCNTENCLR_EL0 and PMCNTENSET_EL0 are trapped to EL2 and reported with EC syndrome value 0x18.
- MRC reads at EL0 using AArch32 of PMCNTENCLR and PMCNTENSET are trapped to EL2 and reported with EC syndrome value 0x03, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

PMCCNTR_EL0, bit [15]

When PMUv3 is implemented:

Trap MRS reads of PMCCNTR_EL0 at EL1 and EL0 using AArch64 and MRC and MRRC reads of PMCCNTR at EL0 using AArch32 when EL1 is using AArch64 to EL2.

0b0 MRS reads of PMCCNTR_EL0 at EL1 and EL0 using AArch64 and MRC and MRRC reads of PMCCNTR at EL0 using AArch32 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- MRS reads of PMCCNTR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.
- MRC and MRRC reads of PMCCNTR at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03 (for MRC) or 0x04 (for MRRC).

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

PMCCFILTR_EL0, bit [14]

When PMUv3 is implemented:

Trap MRS reads of PMCCFILTR_EL0 at EL1 and EL0 using AArch64 and MRC reads of PMCCFILTR at EL0 using AArch32 when EL1 is using AArch64 to EL2.

0b0 MRS reads of PMCCFILTR_EL0 at EL1 and EL0 using AArch64 and MRC reads of PMCCFILTR at EL0 using AArch32 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- MRS reads of PMCCFILTR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.
- MRC reads of PMCCFILTR at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

PMCCFILTR_EL0 can also be accessed in AArch64 state using PMXEVTYPER_EL0 when PMSELR_EL0.SEL == 31, and PMCCFILTR can also be accessed in AArch32 state using PMXEVTYPER when PMSELR.SEL == 31.

Setting this bit to 1 has no effect on accesses to PMXEVTYPER_EL0 and PMXEVTYPER, regardless of the value of PMSELR_EL0.SEL or PMSELR.SEL.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

PMEVTYPERn_EL0, bit [13]

When PMUv3 is implemented:

Trap `MRS` reads and `MRC` reads of multiple System registers.

Enables a trap to EL2 the following operations:

- At EL1 and EL0 using AArch64: `MRS` reads of PMEVTYPER<n>_EL0 and PMXEVTYPER_EL0.
- At EL0 using Arch32 when EL1 is using AArch64: `MRC` reads of PMEVTYPER<n> and PMXEVTYPER.

`0b0` The operations listed above are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- `MRS` reads at EL1 and EL0 using AArch64 of PMEVTYPER<n>_EL0 and PMXEVTYPER_EL0 are trapped to EL2 and reported with EC syndrome value `0x18`.
- `MRC` reads at EL0 using AArch32 of PMEVTYPER<n> and PMXEVTYPER are trapped to EL2 and reported with EC syndrome value `0x03`, unless the read generates a higher priority exception.

When ARMv8.6-FGT is implemented, then, regardless of the value of this bit, for each value n:

- If event counter n is not implemented, the following accesses are UNDEFINED:
  - In AArch64 state, a read of PMEVTYPER<n>_EL0, or, if n is not 31, a read of PMXEVTYPER_EL0 when PMSELR_EL0.SEL == n.
  - In AArch32 state, a read of PMEVTYPER<n>, or, if n is not 31, a read of PMXEVTYPER when PMSELR.SEL == n.
- If event counter n is implemented and EL2 is implemented and enabled in the current Security state, the following generate a Trap exception to EL2 from EL0 or EL1:
  - In AArch64 state, a read of PMEVTYPER<n>_EL0, or a read of PMXEVTYPER_EL0 when PMSELR_EL0.SEL == n, reported with EC syndrome value `0x18`.
  - In AArch32 state, a read of PMEVTYPER<n>, or a read of PMXEVTYPER when PMSELR.SEL == n, reported with EC syndrome value `0x03`.

See also HDFGRTR_EL2.PMCCFILTR_EL0.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

PMEVCNTRn_EL0, bit [12]

When PMUv3 is implemented:

Trap `MRS` reads and `MRC` reads of multiple System registers.

Enables a trap to EL2 the following operations:

- At EL1 and EL0 using AArch64: `MRS` reads of PMEVCNTR<n>_EL0 and PMXEVCNTR_EL0.
- At EL0 using Arch32 when EL1 is using AArch64: `MRC` reads of PMEVCNTR<n> and PMXEVCNTR.

`0b0` The operations listed above are not affected by this bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- `MRS` reads at EL1 and EL0 using AArch64 of PMEVCNTR<n>_EL0 and PMXEVCNTR_EL0 are trapped to EL2 and reported with EC syndrome value `0x18`.
- `MRC` reads at EL0 using AArch32 of PMEVCNTR<n> and PMXEVCNTR are trapped to EL2 and reported with EC syndrome value `0x03`, unless the read generates a higher priority exception.

When ARMv8.6-FGT is implemented, then, regardless of the value of this bit, for each value n:

- If event counter n is not implemented, the following accesses are UNDEFINED:
  - In AArch64 state, a read of PMEVCNTR<n>_EL0, or a read of PMXEVCNTR_EL0 when PMSELR_EL0.SEL == n.
  - In AArch32 state, a read of PMEVCNTR<n>, or a read of PMXEVCNTR when PMSELR.SEL == n.
- If event counter n is implemented, and EL2 is implemented and enabled in the current Security state, the following generate a Trap exception to EL2 from EL0 or EL1:
  - In AArch64 state, a read of PMEVCNTR<n>_EL0, or a read of PMXEVCNTR_EL0 when PMSELR_EL0.SEL == n, reported with EC syndrome value `0x18`.
  - In AArch32 state, a read of PMEVCNTR<n>, or a read of PMXEVCNTR when PMSELR.SEL == n, reported with EC syndrome value `0x03`.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

OSDLR_EL1, bit [11]

When ARMv8.0-DoubleLock is implemented:

Trap `MRS` reads of OSDLR_EL1 at EL1 using AArch64 to EL2.

`0b0` `MRS` reads of OSDLR_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads of OSDLR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

OSECCR_EL1, bit [10]

Trap `MRS` reads of OSECCR_EL1 at EL1 using AArch64 to EL2.

`0b0` `MRS` reads of OSECCR_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads of OSECCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

OSLSR_EL1, bit [9]

Trap `MRS` reads of OSLSR_EL1 at EL1 using AArch64 to EL2.

`0b0` `MRS` reads of OSLSR_EL1 are not affected by this bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads of OSLSR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Bit [8]

Reserved, RES0.

DBGPRCR_EL1, bit [7]

Trap `MRS` reads of DBGPRCR_EL1 at EL1 using AArch64 to EL2.

`0b0` `MRS` reads of DBGPRCR_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads of DBGPRCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

DBGAUTHSTATUS_EL1, bit [6]

Trap `MRS` reads of DBGAUTHSTATUS_EL1 at EL1 using AArch64 to EL2.

`0b0` `MRS` reads of DBGAUTHSTATUS_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads of DBGAUTHSTATUS_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

DBGCLAIM, bit [5]

Trap `MRS` reads of multiple System registers. Enables a trap on `MRS` reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- DBGCLAIMCLR_EL1.
- DBGCLAIMSET_EL1.

`0b0` `MRS` reads of the System registers listed above are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

MDSCR_EL1, bit [4]

Trap `MRS` reads of MDSCR_EL1 at EL1 using AArch64 to EL2.

`0b0` `MRS` reads of MDSCR_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads of MDSCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

DBGWVRn_EL1, bit [3]

Trap `MRS` reads of DBGWVR<n>_EL1 at EL1 using AArch64 to EL2.

`0b0` `MRS` reads of DBGWVR<n>_EL1 are not affected by this bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads of DBGWVR<n>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

If watchpoint n is not implemented, a read of DBGWVR<n>_EL1 is UNDEFINED.

In a system where the PE resets into EL2, this field resets to `0`.

DBGWCRn_EL1, bit [2]

Trap `MRS` reads of DBGWCR<n>_EL1 at EL1 using AArch64 to EL2.

`0b0` `MRS` reads of DBGWCR<n>_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads of DBGWCR<n>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

If watchpoint n is not implemented, a read of DBGWCR<n>_EL1 is UNDEFINED.

In a system where the PE resets into EL2, this field resets to `0`.

DBGBVRn_EL1, bit [1]

Trap `MRS` reads of DBGBVR<n>_EL1 at EL1 using AArch64 to EL2.

`0b0` `MRS` reads of DBGBVR<n>_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads of DBGBVR<n>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

If breakpoint n is not implemented, a read of DBGBVR<n>_EL1 is UNDEFINED.

In a system where the PE resets into EL2, this field resets to `0`.

DBGBCRn_EL1, bit [0]

Trap `MRS` reads of DBGBCR<n>_EL1 at EL1 using AArch64 to EL2.

`0b0` `MRS` reads of DBGBCR<n>_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads of DBGBCR<n>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

If breakpoint n is not implemented, a read of DBGBCR<n>_EL1 is UNDEFINED.

In a system where the PE resets into EL2, this field resets to `0`.

Accessing the HDFGRTR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, HDFGRTR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0011	0b0001	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        return NVMem[0x1D0];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.FGTEn == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return HDFGRTR_EL2;
elsif PSTATE.EL == EL3 then
    return HDFGRTR_EL2;
```

MSR HDFGRTR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0011	0b0001	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NVMem[0x1D0] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.FGTEn == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        HDFGRTR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    HDFGRTR_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.49 HDFGWTR_EL2, Hypervisor Debug Fine-Grained Write Trap Register

The HDFGWTR_EL2 characteristics are:

Purpose

Provides controls for traps of `MSR` and `MCR` writes of debug, trace, PMU, and Statistical Profiling System registers.

Configurations

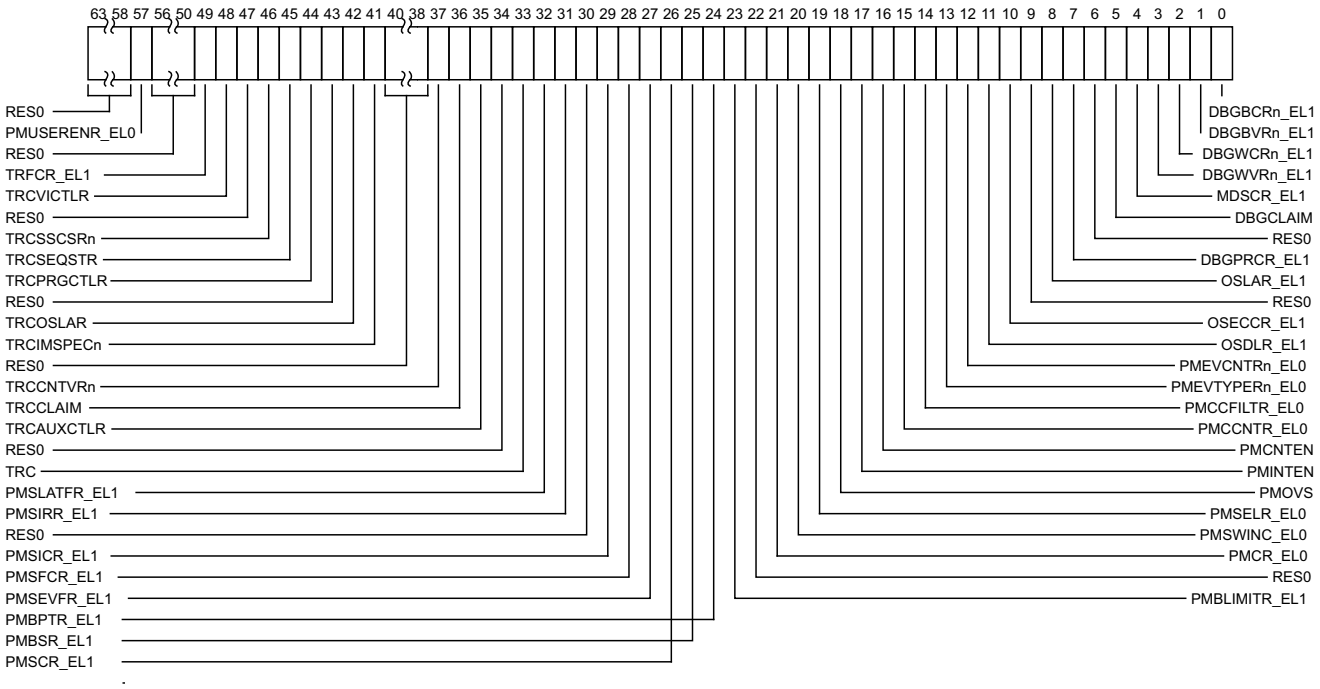
This register is present only when ARMv8.6-FGT is implemented. Otherwise, direct accesses to HDFGWTR_EL2 are UNDEFINED.

Attributes

HDFGWTR_EL2 is a 64-bit register.

Field descriptions

The HDFGWTR_EL2 bit assignments are:



Bits [63:58]

Reserved, RES0.

PMUSERENR_EL0, bit [57]

When PMUv3 is implemented:

Trap `MSR` writes of PMUSERENR_EL0 at EL1 using AArch64 to EL2.

`0b0` `MSR` writes of PMUSERENR_EL0 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes of PMUSERENR_EL0 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

Reserved, RES0.

Bits [56:50]

Reserved, RES0.

TRFCR_EL1, bit [49]

When ARMv8.4-Trace is implemented:

Trap MSR writes of TRFCR_EL1 at EL1 using AArch64 to EL2.

`0b0` MSR writes of TRFCR_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MSR writes of TRFCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

TRCVICTLR, bit [48]

When the Trace Extension is implemented and System register access to the PE Trace Unit registers is implemented:

Trap MSR writes of TRCVICTLR at EL1 using AArch64 to EL2.

`0b0` MSR writes of TRCVICTLR are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MSR writes of TRCVICTLR at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

Bit [47]

Reserved, RES0.

TRCSSCSRn, bit [46]

When the Trace Extension is implemented and System register access to the PE Trace Unit registers is implemented:

Trap MSR writes of TRCSSCSR<n> at EL1 using AArch64 to EL2.

`0b0` MSR writes of TRCSSCSR<n> are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MSR writes of TRCSSCSR<n> at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

If Single-shot Comparator n is not implemented, a write of TRCSSCSR<n> is UNDEFINED.

This bit is RES0 if TRCSSCSR<n> are not implemented.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TRCSEQSTR, bit [45]

When the Trace Extension is implemented and System register access to the PE Trace Unit registers is implemented:

Trap MSR writes of TRCSEQSTR at EL1 using AArch64 to EL2.

0b0 MSR writes of TRCSEQSTR are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MSR writes of TRCSEQSTR at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

This bit is RES0 if TRCSEQSTR is not implemented.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

TRCPRGCTLR, bit [44]

When the Trace Extension is implemented and System register access to the PE Trace Unit registers is implemented:

Trap MSR writes of TRCPRGCTLR at EL1 using AArch64 to EL2.

0b0 MSR writes of TRCPRGCTLR are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MSR writes of TRCPRGCTLR at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

Bit [43]

Reserved, RES0.

TRCOSLAR, bit [42]

When the Trace Extension is implemented, System register access to the PE Trace Unit registers is implemented and ETMv4 is implemented:

Trap MSR writes of TRCOSLAR at EL1 using AArch64 to EL2.

0b0 MSR writes of TRCOSLAR are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MSR writes of TRCOSLAR at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

TRCIMSPECn, bit [41]

When the Trace Extension is implemented and System register access to the PE Trace Unit registers is implemented:

Trap MSR writes of TRCIMSPEC<n> at EL1 using AArch64 to EL2.

0b0 MSR writes of TRCIMSPEC<n> are not affected by this bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes of TRCIMSPEC<n> at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

TRCIMSPEC<1-7> are optional. If TRCIMSPEC<n> is not implemented, a write of TRCIMSPEC<n> is UNDEFINED.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

Bits [40:38]

Reserved, RES0.

TRCCNTVRn, bit [37]

When the Trace Extension is implemented and System register access to the PE Trace Unit registers is implemented:

Trap `MSR` writes of TRCCNTVR<n> at EL1 using AArch64 to EL2.

`0b0` `MSR` writes of TRCCNTVR<n> are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes of TRCCNTVR<n> at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

If Counter n is not implemented, a write of TRCCNTVR<n> is UNDEFINED.

This bit is RES0 if TRCCNTVR<n> are not implemented.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

TRCCLAIM, bit [36]

When the Trace Extension is implemented and System register access to the PE Trace Unit registers is implemented:

Trap `MSR` writes of multiple System registers. Enables a trap on `MSR` writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- TRCCLAIMCLR.
- TRCCLAIMSET.

`0b0` `MSR` writes of the System registers listed above are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

TRCAUXCTLR, bit [35]

When the Trace Extension is implemented and System register access to the PE Trace Unit registers is implemented:

Trap `MSR` writes of TRCAUXCTLR at EL1 using AArch64 to EL2.

`0b0` `MSR` writes of TRCAUXCTLR are not affected by this bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes of TRCAUXCTLR at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

Bit [34]

Reserved, RES0.

TRC, bit [33]

When the Trace Extension is implemented and System register access to the PE Trace Unit registers is implemented:

Trap `MSR` writes of multiple System registers. Enables a trap on `MSR` writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- TRCACATR<n>.
- TRCACVR<n>.
- TRCBBCTLR.
- TRCCCCTLR.
- TRCCIDCCTLR<n>.
- TRCCIDCVR<n>.
- TRCCNTCTLR<n>.
- TRCCNTRLDVR<n>.
- TRCCONFIGR.
- TRCEVENTCTL0R.
- TRCEVENTCTL1R.
- TRCEXTINSELR, if ETMv4 is implemented.
- TRCRSCTLR<n>.
- TRCSEQEVR<n>.
- TRCSEQRSTEVR.
- TRCSSCCR<n>.
- TRCSSPCICR<n>.
- TRCSTALLCTLR.
- TRCSYNCPR.
- TRCTRACEIDR.
- TRCTSCTLR.
- TRCVIIECTLR.
- TRCVIPCSSCTLR.
- TRCVISSCTLR.
- TRCVMIDCCTLR<n>.
- TRCVMIDCVR<n>.

`0b0` `MSR` writes of the System registers listed above are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A write of an unimplemented register is UNDEFINED.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

PMSLATFR_EL1, bit [32]

When SPE is implemented:

Trap `MSR` writes of PMSLATFR_EL1 at EL1 using AArch64 to EL2.

`0b0` `MSR` writes of PMSLATFR_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes of PMSLATFR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

PMSIRR_EL1, bit [31]

When SPE is implemented:

Trap `MSR` writes of PMSIRR_EL1 at EL1 using AArch64 to EL2.

`0b0` `MSR` writes of PMSIRR_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes of PMSIRR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

Bit [30]

Reserved, RES0.

PMSICR_EL1, bit [29]

When SPE is implemented:

Trap `MSR` writes of PMSICR_EL1 at EL1 using AArch64 to EL2.

`0b0` `MSR` writes of PMSICR_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes of PMSICR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

PMSFCR_EL1, bit [28]

When SPE is implemented:

Trap `MSR` writes of PMSFCR_EL1 at EL1 using AArch64 to EL2.

`0b0` `MSR` writes of PMSFCR_EL1 are not affected by this bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes of PMSFCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

PMSEVFR_EL1, bit [27]

When SPE is implemented:

Trap `MSR` writes of PMSEVFR_EL1 at EL1 using AArch64 to EL2.

`0b0` `MSR` writes of PMSEVFR_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes of PMSEVFR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

PMSCR_EL1, bit [26]

When SPE is implemented:

Trap `MSR` writes of PMSCR_EL1 at EL1 using AArch64 to EL2.

`0b0` `MSR` writes of PMSCR_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes of PMSCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

PMBSR_EL1, bit [25]

When SPE is implemented:

Trap `MSR` writes of PMBSR_EL1 at EL1 using AArch64 to EL2.

`0b0` `MSR` writes of PMBSR_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes of PMBSR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

PMBPTR_EL1, bit [24]

When SPE is implemented:

Trap `MSR` writes of PMBPTR_EL1 at EL1 using AArch64 to EL2.

`0b0` `MSR` writes of PMBPTR_EL1 are not affected by this bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes of PMBPTR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

PMBLIMITR_EL1, bit [23]

When SPE is implemented:

Trap `MSR` writes of PMBLIMITR_EL1 at EL1 using AArch64 to EL2.

`0b0` `MSR` writes of PMBLIMITR_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes of PMBLIMITR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

Bit [22]

Reserved, RES0.

PMCR_EL0, bit [21]

When PMUv3 is implemented:

Trap `MSR` writes of PMCR_EL0 at EL1 and EL0 using AArch64 and `MCR` writes of PMCR at EL0 using AArch32 when EL1 is using AArch64 to EL2.

`0b0` `MSR` writes of PMCR_EL0 at EL1 and EL0 using AArch64 and `MCR` writes of PMCR at EL0 using AArch32 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the write generates a higher priority exception:

- `MSR` writes of PMCR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`.
- `MCR` writes of PMCR at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value `0x03`.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

PMSWINC_EL0, bit [20]

When PMUv3 is implemented:

Trap `MSR` writes of PMSWINC_EL0 at EL1 and EL0 using AArch64 and `MCR` writes of PMSWINC at EL0 using AArch32 when EL1 is using AArch64 to EL2.

`0b0` `MSR` writes of PMSWINC_EL0 at EL1 and EL0 using AArch64 and `MCR` writes of PMSWINC at EL0 using AArch32 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the write generates a higher priority exception:

- `MSR` writes of PMSWINC_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- MCR writes of PMSWINC at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

PMSELR_EL0, bit [19]

When PMUv3 is implemented:

Trap MSR writes of PMSELR_EL0 at EL1 and EL0 using AArch64 and MCR writes of PMSELR at EL0 using AArch32 when EL1 is using AArch64 to EL2.

0b0 MSR writes of PMSELR_EL0 at EL1 and EL0 using AArch64 and MCR writes of PMSELR at EL0 using AArch32 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the write generates a higher priority exception:

- MSR writes of PMSELR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.
- MCR writes of PMSELR at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

PMOVS, bit [18]

When PMUv3 is implemented:

Trap MSR writes and MCR writes of multiple System registers.

Enables a trap to EL2 the following operations:

- At EL1 and EL0 using AArch64: MSR writes of PMOVSCLR_EL0 and PMOVSSET_EL0.
- At EL0 using Arch32 when EL1 is using AArch64: MCR writes of PMOVSR and PMOVSSET.

0b0 The operations listed above are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the write generates a higher priority exception:

- MSR writes at EL1 and EL0 using AArch64 of PMOVSCLR_EL0 and PMOVSSET_EL0 are trapped to EL2 and reported with EC syndrome value 0x18.
- MCR writes at EL0 using AArch32 of PMOVSR and PMOVSSET are trapped to EL2 and reported with EC syndrome value 0x03, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

PMINTEN, bit [17]

When PMUv3 is implemented:

Trap MSR writes of multiple System registers. Enables a trap on MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- PMINTENCLR_EL1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- PMINTENSET_EL1.

`0b0` MSR writes of the System registers listed above are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MSR writes at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

PMCNTEN, bit [16]

When PMUv3 is implemented:

Trap MSR writes and MCR writes of multiple System registers.

Enables a trap to EL2 the following operations:

- At EL1 and EL0 using AArch64: MSR writes of PMCNTENCLR_EL0 and PMCNTENSET_EL0.
- At EL0 using Arch32 when EL1 is using AArch64: MCR writes of PMCNTENCLR and PMCNTENSET.

`0b0` The operations listed above are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the write generates a higher priority exception:

- MSR writes at EL1 and EL0 using AArch64 of PMCNTENCLR_EL0 and PMCNTENSET_EL0 are trapped to EL2 and reported with EC syndrome value `0x18`.
- MCR writes at EL0 using AArch32 of PMCNTENCLR and PMCNTENSET are trapped to EL2 and reported with EC syndrome value `0x03`, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

PMCCNTR_EL0, bit [15]

When PMUv3 is implemented:

Trap MSR writes of PMCCNTR_EL0 at EL1 and EL0 using AArch64 and MCR and MCRR writes of PMCCNTR at EL0 using AArch32 when EL1 is using AArch64 to EL2.

`0b0` MSR writes of PMCCNTR_EL0 at EL1 and EL0 using AArch64 and MCR and MCRR writes of PMCCNTR at EL0 using AArch32 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the write generates a higher priority exception:

- MSR writes of PMCCNTR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`.
- MCR and MCRR writes of PMCCNTR at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value `0x03` (for MCR) or `0x04` (for MCRR).

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

PMCCFILTR_EL0, bit [14]

When PMUv3 is implemented:

Trap MSR writes of PMCCFILTR_EL0 at EL1 and EL0 using AArch64 and MCR writes of PMCCFILTR at EL0 using AArch32 when EL1 is using AArch64 to EL2.

`0b0` MSR writes of PMCCFILTR_EL0 at EL1 and EL0 using AArch64 and MCR writes of PMCCFILTR at EL0 using AArch32 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the write generates a higher priority exception:

- MSR writes of PMCCFILTR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`.
- MCR writes of PMCCFILTR at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value `0x03`.

PMCCFILTR_EL0 can also be accessed in AArch64 state using PMXEVTYPER_EL0 when PMSELR_EL0.SEL == 31, and PMCCFILTR can also be accessed in AArch32 state using PMXEVTYPER when PMSELR.SEL == 31.

Setting this bit to 1 has no effect on accesses to PMXEVTYPER_EL0 and PMXEVTYPER, regardless of the value of PMSELR_EL0.SEL or PMSELR.SEL.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

PMEVTYPERn_EL0, bit [13]

When PMUv3 is implemented:

Trap MSR writes and MCR writes of multiple System registers.

Enables a trap to EL2 the following operations:

- At EL1 and EL0 using AArch64: MSR writes of PMEVTYPER<n>_EL0 and PMXEVTYPER_EL0.
- At EL0 using Arch32 when EL1 is using AArch64: MCR writes of PMEVTYPER<n> and PMXEVTYPER.

`0b0` The operations listed above are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the write generates a higher priority exception:

- MSR writes at EL1 and EL0 using AArch64 of PMEVTYPER<n>_EL0 and PMXEVTYPER_EL0 are trapped to EL2 and reported with EC syndrome value `0x18`.
- MCR writes at EL0 using AArch32 of PMEVTYPER<n> and PMXEVTYPER are trapped to EL2 and reported with EC syndrome value `0x03`, unless the write generates a higher priority exception.

When ARMv8.6-FGT is implemented, then, regardless of the value of this bit, for each value n:

- If event counter n is not implemented, the following accesses are UNDEFINED:
  - — In AArch64 state, a write of PMEVTYPER<n>_EL0, or, if n is not 31, a write of PMXEVTYPER_EL0 when PMSELR_EL0.SEL == n.
  - — In AArch32 state, a write of PMEVTYPER<n>, or, if n is not 31, a write of PMXEVTYPER when PMSELR.SEL == n.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If event counter n is implemented and EL2 is implemented and enabled in the current Security state, the following generate a Trap exception to EL2 from EL0 or EL1:
  - In AArch64 state, a write of PMEVTYPER<n>_EL0, or a write of PMXEVTYPER_EL0 when PMSELR_EL0.SEL == n, reported with EC syndrome value `0x18`.
  - In AArch32 state, a write of PMEVTYPER<n>, or a write of PMXEVTYPER when PMSELR.SEL == n, reported with EC syndrome value `0x03`.

See also HDFGWTR_EL2.PMCCFILTR_EL0.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

PMEVCNTRn_EL0, bit [12]

When PMUv3 is implemented:

Trap `MSR` writes and `MCR` writes of multiple System registers.

Enables a trap to EL2 the following operations:

- At EL1 and EL0 using AArch64: `MSR` writes of PMEVCNTR<n>_EL0 and PMXEVCNTR_EL0.
- At EL0 using Arch32 when EL1 is using AArch64: `MCR` writes of PMEVCNTR<n> and PMXEVCNTR.

`0b0` The operations listed above are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the write generates a higher priority exception:

- `MSR` writes at EL1 and EL0 using AArch64 of PMEVCNTR<n>_EL0 and PMXEVCNTR_EL0 are trapped to EL2 and reported with EC syndrome value `0x18`.
- `MCR` writes at EL0 using AArch32 of PMEVCNTR<n> and PMXEVCNTR are trapped to EL2 and reported with EC syndrome value `0x03`, unless the write generates a higher priority exception.

When ARMv8.6-FGT is implemented, then, regardless of the value of this bit, for each value n:

- If event counter n is not implemented, the following accesses are UNDEFINED:
  - In AArch64 state, a write of PMEVCNTR<n>_EL0, or a write of PMXEVCNTR_EL0 when PMSELR_EL0.SEL == n.
  - In AArch32 state, a write of PMEVCNTR<n>, or a write of PMXEVCNTR when PMSELR.SEL == n.
- If event counter n is implemented, and EL2 is implemented and enabled in the current Security state, the following generate a Trap exception to EL2 from EL0 or EL1:
  - In AArch64 state, a write of PMEVCNTR<n>_EL0, or a write of PMXEVCNTR_EL0 when PMSELR_EL0.SEL == n, reported with EC syndrome value `0x18`.
  - In AArch32 state, a write of PMEVCNTR<n>, or a write of PMXEVCNTR when PMSELR.SEL == n, reported with EC syndrome value `0x03`.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

OSDLR_EL1, bit [11]

When ARMv8.0-DoubleLock is implemented:

Trap MSR writes of OSDLR_EL1 at EL1 using AArch64 to EL2.

0b0 MSR writes of OSDLR_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MSR writes of OSDLR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

OSECCR_EL1, bit [10]

Trap MSR writes of OSECCR_EL1 at EL1 using AArch64 to EL2.

0b0 MSR writes of OSECCR_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MSR writes of OSECCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Bit [9]

Reserved, RES0.

OSLAR_EL1, bit [8]

Trap MSR writes of OSLAR_EL1 at EL1 using AArch64 to EL2.

0b0 MSR writes of OSLAR_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MSR writes of OSLAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

DBGPRCR_EL1, bit [7]

Trap MSR writes of DBGPRCR_EL1 at EL1 using AArch64 to EL2.

0b0 MSR writes of DBGPRCR_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MSR writes of DBGPRCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Bit [6]

Reserved, RES0.

DBGCLAIM, bit [5]

Trap MSR writes of multiple System registers. Enables a trap on MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- DBGCLAIMCLR_EL1.
- DBGCLAIMSET_EL1.

0b0 MSR writes of the System registers listed above are not affected by this bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

MDSCR_EL1, bit [4]

Trap `MSR` writes of MDSCR_EL1 at EL1 using AArch64 to EL2.

`0b0` `MSR` writes of MDSCR_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes of MDSCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

DBGWVRn_EL1, bit [3]

Trap `MSR` writes of DBGWVR<n>_EL1 at EL1 using AArch64 to EL2.

`0b0` `MSR` writes of DBGWVR<n>_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes of DBGWVR<n>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

If watchpoint n is not implemented, a write of DBGWVR<n>_EL1 is UNDEFINED.

In a system where the PE resets into EL2, this field resets to `0`.

DBGWCRn_EL1, bit [2]

Trap `MSR` writes of DBGWCR<n>_EL1 at EL1 using AArch64 to EL2.

`0b0` `MSR` writes of DBGWCR<n>_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes of DBGWCR<n>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

If watchpoint n is not implemented, a write of DBGWCR<n>_EL1 is UNDEFINED.

In a system where the PE resets into EL2, this field resets to `0`.

DBGBVRn_EL1, bit [1]

Trap `MSR` writes of DBGBVR<n>_EL1 at EL1 using AArch64 to EL2.

`0b0` `MSR` writes of DBGBVR<n>_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes of DBGBVR<n>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

If breakpoint n is not implemented, a write of DBGBVR<n>_EL1 is UNDEFINED.

In a system where the PE resets into EL2, this field resets to `0`.

DBGBCRn_EL1, bit [0]

Trap `MSR` writes of DBGBCR<n>_EL1 at EL1 using AArch64 to EL2.

`0b0` `MSR` writes of DBGBCR<n>_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes of DBGBCR<n>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If breakpoint n is not implemented, a write of DBGBCR<n>_EL1 is UNDEFINED.

In a system where the PE resets into EL2, this field resets to 0.

Accessing the HDFGWTR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, HDFGWTR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0011	0b0001	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        return NVMem[0x1D8];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.FGTEn == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return HDFGWTR_EL2;
elsif PSTATE.EL == EL3 then
    return HDFGWTR_EL2;
```

MSR HDFGWTR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0011	0b0001	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NVMem[0x1D8] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.FGTEn == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        HDFGWTR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    HDFGWTR_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.50 HFGITR_EL2, Hypervisor Fine-Grained Instruction Trap Register

The HFGITR_EL2 characteristics are:

Purpose

Provides controls for traps of execution of System instructions.

Configurations

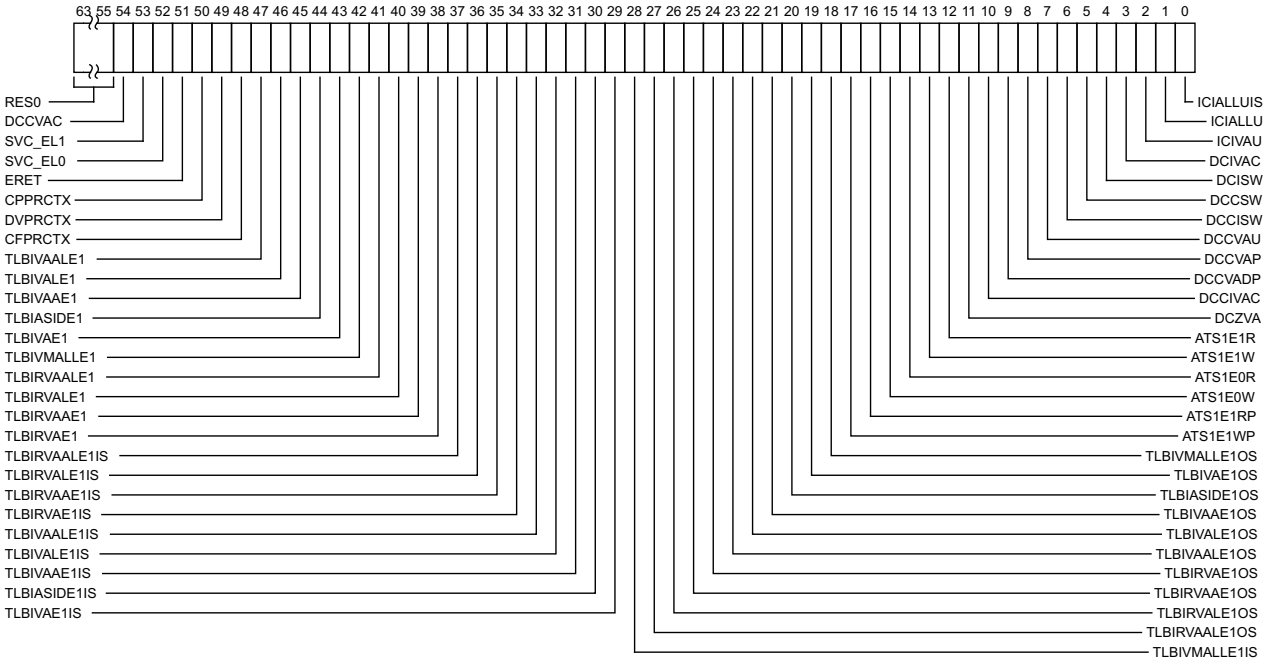
This register is present only when ARMv8.6-FGT is implemented. Otherwise, direct accesses to HFGITR_EL2 are UNDEFINED.

Attributes

HFGITR_EL2 is a 64-bit register.

Field descriptions

The HFGITR_EL2 bit assignments are:



Bits [63:55]

Reserved, RES0.

DCCVAC, bit [54]

Trap execution of multiple System instructions. Enables a trap on execution at EL1 and EL0 using AArch64 of any of the following AArch64 System instructions to EL2:

- DC CGDVAC, if ARMv8.5-MemTag is implemented.
- DC CGVAC, if ARMv8.5-MemTag is implemented.
- DC CVAC.

`0b0` Execution of the System instructions listed above is not affected by this bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution at EL1 and EL0 using AArch64 of any of the System instructions listed above is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

SVC_EL1, bit [53]

Trap execution of `SVC` at EL1 using AArch64 to EL2.

`0b0` Execution of `SVC` is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of `SVC` at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x15`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

SVC_EL0, bit [52]

Trap execution of `SVC` at EL0 using AArch64 and execution of `SVC` at EL0 using AArch32 when EL1 is using AArch64 to EL2.

`0b0` Execution of `SVC` at EL0 using AArch64 and execution of `SVC` at EL0 using AArch32 is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the instruction generates a higher priority exception:

- Execution of `SVC` at EL0 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x15`.
- Execution of `SVC` at EL0 using AArch32 is trapped to EL2 and reported with EC syndrome value `0x11`.

In a system where the PE resets into EL2, this field resets to `0`.

ERET, bit [51]

Trap execution of multiple System instructions. Enables a trap on execution at EL1 using AArch64 of any of the following AArch64 System instructions to EL2:

- `ERET`.
- `ERETAA`, if ARMv8.3-PAuth is implemented.
- `ERETAB`, if ARMv8.3-PAuth is implemented.

`0b0` Execution of the System instructions listed above is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution at EL1 using AArch64 of any of the System instructions listed above is trapped to EL2 and reported with EC syndrome value `0x1A`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

CPPRCTX, bit [50]

When ARMv8.0-PredInv is implemented:

Trap execution of CPP RCTX at EL1 and EL0 using AArch64 and execution of CPPRCTX at EL0 using AArch32 when EL1 is using AArch64 to EL2.

`0b0` Execution of CPP RCTX at EL1 and EL0 using AArch64 and execution of CPPRCTX at EL0 using AArch32 is not affected by this bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the instruction generates a higher priority exception:

- Execution of CPP RCTX at EL1 and EL0 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`.
- Execution of CPPRCTX at EL0 using AArch32 is trapped to EL2 and reported with EC syndrome value `0x03`.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

DVPRCTX, bit [49]

When ARMv8.0-PredInv is implemented:

Trap execution of DVP RCTX at EL1 and EL0 using AArch64 and execution of DVPRCTX at EL0 using AArch32 when EL1 is using AArch64 to EL2.

`0b0` Execution of DVP RCTX at EL1 and EL0 using AArch64 and execution of DVPRCTX at EL0 using AArch32 is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the instruction generates a higher priority exception:

- Execution of DVP RCTX at EL1 and EL0 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`.
- Execution of DVPRCTX at EL0 using AArch32 is trapped to EL2 and reported with EC syndrome value `0x03`.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

CFPRCTX, bit [48]

When ARMv8.0-PredInv is implemented:

Trap execution of CFP RCTX at EL1 and EL0 using AArch64 and execution of CFPRCTX at EL0 using AArch32 when EL1 is using AArch64 to EL2.

`0b0` Execution of CFP RCTX at EL1 and EL0 using AArch64 and execution of CFPRCTX at EL0 using AArch32 is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the instruction generates a higher priority exception:

- Execution of CFP RCTX at EL1 and EL0 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`.
- Execution of CFPRCTX at EL0 using AArch32 is trapped to EL2 and reported with EC syndrome value `0x03`.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

TLBIVAALE1, bit [47]

Trap execution of TLBI VAALE1 at EL1 using AArch64 to EL2.

`0b0` Execution of TLBI VAALE1 is not affected by this bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of TLBI VAALE1 at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

TLBIVALE1, bit [46]

Trap execution of TLBI VALE1 at EL1 using AArch64 to EL2.

`0b0` Execution of TLBI VALE1 is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of TLBI VALE1 at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

TLBIVAAE1, bit [45]

Trap execution of TLBI VAAE1 at EL1 using AArch64 to EL2.

`0b0` Execution of TLBI VAAE1 is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of TLBI VAAE1 at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

TLBIASIDE1, bit [44]

Trap execution of TLBI ASIDE1 at EL1 using AArch64 to EL2.

`0b0` Execution of TLBI ASIDE1 is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of TLBI ASIDE1 at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

TLBIVAE1, bit [43]

Trap execution of TLBI VAE1 at EL1 using AArch64 to EL2.

`0b0` Execution of TLBI VAE1 is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of TLBI VAE1 at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

TLBIVMALLE1, bit [42]

Trap execution of TLBI VMALLE1 at EL1 using AArch64 to EL2.

`0b0` Execution of TLBI VMALLE1 is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of TLBI VMALLE1 at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TLBIRVAALE1, bit [41]

When ARMv8.4-TLBI is implemented:

Trap execution of TLBI RVAALE1 at EL1 using AArch64 to EL2.

`0b0` Execution of TLBI RVAALE1 is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of TLBI RVAALE1 at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

TLBIRVALE1, bit [40]

When ARMv8.4-TLBI is implemented:

Trap execution of TLBI RVALE1 at EL1 using AArch64 to EL2.

`0b0` Execution of TLBI RVALE1 is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of TLBI RVALE1 at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

TLBIRVAAE1, bit [39]

When ARMv8.4-TLBI is implemented:

Trap execution of TLBI RVAAE1 at EL1 using AArch64 to EL2.

`0b0` Execution of TLBI RVAAE1 is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of TLBI RVAAE1 at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

TLBIRVAE1, bit [38]

When ARMv8.4-TLBI is implemented:

Trap execution of TLBI RVAE1 at EL1 using AArch64 to EL2.

`0b0` Execution of TLBI RVAE1 is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of TLBI RVAE1 at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TLBIRVAALE1IS, bit [37]

When ARMv8.4-TLBI is implemented:

Trap execution of TLBI RVAALE1IS at EL1 using AArch64 to EL2.

`0b0` Execution of TLBI RVAALE1IS is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of TLBI RVAALE1IS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

TLBIRVALE1IS, bit [36]

When ARMv8.4-TLBI is implemented:

Trap execution of TLBI RVALE1IS at EL1 using AArch64 to EL2.

`0b0` Execution of TLBI RVALE1IS is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of TLBI RVALE1IS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

TLBIRVAAE1IS, bit [35]

When ARMv8.4-TLBI is implemented:

Trap execution of TLBI RVAAE1IS at EL1 using AArch64 to EL2.

`0b0` Execution of TLBI RVAAE1IS is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of TLBI RVAAE1IS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

TLBIRVAE1IS, bit [34]

When ARMv8.4-TLBI is implemented:

Trap execution of TLBI RVAE1IS at EL1 using AArch64 to EL2.

`0b0` Execution of TLBI RVAE1IS is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of TLBI RVAE1IS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TLBIVAALE1IS, bit [33]

Trap execution of TLBI VAALE1IS at EL1 using AArch64 to EL2.

`0b0` Execution of TLBI VAALE1IS is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of TLBI VAALE1IS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

TLBIVALE1IS, bit [32]

Trap execution of TLBI VALE1IS at EL1 using AArch64 to EL2.

`0b0` Execution of TLBI VALE1IS is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of TLBI VALE1IS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

TLBIVAAE1IS, bit [31]

Trap execution of TLBI VAAE1IS at EL1 using AArch64 to EL2.

`0b0` Execution of TLBI VAAE1IS is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of TLBI VAAE1IS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

TLBIASIDE1IS, bit [30]

Trap execution of TLBI ASIDE1IS at EL1 using AArch64 to EL2.

`0b0` Execution of TLBI ASIDE1IS is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of TLBI ASIDE1IS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

TLBIVAE1IS, bit [29]

Trap execution of TLBI VAE1IS at EL1 using AArch64 to EL2.

`0b0` Execution of TLBI VAE1IS is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of TLBI VAE1IS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

TLBIVMALLE1IS, bit [28]

Trap execution of TLBI VMALLE1IS at EL1 using AArch64 to EL2.

`0b0` Execution of TLBI VMALLE1IS is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of TLBI VMALLE1IS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In a system where the PE resets into EL2, this field resets to `0`.

TLBIRVAALE1OS, bit [27]

When ARMv8.4-TLBI is implemented:

Trap execution of TLBI RVAALE1OS at EL1 using AArch64 to EL2.

`0b0` Execution of TLBI RVAALE1OS is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of TLBI RVAALE1OS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

TLBIRVALE1OS, bit [26]

When ARMv8.4-TLBI is implemented:

Trap execution of TLBI RVALE1OS at EL1 using AArch64 to EL2.

`0b0` Execution of TLBI RVALE1OS is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of TLBI RVALE1OS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

TLBIRVAAE1OS, bit [25]

When ARMv8.4-TLBI is implemented:

Trap execution of TLBI RVAAE1OS at EL1 using AArch64 to EL2.

`0b0` Execution of TLBI RVAAE1OS is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of TLBI RVAAE1OS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

TLBIRVAE1OS, bit [24]

When ARMv8.4-TLBI is implemented:

Trap execution of TLBI RVAE1OS at EL1 using AArch64 to EL2.

`0b0` Execution of TLBI RVAE1OS is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of TLBI RVAE1OS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TLBIVAALE1OS, bit [23]

When ARMv8.4-TLBI is implemented:

Trap execution of TLBI VAALE1OS at EL1 using AArch64 to EL2.

`0b0` Execution of TLBI VAALE1OS is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of TLBI VAALE1OS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

TLBIVALE1OS, bit [22]

When ARMv8.4-TLBI is implemented:

Trap execution of TLBI VALE1OS at EL1 using AArch64 to EL2.

`0b0` Execution of TLBI VALE1OS is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of TLBI VALE1OS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

TLBIVAAE1OS, bit [21]

When ARMv8.4-TLBI is implemented:

Trap execution of TLBI VAAE1OS at EL1 using AArch64 to EL2.

`0b0` Execution of TLBI VAAE1OS is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of TLBI VAAE1OS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

TLBIASIDE1OS, bit [20]

When ARMv8.4-TLBI is implemented:

Trap execution of TLBI ASIDE1OS at EL1 using AArch64 to EL2.

`0b0` Execution of TLBI ASIDE1OS is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of TLBI ASIDE1OS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TLBIVAE1OS, bit [19]

When ARMv8.4-TLBI is implemented:

Trap execution of TLBI VAE1OS at EL1 using AArch64 to EL2.

`0b0` Execution of TLBI VAE1OS is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of TLBI VAE1OS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

TLBIVMALLE1OS, bit [18]

When ARMv8.4-TLBI is implemented:

Trap execution of TLBI VMALLE1OS at EL1 using AArch64 to EL2.

`0b0` Execution of TLBI VMALLE1OS is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of TLBI VMALLE1OS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

ATS1E1WP, bit [17]

When ARMv8.2-ATS1E1 is implemented:

Trap execution of AT S1E1WP at EL1 using AArch64 to EL2.

`0b0` Execution of AT S1E1WP is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of AT S1E1WP at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

ATS1E1RP, bit [16]

When ARMv8.2-ATS1E1 is implemented:

Trap execution of AT S1E1RP at EL1 using AArch64 to EL2.

`0b0` Execution of AT S1E1RP is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of AT S1E1RP at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ATS1E0W, bit [15]

Trap execution of AT S1E0W at EL1 using AArch64 to EL2.

`0b0` Execution of AT S1E0W is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of AT S1E0W at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

ATS1E0R, bit [14]

Trap execution of AT S1E0R at EL1 using AArch64 to EL2.

`0b0` Execution of AT S1E0R is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of AT S1E0R at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

ATS1E1W, bit [13]

Trap execution of AT S1E1W at EL1 using AArch64 to EL2.

`0b0` Execution of AT S1E1W is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of AT S1E1W at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

ATS1E1R, bit [12]

Trap execution of AT S1E1R at EL1 using AArch64 to EL2.

`0b0` Execution of AT S1E1R is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of AT S1E1R at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

DCZVA, bit [11]

Trap execution of multiple System instructions. Enables a trap on execution at EL1 and EL0 using AArch64 of any of the following AArch64 System instructions to EL2:

- DC GVA, if ARMv8.5-MemTag is implemented.
- DC GZVA, if ARMv8.5-MemTag is implemented.
- DC ZVA.

`0b0` Execution of the System instructions listed above is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution at EL1 and EL0 using AArch64 of any of the System instructions listed above is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

DCCIVAC, bit [10]

Trap execution of multiple System instructions. Enables a trap on execution at EL1 and EL0 using AArch64 of any of the following AArch64 System instructions to EL2:

- DC CIGDVAC, if ARMv8.5-MemTag is implemented.
- DC CIGVAC, if ARMv8.5-MemTag is implemented.
- DC CIVAC.

`0b0` Execution of the System instructions listed above is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution at EL1 and EL0 using AArch64 of any of the System instructions listed above is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

DCCVADP, bit [9]

When ARMv8.2-DCCVADP is implemented:

Trap execution of multiple System instructions. Enables a trap on execution at EL1 and EL0 using AArch64 of any of the following AArch64 System instructions to EL2:

- DC CGDVADP, if ARMv8.5-MemTag is implemented.
- DC CGVADP, if ARMv8.5-MemTag is implemented.
- DC CVADP.

`0b0` Execution of the System instructions listed above is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution at EL1 and EL0 using AArch64 of any of the System instructions listed above is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

DCCVAP, bit [8]

Trap execution of multiple System instructions. Enables a trap on execution at EL1 and EL0 using AArch64 of any of the following AArch64 System instructions to EL2:

- DC CGDVAP, if ARMv8.5-MemTag is implemented.
- DC CGVAP, if ARMv8.5-MemTag is implemented.
- DC CVAP.

`0b0` Execution of the System instructions listed above is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution at EL1 and EL0 using AArch64 of any of the System instructions listed above is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

DCCVAU, bit [7]

Trap execution of DC CVAU at EL1 and EL0 using AArch64 to EL2.

`0b0` Execution of DC CVAU is not affected by this bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of DC CVAU at EL1 and EL0 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

DCCISW, bit [6]

Trap execution of multiple System instructions. Enables a trap on execution at EL1 using AArch64 of any of the following AArch64 System instructions to EL2:

- DC CIGDSW, if ARMv8.5-MemTag is implemented.
- DC CIGSW, if ARMv8.5-MemTag is implemented.
- DC CISW.

`0b0` Execution of the System instructions listed above is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution at EL1 using AArch64 of any of the System instructions listed above is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

DCCSW, bit [5]

Trap execution of multiple System instructions. Enables a trap on execution at EL1 using AArch64 of any of the following AArch64 System instructions to EL2:

- DC CGDSW, if ARMv8.5-MemTag is implemented.
- DC CGSW, if ARMv8.5-MemTag is implemented.
- DC CSW.

`0b0` Execution of the System instructions listed above is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution at EL1 using AArch64 of any of the System instructions listed above is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

DCISW, bit [4]

Trap execution of multiple System instructions. Enables a trap on execution at EL1 using AArch64 of any of the following AArch64 System instructions to EL2:

- DC IGDSW, if ARMv8.5-MemTag is implemented.
- DC IGSW, if ARMv8.5-MemTag is implemented.
- DC ISW.

`0b0` Execution of the System instructions listed above is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution at EL1 using AArch64 of any of the System instructions listed above is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

DCIVAC, bit [3]

Trap execution of multiple System instructions. Enables a trap on execution at EL1 using AArch64 of any of the following AArch64 System instructions to EL2:

- DC IGDVAC, if ARMv8.5-MemTag is implemented.
- DC IGVAC, if ARMv8.5-MemTag is implemented.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- DC IVAC.

`0b0` Execution of the System instructions listed above is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution at EL1 using AArch64 of any of the System instructions listed above is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

ICIVAU, bit [2]

Trap execution of IC IVAU at EL1 and EL0 using AArch64 to EL2.

`0b0` Execution of IC IVAU is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of IC IVAU at EL1 and EL0 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

ICIALLU, bit [1]

Trap execution of IC IALLU at EL1 using AArch64 to EL2.

`0b0` Execution of IC IALLU is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of IC IALLU at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

ICIALLUIS, bit [0]

Trap execution of IC IALLUIS at EL1 using AArch64 to EL2.

`0b0` Execution of IC IALLUIS is not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, execution of IC IALLUIS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Accessing the HFGITR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, HFGITR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        return NVMem[0x1C8];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.FGTEn == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return HFGITR_EL2;
elsif PSTATE.EL == EL3 then
    return HFGITR_EL2;
```

MSR HFGITR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NVMem[0x1C8] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.FGTEn == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        HFGITR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    HFGITR_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.51 HFGRTR_EL2, Hypervisor Fine-Grained Read Trap Register

The HFGRTR_EL2 characteristics are:

Purpose

Provides controls for traps of `MRS` and `MRC` reads of System registers.

Configurations

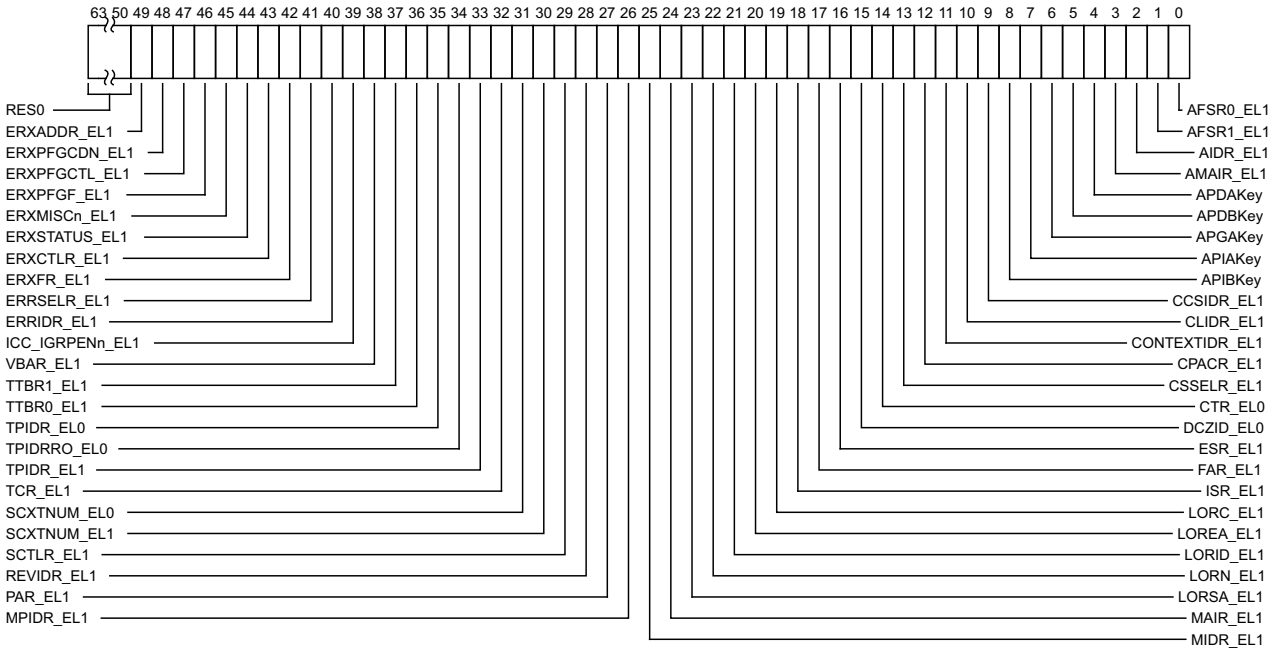
This register is present only when ARMv8.6-FGT is implemented. Otherwise, direct accesses to HFGRTR_EL2 are UNDEFINED.

Attributes

HFGRTR_EL2 is a 64-bit register.

Field descriptions

The HFGRTR_EL2 bit assignments are:



Bits [63:50]

Reserved, RES0.

ERXADDR_EL1, bit [49]

When RAS is implemented:

Trap `MRS` reads of ERXADDR_EL1 at EL1 using AArch64 to EL2.

`0b0` `MRS` reads of ERXADDR_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads of ERXADDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ERXPFGCDN_EL1, bit [48]

When ARMv8.4-RAS is implemented:

Trap MRS reads of ERXPFGCDN_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of ERXPFGCDN_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of ERXPFGCDN_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

ERXPFGCTL_EL1, bit [47]

When ARMv8.4-RAS is implemented:

Trap MRS reads of ERXPFGCTL_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of ERXPFGCTL_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of ERXPFGCTL_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

ERXPFGF_EL1, bit [46]

When RAS is implemented:

Trap MRS reads of ERXPFGF_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of ERXPFGF_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of ERXPFGF_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

ERXMISCn_EL1, bit [45]

When RAS is implemented:

Trap MRS reads of ERXMISC<n>_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of ERXMISC<n>_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of ERXMISC<n>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ERXSTATUS_EL1, bit [44]

When RAS is implemented:

Trap MRS reads of ERXSTATUS_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of ERXSTATUS_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of ERXSTATUS_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

ERXCTLR_EL1, bit [43]

When RAS is implemented:

Trap MRS reads of ERXCTLR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of ERXCTLR_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of ERXCTLR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

ERXFR_EL1, bit [42]

When RAS is implemented:

Trap MRS reads of ERXFR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of ERXFR_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of ERXFR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

ERRSELR_EL1, bit [41]

When RAS is implemented:

Trap MRS reads of ERRSELR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of ERRSELR_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of ERRSELR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ERRIDR_EL1, bit [40]

When RAS is implemented:

Trap MRS reads of ERRIDR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of ERRIDR_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of ERRIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

ICC_IGRPENn_EL1, bit [39]

When GICv3 is implemented:

Trap MRS reads of ICC_IGRPEN<n>_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of ICC_IGRPEN<n>_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of ICC_IGRPEN<n>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

VBAR_EL1, bit [38]

Trap MRS reads of VBAR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of VBAR_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of VBAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

TTBR1_EL1, bit [37]

Trap MRS reads of TTBR1_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of TTBR1_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of TTBR1_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

TTBR0_EL1, bit [36]

Trap MRS reads of TTBR0_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of TTBR0_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of TTBR0_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TPIDR_EL0, bit [35]

Trap MRS reads of TPIDR_EL0 at EL1 and EL0 using AArch64 and MRC reads of TPIDRURW at EL0 using AArch32 when EL1 is using AArch64 to EL2.

0b0 MRS reads of TPIDR_EL0 at EL1 and EL0 using AArch64 and MRC reads of TPIDRURW at EL0 using AArch32 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- MRS reads of TPIDR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.
- MRC reads of TPIDRURW at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

In a system where the PE resets into EL2, this field resets to 0.

TPIDRRO_EL0, bit [34]

Trap MRS reads of TPIDRRO_EL0 at EL1 and EL0 using AArch64 and MRC reads of TPIDRURO at EL0 using AArch32 when EL1 is using AArch64 to EL2.

0b0 MRS reads of TPIDRRO_EL0 at EL1 and EL0 using AArch64 and MRC reads of TPIDRURO at EL0 using AArch32 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- MRS reads of TPIDRRO_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.
- MRC reads of TPIDRURO at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

In a system where the PE resets into EL2, this field resets to 0.

TPIDR_EL1, bit [33]

Trap MRS reads of TPIDR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of TPIDR_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of TPIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

TCR_EL1, bit [32]

Trap MRS reads of TCR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of TCR_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of TCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

SCXTNUM_EL0, bit [31]

When ARMv8.0-CSV2 is implemented:

Trap MRS reads of SCXTNUM_EL0 at EL1 and EL0 using AArch64 to EL2.

0b0 MRS reads of SCXTNUM_EL0 are not affected by this bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads of SCXTNUM_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

SCXTNUM_EL1, bit [30]

When ARMv8.0-CSV2 is implemented:

Trap `MRS` reads of SCXTNUM_EL1 at EL1 using AArch64 to EL2.

`0b0` `MRS` reads of SCXTNUM_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads of SCXTNUM_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

SCTLR_EL1, bit [29]

Trap `MRS` reads of SCTLR_EL1 at EL1 using AArch64 to EL2.

`0b0` `MRS` reads of SCTLR_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads of SCTLR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

REVIDR_EL1, bit [28]

Trap `MRS` reads of REVIDR_EL1 at EL1 using AArch64 to EL2.

`0b0` `MRS` reads of REVIDR_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads of REVIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

PAR_EL1, bit [27]

Trap `MRS` reads of PAR_EL1 at EL1 using AArch64 to EL2.

`0b0` `MRS` reads of PAR_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads of PAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

MPIDR_EL1, bit [26]

Trap `MRS` reads of MPIDR_EL1 at EL1 using AArch64 to EL2.

`0b0` `MRS` reads of MPIDR_EL1 are not affected by this bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads of MPIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

MIDR_EL1, bit [25]

Trap `MRS` reads of MIDR_EL1 at EL1 using AArch64 to EL2.

`0b0` `MRS` reads of MIDR_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads of MIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

MAIR_EL1, bit [24]

Trap `MRS` reads of MAIR_EL1 at EL1 using AArch64 to EL2.

`0b0` `MRS` reads of MAIR_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads of MAIR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

LORSA_EL1, bit [23]

When ARMv8.1-LOR is implemented:

Trap `MRS` reads of LORSA_EL1 at EL1 using AArch64 to EL2.

`0b0` `MRS` reads of LORSA_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads of LORSA_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

LORN_EL1, bit [22]

When ARMv8.1-LOR is implemented:

Trap `MRS` reads of LORN_EL1 at EL1 using AArch64 to EL2.

`0b0` `MRS` reads of LORN_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads of LORN_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

LORID_EL1, bit [21]

When ARMv8.1-LOR is implemented:

Trap MRS reads of LORID_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of LORID_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of LORID_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

LOREA_EL1, bit [20]

When ARMv8.1-LOR is implemented:

Trap MRS reads of LOREA_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of LOREA_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of LOREA_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

LORC_EL1, bit [19]

When ARMv8.1-LOR is implemented:

Trap MRS reads of LORC_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of LORC_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of LORC_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

ISR_EL1, bit [18]

Trap MRS reads of ISR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of ISR_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of ISR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

FAR_EL1, bit [17]

Trap MRS reads of FAR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of FAR_EL1 are not affected by this bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads of FAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

ESR_EL1, bit [16]

Trap `MRS` reads of ESR_EL1 at EL1 using AArch64 to EL2.

`0b0` `MRS` reads of ESR_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads of ESR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

DCZID_EL0, bit [15]

Trap `MRS` reads of DCZID_EL0 at EL1 and EL0 using AArch64 to EL2.

`0b0` `MRS` reads of DCZID_EL0 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads of DCZID_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

CTR_EL0, bit [14]

Trap `MRS` reads of CTR_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of CTR at EL0 using AArch32 when EL1 is using AArch64 to EL2.

`0b0` `MRS` reads of CTR_EL0 at EL1 and EL0 using AArch64 and `MRC` reads of CTR at EL0 using AArch32 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- `MRS` reads of CTR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`.
- `MRC` reads of CTR at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value `0x03`.

In a system where the PE resets into EL2, this field resets to `0`.

CSSELR_EL1, bit [13]

Trap `MRS` reads of CSSELR_EL1 at EL1 using AArch64 to EL2.

`0b0` `MRS` reads of CSSELR_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads of CSSELR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

CPACR_EL1, bit [12]

Trap `MRS` reads of CPACR_EL1 at EL1 using AArch64 to EL2.

`0b0` `MRS` reads of CPACR_EL1 are not affected by this bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads of CPACR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

CONTEXTIDR_EL1, bit [11]

Trap `MRS` reads of CONTEXTIDR_EL1 at EL1 using AArch64 to EL2.

`0b0` `MRS` reads of CONTEXTIDR_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads of CONTEXTIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

CLIDR_EL1, bit [10]

Trap `MRS` reads of CLIDR_EL1 at EL1 using AArch64 to EL2.

`0b0` `MRS` reads of CLIDR_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads of CLIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

CCSIDR_EL1, bit [9]

Trap `MRS` reads of CCSIDR_EL1 at EL1 using AArch64 to EL2.

`0b0` `MRS` reads of CCSIDR_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads of CCSIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

APIBKey, bit [8]

When ARMv8.3-PAuth is implemented:

Trap `MRS` reads of multiple System registers. Enables a trap on `MRS` reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APIBKeyHi_EL1.
- APIBKeyLo_EL1.

`0b0` `MRS` reads of the System registers listed above are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

APIAKey, bit [7]

When ARMv8.3-PAuth is implemented:

Trap `MRS` reads of multiple System registers. Enables a trap on `MRS` reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APIAKeyHi_EL1.
- APIAKeyLo_EL1.

`0b0` `MRS` reads of the System registers listed above are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

APGAKey, bit [6]

When ARMv8.3-PAuth is implemented:

Trap `MRS` reads of multiple System registers. Enables a trap on `MRS` reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APGAKeyHi_EL1.
- APGAKeyLo_EL1.

`0b0` `MRS` reads of the System registers listed above are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

APDBKey, bit [5]

When ARMv8.3-PAuth is implemented:

Trap `MRS` reads of multiple System registers. Enables a trap on `MRS` reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APDBKeyHi_EL1.
- APDBKeyLo_EL1.

`0b0` `MRS` reads of the System registers listed above are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MRS` reads at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

APDAKey, bit [4]

When ARMv8.3-PAuth is implemented:

Trap MRS reads of multiple System registers. Enables a trap on MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APDAKeyHi_EL1.
- APDAKeyLo_EL1.

0b0 MRS reads of the System registers listed above are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

AMAIR_EL1, bit [3]

Trap MRS reads of AMAIR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of AMAIR_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of AMAIR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

AIDR_EL1, bit [2]

Trap MRS reads of AIDR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of AIDR_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of AIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

AFSR1_EL1, bit [1]

Trap MRS reads of AFSR1_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of AFSR1_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of AFSR1_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

AFSR0_EL1, bit [0]

Trap MRS reads of AFSR0_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of AFSR0_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MRS reads of AFSR0_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the HFGRTR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, HFGRTR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        return NVMem[0x1B8];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.FGTEn == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return HFGRTR_EL2;
elsif PSTATE.EL == EL3 then
    return HFGRTR_EL2;
```

MSR HFGRTR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NVMem[0x1B8] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.FGTEn == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        HFGRTR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    HFGRTR_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.52 HFGWTR_EL2, Hypervisor Fine-Grained Write Trap Register

The HFGWTR_EL2 characteristics are:

Purpose

Provides controls for traps of `MSR` and `MCR` writes of System registers.

Configurations

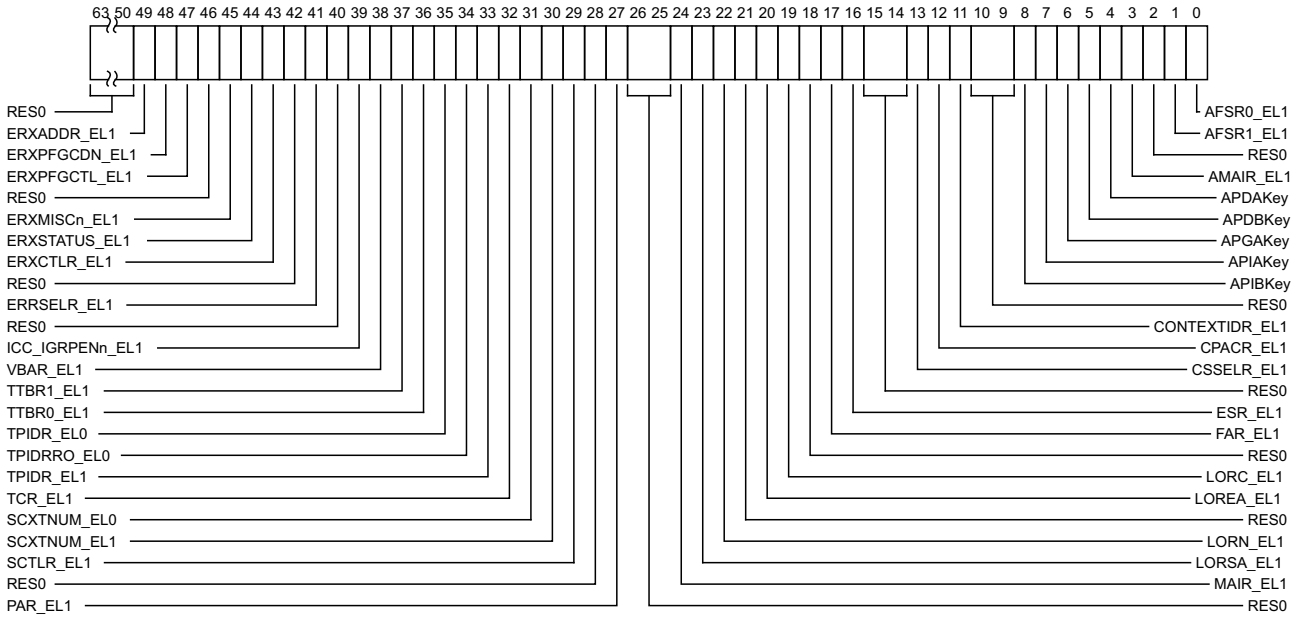
This register is present only when ARMv8.6-FGT is implemented. Otherwise, direct accesses to HFGWTR_EL2 are UNDEFINED.

Attributes

HFGWTR_EL2 is a 64-bit register.

Field descriptions

The HFGWTR_EL2 bit assignments are:



Bits [63:50]

Reserved, RES0.

ERXADDR_EL1, bit [49]

When RAS is implemented:

Trap `MSR` writes of ERXADDR_EL1 at EL1 using AArch64 to EL2.

`0b0` `MSR` writes of ERXADDR_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes of ERXADDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ERXPFGCDN_EL1, bit [48]

When ARMv8.4-RAS is implemented:

Trap MSR writes of ERXPFGCDN_EL1 at EL1 using AArch64 to EL2.

0b0 MSR writes of ERXPFGCDN_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MSR writes of ERXPFGCDN_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

ERXPFGCTL_EL1, bit [47]

When ARMv8.4-RAS is implemented:

Trap MSR writes of ERXPFGCTL_EL1 at EL1 using AArch64 to EL2.

0b0 MSR writes of ERXPFGCTL_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MSR writes of ERXPFGCTL_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

Bit [46]

Reserved, RES0.

ERXMISCn_EL1, bit [45]

When RAS is implemented:

Trap MSR writes of ERXMISC<n>_EL1 at EL1 using AArch64 to EL2.

0b0 MSR writes of ERXMISC<n>_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MSR writes of ERXMISC<n>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

ERXSTATUS_EL1, bit [44]

When RAS is implemented:

Trap MSR writes of ERXSTATUS_EL1 at EL1 using AArch64 to EL2.

0b0 MSR writes of ERXSTATUS_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MSR writes of ERXSTATUS_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ERXCTLR_EL1, bit [43]

When RAS is implemented:

Trap MSR writes of ERXCTLR_EL1 at EL1 using AArch64 to EL2.

0b0 MSR writes of ERXCTLR_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MSR writes of ERXCTLR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

Bit [42]

Reserved, RES0.

ERRSELR_EL1, bit [41]

When RAS is implemented:

Trap MSR writes of ERRSELR_EL1 at EL1 using AArch64 to EL2.

0b0 MSR writes of ERRSELR_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MSR writes of ERRSELR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

Bit [40]

Reserved, RES0.

ICC_IGRPENn_EL1, bit [39]

When GICv3 is implemented:

Trap MSR writes of ICC_IGRPEN<n>_EL1 at EL1 using AArch64 to EL2.

0b0 MSR writes of ICC_IGRPEN<n>_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MSR writes of ICC_IGRPEN<n>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

VBAR_EL1, bit [38]

Trap MSR writes of VBAR_EL1 at EL1 using AArch64 to EL2.

0b0 MSR writes of VBAR_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MSR writes of VBAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TTBR1_EL1, bit [37]

Trap MSR writes of TTBR1_EL1 at EL1 using AArch64 to EL2.

0b0 MSR writes of TTBR1_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MSR writes of TTBR1_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

TTBR0_EL1, bit [36]

Trap MSR writes of TTBR0_EL1 at EL1 using AArch64 to EL2.

0b0 MSR writes of TTBR0_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MSR writes of TTBR0_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

TPIDR_EL0, bit [35]

Trap MSR writes of TPIDR_EL0 at EL1 and EL0 using AArch64 and MCR writes of TPIDRURW at EL0 using AArch32 when EL1 is using AArch64 to EL2.

0b0 MSR writes of TPIDR_EL0 at EL1 and EL0 using AArch64 and MCR writes of TPIDRURW at EL0 using AArch32 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the write generates a higher priority exception:

- MSR writes of TPIDR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.
- MCR writes of TPIDRURW at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

In a system where the PE resets into EL2, this field resets to 0.

TPIDRRO_EL0, bit [34]

Trap MSR writes of TPIDRRO_EL0 at EL1 using AArch64 to EL2.

0b0 MSR writes of TPIDRRO_EL0 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MSR writes of TPIDRRO_EL0 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

TPIDR_EL1, bit [33]

Trap MSR writes of TPIDR_EL1 at EL1 using AArch64 to EL2.

0b0 MSR writes of TPIDR_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MSR writes of TPIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TCR_EL1, bit [32]

Trap MSR writes of TCR_EL1 at EL1 using AArch64 to EL2.

0b0 MSR writes of TCR_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MSR writes of TCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

SCXTNUM_EL0, bit [31]

When ARMv8.0-CSV2 is implemented:

Trap MSR writes of SCXTNUM_EL0 at EL1 and EL0 using AArch64 to EL2.

0b0 MSR writes of SCXTNUM_EL0 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H,TGE} != {1,1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MSR writes of SCXTNUM_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

SCXTNUM_EL1, bit [30]

When ARMv8.0-CSV2 is implemented:

Trap MSR writes of SCXTNUM_EL1 at EL1 using AArch64 to EL2.

0b0 MSR writes of SCXTNUM_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MSR writes of SCXTNUM_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

SCTLR_EL1, bit [29]

Trap MSR writes of SCTLR_EL1 at EL1 using AArch64 to EL2.

0b0 MSR writes of SCTLR_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MSR writes of SCTLR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Bit [28]

Reserved, RES0.

PAR_EL1, bit [27]

Trap MSR writes of PAR_EL1 at EL1 using AArch64 to EL2.

0b0 MSR writes of PAR_EL1 are not affected by this bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes of PAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Bits [26:25]

Reserved, RES0.

MAIR_EL1, bit [24]

Trap `MSR` writes of MAIR_EL1 at EL1 using AArch64 to EL2.

`0b0` `MSR` writes of MAIR_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes of MAIR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

LORSA_EL1, bit [23]

When ARMv8.1-LOR is implemented:

Trap `MSR` writes of LORSA_EL1 at EL1 using AArch64 to EL2.

`0b0` `MSR` writes of LORSA_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes of LORSA_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

LORN_EL1, bit [22]

When ARMv8.1-LOR is implemented:

Trap `MSR` writes of LORN_EL1 at EL1 using AArch64 to EL2.

`0b0` `MSR` writes of LORN_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes of LORN_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

Bit [21]

Reserved, RES0.

LOREA_EL1, bit [20]

When ARMv8.1-LOR is implemented:

Trap `MSR` writes of LOREA_EL1 at EL1 using AArch64 to EL2.

`0b0` `MSR` writes of LOREA_EL1 are not affected by this bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes of LOREA_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

LORC_EL1, bit [19]

When ARMv8.1-LOR is implemented:

Trap `MSR` writes of LORC_EL1 at EL1 using AArch64 to EL2.

`0b0` `MSR` writes of LORC_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes of LORC_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

Bit [18]

Reserved, RES0.

FAR_EL1, bit [17]

Trap `MSR` writes of FAR_EL1 at EL1 using AArch64 to EL2.

`0b0` `MSR` writes of FAR_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes of FAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

ESR_EL1, bit [16]

Trap `MSR` writes of ESR_EL1 at EL1 using AArch64 to EL2.

`0b0` `MSR` writes of ESR_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes of ESR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Bits [15:14]

Reserved, RES0.

CSSELR_EL1, bit [13]

Trap `MSR` writes of CSSELR_EL1 at EL1 using AArch64 to EL2.

`0b0` `MSR` writes of CSSELR_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes of CSSELR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CPACR_EL1, bit [12]

Trap MSR writes of CPACR_EL1 at EL1 using AArch64 to EL2.

0b0 MSR writes of CPACR_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MSR writes of CPACR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

CONTEXTIDR_EL1, bit [11]

Trap MSR writes of CONTEXTIDR_EL1 at EL1 using AArch64 to EL2.

0b0 MSR writes of CONTEXTIDR_EL1 are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MSR writes of CONTEXTIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Bits [10:9]

Reserved, RES0.

APIBKey, bit [8]

When ARMv8.3-PAuth is implemented:

Trap MSR writes of multiple System registers. Enables a trap on MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APIBKeyHi_EL1.
- APIBKeyLo_EL1.

0b0 MSR writes of the System registers listed above are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MSR writes at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

APIAKey, bit [7]

When ARMv8.3-PAuth is implemented:

Trap MSR writes of multiple System registers. Enables a trap on MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APIAKeyHi_EL1.
- APIAKeyLo_EL1.

0b0 MSR writes of the System registers listed above are not affected by this bit.

0b1 If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, MSR writes at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

APGAKey, bit [6]

When ARMv8.3-PAuth is implemented:

Trap `MSR` writes of multiple System registers. Enables a trap on `MSR` writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APGAKeyHi_EL1.
- APGAKeyLo_EL1.

`0b0` `MSR` writes of the System registers listed above are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

APDBKey, bit [5]

When ARMv8.3-PAuth is implemented:

Trap `MSR` writes of multiple System registers. Enables a trap on `MSR` writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APDBKeyHi_EL1.
- APDBKeyLo_EL1.

`0b0` `MSR` writes of the System registers listed above are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

APDAKey, bit [4]

When ARMv8.3-PAuth is implemented:

Trap `MSR` writes of multiple System registers. Enables a trap on `MSR` writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APDAKeyHi_EL1.
- APDAKeyLo_EL1.

`0b0` `MSR` writes of the System registers listed above are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Otherwise:

Reserved, RES0.

AMAIR_EL1, bit [3]

Trap `MSR` writes of AMAIR_EL1 at EL1 using AArch64 to EL2.

`0b0` `MSR` writes of AMAIR_EL1 are not affected by this bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes of AMAIR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Bit [2]

Reserved, RES0.

AFSR1_EL1, bit [1]

Trap `MSR` writes of AFSR1_EL1 at EL1 using AArch64 to EL2.

`0b0` `MSR` writes of AFSR1_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes of AFSR1_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

AFSR0_EL1, bit [0]

Trap `MSR` writes of AFSR0_EL1 at EL1 using AArch64 to EL2.

`0b0` `MSR` writes of AFSR0_EL1 are not affected by this bit.

`0b1` If EL2 is implemented and enabled in the current Security state and either EL3 is not implemented or SCR_EL3.FGTEn == 1, `MSR` writes of AFSR0_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

In a system where the PE resets into EL2, this field resets to `0`.

Accessing the HFGWTR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, HFGWTR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        return NVMem[0x1C0];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.FGTEn == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return HFGWTR_EL2;
elsif PSTATE.EL == EL3 then
    return HFGWTR_EL2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR HFGWTR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NVMem[0x1C0] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.FGTEn == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        HFGWTR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    HFGWTR_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.53 HPFAR_EL2, Hypervisor IPA Fault Address Register

The HPFAR_EL2 characteristics are:

Purpose

Holds the faulting IPA for some aborts on a stage 2 translation taken to EL2.

Configurations

AArch64 System register HPFAR_EL2[31:0] is architecturally mapped to AArch32 System register HPFAR[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

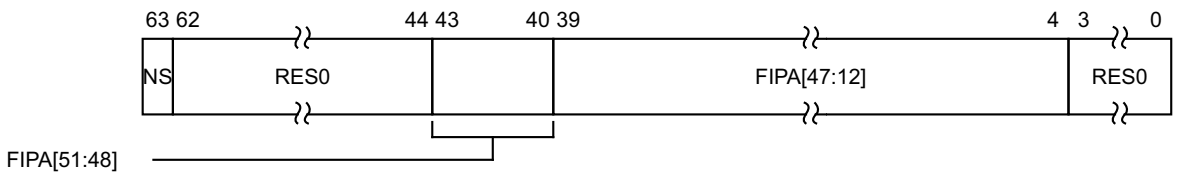
This register has no effect if EL2 is not enabled in the current Security state.

Attributes

HPFAR_EL2 is a 64-bit register.

Field descriptions

The HPFAR_EL2 bit assignments are:



Execution at EL1 or EL0 makes HPFAR_EL2 become UNKNOWN.

NS, bit [63]

When ARMv8.4-SecEL2 is implemented:

Faulting IPA address space.

`0b0` Faulting IPA is from the Secure IPA space.

`0b1` Faulting IPA is from the Non-secure IPA space.

For data or instruction aborts taken to Non-secure EL2, this field is RES0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [62:44]

Reserved, RES0.

FIPA[51:48], bits [43:40]

When ARMv8.2-LPA is implemented:

Extension to FIPA[47:12]. See FIPA[47:12] for more details.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FIPA[47:12], bits [39:4]

Bits [47:12] of the faulting intermediate physical address.

For implementations with fewer than 48 physical address bits, the corresponding upper bits in this field are RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When ARMv8.2-LPA is implemented, and 52-bit addresses and a 64KB translation granule are in use for the stage 1 translation, the FIPA[51:48] bits form the upper part of the address value. For implementations or stage 1 translation granules with fewer than 52 physical address bits the FIPA[51:48] bits are RES0.

The HPFAR_EL2 is written for:

- Translation or Access faults in the second stage of translation.
- An abort in the second stage of translation performed during the translation table walk of a first stage translation, caused by a Translation fault, an Access flag fault, or a Permission fault.
- A stage 2 Address size fault.

Note

The address held in this register is an address accessed by the instruction fetch or data access that caused the exception that gave rise to the instruction or data abort. It is the lowest address that gave rise to the fault. Where different faults from different addresses arise from the same instruction, such as for an instruction that loads or stores a mis-aligned address that crosses a page boundary, the architecture does not prioritize between those different faults.

For all other exceptions taken to EL2, this register is UNKNOWN.

This field resets to an architecturally UNKNOWN value.

Bits [3:0]

Reserved, RES0.

Accessing the HPFAR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, HPFAR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b0000	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return HPFAR_EL2;
elsif PSTATE.EL == EL3 then
    return HPFAR_EL2;
```

MSR HPFAR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b0000	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    HPFAR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    HPFAR_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.54 HSTR_EL2, Hypervisor System Trap Register

The HSTR_EL2 characteristics are:

Purpose

Controls trapping to EL2 of EL1 or lower AArch32 accesses to the System register in the coproc == `0b1111` encoding space, by the CRn value used to access the register using MCR or MRC instruction. When the register is accessible using an MCRR or MRRC instruction, this is the CRm value used to access the register.

Configurations

AArch64 System register HSTR_EL2[31:0] is architecturally mapped to AArch32 System register HSTR[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to HSTR_EL2 are RES0.

If EL2 is not implemented, this register is RES0 from EL3.

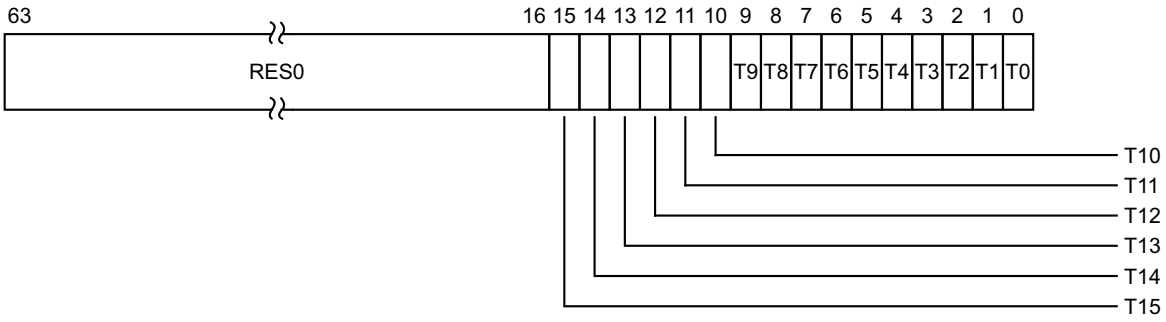
This register has no effect if EL2 is not enabled in the current Security state.

Attributes

HSTR_EL2 is a 64-bit register.

Field descriptions

The HSTR_EL2 bit assignments are:



Bits [63:16]

Reserved, RES0.

T<n>, bit [n], for n = 0 to 15

Fields T14 and T4 are RES0.

The remaining fields control whether EL0 and EL1 accesses, using MCR, MRC, MCRR, and MRRC instructions, to the System registers in the coproc == `0b1111` encoding space are trapped to EL2 as follows:

- MCR or MRC accesses to these registers that are trapped to EL2 are reported using EC syndrome value `0x03`, unless the access is UNDEFINED.
- MCRR or MRRC accesses to these registers that are trapped to EL2 are reported using EC syndrome value `0x04`, unless the access is UNDEFINED.

`0b0` This control has no effect on EL0 or EL1 accesses to System registers.

`0b1` Any EL1 MCR or MRC access with coproc == `0b1111` and `CRn` == <n> is trapped to EL2. An EL0 MCR or MRC access with these values is trapped to EL2 only if the access is not UNDEFINED when the value of this field is 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Any EL1 MCRR or MRRC access with coproc == `0b1111` and `CRm` == <n> is trapped to EL2. An EL0 MCRR or MRRC access with these values is trapped to EL2 only if the access is not UNDEFINED when the value of this field is 0.

It is IMPLEMENTATION DEFINED whether a Non-secure EL0 access using AArch32 to these registers is trapped to EL2, or is UNDEFINED and generates an exception that is taken to Non-secure EL1. If the access is UNDEFINED, and generates an exception that is taken to Non-secure EL1 using AArch64, this is reported with EC syndrome value `0x00`.

Note

Arm expects that trapping to EL2 of Non-secure EL0 accesses to these registers is unusual and used only when the hypervisor must virtualize EL0 operation. Arm recommends that, whenever possible, Non-secure EL0 accesses to these registers behave as they would if the implementation did not include EL2. This means that, if the architecture does not support the Non-secure EL0 access, then the register access instruction is treated as UNDEFINED and generates an exception that is taken to Non-secure EL1.

For example, when HSTR_EL2.T7 is 1, for instructions executed at EL1:

- An MCR or MRC instruction with coproc set to `0b1111` and `<CRn>` set to c7 is trapped to EL2.
- An MCRR or MRRC instruction with coproc set to `0b1111` and `<CRm>` set to c7 is trapped to EL2.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

This field resets to an architecturally UNKNOWN value.

Accessing the HSTR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, HSTR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        return NVMem[0x080];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return HSTR_EL2;
elsif PSTATE.EL == EL3 then
    return HSTR_EL2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR HSTR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NVMem[0x080] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    HSTR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    HSTR_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.55 ID_AA64AFR0_EL1, AArch64 Auxiliary Feature Register 0

The ID_AA64AFR0_EL1 characteristics are:

Purpose

Provides information about the IMPLEMENTATION DEFINED features of the PE in AArch64 state.

For general information about the interpretation of the ID registers, see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

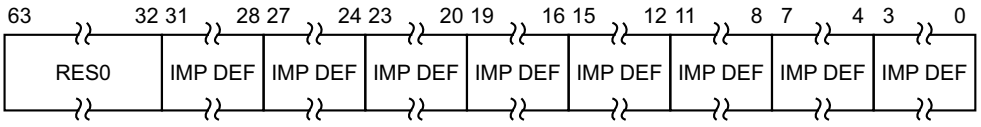
There are no configuration notes.

Attributes

ID_AA64AFR0_EL1 is a 64-bit register.

Field descriptions

The ID_AA64AFR0_EL1 bit assignments are:



Bits [63:32]

Reserved, RES0.

IMPLEMENTATION DEFINED, bits [31:28]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED, bits [27:24]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED, bits [23:20]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED, bits [19:16]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED, bits [15:12]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED, bits [11:8]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED, bits [7:4]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED, bits [3:0]

IMPLEMENTATION DEFINED.

Accessing the ID_AA64AFR0_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, ID_AA64AFR0_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0101	0b100

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_AA64AFR0_EL1;
elsif PSTATE.EL == EL2 then
    return ID_AA64AFR0_EL1;
elsif PSTATE.EL == EL3 then
    return ID_AA64AFR0_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.56 ID_AA64AFR1_EL1, AArch64 Auxiliary Feature Register 1

The ID_AA64AFR1_EL1 characteristics are:

Purpose

Reserved for future expansion of information about the IMPLEMENTATION DEFINED features of the PE in AArch64 state.

For general information about the interpretation of the ID registers, see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

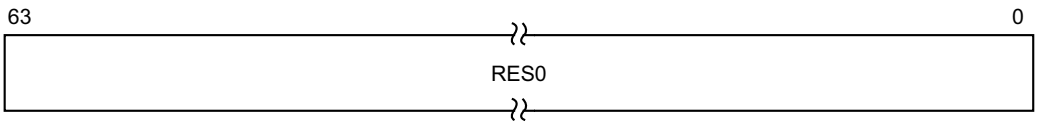
There are no configuration notes.

Attributes

ID_AA64AFR1_EL1 is a 64-bit register.

Field descriptions

The ID_AA64AFR1_EL1 bit assignments are:



Bits [63:0]

Reserved, RES0.

Accessing the ID_AA64AFR1_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ID_AA64AFR1_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0101	0b101

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_AA64AFR1_EL1;
elsif PSTATE.EL == EL2 then
    return ID_AA64AFR1_EL1;
elsif PSTATE.EL == EL3 then
    return ID_AA64AFR1_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.57 ID_AA64DFR0_EL1, AArch64 Debug Feature Register 0

The ID_AA64DFR0_EL1 characteristics are:

Purpose

Provides top level information about the debug system in AArch64 state.

For general information about the interpretation of the ID registers, see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

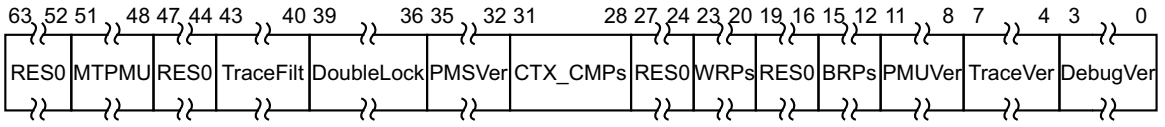
The external register EDDFR gives information from this register.

Attributes

ID_AA64DFR0_EL1 is a 64-bit register.

Field descriptions

The ID_AA64DFR0_EL1 bit assignments are:



Bits [63:52]

Reserved, RES0.

MTPMU, bits [51:48]

Multi-threaded PMU extension. Defined values are:

`0b0000` ARMv8.6-MTPMU not implemented. If PMUv3 is implemented, it is IMPLEMENTATION DEFINED whether PMEVTYPER<n>_EL0.MT are read/write or RES0.

`0b0001` ARMv8.6-MTPMU implemented and PMEVTYPER<n>_EL0.MT are read/write. When ARMv8.6-MTPMU is disabled, the Effective values of PMEVTYPER<n>_EL0.MT are 0.

`0b1111` ARMv8.6-MTPMU not implemented. If PMUv3 is implemented, PMEVTYPER<n>_EL0.MT are RES0.

All other values are reserved.

ARMv8.6-MTPMU implements the functionality identified by the value `0b0001`.

In an Armv8.6-compliant implementation that includes PMUv3, the value `0b0000` is not permitted.

In an implementation that does not include PMUv3, the value `0b0001` is not permitted.

Bits [47:44]

Reserved, RES0.

TraceFilt, bits [43:40]

Armv8.4 Self-hosted Trace Extension version. Defined values are:

`0b0000` Armv8.4 Self-hosted Trace Extension not implemented.

`0b0001` Armv8.4 Self-hosted Trace Extension implemented.

All other values are reserved.

ARMv8.4-Trace implements the functionality identified by the value `0b0001`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

DoubleLock, bits [39:36]

OS Double Lock implemented. Defined values are:

`0b0000` OS Double Lock implemented. OSDLR_EL1 is RW.

`0b1111` OS Double Lock not implemented. OSDLR_EL1 is RAZ/WI.

All other values are reserved.

ARMv8.0-DoubleLock implements the functionality identified by the value `0b0000`.

PMSVer, bits [35:32]

Statistical Profiling Extension version. Defined values are:

`0b0000` Statistical Profiling Extension not implemented.

`0b0001` Statistical Profiling Extension implemented.

`0b0010` As `0b0001` and also includes support for:

- The Event packet Alignment flag.
- If SVE is implemented, the Scalable Vector extensions to Statistical Profiling.

All other values are reserved.

SPE implements the functionality identified by the value 0b0001.

ARMv8.3-SPE implements the functionality added by the value `0b0010`.

If ARMv8.3-SPE is implemented, the values `0b0000` and `0b0001` are not permitted.

CTX_CMPs, bits [31:28]

Number of breakpoints that are context-aware, minus 1. These are the highest numbered breakpoints.

Bits [27:24]

Reserved, RES0.

WRPs, bits [23:20]

Number of watchpoints, minus 1. The value of `0b0000` is reserved.

Bits [19:16]

Reserved, RES0.

BRPs, bits [15:12]

Number of breakpoints, minus 1. The value of `0b0000` is reserved.

PMUVer, bits [11:8]

Performance Monitors Extension version.

This field does not follow the standard ID scheme, but uses the Alternative ID scheme described in *Alternative ID scheme used for the Performance Monitors Extension version* on page D13-2875.

Defined values are:

`0b0000` Performance Monitors Extension not implemented.

`0b0001` Performance Monitors Extension implemented, PMUv3.

`0b0100` PMUv3 for Armv8.1. As `0b0001`, and also includes support for:

- Extended 16-bit PMEVTYPER<n>_EL0.evtCount field.
- If EL2 is implemented, the MDCR_EL2.HPMD control bit.

`0b0101` PMUv3 for Armv8.4. As `0b0100` and also includes support for the PMMIR_EL1 register.

`0b0110` PMUv3 for Armv8.5. As `0b0101` and also includes support for:

- 64-bit event counters.
- If EL2 is implemented, the MDCR_EL2.HCCD control bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If EL3 is implemented, the MDCR_EL3.SCCD control bit.

`0b1111` IMPLEMENTATION DEFINED form of performance monitors supported, PMUv3 not supported. Arm does not recommend this value in new implementations.

All other values are reserved.

ARMv8.1-PMU implements the functionality identified by the value `0b0100`.

ARMv8.4-PMU implements the functionality identified by the value `0b0101`.

ARMv8.5-PMU implements the functionality identified by the value `0b0110`.

In an Armv8.1-compliant implementation that includes PMUv3, the value `0b0001` is not permitted.

In an Armv8.4-compliant implementation that includes PMUv3, the value `0b0100` is not permitted.

In an Armv8.5-compliant implementation that includes PMUv3, the value `0b0101` is not permitted.

TraceVer, bits [7:4]

Trace support. Indicates whether System register interface to a PE trace unit is implemented. Defined values are:

`0b0000` PE trace unit System registers not implemented.

`0b0001` PE trace unit System registers implemented.

All other values are reserved.

See the ETM Architecture Specification for more information.

A value of `0b0000` only indicates that no System register interface to a PE trace unit is implemented. A PE trace unit might nevertheless be implemented without a System register interface.

DebugVer, bits [3:0]

Debug architecture version. Indicates presence of Armv8 debug architecture. Defined values are:

`0b0110` Armv8 debug architecture.

`0b0111` Armv8 debug architecture with Virtualization Host Extensions.

`0b1000` Armv8.2 debug architecture.

`0b1001` Armv8.4 debug architecture.

All other values are reserved.

ARMv8.2-Debug adds the functionality identified by the value `0b1000`.

- If ARMv8.1-VHE is not implemented the only permitted value is `0b0110`.
- In an Armv8.0 implementation the value `0b1000` is not permitted.

Accessing the ID_AA64DFR0_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ID_AA64DFR0_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0101	0b000

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_AA64DFR0_EL1;
elsif PSTATE.EL == EL2 then
    return ID_AA64DFR0_EL1;
elsif PSTATE.EL == EL3 then
    return ID_AA64DFR0_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.58 ID_AA64DFR1_EL1, AArch64 Debug Feature Register 1

The ID_AA64DFR1_EL1 characteristics are:

Purpose

Reserved for future expansion of top level information about the debug system in AArch64 state.

For general information about the interpretation of the ID registers, see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

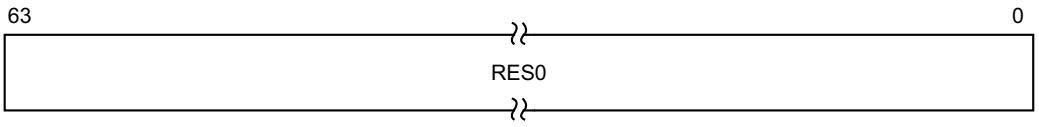
There are no configuration notes.

Attributes

ID_AA64DFR1_EL1 is a 64-bit register.

Field descriptions

The ID_AA64DFR1_EL1 bit assignments are:



Bits [63:0]

Reserved, RES0.

Accessing the ID_AA64DFR1_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ID_AA64DFR1_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0101	0b001

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_AA64DFR1_EL1;
elsif PSTATE.EL == EL2 then
    return ID_AA64DFR1_EL1;
elsif PSTATE.EL == EL3 then
    return ID_AA64DFR1_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.59 ID_AA64ISAR0_EL1, AArch64 Instruction Set Attribute Register 0

The ID_AA64ISAR0_EL1 characteristics are:

Purpose

Provides information about the instructions implemented in AArch64 state.

For general information about the interpretation of the ID registers, see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

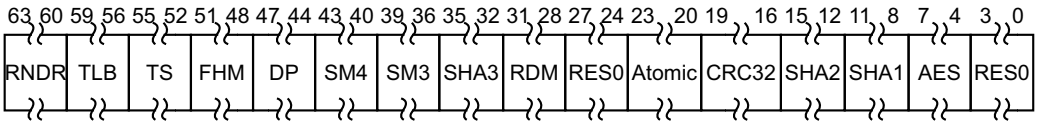
There are no configuration notes.

Attributes

ID_AA64ISAR0_EL1 is a 64-bit register.

Field descriptions

The ID_AA64ISAR0_EL1 bit assignments are:



RNDR, bits [63:60]

Indicates support for Random Number instructions in AArch64 state. Defined values are:

`0b0000` No Random Number instructions are implemented.

`0b0001` RNDR and RNDRRS registers are implemented.

All other values are reserved.

ARMv8.5-RNG implements the functionality identified by the value `0b0001`.

From Armv8.5, the permitted values are `0b0000` and `0b0001`.

TLB, bits [59:56]

Indicates support for Outer shareable and TLB range maintenance instructions. Defined values are:

`0b0000` Outer shareable and TLB range maintenance instructions are not implemented.

`0b0001` Outer shareable TLB maintenance instructions are implemented.

`0b0010` Outer shareable and TLB range maintenance instructions are implemented.

All other values are reserved.

ARMv8.4-TLBI implements the functionality identified by the values `0b0001` and `0b0010`.

From Armv8.4, the only permitted value is `0b0010`.

TS, bits [55:52]

Indicates support for flag manipulation instructions. Defined values are:

`0b0000` No flag manipulation instructions are implemented.

`0b0001` CFINV, RMIF, SETF16, and SETF8 instructions are implemented.

`0b0010` CFINV, RMIF, SETF16, SETF8, AXFLAG, and XAFLAG instructions are implemented.

All other values are reserved.

ARMv8.4-CondM implements the functionality identified by the value `0b0001`.

ARMv8.5-CondM implements the functionality identified by the value `0b0010`.

From Armv8.4, the only permitted value is `0b0001`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

From Armv8.5, the only permitted value is `0b0010`.

FHM, bits [51:48]

Indicates support for FMLAL and FMLSL instructions. Defined values are:

`0b0000` FMLAL and FMLSL instructions are not implemented.

`0b0001` FMLAL and FMLSL instructions are implemented.

All other values are reserved.

ARMv8.2-FHM implements the functionality identified by the value `0b0001`.

From Armv8.2, the permitted values are `0b0000` and `0b0001`.

DP, bits [47:44]

Indicates support for Dot Product instructions in AArch64 state. Defined values are:

`0b0000` No Dot Product instructions implemented.

`0b0001` UDOT and SDOT instructions implemented.

All other values are reserved.

ARMv8.2-DotProd implements the functionality identified by the value `0b0001`.

From Armv8.2, the permitted values are `0b0000` and `0b0001`.

SM4, bits [43:40]

Indicates support for SM4 instructions in AArch64 state. Defined values are:

`0b0000` No SM4 instructions implemented.

`0b0001` SM4E and SM4EKEY instructions implemented.

All other values are reserved.

If ARMv8.2-SM is not implemented, the value `0b0001` is reserved.

From Armv8.2, the permitted values are `0b0000` and `0b0001`.

This field must have the same value as ID_AA64ISAR0_EL1.SM3.

SM3, bits [39:36]

Indicates support for SM3 instructions in AArch64 state. Defined values are:

`0b0000` No SM3 instructions implemented.

`0b0001` SM3SS1, SM3TT1A, SM3TT1B, SM3TT2A, SM3TT2B, SM3PARTW1, and SM3PARTW2 instructions implemented.

All other values are reserved.

If ARMv8.2-SM is not implemented, the value `0b0001` is reserved.

ARMv8.2-SM implements the functionality identified by the value `0b0001`.

From Armv8.2, the permitted values are `0b0000` and `0b0001`.

This field must have the same value as ID_AA64ISAR0_EL1.SM4.

SHA3, bits [35:32]

Indicates support for SHA3 instructions in AArch64 state. Defined values are:

`0b0000` No SHA3 instructions implemented.

`0b0001` EOR3, RAX1, XAR, and BCAX instructions implemented.

All other values are reserved.

If ARMv8.2-SHA is not implemented, the value `0b0001` is reserved.

ARMv8.2-SHA implements the functionality identified by the value `0b0001`.

From Armv8.2, the permitted values are `0b0000` and `0b0001`.

If the value of ID_AA64ISAR0_EL1.SHA1 is `0b0000`, then this field must have the value `0b0000`.

If the value of this field is `0b0001`, then ID_AA64ISAR0_EL1.SHA2 must have the value `0b0010`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

RDM, bits [31:28]

Indicates support for SQRDMLAH and SQRDMLSH instructions in AArch64 state. Defined values are:

`0b0000` No RDMA instructions implemented.

`0b0001` SQRDMLAH and SQRDMLSH instructions implemented.

All other values are reserved.

ARMv8.1-RDMA implements the functionality identified by the value `0b0001`.

From Armv8.1, the only permitted value is `0b0001`.

Bits [27:24]

Reserved, RES0.

Atomic, bits [23:20]

Indicates support for Atomic instructions in AArch64 state. Defined values are:

`0b0000` No Atomic instructions implemented.

`0b0010` LDADD, LDCLR, LDEOR, LDSET, LDSMAX, LDSMIN, LDUMAX, LDUMIN, CAS, CASP, and SWP instructions implemented.

All other values are reserved.

ARMv8.1-LSE implements the functionality identified by the value `0b0010`.

From Armv8.1, the only permitted value is `0b0010`.

CRC32, bits [19:16]

CRC32 instructions implemented in AArch64 state. Defined values are:

`0b0000` No CRC32 instructions implemented.

`0b0001` CRC32B, CRC32H, CRC32W, CRC32X, CRC32CB, CRC32CH, CRC32CW, and CRC32CX instructions implemented.

All other values are reserved.

In Armv8.0, the permitted values are `0b0000` and `0b0001`.

From Armv8.1, the only permitted value is `0b0001`.

SHA2, bits [15:12]

SHA2 instructions implemented in AArch64 state. Defined values are:

`0b0000` No SHA2 instructions implemented.

`0b0001` SHA256H, SHA256H2, SHA256SU0, and SHA256SU1 instructions implemented.

`0b0010` As `0b0001`, plus SHA512H, SHA512H2, SHA512SU0, and SHA512SU1 instructions implemented.

All other values are reserved.

If ARMv8.2-SHA is not implemented the value `0b0010` is reserved.

From Armv8.2, the permitted values are `0b0000`, `0b0001`, and `0b0010`.

If the value of ID_AA64ISAR0_EL1.SHA1 is `0b0000`, then this field must have the value `0b0000`.

If the value of this field is `0b0010`, then ID_AA64ISAR0_EL1.SHA3 must have the value `0b0001`.

SHA1, bits [11:8]

SHA1 instructions implemented in AArch64 state. Defined values are:

`0b0000` No SHA1 instructions implemented.

`0b0001` SHA1C, SHA1P, SHA1M, SHA1H, SHA1SU0, and SHA1SU1 instructions implemented.

All other values are reserved.

From Armv8, the permitted values are `0b0000` and `0b0001`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the value of ID_AA64ISAR0_EL1.SHA2 is `0b0000`, then this field must have the value `0b0000`.

AES, bits [7:4]

AES instructions implemented in AArch64 state. Defined values are:

`0b0000` No AES instructions implemented.

`0b0001` AESE, AESD, AESMC, and AESIMC instructions implemented.

`0b0010` As for `0b0001`, plus PMULL/PMULL2 instructions operating on 64-bit data quantities.

All other values are reserved.

From Armv8, the permitted values are `0b0000`, `0b0001`, and `0b0010`.

Bits [3:0]

Reserved, RES0.

Accessing the ID_AA64ISAR0_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ID_AA64ISAR0_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0110	0b000

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_AA64ISAR0_EL1;
elsif PSTATE.EL == EL2 then
    return ID_AA64ISAR0_EL1;
elsif PSTATE.EL == EL3 then
    return ID_AA64ISAR0_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.60 ID_AA64ISAR1_EL1, AArch64 Instruction Set Attribute Register 1

The ID_AA64ISAR1_EL1 characteristics are:

Purpose

Provides information about the features and instructions implemented in AArch64 state.

For general information about the interpretation of the ID registers, see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

If ID_AA64ISAR1_EL1.{API, APA} == {0000, 0000}, then:

- The TCR_EL1.{TBID,TBID0}, TCR_EL2.{TBID0,TBID1}, TCR_EL2.TBID and TCR_EL3.TBID bits are RES0.
- APIAKeyHi_EL1, APIAKeyLo_EL1, APIBKeyHi_EL1, APIBKeyLo_EL1, APDAKeyHi_EL1, APDAKeyLo_EL1, APDBKeyHi_EL1, APDBKeyLo_EL1 are not allocated.
- SCTLR_ELx.EnIA, SCTLR_ELx.EnIB, SCTLR_ELx.EnDA, SCTLR_ELx.EnDB are all RES0.

If ID_AA64ISAR1_EL1.{GPI, GPA, API, APA} == {0000, 0000, 0000, 0000}, then:

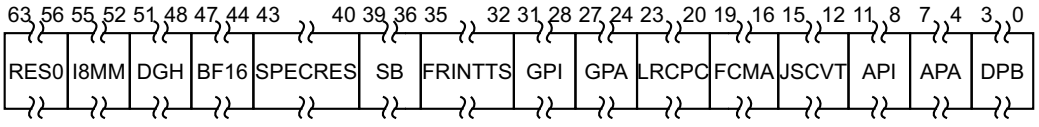
- HCR_EL2.APK and HCR_EL2.API are RES0.
- SCR_EL3.APK and SCR_EL3.API are RES0.

Attributes

ID_AA64ISAR1_EL1 is a 64-bit register.

Field descriptions

The ID_AA64ISAR1_EL1 bit assignments are:



Bits [63:56]

Reserved, RES0.

I8MM, bits [55:52]

Indicates support for Floating-point and Advanced SIMD Int8 matrix multiplication instructions in AArch64 state. Defined values of this field are:

`0b0000` Int8 matrix multiplication instructions are not implemented.

`0b0001` SMMLA, SUDOT, UMMLA, USMMLA, and USDOT instructions are implemented.

All other values are reserved.

ARMv8.2-I8MM implements the functionality identified by `0b0001`.

When Advanced SIMD and SVE are both implemented, this field must return the same value as ID_AA64ZFR0_EL1.I8MM.

From Armv8.6, the only permitted value is `0b0001`.

DGH, bits [51:48]

Indicates support for the Data Gathering Hint instruction. Defined values are:

`0b0000` Data Gathering Hint is not implemented.

`0b0001` Data Gathering Hint is implemented.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

All other values are reserved.

ARMv8.0-DGH implements the functionality identified by `0b0001`.

From ARMv8.0, the permitted values are `0b0000` and `0b0001`.

If the `DGH` instruction has no effect in preventing the merging of memory accesses, the value of this field is 0.

BF16, bits [47:44]

Indicates support for Advanced SIMD and Floating-point BFloat16 instructions in AArch64 state. Defined values are:

`0b0000` BFloat16 instructions are not implemented.

`0b0001` BFDOT, BFMLAL, BFMLAL2, BFMMLA, BFCVT, and BFCVT2 instructions are implemented.

All other values are reserved.

ARMv8.2-BF16 implements the functionality identified by `0b0001`.

When Advanced SIMD and SVE are both implemented, this field must return the same value as ID_AA64ZFR0_EL1.BF16.

From ARMv8.6, the only permitted value is `0b0001`.

SPECRES, bits [43:40]

Indicates support for prediction invalidation instructions in AArch64 state. Defined values are:

`0b0000` CFP RCTX, DVP RCTX, and CPP RCTX instructions are not implemented.

`0b0001` CFP RCTX, DVP RCTX, and CPP RCTX instructions are implemented.

All other values are reserved.

ARMv8.0-PredInv implements the functionality identified by `0b0001`.

From Armv8.5, the only permitted value is `0b0001`.

SB, bits [39:36]

Indicates support for SB instruction in AArch64 state. Defined values are:

`0b0000` SB instruction is not implemented.

`0b0001` SB instruction is implemented.

All other values are reserved.

ARMv8.0-SB implements the functionality identified by `0b0001`.

From Armv8.5, the only permitted value is `0b0001`.

FRINTTS, bits [35:32]

Indicates support for the FRINT32Z, FRINT32X, FRINT64Z, and FRINT64X instructions are implemented. Defined values are:

`0b0000` FRINT32Z, FRINT32X, FRINT64Z, and FRINT64X instructions are not implemented.

`0b0001` FRINT32Z, FRINT32X, FRINT64Z, and FRINT64X instructions are implemented.

All other values are reserved.

ARMv8.5-FRINT implements the functionality identified by `0b0001`.

From Armv8.5, the only permitted value is `0b0001`.

GPI, bits [31:28]

Indicates support for an IMPLEMENTATION DEFINED algorithm is implemented in the PE for generic code authentication in AArch64 state. Defined values are:

`0b0000` Generic Authentication using an IMPLEMENTATION DEFINED algorithm is not implemented.

`0b0001` Generic Authentication using an IMPLEMENTATION DEFINED algorithm is implemented. This includes the PACGA instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

All other values are reserved.

From Armv8.3, the permitted values are `0b0000` and `0b0001`.

If the value of ID_AA64ISAR1_EL1.GPA is non-zero, this field must have the value `0b0000`.

GPA, bits [27:24]

Indicates whether QARMA or Architected algorithm is implemented in the PE for generic code authentication in AArch64 state. Defined values are:

`0b0000` Generic Authentication using an Architected algorithm is not implemented.

`0b0001` Generic Authentication using the QARMA algorithm is implemented. This includes the PACGA instruction.

All other values are reserved.

From Armv8.3, the permitted values are `0b0000` and `0b0001`.

If the value of ID_AA64ISAR1_EL1.GPI is non-zero, this field must have the value `0b0000`.

LRCPC, bits [23:20]

Indicates support for weaker release consistency, RCpc, based model. Defined values are:

`0b0000` The LDAPR*, LDAPUR*, and STLUR* instructions are not implemented.

`0b0001` The LDAPR* instructions are implemented.

`0b0010` The LDAPR*, LDAPUR*, and STLUR* instructions are implemented.

ARMv8.3-RCPC implements the functionality identified by the value `0b0001`.

ARMv8.4-RCPC implements the functionality identified by the value `0b0010`.

In Armv8.3, the only permitted value is `0b0001`.

From Armv8.4, the only permitted value is `0b0010`.

All other values are reserved.

FCMA, bits [19:16]

Indicates support for complex number addition and multiplication, where numbers are stored in vectors. Defined values are:

`0b0000` The FCMLA and FCADD instructions are not implemented.

`0b0001` The FCMLA and FCADD instructions are implemented.

All other values are reserved.

ARMv8.3-CompNum implements the functionality identified by the value `0b0001`.

In Armv8.0, Armv8.1, and Armv8.2, the only permitted value is `0b0000`.

From Armv8.3, if Advanced SIMD or Floating-point is implemented, the only permitted value is `0b0001`.

From Armv8.3, if Advanced SIMD or Floating-point is not implemented, the only permitted value is `0b0000`.

JSCVT, bits [15:12]

Indicates support for JavaScript conversion from double precision floating point values to integers in AArch64 state. Defined values are:

`0b0000` The FJCVTZS instruction is not implemented.

`0b0001` The FJCVTZS instruction is implemented.

All other values are reserved.

ARMv8.3.JSConv implements the functionality identified by `0b0001`.

In Armv8.0, Armv8.1, and Armv8.2, the only permitted value is `0b0000`.

From Armv8.3, if Advanced SIMD or Floating-point is implemented, the only permitted value is `0b0001`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

From Armv8.3, if Advanced SIMD or Floating-point is not implemented, the only permitted value is `0b0000`.

API, bits [11:8]

Indicates whether an IMPLEMENTATION DEFINED algorithm is implemented in the PE for address authentication, in AArch64 state. This applies to all Pointer Authentication instructions other than the PACGA instruction. Defined values are:

`0b0000` Address Authentication using an IMPLEMENTATION DEFINED algorithm is not implemented.

`0b0001` Address Authentication using an IMPLEMENTATION DEFINED algorithm is implemented, with the HaveEnhancedPAC() and HaveEnhancedPAC2() functions returning FALSE.

`0b0010` Address Authentication using an IMPLEMENTATION DEFINED algorithm is implemented, with the HaveEnhancedPAC() function returning TRUE, and the HaveEnhancedPAC2() function returning FALSE.

`0b0011` Address Authentication using an IMPLEMENTATION DEFINED algorithm is implemented, with the HaveEnhancedPAC2() function returning TRUE, and the HaveEnhancedPAC() function returning FALSE.

`0b0100` Address Authentication using an IMPLEMENTATION DEFINED algorithm is implemented, with the HaveEnhancedPAC2() function returning TRUE, the HaveFPAC() function returning TRUE, the HaveFPACCombined() function returning FALSE, and the HaveEnhancedPAC() function returning FALSE.

`0b0101` Address Authentication using an IMPLEMENTATION DEFINED algorithm is implemented, with the HaveEnhancedPAC2() function returning TRUE, the HaveFPAC() function returning TRUE, the HaveFPACCombined() function returning TRUE, and the HaveEnhancedPAC() function returning FALSE.

All other values are reserved.

ARMv8.3-PAuth implements the functionality added by the values `0b0000`, `0b0001`, and `0b0010`.

ARMv8.3-PAuth2 implements the functionality added by the value `0b0011`.

ARMv8.3-FPAC implements the functionality added by the values `0b0100` and `0b0101`.

From Armv8.6, the permitted values are `0b0011`, `0b0100`, and `0b0101`.

If the value of ID_AA64ISAR1_EL1.APA is non-zero, this field must have the value `0b0000`.

APA, bits [7:4]

Indicates whether QARMA or Architected algorithm is implemented in the PE for address authentication, in AArch64 state. This applies to all Pointer Authentication instructions other than the PACGA instruction. Defined values are:

`0b0000` Address Authentication using an Architected algorithm is not implemented.

`0b0001` Address Authentication using the QARMA algorithm is implemented, with the HaveEnhancedPAC() and HaveEnhancedPAC2() functions returning FALSE.

`0b0010` Address Authentication using the QARMA algorithm is implemented, with the HaveEnhancedPAC() function returning TRUE and the HaveEnhancedPAC2() function returning FALSE.

`0b0011` Address Authentication using the QARMA algorithm is implemented, with the HaveEnhancedPAC2() function returning TRUE, the HaveFPAC() function returning FALSE, the HaveFPACCombined() function returning FALSE, and the HaveEnhancedPAC() function returning FALSE.

`0b0100` Address Authentication using the QARMA algorithm is implemented, with the HaveEnhancedPAC2() function returning TRUE, the HaveFPAC() function returning TRUE, the HaveFPACCombined() function returning FALSE, and the HaveEnhancedPAC() function returning FALSE.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b0101` Address Authentication using the QARMA algorithm is implemented, with the HaveEnhancedPAC2() function returning TRUE, the HaveFPAC() function returning TRUE, the HaveFPACCombined() function returning TRUE, and the HaveEnhancedPAC() function returning FALSE.

All other values are reserved.

ARMv8.3-PAuth implements the functionality added by the values `0b0000`, `0b0001`, and `0b0010`.

ARMv8.3-PAuth2 implements the functionality added by the value `0b0011`.

ARMv8.3-FPAC implements the functionality added by the values `0b0100` and `0b0101`.

From Armv8.6, the permitted values are `0b0011`, `0b0100`, and `0b0101`.

If the value of the ID_AA64ISAR1_EL1.API is non-zero, this field must have the value `0b0000`.

DPB, bits [3:0]

Data Persistence writeback. Indicates support for the DC CVAP and DC CVADP instructions in AArch64 state. Defined values are:

`0b0000` DC CVAP not supported.

`0b0001` DC CVAP supported.

`0b0010` DC CVAP and DC CVADP supported.

All other values are reserved.

ARMv8.2-DCPoP implements the functionality identified by the value `0b0001`.

ARMv8.2-DCCVADP implements the functionality identified by the value `0b0010`.

From Armv8.2 to Armv8.4, the only permitted value is `0b0001`.

From Armv8.5, the only permitted value is `0b0010`

Accessing the ID_AA64ISAR1_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ID_AA64ISAR1_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0110	0b001

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_AA64ISAR1_EL1;
elsif PSTATE.EL == EL2 then
    return ID_AA64ISAR1_EL1;
elsif PSTATE.EL == EL3 then
    return ID_AA64ISAR1_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.61 ID_AA64MMFR0_EL1, AArch64 Memory Model Feature Register 0

The ID_AA64MMFR0_EL1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch64 state.

For general information about the interpretation of the ID registers, see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

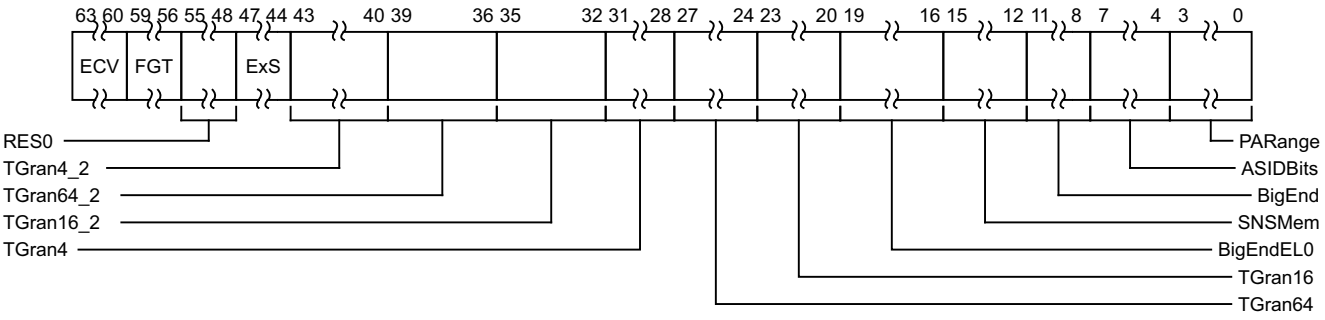
There are no configuration notes.

Attributes

ID_AA64MMFR0_EL1 is a 64-bit register.

Field descriptions

The ID_AA64MMFR0_EL1 bit assignments are:



ECV, bits [63:60]

Indicates presence of Enhanced Counter Virtualization. Defined values are:

`0b0000` Enhanced Counter Virtualization is not implemented.

`0b0001` Enhanced Counter Virtualization is implemented. Supports CNTHCTL_EL2.{EL1TVT, EL1TVCT, EL1NVPCT, EL1NVVCT, EVNTIS}, CNTKCTL_EL1.EVNTIS, CNTPCTSS_EL0 counter views, and CNTVCTSS_EL0 counter views. Extends the PMSCR_EL1.PCT, PMSCR_EL2.PCT, TRFCR_EL1.TS, and TRFCR_EL2.TS fields.

`0b0010` As `0b0001`, and also includes support for CNTHCTL_EL2.ECV and CNTPOFF_EL2.

All other values are reserved.

ARMv8.6-ECV implements the functionality identified by the values `0b0001` and `0b0010`.

From Armv8.6, the only permitted values are `0b0001` and `0b0010`.

FGT, bits [59:56]

Indicates presence of the Fine-Grained Trap controls:

- HAFGRTR_EL2, HDFGRTR_EL2, HDFGWTR_EL2, HFGRTR_EL2, HFGITR_EL2 and HFGWTR_EL2 registers, and their associated traps.
- MDCR_EL2.TDCC and MDCR_EL3.TDCC.
- SCR_EL3.FGTEn.

Defined values are:

`0b0000` The fine-grained trap controls are not implemented.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b0001` The fine-grained trap controls are implemented and can generate fine-grained traps of EL1 and EL0 functionality.

All other values are reserved.

ARMv8.6-FGT implements the functionality identified by the value `0b0001`.

From Armv8.6, the only permitted value is `0b0001`.

Bits [55:48]

Reserved, RES0.

ExS, bits [47:44]

Indicates support for disabling context synchronizing exception entry and exit. Defined values are:

`0b0000` All exception entries and exits are context synchronization events.

`0b0001` Non-context synchronizing exception entry and exit are supported.

All other values are reserved.

ARMv8.5-CSEH implements the functionality identified by the value `0b0001`.

TGran4_2, bits [43:40]

Indicates support for 4KB memory granule size for stage 2. Defined values are:

`0b0000` 4KB Stage 2 granule is identified in the ID_AA64MMFR0_EL1.TGran4 field.

`0b0001` 4KB granule not supported at stage 2.

`0b0010` 4KB granule supported at stage 2.

All other values are reserved.

The `0b0000` value is deprecated.

TGran64_2, bits [39:36]

Indicates support for 64KB memory granule size for stage 2. Defined values are:

`0b0000` 64KB Stage 2 granule is identified in the ID_AA64MMFR0_EL1.TGran64 field.

`0b0001` 64KB granule not supported at stage 2.

`0b0010` 64KB granule supported at stage 2.

All other values are reserved.

The `0b0000` value is deprecated.

TGran16_2, bits [35:32]

Indicates support for 16KB memory granule size for stage 2. Defined values are:

`0b0000` 16KB Stage 2 granule is identified in the ID_AA64MMFR0_EL1.TGran16 field

`0b0001` 16KB granule not supported at stage 2

`0b0010` 16KB granule supported at stage 2

All other values are reserved.

The `0b0000` value is deprecated.

TGran4, bits [31:28]

Indicates support for 4KB memory translation granule size. Defined values are:

`0b0000` 4KB granule supported.

`0b1111` 4KB granule not supported.

All other values are reserved.

TGran64, bits [27:24]

Indicates support for 64KB memory translation granule size. Defined values are:

`0b0000` 64KB granule supported.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1111` 64KB granule not supported.

All other values are reserved.

TGran16, bits [23:20]

Indicates support for 16KB memory translation granule size. Defined values are:

`0b0000` 16KB granule not supported.

`0b0001` 16KB granule supported.

All other values are reserved.

BigEndEL0, bits [19:16]

Indicates support for mixed-endian at EL0 only. Defined values are:

`0b0000` No mixed-endian support at EL0. The SCTLR_EL1.E0E bit has a fixed value.

`0b0001` Mixed-endian support at EL0. The SCTLR_EL1.E0E bit can be configured.

All other values are reserved.

This field is invalid and is RES0 if ID_AA64MMFR0_EL1.BigEnd is not `0b0000`.

SNSMem, bits [15:12]

Indicates support for a distinction between Secure and Non-secure Memory. Defined values are:

`0b0000` Does not support a distinction between Secure and Non-secure Memory.

`0b0001` Does support a distinction between Secure and Non-secure Memory.

All other values are reserved.

BigEnd, bits [11:8]

Indicates support for mixed-endian configuration. Defined values are:

`0b0000` No mixed-endian support. The SCTLR_ELx.EE bits have a fixed value. See the BigEndEL0 field, bits[19:16], for whether EL0 supports mixed-endian.

`0b0001` Mixed-endian support. The SCTLR_ELx.EE and SCTLR_EL1.E0E bits can be configured.

All other values are reserved.

ASIDBits, bits [7:4]

Number of ASID bits. Defined values are:

`0b0000` 8 bits.

`0b0010` 16 bits.

All other values are reserved.

PARange, bits [3:0]

Physical Address range supported. Defined values are:

`0b0000` 32 bits, 4GB.

`0b0001` 36 bits, 64GB.

`0b0010` 40 bits, 1TB.

`0b0011` 42 bits, 4TB.

`0b0100` 44 bits, 16TB.

`0b0101` 48 bits, 256TB.

`0b0110` 52 bits, 4PB.

All other values are reserved.

The value `0b0110` is permitted only if the implementation includes ARMv8.2-LPA, otherwise it is reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the ID_AA64MMFR0_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ID_AA64MMFR0_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0111	0b000

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_AA64MMFR0_EL1;
elsif PSTATE.EL == EL2 then
    return ID_AA64MMFR0_EL1;
elsif PSTATE.EL == EL3 then
    return ID_AA64MMFR0_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.62 ID_AA64MMFR1_EL1, AArch64 Memory Model Feature Register 1

The ID_AA64MMFR1_EL1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch64 state.

For general information about the interpretation of the ID registers, see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

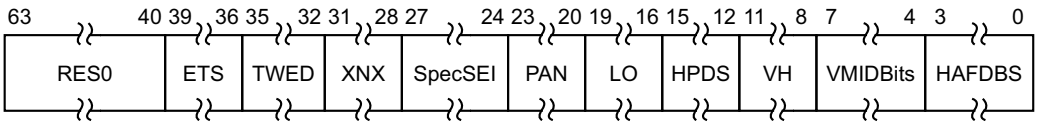
There are no configuration notes.

Attributes

ID_AA64MMFR1_EL1 is a 64-bit register.

Field descriptions

The ID_AA64MMFR1_EL1 bit assignments are:



Bits [63:40]

Reserved, RES0.

ETS, bits [39:36]

Indicates support for Enhanced Translation Synchronization. Defined values are:

`0b0000` Enhanced Translation Synchronization is not supported.

`0b0001` Enhanced Translation Synchronization is supported.

All other values are reserved.

ARMv8.0-ETS implements the functionality identified by the value `0b0001`.

From Armv8.0, the permitted values are `0b0000` and `0b0001`.

TWED, bits [35:32]

Indicates support for the configurable delayed trapping of WFE. Defined values are:

`0b0000` Configurable delayed trapping of WFE is not supported.

`0b0001` Configurable delayed trapping of WFE is supported.

All other values are reserved.

ARMv8.6-TWED implements the functionality identified by the value `0b0001`.

From Armv8.6, the permitted values are `0b0000` and `0b0001`.

XNX, bits [31:28]

Indicates support for execute-never control distinction by Exception level at stage 2. Defined values are:

`0b0000` Distinction between EL0 and EL1 execute-never control at stage 2 not supported.

`0b0001` Distinction between EL0 and EL1 execute-never control at stage 2 supported.

All other values are reserved.

ARMv8.2-TTS2UXN implements the functionality identified by the value `0b0001`.

From Armv8.2, the only permitted value is `0b0001`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

SpecSEI, bits [27:24]

Describes whether the PE can generate SError interrupt exceptions from speculative reads of memory, including speculative instruction fetches. The defined values of this field are:

`0b0000` The PE never generates an SError interrupt due to an External abort on a speculative read.

`0b0001` The PE might generate an SError interrupt due to an External abort on a speculative read.

All other values are reserved.

PAN, bits [23:20]

Privileged Access Never. Indicates support for the PAN bit in PSTATE, SPSR_EL1, SPSR_EL2, SPSR_EL3, and DSPSR_EL0. Defined values are:

`0b0000` PAN not supported.

`0b0001` PAN supported.

`0b0010` PAN supported and AT S1E1RP and AT S1E1WP instructions supported.

All other values are reserved.

ARMv8.1-PAN implements the functionality identified by the value `0b0001`.

ARMv8.2-ATS1E1 implements the functionality added by the value `0b0010`.

In Armv8.1, the only permitted value is `0b0001`.

From Armv8.2, the only permitted value is `0b0010`.

LO, bits [19:16]

LORegions. Indicates support for LORegions. Defined values are:

`0b0000` LORegions not supported.

`0b0001` LORegions supported.

All other values are reserved.

ARMv8.1-LOR implements the functionality identified by the value `0b0001`.

From Armv8.1, the only permitted value is `0b0001`.

HPDS, bits [15:12]

Hierarchical Permission Disables. Indicates support for disabling hierarchical controls in translation tables. Defined values are:

`0b0000` Disabling of hierarchical controls not supported.

`0b0001` Disabling of hierarchical controls supported with the TCR_EL1.{HPD1, HPD0}, TCR_EL2.HPD or TCR_EL2.{HPD1, HPD0}, and TCR_EL3.HPD bits.

`0b0010` As for value `0b0001`, and adds possible hardware allocation of bits[62:59] of the translation table descriptors from the final lookup level for IMPLEMENTATION DEFINED use.

All other values are reserved.

ARMv8.1-HPD implements the functionality identified by the value `0b0001`.

ARMv8.2-TTPBHA implements the functionality identified by the value `0b0010`.

From Armv8.1, the value `0b0000` is not permitted.

VH, bits [11:8]

Virtualization Host Extensions. Defined values are:

`0b0000` Virtualization Host Extensions not supported.

`0b0001` Virtualization Host Extensions supported.

All other values are reserved.

ARMv8.1-VHE implements the functionality identified by the value `0b0001`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

From Armv8.1, the only permitted value is `0b0001`.

VMIDBits, bits [7:4]

Number of VMID bits. Defined values are:

`0b0000` 8 bits

`0b0010` 16 bits

All other values are reserved.

ARMv8.1-VMID16 implements the functionality identified by the value `0b0010`.

From Armv8.1, the permitted values are `0b0000` and `0b0010`.

HAFDBS, bits [3:0]

Hardware updates to Access flag and Dirty state in translation tables. Defined values are:

`0b0000` Hardware update of the Access flag and dirty state are not supported.

`0b0001` Hardware update of the Access flag is supported.

`0b0010` Hardware update of both the Access flag and dirty state is supported.

All other values are reserved.

ARMv8.1-TTHM implements the functionality identified by the values `0b0001` and `0b0010`.

From Armv8.1, the permitted values are `0b0000`, `0b0001`, and `0b0010`.

Accessing the ID_AA64MMFR1_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ID_AA64MMFR1_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0111	0b001

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_AA64MMFR1_EL1;
elsif PSTATE.EL == EL2 then
    return ID_AA64MMFR1_EL1;
elsif PSTATE.EL == EL3 then
    return ID_AA64MMFR1_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.63 ID_AA64MMFR2_EL1, AArch64 Memory Model Feature Register 2

The ID_AA64MMFR2_EL1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch64 state.

For general information about the interpretation of the ID registers, see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

This register is present only when AArch64 is supported at any Exception level. Otherwise, direct accesses to ID_AA64MMFR2_EL1 are RES0.

Note

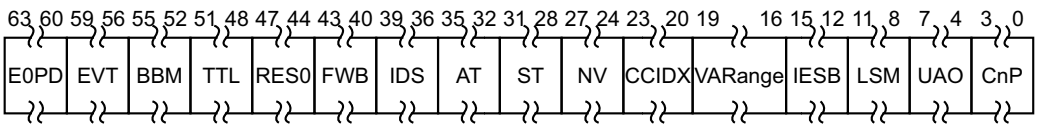
Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.

Attributes

ID_AA64MMFR2_EL1 is a 64-bit register.

Field descriptions

The ID_AA64MMFR2_EL1 bit assignments are:



E0PD, bits [63:60]

Indicates support for the E0PD mechanism. Defined values are:

`0b0000` E0PDx mechanism is not implemented.

`0b0001` E0PDx mechanism is implemented.

All other values are reserved.

ARMv8.5-E0PD implements the functionality identified by the value `0b0001`.

In Armv8.4, the permitted values are `0b0000` and `0b0001`.

From Armv8.5, the only permitted values is `0b0001`.

EVT, bits [59:56]

Enhanced Virtualization Traps. If EL2 is implemented, indicates support for the HCR_EL2.{TTLBOS, TTLBIS, TOCU, TICAB, TID4} traps. Defined values are:

`0b0000` HCR_EL2.{TTLBOS, TTLBIS, TOCU, TICAB, TID4} traps are not supported.

`0b0001` HCR_EL2.{TOCU, TICAB, TID4} traps are supported. HCR_EL2.{TTLBOS, TTLBIS} traps are not supported.

`0b0010` HCR_EL2.{TTLBOS, TTLBIS, TOCU, TICAB, TID4} traps are supported.

All other values are reserved.

ARMv8.2-EVT implements the functionality identified by the values `0b0001` and `0b0010`.

If EL2 is not implemented, the only permitted value is `0b0000`.

From Armv8.1, the permitted values are `0b0000` and `0b0001`.

From Armv8.5, if EL2 is implemented, the only permitted value is `0b0010`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

BBM, bits [55:52]

Allows identification of the requirements of the hardware to have break-before-make sequences when changing block size for a translation.

`0b0000` Level 0 support for changing block size is supported.

`0b0001` Level 1 support for changing block size is supported.

`0b0010` Level 2 support for changing block size is supported.

All other values are reserved.

ARMv8.4-TTRem implements the functionality identified by the values `0b0000`, `0b0001`, and `0b0010`.

From Armv8.4, the permitted values are `0b0000`, `0b0001`, and `0b0010`.

TTL, bits [51:48]

Indicates support for TTL field in address operations. Defined values are:

`0b0000` TLB maintenance instructions by address have bits[47:44] as RES0.

`0b0001` TLB maintenance instructions by address have bits[47:44] holding the TTL field.

All other values are reserved.

ARMv8.4-TTL implements the functionality identified by the value `0b0001`.

This field affects TLBI IPAS2E1, TLBI IPAS2E1IS, TLBI IPAS2E1OS, TLBI IPAS2LE1, TLBI IPAS2LE1IS, TLBI IPAS2LE1OS, TLBI VAAE1, TLBI VAAE1IS, TLBI VAAE1OS, TLBI VAALE1, TLBI VAALE1IS, TLBI VAALE1OS, TLBI VAE1, TLBI VAE1IS, TLBI VAE1OS, TLBI VAE2, TLBI VAE2IS, TLBI VAE2OS, TLBI VAE3, TLBI VAE3IS, TLBI VAE3OS,TLBI VALE1, TLBI VALE1IS, TLBI VALE1OS, TLBI VALE2, TLBI VALE2IS, TLBI VALE2OS, TLBI VALE3, TLBI VALE3IS, TLBI VALE3OS.

From Armv8.4, the only permitted value is `0b0001`.

Bits [47:44]

Reserved, RES0.

FWB, bits [43:40]

Indicates support for HCR_EL2.FWB. Defined values are:

`0b0000` HCR_EL2.FWB bit is not supported.

`0b0001` HCR_EL2.FWB is supported.

All other values reserved.

ARMv8.4-S2FWB implements the functionality identified by the value `0b0001`.

From Armv8.4, the only permitted value is `0b0001`.

IDS, bits [39:36]

Indicates the value of ESR_ELx.EC that reports an exception generated by a read access to the feature ID space. Defined values are:

`0b0000` An exception which is generated by a read access to the feature ID space, other than a trap caused by HCR_EL2.TIDx, SCTLR_EL1.UCT, or SCTLR_EL2.UCT, is reported by ESR_ELx.EC == `0x0`.

`0b0001` All exceptions generated by an AArch64 read access to the feature ID space are reported by ESR_ELx.EC == `0x18`.

All other values are reserved.

The Feature ID space is defined as the System register space in AArch64 with op0==3, op1=={0, 1, 3}, CRn==0, CRm=={0-7}, op2=={0-7}.

ARMv8.4-IDST implements the functionality identified by the value `0b0001`.

From Armv8.4, the only permitted value is `0b0001`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

AT, bits [35:32]

Identifies support for unaligned single-copy atomicity and atomic functions. Defined values are:

`0b0000` Unaligned single-copy atomicity and atomic functions are not supported.

`0b0001` Unaligned single-copy atomicity and atomic functions with a 16-byte address range aligned to 16-bytes are supported.

All other values are reserved.

ARMv8.4-LSE implements the functionality identified by the value `0b0001`.

From Armv8.4, the only permitted value is `0b0001`.

ST, bits [31:28]

Identifies support for small translation tables. Defined values are:

`0b0000` The maximum value of the TCR_ELx.{T0SZ,T1SZ} and VTCR_EL2.T0SZ fields is 39.

`0b0001` The maximum value of the TCR_ELx.{T0SZ,T1SZ} and VTCR_EL2.T0SZ fields is 48 for 4KB and 16KB granules, and 47 for 64KB granules.

All other values are reserved.

ARMv8.4-TTST implements the functionality identified by the value `0b0001`.

If ARMv8.4-SecEL2 is implemented, the only permitted value is `0b0001`.

In an implementation which does not support ARMv8.4-SecEL2, the permitted values are `0b0000` and `0b0001`.

NV, bits [27:24]

Nested Virtualization. If EL2 is implemented, indicates support for the use of nested virtualization. Defined values are:

`0b0000` Nested virtualization is not supported.

`0b0001` The HCR_EL2.NV, HCR_EL2.NV1, HCR_EL2.AT bits are implemented.

`0b0010` The VNCR_EL2 register and the HCR_EL2.{AT, NV, NV1, NV2} bits are implemented.

All other values are reserved.

ARMv8.3-NV implements the functionality identified by the value `0b0001`.

In Armv8.3, the permitted values are:

- When EL2 is not implemented, `0b0000`.
- When EL2 is implemented, `0b0001`.

ARMv8.4-NV implements the functionality identified by the value `0b0010`.

In Armv8.4, the permitted values are:

- When EL2 is not implemented, `0b0000`.
- When EL2 is implemented, `0b0010`.

CCIDX, bits [23:20]

Support for the use of revised CCSIDR_EL1 register format. Defined values are:

`0b0000` 32-bit format implemented for all levels of the CCSIDR_EL1.

`0b0001` 64-bit format implemented for all levels of the CCSIDR_EL1.

All other values are reserved.

ARMv8.3-CCIDX implements the functionality identified by the value `0b0001`.

From Armv8.3, the permitted values are `0b0000` and `0b0001`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

VARange, bits [19:16]

Indicates support for a larger virtual address. Defined values are:

`0b0000` VMSAv8-64 supports 48-bit VAs.

`0b0001` VMSAv8-64 supports 52-bit VAs when using the 64KB translation granule. The other translation granules support 48-bit VAs.

All other values are reserved.

ARMv8.2-LVA implements the functionality identified by the value `0b0001`.

From Armv8.2, the permitted values are `0b0000` and `0b0001`.

IESB, bits [15:12]

Indicates support for the IESB bit in the SCTLR_ELx registers. Defined values are:

`0b0000` IESB bit in the SCTLR_ELx registers is not supported.

`0b0001` IESB bit in the SCTLR_ELx registers is supported.

All other values are reserved.

ARMv8.2-IESB implements the functionality identified by the value `0b0001`.

LSM, bits [11:8]

Indicates support for LSMAOE and nTLSMD bits in SCTLR_EL1 and SCTLR_EL2. Defined values are:

`0b0000` LSMAOE and nTLSMD bits not supported.

`0b0001` LSMAOE and nTLSMD bits supported.

All other values are reserved.

ARMv8.2-LSMAOC implements the functionality identified by the value `0b0001`.

UAO, bits [7:4]

User Access Override. Defined values are:

`0b0000` UAO not supported.

`0b0001` UAO supported.

All other values are reserved.

ARMv8.2-UAO implements the functionality identified by the value `0b0001`.

From Armv8.2, the only permitted value is `0b0001`.

CnP, bits [3:0]

Indicates support for Common not Private translations. Defined values are:

`0b0000` Common not Private translations not supported.

`0b0001` Common not Private translations supported.

All other values are reserved.

ARMv8.2-TTCNP implements the functionality identified by the value `0b0001`.

From Armv8.2, the only permitted value is `0b0001`.

Accessing the ID_AA64MMFR2_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, ID_AA64MMFR2_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0111	0b010

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!IsZero(ID_AA64MMFR2_EL1) || boolean
IMPLEMENTATION_DEFINED "ID_AA64MMFR2 trapped by HCR_EL2.TID3") && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_AA64MMFR2_EL1;
elsif PSTATE.EL == EL2 then
    return ID_AA64MMFR2_EL1;
elsif PSTATE.EL == EL3 then
    return ID_AA64MMFR2_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.64 ID_AA64PFR0_EL1, AArch64 Processor Feature Register 0

The ID_AA64PFR0_EL1 characteristics are:

Purpose

Provides additional information about implemented PE features in AArch64 state.

For general information about the interpretation of the ID registers, see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

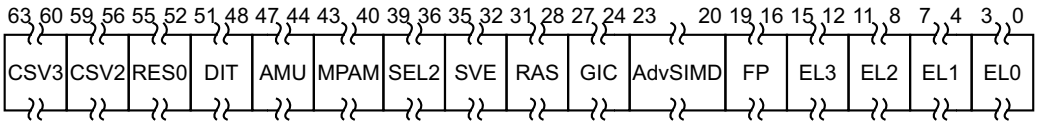
The external register EDPFR gives information from this register.

Attributes

ID_AA64PFR0_EL1 is a 64-bit register.

Field descriptions

The ID_AA64PFR0_EL1 bit assignments are:



CSV3, bits [63:60]

Speculative use of faulting data. Defined values are:

`0b0000` This Device does not disclose whether data loaded under speculation with a permission or domain fault can be used to form an address or generate condition codes or SVE predicate values to be used by instructions newer than the load in the speculative sequence

`0b0001` Data loaded under speculation with a permission or domain fault cannot be used to form an address or generate condition codes or SVE predicate values to be used by instructions newer than the load in the speculative sequence

From Armv8.5, the only permitted value is `0b0001`.

All other values are reserved.

CSV2, bits [59:56]

Speculative use of out of context branch targets. Defined values are:

`0b0000` This Device does not disclose whether branch targets trained in one hardware described context can affect speculative execution in a different hardware described context.

`0b0001` Branch targets trained in one hardware described context can only affect speculative execution in a different hardware described context in a hard-to-determine way. Contexts do not include the SCXTNUM_ELx register contexts, and these registers are not supported.

`0b0010` Branch targets trained in one hardware described context can only affect speculative execution in a different hardware described context in a hard-to-determine way. Contexts include the SCXTNUM_ELx register contexts, and these registers are supported.

From Armv8.5 the only permitted values are `0b0001` or `0b0010`.

All other values are reserved.

Bits [55:52]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

DIT, bits [51:48]

Data Independent Timing. Defined values are:

`0b0000` AArch64 does not guarantee constant execution time of any instructions.

`0b0001` AArch64 provides the PSTATE.DIT mechanism to guarantee constant execution time of certain instructions.

All other values are reserved.

ARMv8.4-DIT implements the functionality identified by the value `0b0001`.

From Armv8.4, the only permitted value is `0b0001`.

AMU, bits [47:44]

Indicates support for Activity Monitors Extension. Defined values are:

`0b0000` Activity Monitors Extension is not implemented.

`0b0001` AMUv1 for Armv8.4 is implemented.

`0b0010` AMUv1 for Armv8.6 is implemented. As `0b0001` and adds support for virtualization of the activity monitor event counters.

All other values are reserved.

AMUv1 implements the functionality identified by the value `0b0001`.

ARMv8.6-AMU implements the functionality identified by the value `0b0010`.

MPAM, bits [43:40]

Indicates support for MPAM Extension. Defined values are:

`0b0000` MPAM is not implemented.

`0b0001` MPAM is implemented.

All other values are reserved.

MPAM implements the functionality identified by the value `0b0001`.

SEL2, bits [39:36]

Secure EL2. Defined values are:

`0b0000` Secure EL2 is not implemented.

`0b0001` Secure EL2 is implemented.

All other values are reserved.

ARMv8.4-SecEL2 implements the functionality identified by the value `0b0001`.

SVE, bits [35:32]

Scalable Vector Extension. Defined values are:

`0b0000` SVE architectural state and programmers' model are not implemented.

`0b0001` SVE architectural state and programmers' model are implemented.

All other values are reserved.

If implemented, refer to ID_AA64ZFR0_EL1 for information about which SVE instructions are available.

RAS, bits [31:28]

RAS Extension version. Defined values are:

`0b0000` No RAS Extension.

`0b0001` RAS Extension present.

`0b0010` ARMv8.4-RAS present. As `0b0001`, and adds support for:

- If EL3 is implemented, ARMv8.4-DFE.
- Additional ERXMISC<m>_EL1 System registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Additional System registers ERXPFGCDN_EL1, ERXPFGCTL_EL1, and ERXPFGF_EL1, and the SCR_EL3.FIEN and HCR_EL2.FIEN trap controls, to support the optional RAS Common Fault Injection Model Extension.

Error records accessed through System registers conform to RAS System Architecture v1.1, which includes simplifications to ERR<n>STATUS and support for the optional RAS Timestamp and RAS Common Fault Injection Model Extensions.

All other values are reserved.

ARMv8.4-RAS implements the functionality identified by the value `0b0010`.

In Armv8.0 and Armv8.1, the permitted values are `0b0000` and `0b0001`.

In Armv8.2, the only permitted value is `0b0001`.

From Armv8.4, when ARMv8.4-DFE is not implemented, and ERRIDR_EL1.NUM is zero, the permitted values are IMPLEMENTATION DEFINED `0b0001` or `0b0010`. Otherwise from Armv8.4, the only permitted value is `0b0010`.

GIC, bits [27:24]

System register GIC CPU interface. Defined values are:

`0b0000` GIC CPU interface system registers not implemented.

`0b0001` System register interface to versions 3.0 and 4.0 of the GIC CPU interface is supported.

`0b0011` System register interface to version 4.1 of the GIC CPU interface is supported.

All other values are reserved.

AdvSIMD, bits [23:20]

Advanced SIMD. Defined values are:

`0b0000` Advanced SIMD is implemented, including support for the following SISD and SIMD operations:

- Integer byte, halfword, word and doubleword element operations.
- Single-precision and double-precision floating-point arithmetic.
- Conversions between single-precision and half-precision data types, and double-precision and half-precision data types.

`0b0001` As for `0b0000`, and also includes support for half-precision floating-point arithmetic.

`0b1111` Advanced SIMD is not implemented.

All other values are reserved.

This field must have the same value as the FP field.

The permitted values are:

- `0b0000` in an implementation with Advanced SIMD support that does not include the ARMv8.2-FP16 extension.
- `0b0001` in an implementation with Advanced SIMD support that includes the ARMv8.2-FP16 extension.
- `0b1111` in an implementation without Advanced SIMD support.

FP, bits [19:16]

Floating-point. Defined values are:

`0b0000` Floating-point is implemented, and includes support for:

- Single-precision and double-precision floating-point types.
- Conversions between single-precision and half-precision data types, and double-precision and half-precision data types.

`0b0001` As for `0b0000`, and also includes support for half-precision floating-point arithmetic.

`0b1111` Floating-point is not implemented.

All other values are reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field must have the same value as the AdvSIMD field.

The permitted values are:

- `0b0000` in an implementation with floating-point support that does not include the ARMv8.2-FP16 extension.
- `0b0001` in an implementation with floating-point support that includes the ARMv8.2-FP16 extension.
- `0b1111` in an implementation without floating-point support.

EL3, bits [15:12]

EL3 Exception level handling. Defined values are:

`0b0000` EL3 is not implemented.

`0b0001` EL3 can be executed in AArch64 state only.

`0b0010` EL3 can be executed in either AArch64 or AArch32 state.

All other values are reserved.

EL2, bits [11:8]

EL2 Exception level handling. Defined values are:

`0b0000` EL2 is not implemented.

`0b0001` EL2 can be executed in AArch64 state only.

`0b0010` EL2 can be executed in either AArch64 or AArch32 state.

All other values are reserved.

EL1, bits [7:4]

EL1 Exception level handling. Defined values are:

`0b0001` EL1 can be executed in AArch64 state only.

`0b0010` EL1 can be executed in either AArch64 or AArch32 state.

All other values are reserved.

EL0, bits [3:0]

EL0 Exception level handling. Defined values are:

`0b0001` EL0 can be executed in AArch64 state only.

`0b0010` EL0 can be executed in either AArch64 or AArch32 state.

All other values are reserved.

Accessing the ID_AA64PFR0_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ID_AA64PFR0_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0100	0b000

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_AA64PFR0_EL1;
elsif PSTATE.EL == EL2 then
    return ID_AA64PFR0_EL1;
elsif PSTATE.EL == EL3 then
    return ID_AA64PFR0_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.65 ID_AA64PFR1_EL1, AArch64 Processor Feature Register 1

The ID_AA64PFR1_EL1 characteristics are:

Purpose

Reserved for future expansion of information about implemented PE features in AArch64 state.

For general information about the interpretation of the ID registers, see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

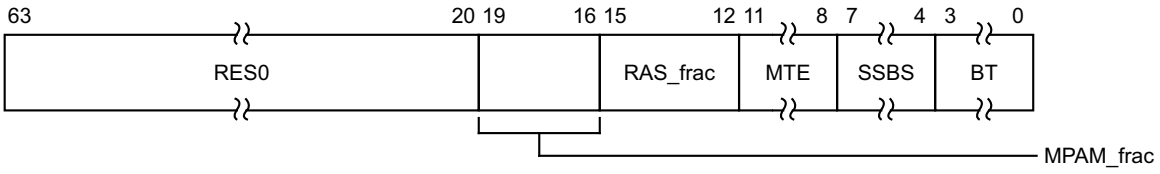
There are no configuration notes.

Attributes

ID_AA64PFR1_EL1 is a 64-bit register.

Field descriptions

The ID_AA64PFR1_EL1 bit assignments are:



Bits [63:20]

Reserved, RES0.

MPAM_frac, bits [19:16]

MPAM Extension fractional field. Defined values are:

`0b0000` If ID_AA64PFR0_EL1.MPAM == `0b0000`, MPAM Extension not implemented.
If ID_AA64PFR0_EL1.MPAM == `0b0001`, MPAM Extension version 1.0 is implemented.

`0b0001` If ID_AA64PFR0_EL1.MPAM == `0b0000`, implements MPAM version 0.1 which is like version 1.1 but reduces support for Secure PARTIDs.
If ID_AA64PFR0_EL1.MPAM == `0b0001`, implements MPAM extension version 1.1 and adds support for TIDR bit in MPAM2_EL2 to provide trapping of MPAMIDR_EL1 when MPAMHCR_EL2 is not present.

All other values are reserved.

RAS_frac, bits [15:12]

RAS Extension fractional field. Defined values are:

`0b0000` If ID_AA64PFR0_EL1.RAS == `0b0001`, RAS Extension implemented.

`0b0001` If ID_AA64PFR0_EL1.RAS == `0b0001`, as `0b0000` and adds support for:

- Additional ERXMISC<m>_EL1 System registers.
- Additional System registers ERXPFGCDN_EL1, ERXPFGCTL_EL1, and ERXPFGF_EL1, and the SCR_EL3.FIEN and HCR_EL2.FIEN trap controls, to support the optional RAS Common Fault Injection Model Extension.

Error records accessed through System registers conform to RAS System Architecture v1.1, which includes simplifications to ERR<n>STATUS, and support for the optional RAS Timestamp and RAS Common Fault Injection Model Extensions.

All other values are reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field is valid only if ID_AA64PFR0_EL1.RAS == `0b0001`.

MTE, bits [11:8]

Support for the Memory Tagging Extension. Defined values are:

`0b0000` Memory Tagging Extension is not implemented.

`0b0001` Memory Tagging Extension instructions accessible at EL0 are implemented. Instructions and System Registers defined by the extension not configurably accessible at EL0 are Unallocated and other System Register fields defined by the extension are RES0.

`0b0010` Memory Tagging Extension is implemented.

All other values are reserved.

ARMv8.5-MemTag implements the functionality identified by the value `0b0001`.

When ID_AA64PFR1_EL1.MTE != `0b0010`:

- All register fields added to existing System registers and Special-purpose registers as part of the extension are RES0, and treated as 0.
- The following System registers are UNDEFINED:
  — GMID_EL1, GCR_EL1, RGSR_EL1, TFSRE0_EL1, and TFSR_ELx.
- The following System instructions are UNDEFINED:
  — DC CGSW, DC CIGSW, DC IGSW, DC CGDSW, DC CIGSW, DC IGDSW, DC IGVAC, and DC IGDVAC.
- The following instructions are UNDEFINED:
  — `LDGM`, `STGM`, and `STZGM`.
- The Tagged memory type encoding in MAIR_ELx is UNPREDICTABLE.

SSBS, bits [7:4]

Speculative Store Bypassing controls in AArch64 state. Defined values are:

`0b0000` AArch64 provides no mechanism to control the use of Speculative Store Bypassing.

`0b0001` AArch64 provides the PSTATE.SSBS mechanism to mark regions that are Speculative Store Bypass Safe.

`0b0010` AArch64 provides the PSTATE.SSBS mechanism to mark regions that are Speculative Store Bypassing Safe, and the MSR and MRS instructions to directly read and write the PSTATE.SSBS field

All other values are reserved.

BT, bits [3:0]

Branch Target Identification mechanism support in AArch64 state. Defined values are:

`0b0000` The Branch Target Identification mechanism is not implemented.

`0b0001` The Branch Target Identification mechanism is implemented.

All other values are reserved.

ARMv8.5-BTI implements the functionality identified by the value `0b0001`.

From Armv8.5, the only permitted value is `0b0001`.

Accessing the ID_AA64PFR1_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, ID_AA64PFR1_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0100	0b001

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_AA64PFR1_EL1;
elsif PSTATE.EL == EL2 then
    return ID_AA64PFR1_EL1;
elsif PSTATE.EL == EL3 then
    return ID_AA64PFR1_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.66 ID_AFR0_EL1, AArch32 Auxiliary Feature Register 0

The ID_AFR0_EL1 characteristics are:

Purpose

Provides information about the IMPLEMENTATION DEFINED features of the PE in AArch32 state.

Must be interpreted with the Main ID Register, MIDR_EL1.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

AArch64 System register ID_AFR0_EL1[31:0] is architecturally mapped to AArch32 System register ID_AFR0[31:0].

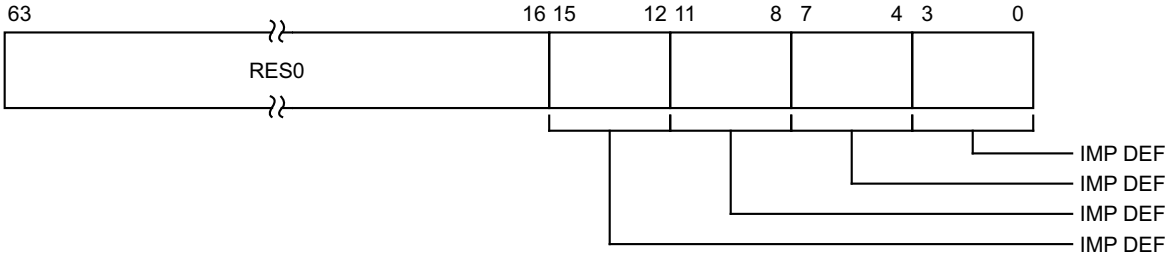
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_AFR0_EL1 are UNKNOWN.

Attributes

ID_AFR0_EL1 is a 64-bit register.

Field descriptions

The ID_AFR0_EL1 bit assignments are:



Bits [63:16]

Reserved, RES0.

IMPLEMENTATION DEFINED, bits [15:12]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED, bits [11:8]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED, bits [7:4]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED, bits [3:0]

IMPLEMENTATION DEFINED.

Accessing the ID_AFR0_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, ID_AFR0_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0001	0b011

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_AFR0_EL1;
elsif PSTATE.EL == EL2 then
    return ID_AFR0_EL1;
elsif PSTATE.EL == EL3 then
    return ID_AFR0_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.67 ID_DFR0_EL1, AArch32 Debug Feature Register 0

The ID_DFR0_EL1 characteristics are:

Purpose

Provides top level information about the debug system in AArch32 state.

Must be interpreted with the Main ID Register, MIDR_EL1.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

AArch64 System register ID_DFR0_EL1[31:0] is architecturally mapped to AArch32 System register ID_DFR0[31:0].

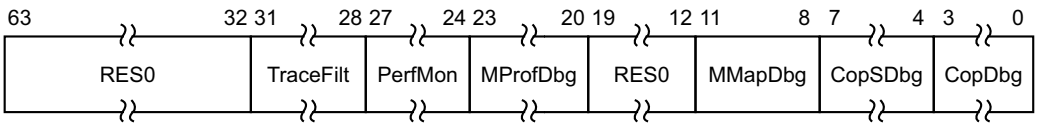
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_DFR0_EL1 are UNKNOWN.

Attributes

ID_DFR0_EL1 is a 64-bit register.

Field descriptions

The ID_DFR0_EL1 bit assignments are:



Bits [63:32]

Reserved, RES0.

TraceFilt, bits [31:28]

Armv8.4 Self-hosted Trace Extension version. Defined values are:

`0b0000` Armv8.4 Self-hosted Trace Extension not implemented.

`0b0001` Armv8.4 Self-hosted Trace Extension implemented.

All other values are reserved.

ARMv8.4-Trace implements the functionality added by the value `0b0001`.

From Armv8.3, the permitted values are `0b0000` and `0b0001`.

PerfMon, bits [27:24]

Performance Monitors Extension version.

This field does not follow the standard ID scheme, but uses the Alternative ID scheme described in *Alternative ID scheme used for the Performance Monitors Extension version* on page D13-2875.

Defined values are:

`0b0000` Performance Monitors Extension not implemented.

`0b0001` Performance Monitors Extension version 1 implemented, PMUv1.

`0b0010` Performance Monitors Extension version 2 implemented, PMUv2.

`0b0011` Performance Monitors Extension version 3 implemented, PMUv3.

`0b0100` PMUv3 for Armv8.1. As `0b0011`, and also includes support for:

- Extended 16-bit PMEVTYPER<n>.evtCount field.
- If EL2 is implemented, the HDCR.HPMD control bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b0101` PMUv3 for Armv8.4. As `0b0100` and also includes support for the PMMIR register.

`0b0110` PMUv3 for Armv8.5. As `0b0101` and also includes support for:

- 64-bit event counters.
- If EL2 is implemented, the HDCR.HCCD control bit.
- If EL3 is implemented, the SDCR.SCCD control bit.

`0b1111` IMPLEMENTATION DEFINED form of performance monitors supported, PMUv3 not supported. Arm does not recommend this value in new implementations.

ARMv8.1-PMU implements the functionality added by the value `0b0100`.

ARMv8.4-PMU implements the functionality added by the value `0b0101`.

ARMv8.5-PMU implements the functionality added by the value `0b0110`.

All other values are reserved.

In any Armv8 implementation, the values `0b0001` and `0b0010` are not permitted.

From Armv8.1, the value `0b0011` is not permitted.

From Armv8.4, the value `0b0100` is not permitted.

From Armv8.5, the value `0b0101` is not permitted.

Note

In Armv7, the value `0b0000` can mean that PMUv1 is implemented. PMUv1 is not permitted in an Armv8 implementation.

MProfDbg, bits [23:20]

M Profile Debug. Support for memory-mapped debug model for M profile processors. Defined values are:

`0b0000` Not supported.

`0b0001` Support for M profile Debug architecture, with memory-mapped access.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0000`.

Bits [19:12]

Reserved, RES0.

MMapDbg, bits [11:8]

Memory Mapped Debug. Support for v7 memory-mapped debug model, for A and R profile processors.

In Armv8-A, this field is RES0.

The optional memory map defined by Armv8 is not compatible with Armv7.

CopSDbg, bits [7:4]

Support for a System registers-based Secure debug model, using registers in the coproc = `0b1110` encoding space, for an A profile processor that includes EL3.

If EL3 is not implemented and the implemented Security state is Non-secure state, this field is RES0. Otherwise, this field reads the same as bits [3:0].

CopDbg, bits [3:0]

Support for System registers-based debug model, using registers in the coproc == `0b1110` encoding space, for A and R profile processors. Defined values are:

`0b0000` Not supported.

`0b0010` Support for Armv6, v6 Debug architecture, with System registers access.

`0b0011` Support for Armv6, v6.1 Debug architecture, with System registers access.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b0100` Support for Armv7, v7 Debug architecture, with System registers access.

`0b0101` Support for Armv7, v7.1 Debug architecture, with System registers access.

`0b0110` Support for Armv8 debug architecture, with System registers access.

`0b0111` Support for Armv8 debug architecture, with System registers access, and Virtualization Host extensions.

`0b1000` Support for Armv8.2 debug architecture.

`0b1001` Support for Armv8.4 debug architecture.

All other values are reserved.

In any Armv8 implementation, the values `0b0000`, `0b0010`, `0b0011`, `0b0100`, and `0b0101` are not permitted.

If ARMv8.1-VHE is not implemented, the only permitted value is `0b0110`.

In an Armv8.0 implementation, the value `0b1000` is not permitted.

Accessing the ID_DFR0_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ID_DFR0_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0001	0b010

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_DFR0_EL1;
elsif PSTATE.EL == EL2 then
    return ID_DFR0_EL1;
elsif PSTATE.EL == EL3 then
    return ID_DFR0_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.68 ID_DFR1_EL1, Debug Feature Register 1

The ID_DFR1_EL1 characteristics are:

Purpose

Provides top level information about the debug system in AArch32.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

AArch64 System register ID_DFR1_EL1[31:0] is architecturally mapped to AArch32 System register ID_DFR1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_DFR1_EL1 are RES0.

Note

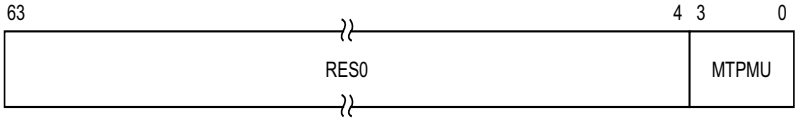
Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.

Attributes

ID_DFR1_EL1 is a 64-bit register.

Field descriptions

The ID_DFR1_EL1 bit assignments are:



Bits [63:4]

Reserved, RES0.

MTPMU, bits [3:0]

Multi-threaded PMU extension. Defined values are:

`0b0000` ARMv8.6-MTPMU not implemented. If PMUv3 is implemented, it is IMPLEMENTATION DEFINED whether PMEVTYPER<n>_EL0.MT are read/write or RES0.

`0b0001` ARMv8.6-MTPMU implemented and PMEVTYPER<n>_EL0.MT are read/write. When ARMv8.6-MTPMU is disabled, the Effective values of PMEVTYPER<n>.MT are 0.

`0b1111` ARMv8.6-MTPMU not implemented. If PMUv3 is implemented, PMEVTYPER<n>_EL0.MT are RES0.

All other values are reserved.

ARMv8.6-MTPMU implements the functionality identified by the value `0b0001`.

In an Armv8.6-compliant implementation that includes PMUv3, the value `0b0000` is not permitted.

In an implementation that does not include PMUv3, the value `0b0001` is not permitted.

Accessing the ID_DFR1_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, ID_DFR1_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0011	0b101

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!IsZero(ID_DFR1_EL1) || boolean IMPLEMENTATION_DEFINED
"ID_DFR1 trapped by HCR_EL2.TID3") && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_DFR1_EL1;
elsif PSTATE.EL == EL2 then
    return ID_DFR1_EL1;
elsif PSTATE.EL == EL3 then
    return ID_DFR1_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.69 ID_ISAR0_EL1, AArch32 Instruction Set Attribute Register 0

The ID_ISAR0_EL1 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, ID_ISAR4_EL1, and ID_ISAR5_EL1.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

AArch64 System register ID_ISAR0_EL1[31:0] is architecturally mapped to AArch32 System register ID_ISAR0[31:0].

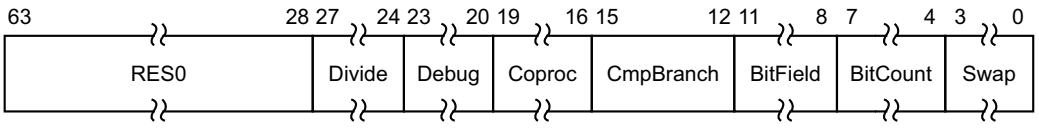
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_ISAR0_EL1 are UNKNOWN.

Attributes

ID_ISAR0_EL1 is a 64-bit register.

Field descriptions

The ID_ISAR0_EL1 bit assignments are:



Bits [63:28]

Reserved, RES0.

Divide, bits [27:24]

Indicates the implemented Divide instructions. Defined values are:

`0b0000` None implemented.

`0b0001` Adds SDIV and UDIV in the T32 instruction set.

`0b0010` As for `0b0001`, and adds SDIV and UDIV in the A32 instruction set.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0010`.

Debug, bits [23:20]

Indicates the implemented Debug instructions. Defined values are:

`0b0000` None implemented.

`0b0001` Adds BKPT.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0001`.

Coproc, bits [19:16]

Indicates the implemented System register access instructions. Defined values are:

`0b0000` None implemented, except for instructions separately attributed by the architecture to provide access to AArch32 System registers and System instructions.

`0b0001` Adds generic CDP, LDC, MCR, MRC, and STC.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b0010` As for `0b0001`, and adds generic CDP2, LDC2, MCR2, MRC2, and STC2.

`0b0011` As for `0b0010`, and adds generic MCRR and MRRC.

`0b0100` As for `0b0011`, and adds generic MCRR2 and MRRC2.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0000`.

CmpBranch, bits [15:12]

Indicates the implemented combined Compare and Branch instructions in the T32 instruction set. Defined values are:

`0b0000` None implemented.

`0b0001` Adds CBNZ and CBZ.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0001`.

BitField, bits [11:8]

Indicates the implemented BitField instructions. Defined values are:

`0b0000` None implemented.

`0b0001` Adds BFC, BFI, SBFX, and UBFX.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0001`.

BitCount, bits [7:4]

Indicates the implemented Bit Counting instructions. Defined values are:

`0b0000` None implemented.

`0b0001` Adds CLZ.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0001`.

Swap, bits [3:0]

Indicates the implemented Swap instructions in the A32 instruction set. Defined values are:

`0b0000` None implemented.

`0b0001` Adds SWP and SWPB.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0000`.

Accessing the ID_ISAR0_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ID_ISAR0_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0010	0b000

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_ISAR0_EL1;
elsif PSTATE.EL == EL2 then
    return ID_ISAR0_EL1;
elsif PSTATE.EL == EL3 then
    return ID_ISAR0_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.70 ID_ISAR1_EL1, AArch32 Instruction Set Attribute Register 1

The ID_ISAR1_EL1 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, ID_ISAR4_EL1, and ID_ISAR5_EL1.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

AArch64 System register ID_ISAR1_EL1[31:0] is architecturally mapped to AArch32 System register ID_ISAR1[31:0].

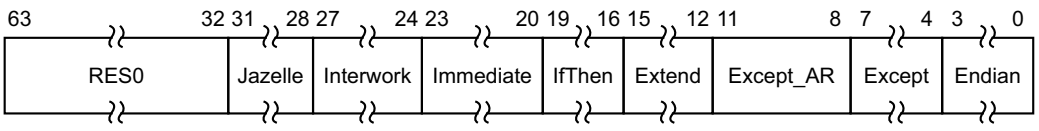
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_ISAR1_EL1 are UNKNOWN.

Attributes

ID_ISAR1_EL1 is a 64-bit register.

Field descriptions

The ID_ISAR1_EL1 bit assignments are:



Bits [63:32]

Reserved, RES0.

Jazelle, bits [31:28]

Indicates the implemented Jazelle extension instructions. Defined values are:

`0b0000` No support for Jazelle.

`0b0001` Adds the BXJ instruction and the J bit in the PSR. This setting might indicate a trivial implementation of the Jazelle extension.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0001`.

Interwork, bits [27:24]

Indicates the implemented Interworking instructions. Defined values are:

`0b0000` None implemented.

`0b0001` Adds the BX instruction, and the T bit in the PSR.

`0b0010` As for `0b0001`, and adds the BLX instruction. PC loads have BX-like behavior.

`0b0011` As for `0b0010`, and guarantees that data-processing instructions in the A32 instruction set with the PC as the destination and the S bit clear have BX-like behavior.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0011`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Immediate, bits [23:20]

Indicates the implemented data-processing instructions with long immediates. Defined values are:

`0b0000` None implemented.

`0b0001` Adds:

- The MOVT instruction.
- The MOV instruction encodings with zero-extended 16-bit immediates.
- The T32 ADD and SUB instruction encodings with zero-extended 12-bit immediates, and the other ADD, ADR, and SUB encodings cross-referenced by the pseudocode for those encodings.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0001`.

IfThen, bits [19:16]

Indicates the implemented If-Then instructions in the T32 instruction set. Defined values are:

`0b0000` None implemented.

`0b0001` Adds the IT instructions, and the IT bits in the PSRs.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0001`.

Extend, bits [15:12]

Indicates the implemented Extend instructions. Defined values are:

`0b0000` No scalar sign-extend or zero-extend instructions are implemented, where scalar instructions means non-Advanced SIMD instructions.

`0b0001` Adds the SXTB, SXTH, UXTB, and UXTH instructions.

`0b0010` As for `0b0001`, and adds the SXTB16, SXTAB, SXTAB16, SXTAH, UXTB16, UXTAB, UXTAB16, and UXTAH instructions.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0010`.

Except_AR, bits [11:8]

Indicates the implemented A and R profile exception-handling instructions. Defined values are:

`0b0000` None implemented.

`0b0001` Adds the SRS and RFE instructions, and the A and R profile forms of the CPS instruction.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0001`.

Except, bits [7:4]

Indicates the implemented exception-handling instructions in the A32 instruction set. Defined values are:

`0b0000` Not implemented. This indicates that the User bank and Exception return forms of the LDM and STM instructions are not implemented.

`0b0001` Adds the LDM (exception return), LDM (user registers), and STM (user registers) instruction versions.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0001`.

Endian, bits [3:0]

Indicates the implemented Endian instructions. Defined values are:

`0b0000` None implemented.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b0001` Adds the SETEND instruction, and the E bit in the PSRs.

All other values are reserved.

In Armv8-A, the permitted values are `0b0000` and `0b0001`.

Accessing the ID_ISAR1_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ID_ISAR1_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0010	0b001

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_ISAR1_EL1;
elsif PSTATE.EL == EL2 then
    return ID_ISAR1_EL1;
elsif PSTATE.EL == EL3 then
    return ID_ISAR1_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.71 ID_ISAR2_EL1, AArch32 Instruction Set Attribute Register 2

The ID_ISAR2_EL1 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR3_EL1, ID_ISAR4_EL1, and ID_ISAR5_EL1.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

AArch64 System register ID_ISAR2_EL1[31:0] is architecturally mapped to AArch32 System register ID_ISAR2[31:0].

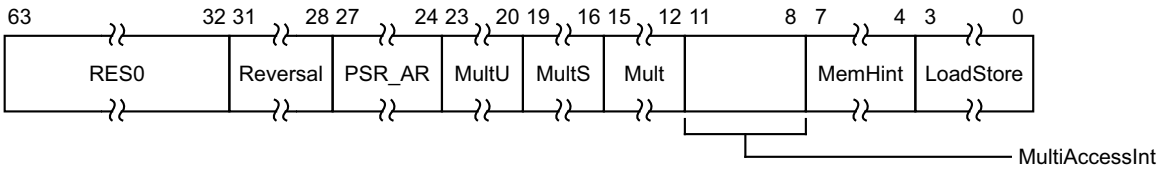
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_ISAR2_EL1 are UNKNOWN.

Attributes

ID_ISAR2_EL1 is a 64-bit register.

Field descriptions

The ID_ISAR2_EL1 bit assignments are:



Bits [63:32]

Reserved, RES0.

Reversal, bits [31:28]

Indicates the implemented Reversal instructions. Defined values are:

`0b0000` None implemented.

`0b0001` Adds the REV, REV16, and REVSH instructions.

`0b0010` As for `0b0001`, and adds the RBIT instruction.

All other values are reserved.

In Armv8-A the only permitted value is `0b0010`.

PSR_AR, bits [27:24]

Indicates the implemented A and R profile instructions to manipulate the PSR. Defined values are:

`0b0000` None implemented.

`0b0001` Adds the MRS and MSR instructions, and the exception return forms of data-processing instructions.

All other values are reserved.

In Armv8-A the only permitted value is `0b0001`.

The exception return forms of the data-processing instructions are:

- In the A32 instruction set, data-processing instructions with the PC as the destination and the S bit set. These instructions might be affected by the WithShifts attribute.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- In the T32 instruction set, the SUBS PC,LR,#N instruction.

MultU, bits [23:20]

Indicates the implemented advanced unsigned Multiply instructions. Defined values are:

`0b0000` None implemented.

`0b0001` Adds the UMULL and UMLAL instructions.

`0b0010` As for `0b0001`, and adds the UMAAL instruction.

All other values are reserved.

In Armv8-A the only permitted value is `0b0010`.

MultS, bits [19:16]

Indicates the implemented advanced signed Multiply instructions. Defined values are:

`0b0000` None implemented.

`0b0001` Adds the SMULL and SMLAL instructions.

`0b0010` As for `0b0001`, and adds the SMLABB, SMLABT, SMLALBB, SMLALBT, SMLALTB, SMLALTT, SMLATB, SMLATT, SMLAWB, SMLAWT, SMULBB, SMULBT, SMULTB, SMULTT, SMULWB, and SMULWT instructions. Also adds the Q bit in the PSRs.

`0b0011` As for `0b0010`, and adds the SMLAD, SMLADX, SMLALD, SMLALDX, SMLSD, SMLSDX, SMLSLD, SMLSLDX, SMMLA, SMMLAR, SMMLS, SMMLSR, SMMUL, SMMULR, SMUAD, SMUADX, SMUSD, and SMUSDX instructions.

All other values are reserved.

In Armv8-A the only permitted value is `0b0011`.

Mult, bits [15:12]

Indicates the implemented additional Multiply instructions. Defined values are:

`0b0000` No additional instructions implemented. This means only MUL is implemented.

`0b0001` Adds the MLA instruction.

`0b0010` As for `0b0001`, and adds the MLS instruction.

All other values are reserved.

In Armv8-A the only permitted value is `0b0010`.

MultiAccessInt, bits [11:8]

Indicates the support for interruptible multi-access instructions. Defined values are:

`0b0000` No support. This means the LDM and STM instructions are not interruptible.

`0b0001` LDM and STM instructions are restartable.

`0b0010` LDM and STM instructions are continuable.

All other values are reserved.

In Armv8-A the only permitted value is `0b0000`.

MemHint, bits [7:4]

Indicates the implemented Memory Hint instructions. Defined values are:

`0b0000` None implemented.

`0b0001` Adds the PLD instruction.

`0b0010` Adds the PLD instruction. (`0b0001` and `0b0010` have identical effects.)

`0b0011` As for `0b0001` (or `0b0010`), and adds the PLI instruction.

`0b0100` As for `0b0011`, and adds the PLDW instruction.

All other values are reserved.

In Armv8-A the only permitted value is `0b0100`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

LoadStore, bits [3:0]

Indicates the implemented additional load/store instructions. Defined values are:

`0b0000` No additional load/store instructions implemented.

`0b0001` Adds the LDRD and STRD instructions.

`0b0010` As for `0b0001`, and adds the Load Acquire (LDAB, LDAH, LDA, LDAEXB, LDAEXH, LDAEX, LDAEXD) and Store Release (STLB, STLH, STL, STLEXB, STLEXH, STLEX, STLEXD) instructions.

All other values are reserved.

In Armv8-A the only permitted value is `0b0010`.

Accessing the ID_ISAR2_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ID_ISAR2_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0010	0b010

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_ISAR2_EL1;
elsif PSTATE.EL == EL2 then
    return ID_ISAR2_EL1;
elsif PSTATE.EL == EL3 then
    return ID_ISAR2_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.72 ID_ISAR3_EL1, AArch32 Instruction Set Attribute Register 3

The ID_ISAR3_EL1 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR4_EL1, and ID_ISAR5_EL1.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

AArch64 System register ID_ISAR3_EL1[31:0] is architecturally mapped to AArch32 System register ID_ISAR3[31:0].

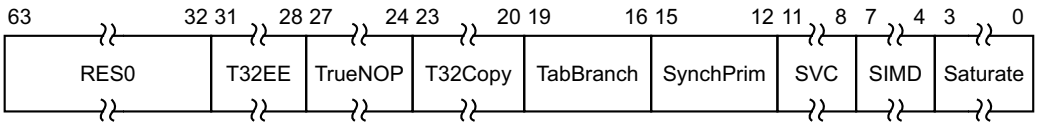
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_ISAR3_EL1 are UNKNOWN.

Attributes

ID_ISAR3_EL1 is a 64-bit register.

Field descriptions

The ID_ISAR3_EL1 bit assignments are:



Bits [63:32]

Reserved, RES0.

T32EE, bits [31:28]

Indicates the implemented T32EE instructions. Defined values are:

`0b0000` None implemented.

`0b0001` Adds the ENTERX and LEAVEX instructions, and modifies the load behavior to include null checking.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0000`.

TrueNOP, bits [27:24]

Indicates the implemented true NOP instructions. Defined values are:

`0b0000` None implemented. This means there are no NOP instructions that do not have any register dependencies.

`0b0001` Adds true NOP instructions in both the T32 and A32 instruction sets. This also permits additional NOP-compatible hints.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0001`.

T32Copy, bits [23:20]

Indicates the support for T32 non flag-setting MOV instructions. Defined values are:

`0b0000` Not supported. This means that in the T32 instruction set, encoding T1 of the MOV (register) instruction does not support a copy from a low register to a low register.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b0001` Adds support for T32 instruction set encoding T1 of the MOV (register) instruction, copying from a low register to a low register.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0001`.

TabBranch, bits [19:16]

Indicates the implemented Table Branch instructions in the T32 instruction set. Defined values are:

`0b0000` None implemented.

`0b0001` Adds the TBB and TBH instructions.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0001`.

SynchPrim, bits [15:12]

Used in conjunction with ID_ISAR4.SynchPrim_frac to indicate the implemented Synchronization Primitive instructions. Defined values are:

`0b0000` If SynchPrim_frac == `0b0000`, no Synchronization Primitives implemented.

`0b0001` If SynchPrim_frac == `0b0000`, adds the LDREX and STREX instructions.
If SynchPrim_frac == `0b0011`, also adds the CLREX, LDREXB, STREXB, and STREXH instructions.

`0b0010` If SynchPrim_frac == `0b0000`, as for [`0b0001`, `0b0011`] and also adds the LDREXD and STREXD instructions.

All other combinations of SynchPrim and SynchPrim_frac are reserved.

In Armv8-A, the only permitted value is `0b0010`.

SVC, bits [11:8]

Indicates the implemented SVC instructions. Defined values are:

`0b0000` Not implemented.

`0b0001` Adds the SVC instruction.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0001`.

SIMD, bits [7:4]

Indicates the implemented SIMD instructions. Defined values are:

`0b0000` None implemented.

`0b0001` Adds the SSAT and USAT instructions, and the Q bit in the PSRs.

`0b0011` As for `0b0001`, and adds the PKHBT, PKHTB, QADD16, QADD8, QASX, QSUB16, QSUB8, QSAX, SADD16, SADD8, SASX, SEL, SHADD16, SHADD8, SHASX, SHSUB16, SHSUB8, SHSAX, SSAT16, SSUB16, SSUB8, SSAX, SXTAB16, SXTB16, UADD16, UADD8, UASX, UHADD16, UHADD8, UHASX, UHSUB16, UHSUB8, UHSAX, UQADD16, UQADD8, UQASX, UQSUB16, UQSUB8, UQSAX, USAD8, USADA8, USAT16, USUB16, USUB8, USAX, UXTAB16, and UXTB16 instructions. Also adds support for the GE[3:0] bits in the PSRs.

All other values are reserved.

In Armv8-A the only permitted value is `0b0011`.

The SIMD field relates only to implemented instructions that perform SIMD operations on the general-purpose registers. In an implementation that supports floating-point and Advanced SIMD instructions, MVFR0, MVFR1, and MVFR2 give information about the implemented Advanced SIMD instructions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Saturate, bits [3:0]

Indicates the implemented Saturate instructions. Defined values are:

`0b0000` None implemented. This means no non-Advanced SIMD saturate instructions are implemented.

`0b0001` Adds the QADD, QDADD, QDSUB, and QSUB instructions, and the Q bit in the PSRs.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0001`.

Accessing the ID_ISAR3_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ID_ISAR3_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0010	0b011

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_ISAR3_EL1;
elsif PSTATE.EL == EL2 then
    return ID_ISAR3_EL1;
elsif PSTATE.EL == EL3 then
    return ID_ISAR3_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.73 ID_ISAR4_EL1, AArch32 Instruction Set Attribute Register 4

The ID_ISAR4_EL1 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, and ID_ISAR5_EL1.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

AArch64 System register ID_ISAR4_EL1[31:0] is architecturally mapped to AArch32 System register ID_ISAR4[31:0].

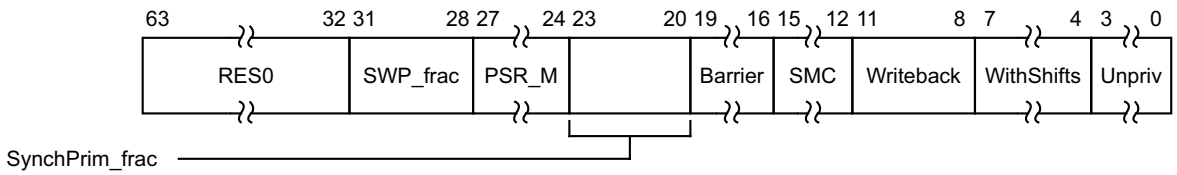
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_ISAR4_EL1 are UNKNOWN.

Attributes

ID_ISAR4_EL1 is a 64-bit register.

Field descriptions

The ID_ISAR4_EL1 bit assignments are:



Bits [63:32]

Reserved, RES0.

SWP_frac, bits [31:28]

Indicates support for the memory system locking the bus for SWP or SWPB instructions. Defined values are:

`0b0000` SWP or SWPB instructions not implemented.

`0b0001` SWP or SWPB implemented but only in a uniprocessor context. SWP and SWPB do not guarantee whether memory accesses from other masters can come between the load memory access and the store memory access of the SWP or SWPB.

All other values are reserved. This field is valid only if ID_ISAR0.Swap is `0b0000`.

In Armv8-A, the only permitted value is `0b0000`.

PSR_M, bits [27:24]

Indicates the implemented M profile instructions to modify the PSRs. Defined values are:

`0b0000` None implemented.

`0b0001` Adds the M profile forms of the CPS, MRS, and MSR instructions.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0000`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

SynchPrim_frac, bits [23:20]

Used in conjunction with ID_ISAR3.SynchPrim to indicate the implemented Synchronization Primitive instructions. Possible values are:

`0b0000` If SynchPrim == `0b0000`, no Synchronization Primitives implemented. If SynchPrim == `0b0001`, adds the LDREX and STREX instructions. If SynchPrim == `0b0010`, also adds the CLREX, LDREXB, LDREXH, STREXB, STREXH, LDREXD, and STREXD instructions.

`0b0011` If SynchPrim == `0b0001`, adds the LDREX, STREX, CLREX, LDREXB, LDREXH, STREXB, and STREXH instructions.

All other combinations of SynchPrim and SynchPrim_frac are reserved.

In Armv8-A, the only permitted value is `0b0000`.

Barrier, bits [19:16]

Indicates the implemented Barrier instructions in the A32 and T32 instruction sets. Defined values are:

`0b0000` None implemented. Barrier operations are provided only as System instructions in the (coproc==`0b1111`) encoding space.

`0b0001` Adds the DMB, DSB, and ISB barrier instructions.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0001`.

SMC, bits [15:12]

Indicates the implemented SMC instructions. Defined values are:

`0b0000` None implemented.

`0b0001` Adds the SMC instruction.

All other values are reserved.

In Armv8-A, the permitted values are `0b0001` and `0b0000`.

If EL1 cannot use AArch32, then this field has the value `0b0000`.

Writeback, bits [11:8]

Indicates the support for Writeback addressing modes. Defined values are:

`0b0000` Basic support. Only the LDM, STM, PUSH, POP, SRS, and RFE instructions support writeback addressing modes. These instructions support all of their writeback addressing modes.

`0b0001` Adds support for all of the writeback addressing modes.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0001`.

WithShifts, bits [7:4]

Indicates the support for instructions with shifts. Defined values are:

`0b0000` Nonzero shifts supported only in MOV and shift instructions.

`0b0001` Adds support for shifts of loads and stores over the range LSL 0-3.

`0b0011` As for `0b0001`, and adds support for other constant shift options, both on load/store and other instructions.

`0b0100` As for `0b0011`, and adds support for register-controlled shift options.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0100`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Unpriv, bits [3:0]

Indicates the implemented unprivileged instructions. Defined values are:

`0b0000` None implemented. No T variant instructions are implemented.

`0b0001` Adds the LDRBT, LDRT, STRBT, and STRT instructions.

`0b0010` As for `0b0001`, and adds the LDRHT, LDRSBT, LDRSHT, and STRHT instructions.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0010`.

Accessing the ID_ISAR4_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ID_ISAR4_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0010	0b100

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_ISAR4_EL1;
elsif PSTATE.EL == EL2 then
    return ID_ISAR4_EL1;
elsif PSTATE.EL == EL3 then
    return ID_ISAR4_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.74 ID_ISAR5_EL1, AArch32 Instruction Set Attribute Register 5

The ID_ISAR5_EL1 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, and ID_ISAR4_EL1.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

AArch64 System register ID_ISAR5_EL1[31:0] is architecturally mapped to AArch32 System register ID_ISAR5[31:0].

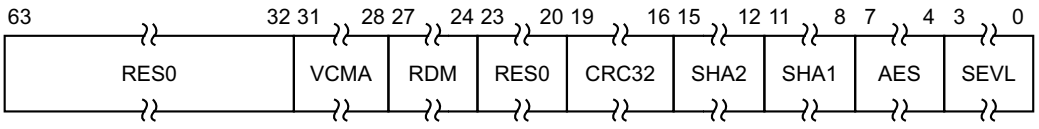
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_ISAR5_EL1 are UNKNOWN.

Attributes

ID_ISAR5_EL1 is a 64-bit register.

Field descriptions

The ID_ISAR5_EL1 bit assignments are:



Bits [63:32]

Reserved, RES0.

VCMA, bits [31:28]

Indicates AArch32 support for complex number addition and multiplication where numbers are stored in vectors. Defined values are:

`0b0000` The VCMLA and VCADD instructions are not implemented in AArch32.

`0b0001` The VCMLA and VCADD instructions are implemented in AArch32.

All other values are reserved.

ARMv8.3-CompNum implements the functionality identified by `0b0001`.

In Armv8.0, Armv8.1, and Armv8.2, the only permitted value is `0b0000`.

From Armv8.3, the only permitted value is `0b0001`.

RDM, bits [27:24]

Indicates whether the VQRDMLAH and VQRDMLSH instructions are implemented in AArch32 state. Defined values are:

`0b0000` No VQRDMLAH and VQRDMLSH instructions implemented.

`0b0001` VQRDMLAH and VQRDMLSH instructions implemented.

All other values are reserved.

ARMv8.1-RDMA implements the functionality identified by the value `0b0001`.

In Armv8.0, the only permitted value is `0b0000`.

From Armv8.1, the only permitted value is `0b0001`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bits [23:20]

Reserved, RES0.

CRC32, bits [19:16]

Indicates whether the CRC32 instructions are implemented in AArch32 state.

`0b0000` No CRC32 instructions implemented.

`0b0001` CRC32B, CRC32H, CRC32W, CRC32CB, CRC32CH, and CRC32CW instructions implemented.

All other values are reserved.

In Armv8.0, the permitted values are `0b0000` and `0b0001`.

From Armv8.1, the only permitted value is `0b0001`.

SHA2, bits [15:12]

Indicates whether the SHA2 instructions are implemented in AArch32 state.

`0b0000` No SHA2 instructions implemented.

`0b0001` SHA256H, SHA256H2, SHA256SU0, and SHA256SU1 implemented.

All other values are reserved.

In Armv8-A, the permitted values are `0b0000` and `0b0001`.

SHA1, bits [11:8]

Indicates whether the SHA1 instructions are implemented in AArch32 state.

`0b0000` No SHA1 instructions implemented.

`0b0001` SHA1C, SHA1P, SHA1M, SHA1H, SHA1SU0, and SHA1SU1 implemented.

All other values are reserved.

In Armv8-A, the permitted values are `0b0000` and `0b0001`.

AES, bits [7:4]

Indicates whether the AES instructions are implemented in AArch32 state.

`0b0000` No AES instructions implemented.

`0b0001` AESE, AESD, AESMC, and AESIMC implemented.

`0b0010` As for `0b0001`, plus VMULL (polynomial) instructions operating on 64-bit data quantities.

All other values are reserved.

In Armv8-A, the permitted values are `0b0000` and `0b0010`.

SEVL, bits [3:0]

Indicates whether the SEVL instruction is implemented in AArch32 state.

`0b0000` SEVL is implemented as a NOP.

`0b0001` SEVL is implemented as Send Event Local.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0001`.

Accessing the ID_ISAR5_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, ID_ISAR5_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0010	0b101

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_ISAR5_EL1;
elsif PSTATE.EL == EL2 then
    return ID_ISAR5_EL1;
elsif PSTATE.EL == EL3 then
    return ID_ISAR5_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.75 ID_ISAR6_EL1, AArch32 Instruction Set Attribute Register 6

The ID_ISAR6_EL1 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, ID_ISAR4_EL1 and ID_ISAR5_EL1.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

AArch64 System register ID_ISAR6_EL1[31:0] is architecturally mapped to AArch32 System register ID_ISAR6[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_ISAR6_EL1 are UNKNOWN.

Note

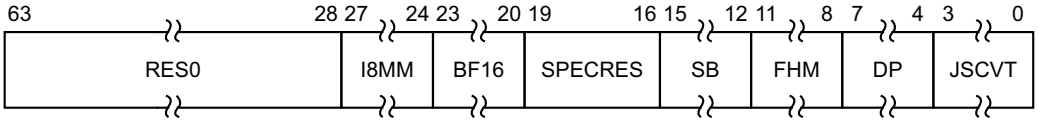
Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.

Attributes

ID_ISAR6_EL1 is a 64-bit register.

Field descriptions

The ID_ISAR6_EL1 bit assignments are:



Bits [63:28]

Reserved, RES0.

I8MM, bits [27:24]

Indicates support for Advanced SIMD and floating-point Int8 matrix multiplication instructions in AArch32 state. Defined values of this field are:

`0b0000` Int8 matrix multiplication instructions are not implemented.

`0b0001` VSMMLA, VSUDOT, VUMMLA, VUSMMLA, and VUSDOT instructions are implemented.

All other values are reserved.

ARMv8.2-AA32I8MM implements the functionality identified by `0b0001`.

BF16, bits [23:20]

Indicates support for Advanced SIMD and floating-point BFloat16 instructions in AArch32 state. Defined values are:

`0b0000` BFloat16 instructions are not implemented.

`0b0001` VCVT, VCVTB, VCVTT, VDOT, VFMAL, and VMMLA instructions with BF16 operand or result types are implemented.

All other values are reserved.

ARMv8.2-AA32BF16 implements the functionality identified by `0b0001`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

SPECRES, bits [19:16]

Indicates support for Speculation invalidation instructions in AArch32 state. Defined values are:

`0b0000` Prediction invalidation instructions are not implemented.

`0b0001` CFPRCTX, DVPRCTX, and CPPRCTX instructions are implemented.

All other values are reserved.

ARMv8.0-PredInv implements the functionality identified by `0b0001`.

From Armv8.5, the only permitted value is `0b0001`.

SB, bits [15:12]

Indicates support for the SB instruction in AArch32 state. Defined values are:

`0b0000` SB instruction is not implemented.

`0b0001` SB instruction is implemented.

All other values are reserved.

ARMv8.0-SB implements the functionality identified by `0b0001`.

From Armv8.5, the only permitted value is `0b0001`.

FHM, bits [11:8]

Indicates support for Advanced SIMD and floating-point VFMAL and VFMSL instructions in AArch32 state. Defined values are:

`0b0000` VFMAL and VMFSL instructions are not implemented.

`0b0001` VFMAL and VMFSL instructions are implemented.

All other values are reserved.

ARMv8.2-FHM implements the functionality identified by `0b0001`.

From Armv8.2, the permitted values are `0b0000` and `0b0001`.

DP, bits [7:4]

Indicates support for Advanced SIMD and floating-point VFMAL and VFMSL instructions in AArch32 state. Defined values are:

`0b0000` Dot product instructions are not implemented.

`0b0001` UDOT and VSDOT instructions are implemented.

All other values are reserved.

ARMv8.2-DotProd implements the functionality identified by `0b0001`.

In Armv8.2, the permitted values are `0b0000` and `0b0001`.

From Armv8.4, the only permitted value is `0b0001`.

JSCVT, bits [3:0]

Indicates support for the VJCVT instruction in AArch32 state. Defined values are:

`0b0000` The VJCVT instruction is not implemented.

`0b0001` The VJCVT instruction is implemented.

All other values are reserved.

ARMv8.3-JSconv implements the functionality identified by `0b0001`.

In Armv8.0, Armv8.1, and Armv8.2, the only permitted value is `0b0000`.

From Armv8.3, if Advanced SIMD or Floating-point is implemented, the only permitted value is `0b0001`.

From Armv8.3, if Advanced SIMD or Floating-point is not implemented, the only permitted value is `0b0000`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the ID_ISAR6_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ID_ISAR6_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0010	0b111

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!IsZero(ID_ISAR6_EL1) || boolean IMPLEMENTATION_DEFINED
"ID_ISAR6_EL1 trapped by HCR_EL2.TID3") && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_ISAR6_EL1;
elsif PSTATE.EL == EL2 then
    return ID_ISAR6_EL1;
elsif PSTATE.EL == EL3 then
    return ID_ISAR6_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.76 ID_MMFR0_EL1, AArch32 Memory Model Feature Register 0

The ID_MMFR0_EL1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

Must be interpreted with ID_MMFR1_EL1, ID_MMFR2_EL1, ID_MMFR3_EL1, and ID_MMFR4_EL1.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

AArch64 System register ID_MMFR0_EL1[31:0] is architecturally mapped to AArch32 System register ID_MMFR0[31:0].

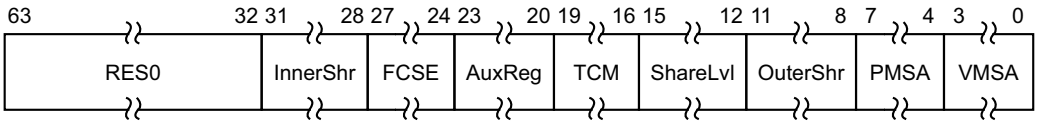
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_MMFR0_EL1 are UNKNOWN.

Attributes

ID_MMFR0_EL1 is a 64-bit register.

Field descriptions

The ID_MMFR0_EL1 bit assignments are:



Bits [63:32]

Reserved, RES0.

InnerShr, bits [31:28]

Innermost Shareability. Indicates the innermost shareability domain implemented. Defined values are:

`0b0000` Implemented as Non-cacheable.

`0b0001` Implemented with hardware coherency support.

`0b1111` Shareability ignored.

All other values are reserved.

From Armv8 the permitted values are `0b0000`, `0b0001`, and `0b1111`.

This field is valid only if the implementation supports two levels of shareability, as indicated by ID_MMFR0_EL1.ShareLvl having the value `0b0001`.

When ID_MMFR0_EL1.ShareLvl is zero, this field is UNKNOWN.

FCSE, bits [27:24]

Indicates whether the implementation includes the FCSE. Defined values are:

`0b0000` Not supported.

`0b0001` Support for FCSE.

All other values are reserved.

From Armv8 the only permitted value is `0b0000`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

AuxReg, bits [23:20]

Auxiliary Registers. Indicates support for Auxiliary registers. Defined values are:

`0b0000` None supported.

`0b0001` Support for Auxiliary Control Register only.

`0b0010` Support for Auxiliary Fault Status Registers (AIFSR and ADFSR) and Auxiliary Control Register.

All other values are reserved.

From Armv8 the only permitted value is `0b0010`.

Note

Accesses to unimplemented Auxiliary registers are UNDEFINED.

TCM, bits [19:16]

Indicates support for TCMs and associated DMAs. Defined values are:

`0b0000` Not supported.

`0b0001` Support is IMPLEMENTATION DEFINED. Armv7 requires this setting.

`0b0010` Support for TCM only, Armv6 implementation.

`0b0011` Support for TCM and DMA, Armv6 implementation.

All other values are reserved.

In Armv8-A the only permitted value is `0b0000`.

ShareLvl, bits [15:12]

Shareability Levels. Indicates the number of shareability levels implemented. Defined values are:

`0b0000` One level of shareability implemented.

`0b0001` Two levels of shareability implemented.

All other values are reserved.

From Armv8 the only permitted value is `0b0001`.

OuterShr, bits [11:8]

Outermost Shareability. Indicates the outermost shareability domain implemented. Defined values are:

`0b0000` Implemented as Non-cacheable.

`0b0001` Implemented with hardware coherency support.

`0b1111` Shareability ignored.

All other values are reserved.

From Armv8 the permitted values are `0b0000`, `0b0001`, and `0b1111`.

PMSA, bits [7:4]

Indicates support for a PMSA. Defined values are:

`0b0000` Not supported.

`0b0001` Support for IMPLEMENTATION DEFINED PMSA.

`0b0010` Support for PMSAv6, with a Cache Type Register implemented.

`0b0011` Support for PMSAv7, with support for memory subsections. Armv7-R profile.

All other values are reserved.

In Armv8-A the only permitted value is `0b0000`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

VMSA, bits [3:0]

Indicates support for a VMSA. Defined values are:

`0b0000` Not supported.

`0b0001` Support for IMPLEMENTATION DEFINED VMSA.

`0b0010` Support for VMSAv6, with Cache and TLB Type Registers implemented.

`0b0011` Support for VMSAv7, with support for remapping and the Access flag. Armv7-A profile.

`0b0100` As for `0b0011`, and adds support for the PXN bit in the Short-descriptor translation table format descriptors.

`0b0101` As for `0b0100`, and adds support for the Long-descriptor translation table format.

All other values are reserved.

In Armv8-A the only permitted value is `0b0101`.

Accessing the ID_MMFR0_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ID_MMFR0_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0001	0b100

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_MMFR0_EL1;
elsif PSTATE.EL == EL2 then
    return ID_MMFR0_EL1;
elsif PSTATE.EL == EL3 then
    return ID_MMFR0_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.77 ID_MMFR1_EL1, AArch32 Memory Model Feature Register 1

The ID_MMFR1_EL1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

Must be interpreted with ID_MMFR0_EL1, ID_MMFR2_EL1, ID_MMFR3_EL1, and ID_MMFR4_EL1.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

AArch64 System register ID_MMFR1_EL1[31:0] is architecturally mapped to AArch32 System register ID_MMFR1[31:0].

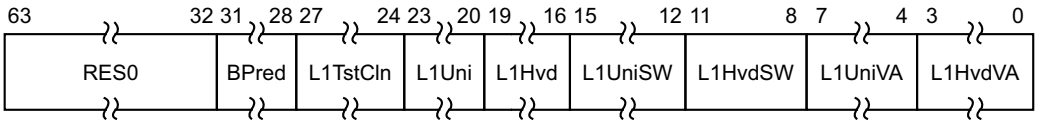
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_MMFR1_EL1 are UNKNOWN.

Attributes

ID_MMFR1_EL1 is a 64-bit register.

Field descriptions

The ID_MMFR1_EL1 bit assignments are:



Bits [63:32]

Reserved, RES0.

BPred, bits [31:28]

Branch Predictor. Indicates branch predictor management requirements. Defined values are:

`0b0000` No branch predictor, or no MMU present. Implies a fixed MPU configuration.

`0b0001` Branch predictor requires flushing on:

- Enabling or disabling a stage of address translation.
- Writing new data to instruction locations.
- Writing new mappings to the translation tables.
- Changes to the TTBR0, TTBR1, or TTBCR registers.
- Changes to the ContextID or ASID, or to the FCSE ProcessID if this is supported.

`0b0010` Branch predictor requires flushing on:

- Enabling or disabling a stage of address translation.
- Writing new data to instruction locations.
- Writing new mappings to the translation tables.
- Any change to the TTBR0, TTBR1, or TTBCR registers without a change to the corresponding ContextID or ASID, or FCSE ProcessID if this is supported.

`0b0011` Branch predictor requires flushing only on writing new data to instruction locations.

`0b0100` For execution correctness, branch predictor requires no flushing at any time.

All other values are reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In Armv8-A the permitted values are `0b0010`, `0b0011`, or `0b0100`. For values other than `0b0000` and `0b0100` the Arm Architecture Reference Manual, or the product documentation, might give more information about the required maintenance.

L1TstCln, bits [27:24]

Level 1 cache Test and Clean. Indicates the supported Level 1 data cache test and clean operations, for Harvard or unified cache implementations. Defined values are:

`0b0000` None supported.

`0b0001` Supported Level 1 data cache test and clean operations are:

- Test and clean data cache.

`0b0010` As for 0001, and adds:

- Test, clean, and invalidate data cache.

All other values are reserved.

In Armv8-A the only permitted value is `0b0000`.

L1Uni, bits [23:20]

Level 1 Unified cache. Indicates the supported entire Level 1 cache maintenance operations for a unified cache implementation. Defined values are:

`0b0000` None supported.

`0b0001` Supported entire Level 1 cache operations are:

- Invalidate cache, including branch predictor if appropriate.
- Invalidate branch predictor, if appropriate.

`0b0010` As for 0001, and adds:

- Clean cache, using a recursive model that uses the cache dirty status bit.
- Clean and invalidate cache, using a recursive model that uses the cache dirty status bit.

All other values are reserved.

In Armv8-A the only permitted value is `0b0000`.

L1Hvd, bits [19:16]

Level 1 Harvard cache. Indicates the supported entire Level 1 cache maintenance operations for a Harvard cache implementation. Defined values are:

`0b0000` None supported.

`0b0001` Supported entire Level 1 cache operations are:

- Invalidate instruction cache, including branch predictor if appropriate.
- Invalidate branch predictor, if appropriate.

`0b0010` As for 0001, and adds:

- Invalidate data cache.
- Invalidate data cache and instruction cache, including branch predictor if appropriate.

`0b0011` As for 0010, and adds:

- Clean data cache, using a recursive model that uses the cache dirty status bit.
- Clean and invalidate data cache, using a recursive model that uses the cache dirty status bit.

All other values are reserved.

In Armv8-A the only permitted value is `0b0000`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

L1UniSW, bits [15:12]

Level 1 Unified cache by Set/Way. Indicates the supported Level 1 cache line maintenance operations by set/way, for a unified cache implementation. Defined values are:

`0b0000` None supported.

`0b0001` Supported Level 1 unified cache line maintenance operations by set/way are:

- Clean cache line by set/way.

`0b0010` As for 0001, and adds:

- Clean and invalidate cache line by set/way.

`0b0011` As for 0010, and adds:

- Invalidate cache line by set/way.

All other values are reserved.

In Armv8-A the only permitted value is `0b0000`.

L1HvdSW, bits [11:8]

Level 1 Harvard cache by Set/Way. Indicates the supported Level 1 cache line maintenance operations by set/way, for a Harvard cache implementation. Defined values are:

`0b0000` None supported.

`0b0001` Supported Level 1 Harvard cache line maintenance operations by set/way are:

- Clean data cache line by set/way.
- Clean and invalidate data cache line by set/way.

`0b0010` As for 0001, and adds:

- Invalidate data cache line by set/way.

`0b0011` As for 0010, and adds:

- Invalidate instruction cache line by set/way.

All other values are reserved.

In Armv8-A the only permitted value is `0b0000`.

L1UniVA, bits [7:4]

Level 1 Unified cache by Virtual Address. Indicates the supported Level 1 cache line maintenance operations by VA, for a unified cache implementation. Defined values are:

`0b0000` None supported.

`0b0001` Supported Level 1 unified cache line maintenance operations by VA are:

- Clean cache line by VA.
- Invalidate cache line by VA.
- Clean and invalidate cache line by VA.

`0b0010` As for 0001, and adds:

- Invalidate branch predictor by VA, if branch predictor is implemented.

All other values are reserved.

In Armv8-A the only permitted value is `0b0000`.

L1HvdVA, bits [3:0]

Level 1 Harvard cache by Virtual Address. Indicates the supported Level 1 cache line maintenance operations by VA, for a Harvard cache implementation. Defined values are:

`0b0000` None supported.

`0b0001` Supported Level 1 Harvard cache line maintenance operations by VA are:

- Clean data cache line by VA.
- Invalidate data cache line by VA.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Clean and invalidate data cache line by VA.
- Clean instruction cache line by VA.

`0b0010` As for 0001, and adds:

- Invalidate branch predictor by VA, if branch predictor is implemented.

All other values are reserved.

In Armv8-A the only permitted value is `0b0000`.

Accessing the ID_MMFR1_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ID_MMFR1_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0001	0b101

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_MMFR1_EL1;
elsif PSTATE.EL == EL2 then
    return ID_MMFR1_EL1;
elsif PSTATE.EL == EL3 then
    return ID_MMFR1_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.78 ID_MMFR2_EL1, AArch32 Memory Model Feature Register 2

The ID_MMFR2_EL1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

Must be interpreted with ID_MMFR0_EL1, ID_MMFR1_EL1, ID_MMFR3_EL1, and ID_MMFR4_EL1.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

AArch64 System register ID_MMFR2_EL1[31:0] is architecturally mapped to AArch32 System register ID_MMFR2[31:0].

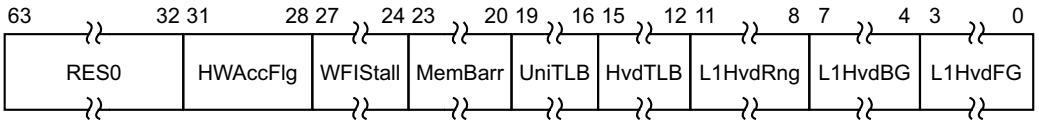
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_MMFR2_EL1 are UNKNOWN.

Attributes

ID_MMFR2_EL1 is a 64-bit register.

Field descriptions

The ID_MMFR2_EL1 bit assignments are:



Bits [63:32]

Reserved, RES0.

HWAccFlg, bits [31:28]

Hardware Access Flag. In earlier versions of the Arm Architecture, this field indicates support for a Hardware Access flag, as part of the VMSAv7 implementation. Defined values are:

`0b0000` Not supported.

`0b0001` Support for VMSAv7 Access flag, updated in hardware.

All other values are reserved.

From Armv8 the only permitted value is `0b0000`.

WFIStall, bits [27:24]

Wait For Interrupt Stall. Indicates the support for Wait For Interrupt (WFI) stalling. Defined values are:

`0b0000` Not supported.

`0b0001` Support for WFI stalling.

All other values are reserved.

From Armv8 the permitted values are `0b0000` and `0b0001`.

MemBarr, bits [23:20]

Memory Barrier. Indicates the supported memory barrier System instructions in the (coproc==`0b1111`) encoding space:

`0b0000` None supported.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b0001` Supported memory barrier System instructions are:

- Data Synchronization Barrier (DSB).

`0b0010` As for 0001, and adds:

- Instruction Synchronization Barrier (ISB).
- Data Memory Barrier (DMB).

All other values are reserved.

From Armv8 the only permitted value is `0b0010`.

Arm deprecates the use of these operations. ID_ISAR4.Barrier_instrs indicates the level of support for the preferred barrier instructions.

UniTLB, bits [19:16]

Unified TLB. Indicates the supported TLB maintenance operations, for a unified TLB implementation. Defined values are:

`0b0000` Not supported.

`0b0001` Supported unified TLB maintenance operations are:

- Invalidate all entries in the TLB.
- Invalidate TLB entry by VA.

`0b0010` As for 0001, and adds:

- Invalidate TLB entries by ASID match.

`0b0011` As for 0010, and adds:

- Invalidate instruction TLB and data TLB entries by VA All ASID. This is a shared unified TLB operation.

`0b0100` As for 0011, and adds:

- Invalidate Hyp mode unified TLB entry by VA.
- Invalidate entire Non-secure PL1&0 unified TLB.
- Invalidate entire Hyp mode unified TLB.

`0b0101` As for `0b0100`, and adds the following operations: TLBIMVALIS, TLBIMVAALIS, TLBIMVALHIS, TLBIMVAL, TLBIMVAAL, TLBIMVALH.

`0b0110` As for `0b0101`, and adds the following operations: TLBIIPAS2IS, TLBIIPAS2LIS, TLBIIPAS2, TLBIIPAS2L.

All other values are reserved.

In Armv8-A the only permitted value is `0b0110`.

HvdTLB, bits [15:12]

If the Unified TLB field (UniTLB, bits [19:16]) is not 0000, then the meaning of this field is IMPLEMENTATION DEFINED. Arm deprecates the use of this field by software.

L1HvdRng, bits [11:8]

Level 1 Harvard cache Range. Indicates the supported Level 1 cache maintenance range operations, for a Harvard cache implementation. Defined values are:

`0b0000` Not supported.

`0b0001` Supported Level 1 Harvard cache maintenance range operations are:

- Invalidate data cache range by VA.
- Invalidate instruction cache range by VA.
- Clean data cache range by VA.
- Clean and invalidate data cache range by VA.

All other values are reserved.

From Armv8 the only permitted value is `0b0000`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

L1HvdBG, bits [7:4]

Level 1 Harvard cache Background fetch. Indicates the supported Level 1 cache background fetch operations, for a Harvard cache implementation. When supported, background fetch operations are non-blocking operations. Defined values are:

`0b0000` Not supported.

`0b0001` Supported Level 1 Harvard cache background fetch operations are:

- Fetch instruction cache range by VA.
- Fetch data cache range by VA.

All other values are reserved.

From Armv8 the only permitted value is `0b0000`.

L1HvdFG, bits [3:0]

Level 1 Harvard cache Foreground fetch. Indicates the supported Level 1 cache foreground fetch operations, for a Harvard cache implementation. When supported, foreground fetch operations are blocking operations. Defined values are:

`0b0000` Not supported.

`0b0001` Supported Level 1 Harvard cache foreground fetch operations are:

- Fetch instruction cache range by VA.
- Fetch data cache range by VA.

All other values are reserved.

From Armv8 the only permitted value is `0b0000`.

Accessing the ID_MMFR2_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ID_MMFR2_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0001	0b110

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_MMFR2_EL1;
elsif PSTATE.EL == EL2 then
    return ID_MMFR2_EL1;
elsif PSTATE.EL == EL3 then
    return ID_MMFR2_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.79 ID_MMFR3_EL1, AArch32 Memory Model Feature Register 3

The ID_MMFR3_EL1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

Must be interpreted with ID_MMFR0_EL1, ID_MMFR1_EL1, ID_MMFR2_EL1, and ID_MMFR4_EL1.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

AArch64 System register ID_MMFR3_EL1[31:0] is architecturally mapped to AArch32 System register ID_MMFR3[31:0].

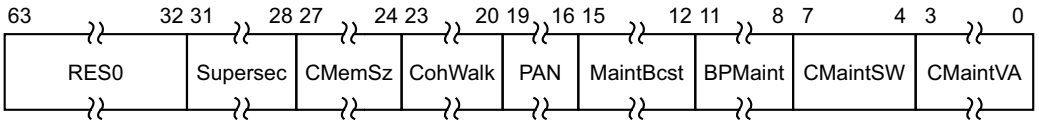
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_MMFR3_EL1 are UNKNOWN.

Attributes

ID_MMFR3_EL1 is a 64-bit register.

Field descriptions

The ID_MMFR3_EL1 bit assignments are:



Bits [63:32]

Reserved, RES0.

Supersec, bits [31:28]

Supersections. On a VMSA implementation, indicates whether Supersections are supported. Defined values are:

`0b0000` Supersections supported.

`0b1111` Supersections not supported.

All other values are reserved.

In Armv8-A the permitted values are `0b0000` and `0b1111`.

CMemSz, bits [27:24]

Cached Memory Size. Indicates the physical memory size supported by the caches. Defined values are:

`0b0000` 4GB, corresponding to a 32-bit physical address range.

`0b0001` 64GB, corresponding to a 36-bit physical address range.

`0b0010` 1TB or more, corresponding to a 40-bit or larger physical address range.

All other values are reserved.

In Armv8-A the permitted values are `0b0000`, `0b0001`, and `0b0010`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CohWalk, bits [23:20]

Coherent Walk. Indicates whether Translation table updates require a clean to the Point of Unification. Defined values are:

`0b0000` Updates to the translation tables require a clean to the Point of Unification to ensure visibility by subsequent translation table walks.

`0b0001` Updates to the translation tables do not require a clean to the Point of Unification to ensure visibility by subsequent translation table walks.

All other values are reserved.

In Armv8-A the only permitted value is `0b0001`.

PAN, bits [19:16]

Privileged Access Never. Indicates support for the PAN bit in CPSR, SPSR, and DSPSR in AArch32 state. Defined values are:

`0b0000` PAN not supported.

`0b0001` PAN supported.

`0b0010` PAN supported and ATS1CPRP and ATS1CPWP instructions supported.

All other values are reserved.

ARMv8.1-PAN implements the functionality identified by the value `0b0001`.

ARMv8.2-ATS1E1 implements the functionality added by the value `0b0010`.

In Armv8.1 the value `0b0000` is not permitted.

From Armv8.2, the only permitted value is `0b0010`.

MaintBcst, bits [15:12]

Maintenance Broadcast. Indicates whether Cache, TLB, and branch predictor operations are broadcast. Defined values are:

`0b0000` Cache, TLB, and branch predictor operations only affect local structures.

`0b0001` Cache and branch predictor operations affect structures according to shareability and defined behavior of instructions. TLB operations only affect local structures.

`0b0010` Cache, TLB, and branch predictor operations affect structures according to shareability and defined behavior of instructions.

All other values are reserved.

In Armv8-A the only permitted value is `0b0010`.

BPMaint, bits [11:8]

Branch Predictor Maintenance. Indicates the supported branch predictor maintenance operations in an implementation with hierarchical cache maintenance operations. Defined values are:

`0b0000` None supported.

`0b0001` Supported branch predictor maintenance operations are:

- Invalidate all branch predictors.

`0b0010` As for 0001, and adds:

- Invalidate branch predictors by VA.

All other values are reserved.

In Armv8-A the only permitted value is `0b0010`.

CMaintSW, bits [7:4]

Cache Maintenance by Set/Way. Indicates the supported cache maintenance operations by set/way, in an implementation with hierarchical caches. Defined values are:

`0b0000` None supported.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b0001` Supported hierarchical cache maintenance instructions by set/way are:

- Invalidate data cache by set/way.
- Clean data cache by set/way.
- Clean and invalidate data cache by set/way.

All other values are reserved.

In Armv8-A the only permitted value is `0b0001`.

In a unified cache implementation, the data cache maintenance operations apply to the unified caches.

CMaintVA, bits [3:0]

Cache Maintenance by Virtual Address. Indicates the supported cache maintenance operations by VA, in an implementation with hierarchical caches. Defined values are:

`0b0000` None supported.

`0b0001` Supported hierarchical cache maintenance operations by VA are:

- Invalidate data cache by VA.
- Clean data cache by VA.
- Clean and invalidate data cache by VA.
- Invalidate instruction cache by VA.
- Invalidate all instruction cache entries.

All other values are reserved.

In Armv8-A the only permitted value is `0b0001`.

In a unified cache implementation, data cache maintenance operations apply to the unified caches, and the instruction cache maintenance instructions are not implemented.

Accessing the ID_MMFR3_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ID_MMFR3_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0001	0b111

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_MMFR3_EL1;
elsif PSTATE.EL == EL2 then
    return ID_MMFR3_EL1;
elsif PSTATE.EL == EL3 then
    return ID_MMFR3_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.80 ID_MMFR4_EL1, AArch32 Memory Model Feature Register 4

The ID_MMFR4_EL1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

Must be interpreted with ID_MMFR0_EL1, ID_MMFR1_EL1, ID_MMFR2_EL1, and ID_MMFR3_EL1.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

AArch64 System register ID_MMFR4_EL1[31:0] is architecturally mapped to AArch32 System register ID_MMFR4[31:0].

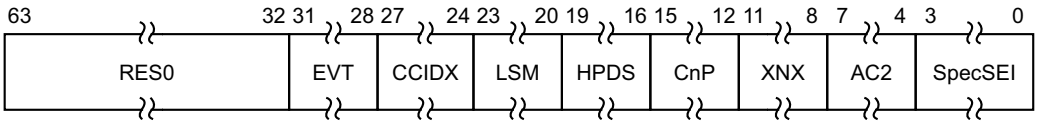
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_MMFR4_EL1 are UNKNOWN.

Attributes

ID_MMFR4_EL1 is a 64-bit register.

Field descriptions

The ID_MMFR4_EL1 bit assignments are:



Bits [63:32]

Reserved, RES0.

EVT, bits [31:28]

Enhanced Virtualization Traps. If EL2 is implemented, indicates support for the HCR2.{TTLBIS, TOCU, TICAB, TID4} traps. Defined values are:

`0b0000` HCR2.{TTLBIS, TOCU, TICAB, TID4} traps are not supported.

`0b0001` HCR2.{TOCU, TICAB, TID4} traps are supported. HCR2.TTLBIS trap is not supported.

`0b0010` HCR2.{TTLBIS, TOCU, TICAB, TID4} traps are supported.

All other values are reserved.

ARMv8.2-EVT implements the functionality identified by the values `0b0001` and `0b0010`.

If EL2 is not implemented, the only permitted value is `0b0000`.

From Armv8.1, the permitted values are `0b0000` and `0b0001`.

From Armv8.5, if EL2 is implemented, the only permitted value is `0b0010`.

CCIDX, bits [27:24]

Support for use of the revised CCSIDR format and the presence of the CCSIDR2 is indicated. Defined values are:

`0b0000` 32-bit format implemented for all levels of the CCSIDR, and the CCSIDR2 register is not implemented.

`0b0001` 64-bit format implemented for all levels of the CCSIDR, and the CCSIDR2 register is implemented.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

All other values are reserved.

ARMv8.3-CCIDX implements the functionality identified by `0b0001`.

From Armv8.3, the permitted values are `0b0000` and `0b0001`.

LSM, bits [23:20]

Indicates support for LSMAOE and nTLSMD bits in HSCTLR and SCTLR. Defined values are:

`0b0000` LSMAOE and nTLSMD bits not supported.

`0b0001` LSMAOE and nTLSMD bits supported.

All other values are reserved.

ARMv8.2-LSMAOC implements the functionality identified by the value `0b0001`.

From Armv8.2, the permitted values are `0b0000` and `0b0001`.

HPDS, bits [19:16]

Hierarchical permission disables bits in translation tables. Defined values are:

`0b0000` Disabling of hierarchical controls not supported.

`0b0001` Supports disabling of hierarchical controls using the TTBCR2.HPD0, TTBCR2.HPD1, and HTCR.HPD bits.

`0b0010` As for value `0b0001`, and adds possible hardware allocation of bits[62:59] of the translation table descriptors from the final lookup level for IMPLEMENTATION DEFINED use.

All other values are reserved.

ARMv8.2-AA32HPD implements the functionality identified by the value `0b0001`.

ARMv8.2-TTPBHA implements the functionality added by the value `0b0010`.

Note

The value `0b0000` implies that the encoding for TTBCR2 is UNDEFINED.

CnP, bits [15:12]

Common not Private translations. Defined values are:

`0b0000` Common not Private translations not supported.

`0b0001` Common not Private translations supported.

All other values are reserved.

ARMv8.2-TTCNP implements the functionality identified by the value `0b0001`.

From Armv8.2 the only permitted value is `0b0001`.

XNX, bits [11:8]

Support for execute-never control distinction by Exception level at stage 2. Defined values are:

`0b0000` Distinction between EL0 and EL1 execute-never control at stage 2 not supported.

`0b0001` Distinction between EL0 and EL1 execute-never control at stage 2 supported.

All other values are reserved.

ARMv8.2-TTS2UXN implements the functionality identified by the value `0b0001`.

When ARMv8.2-TTS2UXN is implemented:

- If all of the following conditions are true, it is IMPLEMENTATION DEFINED whether the value of ID_MMFR4_EL1.XNX is `0b0000` or `0b0001`:
  - ID_AA64MMFR1_EL1.XNX ==1.
  - EL2 cannot use AArch32.
  - EL1 can use AArch32.
- If EL2 can use AArch32 then the only permitted value is `0b0001`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

AC2, bits [7:4]

Indicates the extension of the ACTLR and HACTLR registers using ACTLR2 and HACTLR2. Defined values are:

`0b0000` ACTLR2 and HACTLR2 are not implemented.

`0b0001` ACTLR2 and HACTLR2 are implemented.

All other values are reserved.

In Armv8.0 and Armv8.1 the permitted values are `0b0000` and `0b0001`.

From Armv8.2, the only permitted value is `0b0001`.

SpecSEI, bits [3:0]

Describes whether the PE can generate SError interrupt exceptions from speculative reads of memory, including speculative instruction fetches. The defined values of this field are:

`0b0000` The PE never generates an SError interrupt due to an External abort on a speculative read.

`0b0001` The PE might generate an SError interrupt due to an External abort on a speculative read.

All other values are reserved.

Accessing the ID_MMFR4_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ID_MMFR4_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0010	0b110

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!IsZero(ID_MMFR4_EL1) || boolean IMPLEMENTATION_DEFINED
"ID_MMFR4_EL1 trapped by HCR_EL2.TID3") && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_MMFR4_EL1;
elsif PSTATE.EL == EL2 then
    return ID_MMFR4_EL1;
elsif PSTATE.EL == EL3 then
    return ID_MMFR4_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.81 ID_MMFR5_EL1, AArch32 Memory Model Feature Register 5

The ID_MMFR5_EL1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

AArch64 System register ID_MMFR5_EL1[31:0] is architecturally mapped to AArch32 System register ID_MMFR5[31:0].

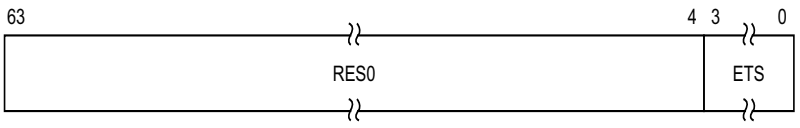
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_MMFR5_EL1 are UNKNOWN.

Attributes

ID_MMFR5_EL1 is a 64-bit register.

Field descriptions

The ID_MMFR5_EL1 bit assignments are:



Bits [63:4]

Reserved, RES0.

ETS, bits [3:0]

Support for Enhanced Translation Synchronization. Defined values are:

`0b0000` Enhanced Translation Synchronization is not supported.

`0b0001` Enhanced Translation Synchronization is supported.

All other values are reserved.

ARMv8.0-ETS implements the functionality identified by the value `0b0001`.

From Armv8.0, the permitted values are `0b0000` and `0b0001`.

Accessing the ID_MMFR5_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ID_MMFR5_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0011	0b110

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!IsZero(ID_MMFR5_EL1) || boolean IMPLEMENTATION_DEFINED
"ID_MMFR5_EL1 trapped by HCR_EL2.TID3") && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_MMFR5_EL1;
elsif PSTATE.EL == EL2 then
    return ID_MMFR5_EL1;
elsif PSTATE.EL == EL3 then
    return ID_MMFR5_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.82 ID_PFR0_EL1, AArch32 Processor Feature Register 0

The ID_PFR0_EL1 characteristics are:

Purpose

Gives top-level information about the instruction sets supported by the PE in AArch32 state.

Must be interpreted with ID_PFR1_EL1.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

AArch64 System register ID_PFR0_EL1[31:0] is architecturally mapped to AArch32 System register ID_PFR0[31:0].

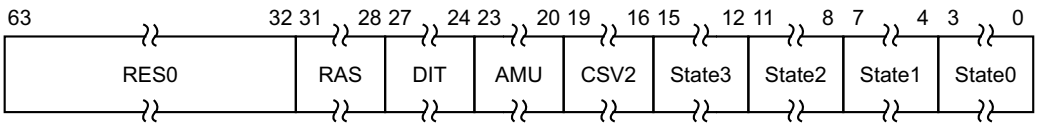
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_PFR0_EL1 are UNKNOWN.

Attributes

ID_PFR0_EL1 is a 64-bit register.

Field descriptions

The ID_PFR0_EL1 bit assignments are:



Bits [63:32]

Reserved, RES0.

RAS, bits [31:28]

RAS Extension version. Defined values are:

`0b0000` No RAS Extension.

`0b0001` RAS Extension present.

`0b0010` ARMv8.4-RAS present. As `0b0001`, and adds support for additional ERXMISC<m> System registers.

Error records accessed through System registers conform to RAS System Architecture v1.1, which includes simplifications to ERR<n>STATUS and support for the optional RAS Timestamp Extension.

All other values are reserved.

ARMv8.4-RAS implements the functionality identified by the value `0b0010`.

In Armv8.0 and Armv8.1, the permitted values are `0b0000` and `0b0001`.

In Armv8.2, the only permitted value is `0b0001`.

From Armv8.4, when ARMv8.4-DFE is not implemented, and ERRIDR.NUM is zero, the permitted values are IMPLEMENTATION DEFINED `0b0001` or `0b0010`. Otherwise from Armv8.4 the only permitted value is `0b0010`.

DIT, bits [27:24]

Data Independent Timing. Defined values are:

`0b0000` AArch32 does not guarantee constant execution time of any instructions.

`0b0001` AArch32 provides the CPSR.DIT mechanism to guarantee constant execution time of certain instructions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

All other values are reserved.

ARMv8.4-DIT implements the functionality identified by the value `0b0001`.

From Armv8.4, the only permitted value is `0b0001`.

AMU, bits [23:20]

Activity Monitors Extension. Defined values are:

`0b0000` Activity Monitors Extension is not implemented.

`0b0001` AMUv1 for Armv8.4 is implemented.

`0b0010` AMUv1 for Armv8.6 is implemented. As `0b0001` and adds support for virtualization of the activity monitor event counters.

All other values are reserved.

AMUv1 implements the functionality identified by the value `0b0001`.

ARMv8.6-AMU implements the functionality identified by the value `0b0010`.

CSV2, bits [19:16]

Speculative use of out of context branch targets. Defined values are:

`0b0000` This Device does not disclose whether branch targets trained in one hardware described context can affect speculative execution in a different hardware described context.

`0b0001` Branch targets trained in one hardware described context can only affect speculative execution in a different hardware described context in a hard-to-determine way.

All other values are reserved.

ARMv8.0-CSV2 implements the functionality identified by `0b0001`.

From Armv8.5, the only permitted value is `0b0001`.

State3, bits [15:12]

T32EE instruction set support. Defined values are:

`0b0000` Not implemented.

`0b0001` T32EE instruction set implemented.

All other values are reserved.

In Armv8-A the only permitted value is `0b0000`.

State2, bits [11:8]

Jazelle extension support. Defined values are:

`0b0000` Not implemented.

`0b0001` Jazelle extension implemented, without clearing of JOSCR.CV on exception entry.

`0b0010` Jazelle extension implemented, with clearing of JOSCR.CV on exception entry.

All other values are reserved.

In Armv8-A the only permitted value is `0b0001`.

State1, bits [7:4]

T32 instruction set support. Defined values are:

`0b0000` T32 instruction set not implemented.

`0b0001` T32 encodings before the introduction of Thumb-2 technology implemented:

- All instructions are 16-bit.
- A BL or BLX is a pair of 16-bit instructions.
- 32-bit instructions other than BL and BLX cannot be encoded.

`0b0011` T32 encodings after the introduction of Thumb-2 technology implemented, for all 16-bit and 32-bit T32 basic instructions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

All other values are reserved.

In Armv8-A the only permitted value is `0b0011`.

State0, bits [3:0]

A32 instruction set support. Defined values are:

`0b0000` A32 instruction set not implemented.

`0b0001` A32 instruction set implemented.

All other values are reserved.

In Armv8-A the only permitted value is `0b0001`.

Accessing the ID_PFR0_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ID_PFR0_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0001	0b000

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_PFR0_EL1;
elsif PSTATE.EL == EL2 then
    return ID_PFR0_EL1;
elsif PSTATE.EL == EL3 then
    return ID_PFR0_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.83 ID_PFR1_EL1, AArch32 Processor Feature Register 1

The ID_PFR1_EL1 characteristics are:

Purpose

Gives information about the AArch32 programmers' model.

Must be interpreted with ID_PFR0_EL1.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

AArch64 System register ID_PFR1_EL1[31:0] is architecturally mapped to AArch32 System register ID_PFR1[31:0].

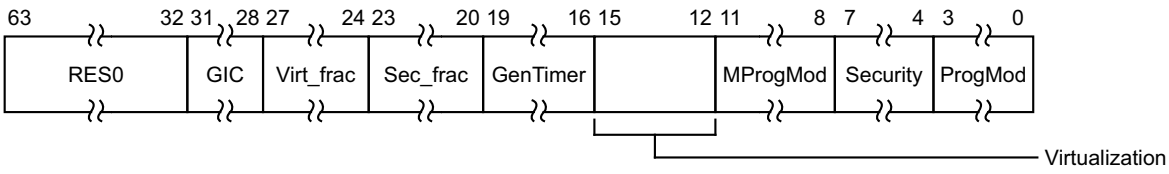
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_PFR1_EL1 are UNKNOWN.

Attributes

ID_PFR1_EL1 is a 64-bit register.

Field descriptions

The ID_PFR1_EL1 bit assignments are:



Bits [63:32]

Reserved, RES0.

GIC, bits [31:28]

System register GIC CPU interface. Defined values are:

`0b0000` GIC CPU interface system registers not implemented.

`0b0001` System register interface to versions 3.0 and 4.0 of the GIC CPU interface is supported.

`0b0011` System register interface to version 4.1 of the GIC CPU interface is supported.

All other values are reserved.

Virt_frac, bits [27:24]

Virtualization fractional field. When the Virtualization field is `0b0000`, determines the support for features from the ARMv7 Virtualization Extensions. Defined values are:

`0b0000` No features from the ARMv7 Virtualization Extensions are implemented.

`0b0001` The following features of the ARMv7 Virtualization Extensions are implemented:

- The SCR.SIF bit, if EL3 is implemented.
- The modifications to the SCR.AW and SCR.FW bits described in the Virtualization Extensions, if EL3 is implemented.
- The MSR (banked register) and MRS (banked register) instructions.
- The ERET instruction.

All other values are reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In Armv8-A the permitted values are:

- `0b0000` when EL2 is implemented.
- `0b0001` when EL2 is not implemented.

This field is only valid when the value of ID_PFR1_EL1.Virtualization is 0, otherwise it holds the value `0b0000`.

Note

The ID_ISAR registers do not identify whether the instructions added by the ARMv7 Virtualization Extensions are implemented.

Sec_frac, bits [23:20]

Security fractional field. When the Security field is `0b0000`, determines the support for features from the ARMv7 Security Extensions. Defined values are:

`0b0000` No features from the ARMv7 Security Extensions are implemented.

`0b0001` The following features from the ARMv7 Security Extensions are implemented:

- The VBAR register.
- The TTBCR.PD0 and TTBCR.PD1 bits.

`0b0010` As for `0b0001`, plus the ability to access Secure or Non-secure physical memory is supported.

All other values are reserved.

In Armv8-A the permitted values are:

- `0b0000` when EL3 is implemented.
- `0b0001` or `0b0010` when EL3 is not implemented.

This field is only valid when the value of ID_PFR1_EL1.Security is 0, otherwise it holds the value `0b0000`.

GenTimer, bits [19:16]

Generic Timer support. Defined values are:

`0b0000` Generic Timer is not implemented.

`0b0001` Generic Timer is implemented.

`0b0010` Generic Timer is implemented, and also includes support for CNTHCTL.EVNTIS and CNTKCTL.EVNTIS fields, and CNTPCTSS and CNTVCTSS counter views.

All other values are reserved.

ARMv8.6-ECV implements the functionality identified by the value `0b0010`.

In Armv8.0, Armv8.1, Armv8.2, Armv8.3, Armv8.4, and Armv8.5, the only permitted value is `0b0001`.

From Armv8.6, the only permitted value is `0b0010`.

Virtualization, bits [15:12]

Virtualization support. Defined values are:

`0b0000` EL2, Hyp mode, and the HVC instruction not implemented.

`0b0001` EL2, Hyp mode, the HVC instruction, and all the features described by Virt_frac == `0b0001` implemented.

All other values are reserved.

In Armv8-A the permitted values are:

- `0b0000` when EL2 is not implemented.
- `0b0001` when EL2 is implemented.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In an implementation that includes EL2, if EL2 cannot use AArch32 but EL1 can use AArch32 then this field has the value `0b0001`.

If EL1 cannot use AArch32 then this field has the value `0b0000`.

Note

The ID_ISARs do not identify whether the HVC instruction is implemented.

MProgMod, bits [11:8]

M profile programmers' model support. Defined values are:

`0b0000` Not supported.

`0b0010` Support for two-stack programmers' model.

All other values are reserved.

In Armv8-A the only permitted value is `0b0000`.

Security, bits [7:4]

Security support. Defined values are:

`0b0000` EL3, Monitor mode, and the SMC instruction not implemented.

`0b0001` EL3, Monitor mode, the SMC instruction, and all the features described by Sec_frac == `0b0001` implemented.

`0b0010` As for `0b0001`, and adds the ability to set the NSACR.RFR bit. Not permitted in Armv8 as the NSACR.RFR bit is RES0.

All other values are reserved.

In Armv8-A the permitted values are:

- `0b0000` when EL3 is not implemented.
- `0b0001` when EL3 is implemented.

In an implementation that includes EL3, if EL3 cannot use AArch32 but EL1 can use AArch32 then this field has the value `0b0001`.

If EL1 cannot use AArch32 then this field has the value `0b0000`.

ProgMod, bits [3:0]

Support for the standard programmers' model for Armv4 and later. Model must support User, FIQ, IRQ, Supervisor, Abort, Undefined, and System modes. Defined values are:

`0b0000` Not supported.

`0b0001` Supported.

All other values are reserved.

In Armv8-A the permitted values are `0b0001` and `0b0000`.

If EL1 cannot use AArch32 then this field has the value `0b0000`.

Accessing the ID_PFR1_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ID_PFR1_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0001	0b001

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_PFR1_EL1;
elsif PSTATE.EL == EL2 then
    return ID_PFR1_EL1;
elsif PSTATE.EL == EL3 then
    return ID_PFR1_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.84 ID_PFR2_EL1, AArch32 Processor Feature Register 2

The ID_PFR2_EL1 characteristics are:

Purpose

Gives information about the AArch32 programmers' model.

Must be interpreted with ID_PFR0_EL1 and ID_PFR1_EL1.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

AArch64 System register ID_PFR2_EL1[31:0] is architecturally mapped to AArch32 System register ID_PFR2[31:0].

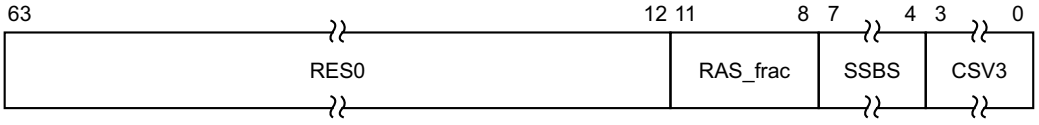
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_PFR2_EL1 are UNKNOWN.

Attributes

ID_PFR2_EL1 is a 64-bit register.

Field descriptions

The ID_PFR2_EL1 bit assignments are:



Bits [63:12]

Reserved, RES0.

RAS_frac, bits [11:8]

RAS Extension fractional field. Defined values are:

`0b0000` If ID_PFR0_EL1.RAS == `0b0001`, RAS Extension implemented.

`0b0001` If ID_PFR0_EL1.RAS == `0b0001`, as `0b0000` and adds support for additional ERXMISC<m> System registers.

Error records accessed through System registers conform to RAS System Architecture v1.1, which includes simplifications to ERR<n>STATUS and support for the optional RAS Timestamp Extension.

All other values are reserved.

This field is valid only if ID_PFR0_EL1.RAS == `0b0001`.

SSBS, bits [7:4]

Speculative Store Bypassing controls in AArch64 state. Defined values are:

`0b0000` AArch32 provides no mechanism to control the use of Speculative Store Bypassing.

`0b0001` AArch32 provides the PSTATE.SSBS mechanism to mark regions that are Speculative Store Bypass Safe.

From Armv8.0, the permitted values are `0b0000` and `0b0001`.

From Armv8.5, the only permitted value is `0b0001`.

All other values are reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CSV3, bits [3:0]

Speculative use of faulting data. Defined values are:

`0b0000` This Device does not disclose whether data loaded under speculation with a permission or domain fault can be used to form an address or generate condition codes or SVE predicate values to be used by instructions newer than the load in the speculative sequence

`0b0001` Data loaded under speculation with a permission or domain fault cannot be used to form an address or generate condition codes or SVE predicate values to be used by instructions newer than the load in the speculative sequence

From Armv8.0, the permitted values are `0b0000` and `0b0001`.

From Armv8.5, the only permitted value is `0b0001`.

All other values are reserved.

Accessing the ID_PFR2_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ID_PFR2_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0011	0b100

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_PFR2_EL1;
elsif PSTATE.EL == EL2 then
    return ID_PFR2_EL1;
elsif PSTATE.EL == EL3 then
    return ID_PFR2_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.85 IFSR32_EL2, Instruction Fault Status Register (EL2)

The IFSR32_EL2 characteristics are:

Purpose

Allows access to the AArch32 IFSR register from AArch64 state only. Its value has no effect on execution in AArch64 state.

Configurations

AArch64 System register IFSR32_EL2[31:0] is architecturally mapped to AArch32 System register IFSR[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to IFSR32_EL2 are UNDEFINED.

If EL2 is not implemented but EL3 is implemented, and EL1 is capable of using AArch32, then this register is not RES0.

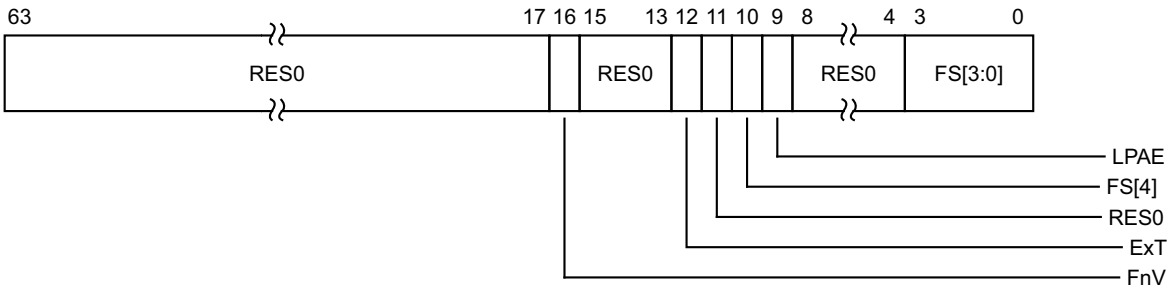
Attributes

IFSR32_EL2 is a 64-bit register.

Field descriptions

The IFSR32_EL2 bit assignments are:

When TTBCR.EAE == 0:



Bits [63:17]

Reserved, RES0.

FnV, bit [16]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

`0b0` IFAR is valid.

`0b1` IFAR is not valid, and holds an UNKNOWN value.

This field is only valid for a synchronous External abort other than a synchronous External abort on a translation table walk. It is RES0 for all other Prefetch Abort exceptions.

This field resets to an architecturally UNKNOWN value.

Bits [15:13]

Reserved, RES0.

ExT, bit [12]

External abort type. This bit can be used to provide an IMPLEMENTATION DEFINED classification of External aborts.

In an implementation that does not provide any classification of External aborts, this bit is RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For aborts other than External aborts this bit always returns 0.

This field resets to an architecturally UNKNOWN value.

Bit [11]

Reserved, RES0.

FS[4], bit [10]

This field is bit[4] of FS[4:0].

Fault Status bits. Bits [10] and [3:0] are interpreted together.

`0b00001` PC alignment fault.

`0b00010` Debug exception.

`0b00011` Access flag fault, level 1.

`0b00101` Translation fault, level 1.

`0b00110` Access flag fault, level 2.

`0b00111` Translation fault, level 2.

`0b01000` Synchronous External abort, not on translation table walk.

`0b01001` Domain fault, level 1.

`0b01011` Domain fault, level 2.

`0b01100` Synchronous External abort, on translation table walk, level 1.

`0b01101` Permission fault, level 1.

`0b01110` Synchronous External abort, on translation table walk, level 2.

`0b01111` Permission fault, level 2.

`0b10000` TLB conflict abort.

`0b10100` IMPLEMENTATION DEFINED fault (Lockdown fault).

`0b11001` **When RAS is not implemented** Synchronous parity or ECC error on memory access, not on translation table walk.

`0b11100` **When RAS is not implemented** Synchronous parity or ECC error on translation table walk, level 1.

`0b11110` **When RAS is not implemented** Synchronous parity or ECC error on translation table walk, level 2.

All other values are reserved.

For more information about the lookup level associated with a fault, see *The level associated with MMU faults on a Short-descriptor translation table lookup* on page G5-6030.

The FS field is split as follows:

- FS[4] is IFSR32_EL2[10].
- FS[3:0] is IFSR32_EL2[3:0].

This field resets to an architecturally UNKNOWN value.

LPAE, bit [9]

On taking a Data Abort exception, this bit is set as follows:

`0b0` Using the Short-descriptor translation table formats.

`0b1` Using the Long-descriptor translation table formats.

Hardware does not interpret this bit to determine the behavior of the memory system, and therefore software can set this bit to 0 or 1 without affecting operation.

This field resets to an architecturally UNKNOWN value.

Bits [8:4]

Reserved, RES0.

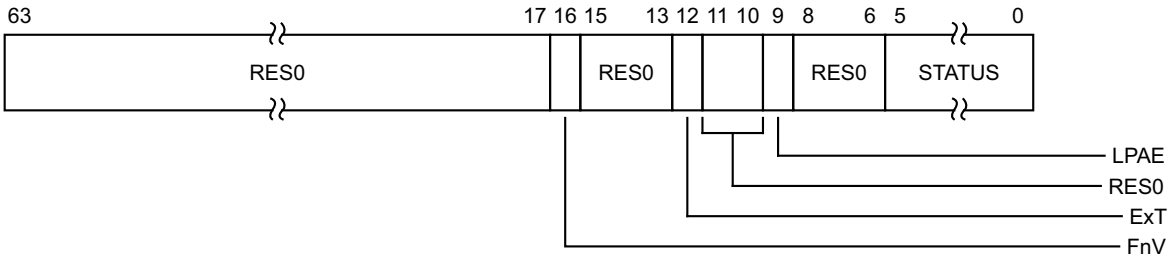
Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

FS[3:0], bits [3:0]

This field is bits[3:0] of FS[4:0].

See FS[4] for the field description.

When TTBCR.EAE == 1:



Bits [63:17]

Reserved, RES0.

FnV, bit [16]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

`0b0` IFAR is valid.

`0b1` IFAR is not valid, and holds an UNKNOWN value.

This field is only valid for a synchronous External abort other than a synchronous External abort on a translation table walk. It is RES0 for all other Prefetch Abort exceptions.

This field resets to an architecturally UNKNOWN value.

Bits [15:13]

Reserved, RES0.

ExT, bit [12]

External abort type. This bit can be used to provide an IMPLEMENTATION DEFINED classification of External aborts.

In an implementation that does not provide any classification of External aborts, this bit is RES0.

For aborts other than External aborts this bit always returns 0.

This field resets to an architecturally UNKNOWN value.

Bits [11:10]

Reserved, RES0.

LPAE, bit [9]

On taking a Data Abort exception, this bit is set as follows:

`0b0` Using the Short-descriptor translation table formats.

`0b1` Using the Long-descriptor translation table formats.

Hardware does not interpret this bit to determine the behavior of the memory system, and therefore software can set this bit to 0 or 1 without affecting operation.

This field resets to an architecturally UNKNOWN value.

Bits [8:6]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

STATUS, bits [5:0]

Fault status bits. Possible values of this field are:

`0b000000` Address size fault in translation table base register.

`0b000001` Address size fault, level 1.

`0b000010` Address size fault, level 2.

`0b000011` Address size fault, level 3.

`0b000101` Translation fault, level 1.

`0b000110` Translation fault, level 2.

`0b000111` Translation fault, level 3.

`0b001001` Access flag fault, level 1.

`0b001010` Access flag fault, level 2.

`0b001011` Access flag fault, level 3.

`0b001101` Permission fault, level 1.

`0b001110` Permission fault, level 2.

`0b001111` Permission fault, level 3.

`0b010000` Synchronous External abort, not on translation table walk.

`0b010101` Synchronous External abort, on translation table walk, level 1.

`0b010110` Synchronous External abort, on translation table walk, level 2.

`0b010111` Synchronous External abort, on translation table walk, level 3.

`0b011000` **When RAS is not implemented** Synchronous parity or ECC error on memory access, not on translation table walk.

`0b011101` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk, level 1.

`0b011110` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk, level 2.

`0b011111` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk, level 3.

`0b100001` PC alignment fault.

`0b100010` Debug exception.

`0b110000` TLB conflict abort.

All other values are reserved.

When the RAS Extension is implemented, `0b011000`, `0b011101`, `0b011110`, and `0b011111`, are reserved.

For more information about the lookup level associated with a fault, see *The level associated with MMU faults on a Long-descriptor translation table lookup* on page G5-6032.

This field resets to an architecturally UNKNOWN value.

Accessing the IFSR32_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, IFSR32_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return IFSR32_EL2;
elsif PSTATE.EL == EL3 then
    return IFSR32_EL2;
```

MSR IFSR32_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    IFSR32_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    IFSR32_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.86 ISR_EL1, Interrupt Status Register

The ISR_EL1 characteristics are:

Purpose

Shows the pending status of the IRQ, FIQ, or SError interrupt.

When executing at EL2, EL3 or Secure EL1 when SCR_EL3.EEL2 == `0b0`, this shows the pending status of the physical IRQ, FIQ, or SError interrupts.

When executing at either Non-secure EL1 or at Secure EL1 when SCR_EL3.EEL2 == `0b1`:

- If the HCR_EL2.{IMO,FMO,AMO} bit has a value of 1, the corresponding ISR_EL1.{I,F,A} bit shows the pending status of the virtual IRQ, FIQ, or SError.
- If the HCR_EL2.{IMO,FMO,AMO} bit has a value of 0, the corresponding ISR_EL1.{I,F,A} bit shows the pending status of the physical IRQ, FIQ, or SError.

Configurations

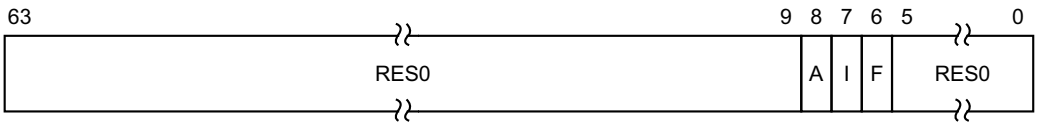
AArch64 System register ISR_EL1[31:0] is architecturally mapped to AArch32 System register ISR[31:0].

Attributes

ISR_EL1 is a 64-bit register.

Field descriptions

The ISR_EL1 bit assignments are:



Bits [63:9]

Reserved, RES0.

A, bit [8]

SError interrupt pending bit.

`0b0` No pending SError.

`0b1` An SError interrupt is pending.

If the SError interrupt is edge-triggered, this field is cleared to zero when the physical SError interrupt is taken.

I, bit [7]

IRQ pending bit. Indicates whether an IRQ interrupt is pending:

`0b0` No pending IRQ.

`0b1` An IRQ interrupt is pending.

F, bit [6]

FIQ pending bit. Indicates whether an FIQ interrupt is pending.

`0b0` No pending FIQ.

`0b1` An FIQ interrupt is pending.

Bits [5:0]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the ISR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ISR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.ISR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ISR_EL1;
elsif PSTATE.EL == EL2 then
    return ISR_EL1;
elsif PSTATE.EL == EL3 then
    return ISR_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.87 LORC_EL1, LORegion Control (EL1)

The LORC_EL1 characteristics are:

Purpose

Enables and disables LORegions, and selects the current LORegion descriptor.

Configurations

This register is present only when ARMv8.1-LOR is implemented. Otherwise, direct accesses to LORC_EL1 are UNDEFINED.

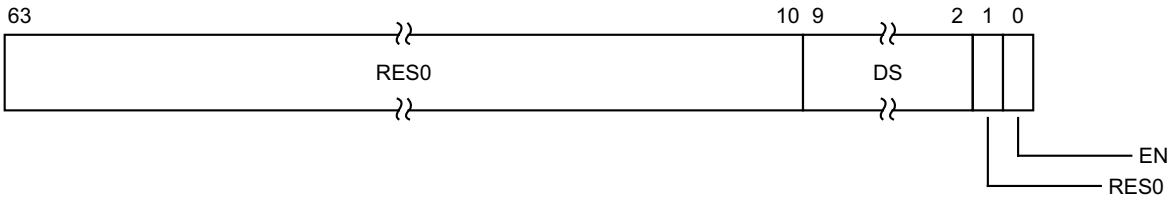
If no LORegion descriptors are supported by the PE, then this register is RES0.

Attributes

LORC_EL1 is a 64-bit register.

Field descriptions

The LORC_EL1 bit assignments are:



Bits [63:10]

Reserved, RES0.

DS, bits [9:2]

Descriptor Select. Selects the current LORegion descriptor accessed by LORSA_EL1, LOREA_EL1, and LORN_EL1.

The number of LORegion descriptors in IMPLEMENTATION DEFINED. The maximum number of LORegion descriptors supported is 256. If the number is less than 256, then bits[63:M+2] are RES0, where M is Log_2(Number of LORegion descriptors supported by the implementation).

If this field points to an LORegion descriptor that is not supported by an implementation, then the registers LORN_EL1, LOREA_EL1, and LORSA_EL1 are RES0.

This field resets to an architecturally UNKNOWN value.

Bit [1]

Reserved, RES0.

EN, bit [0]

Enable. Indicates whether LORegions are enabled.

`0b0` Disabled. Memory accesses do not match any LORegions.

`0b1` Enabled. Memory accesses may match a LORegion.

This bit is permitted to be cached in a TLB.

This field resets to `0`.

Accessing the LORC_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, LORC_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0100	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if SCR_EL3.NS == '0' then
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TLOR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.LORC_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TLOR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return LORC_EL1;
elsif PSTATE.EL == EL2 then
    if SCR_EL3.NS == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TLOR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return LORC_EL1;
elsif PSTATE.EL == EL3 then
    if SCR_EL3.NS == '0' then
        UNDEFINED;
    else
        return LORC_EL1;
```

MSR LORC_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0100	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if SCR_EL3.NS == '0' then
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TLOR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.LORC_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TLOR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        LORC_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if SCR_EL3.NS == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TLOR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        LORC_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    if SCR_EL3.NS == '0' then
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
else
    LORC_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.88 LOREA_EL1, LORegion End Address (EL1)

The LOREA_EL1 characteristics are:

Purpose

Holds the physical address of the end of the LORegion described in the current LORegion descriptor selected by LORC_EL1.DS.

Configurations

This register is present only when ARMv8.1-LOR is implemented. Otherwise, direct accesses to LOREA_EL1 are UNDEFINED.

This register is RES0 if any of the following apply:

- No LORegion descriptors are supported by the PE.
- LORC_EL1.DS points to a LORegion that is not supported by the PE.

Attributes

LOREA_EL1 is a 64-bit register.

Field descriptions

The LOREA_EL1 bit assignments are:

Any of the fields in this register are permitted to be cached in a TLB.

Bits [63:52]

Reserved, RES0.

EA[51:48], bits [51:48]

When ARMv8.2-LPA is implemented:

Extension to EA[47:16]. See EA[47:16] for more details.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EA[47:16], bits [47:16]

Bits [47:16] of the end physical address of an LORegion described in the current LORegion descriptor selected by LORC_EL1.DS. Bits[15:0] of this address are defined to be `0xFFFF`. For implementations with fewer than 48 bits, the upper bits of this field are RES0.

When ARMv8.2-LPA is implemented, and 52-bit addresses and a 64KB translation granule are in use, EA[51:48] form the upper part of the address value. Otherwise, for implementations with fewer than 52 physical address bits, EA[51:48] are RES0.

This field resets to an architecturally UNKNOWN value.

Bits [15:0]

Reserved, RES0.

Accessing the LOREA_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, LOREA_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0100	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if SCR_EL3.NS == '0' then
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TLOR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.LOREA_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TLOR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return LOREA_EL1;
elsif PSTATE.EL == EL2 then
    if SCR_EL3.NS == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TLOR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return LOREA_EL1;
elsif PSTATE.EL == EL3 then
    if SCR_EL3.NS == '0' then
        UNDEFINED;
    else
        return LOREA_EL1;
```

MSR LOREA_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0100	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if SCR_EL3.NS == '0' then
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TLOR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.LOREA_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TLOR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        LOREA_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if SCR_EL3.NS == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TLOR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        LOREA_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    if SCR_EL3.NS == '0' then
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
else
    LOREA_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.89 LORID_EL1, LORegionID (EL1)

The LORID_EL1 characteristics are:

Purpose

Indicates the number of LORegions and LORegion descriptors supported by the PE.

Configurations

This register is present only when ARMv8.1-LOR is implemented. Otherwise, direct accesses to LORID_EL1 are UNDEFINED.

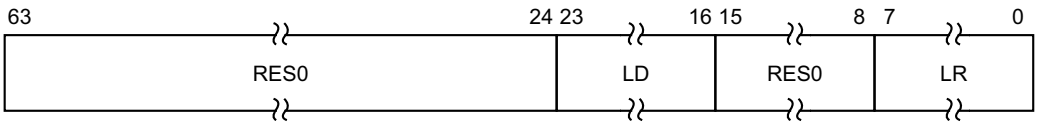
If no LORegion descriptors are implemented, then the registers LORC_EL1, LORN_EL1, LOREA_EL1, and LORSA_EL1 are RES0.

Attributes

LORID_EL1 is a 64-bit register.

Field descriptions

The LORID_EL1 bit assignments are:



Bits [63:24]

Reserved, RES0.

LD, bits [23:16]

Number of LORegion descriptors supported by the PE. This is an 8-bit binary number.

Bits [15:8]

Reserved, RES0.

LR, bits [7:0]

Number of LORegions supported by the PE. This is an 8-bit binary number.

Note

If LORID_EL1 indicates that no LORegions are implemented, then LoadLOAcquire and StoreLORelease will behave as LoadAcquire and StoreRelease.

Accessing the LORID_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, LORID_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0100	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TLOR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.LORID_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TLOR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return LORID_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TLOR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return LORID_EL1;
elsif PSTATE.EL == EL3 then
    return LORID_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.90 LORN_EL1, LORegion Number (EL1)

The LORN_EL1 characteristics are:

Purpose

Holds the number of the LORegion described in the current LORegion descriptor selected by LORC_EL1.DS.

Configurations

This register is present only when ARMv8.1-LOR is implemented. Otherwise, direct accesses to LORN_EL1 are UNDEFINED.

This register is RES0 if any of the following apply:

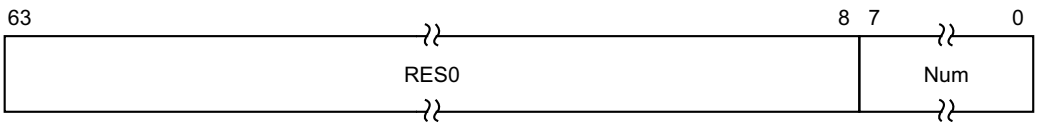
- No LORegion descriptors are supported by the PE.
- LORC_EL1.DS points to a LORegion that is not supported by the PE.

Attributes

LORN_EL1 is a 64-bit register.

Field descriptions

The LORN_EL1 bit assignments are:



Any of the fields in this register are permitted to be cached in a TLB.

Bits [63:8]

Reserved, RES0.

Num, bits [7:0]

Number of the LORegion described in the current LORegion descriptor selected by LORC_EL1.DS.

The maximum number of LORegions supported by the PE is 256. If the maximum number is less than 256, then bits[8:N] are RES0, where N is (Log_2(Number of LORegions supported by the PE)).

If this field points to a LORegion that is not supported by the PE, then the current LORegion descriptor does not match any LORegion.

This field resets to an architecturally UNKNOWN value.

Accessing the LORN_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, LORN_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0100	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if SCR_EL3.NS == '0' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TLOR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.LORN_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TLOR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return LORN_EL1;
elsif PSTATE.EL == EL2 then
    if SCR_EL3.NS == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TLOR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return LORN_EL1;
elsif PSTATE.EL == EL3 then
    if SCR_EL3.NS == '0' then
        UNDEFINED;
    else
        return LORN_EL1;
```

MSR LORN_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0100	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if SCR_EL3.NS == '0' then
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TLOR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.LORN_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TLOR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        LORN_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if SCR_EL3.NS == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TLOR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        LORN_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    if SCR_EL3.NS == '0' then
        UNDEFINED;
    else
        LORN_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.91 LORSA_EL1, LORegion Start Address (EL1)

The LORSA_EL1 characteristics are:

Purpose

Indicates whether the current LORegion descriptor selected by LORC_EL1.DS is enabled, and holds the physical address of the start of the LORegion.

Configurations

This register is present only when ARMv8.1-LOR is implemented. Otherwise, direct accesses to LORSA_EL1 are UNDEFINED.

This register is RES0 if any of the following apply:

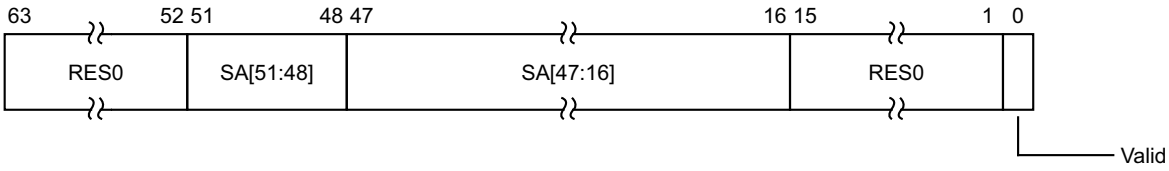
- No LORegion descriptors are supported by the PE.
- LORC_EL1.DS points to a LORegion that is not supported by the PE.

Attributes

LORSA_EL1 is a 64-bit register.

Field descriptions

The LORSA_EL1 bit assignments are:



Any of the fields in this register are permitted to be cached in a TLB.

Bits [63:52]

Reserved, RES0.

SA[51:48], bits [51:48]

When ARMv8.2-LPA is implemented:

Extension to SA[47:16]. See SA[47:16] for more details.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SA[47:16], bits [47:16]

Bits [47:16] of the start physical address of the LORegion described in the current LORegion descriptor selected by LORC_EL1.DS. Bits[15:0] of this address are defined to be `0x0000`. For implementations with fewer than 48 bits, the upper bits of this field are RES0.

When ARMv8.2-LPA is implemented, and 52-bit addresses and a 64KB translation granule are in use, SA[51:48] form the upper part of the address value. Otherwise, for implementations with fewer than 52 physical address bits, SA[51:48] are RES0.

This field resets to an architecturally UNKNOWN value.

Bits [15:1]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Valid, bit [0]

Indicates whether the current LORegion Descriptor is enabled.

`0b0` Disabled

`0b1` Enabled

This field resets to `0`.

Accessing the LORSA_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, LORSA_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0100	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if SCR_EL3.NS == '0' then
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TLOR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.LORSA_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TLOR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return LORSA_EL1;
elsif PSTATE.EL == EL2 then
    if SCR_EL3.NS == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TLOR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return LORSA_EL1;
elsif PSTATE.EL == EL3 then
    if SCR_EL3.NS == '0' then
        UNDEFINED;
    else
        return LORSA_EL1;
```

MSR LORSA_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0100	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if SCR_EL3.NS == '0' then
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TLOR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.LORSA_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TLOR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        LORSA_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if SCR_EL3.NS == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TLOR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        LORSA_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    if SCR_EL3.NS == '0' then
        UNDEFINED;
    else
        LORSA_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.92 MAIR_EL1, Memory Attribute Indirection Register (EL1)

The MAIR_EL1 characteristics are:

Purpose

Provides the memory attribute encodings corresponding to the possible AttrIndx values in a Long-descriptor format translation table entry for stage 1 translations at EL1.

Configurations

AArch64 System register MAIR_EL1[31:0] is architecturally mapped to AArch32 System register PRRR[31:0] when TTBCR.EAE == 0.

AArch64 System register MAIR_EL1[31:0] is architecturally mapped to AArch32 System register MAIR0[31:0] when TTBCR.EAE == 1.

AArch64 System register MAIR_EL1[63:32] is architecturally mapped to AArch32 System register NMRR[31:0] when TTBCR.EAE == 0.

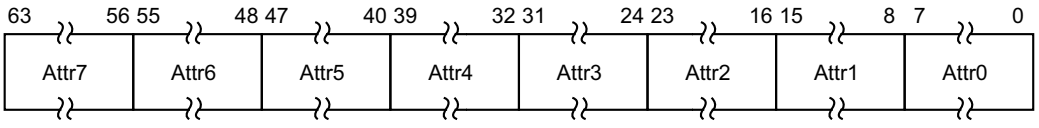
AArch64 System register MAIR_EL1[63:32] is architecturally mapped to AArch32 System register MAIR1[31:0] when TTBCR.EAE == 1.

Attributes

MAIR_EL1 is a 64-bit register.

Field descriptions

The MAIR_EL1 bit assignments are:



MAIR_EL1 is permitted to be cached in a TLB.

Attr<n>, bits [8n+7:8n], for n = 0 to 7

The memory attribute encoding for an AttrIndx[2:0] entry in a Long descriptor format translation table entry, where AttrIndx[2:0] gives the value of <n> in Attr<n>.

Attr is encoded as follows:

Attr	Meaning
`0b0000dd00`	Device memory. See encoding of 'dd' for the type of Device memory.
`0b0000ddxx`, (xx != 00)	UNPREDICTABLE
`0boooo iiii`, (oooo != 0000 and iiii != 0000)	Normal memory. See encoding of 'oooo' and 'iiii' for the type of Normal Memory.
`0b11110000`	Tagged Normal Memory. Inner+Outer Write-Back Non-transient memory, Inner+Outer Read-Allocate, Inner+Outer Write-Allocate.
`0bxxxx0000`, (xxxx != 0000 and xxxx != 1111)	UNPREDICTABLE

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

'dd' is encoded as follows:

dd	Meaning
0b00	Device-nGnRnE memory
0b01	Device-nGnRE memory
0b10	Device-nGRE memory
0b11	Device-GRE memory

'oooo' is encoded as follows:

'oooo'	Meaning
0b0000	See encoding of Attr
0b00RW, RW not 0b00	Normal memory, Outer Write-Through Transient
0b0100	Normal memory, Outer Non-cacheable
0b01RW, RW not 0b00	Normal memory, Outer Write-Back Transient
0b10RW	Normal memory, Outer Write-Through Non-transient
0b11RW	Normal memory, Outer Write-Back Non-transient

R = Outer Read-Allocate policy, W = Outer Write-Allocate policy.

'iiii' is encoded as follows:

'iiii'	Meaning
0b0000	See encoding of Attr
0b00RW, RW not 0b00	Normal memory, Inner Write-Through Transient
0b0100	Normal memory, Inner Non-cacheable
0b01RW, RW not 0b00	Normal memory, Inner Write-Back Transient
0b10RW	Normal memory, Inner Write-Through Non-transient
0b11RW	Normal memory, Inner Write-Back Non-transient

R = Inner Read-Allocate policy, W = Inner Write-Allocate policy.

The R and W bits in 'oooo' and 'iiii' fields have the following meanings:

R or W	Meaning
0b0	No Allocate
0b1	Allocate

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the MAIR_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic MAIR_EL1 or MAIR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, MAIR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.MAIR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x140];
    else
        return MAIR_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return MAIR_EL2;
    else
        return MAIR_EL1;
elsif PSTATE.EL == EL3 then
    return MAIR_EL1;
```

MSR MAIR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.MAIR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x140] = X[t];
    else
        MAIR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        MAIR_EL2 = X[t];
    else
        MAIR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    MAIR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, MAIR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b1010	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        return NVMem[0x140];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return MAIR_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        return MAIR_EL1;
    else
        UNDEFINED;
```

MSR MAIR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b1010	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        NVMem[0x140] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        MAIR_EL1 = X[t];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        MAIR_EL1 = X[t];
    else
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.93 MAIR_EL2, Memory Attribute Indirection Register (EL2)

The MAIR_EL2 characteristics are:

Purpose

Provides the memory attribute encodings corresponding to the possible AttrIndx values in a Long-descriptor format translation table entry for stage 1 translations at EL2.

Configurations

AArch64 System register MAIR_EL2[31:0] is architecturally mapped to AArch32 System register HMAIR0[31:0].

AArch64 System register MAIR_EL2[63:32] is architecturally mapped to AArch32 System register HMAIR1[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

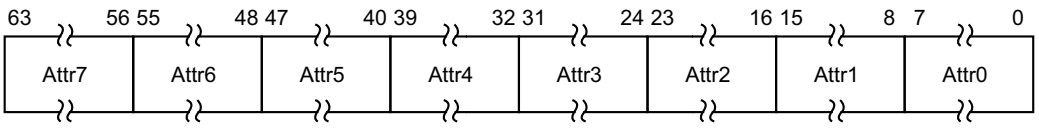
This register has no effect if EL2 is not enabled in the current Security state.

Attributes

MAIR_EL2 is a 64-bit register.

Field descriptions

The MAIR_EL2 bit assignments are:



MAIR_EL2 is permitted to be cached in a TLB.

Attr<n>, bits [8n+7:8n], for n = 0 to 7

The memory attribute encoding for an AttrIndx[2:0] entry in a Long descriptor format translation table entry, where AttrIndx[2:0] gives the value of <n> in Attr<n>.

Attr is encoded as follows:

Attr	Meaning
`0b0000dd00`	Device memory. See encoding of 'dd' for the type of Device memory.
`0b0000ddxx`, (`xx` != `00`)	UNPREDICTABLE
`0boooo iiii`, (`oooo` != `0000` and `iiii` != `0000`)	Normal memory. See encoding of '`oooo`' and '`iiii`' for the type of Normal Memory.
`0b11110000`	Tagged Normal Memory. Inner+Outer Write-Back Non-transient memory, Inner+Outer Read-Allocate, Inner+Outer Write-Allocate.
`0bxxxx0000`, (`xxxx` != `0000` and `xxxx` != `1111`)	UNPREDICTABLE

'dd' is encoded as follows:

dd	Meaning
`0b00`	Device-nGnRnE memory

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

dd	Meaning
0b01	Device-nGnRE memory
0b10	Device-nGRE memory
0b11	Device-GRE memory

'oooo' is encoded as follows:

'oooo'	Meaning
0b0000	See encoding of Attr
0b00RW, RW not 0b00	Normal memory, Outer Write-Through Transient
0b0100	Normal memory, Outer Non-cacheable
0b01RW, RW not 0b00	Normal memory, Outer Write-Back Transient
0b10RW	Normal memory, Outer Write-Through Non-transient
0b11RW	Normal memory, Outer Write-Back Non-transient

R = Outer Read-Allocate policy, W = Outer Write-Allocate policy.

'iiii' is encoded as follows:

'iiii'	Meaning
0b0000	See encoding of Attr
0b00RW, RW not 0b00	Normal memory, Inner Write-Through Transient
0b0100	Normal memory, Inner Non-cacheable
0b01RW, RW not 0b00	Normal memory, Inner Write-Back Transient
0b10RW	Normal memory, Inner Write-Through Non-transient
0b11RW	Normal memory, Inner Write-Back Non-transient

R = Inner Read-Allocate policy, W = Inner Write-Allocate policy.

The R and W bits in 'oooo' and 'iiii' fields have the following meanings:

R or W	Meaning
0b0	No Allocate
0b1	Allocate

This field resets to an architecturally UNKNOWN value.

Accessing the MAIR_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic MAIR_EL2 or MAIR_EL1 is not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, MAIR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return MAIR_EL2;
elsif PSTATE.EL == EL3 then
    return MAIR_EL2;
```

MSR MAIR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    MAIR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    MAIR_EL2 = X[t];
```

MRS <Xt>, MAIR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.MAIR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x140];
    else
        return MAIR_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return MAIR_EL2;
    else
        return MAIR_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    return MAIR_EL1;
```

MSR MAIR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.MAIR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x140] = X[t];
    else
        MAIR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        MAIR_EL2 = X[t];
    else
        MAIR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    MAIR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.94 MAIR_EL3, Memory Attribute Indirection Register (EL3)

The MAIR_EL3 characteristics are:

Purpose

Provides the memory attribute encodings corresponding to the possible AttrIndx values in a Long-descriptor format translation table entry for stage 1 translations at EL3.

Configurations

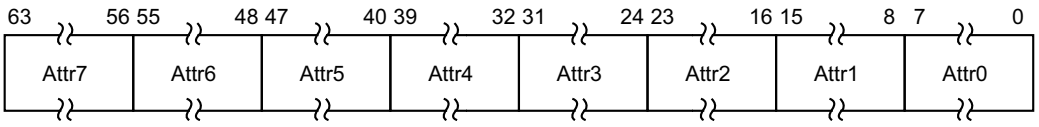
This register is present only when EL3 is implemented. Otherwise, direct accesses to MAIR_EL3 are UNDEFINED.

Attributes

MAIR_EL3 is a 64-bit register.

Field descriptions

The MAIR_EL3 bit assignments are:



MAIR_EL3 is permitted to be cached in a TLB.

Attr<n>, bits [8n+7:8n], for n = 0 to 7

The memory attribute encoding for an AttrIndx[2:0] entry in a Long descriptor format translation table entry, where AttrIndx[2:0] gives the value of <n> in Attr<n>.

Attr is encoded as follows:

Attr	Meaning
`0b0000dd00`	Device memory. See encoding of '`dd`' for the type of Device memory.
`0b0000ddxx`, (`xx` != `00`)	UNPREDICTABLE
`0boooo iiii`, (`oooo` != `0000` and `iiii` != `0000`)	Normal memory. See encoding of '`oooo`' and '`iiii`' for the type of Normal Memory.
`0b11110000`	Tagged Normal Memory. Inner+Outer Write-Back Non-transient memory, Inner+Outer Read-Allocate, Inner+Outer Write-Allocate.
`0bxxxx0000`, (`xxxx` != `0000` and `xxxx` != `1111`)	UNPREDICTABLE

'dd' is encoded as follows:

dd	Meaning
`0b00`	Device-nGnRnE memory
`0b01`	Device-nGnRE memory
`0b10`	Device-nGRE memory
`0b11`	Device-GRE memory

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

'oooo' is encoded as follows:

'oooo'	Meaning
0b0000	See encoding of Attr
0b00RW, RW not 0b00	Normal memory, Outer Write-Through Transient
0b0100	Normal memory, Outer Non-cacheable
0b01RW, RW not 0b00	Normal memory, Outer Write-Back Transient
0b10RW	Normal memory, Outer Write-Through Non-transient
0b11RW	Normal memory, Outer Write-Back Non-transient

R = Outer Read-Allocate policy, W = Outer Write-Allocate policy.

'iiii' is encoded as follows:

'iiii'	Meaning
0b0000	See encoding of Attr
0b00RW, RW not 0b00	Normal memory, Inner Write-Through Transient
0b0100	Normal memory, Inner Non-cacheable
0b01RW, RW not 0b00	Normal memory, Inner Write-Back Transient
0b10RW	Normal memory, Inner Write-Through Non-transient
0b11RW	Normal memory, Inner Write-Back Non-transient

R = Inner Read-Allocate policy, W = Inner Write-Allocate policy.

The R and W bits in 'oooo' and 'iiii' fields have the following meanings:

R or W	Meaning
0b0	No Allocate
0b1	Allocate

This field resets to an architecturally UNKNOWN value.

Accessing the MAIR_EL3

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, MAIR_EL3

op0	op1	CRn	CRm	op2
0b11	0b110	0b1010	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    return MAIR_EL3;
```

MSR MAIR_EL3, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b110	0b1010	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    MAIR_EL3 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.95 MIDR_EL1, Main ID Register

The MIDR_EL1 characteristics are:

Purpose

Provides identification information for the PE, including an implementer code for the device and a device ID number.

Configurations

AArch64 System register MIDR_EL1[31:0] is architecturally mapped to AArch32 System register MIDR[31:0].

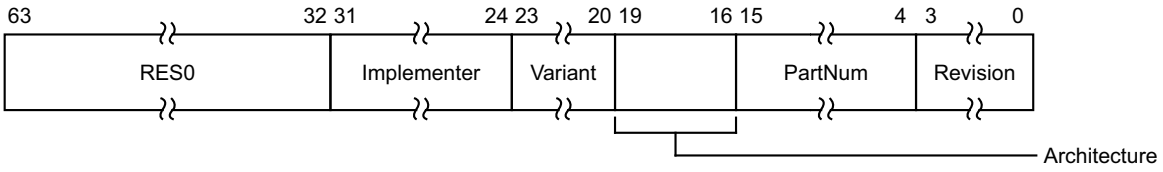
AArch64 System register MIDR_EL1[31:0] is architecturally mapped to External register MIDR_EL1[31:0].

Attributes

MIDR_EL1 is a 64-bit register.

Field descriptions

The MIDR_EL1 bit assignments are:



Bits [63:32]

Reserved, RES0.

Implementer, bits [31:24]

The Implementer code. This field must hold an implementer code that has been assigned by Arm. Assigned codes include the following:

Hex representation	Implementer
0x00	Reserved for software use
0xC0	Ampere Computing
0x41	Arm Limited
0x42	Broadcom Corporation
0x43	Cavium Inc.
0x44	Digital Equipment Corporation
0x46	Fujitsu Ltd.
0x49	Infineon Technologies AG
0x4D	Motorola or Freescale Semiconductor Inc.
0x4E	NVIDIA Corporation
0x50	Applied Micro Circuits Corporation

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Hex representation	Implementer
0x51	Qualcomm Inc.
0x56	Marvell International Ltd.
0x69	Intel Corporation

Arm can assign codes that are not published in this manual. All values not assigned by Arm are reserved and must not be used.

Variant, bits [23:20]

An IMPLEMENTATION DEFINED variant number. Typically, this field is used to distinguish between different product variants, or major revisions of a product.

Architecture, bits [19:16]

The permitted values of this field are:

0b0001 Armv4.

0b0010 Armv4T.

0b0011 Armv5 (obsolete).

0b0100 Armv5T.

0b0101 Armv5TE.

0b0110 Armv5TEJ.

0b0111 Armv6.

0b1111 Architectural features are individually identified in the ID_* registers, see *ID registers* on page K14-8060.

All other values are reserved.

PartNum, bits [15:4]

An IMPLEMENTATION DEFINED primary part number for the device.

On processors implemented by Arm, if the top four bits of the primary part number are 0x0 or 0x7, the variant and architecture are encoded differently.

Revision, bits [3:0]

An IMPLEMENTATION DEFINED revision number for the device.

Accessing the MIDR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, MIDR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0000	0b000

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.MIDR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) then
        return VPIDR_EL2;
    else
        return MIDR_EL1;
elsif PSTATE.EL == EL2 then
    return MIDR_EL1;
elsif PSTATE.EL == EL3 then
    return MIDR_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.96 MPIDR_EL1, Multiprocessor Affinity Register

The MPIDR_EL1 characteristics are:

Purpose

In a multiprocessor system, provides an additional PE identification mechanism for scheduling purposes.

Configurations

AArch64 System register MPIDR_EL1[31:0] is architecturally mapped to AArch32 System register MPIDR[31:0].

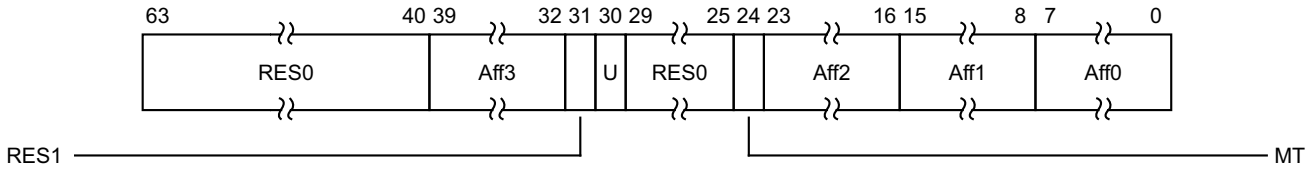
In a uniprocessor system Arm recommends that each Aff<n> field of this register returns a value of 0.

Attributes

MPIDR_EL1 is a 64-bit register.

Field descriptions

The MPIDR_EL1 bit assignments are:



Bits [63:40]

Reserved, RES0.

Aff3, bits [39:32]

Affinity level 3. See the description of Aff0 for more information.

Aff3 is not supported in AArch32 state.

Bit [31]

Reserved, RES1.

U, bit [30]

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system. The possible values of this bit are:

`0b0` Processor is part of a multiprocessor system.

`0b1` Processor is part of a uniprocessor system.

Bits [29:25]

Reserved, RES0.

MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using a multithreading type approach. See the description of Aff0 for more information about affinity levels. The possible values of this bit are:

`0b0` Performance of PEs at the lowest affinity level, or PEs with MPIDR_EL1.MT set to 1, different affinity level 0 values, and the same values for affinity level 1 and higher, is largely independent.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` Performance of PEs at the lowest affinity level, or PEs with MPIDR_EL1.MT set to 1, different affinity level 0 values, and the same values for affinity level 1 and higher, is very interdependent.

Aff2, bits [23:16]

Affinity level 2. See the description of Aff0 for more information.

Aff1, bits [15:8]

Affinity level 1. See the description of Aff0 for more information.

Aff0, bits [7:0]

Affinity level 0. This is the affinity level that is most significant for determining PE behavior. Higher affinity levels are increasingly less significant in determining PE behavior. The assigned value of the MPIDR.{Aff2, Aff1, Aff0} or MPIDR_EL1.{Aff3, Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

Accessing the MPIDR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, MPIDR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0000	0b101

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.MPIDR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) then
        return VMPIDR_EL2;
    else
        return MPIDR_EL1;
elsif PSTATE.EL == EL2 then
    return MPIDR_EL1;
elsif PSTATE.EL == EL3 then
    return MPIDR_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.97 MVFR0_EL1, AArch32 Media and VFP Feature Register 0

The MVFR0_EL1 characteristics are:

Purpose

Describes the features provided by the AArch32 Advanced SIMD and Floating-point implementation.

Must be interpreted with MVFR1_EL1 and MVFR2_EL1.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

AArch64 System register MVFR0_EL1[31:0] is architecturally mapped to AArch32 System register MVFR0[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to MVFR0_EL1 are UNKNOWN.

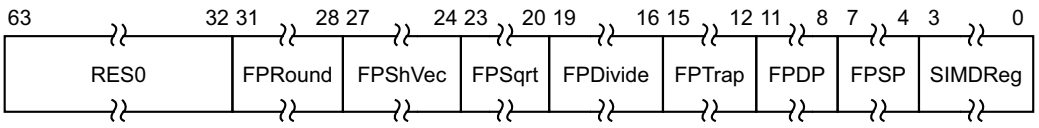
In an implementation where at least one Exception level supports execution in AArch32 state, but there is no support for Advanced SIMD and floating-point operation, this register is RAZ.

Attributes

MVFR0_EL1 is a 64-bit register.

Field descriptions

The MVFR0_EL1 bit assignments are:



Bits [63:32]

Reserved, RES0.

FPRound, bits [31:28]

Floating-Point Rounding modes. Indicates whether the floating-point implementation provides support for rounding modes. Defined values are:

`0b0000` Not implemented, or only Round to Nearest mode supported, except that Round towards Zero mode is supported for VCVT instructions that always use that rounding mode regardless of the FPSCR setting.

`0b0001` All rounding modes supported.

All other values are reserved.

In Armv8-A the permitted values are `0b0000` and `0b0001`.

FPShVec, bits [27:24]

Short Vectors. Indicates whether the floating-point implementation provides support for the use of short vectors. Defined values are:

`0b0000` Short vectors not supported.

`0b0001` Short vector operation supported.

All other values are reserved.

In Armv8-A the only permitted value is `0b0000`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

FPSqrt, bits [23:20]

Square Root. Indicates whether the floating-point implementation provides support for the ARMv6 VFP square root operations. Defined values are:

`0b0000` Not supported in hardware.

`0b0001` Supported.

All other values are reserved.

In Armv8-A the permitted values are `0b0000` and `0b0001`.

The VSQRT.F32 instruction also requires the single-precision floating-point attribute, bits [7:4], and the VSQRT.F64 instruction also requires the double-precision floating-point attribute, bits [11:8].

FPDivide, bits [19:16]

Indicates whether the floating-point implementation provides support for VFP divide operations. Defined values are:

`0b0000` Not supported in hardware.

`0b0001` Supported.

All other values are reserved.

In Armv8-A the permitted values are `0b0000` and `0b0001`.

The VDIV.F32 instruction also requires the single-precision floating-point attribute, bits [7:4], and the VDIV.F64 instruction also requires the double-precision floating-point attribute, bits [11:8].

FPTrap, bits [15:12]

Floating Point Exception Trapping. Indicates whether the floating-point implementation provides support for exception trapping. Defined values are:

`0b0000` Not supported.

`0b0001` Supported.

All other values are reserved.

A value of `0b0001` indicates that, when the corresponding trap is enabled, a floating-point exception generates an exception.

FPDP, bits [11:8]

Double Precision. Indicates whether the floating-point implementation provides support for double-precision operations. Defined values are:

`0b0000` Not supported in hardware.

`0b0001` Supported, VFPv2.

`0b0010` Supported, VFPv3, VFPv4, or Armv8. VFPv3 and Armv8 add an instruction to load a double-precision floating-point constant, and conversions between double-precision and fixed-point values.

All other values are reserved.

In Armv8-A the permitted values are `0b0000` and `0b0010`.

A value of `0b0001` or `0b0010` indicates support for all VFP double-precision instructions in the supported version of VFP, except that, in addition to this field being nonzero:

- VSQRT.F64 is only available if the Square root field is `0b0001`.
- VDIV.F64 is only available if the Divide field is `0b0001`.
- Conversion between double-precision and single-precision is only available if the single-precision field is nonzero.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

FPSP, bits [7:4]

Single Precision. Indicates whether the floating-point implementation provides support for single-precision operations. Defined values are:

`0b0000` Not supported in hardware.

`0b0001` Supported, VFPv2.

`0b0010` Supported, VFPv3 or VFPv4. VFPv3 adds an instruction to load a single-precision floating-point constant, and conversions between single-precision and fixed-point values.

All other values are reserved.

In Armv8-A the permitted values are `0b0000` and `0b0010`.

A value of `0b0001` or `0b0010` indicates support for all VFP single-precision instructions in the supported version of VFP, except that, in addition to this field being nonzero:

- VSQRT.F32 is only available if the Square root field is `0b0001`.
- VDIV.F32 is only available if the Divide field is `0b0001`.
- Conversion between double-precision and single-precision is only available if the double-precision field is nonzero.

SIMDReg, bits [3:0]

Advanced SIMD registers. Indicates whether the Advanced SIMD and floating-point implementation provides support for the Advanced SIMD and floating-point register bank. Defined values are:

`0b0000` The implementation has no Advanced SIMD and floating-point support.

`0b0001` The implementation includes floating-point support with 16 x 64-bit registers.

`0b0010` The implementation includes Advanced SIMD and floating-point support with 32 x 64-bit registers.

All other values are reserved.

In Armv8-A the permitted values are `0b0000` and `0b0010`.

Accessing the MVFR0_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, MVFR0_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0011	0b000

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return MVFR0_EL1;
elsif PSTATE.EL == EL2 then
    return MVFR0_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    return MVFR0_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.98 MVFR1_EL1, AArch32 Media and VFP Feature Register 1

The MVFR1_EL1 characteristics are:

Purpose

Describes the features provided by the AArch32 Advanced SIMD and Floating-point implementation.

Must be interpreted with MVFR0_EL1 and MVFR2_EL1.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

AArch64 System register MVFR1_EL1[31:0] is architecturally mapped to AArch32 System register MVFR1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to MVFR1_EL1 are UNKNOWN.

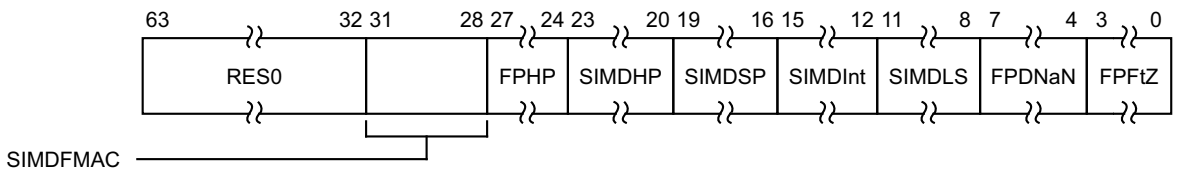
In an implementation where at least one Exception level supports execution in AArch32 state, but there is no support for Advanced SIMD and floating-point operation, this register is RAZ.

Attributes

MVFR1_EL1 is a 64-bit register.

Field descriptions

The MVFR1_EL1 bit assignments are:



Bits [63:32]

Reserved, RES0.

SIMDFMAC, bits [31:28]

Advanced SIMD Fused Multiply-Accumulate. Indicates whether the Advanced SIMD implementation provides fused multiply accumulate instructions. Defined values are:

`0b0000` Not implemented.

`0b0001` Implemented.

All other values are reserved.

In Armv8-A, the permitted values are `0b0000` and `0b0001`.

The Advanced SIMD and floating-point implementations must provide the same level of support for these instructions.

FPHP, bits [27:24]

Floating Point Half Precision. Indicates the level of half-precision floating-point support. Defined values are:

`0b0000` Not supported.

`0b0001` Floating-point half-precision conversion instructions are supported for conversion between single-precision and half-precision.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b0010` As for `0b0001`, and adds instructions for conversion between double-precision and half-precision.

`0b0011` As for `0b0010`, and adds support for half-precision floating-point arithmetic.

All other values are reserved.

In Armv8-A the permitted values are:

- `0b0000` in an implementation without floating-point support.
- `0b0010` in an implementation with floating-point support that does not include the ARMv8.2-FP16 extension.
- `0b0011` in an implementation with floating-point support that includes the ARMv8.2-FP16 extension.

The level of support indicated by this field must be equivalent to the level of support indicated by the SIMDHP field, meaning the permitted values are:

Half Precision instructions supported	FPHP	SIMDHP
No support	`0b0000`	`0b0000`
Conversions only	`0b0010`	`0b0001`
Conversions and arithmetic	`0b0011`	`0b0010`

SIMDHP, bits [23:20]

Advanced SIMD Half Precision. Indicates the level of half-precision floating-point support. Defined values are:

`0b0000` Not supported.

`0b0001` SIMD half-precision conversion instructions are supported for conversion between single-precision and half-precision.

`0b0010` As for `0b0001`, and adds support for half-precision floating-point arithmetic.

All other values are reserved.

In Armv8-A the permitted values are:

- `0b0000` in an implementation without SIMD floating-point support.
- `0b0010` in an implementation with SIMD floating-point support that does not include the ARMv8.2-FP16 extension.
- `0b0011` in an implementation with SIMD floating-point support that includes the ARMv8.2-FP16 extension.

The level of support indicated by this field must be equivalent to the level of support indicated by the FPHP field, meaning the permitted values are:

Half Precision instructions supported	FPHP	SIMDHP
No support	`0b0000`	`0b0000`
Conversions only	`0b0010`	`0b0001`
Conversions and arithmetic	`0b0011`	`0b0010`

SIMDSP, bits [19:16]

Advanced SIMD Single Precision. Indicates whether the Advanced SIMD and floating-point implementation provides single-precision floating-point instructions. Defined values are:

`0b0000` Not implemented.

`0b0001` Implemented. This value is permitted only if the SIMDInt field is `0b0001`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

All other values are reserved.

In Armv8-A, the permitted values are `0b0000` and `0b0001`.

SIMDInt, bits [15:12]

Advanced SIMD Integer. Indicates whether the Advanced SIMD and floating-point implementation provides integer instructions. Defined values are:

`0b0000` Not implemented.

`0b0001` Implemented.

All other values are reserved.

In Armv8-A, the permitted values are `0b0000` and `0b0001`.

SIMDLS, bits [11:8]

Advanced SIMD Load/Store. Indicates whether the Advanced SIMD and floating-point implementation provides load/store instructions. Defined values are:

`0b0000` Not implemented.

`0b0001` Implemented.

All other values are reserved.

In Armv8-A, the permitted values are `0b0000` and `0b0001`.

FPDNaN, bits [7:4]

Default NaN mode. Indicates whether the floating-point implementation provides support only for the Default NaN mode. Defined values are:

`0b0000` Not implemented, or hardware supports only the Default NaN mode.

`0b0001` Hardware supports propagation of NaN values.

All other values are reserved.

In Armv8-A, the permitted values are `0b0000` and `0b0001`.

FPFtZ, bits [3:0]

Flush to Zero mode. Indicates whether the floating-point implementation provides support only for the Flush-to-Zero mode of operation. Defined values are:

`0b0000` Not implemented, or hardware supports only the Flush-to-Zero mode of operation.

`0b0001` Hardware supports full denormalized number arithmetic.

All other values are reserved.

In Armv8-A, the permitted values are `0b0000` and `0b0001`.

Accessing the MVFR1_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, MVFR1_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0011	0b001

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return MVFR1_EL1;
elsif PSTATE.EL == EL2 then
    return MVFR1_EL1;
elsif PSTATE.EL == EL3 then
    return MVFR1_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.99 MVFR2_EL1, AArch32 Media and VFP Feature Register 2

The MVFR2_EL1 characteristics are:

Purpose

Describes the features provided by the AArch32 Advanced SIMD and Floating-point implementation.

Must be interpreted with MVFR0_EL1 and MVFR1_EL1.

For general information about the interpretation of the ID registers, see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

AArch64 System register MVFR2_EL1[31:0] is architecturally mapped to AArch32 System register MVFR2[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to MVFR2_EL1 are UNKNOWN.

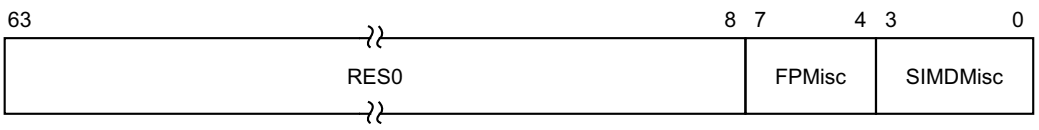
In an implementation where at least one Exception level supports execution in AArch32 state, but there is no support for Advanced SIMD and floating-point operation, this register is RAZ.

Attributes

MVFR2_EL1 is a 64-bit register.

Field descriptions

The MVFR2_EL1 bit assignments are:



Bits [63:8]

Reserved, RES0.

FPMisc, bits [7:4]

Indicates whether the floating-point implementation provides support for miscellaneous VFP features.

`0b0000` Not implemented, or no support for miscellaneous features.

`0b0001` Support for Floating-point selection.

`0b0010` As `0b0001`, and Floating-point Conversion to Integer with Directed Rounding modes.

`0b0011` As `0b0010`, and Floating-point Round to Integer Floating-point.

`0b0100` As `0b0011`, and Floating-point MaxNum and MinNum.

All other values are reserved.

In Armv8-A, the permitted values are `0b0000` and `0b0100`.

SIMDMisc, bits [3:0]

Indicates whether the Advanced SIMD implementation provides support for miscellaneous Advanced SIMD features.

`0b0000` Not implemented, or no support for miscellaneous features.

`0b0001` Floating-point Conversion to Integer with Directed Rounding modes.

`0b0010` As `0b0001`, and Floating-point Round to Integer Floating-point.

`0b0011` As `0b0010`, and Floating-point MaxNum and MinNum.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

All other values are reserved.

In Armv8-A, the permitted values are `0b0000` and `0b0011`.

Accessing the MVFR2_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, MVFR2_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0011	0b010

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return MVFR2_EL1;
elsif PSTATE.EL == EL2 then
    return MVFR2_EL1;
elsif PSTATE.EL == EL3 then
    return MVFR2_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.100 PAR_EL1, Physical Address Register

The PAR_EL1 characteristics are:

Purpose

Returns the output address (OA) from an Address translation instruction that executed successfully, or fault information if the instruction did not execute successfully.

Configurations

AArch64 System register PAR_EL1[63:0] is architecturally mapped to AArch32 System register PAR[63:0].

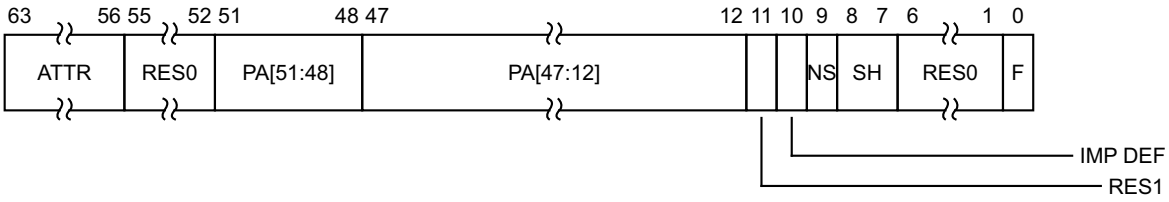
Attributes

PAR_EL1 is a 64-bit register.

Field descriptions

The PAR_EL1 bit assignments are:

When PAR_EL1.F == 0b0:



This section describes the register value returned by the successful execution of an Address translation instruction. Software might subsequently write a different value to the register, and that write does not affect the operation of the PE.

On a successful conversion, the PAR_EL1 can return a value that indicates the resulting attributes, rather than the values that appear in the translation table descriptors. More precisely:

- The PAR_EL1.{ATTR, SH} fields are permitted to report the resulting attributes, as determined by any permitted implementation choices and any applicable configuration bits, instead of reporting the values that appear in the translation table descriptors.
- See the PAR_EL1.NS bit description for constraints on the value it returns.

ATTR, bits [63:56]

Memory attributes for the returned output address. This field uses the same encoding as the Attr<n> fields in MAIR_EL1, MAIR_EL2, and MAIR_EL3.

The value returned in this field can be the resulting attribute, as determined by any permitted implementation choices and any applicable configuration bits, instead of the value that appears in the translation table descriptor.

This field resets to an architecturally UNKNOWN value.

Bits [55:52]

Reserved, RES0.

PA[51:48], bits [51:48]

When ARMv8.2-LPA is implemented:

Extension to PA[47:12]. See PA[47:12] for more details.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

Reserved, RES0.

PA[47:12], bits [47:12]

Output address. The output address (OA) corresponding to the supplied input address. This field returns address bits[47:12].

When ARMv8.2-LPA is implemented, and 52-bit addresses and a 64KB translation granule are in use, the PA[51:48] bits form the upper part of the address value. Otherwise the PA[51:48] bits are RES0.

For implementations with fewer than 48 physical address bits, the corresponding upper bits in this field are RES0.

This field resets to an architecturally UNKNOWN value.

Bit [11]

Reserved, RES1.

IMPLEMENTATION DEFINED, bit [10]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

NS, bit [9]

Non-secure. The NS attribute for a translation table entry from a Secure translation regime.

For a result from a Secure translation regime, when SCR_EL3.EEL2 is 1, this bit reflects the Security state of the intermediate physical address space of the translation for the instructions:

- In AArch64 state: AT S1E1R, AT S1E1W, AT S1E1RP, AT S1E1WP, AT S1E0R, and AT S1E0W.
- In AArch32 state: ATS1CPR, ATS1CPW, ATS1CPRP, ATS1CPWP, ATS1CUR, and ATS1CUW.

Otherwise, this bit reflects the Security state of the physical address space of the translation. This means it reflects the effect of the NSTable bits of earlier levels of the translation table walk if those NSTable bits have an effect on the translation.

For a result from a Non-secure translation regime, this bit is UNKNOWN.

This field resets to an architecturally UNKNOWN value.

SH, bits [8:7]

Shareability attribute, for the returned output address. Permitted values are:

`0b00` Non-shareable.

`0b10` Outer Shareable.

`0b11` Inner Shareable.

The value `0b01` is reserved.

Note

This field returns the value `0b10` for:

- Any type of Device memory.
- Normal memory with both Inner Non-cacheable and Outer Non-cacheable attributes.

The value returned in this field can be the resulting attribute, as determined by any permitted implementation choices and any applicable configuration bits, instead of the value that appears in the translation table descriptor.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bits [6:1]

Reserved, RES0.

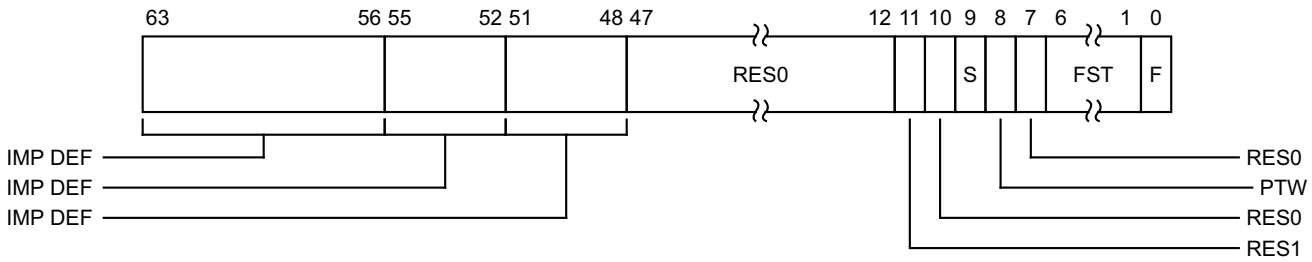
F, bit [0]

Indicates whether the instruction performed a successful address translation.

`0b0` Address translation completed successfully.

This field resets to an architecturally UNKNOWN value.

When PAR_EL1.F == 0b1:



This section describes the register value returned by a fault on the execution of an Address translation instruction. Software might subsequently write a different value to the register, and that write does not affect the operation of the PE.

IMPLEMENTATION DEFINED, bits [63:56]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

IMPLEMENTATION DEFINED, bits [55:52]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

IMPLEMENTATION DEFINED, bits [51:48]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

Bits [47:12]

Reserved, RES0.

Bit [11]

Reserved, RES1.

Bit [10]

Reserved, RES0.

S, bit [9]

Indicates the translation stage at which the translation aborted:

`0b0` Translation aborted because of a fault in the stage 1 translation.

`0b1` Translation aborted because of a fault in the stage 2 translation.

This field resets to an architecturally UNKNOWN value.

PTW, bit [8]

If this bit is set to 1, it indicates the translation aborted because of a stage 2 fault during a stage 1 translation table walk.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

Bit [7]

Reserved, RES0.

FST, bits [6:1]

Fault status code, as shown in the Data Abort ESR encoding.

`0b000000` Address size fault, level 0 of translation or translation table base register.

`0b000001` Address size fault, level 1.

`0b000010` Address size fault, level 2.

`0b000011` Address size fault, level 3.

`0b000100` Translation fault, level 0.

`0b000101` Translation fault, level 1.

`0b000110` Translation fault, level 2.

`0b000111` Translation fault, level 3.

`0b001001` Access flag fault, level 1.

`0b001010` Access flag fault, level 2.

`0b001011` Access flag fault, level 3.

`0b001101` Permission fault, level 1.

`0b001110` Permission fault, level 2.

`0b001111` Permission fault, level 3.

`0b010000` Synchronous External abort, not on translation table walk or hardware update of translation table.

`0b010001` **When ARMv8.5-MemTag is implemented** Synchronous Tag Check Fault.

`0b010100` Synchronous External abort, on translation table walk or hardware update of translation table, level 0.

`0b010101` Synchronous External abort, on translation table walk or hardware update of translation table, level 1.

`0b010110` Synchronous External abort, on translation table walk or hardware update of translation table, level 2.

`0b010111` Synchronous External abort, on translation table walk or hardware update of translation table, level 3.

`0b011000` **When RAS is not implemented** Synchronous parity or ECC error on memory access, not on translation table walk.

`0b011100` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 0.

`0b011101` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 1.

`0b011110` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 2.

`0b011111` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 3.

`0b100001` Alignment fault.

`0b110000` TLB conflict abort.

`0b110001` **When ARMv8.1-TTHM is implemented** Unsupported atomic hardware update fault.

`0b111101` **When AArch32 is supported at any Exception level and EL1 is capable of using AArch32** Section Domain fault, from an AArch32 stage 1 EL1&0 translation regime using Short-descriptor translation table format.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b111110` **When AArch32 is supported at any Exception level and EL1 is capable of using AArch32** Page Domain fault, from an AArch32 stage 1 EL1&0 translation regime using Short-descriptor translation table format.

This field resets to an architecturally UNKNOWN value.

F, bit [0]

Indicates whether the instruction performed a successful address translation.

`0b1` Address translation aborted.

This field resets to an architecturally UNKNOWN value.

Accessing the PAR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PAR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0111	0b0100	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.PAR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return PAR_EL1;
elsif PSTATE.EL == EL2 then
    return PAR_EL1;
elsif PSTATE.EL == EL3 then
    return PAR_EL1;
```

MSR PAR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0111	0b0100	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.PAR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        PAR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    PAR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    PAR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.101 REVIDR_EL1, Revision ID Register

The REVIDR_EL1 characteristics are:

Purpose

Provides implementation-specific minor revision information.

Configurations

AArch64 System register REVIDR_EL1[31:0] is architecturally mapped to AArch32 System register REVIDR[31:0].

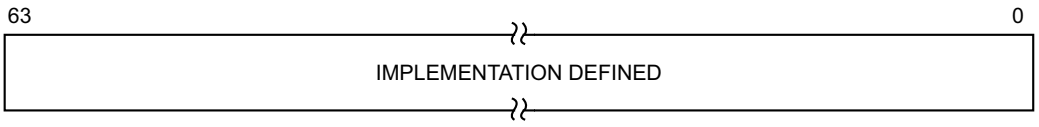
If REVIDR_EL1 has the same value as MIDR_EL1, then its contents have no significance.

Attributes

REVIDR_EL1 is a 64-bit register.

Field descriptions

The REVIDR_EL1 bit assignments are:



IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

Accessing the REVIDR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, REVIDR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0000	0b110

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.REVIDR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return REVIDR_EL1;
elsif PSTATE.EL == EL2 then
    return REVIDR_EL1;
elsif PSTATE.EL == EL3 then
    return REVIDR_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.102 RGSR_EL1, Random Allocation Tag Seed Register.

The RGSR_EL1 characteristics are:

Purpose

Random Allocation Tag Seed Register.

Configurations

This register is present only when ARMv8.5-MemTag is implemented and ID_AA64PFR1_EL1.MTE != 0b0001. Otherwise, direct accesses to RGSR_EL1 are UNDEFINED.

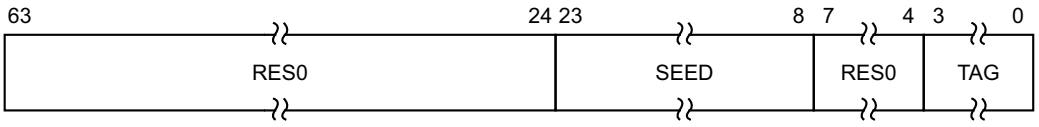
When GCR_EL1.RRND==`0b1`, updates to RGSR_EL1 are implementation-specific.

Attributes

RGSR_EL1 is a 64-bit register.

Field descriptions

The RGSR_EL1 bit assignments are:



Bits [63:24]

Reserved, RES0.

SEED, bits [23:8]

Seed register used for generating values returned by RandomAllocationTag().

This field resets to an architecturally UNKNOWN value.

Bits [7:4]

Reserved, RES0.

TAG, bits [3:0]

Tag generated by the most recent IRG instruction.

This field resets to an architecturally UNKNOWN value.

Accessing the RGSR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, RGSR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.ATA == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.ATA == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        return RGSR_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.ATA == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return RGSR_EL1;
elsif PSTATE.EL == EL3 then
    return RGSR_EL1;
```

MSR RGSR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.ATA == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.ATA == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        RGSR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.ATA == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        RGSR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    RGSR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.103 RMR_EL1, Reset Management Register (EL1)

The RMR_EL1 characteristics are:

Purpose

When this register is implemented:

- A write to the register at EL1 can request a Warm reset.
- If EL1 can use all Execution states, this register specifies the Execution state that the PE boots into on a Warm reset.

Configurations

AArch64 System register RMR_EL1[31:0] is architecturally mapped to AArch32 System register RMR[31:0] when the highest implemented Exception level is EL1.

This register is present only when the highest implemented Exception level is EL1. Otherwise, direct accesses to RMR_EL1 are UNDEFINED.

When EL1 is the highest implemented Exception level:

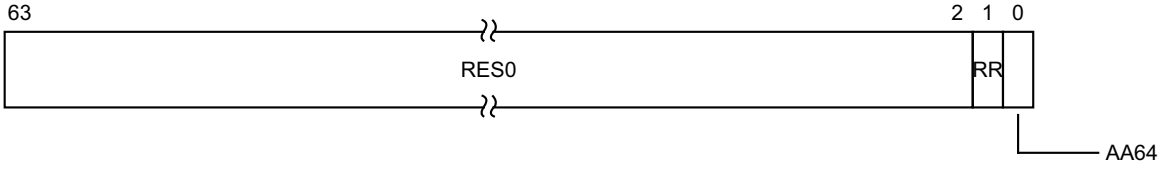
- If EL1 can use all Execution states then this register must be implemented.
- If EL1 cannot use AArch32 then it is IMPLEMENTATION DEFINED whether the register is implemented.

Attributes

RMR_EL1 is a 64-bit register.

Field descriptions

The RMR_EL1 bit assignments are:



Bits [63:2]

Reserved, RES0.

RR, bit [1]

Reset Request. Setting this bit to 1 requests a Warm reset.

This field resets to `0`.

AA64, bit [0]

When AArch32 is supported at any Exception level and EL1 is capable of using AArch32:

When EL1 can use AArch32, determines which Execution state the PE boots into after a Warm reset:

`0b0` AArch32.

`0b1` AArch64.

On coming out of the Warm reset, execution starts at the IMPLEMENTATION DEFINED reset vector address of the specified Execution state.

If EL1 can only use AArch64 state, this bit is RAO/WI.

When implemented as a RW field, this field resets to 1 on a Cold reset.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

Reserved, RAO/WI.

Accessing the RMR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, RMR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0000	0b010

```
if PSTATE.EL == EL1 && IsHighestEL(EL1) then
    return RMR_EL1;
else
    UNDEFINED;
```

MSR RMR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0000	0b010

```
if PSTATE.EL == EL1 && IsHighestEL(EL1) then
    RMR_EL1 = X[t];
else
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.104 RMR_EL2, Reset Management Register (EL2)

The RMR_EL2 characteristics are:

Purpose

When this register is implemented:

- A write to the register at EL2 can request a Warm reset.
- If EL2 can use all Execution states, this register specifies the Execution state that the PE boots into on a Warm reset.

Configurations

AArch64 System register RMR_EL2[31:0] is architecturally mapped to AArch32 System register HRMR[31:0] when the highest implemented Exception level is EL2.

This register is present only when the highest implemented Exception level is EL2. Otherwise, direct accesses to RMR_EL2 are UNDEFINED.

When EL2 is the highest implemented Exception level:

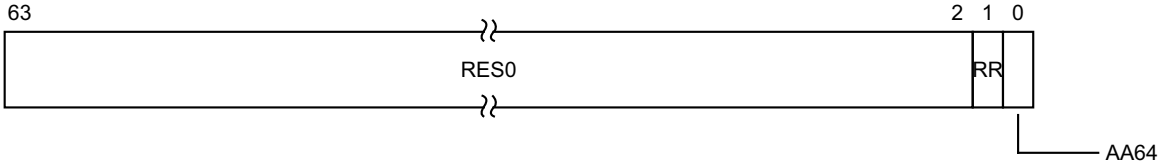
- If EL2 can use all Execution states then this register must be implemented.
- If EL2 cannot use AArch32 then it is IMPLEMENTATION DEFINED whether the register is implemented.

Attributes

RMR_EL2 is a 64-bit register.

Field descriptions

The RMR_EL2 bit assignments are:



Bits [63:2]

Reserved, RES0.

RR, bit [1]

Reset Request. Setting this bit to 1 requests a Warm reset.

This field resets to `0`.

AA64, bit [0]

When AArch32 is supported at any Exception level and EL2 is capable of using AArch32:

When EL2 can use AArch32, determines which Execution state the PE boots into after a Warm reset:

`0b0` AArch32.

`0b1` AArch64.

On coming out of the Warm reset, execution starts at the IMPLEMENTATION DEFINED reset vector address of the specified Execution state.

If EL2 can only use AArch64 state, this bit is RAO/WI.

When implemented as a RW field, this field resets to 1 on a Cold reset.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

Reserved, RAO/WI.

Accessing the RMR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, RMR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1100	0b0000	0b010

```
if PSTATE.EL == EL1 && EL2Enabled() && IsHighestEL(EL2) && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif PSTATE.EL == EL2 && IsHighestEL(EL2) then
    return RMR_EL2;
else
    UNDEFINED;
```

MSR RMR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1100	0b0000	0b010

```
if PSTATE.EL == EL1 && EL2Enabled() && IsHighestEL(EL2) && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif PSTATE.EL == EL2 && IsHighestEL(EL2) then
    RMR_EL2 = X[t];
else
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.105 RMR_EL3, Reset Management Register (EL3)

The RMR_EL3 characteristics are:

Purpose

If EL3 is the implemented and this register is implemented:

- A write to the register at EL3 can request a Warm reset.
- If EL3 can use all Execution states, this register specifies the Execution state that the PE boots into on a Warm reset.

Configurations

AArch64 System register RMR_EL3[31:0] is architecturally mapped to AArch32 System register RMR[31:0] when EL3 is implemented.

This register is present only when EL3 is implemented. Otherwise, direct accesses to RMR_EL3 are UNDEFINED.

When EL3 is implemented:

- If EL3 can use all Execution states then this register must be implemented.
- If EL3 cannot use AArch32 then it is IMPLEMENTATION DEFINED whether the register is implemented.

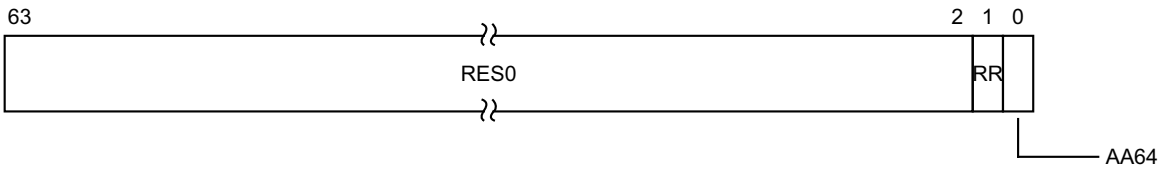
Otherwise, direct accesses to RMR_EL3 are UNDEFINED.

Attributes

RMR_EL3 is a 64-bit register.

Field descriptions

The RMR_EL3 bit assignments are:



Bits [63:2]

Reserved, RES0.

RR, bit [1]

Reset Request. Setting this bit to 1 requests a Warm reset.

This field resets to `0`.

AA64, bit [0]

When AArch32 is supported at any Exception level and EL3 is capable of using AArch32:

When EL3 can use AArch32, determines which Execution state the PE boots into after a Warm reset:

`0b0` AArch32.

`0b1` AArch64.

On coming out of the Warm reset, execution starts at the IMPLEMENTATION DEFINED reset vector address of the specified Execution state.

If EL3 can only use AArch64 state, this bit is RAO/WI.

When implemented as a RW field, this field resets to 1 on a Cold reset.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

Reserved, RAO/WI.

Accessing the RMR_EL3

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, RMR_EL3

op0	op1	CRn	CRm	op2
0b11	0b110	0b1100	0b0000	0b010

```
if PSTATE.EL == EL3 && IsHighestEL(EL3) then
    return RMR_EL3;
else
    UNDEFINED;
```

MSR RMR_EL3, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b110	0b1100	0b0000	0b010

```
if PSTATE.EL == EL3 && IsHighestEL(EL3) then
    RMR_EL3 = X[t];
else
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.106 RNDR, Random Number

The RNDR characteristics are:

Purpose

Random Number. Returns a 64-bit random number which is reseeded from the True Random Number source at an IMPLEMENTATION DEFINED rate.

If the hardware returns a genuine random number, PSTATE.NZCV is set to `0b0000`.

If the instruction cannot return a genuine random number in a reasonable period of time, PSTATE.NZCV is set to `0b0100` and the data value returned is 0.

RNDR is a read-only register.

Configurations

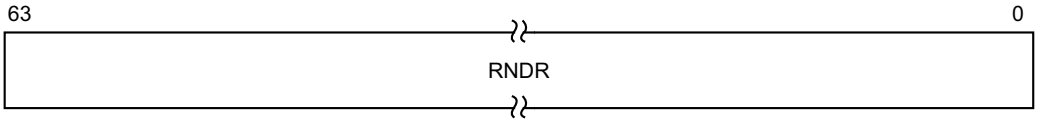
This register is present only when ARMv8.5-RNG is implemented. Otherwise, direct accesses to RNDR are UNDEFINED.

Attributes

RNDR is a 64-bit register.

Field descriptions

The RNDR bit assignments are:



RNDR, bits [63:0]

Random Number. Returns a 64-bit Random Number which is reseeded from the True Random Number source at an IMPLEMENTATION DEFINED rate.

This field resets to an architecturally UNKNOWN value.

Accessing the RNDR

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, RNDR

op0	op1	CRn	CRm	op2
0b11	0b011	0b0010	0b0100	0b000

```
if PSTATE.EL == EL0 then
    return RNDR;
elsif PSTATE.EL == EL1 then
    return RNDR;
elsif PSTATE.EL == EL2 then
    return RNDR;
elsif PSTATE.EL == EL3 then
    return RNDR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.107 RNDRRS, Reseeded Random Number

The RNDRRS characteristics are:

Purpose

Reseeded Random Number. Returns a 64-bit random number which is reseeded from the True Random Number source at an IMPLEMENTATION DEFINED rate.

If the hardware returns a genuine random number, PSTATE.NZCV is set to `0b0000`.

If the instruction cannot return a genuine random number in a reasonable period of time, PSTATE.NZCV is set to `0b0100` and the data value returned is 0.

RNDRRS is a read-only register.

Configurations

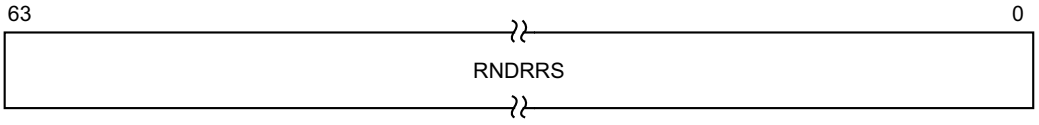
This register is present only when ARMv8.5-RNG is implemented. Otherwise, direct accesses to RNDRRS are UNDEFINED.

Attributes

RNDRRS is a 64-bit register.

Field descriptions

The RNDRRS bit assignments are:



RNDRRS, bits [63:0]

Reseeded Random Number. Returns a 64-bit Random Number which is reseeded from the True Random Number source immediately before this read.

This field resets to an architecturally UNKNOWN value.

Accessing the RNDRRS

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, RNDRRS

op0	op1	CRn	CRm	op2
0b11	0b011	0b0010	0b0100	0b001

```
if PSTATE.EL == EL0 then
    return RNDRRS;
elsif PSTATE.EL == EL1 then
    return RNDRRS;
elsif PSTATE.EL == EL2 then
    return RNDRRS;
elsif PSTATE.EL == EL3 then
    return RNDRRS;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.108 RVBAR_EL1, Reset Vector Base Address Register (if EL2 and EL3 not implemented)

The RVBAR_EL1 characteristics are:

Purpose

If EL1 is the highest Exception level implemented, contains the IMPLEMENTATION DEFINED address that execution starts from after reset when executing in AArch64 state.

Configurations

This register is present only when the highest implemented Exception level is EL1. Otherwise, direct accesses to RVBAR_EL1 are UNDEFINED.

Attributes

RVBAR_EL1 is a 64-bit register.

Field descriptions

The RVBAR_EL1 bit assignments are:

Bits [63:0]

Reset Address. The IMPLEMENTATION DEFINED address that execution starts from after reset when executing in 64-bit state. Bits[1:0] of this register are 00, as this address must be aligned, and the address must be within the physical address size supported by the PE.

Accessing the RVBAR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, RVBAR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0000	0b001

```
if PSTATE.EL == EL1 && IsHighestEL(EL1) then
    return RVBAR_EL1;
else
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.109 RVBAR_EL2, Reset Vector Base Address Register (if EL3 not implemented)

The RVBAR_EL2 characteristics are:

Purpose

If EL2 is the highest Exception level implemented, contains the IMPLEMENTATION DEFINED address that execution starts from after reset when executing in AArch64 state.

Configurations

This register is present only when the highest implemented Exception level is EL2. Otherwise, direct accesses to RVBAR_EL2 are UNDEFINED.

Attributes

RVBAR_EL2 is a 64-bit register.

Field descriptions

The RVBAR_EL2 bit assignments are:

Bits [63:0]

Reset Address. The IMPLEMENTATION DEFINED address that execution starts from after reset when executing in 64-bit state. Bits[1:0] of this register are 00, as this address must be aligned, and the address must be within the physical address size supported by the PE.

Accessing the RVBAR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, RVBAR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1100	0b0000	0b001

```
if PSTATE.EL == EL1 && EL2Enabled() && IsHighestEL(EL2) && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif PSTATE.EL == EL2 && IsHighestEL(EL2) then
    return RVBAR_EL2;
else
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.110 RVBAR_EL3, Reset Vector Base Address Register (if EL3 implemented)

The RVBAR_EL3 characteristics are:

Purpose

If EL3 is the highest Exception level implemented, contains the IMPLEMENTATION DEFINED address that execution starts from after reset when executing in AArch64 state.

Configurations

This register is present only when EL3 is implemented. Otherwise, direct accesses to RVBAR_EL3 are UNDEFINED.

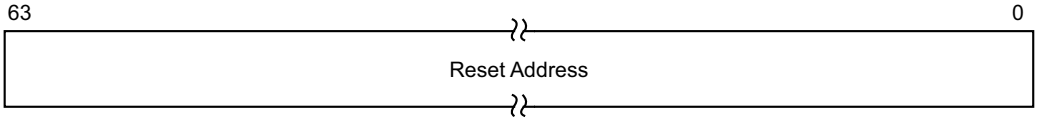
Only implemented if the highest Exception level implemented is EL3.

Attributes

RVBAR_EL3 is a 64-bit register.

Field descriptions

The RVBAR_EL3 bit assignments are:



Bits [63:0]

Reset Address. The IMPLEMENTATION DEFINED address that execution starts from after reset when executing in 64-bit state. Bits[1:0] of this register are 00, as this address must be aligned, and the address must be within the physical address size supported by the PE.

Accessing the RVBAR_EL3

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, RVBAR_EL3

op0	op1	CRn	CRm	op2
0b11	0b110	0b1100	0b0000	0b001

```
if PSTATE.EL == EL3 && IsHighestEL(EL3) then
    return RVBAR_EL3;
else
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.111 S3_<op1>_<Cn>_<Cm>_<op2>, IMPLEMENTATION DEFINED registers

The S3_<op1>_<Cn>_<Cm>_<op2> characteristics are:

Purpose

This area of the instruction set space is reserved for IMPLEMENTATION DEFINED registers.

Configurations

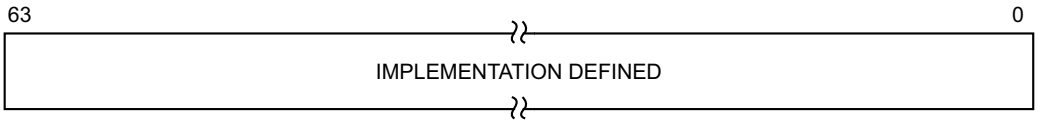
There are no configuration notes.

Attributes

S3_<op1>_<Cn>_<Cm>_<op2> is a 64-bit register.

Field descriptions

The S3_<op1>_<Cn>_<Cm>_<op2> bit assignments are:



IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

Accessing the S3_<op1>_<Cn>_<Cm>_<op2>

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, S3_<op1>_C<Cn>_C<Cm>_<op2>

op0	op1	CRn	CRm	op2
0b11	op1[2:0]	0b1x11	Cm[3:0]	op2[2:0]

```
if PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TIDCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        IMPLEMENTATION_DEFINED "";
else
    IMPLEMENTATION_DEFINED "";
```

MSR S3_<op1>_C<Cn>_C<Cm>_<op2>, <Xt>

op0	op1	CRn	CRm	op2
0b11	op1[2:0]	0b1x11	Cm[3:0]	op2[2:0]

```
if PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TIDCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        IMPLEMENTATION_DEFINED "";
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
else
    IMPLEMENTATION_DEFINED "";
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.112 SCR_EL3, Secure Configuration Register

The SCR_EL3 characteristics are:

Purpose

Defines the configuration of the current Security state. It specifies:

- The Security state of EL0, EL1, and EL2. The Security state is either Secure or Non-secure.
- The Execution state at lower Exception levels.
- Whether IRQ, FIQ, SError interrupts, and External abort exceptions are taken to EL3.
- Whether various operations are trapped to EL3.

Configurations

AArch64 System register SCR_EL3[31:0] can be mapped to AArch32 System register SCR[31:0], but this is not architecturally mandated.

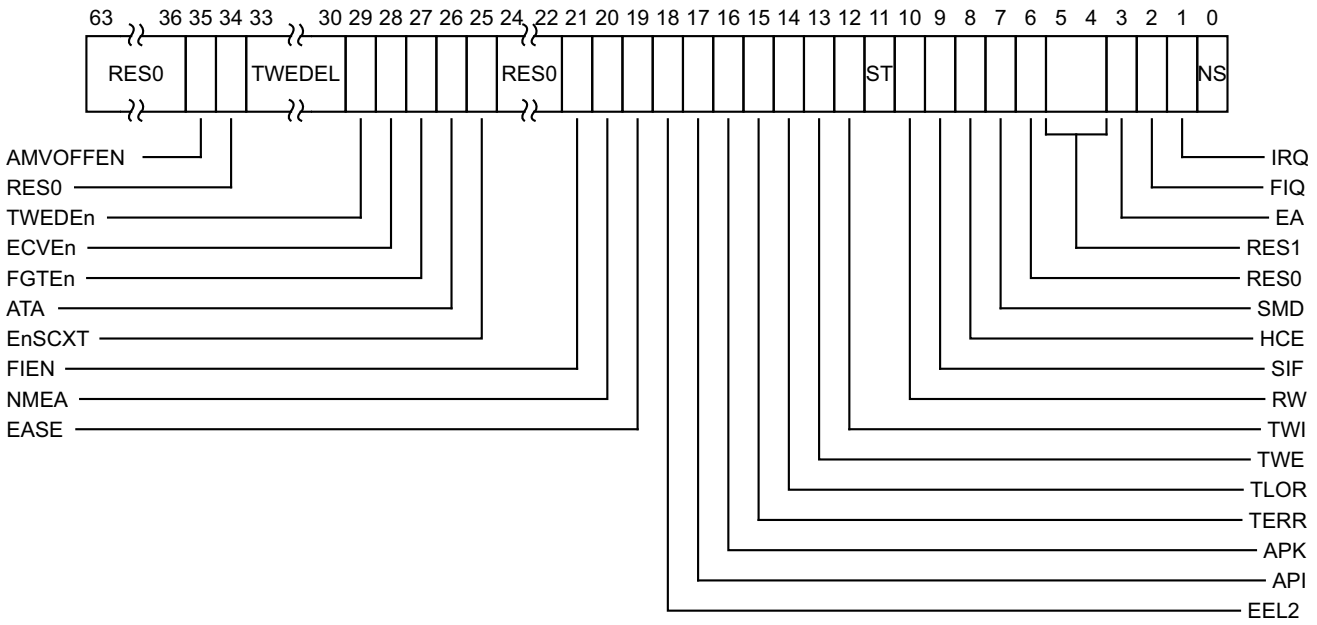
This register is present only when EL3 is implemented. Otherwise, direct accesses to SCR_EL3 are UNDEFINED.

Attributes

SCR_EL3 is a 64-bit register.

Field descriptions

The SCR_EL3 bit assignments are:



Bits [63:36]

Reserved, RES0.

AMVOFFEN, bit [35]

When ARMv8.6-AMU is implemented:

Activity Monitors Virtual Offsets Enable.

`0b0` Accesses to AMEVCNTVOFF0<n>_EL2 and AMEVCNTVOFF1<n>_EL2 at EL2 are trapped to EL3. Indirect reads of the virtual offset registers are zero.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` Accesses to AMEVCNTVOFF0<n>_EL2 and AMEVCNTVOFF1<n>_EL2 are not affected by this field.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [34]

Reserved, RES0.

TWEDEL, bits [33:30]

When ARMv8.6-TWED is implemented:

TWE Delay. A 4-bit unsigned number that, when SCR_EL3.TWEDEn is 1, encodes the minimum delay in taking a trap of WFE caused by SCR_EL3.TWE as 2^(TWEDEL + 8) cycles.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TWEDEn, bit [29]

When ARMv8.6-TWED is implemented:

TWE Delay Enable. Enables a configurable delayed trap of the WFE instruction caused by SCR_EL3.TWE.

`0b0` The delay for taking a WFE trap is IMPLEMENTATION DEFINED.

`0b1` The delay for taking a WFE trap is at least the number of cycles defined in SCR_EL3.TWEDEL.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ECVEn, bit [28]

When ARMv8.6-ECV is implemented:

ECV Enable. Enables access to the CNTPOFF_EL2 register.

`0b0` EL2 accesses to CNTPOFF_EL2 are trapped to EL3, and the value of CNTPOFF_EL2 is treated as 0 for all purposes other than direct reads or writes to the register from EL3.

`0b1` EL2 accesses to CNTPOFF_EL2 are not trapped to EL3 by this mechanism.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FGTEn, bit [27]

When ARMv8.6-FGT is implemented:

Enables access to the Fine-Grained Traps registers: HAFGRTR_EL2, HDFGRTR_EL2, HDFGWTR_EL2, HFGRTR_EL2, HFGITR_EL2 and HFGWTR_EL2.

`0b0` EL2 Accesses to HAFGRTR_EL2, HDFGRTR_EL2, HDFGWTR_EL2, HFGRTR_EL2, HFGITR_EL2 and HFGWTR_EL2 registers are trapped to EL3, and those registers behave as if all bits are set to 0.

`0b1` EL2 Accesses to HAFGRTR_EL2, HDFGRTR_EL2, HDFGWTR_EL2, HFGRTR_EL2, HFGITR_EL2 and HFGWTR_EL2 registers are not trapped to EL3 by this mechanism.

Traps caused by accesses to the fine-grained trap registers are reported using the EC code of `0x18` and its associated ISS.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

Reserved, RES0.

ATA, bit [26]

When ARMv8.5-MemTag is implemented:

Allocation Tag Access. Controls access at EL2, EL1 and EL0 to Allocation Tags.

When access is prevented:

- Instructions which Load or Store data are Unchecked.
- Instructions which Load or Store Allocation Tags treat the Allocation Tag as RAZ/WI.
- Instructions which insert Logical Address Tags into addresses treat the Allocation Tag used to generate the Logical Address Tag as 0.
- Cache maintenance instructions which invalidate Allocation Tags from caches behave as the equivalent Clean and Invalidate operation on Allocation Tags.
- MRS and MSR instructions at EL1 and EL2 using GCR_EL1, RGSR_EL1, TFSR_EL1, TFSR_EL2 or TFSRE0_EL1 that are not UNDEFINED or trapped to a lower Exception level are trapped to EL3.
- MRS and MSR instructions at EL2 using TFSR_EL1 that are not UNDEFINED are trapped to EL3.

`0b0` Access is prevented.

`0b1` Access is not prevented.

This field is permitted to be cached in a TLB.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnSCXT, bit [25]

When ARMv8.0-CSV2 is implemented:

Enable access to the SCXTNUM_EL2, SCXTNUM_EL1, and SCXTNUM_EL0 registers. The defined values are:

`0b0` EL2, EL1 and EL0 access to SCXTNUM_EL0, EL2 and EL1 access to SCXTNUM_EL1, EL2 access to SCXTNUM_EL2 registers are disabled by this mechanism, causing an exception to EL3, and the values of these registers to be treated as 0.

`0b1` This control does not cause accesses to SCXTNUM_EL0, SCXTNUM_EL1, SCXTNUM_EL2 to be trapped.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [24:22]

Reserved, RES0.

FIEN, bit [21]

When ARMv8.4-RAS is implemented:

Fault Injection enable. Trap accesses to the RAS ERX* registers ERXPFGCDN_EL1, ERXPFGCTL_EL1, and ERXPFGF_EL1 from EL1 and EL2 to EL3, reported using EC syndrome value `0x18`.

`0b0` Accesses to the specified registers from EL1 and EL2 generate a Trap exception to EL3.

`0b1` This control does not cause any instructions to be trapped.

If EL3 is not implemented, the Effective value of SCR_EL3.FIEN is `0b1`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If ERRIDR_EL1.NUM is zero, meaning no error records are implemented, or no error record accessible using System registers is owned by a node that implements the RAS Common Fault Injection Model Extension, then this bit might be RES0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NMEA, bit [20]

When ARMv8.4-DFE is implemented:

Non-maskable External Aborts. When SCR_EL3.EA == 1, controls whether PSTATE.A masks SError interrupts at EL3.

`0b0` If SCR_EL3.EA == 1, asserted SError interrupts are not taken at EL3 if PSTATE.A == 1.

`0b1` If SCR_EL3.EA == 1, asserted SError interrupts are taken at EL3 regardless of the value of PSTATE.A.

When SCR_EL3.EA == 0:

- Asserted SError interrupts are not taken at EL3 regardless of the value of PSTATE.A and this field.
- This field is ignored and its Effective value is 0.

This field resets to `0`.

Otherwise:

Reserved, RES0.

EASE, bit [19]

When ARMv8.4-DFE is implemented:

External aborts to SError interrupt vector.

`0b0` Synchronous External abort exceptions taken to EL3 are taken to the appropriate synchronous exception vector offset from VBAR_EL3.

`0b1` Synchronous External abort exceptions taken to EL3 are taken to the appropriate SError interrupt vector offset from VBAR_EL3.

This field resets to `0`.

Otherwise:

Reserved, RES0.

EEL2, bit [18]

When ARMv8.4-SecEL2 is implemented:

Secure EL2 Enable.

`0b0` All behaviors associated with Secure EL2 are disabled. All registers, including timer registers, defined by ARMv8.4-SecEL2 are UNDEFINED, and those timers are disabled.

`0b1` All behaviors associated with Secure EL2 are enabled.

When the value of this bit is 1, then:

- When SCR_EL3.NS == 0, the SCR_EL3.RW bit is treated as 1 for all purposes other than reading or writing the register.
- If Secure EL1 is using AArch32, then any of the following operations, executed in Secure EL1, is trapped to Secure EL2, using the EC value of ESR_EL2.EC== `0x3`:
  - A read or write of the SCR.
  - A read or write of the NSACR.
  - A read or write of the MVBAR.
  - A read or write of the SDCR.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— Execution of an ATS12NSO** instruction.

- If Secure EL1 is using AArch32, then any of the following operations, executed in Secure EL1, is trapped to Secure EL2 using the EC value of ESR_EL2.EC== `0x0`:
  — Execution of an SRS instruction that uses R13_mon.
  — Execution of an MRS (Banked register) or MSR (Banked register) instruction that would access SPSR_mon, R13_mon, or R14_mon.

Note

If the Effective value of SCR_EL3.EEL2 is 0, then these operations executed in Secure EL1 using AArch32 are trapped to EL3.

In a Secure only implementation that does not implement EL3 but implements EL2, behaves as if SCR_EL3.EEL2 == 1.

This bit is permitted to be cached in a TLB.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

API, bit [17]

When ARMv8.4-SecEL2 is implemented and ARMv8.3-PAuth is implemented:

Controls the use of the following instructions related to Pointer Authentication. Traps are reported using EC syndrome value `0x09`:

- PACGA, which is always enabled.
- AUTDA, AUTDB, AUTDZA, AUTDZB, AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZA, AUTIZB, PACDA, PACDB, PACDZA, PACDZB, PACIA, PACIA1716, PACIASP, PACIAZ, PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZA, PACIZB, RETAA, RETAB, BRAA, BRAB, BLRAA, BLRAB, BRAAZ, BRABZ, BLRAAZ, BLRABZ, ERETAA, ERETAB, LDRAA and LDRAB when:
  — In EL0, when HCR_EL2.TGE==0 or HCR_EL2.E2H==0, and the associated SCTLR_EL1.En<N><M> == 1.
  — In EL0, when HCR_EL2.TGE==1 and HCR_EL2.E2H==1, and the associated SCTLR_EL2.En<N><M> == 1.
  — In EL1, when the associated SCTLR_EL1.En<N><M> == 1.
  — In EL2, when the associated SCTLR_EL2.En<N><M> == 1.

`0b0` The use of any instruction related to pointer authentication in any Exception level except EL3 when the instructions are enabled are trapped to EL3 unless they are trapped to EL2 as a result of the HCR_EL2.API bit.

`0b1` This control does not cause any instructions to be trapped.

An instruction is trapped only if Pointer Authentication is enabled for that instruction, for more information, see *System register control of pointer authentication* on page D5-2539.

Note

If ARMv8.3-PAuth is implemented but EL3 is not implemented, the system behaves as if this bit is 1.

This field resets to an architecturally UNKNOWN value.

When ARMv8.4-SecEL2 is not implemented and ARMv8.3-PAuth is implemented:

Controls the use of instructions related to Pointer Authentication:

- PACGA.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- AUTDA, AUTDB, AUTDZA, AUTDZB, AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZA, AUTIZB, PACDA, PACDB, PACDZA, PACDZB, PACIA, PACIA1716, PACIASP, PACIAZ, PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZA, PACIZ, RETAA, RETAB, BRAA, BRAB, BLRAA, BLRAB, BRAAZ, BRABZ, BLRAAZ, BLRABZ, ERETAA, ERETAB, LDRAA and LDRAB when:
  - In Non-secure EL0, when HCR_EL2.TGE==0 or HCR_EL2.E2H==0, and the associated SCTLR_EL1.En<N><M>== 1.
  - In Non-secure EL0, when HCR_EL2.TGE==1 and HCR_EL2.E2H==1, and the associated SCTLR_EL2.En<N><M> == 1.
  - In Secure EL0, when the associated SCTLR_EL2.En<N><M> == 1.
  - In Secure or Non-secure EL1, when the associated SCTLR_EL1.En<N><M> == 1.
  - In EL2, when the associated SCTLR_EL2.En<N><M> == 1.

`0b0` The use of any instruction related to pointer authentication in any Exception level except EL3 when the instructions are enabled are trapped to EL3 unless they are trapped to EL2 as a result of the HCR_EL2.API bit.

`0b1` This control does not cause any instructions to be trapped.

Note

If ARMv8.3-PAuth is implemented but EL3 is not implemented, the system behaves as if this bit is 1.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

APK, bit [16]

When ARMv8.3-PAuth is implemented:

Trap registers holding "key" values for Pointer Authentication. Traps accesses to the following registers, using EC syndrome value `0x18`, from EL1 or EL2 to EL3 unless they are trapped to EL2 as a result of the HCR_EL2.APK bit or other traps:

- APIAKeyLo_EL1, APIAKeyHi_EL1, APIBKeyLo_EL1, APIBKeyHi_EL1.
- APDAKeyLo_EL1, APDAKeyHi_EL1, APDBKeyLo_EL1, APDBKeyHi_EL1.
- APGAKeyLo_EL1, and APGAKeyHi_EL1.

`0b0` Access to the registers holding "key" values for pointer authentication from EL1 or EL2 are trapped to EL3 unless they are trapped to EL2 as a result of the HCR_EL2.APK bit or other traps.

`0b1` This control does not cause any instructions to be trapped.

For more information, see *System register control of pointer authentication* on page D5-2539.

Note

If ARMv8.3-PAuth is implemented but EL3 is not implemented, the system behaves as if this bit is 1.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TERR, bit [15]

When RAS is implemented:

Trap Error record accesses. Accesses to the RAS ERR and RAS ERX registers from EL1 and EL2 to EL3 are trapped as follows:

- If EL1 is using AArch64, ERRIDR_EL1, ERRSELR_EL1, ERXADDR_EL1, ERXCTLR_EL1, ERXFR_EL1, ERXMISC0_EL1, ERXMISC1_EL1, and ERXSTATUS_EL1, are trapped and reported using EC syndrome value `0x18`.
- If ARMv8.4-RAS is implemented, ERXMISC2_EL1, and ERXMISC3_EL1, are trapped and reported using EC syndrome value `0x18`.
- If EL1 is using AArch32, accesses by MCR or MRC to the following registers are reported using EC syndrome value `0x03`, accesses by MCRR or MRRC are reported using EC syndrome value `0x04`:
  - ERRIDR, ERRSELR, ERXADDR, ERXADDR2, ERXCTLR, ERXCTLR2, ERXFR, ERXFR2, ERXMISC0, ERXMISC1, ERXMISC2, ERXMISC3, and ERXSTATUS.
- If ARMv8.4-RAS is implemented, accesses by MCR or MRC to the following registers are reported using EC syndrome value `0x03`, accesses by MCRR or MRRC are reported using EC syndrome value `0x04`:
  - ERXMISC4, ERXMISC5, ERXMISC6, and ERXMISC7.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Accesses to the specified registers from EL1 and EL2 generate a Trap exception to EL3.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TLOR, bit [14]

When ARMv8.1-LOR is implemented:

Trap LOR registers. Traps accesses to the LORSA_EL1, LOREA_EL1, LORN_EL1, LORC_EL1, and LORID_EL1 registers from EL1 and EL2 to EL3, unless the access has been trapped to EL2.

`0b0` This control does not cause any instructions to be trapped.

`0b1` EL1 and EL2 accesses to the LOR registers that are not UNDEFINED are trapped to EL3, unless it is trapped HCR_EL2.TLOR.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TWE, bit [13]

Traps EL2, EL1, and EL0 execution of WFE instructions to EL3, from both Security states and both Execution states, reported using EC syndrome value `0x01`.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Any attempt to execute a WFE instruction at any Exception level lower than EL3 is trapped to EL3, if the instruction would otherwise have caused the PE to enter a low-power state and it is not trapped by SCTLR.nTWE, HCR.TWE, SCTLR_EL1.nTWE, SCTLR_EL2.nTWE, or HCR_EL2.TWE.

In AArch32 state, the attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

For more information about when WFE instructions can cause the PE to enter a low-power state, see *Wait for Event mechanism and Send event* on page D1-2397.

This field resets to an architecturally UNKNOWN value.

TWI, bit [12]

Traps EL2, EL1, and EL0 execution of WFI instructions to EL3, from both Security states and both Execution states, reported using EC syndrome value `0x01`.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Any attempt to execute a WFI instruction at any Exception level lower than EL3 is trapped to EL3, if the instruction would otherwise have caused the PE to enter a low-power state and it is not trapped by SCTLR.nTWI, HCR.TWI, SCTLR_EL1.nTWI, SCTLR_EL2.nTWI, or HCR_EL2.TWI.

In AArch32 state, the attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

Note

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

For more information about when WFI instructions can cause the PE to enter a low-power state, see *Wait For Interrupt* on page D1-2400.

This field resets to an architecturally UNKNOWN value.

ST, bit [11]

Traps Secure EL1 accesses to the Counter-timer Physical Secure timer registers to EL3, from AArch64 state only, reported using EC syndrome value `0x18`.

`0b0` Secure EL1 using AArch64 accesses to the CNTPS_TVAL_EL1, CNTPS_CTL_EL1, and CNTPS_CVAL_EL1 are trapped to EL3 when Secure EL2 is disabled. If Secure EL2 is enabled, the behavior is as if the value of this field was `0b1`.

`0b1` This control does not cause any instructions to be trapped.

Note

Accesses to the Counter-timer Physical Secure timer registers are always enabled at EL3. These registers are not accessible at EL0.

This field resets to an architecturally UNKNOWN value.

RW, bit [10]

When AArch32 is supported at any Exception level:

Execution state control for lower Exception levels.

`0b0` Lower levels are all AArch32.

`0b1` The next lower level is AArch64.

If EL2 is present:

- EL2 is AArch64.
- EL2 controls EL1 and EL0 behaviors.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If EL2 is not present:

- EL1 is AArch64.
- EL0 is determined by the Execution state described in the current process state when executing at EL0.

If AArch32 state is not supported by the implementation at EL2 and AArch32 state is not supported by the implementation at EL1, then this bit is RAO/WI.

If AArch32 state is supported by the implementation at EL1, SCR_EL3.NS == 1 and AArch32 state is not supported by the implementation at EL2, the Effective value of this bit is 1.

If AArch32 state is supported by the implementation at EL1, ARMv8.4-SecEL2 is implemented and SCR_EL3.{EEL2, NS} == {1, 0}, the Effective value of this bit is 1.

This bit is permitted to be cached in a TLB.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAO/WI.

SIF, bit [9]

When ARMv8.4-SecEL2 is implemented:

Secure instruction fetch. When the PE is in Secure state, this bit disables instruction fetch from memory marked in the first stage of translation as being Non-secure. The possible values for this bit are:

`0b0` Secure state instruction fetches from memory marked in the first stage of translation as being Non-secure are permitted.

`0b1` Secure state instruction fetches from memory marked in the first stage of translation as being Non-secure are not permitted.

This bit is permitted to be cached in a TLB.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Secure instruction fetch. When the PE is in Secure state, this bit disables instruction fetch from Non-secure memory.

`0b0` Secure state instruction fetches from Non-secure memory are permitted.

`0b1` Secure state instruction fetches from Non-secure memory are not permitted.

This bit is permitted to be cached in a TLB.

This field resets to an architecturally UNKNOWN value.

HCE, bit [8]

Hypervisor Call instruction enable. Enables HVC instructions at EL3 and, if EL2 is enabled in the current Security state, at EL2 and EL1, in both Execution states, reported using EC syndrome value `0x00`.

`0b0` HVC instructions are UNDEFINED.

`0b1` HVC instructions are enabled at EL3, EL2, and EL1.

Note

HVC instructions are always UNDEFINED at EL0 and, if Secure EL2 is disabled, at Secure EL1. Any resulting exception is taken from the current Exception level to the current Exception level.

If EL2 is not implemented, this bit is RES0.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

SMD, bit [7]

Secure Monitor Call disable. Disables `SMC` instructions at EL1 and above, from both Security states and both Execution states, reported using EC syndrome value `0x00`.

`0b0` `SMC` instructions are enabled at EL3, EL2 and EL1.

`0b1` `SMC` instructions are UNDEFINED.

Note

`SMC` instructions are always UNDEFINED at EL0. Any resulting exception is taken from the current Exception level to the current Exception level.

If HCR_EL2.TSC or HCR.TSC traps attempted EL1 execution of `SMC` instructions to EL2, that trap has priority over this disable.

This field resets to an architecturally UNKNOWN value.

Bit [6]

Reserved, RES0.

Bits [5:4]

Reserved, RES1.

EA, bit [3]

External Abort and SError interrupt routing.

`0b0` When executing at Exception levels below EL3, External aborts and SError interrupts are not taken to EL3.

In addition, when executing at EL3:

- SError interrupts are not taken.
- External aborts are taken to EL3.

`0b1` When executing at any Exception level, External aborts and SError interrupts are taken to EL3.

For more information, see *Asynchronous exception routing* on page D1-2364.

This field resets to an architecturally UNKNOWN value.

FIQ, bit [2]

Physical FIQ Routing.

`0b0` When executing at Exception levels below EL3, physical FIQ interrupts are not taken to EL3.

When executing at EL3, physical FIQ interrupts are not taken.

`0b1` When executing at any Exception level, physical FIQ interrupts are taken to EL3.

For more information, see *Asynchronous exception routing* on page D1-2364.

This field resets to an architecturally UNKNOWN value.

IRQ, bit [1]

Physical IRQ Routing.

`0b0` When executing at Exception levels below EL3, physical IRQ interrupts are not taken to EL3.

When executing at EL3, physical IRQ interrupts are not taken.

`0b1` When executing at any Exception level, physical IRQ interrupts are taken to EL3.

For more information, see *Asynchronous exception routing* on page D1-2364.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

NS, bit [0]

Non-secure bit.

`0b0` Indicates that EL0 and EL1 are in Secure state.

`0b1` Indicates that Exception levels lower than EL3 are in Non-secure state, and so memory accesses from those Exception levels cannot access Secure memory.

When SCR_EL3.{EEL2, NS} == {1, 0}, then EL2 is using AArch64 and in Secure state.

This field resets to an architecturally UNKNOWN value.

Accessing the SCR_EL3

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, SCR_EL3

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    return SCR_EL3;
```

MSR SCR_EL3, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    SCR_EL3 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.113 SCTLR_EL1, System Control Register (EL1)

The SCTLR_EL1 characteristics are:

Purpose

Provides top level control of the system, including its memory system, at EL1 and EL0.

Configurations

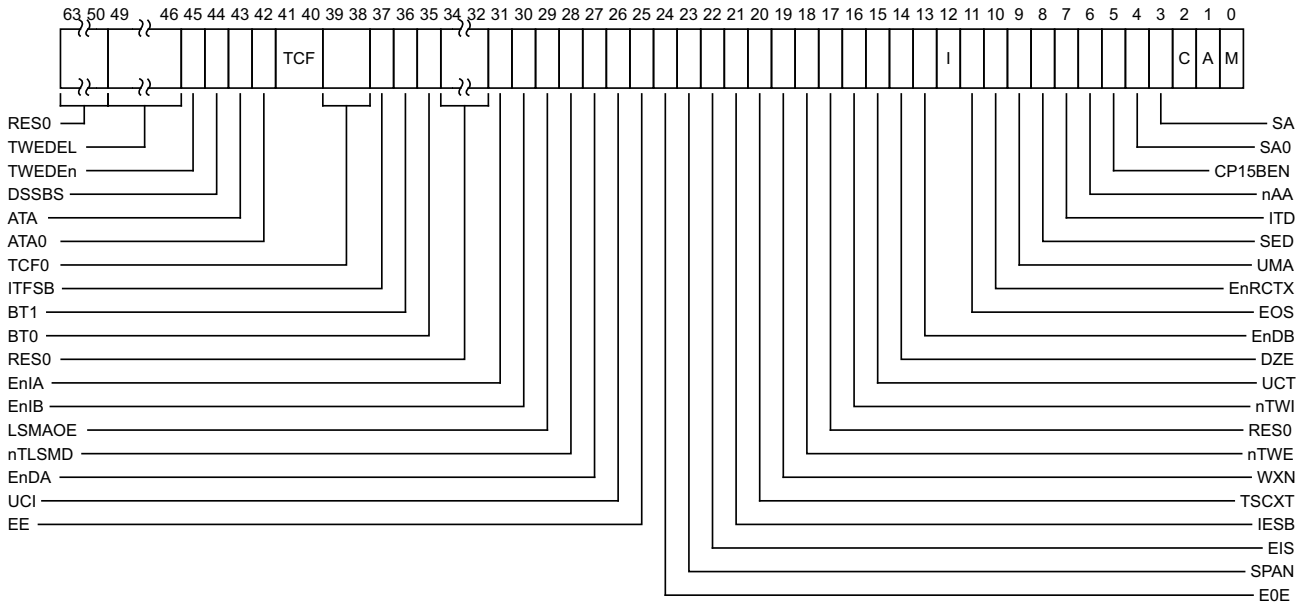
AArch64 System register SCTLR_EL1[31:0] is architecturally mapped to AArch32 System register SCTLR[31:0].

Attributes

SCTLR_EL1 is a 64-bit register.

Field descriptions

The SCTLR_EL1 bit assignments are:



Bits [63:50]

Reserved, RES0.

TWEDEL, bits [49:46]

When ARMv8.6-TWED is implemented:

TWE Delay. A 4-bit unsigned number that, when SCTLR_EL1.TWEDEn is 1, encodes the minimum delay in taking a trap of WFE caused by SCTLR_EL1.nTWE as $2^{(TWEDEL + 8)}$ cycles.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TWEDEn, bit [45]

When ARMv8.6-TWED is implemented:

TWE Delay Enable. Enables a configurable delayed trap of the WFE instruction caused by SCTLR_EL1.nTWE.

`0b0` The delay for taking a WFE trap is IMPLEMENTATION DEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` The delay for taking a WFE trap is at least the number of cycles defined in SCTLR_EL1.TWEDEL.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DSSBS, bit [44]

When ARMv8.0-SSBS is implemented:

Default PSTATE.SSBS value on Exception Entry. The defined values are:

`0b0` PSTATE.SSBS is set to 0 on an exception to EL1.

`0b1` PSTATE.SSBS is set to 1 on an exception to EL1.

In a system where the PE resets into EL1, this field resets to an IMPLEMENTATION DEFINED value.

Otherwise:

Reserved, RES0.

ATA, bit [43]

When ARMv8.5-MemTag is implemented:

Allocation Tag Access in EL1. When SCR_EL3.ATA=1 and HCR_EL2.ATA=1, controls EL1 access to Allocation Tags.

When access to Allocation Tags is prevented:

- Instructions which Load or Store data are Unchecked.
- Instructions which Load or Store Allocation Tags treat the Allocation Tag as RAZ/WI.
- Instructions which insert Logical Address Tags into addresses treat the Allocation Tag used to generate the Logical Address Tag as 0.
- Cache maintenance instructions which invalidate Allocation Tags from caches behave as the equivalent Clean and Invalidate operation on Allocation Tags.

`0b0` Access to Allocation Tags is prevented.

`0b1` Access to Allocation Tags is not prevented.

This bit is permitted to be cached in a TLB.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ATA0, bit [42]

When ARMv8.5-MemTag is implemented:

Allocation Tag Access in EL0. When SCR_EL3.ATA=1, HCR_EL2.ATA=1, and HCR_EL2.{E2H,TGE} != {1,1}, controls EL0 access to Allocation Tags.

When access to Allocation Tags is prevented:

- Instructions which Load or Store data are Unchecked.
- Instructions which Load or Store Allocation Tags treat the Allocation Tag as RAZ/WI.
- Instructions which insert Logical Address Tags into addresses treat the Allocation Tag used to generate the Logical Address Tag as 0.
- Cache maintenance instructions which invalidate Allocation Tags from caches behave as the equivalent Clean and Invalidate operation on Allocation Tags.

`0b0` Access to Allocation Tags is prevented.

`0b1` Access to Allocation Tags is not prevented.

This field is permitted to be cached in a TLB.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

Software may change this control bit on a context switch.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TCF, bits [41:40]

When ARMv8.5-MemTag is implemented:

Tag Check Fault in EL1. Controls the effect of Tag Check Faults due to Loads and Stores in EL1.

`0b00` Tag Check Faults have no effect on the PE.

`0b01` Tag Check Faults causes a synchronous exception.

`0b10` Tag Check Faults are asynchronously accumulated.

The value `0b11` is reserved.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TCF0, bits [39:38]

When ARMv8.5-MemTag is implemented:

Tag Check Fault in EL0. When HCR_EL2.{E2H,TGE} != {1,1}, controls the effect of Tag Check Faults due to Loads and Stores in EL0.

`0b00` Tag Check Faults have no effect on the PE.

`0b01` Tag Check Faults causes a synchronous exception.

`0b10` Tag Check Faults are asynchronously accumulated.

The value `0b11` is reserved.

Note

Software may change this control bit on a context switch.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ITFSB, bit [37]

When ARMv8.5-MemTag is implemented:

When synchronous exceptions are not being generated by Tag Check Faults which are generated for Loads and Stores in EL0 or EL1, controls the auto-synchronization of Tag Check Faults into TFSRE0_EL1 and TFSR_EL1.

`0b0` Tag Check Faults are not synchronized on entry to EL1.

`0b1` Tag Check Faults are synchronized on entry to EL1.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

BT1, bit [36]

When ARMv8.5-BTI is implemented:

PAC Branch Type compatibility at EL1.

`0b0` When the PE is executing at EL1, PACIASP and PACIBSP are compatible with PSTATE.BTYPE == `0b11`.

`0b1` When the PE is executing at EL1, PACIASP and PACIBSP are not compatible with PSTATE.BTYPE == `0b11`.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

BT0, bit [35]

When ARMv8.5-BTI is implemented:

PAC Branch Type compatibility at EL0.

`0b0` When the PE is executing at EL0, PACIASP and PACIBSP are compatible with PSTATE.BTYPE == `0b11`.

`0b1` When the PE is executing at EL0, PACIASP and PACIBSP are not compatible with PSTATE.BTYPE == `0b11`.

When HCR_EL2.E2H == 1 && HCR_EL2.TGE == 1, the value of the SCTLR_EL1.BT0 has no effect on execution at EL0

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [34:32]

Reserved, RES0.

EnIA, bit [31]

When ARMv8.3-PAuth is implemented:

Controls enabling of pointer authentication (using the APIAKey_EL1 key) of instruction addresses in the EL1&0 translation regime.

For more information, see *System register control of pointer authentication* on page D5-2539.

`0b0` Pointer authentication (using the APIAKey_EL1 key) of instruction addresses is not enabled.

`0b1` Pointer authentication (using the APIAKey_EL1 key) of instruction addresses is enabled.

Note

This field controls the behavior of the AddPACIA and AuthIA pseudocode functions. Specifically, when the field is 1, AddPACIA returns a copy of a pointer to which a pointer authentication code has been added, and AuthIA returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnIB, bit [30]

When ARMv8.3-PAuth is implemented:

Controls enabling of pointer authentication (using the APIBKey_EL1 key) of instruction addresses in the EL1&0 translation regime.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For more information, see *System register control of pointer authentication* on page D5-2539.

`0b0` Pointer authentication (using the APIBKey_EL1 key) of instruction addresses is not enabled.

`0b1` Pointer authentication (using the APIBKey_EL1 key) of instruction addresses is enabled.

Note

This field controls the behavior of the AddPACIB and AuthIB pseudocode functions. Specifically, when the field is 1, AddPACIB returns a copy of a pointer to which a pointer authentication code has been added, and AuthIB returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

LSMAOE, bit [29]

When ARMv8.2-LSMAOC is implemented:

Load Multiple and Store Multiple Atomicity and Ordering Enable.

`0b0` For all memory accesses at EL0, A32 and T32 Load Multiple and Store Multiple can have an interrupt taken during the sequence memory accesses, and the memory accesses are not required to be ordered.

`0b1` The ordering and interrupt behavior of A32 and T32 Load Multiple and Store Multiple at EL0 is as defined for Armv8.0.

This bit is permitted to be cached in a TLB.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1,1}, this bit has no effect on execution at EL0.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

nTLSMD, bit [28]

When ARMv8.2-LSMAOC is implemented:

No Trap Load Multiple and Store Multiple to Device-nGRE/Device-nGnRE/Device-nGnRnE memory.

`0b0` All memory accesses by A32 and T32 Load Multiple and Store Multiple at EL0 that are marked at stage 1 as Device-nGRE/Device-nGnRE/Device-nGnRnE memory are trapped and generate a stage 1 Alignment fault.

`0b1` All memory accesses by A32 and T32 Load Multiple and Store Multiple at EL0 that are marked at stage 1 as Device-nGRE/Device-nGnRE/Device-nGnRnE memory are not trapped.

This bit is permitted to be cached in a TLB.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1,1}, this bit has no effect on execution at EL0.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

EnDA, bit [27]

When ARMv8.3-PAuth is implemented:

Controls enabling of pointer authentication (using the APDAKey_EL1 key) of instruction addresses in the EL1&0 translation regime.

For more information, see *System register control of pointer authentication* on page D5-2539.

`0b0` Pointer authentication (using the APDAKey_EL1 key) of data addresses is not enabled.

`0b1` Pointer authentication (using the APDAKey_EL1 key) of data addresses is enabled.

Note

This field controls the behavior of the AddPACDA and AuthDA pseudocode functions. Specifically, when the field is 1, AddPACDA returns a copy of a pointer to which a pointer authentication code has been added, and AuthDA returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

UCI, bit [26]

Traps EL0 execution of cache maintenance instructions, to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from AArch64 state only, reported using EC syndrome value `0x18`.

This applies to DC CVAU, DC CIVAC, DC CVAC, DC CVAP, and IC IVAU.

If ARMv8.2-DCCVADP is implemented, this trap also applies to DC CVADP.

If ARMv8.5-MemTag is implemented, this trap also applies to DC CIGVAC, DC CIGDVAC, DC CGVAC, DC CGDVAC, DC CGVAP, and DC CGDVAP.

If ARMv8.2-DCCVADP and ARMv8.5-MemTag are implemented, this trap also applies to DC CGVADP and DC CGDVADP.

`0b0` Execution of the specified instructions at EL0 using AArch64 is trapped.

`0b1` This control does not cause any instructions to be trapped.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

If the Point of Coherency is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean, or clean and invalidate instruction that operates by VA to the point of coherency can be trapped when the value of this control is 1.

If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean by VA to the Point of Unification instruction can be trapped when the value of this control is 1.

If the Point of Unification is before any level of instruction cache, it is IMPLEMENTATION DEFINED whether the execution of any instruction cache invalidate by VA to the Point of Unification instruction can be trapped when the value of this control is 1.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

EE, bit [25]

Endianness of data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime.

The possible values of this bit are:

`0b0` Explicit data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime are little-endian.

`0b1` Explicit data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime are big-endian.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If an implementation does not provide Big-endian support at Exception Levels higher than EL0, this bit is RES0.

If an implementation does not provide Little-endian support at Exception Levels higher than EL0, this bit is RES1.

The EE bit is permitted to be cached in a TLB.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on the PE.

In a system where the PE resets into EL1, this field resets to an IMPLEMENTATION DEFINED value.

E0E, bit [24]

Endianness of data accesses at EL0.

The possible values of this bit are:

`0b0` Explicit data accesses at EL0 are little-endian.

`0b1` Explicit data accesses at EL0 are big-endian.

If an implementation only supports Little-endian accesses at EL0 then this bit is RES0. This option is not permitted when SCTLR_EL1.EE is RES1.

If an implementation only supports Big-endian accesses at EL0 then this bit is RES1. This option is not permitted when SCTLR_EL1.EE is RES0.

This bit has no effect on the endianness of `LDTR`, `LDTRH`, `LDTRSH`, `LDTRSW`, `STTR`, and `STTRH` instructions executed at EL1.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

SPAN, bit [23]

When ARMv8.1-PAN is implemented:

Set Privileged Access Never, on taking an exception to EL1.

`0b0` PSTATE.PAN is set to 1 on taking an exception to EL1.

`0b1` The value of PSTATE.PAN is left unchanged on taking an exception to EL1.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

EIS, bit [22]

When ARMv8.5-CSEH is implemented:

Exception Entry is Context Synchronizing. The defined values are:

`0b0` The taking of an exception to EL1 is not a context synchronizing event.

`0b1` The taking of an exception to EL1 is a context synchronizing event.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1,1}, this bit has no effect on execution at EL0.

If SCTLR_EL1.EIS is set to `0b0`:

- Indirect writes to ESR_EL1, FAR_EL1, SPSR_EL1, ELR_EL1 are synchronized on exception entry to EL1, so that a direct read of the register after exception entry sees the indirectly written value caused by the exception entry.
- Memory transactions, including instruction fetches, from an Exception level always use the translation resources associated with that translation regime.
- Exception Catch debug events are synchronous debug events.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- DCPS* and DRPS instructions are context synchronization events.

The following are not affected by the value of SCTLR_EL1.EIS:

- Changes to the PSTATE information on entry to EL1.
- Behavior of accessing the banked copies of the stack pointer using the SP register name for loads, stores and data processing instructions.
- Exit from Debug state.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

IESB, bit [21]

When ARMv8.2-IESB is implemented:

Implicit Error Synchronization event enable. Possible values are:

`0b0` Disabled.

`0b1` An implicit error synchronization event is added:

- At each exception taken to EL1.
- Before the operational pseudocode of each `ERET` instruction executed at EL1.

When the PE is in Debug state, the effect of this field is CONSTRAINED UNPREDICTABLE, and its Effective value might be 0 or 1 regardless of the value of the field. If the Effective value of the field is 1, then an implicit error synchronization event is added after each `DCPSx` instruction taken to EL1 and before each `DRPS` instruction executed at EL1, in addition to the other cases where it is added.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TSCXT, bit [20]

When ARMv8.0-CSV2 is implemented:

Trap EL0 Access to the SCXTNUM_EL0 register, when EL0 is using AArch64. The defined values are:

`0b0` EL0 access to SCXTNUM_EL0 is not disabled by this mechanism.

`0b1` EL0 access to SCXTNUM_EL0 is disabled, causing an exception to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1.

The value of SCXTNUM_EL0 is treated as 0.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1,1}, this bit has no effect on execution at EL0.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

WXN, bit [19]

Write permission implies XN (Execute-never). For the EL1&0 translation regime, this bit can force all memory regions that are writable to be treated as XN. The possible values of this bit are:

`0b0` This control has no effect on memory access permissions.

`0b1` Any region that is writable in the EL1&0 translation regime is forced to XN for accesses from software executing at EL1 or EL0.

This bit applies only when SCTLR_EL1.M bit is set.

The WXN bit is permitted to be cached in a TLB.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on the PE.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

nTWE, bit [18]

Traps EL0 execution of WFE instructions to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from both Execution states, reported using EC syndrome value `0x01`.

`0b0` Any attempt to execute a WFE instruction at EL0 is trapped, if the instruction would otherwise have caused the PE to enter a low-power state.

`0b1` This control does not cause any instructions to be trapped.

In AArch32 state, the attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

Note

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Bit [17]

Reserved, RES0.

nTWI, bit [16]

Traps EL0 execution of WFI instructions to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from both Execution states, reported using EC syndrome value `0x01`.

`0b0` Any attempt to execute a WFI instruction at EL0 is trapped, if the instruction would otherwise have caused the PE to enter a low-power state.

`0b1` This control does not cause any instructions to be trapped.

In AArch32 state, the attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

Note

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

UCT, bit [15]

Traps EL0 accesses to the CTR_EL0 to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from AArch64 state only, reported using EC value `0x18`.

`0b0` Accesses to the CTR_EL0 from EL0 using AArch64 are trapped.

`0b1` This control does not cause any instructions to be trapped.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

DZE, bit [14]

Traps EL0 execution of DC ZVA instructions to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from AArch64 state only, reported using EC syndrome value `0x18`.

If ARMv8.5-MemTag is implemented, this trap also applies to DC GVA and DC GZVA.

`0b0` Any attempt to execute an instruction that this trap applies to at EL0 using AArch64 is trapped.

Reading DCZID_EL0.DZP from EL0 returns 1, indicating that the instructions this trap applies to are not supported.

`0b1` This control does not cause any instructions to be trapped.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

EnDB, bit [13]

When ARMv8.3-PAuth is implemented:

Controls enabling of pointer authentication (using the APDBKey_EL1 key) of instruction addresses in the EL1&0 translation regime.

For more information, see *System register control of pointer authentication* on page D5-2539.

`0b0` Pointer authentication (using the APDBKey_EL1 key) of data addresses is not enabled.

`0b1` Pointer authentication (using the APDBKey_EL1 key) of data addresses is enabled.

Note

This field controls the behavior of the AddPACDB and AuthDB pseudocode functions. Specifically, when the field is 1, AddPACDB returns a copy of a pointer to which a pointer authentication code has been added, and AuthDB returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

I, bit [12]

Instruction access Cacheability control, for accesses at EL0 and EL1:

`0b0` All instruction access to Normal memory from EL0 and EL1 are Non-cacheable for all levels of instruction and unified cache.

If the value of SCTLR_EL1.M is 0, instruction accesses from stage 1 of the EL1&0 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.

`0b1` This control has no effect on the Cacheability of instruction access to Normal memory from EL0 and EL1.

If the value of SCTLR_EL1.M is 0, instruction accesses from stage 1 of the EL1&0 translation regime are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.

When the value of the HCR_EL2.DC bit is 1, then instruction access to Normal memory from EL0 and EL1 are Cacheable regardless of the value of the SCTLR_EL1.I bit.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on the PE.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In a system where the PE resets into EL1, this field resets to `0`.

EOS, bit [11]

When ARMv8.5-CSEH is implemented:

Exception Exit is Context Synchronizing. The defined values are:

`0b0` An exception return from EL1 is not a context synchronizing event

`0b1` An exception return from EL1 is a context synchronizing event

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1,1}, this bit has no effect on execution at EL0.

If SCTLR_EL1.EOS is set to `0b0`:

- Memory transactions, including instruction fetches, from an Exception level always use the translation resources associated with that translation regime.
- Exception Catch debug events are synchronous debug events.
- DCPS* and DRPS instructions are context synchronization events.

The following are not affected by the value of SCTLR_EL1.EOS:

- The indirect write of the PSTATE and PC values from SPSR_EL1 and ELR_EL1 on exception return is synchronized.
- Behavior of accessing the banked copies of the stack pointer using the SP register name for loads, stores and data processing instructions.
- Exit from Debug state.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

EnRCTX, bit [10]

When ARMv8.0-PredInv is implemented:

Enable EL0 Access to the following instructions:

- AArch32 CFPRCTX, DVPRCTX and CPPRCTX instructions.
- AArch64 CFP RCTX, DVP RCT and CPP RCTX instructions.

The defined values are:

`0b0` EL0 access to these instructions is disabled, and these instructions are trapped to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1.

`0b1` EL0 access to these instructions is enabled.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1,1}, this bit has no effect on execution at EL0.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

UMA, bit [9]

User Mask Access. Traps EL0 execution of MSR and MRS instructions that access the PSTATE.{D, A, I, F} masks to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from AArch64 state only, reported using EC syndrome value `0x18`.

`0b0` Any attempt at EL0 using AArch64 to execute an `MRS`, `MSR(register)`, or `MSR(immediate)` instruction that accesses the DAIF is trapped.

`0b1` This control does not cause any instructions to be trapped.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

SED, bit [8]

When AArch32 is supported at any Exception level and EL0 is capable of using AArch32:

SETEND instruction disable. Disables SETEND instructions at EL0 using AArch32.

`0b0` SETEND instruction execution is enabled at EL0 using AArch32.

`0b1` SETEND instructions are UNDEFINED at EL0 using AArch32 and any attempt at EL0 to access a SETEND instruction generates an exception to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, reported using EC syndrome value `0x00`.

If the implementation does not support mixed-endian operation at any Exception level, this bit is RES1.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

ITD, bit [7]

When AArch32 is supported at any Exception level and EL0 is capable of using AArch32:

IT Disable. Disables some uses of IT instructions at EL0 using AArch32.

`0b0` All IT instruction functionality is enabled at EL0 using AArch32.

`0b1` Any attempt at EL0 using AArch32 to execute any of the following is UNDEFINED and generates an exception, reported using EC syndrome value `0x00`, to EL1 or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1:

- All encodings of the IT instruction with hw1[3:0]!=1000.
- All encodings of the subsequent instruction with the following values for hw1:
  - `0b11xxxxxxxxxxxxxx`: All 32-bit instructions, and the 16-bit instructions B, UDF, SVC, LDM, and STM.
  - `0b1011xxxxxxxxxxxx`: All instructions in 'Miscellaneous 16-bit instructions' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile, section F3.2.5.
  - `0b10100xxxxxxxxxxx`: ADD Rd, PC, #imm
  - `0b01001xxxxxxxxxxx`: LDR Rd, [PC, #imm]
  - `0b0100x1xxx1111xxx`: ADD Rdn, PC; CMP Rn, PC; MOV Rd, PC; BX PC; BLX PC.
  - `0b010001xx1xxxx111`: ADD PC, Rm; CMP PC, Rm; MOV PC, Rm. This pattern also covers unpredictable cases with BLX Rn.

These instructions are always UNDEFINED, regardless of whether they would pass or fail the condition code check that applies to them as a result of being in an IT block.

It is IMPLEMENTATION DEFINED whether the IT instruction is treated as:

- A 16-bit instruction, that can only be followed by another 16-bit instruction.
- The first half of a 32-bit instruction.

This means that, for the situations that are UNDEFINED, either the second 16-bit instruction or the 32-bit instruction is UNDEFINED.

An implementation might vary dynamically as to whether IT is treated as a 16-bit instruction or the first half of a 32-bit instruction.

If an instruction in an active IT block that would be disabled by this field sets this field to 1 then behavior is CONSTRAINED UNPREDICTABLE. For more information see *Changes to an ITD control by an instruction in an IT block* on page E1-3958

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ITD is optional, but if it is implemented in the SCTLR then it must also be implemented in the SCTLR_EL1. If it is not implemented then this bit is RAZ/WI.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

nAA, bit [6]

When ARMv8.4-LSE is implemented:

Non-aligned access. This bit controls generation of Alignment faults at EL1 and EL0 under certain conditions.

`0b0` LDAPR, LDAPRH, LDAPUR, LDAPURH, LDAPURSH, LDAPURSW, LDAR, LDARH, LDLAR, LDLARH, STLLR, STLLRH, STLR, STLRH, STLUR, and STLURH generate an Alignment fault if all bytes being accessed are not within a single 16-byte quantity, aligned to 16 bytes for accesses.

`0b1` This control bit does not cause LDAPR, LDAPRH, LDAPUR, LDAPURH, LDAPURSH, LDAPURSW, LDAR, LDARH, LDLAR, LDLARH, STLLR, STLLRH, STLR, STLRH, STLUR, or STLURH to generate an Alignment fault if all bytes being accessed are not within a single 16-byte quantity, aligned to 16 bytes.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

CP15BEN, bit [5]

When AArch32 is supported at any Exception level and EL0 is capable of using AArch32:

System instruction memory barrier enable. Enables accesses to the DMB, DSB, and ISB System instructions in the (coproc==`0b1111`) encoding space from EL0:

`0b0` EL0 using AArch32: EL0 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is UNDEFINED and generates an exception to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1. The exception is reported using EC syndrome value `0x00`.

`0b1` EL0 using AArch32: EL0 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is enabled.

CP15BEN is optional, but if it is implemented in the SCTLR then it must also be implemented in the SCTLR_EL1. If it is not implemented then this bit is RAO/WI.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SA0, bit [4]

SP Alignment check enable for EL0. When set to 1, if a load or store instruction executed at EL0 uses the SP as the base address and the SP is not aligned to a 16-byte boundary, then a SP alignment fault exception is generated. For more information, see *SP alignment checking* on page D1-2333.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

SA, bit [3]

SP Alignment check enable. When set to 1, if a load or store instruction executed at EL1 uses the SP as the base address and the SP is not aligned to a 16-byte boundary, then a SP alignment fault exception is generated. For more information, see *SP alignment checking* on page D1-2333.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on the PE.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

C, bit [2]

Cacheability control, for data accesses.

`0b0` All data access to Normal memory from EL0 and EL1, and all Normal memory accesses to the EL1&0 stage 1 translation tables, are Non-cacheable for all levels of data and unified cache.

`0b1` This control has no effect on the Cacheability of:

- Data access to Normal memory from EL0 and EL1.
- Normal memory accesses to the EL1&0 stage 1 translation tables.

When the value of the HCR_EL2.DC bit is 1, the PE ignores SCLTR.C. This means that Non-secure EL0 and Non-secure EL1 data accesses to Normal memory are Cacheable.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on the PE.

In a system where the PE resets into EL1, this field resets to `0`.

A, bit [1]

Alignment check enable. This is the enable bit for Alignment fault checking at EL1 and EL0 .

`0b0` Alignment fault checking disabled when executing at EL1 or EL0.

Instructions that load or store one or more registers, other than load/store exclusive and load-acquire/store-release, do not check that the address being accessed is aligned to the size of the data element(s) being accessed.

`0b1` Alignment fault checking enabled when executing at EL1 or EL0.

All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element(s) being accessed. If this check fails it causes an Alignment fault, which is taken as a Data Abort exception.

Load/store exclusive and load-acquire/store-release instructions have an alignment check regardless of the value of the A bit.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

M, bit [0]

MMU enable for EL1&0 stage 1 address translation.

`0b0` EL1&0 stage 1 address translation disabled.

See the SCTLR_EL1.I field for the behavior of instruction accesses to Normal memory.

`0b1` EL1&0 stage 1 address translation enabled.

If the value of HCR_EL2.{DC, TGE} is not {0, 0} then in Non-secure state the PE behaves as if the value of the SCTLR_EL1.M field is 0 for all purposes other than returning the value of a direct read of the field.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on the PE.

In a system where the PE resets into EL1, this field resets to `0`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the SCTLR_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic SCTLR_EL1 or SCTLR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, SCTLR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.SCTLR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x110];
    else
        return SCTLR_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return SCTLR_EL2;
    else
        return SCTLR_EL1;
elsif PSTATE.EL == EL3 then
    return SCTLR_EL1;
```

MSR SCTLR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.SCTLR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x110] = X[t];
    else
        SCTLR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        SCTLR_EL2 = X[t];
    else
        SCTLR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    SCTLR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, SCTLR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        return NVMem[0x110];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return SCTLR_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        return SCTLR_EL1;
    else
        UNDEFINED;
```

MSR SCTLR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        NVMem[0x110] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        SCTLR_EL1 = X[t];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        SCTLR_EL1 = X[t];
    else
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.114 SCTLR_EL2, System Control Register (EL2)

The SCTLR_EL2 characteristics are:

Purpose

Provides top level control of the system, including its memory system, at EL2.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, these controls apply also to execution at EL0.

Configurations

AArch64 System register SCTLR_EL2[31:0] is architecturally mapped to AArch32 System register HSCTLR[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

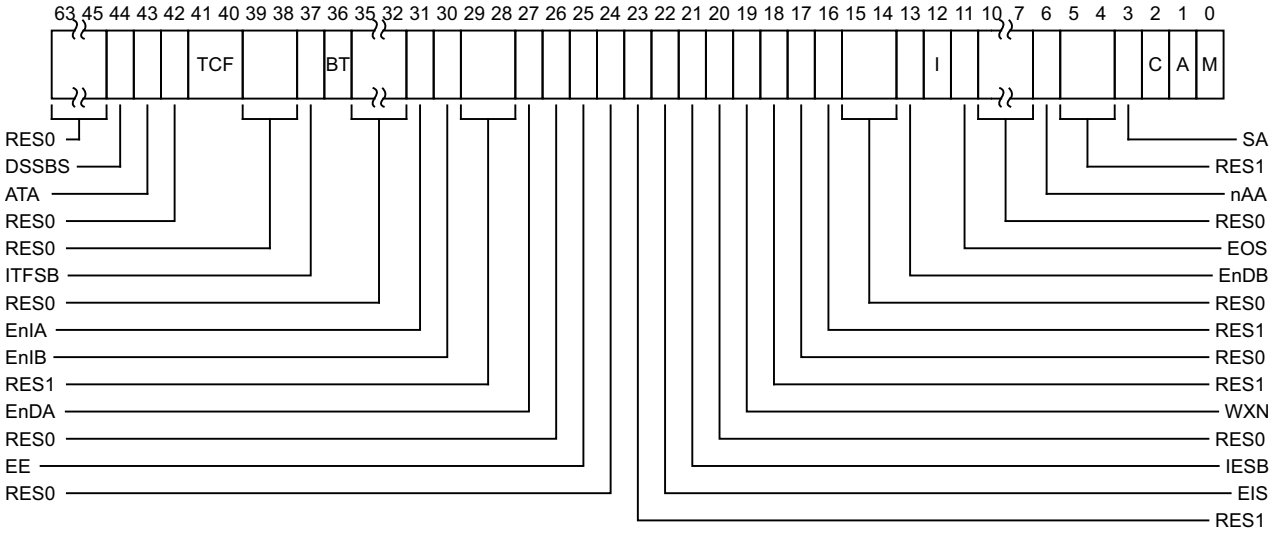
Attributes

SCTLR_EL2 is a 64-bit register.

Field descriptions

The SCTLR_EL2 bit assignments are:

When HCR_EL2.E2H != 0b1 or HCR_EL2.TGE != 0b1:



This format applies in all Armv8.0 implementations, and from Armv8.1 when the Effective value of HCR_EL2.{E2H, TGE} != {1, 1}.

Bits [63:45]

Reserved, RES0.

DSSBS, bit [44]

When ARMv8.0-SSBS is implemented:

Default PSTATE.SSBS value on Exception Entry.

`0b0` PSTATE.SSBS is set to 0 on an exception to EL2.

`0b1` PSTATE.SSBS is set to 1 on an exception to EL2.

In a system where the PE resets into EL2, this field resets to an IMPLEMENTATION DEFINED value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

Reserved, RES0.

ATA, bit [43]

When ARMv8.5-MemTag is implemented:

Allocation Tag Access. When SCR_EL3.ATA=1, controls EL2 access to Allocation Tags.

When access to Allocation Tags is prevented:

- Instructions that Load or Store data are Unchecked.
- Instructions that Load or Store Allocation Tags treat the Allocation Tag as RAZ/WI.
- Instructions that insert Logical Address Tags into addresses treat the Allocation Tag used to generate the Logical Address Tag as 0.
- Cache maintenance instructions that invalidate Allocation Tags from caches behave as the equivalent Clean and Invalidate operation on Allocation Tags.

`0b0` Access to Allocation Tags is prevented.

`0b1` Access to Allocation Tags is not prevented.

This bit is permitted to be cached in a TLB.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [42]

Reserved, RES0.

TCF, bits [41:40]

When ARMv8.5-MemTag is implemented:

Tag Check Fault. Controls the effect of Tag Check Faults due to Loads and Stores in EL2.

`0b00` Tag Check Faults have no effect on the PE.

`0b01` Tag Check Faults cause a synchronous exception.

`0b10` Tag Check Faults are asynchronously accumulated.

The value `0b11` is reserved.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [39:38]

Reserved, RES0.

ITFSB, bit [37]

When ARMv8.5-MemTag is implemented:

When synchronous exceptions are not being generated by Tag Check Faults, which are generated for Loads and Stores in EL0, EL1 or EL2, controls the auto-synchronization of Tag Check Faults into TFSRE0_EL1, TFSR_EL1, and TFSR_EL2.

`0b0` Tag Check Faults are not synchronized on entry to EL2.

`0b1` Tag Check Faults are synchronized on entry to EL2.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

BT, bit [36]

When ARMv8.5-BTI is implemented:

PAC Branch Type compatibility at EL2.

`0b0` When the PE is executing at EL2, PACIASP and PACIBSP are compatible with PSTATE.BTYPE == `0b11`.

`0b1` When the PE is executing at EL2, PACIASP and PACIBSP are not compatible with PSTATE.BTYPE == `0b11`.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [35:32]

Reserved, RES0.

EnIA, bit [31]

When ARMv8.3-PAuth is implemented:

Controls enabling of pointer authentication (using the APIAKey_EL1 key) of instruction addresses in the EL2&0 translation regime.

For more information, see *System register control of pointer authentication* on page D5-2539.

`0b0` Pointer authentication (using the APIAKey_EL1 key) of instruction addresses is not enabled.

`0b1` Pointer authentication (using the APIAKey_EL1 key) of instruction addresses is enabled.

Note

This field controls the behavior of the AddPACIA and AuthIA pseudocode functions. Specifically, when the field is 1, AddPACIA returns a copy of a pointer to which a pointer authentication code has been added, and AuthIA returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnIB, bit [30]

When ARMv8.3-PAuth is implemented:

Controls enabling of pointer authentication (using the APIBKey_EL1 key) of instruction addresses in the EL2&0 translation regime.

For more information, see *System register control of pointer authentication* on page D5-2539.

`0b0` Pointer authentication (using the APIBKey_EL1 key) of instruction addresses is not enabled.

`0b1` Pointer authentication (using the APIBKey_EL1 key) of instruction addresses is enabled.

Note

This field controls the behavior of the AddPACIB and AuthIB pseudocode functions. Specifically, when the field is 1, AddPACIB returns a copy of a pointer to which a pointer authentication code has been added, and AuthIB returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

Reserved, RES0.

Bits [29:28]

Reserved, RES1.

EnDA, bit [27]

When ARMv8.3-PAuth is implemented:

Controls enabling of pointer authentication (using the APDAKey_EL1 key) of instruction addresses in the EL2&0 translation regime.

For more information, see *System register control of pointer authentication* on page D5-2539.

`0b0` Pointer authentication (using the APDAKey_EL1 key) of data addresses is not enabled.

`0b1` Pointer authentication (using the APDAKey_EL1 key) of data addresses is enabled.

Note

This field controls the behavior of the AddPACDA and AuthDA pseudocode functions. Specifically, when the field is 1, AddPACDA returns a copy of a pointer to which a pointer authentication code has been added, and AuthDA returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [26]

Reserved, RES0.

EE, bit [25]

Endianness of data accesses at EL2, stage 1 translation table walks in the EL2 or EL2&0 translation regime, and stage 2 translation table walks in the EL1&0 translation regime.

`0b0` Explicit data accesses at EL2, stage 1 translation table walks in the EL2 or EL2&0 translation regime, and stage 2 translation table walks in the EL1&0 translation regime are little-endian.

`0b1` Explicit data accesses at EL2, stage 1 translation table walks in the EL2 or EL2&0 translation regime, and stage 2 translation table walks in the EL1&0 translation regime are big-endian.

If an implementation does not provide Big-endian support at Exception Levels higher than EL0, this bit is RES0.

If an implementation does not provide Little-endian support at Exception Levels higher than EL0, this bit is RES1.

The EE bit is permitted to be cached in a TLB.

In a system where the PE resets into EL2, this field resets to an IMPLEMENTATION DEFINED value.

Bit [24]

Reserved, RES0.

Bit [23]

Reserved, RES1.

EIS, bit [22]

When ARMv8.5-CSEH is implemented:

Exception entry is a context synchronization event. The defined values are:

`0b0` The taking of an exception to EL2 is not a context synchronization event.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` The taking of an exception to EL2 is a context synchronization event.

If SCTLR_EL2.EIS is set to `0b0`:

- Indirect writes to ESR_EL2, FAR_EL2, SPSR_EL2, ELR_EL2, and HPFAR_EL2 are synchronized on exception entry to EL2, so that a direct read of the register after exception entry sees the indirectly written value caused by the exception entry.
- Memory transactions, including instruction fetches, from an Exception level always use the translation resources associated with that translation regime.
- Exception Catch debug events are synchronous debug events.
- DCPS* and DRPS instructions are context synchronization events.

The following are not affected by the value of SCTLR_EL2.EIS:

- Changes to the PSTATE information on entry to EL2.
- Behavior of accessing the banked copies of the stack pointer using the SP register name for loads, stores, and data processing instructions.
- Exit from Debug state.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

IESB, bit [21]

When ARMv8.2-IESB is implemented:

Implicit Error Synchronization event enable.

`0b0` Disabled.

`0b1` An implicit error synchronization event is added:

- At each exception taken to EL2.
- Before the operational pseudocode of each `ERET` instruction executed at EL2.

When the PE is in Debug state, the effect of this field is CONSTRAINED UNPREDICTABLE, and its Effective value might be 0 or 1 regardless of the value of the field. If the Effective value of the field is 1, then an implicit error synchronization event is added after each `DCPSx` instruction taken to EL2 and before each `DRPS` instruction executed at EL2, in addition to the other cases where it is added.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [20]

Reserved, RES0.

WXN, bit [19]

Write permission implies XN (Execute-never). For the EL2 or EL2&0 translation regime, this bit can force all memory regions that are writable to be treated as XN:

`0b0` This control has no effect on memory access permissions.

`0b1` Any region that is writable in the EL2 or EL2&0 translation regime is forced to XN for accesses from software executing at EL2.

This bit applies only when SCTLR_EL2.M bit is set.

The WXN bit is permitted to be cached in a TLB.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Bit [18]

Reserved, RES1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bit [17]

Reserved, RES0.

Bit [16]

Reserved, RES1.

Bits [15:14]

Reserved, RES0.

EnDB, bit [13]

When ARMv8.3-PAuth is implemented:

Controls enabling of pointer authentication (using the APDBKey_EL1 key) of instruction addresses in the EL2&0 translation regime.

For more information, see *System register control of pointer authentication* on page D5-2539.

`0b0` Pointer authentication (using the APDBKey_EL1 key) of data addresses is not enabled.

`0b1` Pointer authentication (using the APDBKey_EL1 key) of data addresses is enabled.

Note

This field controls the behavior of the AddPACDB and AuthDB pseudocode functions. Specifically, when the field is 1, AddPACDB returns a copy of a pointer to which a pointer authentication code has been added, and AuthDB returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

I, bit [12]

Instruction access Cacheability control, for accesses at EL2:

`0b0` All instruction accesses to Normal memory from EL2 are Non-cacheable for all levels of instruction and unified cache.

If the value of SCTLR_EL2.M is 0, instruction accesses from stage 1 of the EL2 or EL2&0 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.

`0b1` This control has no effect on the Cacheability of instruction access to Normal memory from EL2.

If the value of SCTLR_EL2.M is 0, instruction accesses from stage 1 of the EL2 or EL2&0 translation regime are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.

This bit has no effect on the EL1&0 or EL3 translation regimes.

In a system where the PE resets into EL2, this field resets to `0`.

EOS, bit [11]

When ARMv8.5-CSEH is implemented:

Exception exit is a context synchronization Event.

`0b0` An exception return from EL2 is not a context synchronization event.

`0b1` An exception return from EL2 is a context synchronization event.

If SCTLR_EL2.EOS is set to `0b0`:

- Memory transactions, including instruction fetches, from an Exception level always use the translation resources associated with that translation regime.
- Exception Catch debug events are synchronous debug events.
- DCPS* and DRPS instructions are context synchronization events.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following are not affected by the value of SCTLR_EL2.EOS:

- The indirect write of the PSTATE and PC values from SPSR_EL2 and ELR_EL2 on exception return is synchronized.
- Behavior of accessing the banked copies of the stack pointer using the SP register name for loads, stores, and data processing instructions.
- Exit from Debug state.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

Bits [10:7]

Reserved, RES0.

nAA, bit [6]

When ARMv8.4-LSE is implemented:

Non-aligned access. This bit controls generation of Alignment faults at EL2 under certain conditions.

`0b0` LDAPR, LDAPRH, LDAPUR, LDAPURH, LDAPURSH, LDAPURSW, LDAR, LDARH, LDLAR, LDLARH, STLLR, STLLRH, STLR, STLRH, STLUR, and STLURH generate an Alignment fault if all bytes being accessed are not within a single 16-byte quantity, aligned to 16 bytes for accesses.

`0b1` This control bit does not cause LDAPR, LDAPRH, LDAPUR, LDAPURH, LDAPURSH, LDAPURSW, LDAR, LDARH, LDLAR, LDLARH, STLLR, STLLRH, STLR, STLRH, STLUR, or STLURH to generate an Alignment fault if all bytes being accessed are not within a single 16-byte quantity, aligned to 16 bytes.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [5:4]

Reserved, RES1.

SA, bit [3]

SP Alignment check enable. When set to 1, if a load or store instruction executed at EL2 uses the SP as the base address and the SP is not aligned to a 16-byte boundary, then an SP alignment fault exception is generated. For more information, see *SP alignment checking* on page D1-2333.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

C, bit [2]

Cacheability control, for data accesses.

`0b0` All data accesses to Normal memory from EL2, and all Normal memory accesses to the EL2 translation tables, are Non-cacheable for all levels of data and unified cache.

`0b1` This control has no effect on the Cacheability of:

- Data access to Normal memory from EL2.
- Normal memory accesses to the EL2 translation tables.

This bit has no effect on the EL1&0 or EL3 translation regimes.

In a system where the PE resets into EL2, this field resets to `0`.

A, bit [1]

Alignment check enable. This is the enable bit for Alignment fault checking at EL2:

`0b0` Alignment fault checking disabled when executing at EL2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Instructions that load or store one or more registers, other than load/store exclusive and load-acquire/store-release, do not check that the address being accessed is aligned to the size of the data element(s) being accessed.

`0b1` Alignment fault checking enabled when executing at EL2.

All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element(s) being accessed. If this check fails it causes an Alignment fault, which is taken as a Data Abort exception.

Load/store exclusive and load-acquire/store-release instructions have an alignment check regardless of the value of the A bit.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

M, bit [0]

MMU enable for EL2 stage 1 address translation.

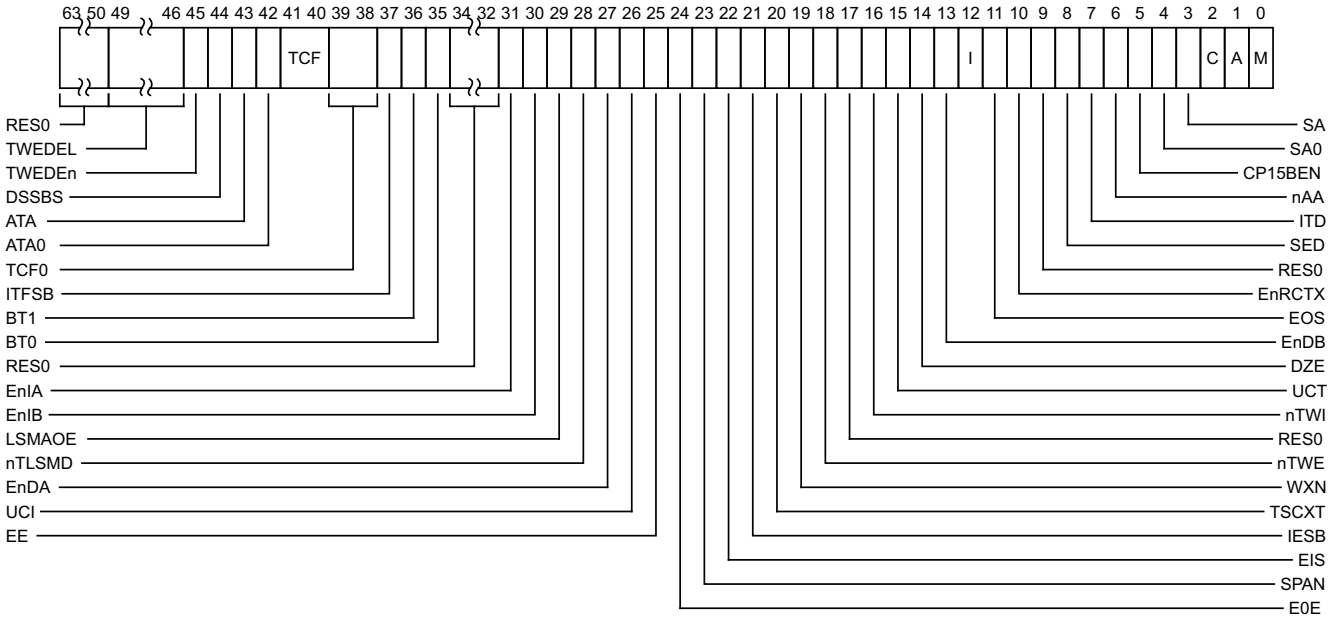
`0b0` EL2 stage 1 address translation disabled.

See the SCTLR_EL2.I field for the behavior of instruction accesses to Normal memory.

`0b1` EL2 stage 1 address translation enabled.

In a system where the PE resets into EL2, this field resets to `0`.

When HCR_EL2.E2H == 0b1 and HCR_EL2.TGE == 0b1:



This format applies only from Armv8.1 when EL2 is enabled in the current Security state and HCR_EL2.{E2H, TGE} == {1, 1}.

Bits [63:50]

Reserved, RES0.

TWEDEL, bits [49:46]

When ARMv8.6-TWED is implemented:

TWE Delay. A 4-bit unsigned number that, when SCTLR_EL2.TWEDEn is 1, encodes the minimum delay in taking a trap of WFE caused by SCTLR_EL2.nTWE as 2^(TWEDEL + 8) cycles.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

Reserved, RES0.

TWEDEn, bit [45]

When ARMv8.6-TWED is implemented:

TWE Delay Enable. Enables a configurable delayed trap of the WFE instruction caused by SCTLR_EL2.nTWE.

`0b0` The delay for taking a WFE trap is IMPLEMENTATION DEFINED.

`0b1` The delay for taking a WFE trap is at least the number of cycles defined in SCTLR_EL2.TWEDEL.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DSSBS, bit [44]

When ARMv8.0-SSBS is implemented:

Default PSTATE.SSBS value on Exception Entry.

`0b0` PSTATE.SSBS is set to 0 on an exception to EL2.

`0b1` PSTATE.SSBS is set to 1 on an exception to EL2.

In a system where the PE resets into EL2, this field resets to an IMPLEMENTATION DEFINED value.

Otherwise:

Reserved, RES0.

ATA, bit [43]

When ARMv8.5-MemTag is implemented:

Allocation Tag Access in EL2. When SCR_EL3.ATA=1, controls EL2 access to Allocation Tags.

When access to Allocation Tags is prevented:

- Instructions which Load or Store data are Unchecked.
- Instructions which Load or Store Allocation Tags treat the Allocation Tag as RAZ/WI.
- Instructions which insert Logical Address Tags into addresses treat the Allocation Tag used to generate the Logical Address Tag as 0.
- Cache maintenance instructions which invalidate Allocation Tags from caches behave as the equivalent Clean and Invalidate operation on Allocation Tags.

`0b0` Access to Allocation Tags is prevented.

`0b1` Access to Allocation Tags is not prevented.

This bit is permitted to be cached in a TLB.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ATA0, bit [42]

When ARMv8.5-MemTag is implemented:

Allocation Tag Access in EL0. When SCR_EL3.ATA=1, controls EL0 access to Allocation Tags.

When access to Allocation Tags is prevented:

- Instructions which Load or Store data are Unchecked.
- Instructions which Load or Store Allocation Tags treat the Allocation Tag as RAZ/WI.
- Instructions which insert Logical Address Tags into addresses treat the Allocation Tag used to generate the Logical Address Tag as 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Cache maintenance instructions which invalidate Allocation Tags from caches behave as the equivalent Clean and Invalidate operation on Allocation Tags.

`0b0` Access to Allocation Tags is prevented.

`0b1` Access to Allocation Tags is not prevented.

This field is permitted to be cached in a TLB.

Note

Software may change this control bit on a context switch.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TCF, bits [41:40]

When ARMv8.5-MemTag is implemented:

Tag Check Fault in EL2. Controls the effect of Tag Check Faults due to Loads and Stores in EL2.

`0b00` Tag Check Faults have no effect on the PE.

`0b01` Tag Check Faults causes a synchronous exception.

`0b10` Tag Check Faults are asynchronously accumulated.

The value `0b11` is reserved.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TCF0, bits [39:38]

When ARMv8.5-MemTag is implemented:

Tag Check Fault in EL0. Controls the effect of Tag Check Faults due to Loads and Stores in EL0.

`0b00` Tag Check Faults have no effect on the PE.

`0b01` Tag Check Faults causes a synchronous exception.

`0b10` Tag Check Faults are asynchronously accumulated.

The value `0b11` is reserved.

Note

Software may change this control bit on a context switch.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ITFSB, bit [37]

When ARMv8.5-MemTag is implemented:

When synchronous exceptions are not being generated by Tag Check Faults which are generated for Loads and Stores in EL0, EL1 or EL2, controls the auto-synchronization of Tag Check Faults into TFSRE0_EL1, TFSR_EL1 and TFSR_EL2.

`0b0` Tag Check Faults are not synchronized on entry to EL2.

`0b1` Tag Check Faults are synchronized on entry to EL2.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

BT1, bit [36]

When ARMv8.5-BTI is implemented:

PAC Branch Type compatibility at EL2.

`0b0` When the PE is executing at EL2, PACIASP and PACIBSP are compatible with PSTATE.BTYPE == `0b11`.

`0b1` When the PE is executing at EL2, PACIASP and PACIBSP are not compatible with PSTATE.BTYPE == `0b11`.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

BT0, bit [35]

When ARMv8.5-BTI is implemented:

PAC Branch Type compatibility at EL0.

`0b0` When the PE is executing at EL0, PACIASP and PACIBSP are compatible with PSTATE.BTYPE == `0b11`.

`0b1` When the PE is executing at EL0, PACIASP and PACIBSP are not compatible with PSTATE.BTYPE == `0b11`.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [34:32]

Reserved, RES0.

EnIA, bit [31]

When ARMv8.3-PAuth is implemented:

Controls enabling of pointer authentication (using the APIAKey_EL1 key) of instruction addresses in the EL2&0 translation regime.

For more information, see *System register control of pointer authentication* on page D5-2539.

`0b0` Pointer authentication (using the APIAKey_EL1 key) of instruction addresses is not enabled.

`0b1` Pointer authentication (using the APIAKey_EL1 key) of instruction addresses is enabled.

Note

This field controls the behavior of the AddPACIA and AuthIA pseudocode functions. Specifically, when the field is 1, AddPACIA returns a copy of a pointer to which a pointer authentication code has been added, and AuthIA returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnIB, bit [30]

When ARMv8.3-PAuth is implemented:

Controls enabling of pointer authentication (using the APIBKey_EL1 key) of instruction addresses in the EL2&0 translation regime.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For more information, see *System register control of pointer authentication* on page D5-2539.

`0b0` Pointer authentication (using the APIBKey_EL1 key) of instruction addresses is not enabled.

`0b1` Pointer authentication (using the APIBKey_EL1 key) of instruction addresses is enabled.

Note

This field controls the behavior of the AddPACIB and AuthIB pseudocode functions. Specifically, when the field is 1, AddPACIB returns a copy of a pointer to which a pointer authentication code has been added, and AuthIB returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

LSMAOE, bit [29]

When ARMv8.2-LSMAOC is implemented:

Load Multiple and Store Multiple Atomicity and Ordering Enable.

`0b0` For all memory accesses at EL0, A32 and T32 Load Multiple and Store Multiple can have an interrupt taken during the sequence memory accesses, and the memory accesses are not required to be ordered.

`0b1` The ordering and interrupt behavior of A32 and T32 Load Multiple and Store Multiple at EL0 is as defined for Armv8.0.

This bit is permitted to be cached in a TLB.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

nTLSMD, bit [28]

When ARMv8.2-LSMAOC is implemented:

No Trap Load Multiple and Store Multiple to Device-nGRE/Device-nGnRE/Device-nGnRnE memory.

`0b0` All memory accesses by A32 and T32 Load Multiple and Store Multiple at EL0 that are marked at stage 1 as Device-nGRE/Device-nGnRE/Device-nGnRnE memory are trapped and generate a stage 1 Alignment fault.

`0b1` All memory accesses by A32 and T32 Load Multiple and Store Multiple at EL0 that are marked at stage 1 as Device-nGRE/Device-nGnRE/Device-nGnRnE memory are not trapped.

This bit is permitted to be cached in a TLB.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

EnDA, bit [27]

When ARMv8.3-PAuth is implemented:

Controls enabling of pointer authentication (using the APDAKey_EL1 key) of instruction addresses in the EL2&0 translation regime.

For more information, see *System register control of pointer authentication* on page D5-2539.

`0b0` Pointer authentication (using the APDAKey_EL1 key) of data addresses is not enabled.

`0b1` Pointer authentication (using the APDAKey_EL1 key) of data addresses is enabled.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

This field controls the behavior of the AddPACDA and AuthDA pseudocode functions. Specifically, when the field is 1, AddPACDA returns a copy of a pointer to which a pointer authentication code has been added, and AuthDA returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

UCI, bit [26]

Traps EL0 execution of cache maintenance instructions to EL2, from AArch64 state only. This applies to DC CVAU, DC CIVAC, DC CVAC, DC CVAP, and IC IVAU.

If ARMv8.2-DCCVADP is implemented, this trap also applies to DC CVADP.

If ARMv8.5-MemTag is implemented, this trap also applies to DC CIGVAC, DC CIGDVAC, DC CGVAC, DC CGDVAC, DC CGVAP, and DC CGDVAP.

If ARMv8.2-DCCVADP and ARMv8.5-MemTag are implemented, this trap also applies to DC CGVADP and DC CGDVADP.

`0b0` Any attempt to execute an instruction that this trap applies to at EL0 using AArch64 is trapped to EL2.

`0b1` This control does not cause any instructions to be trapped.

If the Point of Coherency is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean, or clean and invalidate instruction that operates by VA to the point of coherency can be trapped when the value of this control is 1.

If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean by VA to the Point of Unification instruction can be trapped when the value of this control is 1.

If the Point of Unification is before any level of instruction cache, it is IMPLEMENTATION DEFINED whether the execution of any instruction cache invalidate by VA to the Point of Unification instruction can be trapped when the value of this control is 1.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

EE, bit [25]

Endianness of data accesses at EL2, stage 1 translation table walks in the EL2 or EL2&0 translation regime, and stage 2 translation table walks in the EL2&0 translation regime.

The possible values of this bit are:

`0b0` Explicit data accesses at EL2, stage 1 translation table walks in the EL2 or EL2&0 translation regime, and stage 2 translation table walks in the EL2&0 translation regime are little-endian.

`0b1` Explicit data accesses at EL2, stage 1 translation table walks in the EL2 or EL2&0 translation regime, and stage 2 translation table walks in the EL2&0 translation regime are big-endian.

If an implementation does not provide Big-endian support at Exception Levels higher than EL0, this bit is RES0.

If an implementation does not provide Little-endian support at Exception Levels higher than EL0, this bit is RES1.

The EE bit is permitted to be cached in a TLB.

In a system where the PE resets into EL2, this field resets to an IMPLEMENTATION DEFINED value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

E0E, bit [24]

Endianness of data accesses at EL0.

`0b0` Explicit data accesses at EL0 are little-endian.

`0b1` Explicit data accesses at EL0 are big-endian.

If an implementation only supports Little-endian accesses at EL0 then this bit is RES0. This option is not permitted when SCTLR_EL1.EE is RES1.

If an implementation only supports Big-endian accesses at EL0 then this bit is RES1. This option is not permitted when SCTLR_EL1.EE is RES0.

This bit has no effect on the endianness of `LDTR`, `LDTRH`, `LDTRSH`, `LDTRSW`, `STTR`, and `STTRH` instructions executed at EL1.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

SPAN, bit [23]

Set Privileged Access Never, on taking an exception to EL2.

`0b0` PSTATE.PAN is set to 1 on taking an exception to EL2.

`0b1` The value of PSTATE.PAN is left unchanged on taking an exception to EL2.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

EIS, bit [22]

When ARMv8.5-CSEH is implemented:

Exception Entry is a context synchronization event.

`0b0` The taking of an exception to EL2 is not a context synchronization event.

`0b1` The taking of an exception to EL2 is a context synchronization event.

If SCTLR_EL2.EIS is set to `0b0`:

- Indirect writes to ESR_EL2, FAR_EL2, SPSR_EL2, ELR_EL2, HPFAR_EL2 are synchronized on exception entry to EL2, so that a direct read of the register after exception entry sees the indirectly written value caused by the exception entry.
- Memory transactions, including instruction fetches, from an Exception level always use the translation resources associated with that translation regime.
- Exception Catch debug events are synchronous debug events.
- DCPS* and DRPS instructions are context synchronization events.

The following are not affected by the value of SCTLR_EL2.EIS:

- Changes to the PSTATE information on entry to EL2.
- Behavior of accessing the banked copies of the stack pointer using the SP register name for loads, stores and data processing instructions.
- Exit from Debug state.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

IESB, bit [21]

When ARMv8.2-IESB is implemented:

Implicit Error Synchronization event Enable.

`0b0` Disabled.

`0b1` An implicit error synchronization event is added:

- After each exception taken to EL2.
- Before the operational pseudocode of each ERET instruction executed at EL2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When the PE is in Debug state, the effect of this field is CONSTRAINED UNPREDICTABLE, and its Effective value might be 0 or 1 regardless of the value of the field. If the Effective value of the field is 1, then an implicit error synchronization event is added after each DCPSx instruction taken to EL2 and before each DRPS instruction executed at EL2, in addition to the other cases where it is added.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TSCXT, bit [20]

When ARMv8.0-CSV2 is implemented:

Trap EL0 Access to the SCXTNUM_EL0 register, when EL0 is using AArch64.

`0b0` EL0 access to SCXTNUM_EL0 is not disabled by this mechanism.

`0b1` EL0 access to SCXTNUM_EL0 is disabled, causing an exception to EL2, and the SCXTNUM_EL0 value is treated at 0.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

WXN, bit [19]

Write permission implies XN (Execute-never). For the EL2 or EL2&0 translation regime, this bit can force all memory regions that are writable to be treated as XN. The possible values of this bit are:

`0b0` This control has no effect on memory access permissions.

`0b1` Any region that is writable in the EL2 or EL2&0 translation regime is forced to XN for accesses from software executing at EL2.

This bit applies only when SCTLR_EL2.M bit is set.

The WXN bit is permitted to be cached in a TLB.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

nTWE, bit [18]

Traps EL0 execution of WFE instructions to EL2, from both Execution states.

`0b0` Any attempt to execute a WFE instruction at EL0 is trapped to EL2, if the instruction would otherwise have caused the PE to enter a low-power state.

`0b1` This control does not cause any instructions to be trapped.

In AArch32 state, the attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

Note

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Bit [17]

Reserved, RES0.

nTWI, bit [16]

Traps EL0 execution of WFI instructions to EL2, from both Execution states.

`0b0` Any attempt to execute a WFI instruction at EL0 is trapped EL2, if the instruction would otherwise have caused the PE to enter a low-power state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` This control does not cause any instructions to be trapped.

In AArch32 state, the attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

Note

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

UCT, bit [15]

Traps EL0 accesses to the CTR_EL0 to EL2, from AArch64 state only.

`0b0` Accesses to the CTR_EL0 from EL0 using AArch64 are trapped to EL2.

`0b1` This control does not cause any instructions to be trapped.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

DZE, bit [14]

Traps EL0 execution of DC ZVA instructions to EL2, from AArch64 state only.

If ARMv8.5-MemTag is implemented, this trap also applies to DC GVA and DC GZVA.

`0b0` Any attempt to execute an instruction that this trap applies to at EL0 using AArch64 is trapped to EL2. Reading DCZID_EL0.DZP from EL0 returns 1, indicating that the instructions that this trap applies to are not supported.

`0b1` This control does not cause any instructions to be trapped.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

EnDB, bit [13]

When ARMv8.3-PAuth is implemented:

Controls enabling of pointer authentication (using the APDBKey_EL1 key) of instruction addresses in the EL2&0 translation regime.

For more information, see *System register control of pointer authentication* on page D5-2539.

`0b0` Pointer authentication (using the APDBKey_EL1 key) of data addresses is not enabled.

`0b1` Pointer authentication (using the APDBKey_EL1 key) of data addresses is enabled.

Note

This field controls the behavior of the AddPACDB and AuthDB pseudocode functions. Specifically, when the field is 1, AddPACDB returns a copy of a pointer to which a pointer authentication code has been added, and AuthDB returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

I, bit [12]

Instruction access Cacheability control, for accesses at EL2 and EL0:

`0b0` All instruction access to Normal memory from EL2 and EL0 are Non-cacheable for all levels of instruction and unified cache.

If the value of SCTLR_EL2.M is 0, instruction accesses from stage 1 of the EL2&0 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` This control has no effect on the Cacheability of instruction access to Normal memory from EL2 and EL0.

If the value of SCTLR_EL2.M is 0, instruction accesses from stage 1 of the EL2&0 translation regime are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.

This bit has no effect on the EL3 translation regimes.

In a system where the PE resets into EL2, this field resets to `0`.

EOS, bit [11]

When ARMv8.5-CSEH is implemented:

Exception exit is a context synchronization event.

`0b0` An exception return from EL2 is not a context synchronization event.

`0b1` An exception return from EL2 is a context synchronization event.

If SCTLR_EL2.EOS is set to `0b0`:

- Memory transactions, including instruction fetches, from an Exception level always use the translation resources associated with that translation regime.
- Exception Catch debug events are synchronous debug events.
- DCPS* and DRPS instructions are context synchronization events.

The following are not affected by the value of SCTLR_EL2.EOS:

- The indirect write of the PSTATE and PC values from SPSR_EL2 and ELR_EL2 on exception return is synchronized.
- Behavior of accessing the banked copies of the stack pointer using the SP register name for loads, stores and data processing instructions.
- Exit from Debug state.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

EnRCTX, bit [10]

When ARMv8.0-PredInv is implemented:

Enable EL0 Access to the following instructions:

- AArch32 CFPRCTX, DVPRCTX and CPPRCTX instructions.
- AArch64 CFP RCTX, DVP RCT and CPP RCTX instructions.

The defined values are:

`0b0` EL0 access to these instructions is disabled, and these instructions are trapped to EL1.

`0b1` EL0 access to these instructions is enabled.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [9]

Reserved, RES0.

SED, bit [8]

When AArch32 is supported at any Exception level and EL0 is capable of using AArch32:

SETEND instruction disable. Disables SETEND instructions at EL0 using AArch32.

`0b0` SETEND instruction execution is enabled at EL0 using AArch32.

`0b1` SETEND instructions are UNDEFINED at EL0 using AArch32.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the implementation does not support mixed-endian operation at any Exception level, this bit is RES1.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

ITD, bit [7]

When AArch32 is supported at any Exception level and EL0 is capable of using AArch32:

IT Disable. Disables some uses of IT instructions at EL0 using AArch32.

`0b0` All IT instruction functionality is enabled at EL0 using AArch32.

`0b1` Any attempt at EL0 using AArch32 to execute any of the following is UNDEFINED:

- All encodings of the IT instruction with hw1[3:0]!=1000.
- All encodings of the subsequent instruction with the following values for hw1:
  - `0b11xxxxxxxxxxxxxx`: All 32-bit instructions, and the 16-bit instructions B, UDF, SVC, LDM, and STM.
  - `0b1011xxxxxxxxxxxx`: All instructions in 'Miscellaneous 16-bit instructions' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile, section F3.2.5.
  - `0b10100xxxxxxxxxxx`: ADD Rd, PC, #imm
  - `0b01001xxxxxxxxxxx`: LDR Rd, [PC, #imm]
  - `0b0100x1xxx1111xxx`: ADD Rdn, PC; CMP Rn, PC; MOV Rd, PC; BX PC; BLX PC.
  - `0b010001xx1xxxx111`: ADD PC, Rm; CMP PC, Rm; MOV PC, Rm. This pattern also covers UNPREDICTABLE cases with BLX Rn.

These instructions are always UNDEFINED, regardless of whether they would pass or fail the condition code check that applies to them as a result of being in an IT block.

It is IMPLEMENTATION DEFINED whether the IT instruction is treated as:

- A 16-bit instruction, that can only be followed by another 16-bit instruction.
- The first half of a 32-bit instruction.

This means that, for the situations that are UNDEFINED, either the second 16-bit instruction or the 32-bit instruction is UNDEFINED.

An implementation might vary dynamically as to whether IT is treated as a 16-bit instruction or the first half of a 32-bit instruction.

If an instruction in an active IT block that would be disabled by this field sets this field to 1 then behavior is CONSTRAINED UNPREDICTABLE. For more information see *Changes to an ITD control by an instruction in an IT block* on page E1-3958

ITD is optional, but if it is implemented in the SCTLR then it must also be implemented in the SCTLR_EL1. If it is not implemented then this bit is RAZ/WI.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

nAA, bit [6]

When ARMv8.4-LSE is implemented:

Non-aligned access. This bit controls generation of Alignment faults at EL2 and EL0 under certain conditions.

`0b0` LDAPR, LDAPRH, LDAPUR, LDAPURH, LDAPURSH, LDAPURSW, LDAR, LDARH, LDLAR, LDLARH, STLLR, STLLRH, STLR, STLRH, STLUR, and STLURH generate an Alignment fault if all bytes being accessed are not within a single 16-byte quantity, aligned to 16 bytes for accesses.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` This control bit does not cause LDAPR, LDAPRH, LDAPUR, LDAPURH, LDAPURSH, LDAPURSW, LDAR, LDARH, LDLAR, LDLARH, STLLR, STLLRH, STLR, STLRH, STLUR, or STLURH to generate an Alignment fault if all bytes being accessed are not within a single 16-byte quantity, aligned to 16 bytes.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

CP15BEN, bit [5]

When AArch32 is supported at any Exception level and EL0 is capable of using AArch32:

System instruction memory barrier enable. Enables accesses to the DMB, DSB, and ISB System instructions in the (coproc==`0b1111`) encoding space from EL0:

`0b0` EL0 using AArch32: EL0 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is UNDEFINED.

`0b1` EL0 using AArch32: EL0 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is enabled.

CP15BEN is optional, but if it is implemented in the SCTLR then it must also be implemented in the SCTLR_EL1. If it is not implemented then this bit is RAO/WI.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SA0, bit [4]

SP Alignment check enable for EL0. When set to 1, if a load or store instruction executed at EL0 uses the SP as the base address and the SP is not aligned to a 16-byte boundary, then a SP alignment fault exception is generated. For more information, see *SP alignment checking* on page D1-2333.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

SA, bit [3]

SP Alignment check enable. When set to 1, if a load or store instruction executed at EL2 uses the SP as the base address and the SP is not aligned to a 16-byte boundary, then a SP alignment fault exception is generated. For more information, see *SP alignment checking* on page D1-2333.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

C, bit [2]

Cacheability control, for data accesses.

`0b0` All data access to Normal memory from EL2 and EL0, and all Normal memory accesses to the EL2&0 translation tables, are Non-cacheable for all levels of data and unified cache.

`0b1` This control has no effect on the Cacheability of:

- Data access to Normal memory from EL2 and EL0.
- Normal memory accesses to the EL2&0 translation tables.

This bit has no effect on the EL3 translation regimes.

In a system where the PE resets into EL2, this field resets to `0`.

A, bit [1]

Alignment check enable. This is the enable bit for Alignment fault checking at EL2 and EL0.

`0b0` Alignment fault checking disabled when executing at EL2 and EL0.

Instructions that load or store one or more registers, other than load/store exclusive and load-acquire/store-release, do not check that the address being accessed is aligned to the size of the data element(s) being accessed.

`0b1` Alignment fault checking enabled when executing at EL2 and EL0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element(s) being accessed. If this check fails it causes an Alignment fault, which is taken as a Data Abort exception.

Load/store exclusive and load-acquire/store-release instructions have an alignment check regardless of the value of the A bit.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

M, bit [0]

MMU enable for EL2&0 stage 1 address translation. Possible values of this bit are:

`0b0` EL2&0 stage 1 address translation disabled.
See the SCTLR_EL2.I field for the behavior of instruction accesses to Normal memory.

`0b1` EL2&0 stage 1 address translation enabled.

In a system where the PE resets into EL2, this field resets to `0`.

Accessing the SCTLR_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic SCTLR_EL2 or SCTLR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, SCTLR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return SCTLR_EL2;
elsif PSTATE.EL == EL3 then
    return SCTLR_EL2;
```

MSR SCTLR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    SCTLR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    SCTLR_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, SCTLR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.SCTLR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x110];
    else
        return SCTLR_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return SCTLR_EL2;
    else
        return SCTLR_EL1;
elsif PSTATE.EL == EL3 then
    return SCTLR_EL1;
```

MSR SCTLR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.SCTLR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x110] = X[t];
    else
        SCTLR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        SCTLR_EL2 = X[t];
    else
        SCTLR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    SCTLR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.115 SCTLR_EL3, System Control Register (EL3)

The SCTLR_EL3 characteristics are:

Purpose

Provides top level control of the system, including its memory system, at EL3.

Configurations

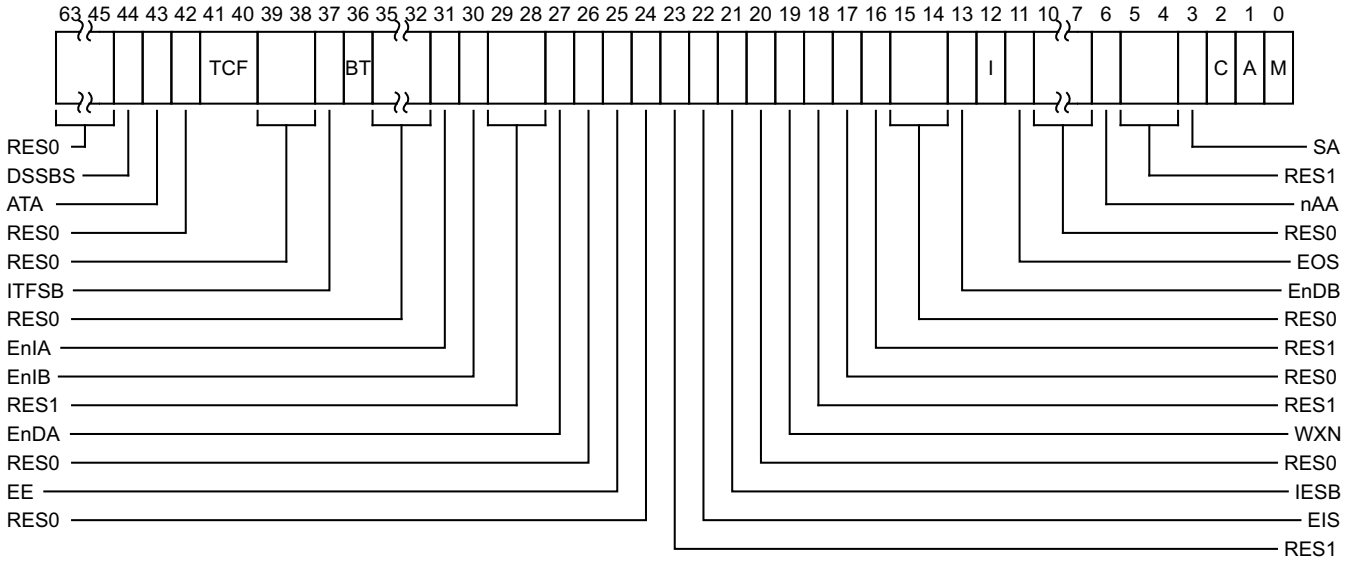
This register is present only when EL3 is implemented. Otherwise, direct accesses to SCTLR_EL3 are UNDEFINED.

Attributes

SCTLR_EL3 is a 64-bit register.

Field descriptions

The SCTLR_EL3 bit assignments are:



Bits [63:45]

Reserved, RES0.

DSSBS, bit [44]

When ARMv8.0-SSBS is implemented:

Default PSTATE.SSBS value on Exception Entry.

`0b0` PSTATE.SSBS is set to 0 on an exception to EL3.

`0b1` PSTATE.SSBS is set to 1 on an exception to EL3.

In a system where the PE resets into EL3, this field resets to an IMPLEMENTATION DEFINED value.

Otherwise:

Reserved, RES0.

ATA, bit [43]

When ARMv8.5-MemTag is implemented:

Allocation Tag Access in EL3. Controls EL3 access to Allocation Tags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When access to Allocation Tags is prevented:

- Instructions which Load or Store data are Unchecked.
- Instructions which Load or Store Allocation Tags treat the Allocation Tag as RAZ/WI.
- Instructions which insert Logical Address Tags into addresses treat the Allocation Tag used to generate the Logical Address Tag as 0.
- Cache maintenance instructions which invalidate Allocation Tags from caches behave as the equivalent Clean and Invalidate operation on Allocation Tags.

`0b0` Access to Allocation Tags is prevented.

`0b1` Access to Allocation Tags is not prevented.

This bit is permitted to be cached in a TLB.

In a system where the PE resets into EL3, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [42]

Reserved, RES0.

TCF, bits [41:40]

When ARMv8.5-MemTag is implemented:

Tag Check Fault in EL3. Controls the effect of Tag Check Faults due to Loads and Stores in EL3.

`0b00` Tag Check Faults have no effect on the PE.

`0b01` Tag Check Faults causes a synchronous exception.

`0b10` Tag Check Faults are asynchronously accumulated.

The value `0b11` is reserved.

In a system where the PE resets into EL3, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [39:38]

Reserved, RES0.

ITFSB, bit [37]

When ARMv8.5-MemTag is implemented:

When asynchronous exceptions are being generated by Tag Check Faults which are generated for Loads and Stores at any exception level, controls the auto-synchronization of Tag Check Faults into TFSRE0_EL1 and TFSR_ELx.

`0b0` Tag Check Faults are not synchronized on entry to EL3.

`0b1` Tag Check Faults are synchronized on entry to EL3.

In a system where the PE resets into EL3, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

BT, bit [36]

When ARMv8.5-BTI is implemented:

PAC Branch Type compatibility at EL3.

`0b0` When the PE is executing at EL3, PACIASP and PACIBSP are compatible with PSTATE.BTYPE == `0b11`.

`0b1` When the PE is executing at EL3, PACIASP and PACIBSP are not compatible with PSTATE.BTYPE == `0b11`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In a system where the PE resets into EL3, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [35:32]

Reserved, RES0.

EnIA, bit [31]

When ARMv8.3-PAuth is implemented:

Controls enabling of pointer authentication (using the APIAKey_EL1 key) of instruction addresses in the EL3 translation regime.

Possible values of this bit are:

`0b0` Pointer authentication (using the APIAKey_EL1 key) of instruction addresses is not enabled.

`0b1` Pointer authentication (using the APIAKey_EL1 key) of instruction addresses is enabled.

For more information, see *System register control of pointer authentication* on page D5-2539.

Note

This field controls the behavior of the AddPACIA and AuthIA pseudocode functions. Specifically, when the field is 1, AddPACIA returns a copy of a pointer to which a pointer authentication code has been added, and AuthIA returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

In a system where the PE resets into EL3, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnIB, bit [30]

When ARMv8.3-PAuth is implemented:

Controls enabling of pointer authentication (using the APIBKey_EL1 key) of instruction addresses in the EL3 translation regime.

Possible values of this bit are:

`0b0` Pointer authentication (using the APIBKey_EL1 key) of instruction addresses is not enabled.

`0b1` Pointer authentication (using the APIBKey_EL1 key) of instruction addresses is enabled.

For more information, see *System register control of pointer authentication* on page D5-2539.

Note

This field controls the behavior of the AddPACIB and AuthIB pseudocode functions. Specifically, when the field is 1, AddPACIB returns a copy of a pointer to which a pointer authentication code has been added, and AuthIB returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

In a system where the PE resets into EL3, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [29:28]

Reserved, RES1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

EnDA, bit [27]

When ARMv8.3-PAuth is implemented:

Controls enabling of pointer authentication (using the APDAKey_EL1 key) of instruction addresses in the EL3 translation regime.

`0b0` Pointer authentication (using the APDAKey_EL1 key) of data addresses is not enabled.

`0b1` Pointer authentication (using the APDAKey_EL1 key) of data addresses is enabled.

For more information, see *System register control of pointer authentication* on page D5-2539.

Note

This field controls the behavior of the AddPACDA and AuthDA pseudocode functions. Specifically, when the field is 1, AddPACDA returns a copy of a pointer to which a pointer authentication code has been added, and AuthDA returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

In a system where the PE resets into EL3, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [26]

Reserved, RES0.

EE, bit [25]

Endianness of data accesses at EL3, and stage 1 translation table walks in the EL3 translation regime.

`0b0` Explicit data accesses at EL3, and stage 1 translation table walks in the EL3 translation regime are little-endian.

`0b1` Explicit data accesses at EL3, and stage 1 translation table walks in the EL3 translation regime are big-endian.

If an implementation does not provide Big-endian support at Exception Levels higher than EL0, this bit is RES0.

If an implementation does not provide Little-endian support at Exception Levels higher than EL0, this bit is RES1.

The EE bit is permitted to be cached in a TLB.

In a system where the PE resets into EL3, this field resets to an IMPLEMENTATION DEFINED value.

Bit [24]

Reserved, RES0.

Bit [23]

Reserved, RES1.

EIS, bit [22]

When ARMv8.5-CSEH is implemented:

Exception Entry is Context Synchronizing.

`0b0` The taking of an exception to EL3 is not a context synchronizing event.

`0b1` The taking of an exception to EL3 is a context synchronizing event.

If SCTLR_EL3.EIS is set to `0b0`:

- Indirect writes to ESR_EL3, FAR_EL3, SPSR_EL3, ELR_EL3 are synchronized on exception entry to EL3, so that a direct read of the register after exception entry sees the indirectly written value caused by the exception entry.
- Memory transactions, including instruction fetches, from an Exception level always use the translation resources associated with that translation regime.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Exception Catch debug events are synchronous debug events.
- DCPS* and DRPS instructions are context synchronization events.

The following are not affected by the value of SCTLR_EL3.EIS:

- Changes to the PSTATE information on entry to EL3.
- Behavior of accessing the banked copies of the stack pointer using the SP register name for loads, stores and data processing instructions.
- Debug state exit.

In a system where the PE resets into EL3, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

IESB, bit [21]

When ARMv8.2-IESB is implemented:

Implicit Error Synchronization event enable.

`0b0` Disabled.

`0b1` An implicit error synchronization event is added:

- At each exception taken to EL3.
- Before the operational pseudocode of each `ERET` instruction executed at EL3.

When the PE is in Debug state, the effect of this field is CONSTRAINED UNPREDICTABLE, and its Effective value might be 0 or 1 regardless of the value of the field. If the Effective value of the field is 1, then an implicit error synchronization event is added after each `DCPSx` instruction taken to EL3 and before each `DRPS` instruction executed at EL3, in addition to the other cases where it is added.

When ARMv8.4-DFE is implemented, and the Effective value of SCR_EL3.NMEA is 1, this field is ignored and its Effective value is 1.

In a system where the PE resets into EL3, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [20]

Reserved, RES0.

WXN, bit [19]

Write permission implies XN (Execute-never). For the EL3 translation regime, this bit can force all memory regions that are writable to be treated as XN. The possible values of this bit are:

`0b0` This control has no effect on memory access permissions.

`0b1` Any region that is writable in the EL3 translation regime is forced to XN for accesses from software executing at EL3.

This bit applies only when SCTLR_EL3.M bit is set.

The WXN bit is permitted to be cached in a TLB.

In a system where the PE resets into EL3, this field resets to an architecturally UNKNOWN value.

Bit [18]

Reserved, RES1.

Bit [17]

Reserved, RES0.

Bit [16]

Reserved, RES1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bits [15:14]

Reserved, RES0.

EnDB, bit [13]

When ARMv8.3-PAuth is implemented:

Controls enabling of pointer authentication (using the APDBKey_EL1 key) of instruction addresses in the EL3 translation regime.

`0b0` Pointer authentication (using the APDBKey_EL1 key) of data addresses is not enabled.

`0b1` Pointer authentication (using the APDBKey_EL1 key) of data addresses is enabled.

For more information, see *System register control of pointer authentication* on page D5-2539.

Note

This field controls the behavior of the AddPACDB and AuthDB pseudocode functions. Specifically, when the field is 1, AddPACDB returns a copy of a pointer to which a pointer authentication code has been added, and AuthDB returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

In a system where the PE resets into EL3, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

I, bit [12]

Instruction access Cacheability control, for accesses at EL3:

`0b0` All instruction access to Normal memory from EL3 are Non-cacheable for all levels of instruction and unified cache.

If the value of SCTLR_EL3.M is 0, instruction accesses from stage 1 of the EL3 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.

`0b1` This control has no effect on the Cacheability of instruction access to Normal memory from EL3.

If the value of SCTLR_EL3.M is 0, instruction accesses from stage 1 of the EL3 translation regime are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.

This bit has no effect on the EL1&0, EL2, or EL2&0 translation regimes.

In a system where the PE resets into EL3, this field resets to `0`.

EOS, bit [11]

When ARMv8.5-CSEH is implemented:

Exception Exit is Context Synchronizing.

`0b0` An exception return from EL3 is not a context synchronizing event

`0b1` An exception return from EL3 is a context synchronizing event

If SCTLR_EL3.EOS is set to `0b0`:

- Memory transactions, including instruction fetches, from an Exception level always use the translation resources associated with that translation regime.
- Exception Catch debug events are synchronous debug events.
- DCPS* and DRPS instructions are context synchronization events.

The following are not affected by the value of SCTLR_EL3.EOS:

- The indirect write of the PSTATE and PC values from SPSR_EL3 and ELR_EL3 on exception return is synchronized.
- If the PE enters Debug state before the first instruction after an Exception return from EL3 to Non-secure state, any pending Halting debug event completes execution.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The GIC behavior that allocates interrupts to FIQ or IRQ changes simultaneously with leaving the EL3 Exception level.
- Behavior of accessing the banked copies of the stack pointer using the SP register name for loads, stores and data processing instructions.
- Exit from Debug state.

In a system where the PE resets into EL3, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

Bits [10:7]

Reserved, RES0.

nAA, bit [6]

When ARMv8.4-LSE is implemented:

Non-aligned access. This bit controls generation of Alignment faults at EL3 under certain conditions.

`0b0` LDAPR, LDAPRH, LDAPUR, LDAPURH, LDAPURSH, LDAPURSW, LDAR, LDARH, LDLAR, LDLARH, STLLR, STLLRH, STLR, STLRH, STLUR, and STLURH generate an Alignment fault if all bytes being accessed are not within a single 16-byte quantity, aligned to 16 bytes for accesses.

`0b1` This control bit does not cause LDAPR, LDAPRH, LDAPUR, LDAPURH, LDAPURSH, LDAPURSW, LDAR, LDARH, LDLAR, LDLARH, STLLR, STLLRH, STLR, STLRH, STLUR, or STLURH to generate an Alignment fault if all bytes being accessed are not within a single 16-byte quantity, aligned to 16 bytes.

In a system where the PE resets into EL3, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [5:4]

Reserved, RES1.

SA, bit [3]

SP Alignment check enable. When set to 1, if a load or store instruction executed at EL3 uses the SP as the base address and the SP is not aligned to a 16-byte boundary, then a SP alignment fault exception is generated. For more information, see *SP alignment checking* on page D1-2333.

In a system where the PE resets into EL3, this field resets to an architecturally UNKNOWN value.

C, bit [2]

Cacheability control, for data accesses.

`0b0` All data access to Normal memory from EL3, and all Normal memory accesses to the EL3 translation tables, are Non-cacheable for all levels of data and unified cache.

`0b1` This control has no effect on the Cacheability of:

- Data access to Normal memory from EL3.
- Normal memory accesses to the EL3 translation tables.

This bit has no effect on the EL1&0, EL2, or EL2&0 translation regimes.

In a system where the PE resets into EL3, this field resets to `0`.

A, bit [1]

Alignment check enable. This is the enable bit for Alignment fault checking at EL3.

`0b0` Alignment fault checking disabled when executing at EL3.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Instructions that load or store one or more registers, other than load/store exclusive and load-acquire/store-release, do not check that the address being accessed is aligned to the size of the data element(s) being accessed.

`0b1` Alignment fault checking enabled when executing at EL3.

All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element(s) being accessed. If this check fails it causes an Alignment fault, which is taken as a Data Abort exception.

Load/store exclusive and load-acquire/store-release instructions have an alignment check regardless of the value of the A bit.

In a system where the PE resets into EL3, this field resets to an architecturally UNKNOWN value.

M, bit [0]

MMU enable for EL3 stage 1 address translation. Possible values of this bit are:

`0b0` EL3 stage 1 address translation disabled.

See the SCTLR_EL3.I field for the behavior of instruction accesses to Normal memory.

`0b1` EL3 stage 1 address translation enabled.

In a system where the PE resets into EL3, this field resets to `0`.

Accessing the SCTLR_EL3

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, SCTLR_EL3

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    return SCTLR_EL3;
```

MSR SCTLR_EL3, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    SCTLR_EL3 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.116 SCXTNUM_EL0, EL0 Read/Write Software Context Number

The SCXTNUM_EL0 characteristics are:

Purpose

Provides a number that can be used to separate out different context numbers with the EL0 exception level, for the purpose of protecting against side-channels using branch prediction and similar resources.

Configurations

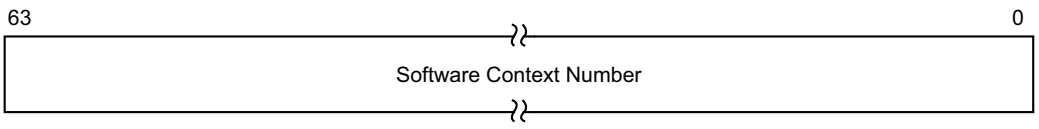
This register is present only when ARMv8.0-CSV2 is implemented. Otherwise, direct accesses to SCXTNUM_EL0 are UNDEFINED.

Attributes

SCXTNUM_EL0 is a 64-bit register.

Field descriptions

The SCXTNUM_EL0 bit assignments are:



Bits [63:0]

Software Context Number. A number to identify the context within the EL0 exception level.

This field resets to an architecturally UNKNOWN value.

Accessing the SCXTNUM_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, SCXTNUM_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b111

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.TSCXT == '1'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.EnSCXT == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HFGRTR_EL2.SCXTNUM_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.TSCXT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.EnSCXT == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return SCXTNUM_EL0;
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.EnSCXT == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.SCXTNUM_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.EnSCXT == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return SCXTNUM_EL0;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.EnSCXT == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return SCXTNUM_EL0;
elsif PSTATE.EL == EL3 then
    return SCXTNUM_EL0;
```

MSR SCXTNUM_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b111

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.TSCXT == '1'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.EnSCXT == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HFGWTR_EL2.SCXTNUM_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.TSCXT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.EnSCXT == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        SCXTNUM_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.EnSCXT == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.SCXTNUM_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.EnSCXT == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        SCXTNUM_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.EnSCXT == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        SCXTNUM_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    SCXTNUM_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.117 SCXTNUM_EL1, EL1 Read/Write Software Context Number

The SCXTNUM_EL1 characteristics are:

Purpose

Provides a number that can be used to separate out different context numbers with the EL1 exception level, for the purpose of protecting against side-channels using branch prediction and similar resources.

Configurations

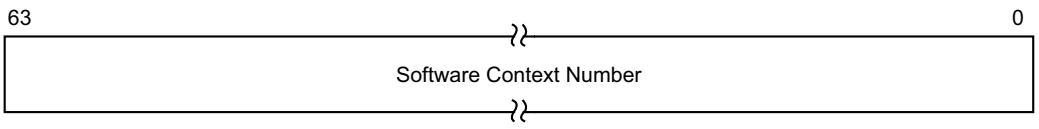
This register is present only when ARMv8.0-CSV2 is implemented. Otherwise, direct accesses to SCXTNUM_EL1 are UNDEFINED.

Attributes

SCXTNUM_EL1 is a 64-bit register.

Field descriptions

The SCXTNUM_EL1 bit assignments are:



Bits [63:0]

Software Context Number. A number to identify the context within the EL1 exception level.

This field resets to an architecturally UNKNOWN value.

Accessing the SCXTNUM_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, SCXTNUM_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.EnSCXT == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.SCXTNUM_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.EnSCXT == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x188];
    else
        return SCXTNUM_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.EnSCXT == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        return SCXTNUM_EL2;
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        return SCXTNUM_EL1;
elsif PSTATE.EL == EL3 then
    return SCXTNUM_EL1;
```

MSR SCXTNUM_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.EnSCXT == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.SCXTNUM_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.EnSCXT == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x188] = X[t];
    else
        SCXTNUM_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.EnSCXT == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        SCXTNUM_EL2 = X[t];
    else
        SCXTNUM_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    SCXTNUM_EL1 = X[t];
```

MRS <Xt>, SCXTNUM_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b1101	0b0000	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        return NVMem[0x188];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.EnSCXT == '0' then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            return SCXTNUM_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        return SCXTNUM_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        UNDEFINED;
```

MSR SCXTNUM_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b1101	0b0000	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        NVMem[0x188] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.EnSCXT == '0' then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            SCXTNUM_EL1 = X[t];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        SCXTNUM_EL1 = X[t];
    else
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.118 SCXTNUM_EL2, EL2 Read/Write Software Context Number

The SCXTNUM_EL2 characteristics are:

Purpose

Provides a number that can be used to separate out different context numbers with the EL2 exception level, for the purpose of protecting against side-channels using branch prediction and similar resources.

Configurations

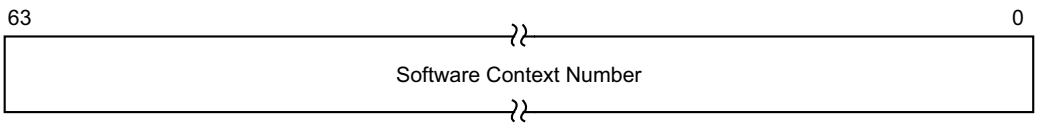
This register is present only when ARMv8.0-CSV2 is implemented. Otherwise, direct accesses to SCXTNUM_EL2 are UNDEFINED.

Attributes

SCXTNUM_EL2 is a 64-bit register.

Field descriptions

The SCXTNUM_EL2 bit assignments are:



Bits [63:0]

Software Context Number. A number to identify the context within the EL2 exception level.

This field resets to an architecturally UNKNOWN value.

Accessing the SCXTNUM_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic SCXTNUM_EL2 or SCXTNUM_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, SCXTNUM_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1101	0b0000	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.EnSCXT == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return SCXTNUM_EL2;
elsif PSTATE.EL == EL3 then
    return SCXTNUM_EL2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR SCXTNUM_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1101	0b0000	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.EnSCXT == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        SCXTNUM_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    SCXTNUM_EL2 = X[t];
```

MRS <Xt>, SCXTNUM_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.EnSCXT == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.SCXTNUM_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.EnSCXT == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x188];
    else
        return SCXTNUM_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.EnSCXT == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        return SCXTNUM_EL2;
    else
        return SCXTNUM_EL1;
elsif PSTATE.EL == EL3 then
    return SCXTNUM_EL1;
```

MSR SCXTNUM_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.EnSCXT == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.SCXTNUM_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.EnSCXT == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x188] = X[t];
    else
        SCXTNUM_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.EnSCXT == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        SCXTNUM_EL2 = X[t];
    else
        SCXTNUM_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    SCXTNUM_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.119 SCXTNUM_EL3, EL3 Read/Write Software Context Number

The SCXTNUM_EL3 characteristics are:

Purpose

Provides a number that can be used to separate out different context numbers with the EL3 exception level, for the purpose of protecting against side-channels using branch prediction and similar resources.

Configurations

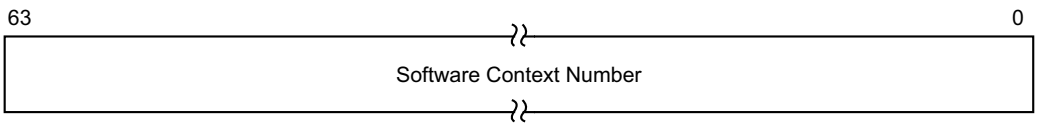
This register is present only when EL3 is implemented and ARMv8.0-CSV2 is implemented. Otherwise, direct accesses to SCXTNUM_EL3 are UNDEFINED.

Attributes

SCXTNUM_EL3 is a 64-bit register.

Field descriptions

The SCXTNUM_EL3 bit assignments are:



Bits [63:0]

Software Context Number. A number to identify the context within the EL3 exception level.

This field resets to an architecturally UNKNOWN value.

Accessing the SCXTNUM_EL3

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, SCXTNUM_EL3

op0	op1	CRn	CRm	op2
0b11	0b110	0b1101	0b0000	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    return SCXTNUM_EL3;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR SCXTNUM_EL3, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b110	0b1101	0b0000	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    SCXTNUM_EL3 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.120 TCR_EL1, Translation Control Register (EL1)

The TCR_EL1 characteristics are:

Purpose

The control register for stage 1 of the EL1&0 translation regime.

Configurations

AArch64 System register TCR_EL1[31:0] is architecturally mapped to AArch32 System register TTBCR[31:0].

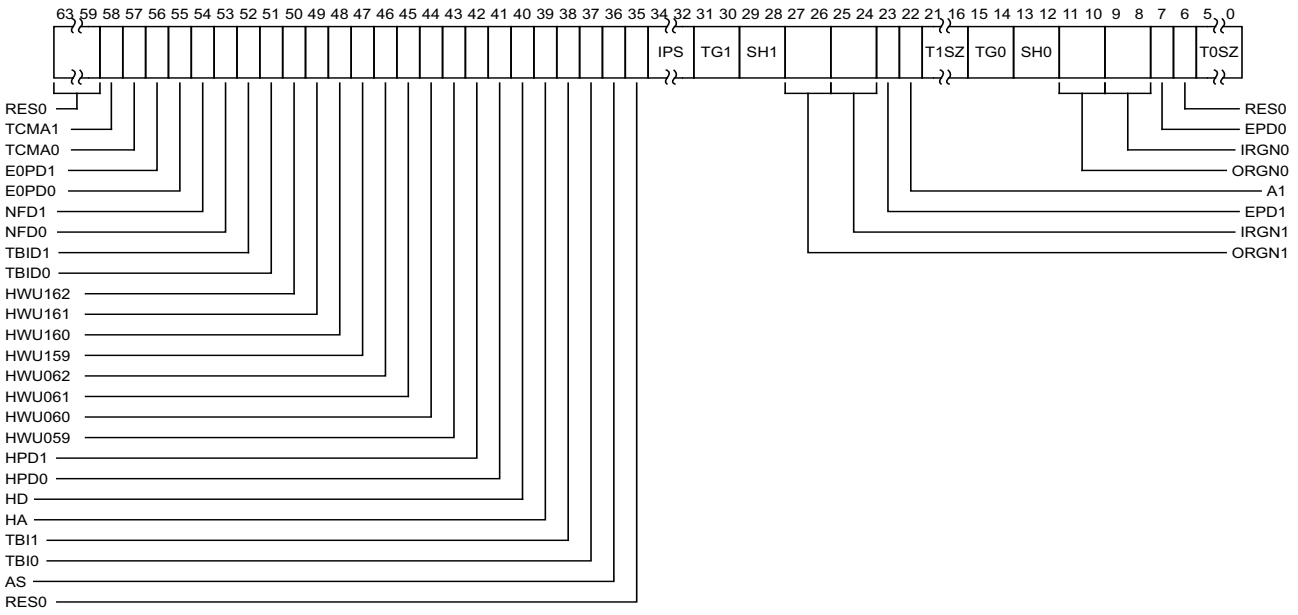
AArch64 System register TCR_EL1[63:32] is architecturally mapped to AArch32 System register TTBCR2[31:0].

Attributes

TCR_EL1 is a 64-bit register.

Field descriptions

The TCR_EL1 bit assignments are:



Any of the bits in TCR_EL1, other than the A1 bit and the EPDx bits when they have the value 1, are permitted to be cached in a TLB.

Bits [63:59]

Reserved, RES0.

TCMA1, bit [58]

When ARMv8.5-MemTag is implemented:

Controls the generation of Unchecked accesses at EL1, and at EL0 if HCR_EL2.{E2H,TGE}!={1,1}, when address[59:55] = `0b11111`.

`0b0` This control has no effect on the generation of Unchecked accesses at EL1 or EL0.

`0b1` All accesses at EL1 and EL0 are Unchecked.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

Software may change this control bit on a context switch.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TCMA0, bit [57]

When ARMv8.5-MemTag is implemented:

Controls the generation of Unchecked accesses at EL1, and at EL0 if HCR_EL2.{E2H,TGE}!={1,1}, when address[59:55] = `0b00000`.

`0b0` This control has no effect on the generation of Unchecked accesses at EL1 or EL0.

`0b1` All accesses at EL1 and EL0 are Unchecked.

Note

Software may change this control bit on a context switch.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

E0PD1, bit [56]

When ARMv8.5-E0PD is implemented:

Faulting control for EL0 access to any address translated by TTBR1_EL1.

`0b0` Unprivileged access to any address translated by TTBR1_EL1 will not generate a fault by this mechanism.

`0b1` Unprivileged access to any address translated by TTBR1_EL1 will generate a level 0 translation fault

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

E0PD0, bit [55]

When ARMv8.5-E0PD is implemented:

Faulting control for EL0 access to any address translated by TTBR0_EL1.

`0b0` Unprivileged access to any address translated by TTBR0_EL1 will not generate a fault by this mechanism.

`0b1` Unprivileged access to any address translated by TTBR0_EL1 will generate a level 0 translation fault

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NFD1, bit [54]

When SVE is implemented:

Non-fault translation table walk disable for stage 1 translations using TTBR1_EL1.

This bit controls whether to perform a stage 1 translation table walk in response to a non-fault access from EL0 for a virtual address that is translated using TTBR1_EL1.

If SVE is implemented, the affected access types include:

- All accesses due to an SVE non-fault contiguous load instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Accesses due to an SVE first-fault gather load instruction that are not for the First active element. Accesses due to an SVE first-fault contiguous load instruction are not affected.
- Accesses due to prefetch instructions might be affected, but the effect is not architecturally visible.

See SVE for more information.

Defined values are:

`0b0` Does not disable stage 1 translation table walks using TTBR1_EL1.

`0b1` A TLB miss on a virtual address that is translated using TTBR1_EL1 due to the specified access types causes the access to fail without taking an exception. No stage 1 translation table walk is performed.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NFD0, bit [53]

When SVE is implemented:

Non-fault translation table walk disable for stage 1 translations using TTBR0_EL1.

This bit controls whether to perform a stage 1 translation table walk in response to a non-fault access from EL0 for a virtual address that is translated using TTBR0_EL1.

If SVE is implemented, the affected access types include:

- All accesses due to an SVE non-fault contiguous load instruction.
- Accesses due to an SVE first-fault gather load instruction that are not for the First active element. Accesses due to an SVE first-fault contiguous load instruction are not affected.
- Accesses due to prefetch instructions might be affected, but the effect is not architecturally visible.

See SVE for more information.

Defined values are:

`0b0` Does not disable stage 1 translation table walks using TTBR0_EL1.

`0b1` A TLB miss on a virtual address that is translated using TTBR0_EL1 due to the specified access types causes the access to fail without taking an exception. No stage 1 translation table walk is performed.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TBID1, bit [52]

When ARMv8.3-PAuth is implemented:

Controls the use of the top byte of instruction addresses for address matching.

For the purpose of this field, all cache maintenance and address translation instructions that perform address translation are treated as data accesses.

For more information, see *Address tagging in AArch64 state* on page D5-2534.

`0b0` TCR_EL1.TBI1 applies to Instruction and Data accesses.

`0b1` TCR_EL1.TBI1 applies to Data accesses only.

This affects addresses where the address would be translated by tables pointed to by TTBR1_EL1.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TBID0, bit [51]

When ARMv8.3-PAuth is implemented:

Controls the use of the top byte of instruction addresses for address matching.

For the purpose of this field, all cache maintenance and address translation instructions that perform address translation are treated as data accesses.

For more information, see *Address tagging in AArch64 state* on page D5-2534.

`0b0` TCR_EL1.TBI0 applies to Instruction and Data accesses.

`0b1` TCR_EL1.TBI0 applies to Data accesses only.

This affects addresses where the address would be translated by tables pointed to by TTBR0_EL1.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU162, bit [50]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 1 translation table Block or Page entry for translations using TTBR1_EL1.

`0b0` For translations using TTBR1_EL1, bit[62] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` For translations using TTBR1_EL1, bit[62] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD1 is 1.

The Effective value of this field is 0 if the value of TCR_EL1.HPD1 is 0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU161, bit [49]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 1 translation table Block or Page entry for translations using TTBR1_EL1.

`0b0` For translations using TTBR1_EL1, bit[61] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` For translations using TTBR1_EL1, bit[61] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD1 is 1.

The Effective value of this field is 0 if the value of TCR_EL1.HPD1 is 0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU160, bit [48]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 1 translation table Block or Page entry for translations using TTBR1_EL1.

`0b0` For translations using TTBR1_EL1, bit[60] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` For translations using TTBR1_EL1, bit[60] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD1 is 1.

The Effective value of this field is 0 if the value of TCR_EL1.HPD1 is 0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU159, bit [47]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 1 translation table Block or Page entry for translations using TTBR1_EL1.

`0b0` For translations using TTBR1_EL1, bit[59] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` For translations using TTBR1_EL1, bit[59] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD1 is 1.

The Effective value of this field is 0 if the value of TCR_EL1.HPD1 is 0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU062, bit [46]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 1 translation table Block or Page entry for translations using TTBR0_EL1.

`0b0` For translations using TTBR0_EL1, bit[62] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` For translations using TTBR0_EL1, bit[62] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD0 is 1.

The Effective value of this field is 0 if the value of TCR_EL1.HPD0 is 0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU061, bit [45]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 1 translation table Block or Page entry for translations using TTBR0_EL1.

`0b0` For translations using TTBR0_EL1, bit[61] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` For translations using TTBR0_EL1, bit[61] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD0 is 1.

The Effective value of this field is 0 if the value of TCR_EL1.HPD0 is 0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

HWU060, bit [44]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 1 translation table Block or Page entry for translations using TTBR0_EL1.

`0b0` For translations using TTBR0_EL1, bit[60] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` For translations using TTBR0_EL1, bit[60] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD0 is 1.

The Effective value of this field is 0 if the value of TCR_EL1.HPD0 is 0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU059, bit [43]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 1 translation table Block or Page entry for translations using TTBR0_EL1.

`0b0` For translations using TTBR0_EL1, bit[59] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` For translations using TTBR0_EL1, bit[59] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD0 is 1.

The Effective value of this field is 0 if the value of TCR_EL1.HPD0 is 0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HPD1, bit [42]

When ARMv8.1-HPD is implemented:

Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, PXNTable, and UXNTable, except NSTable, in the translation tables pointed to by TTBR1_EL1.

`0b0` Hierarchical permissions are enabled.

`0b1` Hierarchical permissions are disabled.

When disabled, the permissions are treated as if the bits are zero.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HPD0, bit [41]

When ARMv8.1-HPD is implemented:

Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, PXNTable, and UXNTable, except NSTable, in the translation tables pointed to by TTBR0_EL1.

`0b0` Hierarchical permissions are enabled.

`0b1` Hierarchical permissions are disabled.

When disabled, the permissions are treated as if the bits are zero.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

HD, bit [40]

When ARMv8.1-TTHM is implemented:

Hardware management of dirty state in stage 1 translations from EL0 and EL1.

`0b0` Stage 1 hardware management of dirty state disabled.

`0b1` Stage 1 hardware management of dirty state enabled, only if the HA bit is also set to 1.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HA, bit [39]

When ARMv8.1-TTHM is implemented:

Hardware Access flag update in stage 1 translations from EL0 and EL1.

`0b0` Stage 1 Access flag update disabled.

`0b1` Stage 1 Access flag update enabled.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TBI1, bit [38]

Top Byte ignored - indicates whether the top byte of an address is used for address match for the TTBR1_EL1 region, or ignored and used for tagged addresses. Defined values are:

`0b0` Top Byte used in the address calculation.

`0b1` Top Byte ignored in the address calculation.

This affects addresses generated in EL0 and EL1 using AArch64 where the address would be translated by tables pointed to by TTBR1_EL1. It has an effect whether the EL1&0 translation regime is enabled or not.

If ARMv8.3-PAuth is implemented and TCR_EL1.TBID1 is 1, then this field only applies to Data accesses.

Otherwise, if the value of TBI1 is 1 and bit [55] of the target address to be stored to the PC is 1, then bits[63:56] of that target address are also set to 1 before the address is stored in the PC, in the following cases:

- A branch or procedure return within EL0 or EL1.
- An exception taken to EL1.
- An exception return to EL0 or EL1.

This field resets to an architecturally UNKNOWN value.

TBI0, bit [37]

Top Byte ignored - indicates whether the top byte of an address is used for address match for the TTBR0_EL1 region, or ignored and used for tagged addresses. Defined values are:

`0b0` Top Byte used in the address calculation.

`0b1` Top Byte ignored in the address calculation.

This affects addresses generated in EL0 and EL1 using AArch64 where the address would be translated by tables pointed to by TTBR0_EL1. It has an effect whether the EL1&0 translation regime is enabled or not.

If ARMv8.3-PAuth is implemented and TCR_EL1.TBID0 is 1, then this field only applies to Data accesses.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise, if the value of TBI0 is 1 and bit [55] of the target address to be stored to the PC is 0, then bits[63:56] of that target address are also set to 0 before the address is stored in the PC, in the following cases:

- A branch or procedure return within EL0 or EL1.
- An exception taken to EL1.
- An exception return to EL0 or EL1.

This field resets to an architecturally UNKNOWN value.

AS, bit [36]

ASID Size. Defined values are:

`0b0` 8 bit - the upper 8 bits of TTBR0_EL1 and TTBR1_EL1 are ignored by hardware for every purpose except reading back the register, and are treated as if they are all zeros for when used for allocation and matching entries in the TLB.

`0b1` 16 bit - the upper 16 bits of TTBR0_EL1 and TTBR1_EL1 are used for allocation and matching in the TLB.

If the implementation has only 8 bits of ASID, this field is RES0.

This field resets to an architecturally UNKNOWN value.

Bit [35]

Reserved, RES0.

IPS, bits [34:32]

Intermediate Physical Address Size.

`0b000` 32 bits, 4GB.

`0b001` 36 bits, 64GB.

`0b010` 40 bits, 1TB.

`0b011` 42 bits, 4TB.

`0b100` 44 bits, 16TB.

`0b101` 48 bits, 256TB.

`0b110` 52 bits, 4PB.

Other values are reserved.

The reserved values behave in the same way as the `0b101` or `0b110` encoding, but software must not rely on this property as the behavior of the reserved values might change in a future revision of the architecture.

The value `0b110` is permitted only if ARMv8.2-LPA is implemented and the translation granule size is 64KB.

In an implementation that supports 52-bit PAs, if the value of this field is not `0b110` or a value treated as `0b110`, then bits[51:48] of every translation table base address for the stage of translation controlled by TCR_EL1 are `0b0000`.

This field resets to an architecturally UNKNOWN value.

TG1, bits [31:30]

Granule size for the TTBR1_EL1.

`0b01` 16KB.

`0b10` 4KB.

`0b11` 64KB.

Other values are reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the value is programmed to either a reserved value, or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

This field resets to an architecturally UNKNOWN value.

SH1, bits [29:28]

Shareability attribute for memory associated with translation table walks using TTBR1_EL1.

`0b00` Non-shareable.

`0b10` Outer Shareable.

`0b11` Inner Shareable.

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-7871.

This field resets to an architecturally UNKNOWN value.

ORGN1, bits [27:26]

Outer cacheability attribute for memory associated with translation table walks using TTBR1_EL1.

`0b00` Normal memory, Outer Non-cacheable.

`0b01` Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.

`0b10` Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.

`0b11` Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.

This field resets to an architecturally UNKNOWN value.

IRGN1, bits [25:24]

Inner cacheability attribute for memory associated with translation table walks using TTBR1_EL1.

`0b00` Normal memory, Inner Non-cacheable.

`0b01` Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.

`0b10` Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.

`0b11` Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

This field resets to an architecturally UNKNOWN value.

EPD1, bit [23]

Translation table walk disable for translations using TTBR1_EL1. This bit controls whether a translation table walk is performed on a TLB miss, for an address that is translated using TTBR1_EL1. The encoding of this bit is:

`0b0` Perform translation table walks using TTBR1_EL1.

`0b1` A TLB miss on an address that is translated using TTBR1_EL1 generates a Translation fault. No translation table walk is performed.

This field resets to an architecturally UNKNOWN value.

A1, bit [22]

Selects whether TTBR0_EL1 or TTBR1_EL1 defines the ASID. The encoding of this bit is:

`0b0` TTBR0_EL1.ASID defines the ASID.

`0b1` TTBR1_EL1.ASID defines the ASID.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1SZ, bits [21:16]

The size offset of the memory region addressed by TTBR1_EL1. The region size is $2^{(64-T1SZ)}$ bytes.

The maximum and minimum possible values for T1SZ depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.

This field resets to an architecturally UNKNOWN value.

TG0, bits [15:14]

Granule size for the TTBR0_EL1.

`0b00` 4KB

`0b01` 64KB

`0b10` 16KB

Other values are reserved.

If the value is programmed to either a reserved value, or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

This field resets to an architecturally UNKNOWN value.

SH0, bits [13:12]

Shareability attribute for memory associated with translation table walks using TTBR0_EL1.

`0b00` Non-shareable

`0b10` Outer Shareable

`0b11` Inner Shareable

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-7871.

This field resets to an architecturally UNKNOWN value.

ORGN0, bits [11:10]

Outer cacheability attribute for memory associated with translation table walks using TTBR0_EL1.

`0b00` Normal memory, Outer Non-cacheable.

`0b01` Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.

`0b10` Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.

`0b11` Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.

This field resets to an architecturally UNKNOWN value.

IRGN0, bits [9:8]

Inner cacheability attribute for memory associated with translation table walks using TTBR0_EL1.

`0b00` Normal memory, Inner Non-cacheable.

`0b01` Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.

`0b10` Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.

`0b11` Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

EPD0, bit [7]

Translation table walk disable for translations using TTBR0_EL1. This bit controls whether a translation table walk is performed on a TLB miss, for an address that is translated using TTBR0_EL1. The encoding of this bit is:

`0b0` Perform translation table walks using TTBR0_EL1.

`0b1` A TLB miss on an address that is translated using TTBR0_EL1 generates a Translation fault. No translation table walk is performed.

This field resets to an architecturally UNKNOWN value.

Bit [6]

Reserved, RES0.

T0SZ, bits [5:0]

The size offset of the memory region addressed by TTBR0_EL1. The region size is $2^{(64-T0SZ)}$ bytes.

The maximum and minimum possible values for T0SZ depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.

This field resets to an architecturally UNKNOWN value.

Accessing the TCR_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic TCR_EL1 or TCR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, TCR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.TCR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x120];
    else
        return TCR_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return TCR_EL2;
    else
        return TCR_EL1;
elsif PSTATE.EL == EL3 then
    return TCR_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR TCR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.TCR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x120] = X[t];
    else
        TCR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        TCR_EL2 = X[t];
    else
        TCR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    TCR_EL1 = X[t];
```

MRS <Xt>, TCR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0010	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        return NVMem[0x120];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return TCR_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        return TCR_EL1;
    else
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR TCR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0010	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        NVMem[0x120] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        TCR_EL1 = X[t];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        TCR_EL1 = X[t];
    else
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.121 TCR_EL2, Translation Control Register (EL2)

The TCR_EL2 characteristics are:

Purpose

The control register for stage 1 of the EL2, or EL2&0, translation regime:

- When the Effective value of HCR_EL2.E2H is 0, this register controls stage 1 of the EL2 translation regime, that supports a single VA range, translated using TTBR0_EL2.
- When the value of HCR_EL2.E2H is 1, this register controls stage 1 of the EL2&0 translation regime, that supports both:
  - A lower VA range, translated using TTBR0_EL2.
  - A higher VA range, translated using TTBR1_EL2.

Configurations

AArch64 System register TCR_EL2[31:0] is architecturally mapped to AArch32 System register HTCR[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

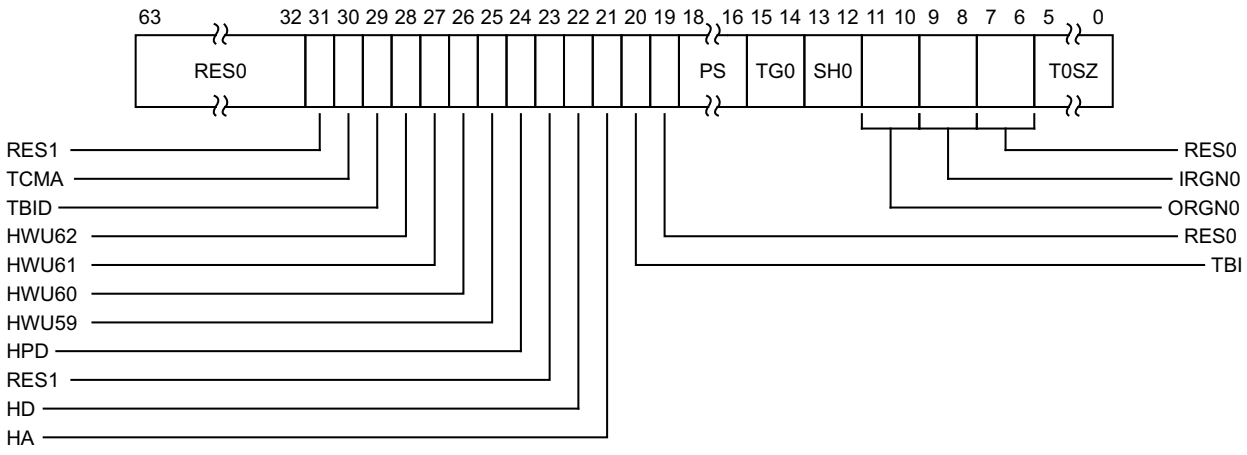
Attributes

TCR_EL2 is a 64-bit register.

Field descriptions

The TCR_EL2 bit assignments are:

When HCR_EL2.E2H == 0:



Any of the bits in TCR_EL2, other than the A1 bit and the EPDx bits when they have the value 1, are permitted to be cached in a TLB.

Bits [63:32]

Reserved, RES0.

Bit [31]

Reserved, RES1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TCMA, bit [30]

When ARMv8.5-MemTag is implemented:

Controls the generation of Unchecked accesses at EL2 when address [59:56] = `0b0000`.

`0b0` This control has no effect on the generation of Unchecked accesses.

`0b1` All accesses are Unchecked.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TBID, bit [29]

When ARMv8.3-PAuth is implemented:

Controls the use of the top byte of instruction addresses for address matching.

For the purpose of this field, all cache maintenance and address translation instructions that perform address translation are treated as data accesses.

For more information, see *Address tagging in AArch64 state* on page D5-2534.

`0b0` TCR_EL2.TBI applies to Instruction and Data accesses.

`0b1` TCR_EL2.TBI applies to Data accesses only.

This affects addresses where the address would be translated by tables pointed to by TTBR0_EL2.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU62, bit [28]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 1 translation table Block or Page entry.

`0b0` Bit[62] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` Bit[62] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL2.HPD is 1.

The Effective value of this field is 0 if the value of TCR_EL2.HPD is 0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU61, bit [27]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 1 translation table Block or Page entry.

`0b0` Bit[61] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` Bit[61] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL2.HPD is 1.

The Effective value of this field is 0 if the value of TCR_EL2.HPD is 0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

HWU60, bit [26]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 1 translation table Block or Page entry.

`0b0` Bit[60] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` Bit[60] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL2.HPD is 1.

The Effective value of this field is 0 if the value of TCR_EL2.HPD is 0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU59, bit [25]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 1 translation table Block or Page entry.

`0b0` Bit[59] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` Bit[59] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL2.HPD is 1.

The Effective value of this field is 0 if the value of TCR_EL2.HPD is 0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HPD, bit [24]

When ARMv8.1-HPD is implemented:

Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, PXNTable, and UXNTable, except NSTable, in the translation tables pointed to by TTBR0_EL2.

`0b0` Hierarchical permissions are enabled.

`0b1` Hierarchical permissions are disabled.

Note

In this case bit[61] (APTable[0]) and bit[59] (PXNTable) of the next level descriptor attributes are required to be ignored by the PE, and are no longer reserved, allowing them to be used by software.

When disabled, the permissions are treated as if the bits are zero.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [23]

Reserved, RES1.

HD, bit [22]

When ARMv8.1-TTHM is implemented:

Hardware management of dirty state in stage 1 translations from EL2.

`0b0` Stage 1 hardware management of dirty state disabled.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` Stage 1 hardware management of dirty state enabled, only if the HA bit is also set to 1.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HA, bit [21]

When ARMv8.1-TTHM is implemented:

Hardware Access flag update in stage 1 translations from EL2.

`0b0` Stage 1 Access flag update disabled.

`0b1` Stage 1 Access flag update enabled.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TBI, bit [20]

Top Byte Ignored. Indicates whether the top byte of an address is used for address match for the TTBR0_EL2 region, or ignored and used for tagged addresses.

For more information, see *Address tagging in AArch64 state* on page D5-2534.

`0b0` Top Byte used in the address calculation.

`0b1` Top Byte ignored in the address calculation.

This affects addresses generated in EL2 using AArch64 where the address would be translated by tables pointed to by TTBR0_EL2. It has an effect whether the EL2, or EL2&0, translation regime is enabled or not.

If ARMv8.3-PAuth is implemented and TCR_EL2.TBID is 1, then this field only applies to Data accesses.

If the value of TBI is 1, then bits[63:56] of that target address are also set to 0 before the address is stored in the PC, in the following cases:

- A branch or procedure return within EL2.
- An exception taken to EL2.
- An exception return to EL2.

This field resets to an architecturally UNKNOWN value.

Bit [19]

Reserved, RES0.

PS, bits [18:16]

Physical Address Size.

`0b000` 32 bits, 4GB.

`0b001` 36 bits, 64GB.

`0b010` 40 bits, 1TB.

`0b011` 42 bits, 4TB.

`0b100` 44 bits, 16TB.

`0b101` 48 bits, 256TB.

`0b110` 52 bits, 4PB.

Other values are reserved.

The reserved values behave in the same way as the `0b101` or `0b110` encoding, but software must not rely on this property as the behavior of the reserved values might change in a future revision of the architecture.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The value `0b110` is permitted only if ARMv8.2-LPA is implemented and the translation granule size is 64KB.

In an implementation that supports 52-bit PAs, if the value of this field is not `0b110` or a value treated as `0b110`, then bits[51:48] of every translation table base address for the stage of translation controlled by TCR_EL2 are `0b0000`.

This field resets to an architecturally UNKNOWN value.

TG0, bits [15:14]

Granule size for the TTBR0_EL2.

`0b00` 4KB.

`0b01` 64KB.

`0b10` 16KB.

Other values are reserved.

If the value is programmed to either a reserved value, or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

This field resets to an architecturally UNKNOWN value.

SH0, bits [13:12]

Shareability attribute for memory associated with translation table walks using TTBR0_EL2.

`0b00` Non-shareable.

`0b10` Outer Shareable.

`0b11` Inner Shareable.

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-7871.

This field resets to an architecturally UNKNOWN value.

ORGN0, bits [11:10]

Outer cacheability attribute for memory associated with translation table walks using TTBR0_EL2.

`0b00` Normal memory, Outer Non-cacheable.

`0b01` Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.

`0b10` Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.

`0b11` Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.

This field resets to an architecturally UNKNOWN value.

IRGN0, bits [9:8]

Inner cacheability attribute for memory associated with translation table walks using TTBR0_EL2.

`0b00` Normal memory, Inner Non-cacheable.

`0b01` Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.

`0b10` Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.

`0b11` Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

This field resets to an architecturally UNKNOWN value.

Bits [7:6]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

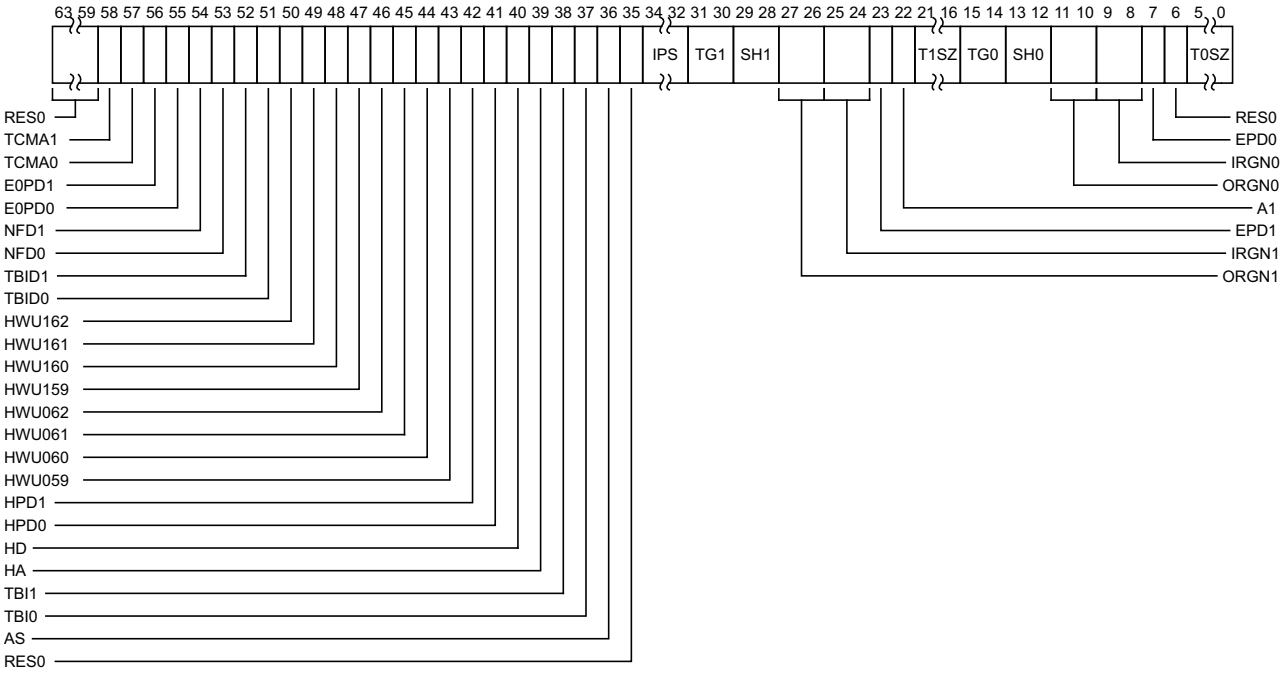
T0SZ, bits [5:0]

The size offset of the memory region addressed by TTBR0_EL2. The region size is $2^{(64-T0SZ)}$ bytes.

The maximum and minimum possible values for T0SZ depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.

This field resets to an architecturally UNKNOWN value.

When ARMv8.1-VHE is implemented and HCR_EL2.E2H == 1:



This view of the register is only valid from Armv8.1 when HCR_EL2.E2H is 1.

Any of the bits in TCR_EL2 are permitted to be cached in a TLB.

Bits [63:59]

Reserved, RES0.

TCMA1, bit [58]

When ARMv8.5-MemTag is implemented:

Controls the generation of Unchecked accesses at EL2, and at EL0 if HCR_EL2.TGE=1, when address[59:55] = `0b11111`.

`0b0` This control has no effect on the generation of Unchecked accesses at EL2 or EL0.

`0b1` All accesses are Unchecked.

Note

Software may change this control bit on a context switch.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TCMA0, bit [57]

When ARMv8.5-MemTag is implemented:

Controls the generation of Unchecked accesses at EL2, and at EL0 if HCR_EL2.TGE=1, when address[59:55] = `0b00000`.

`0b0` This control has no effect on the generation of Unchecked accesses at EL2 or EL0.

`0b1` All accesses are Unchecked.

Note

Software may change this control bit on a context switch.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

E0PD1, bit [56]

When ARMv8.5-E0PD is implemented:

Faulting control for EL0 access to any address translated by TTBR1_EL2.

`0b0` Unprivileged access to any address translated by TTBR1_EL2 will not generate a fault by this mechanism.

`0b1` Unprivileged access to any address translated by TTBR1_EL2 will generate a level 0 translation fault

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

E0PD0, bit [55]

When ARMv8.5-E0PD is implemented:

Faulting control for EL0 access to any address translated by TTBR0_EL2.

`0b0` Unprivileged access to any address translated by TTBR0_EL2 will not generate a fault by this mechanism.

`0b1` Unprivileged access to any address translated by TTBR0_EL2 will generate a level 0 translation fault

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NFD1, bit [54]

When SVE is implemented:

Non-fault translation table walk disable for stage 1 translations using TTBR1_EL2.

This bit controls whether to perform a stage 1 translation table walk in response to a non-fault access from EL0 for a virtual address that is translated using TTBR1_EL2.

If SVE is implemented, the affected access types include:

- All accesses due to an SVE non-fault contiguous load instruction.
- Accesses due to an SVE first-fault gather load instruction that are not for the First active element. Accesses due to an SVE first-fault contiguous load instruction are not affected.
- Accesses due to prefetch instructions might be affected, but the effect is not architecturally visible.

See SVE for more information.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Defined values are:

`0b0` Does not disable stage 1 translation table walks using TTBR1_EL2.

`0b1` A TLB miss on a virtual address that is translated using TTBR1_EL2 due to the specified access types causes the access to fail without taking an exception. No stage 1 translation table walk is performed.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NFD0, bit [53]

When SVE is implemented:

Non-fault translation table walk disable for stage 1 translations using TTBR0_EL2.

This bit controls whether to perform a stage 1 translation table walk in response to a non-fault access from EL0 for a virtual address that is translated using TTBR0_EL2.

If SVE is implemented, the affected access types include:

- All accesses due to an SVE non-fault contiguous load instruction.
- Accesses due to an SVE first-fault gather load instruction that are not for the First active element. Accesses due to an SVE first-fault contiguous load instruction are not affected.
- Accesses due to prefetch instructions might be affected, but the effect is not architecturally visible.

See SVE for more information.

Defined values are:

`0b0` Does not disable stage 1 translation table walks using TTBR0_EL2.

`0b1` A TLB miss on a virtual address that is translated using TTBR0_EL2 due to the specified access types causes the access to fail without taking an exception. No stage 1 translation table walk is performed.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TBID1, bit [52]

When ARMv8.3-PAuth is implemented:

Controls the use of the top byte of instruction addresses for address matching.

For the purpose of this field, all cache maintenance and address translation instructions that perform address translation are treated as data accesses.

For more information, see *Address tagging in AArch64 state* on page D5-2534.

`0b0` TCR_EL2.TBI1 applies to Instruction and Data accesses.

`0b1` TCR_EL2.TBI1 applies to Data accesses only.

This affects addresses where the address would be translated by tables pointed to by TTBR1_EL2.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TBID0, bit [51]

When ARMv8.3-PAuth is implemented:

Controls the use of the top byte of instruction addresses for address matching.

For more information, see *Address tagging in AArch64 state* on page D5-2534.

`0b0` TCR_EL2.TBI0 applies to Instruction and Data accesses.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` TCR_EL2.TBI0 applies to Data accesses only.

This affects addresses where the address would be translated by tables pointed to by TTBR0_EL2.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU162, bit [50]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 1 translation table Block or Page entry for translations using TTBR1_EL2.

`0b0` For translations using TTBR1_EL2, bit[62] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` For translations using TTBR1_EL2, bit[62] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL2.HPD1 is 1.

The Effective value of this field is 0 if the value of TCR_EL2.HPD1 is 0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU161, bit [49]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 1 translation table Block or Page entry for translations using TTBR1_EL2.

`0b0` For translations using TTBR1_EL2, bit[61] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` For translations using TTBR1_EL2, bit[61] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL2.HPD1 is 1.

The Effective value of this field is 0 if the value of TCR_EL2.HPD1 is 0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU160, bit [48]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 1 translation table Block or Page entry for translations using TTBR1_EL2.

`0b0` For translations using TTBR1_EL2, bit[60] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` For translations using TTBR1_EL2, bit[60] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL2.HPD1 is 1.

The Effective value of this field is 0 if the value of TCR_EL2.HPD1 is 0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

HWU159, bit [47]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 1 translation table Block or Page entry for translations using TTBR1_EL2.

`0b0` For translations using TTBR1_EL2, bit[59] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` For translations using TTBR1_EL2, bit[59] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL2.HPD1 is 1.

The Effective value of this field is 0 if the value of TCR_EL2.HPD1 is 0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU062, bit [46]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 1 translation table Block or Page entry for translations using TTBR0_EL1.

`0b0` For translations using TTBR0_EL1, bit[62] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` For translations using TTBR0_EL1, bit[62] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL2.HPD0 is 1.

The Effective value of this field is 0 if the value of TCR_EL2.HPD0 is 0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU061, bit [45]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 1 translation table Block or Page entry for translations using TTBR0_EL1.

`0b0` For translations using TTBR0_EL1, bit[61] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` For translations using TTBR0_EL1, bit[61] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL2.HPD0 is 1.

The Effective value of this field is 0 if the value of TCR_EL2.HPD0 is 0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU060, bit [44]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 1 translation table Block or Page entry for translations using TTBR0_EL1.

`0b0` For translations using TTBR0_EL1, bit[60] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` For translations using TTBR0_EL1, bit[60] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL2.HPD0 is 1.

The Effective value of this field is 0 if the value of TCR_EL2.HPD0 is 0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU059, bit [43]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 1 translation table Block or Page entry for translations using TTBR0_EL1.

`0b0` For translations using TTBR0_EL1, bit[59] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` For translations using TTBR0_EL1, bit[59] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL2.HPD0 is 1.

The Effective value of this field is 0 if the value of TCR_EL2.HPD0 is 0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HPD1, bit [42]

When ARMv8.1-HPD is implemented:

Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, PXNTable, and UXNTable, except NSTable, in the translation tables pointed to by TTBR1_EL2.

`0b0` Hierarchical permissions are enabled.

`0b1` Hierarchical permissions are disabled.

When disabled, the permissions are treated as if the bits are zero.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HPD0, bit [41]

When ARMv8.1-HPD is implemented:

Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, PXNTable, and UXNTable, except NSTable, in the translation tables pointed to by TTBR0_EL2.

`0b0` Hierarchical permissions are enabled.

`0b1` Hierarchical permissions are disabled.

When disabled, the permissions are treated as if the bits are zero.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HD, bit [40]

When ARMv8.1-TTHM is implemented:

Hardware management of dirty state in stage 1 translations from EL2.

`0b0` Stage 1 hardware management of dirty state disabled.

`0b1` Stage 1 hardware management of dirty state enabled, only if the HA bit is also set to 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HA, bit [39]

When ARMv8.1-TTHM is implemented:

Hardware Access flag update in stage 1 translations from EL2.

`0b0` Stage 1 Access flag update disabled.

`0b1` Stage 1 Access flag update enabled.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TBI1, bit [38]

Top Byte Ignored. Indicates whether the top byte of an address is used for address match for the TTBR1_EL2 region, or ignored and used for tagged addresses.

For more information, see *Address tagging in AArch64 state* on page D5-2534.

`0b0` Top Byte used in the address calculation.

`0b1` Top Byte ignored in the address calculation.

This affects addresses generated in EL0 and EL2 using AArch64 where the address would be translated by tables pointed to by TTBR1_EL2. It has an effect whether the EL2, or EL2&0, translation regime is enabled or not.

If ARMv8.3-PAuth is implemented and TCR_EL2.TBID1 is 1, then this field only applies to Data accesses.

If the value of TBI1 is 1 and bit [55] of the target address to be stored to the PC is 1, then bits[63:56] of that target address are also set to 1 before the address is stored in the PC, in the following cases:

- A branch or procedure return within EL0 or EL1.
- An exception taken to EL1.
- An exception return to EL0 or EL1.

This field resets to an architecturally UNKNOWN value.

TBI0, bit [37]

Top Byte Ignored. Indicates whether the top byte of an address is used for address match for the TTBR0_EL2 region, or ignored and used for tagged addresses.

For more information, see *Address tagging in AArch64 state* on page D5-2534.

`0b0` Top Byte used in the address calculation.

`0b1` Top Byte ignored in the address calculation.

This affects addresses generated in EL0 and EL2 using AArch64 where the address would be translated by tables pointed to by TTBR0_EL2. It has an effect whether the EL2, or EL2&0, translation regime is enabled or not.

If ARMv8.3-PAuth is implemented and TCR_EL2.TBID0 is 1, then this field only applies to Data accesses.

If the value of TBI0 is 1 and bit [55] of the target address to be stored to the PC is 0, then bits[63:56] of that target address are also set to 0 before the address is stored in the PC, in the following cases:

- A branch or procedure return within EL0 or EL1.
- An exception taken to EL1.
- An exception return to EL0 or EL1.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

AS, bit [36]

ASID Size. Defined values are:

`0b0` 8 bit - the upper 8 bits of TTBR0_EL2 and TTBR1_EL2 are ignored by hardware for every purpose except reading back the register, and are treated as if they are all zeros for when used for allocation and matching entries in the TLB.

`0b1` 16 bit - the upper 16 bits of TTBR0_EL2 and TTBR1_EL2 are used for allocation and matching in the TLB.

If the implementation has only 8 bits of ASID, this field is RES0.

This field resets to an architecturally UNKNOWN value.

Bit [35]

Reserved, RES0.

IPS, bits [34:32]

Intermediate Physical Address Size.

`0b000` 32 bits, 4GB.

`0b001` 36 bits, 64GB.

`0b010` 40 bits, 1TB.

`0b011` 42 bits, 4TB.

`0b100` 44 bits, 16TB.

`0b101` 48 bits, 256TB.

`0b110` **When ARMv8.2-LPA is implemented** 52 bits, 4PB.

Other values are reserved.

The reserved values behave in the same way as the `0b101` or `0b110` encoding, but software must not rely on this property as the behavior of the reserved values might change in a future revision of the architecture.

The value `0b110` is permitted only if ARMv8.2-LPA is implemented and the translation granule size is 64KB.

In an implementation that supports 52-bit PAs, if the value of this field is not `0b110` or a value treated as `0b110`, then bits[51:48] of every translation table base address for the stage of translation controlled by TCR_EL2 are `0b0000`.

This field resets to an architecturally UNKNOWN value.

TG1, bits [31:30]

Granule size for the TTBR1_EL2.

`0b01` 16KB.

`0b10` 4KB.

`0b11` 64KB.

Other values are reserved.

If the value is programmed to either a reserved value, or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

SH1, bits [29:28]

Shareability attribute for memory associated with translation table walks using TTBR1_EL2. Defined values are:

`0b00` Non-shareable.

`0b10` Outer Shareable.

`0b11` Inner Shareable.

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-7871.

This field resets to an architecturally UNKNOWN value.

ORGN1, bits [27:26]

Outer cacheability attribute for memory associated with translation table walks using TTBR1_EL2.

`0b00` Normal memory, Outer Non-cacheable.

`0b01` Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.

`0b10` Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.

`0b11` Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.

This field resets to an architecturally UNKNOWN value.

IRGN1, bits [25:24]

Inner cacheability attribute for memory associated with translation table walks using TTBR1_EL2.

`0b00` Normal memory, Inner Non-cacheable.

`0b01` Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.

`0b10` Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.

`0b11` Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

This field resets to an architecturally UNKNOWN value.

EPD1, bit [23]

Translation table walk disable for translations using TTBR1_EL2. This bit controls whether a translation table walk is performed on a TLB miss, for an address that is translated using TTBR1_EL2. The encoding of this bit is:

`0b0` Perform translation table walks using TTBR1_EL2.

`0b1` A TLB miss on an address that is translated using TTBR1_EL2 generates a Translation fault. No translation table walk is performed.

This field resets to an architecturally UNKNOWN value.

A1, bit [22]

Selects whether TTBR0_EL2 or TTBR1_EL2 defines the ASID. The encoding of this bit is:

`0b0` TTBR0_EL2.ASID defines the ASID.

`0b1` TTBR1_EL2.ASID defines the ASID.

This field resets to an architecturally UNKNOWN value.

T1SZ, bits [21:16]

The size offset of the memory region addressed by TTBR1_EL2. The region size is $2^{(64-T1SZ)}$ bytes.

The maximum and minimum possible values for T1SZ depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TG0, bits [15:14]

Granule size for the TTBR0_EL2.

`0b00` 4KB.

`0b01` 64KB.

`0b10` 16KB.

Other values are reserved.

If the value is programmed to either a reserved value, or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

This field resets to an architecturally UNKNOWN value.

SH0, bits [13:12]

Shareability attribute for memory associated with translation table walks using TTBR0_EL2.

`0b00` Non-shareable.

`0b10` Outer Shareable.

`0b11` Inner Shareable.

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-7871.

This field resets to an architecturally UNKNOWN value.

ORGN0, bits [11:10]

Outer cacheability attribute for memory associated with translation table walks using TTBR0_EL2.

`0b00` Normal memory, Outer Non-cacheable.

`0b01` Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.

`0b10` Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.

`0b11` Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.

This field resets to an architecturally UNKNOWN value.

IRGN0, bits [9:8]

Inner cacheability attribute for memory associated with translation table walks using TTBR0_EL2.

`0b00` Normal memory, Inner Non-cacheable.

`0b01` Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.

`0b10` Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.

`0b11` Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

This field resets to an architecturally UNKNOWN value.

EPD0, bit [7]

Translation table walk disable for translations using TTBR0_EL2. This bit controls whether a translation table walk is performed on a TLB miss, for an address that is translated using TTBR0_EL2. The encoding of this bit is:

`0b0` Perform translation table walks using TTBR0_EL2.

`0b1` A TLB miss on an address that is translated using TTBR0_EL2 generates a Translation fault. No translation table walk is performed.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bit [6]

Reserved, RES0.

T0SZ, bits [5:0]

The size offset of the memory region addressed by TTBR0_EL2. The region size is $2^{(64-T0SZ)}$ bytes.

The maximum and minimum possible values for T0SZ depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.

This field resets to an architecturally UNKNOWN value.

Accessing the TCR_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic TCR_EL2 or TCR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, TCR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return TCR_EL2;
elsif PSTATE.EL == EL3 then
    return TCR_EL2;
```

MSR TCR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TCR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    TCR_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, TCR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.TCR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x120];
    else
        return TCR_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return TCR_EL2;
    else
        return TCR_EL1;
elsif PSTATE.EL == EL3 then
    return TCR_EL1;
```

MSR TCR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.TCR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x120] = X[t];
    else
        TCR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        TCR_EL2 = X[t];
    else
        TCR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    TCR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.122 TCR_EL3, Translation Control Register (EL3)

The TCR_EL3 characteristics are:

Purpose

The control register for stage 1 of the EL3 translation regime.

Configurations

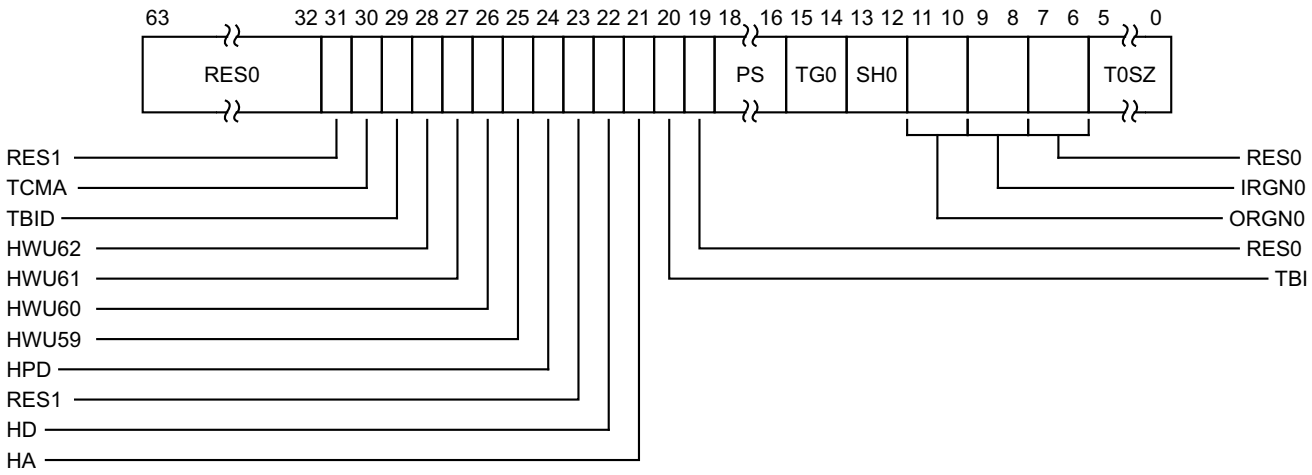
This register is present only when EL3 is implemented. Otherwise, direct accesses to TCR_EL3 are UNDEFINED.

Attributes

TCR_EL3 is a 64-bit register.

Field descriptions

The TCR_EL3 bit assignments are:



Any of the bits in TCR_EL3 are permitted to be cached in a TLB.

Bits [63:32]

Reserved, RES0.

Bit [31]

Reserved, RES1.

TCMA, bit [30]

When ARMv8.5-MemTag is implemented:

Controls the generation of Unchecked accesses at EL3 when address [59:56] = `0b0000`.

`0b0` This control has no effect on the generation of Unchecked accesses.

`0b1` All accesses are Unchecked.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TBID, bit [29]

When ARMv8.3-PAuth is implemented:

Controls the use of the top byte of instruction addresses for address matching.

`0b0` TCR_EL3.TBI applies to Instruction and Data accesses.

`0b1` TCR_EL3.TBI applies to Data accesses only.

This affects addresses where the address would be translated by tables pointed to by TTBR0_EL3.

For the purpose of this field, all cache maintenance and address translation instructions that perform address translation are treated as data accesses.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU62, bit [28]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 1 translation table Block or Page entry.

`0b0` Bit[62] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` Bit[62] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL3.HPD is 1.

The Effective value of this field is 0 if the value of TCR_EL3.HPD is 0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU61, bit [27]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 1 translation table Block or Page entry.

`0b0` Bit[61] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` Bit[61] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL3.HPD is 1.

The Effective value of this field is 0 if the value of TCR_EL3.HPD is 0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU60, bit [26]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 1 translation table Block or Page entry.

`0b0` Bit[60] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` Bit[60] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL3.HPD is 1.

The Effective value of this field is 0 if the value of TCR_EL3.HPD is 0.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

Reserved, RES0.

HWU59, bit [25]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 1 translation table Block or Page entry.

`0b0` Bit[59] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` Bit[59] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL3.HPD is 1.

The Effective value of this field is 0 if the value of TCR_EL3.HPD is 0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HPD, bit [24]

When ARMv8.1-HPD is implemented:

Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, PXNTable, and UXNTable, except NSTable, in the translation tables pointed to by TTBR0_EL3.

`0b0` Hierarchical permissions are enabled.

`0b1` Hierarchical permissions are disabled.

Note

In this case bit[61] (APTable[0]) and bit[59] (PXNTable) of the next level descriptor attributes are required to be ignored by the PE, and are no longer reserved, allowing them to be used by software.

When disabled, the permissions are treated as if the bits are zero.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [23]

Reserved, RES1.

HD, bit [22]

When ARMv8.1-TTHM is implemented:

Hardware management of dirty state in stage 1 translations from EL3.

`0b0` Stage 1 hardware management of dirty state disabled.

`0b1` Stage 1 hardware management of dirty state enabled, only if the HA bit is also set to 1.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HA, bit [21]

When ARMv8.1-TTHM is implemented:

Hardware Access flag update in stage 1 translations from EL3.

`0b0` Stage 1 Access flag update disabled.

`0b1` Stage 1 Access flag update enabled.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TBI, bit [20]

Top Byte Ignored. Indicates whether the top byte of an address is used for address match for the TTBR0_EL3 region, or ignored and used for tagged addresses.

`0b0` Top Byte used in the address calculation.

`0b1` Top Byte ignored in the address calculation.

This affects addresses generated in EL3 using AArch64 where the address would be translated by tables pointed to by TTBR0_EL3. It has an effect whether the EL3 translation regime is enabled or not.

If ARMv8.3-PAuth is implemented and TCR_EL3.TBID is 1, then this field only applies to Data accesses.

Otherwise, if the value of TBI is 1, then bits[63:56] of that target address are also set to 0 before the address is stored in the PC, in the following cases:

- A branch or procedure return within EL3.
- A exception taken to EL3.
- An exception return to EL3.

For more information, see *Address tagging in AArch64 state* on page D5-2534.

Note

This control determines the scope of address tagging. It never causes an exception to be generated.

This field resets to an architecturally UNKNOWN value.

Bit [19]

Reserved, RES0.

PS, bits [18:16]

Physical Address Size.

`0b000` 32 bits, 4GB.

`0b001` 36 bits, 64GB.

`0b010` 40 bits, 1TB.

`0b011` 42 bits, 4TB.

`0b100` 44 bits, 16TB.

`0b101` 48 bits, 256TB.

`0b110` 52 bits, 4PB.

Other values are reserved.

The reserved values behave in the same way as the `0b101` or `0b110` encoding, but software must not rely on this property as the behavior of the reserved values might change in a future revision of the architecture.

The value `0b110` is permitted only if ARMv8.2-LPA is implemented and the translation granule size is 64KB.

In an implementation that supports 52-bit PAs, if the value of this field is not `0b110` or a value treated as `0b110`, then bits[51:48] of every translation table base address for the stage of translation controlled by TCR_EL3 are `0b0000`.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TG0, bits [15:14]

Granule size for the TTBR0_EL3.

`0b00` 4KB.

`0b01` 64KB.

`0b10` 16KB.

Other values are reserved.

If the value is programmed to either a reserved value, or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

This field resets to an architecturally UNKNOWN value.

SH0, bits [13:12]

Shareability attribute for memory associated with translation table walks using TTBR0_EL3.

`0b00` Non-shareable.

`0b10` Outer Shareable.

`0b11` Inner Shareable.

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-7871.

This field resets to an architecturally UNKNOWN value.

ORGN0, bits [11:10]

Outer cacheability attribute for memory associated with translation table walks using TTBR0_EL3.

`0b00` Normal memory, Outer Non-cacheable.

`0b01` Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.

`0b10` Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.

`0b11` Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.

This field resets to an architecturally UNKNOWN value.

IRGN0, bits [9:8]

Inner cacheability attribute for memory associated with translation table walks using TTBR0_EL3.

`0b00` Normal memory, Inner Non-cacheable.

`0b01` Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.

`0b10` Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.

`0b11` Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

This field resets to an architecturally UNKNOWN value.

Bits [7:6]

Reserved, RES0.

T0SZ, bits [5:0]

The size offset of the memory region addressed by TTBR0_EL3. The region size is $2^{(64-T0SZ)}$ bytes.

The maximum and minimum possible values for T0SZ depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the TCR_EL3

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, TCR_EL3

op0	op1	CRn	CRm	op2
0b11	0b110	0b0010	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    return TCR_EL3;
```

MSR TCR_EL3, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b110	0b0010	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    TCR_EL3 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.123 TFSRE0_EL1, Tag Fault Status Register (EL0).

The TFSRE0_EL1 characteristics are:

Purpose

Holds accumulated Tag Check Faults occurring in EL0 that are not taken precisely.

Configurations

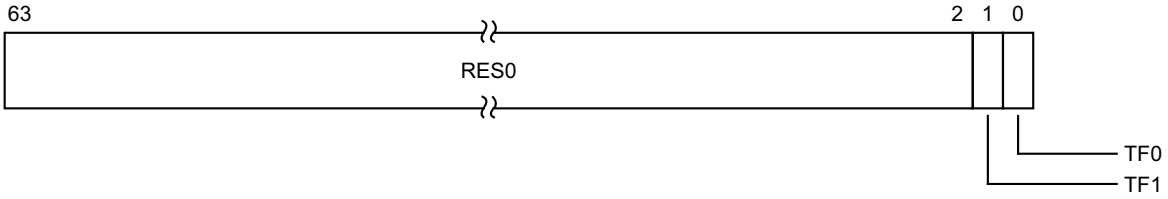
This register is present only when ARMv8.5-MemTag is implemented and ID_AA64PFR1_EL1.MTE != 0b0001. Otherwise, direct accesses to TFSRE0_EL1 are UNDEFINED.

Attributes

TFSRE0_EL1 is a 64-bit register.

Field descriptions

The TFSRE0_EL1 bit assignments are:



Bits [63:2]

Reserved, RES0.

TF1, bit [1]

Tag Check Fault. Asynchronously set to 1 when a Tag Check Fault using a virtual address with bit[55] == `0b1` occurs.

This field resets to an architecturally UNKNOWN value.

TF0, bit [0]

Tag Check Fault. Asynchronously set to 1 when a Tag Check Fault using a virtual address with bit[55] == `0b0` occurs.

This field resets to an architecturally UNKNOWN value.

Accessing the TFSRE0_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, TFSRE0_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0110	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.ATA == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.ATA == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        return TFSRE0_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.ATA == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return TFSRE0_EL1;
elsif PSTATE.EL == EL3 then
    return TFSRE0_EL1;
```

MSR TFSRE0_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0110	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.ATA == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.ATA == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        TFSRE0_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.ATA == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        TFSRE0_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    TFSRE0_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.124 TFSR_EL1, Tag Fault Status Register (EL1)

The TFSR_EL1 characteristics are:

Purpose

Holds accumulated Tag Check Faults occurring in EL1 that are not taken precisely.

Configurations

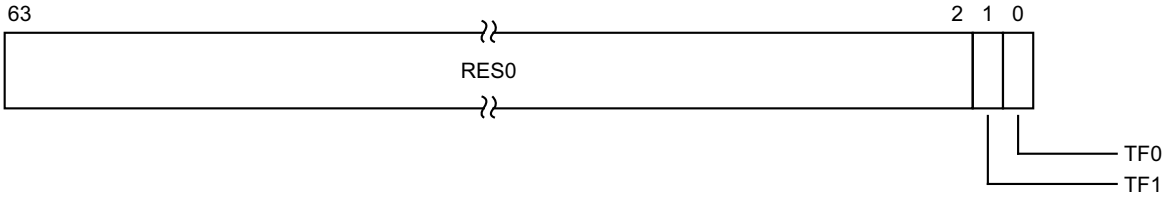
This register is present only when ARMv8.5-MemTag is implemented and ID_AA64PFR1_EL1.MTE != 0b0001. Otherwise, direct accesses to TFSR_EL1 are UNDEFINED.

Attributes

TFSR_EL1 is a 64-bit register.

Field descriptions

The TFSR_EL1 bit assignments are:



Bits [63:2]

Reserved, RES0.

TF1, bit [1]

Tag Check Fault. Asynchronously set to 1 when a Tag Check Fault using a virtual address with bit[55] == `0b1` occurs.

This field resets to an architecturally UNKNOWN value.

TF0, bit [0]

Tag Check Fault. Asynchronously set to 1 when a Tag Check Fault using a virtual address with bit[55] == `0b0` occurs.

This field resets to an architecturally UNKNOWN value.

Accessing the TFSR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, TFSR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0110	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1> == '01' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.ATA == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.ATA == '0' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x190];
    else
        return TFSR_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.ATA == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        return TFSR_EL2;
    else
        return TFSR_EL1;
elsif PSTATE.EL == EL3 then
    return TFSR_EL1;
```

MSR TFSR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0110	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1> == '01' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.ATA == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.ATA == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x190] = X[t];
    else
        TFSR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.ATA == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        TFSR_EL2 = X[t];
    else
        TFSR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    TFSR_EL1 = X[t];
```

MRS <Xt>, TFSR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0101	0b0110	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        return NVMem[0x190];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.ATA == '0' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            return TFSR_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        return TFSR_EL1;
    else
        UNDEFINED;
```

MSR TFSR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0101	0b0110	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        NVMem[0x190] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.ATA == '0' then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            TFSR_EL1 = X[t];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        TFSR_EL1 = X[t];
    else
        UNDEFINED;
```

MRS <Xt>, TFSR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0110	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.ATA == '0' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.ATA == '0' then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            return TFSR_EL1;
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.ATA == '0' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return TFSR_EL2;
elsif PSTATE.EL == EL3 then
    return TFSR_EL2;
```

MSR TFSR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0110	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.ATA == '0' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.ATA == '0' then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            TFSR_EL1 = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.ATA == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        TFSR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    TFSR_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.125 TFSR_EL2, Tag Fault Status Register (EL2)

The TFSR_EL2 characteristics are:

Purpose

Holds accumulated Tag Check Faults occurring in EL2 that are not taken precisely.

Configurations

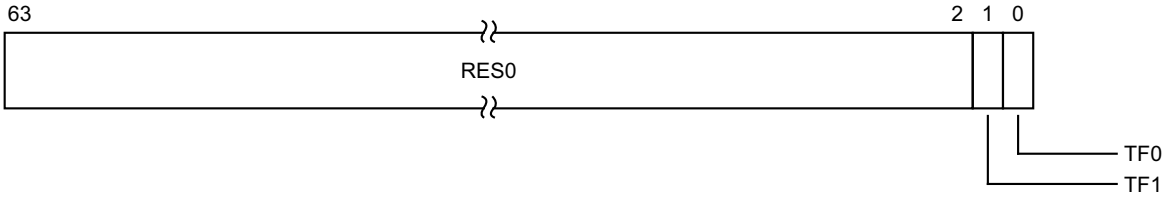
This register is present only when ARMv8.5-MemTag is implemented and ID_AA64PFR1_EL1.MTE != 0b0001. Otherwise, direct accesses to TFSR_EL2 are UNDEFINED.

Attributes

TFSR_EL2 is a 64-bit register.

Field descriptions

The TFSR_EL2 bit assignments are:



Bits [63:2]

Reserved, RES0.

TF1, bit [1]

Tag Check Fault. Asynchronously set to 1 when a Tag Check Fault using a virtual address with bit[55] == `0b1` occurs.

When HCR_EL2.E2H==`0b0`, this field is RES0.

This field resets to an architecturally UNKNOWN value.

TF0, bit [0]

Tag Check Fault. Asynchronously set to 1 when a Tag Check Fault using a virtual address with bit[55] == `0b0` occurs.

This field resets to an architecturally UNKNOWN value.

Accessing the TFSR_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic TFSR_EL2 or TFSR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, TFSR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0110	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.ATA == '0' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.ATA == '0' then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            return TFSR_EL1;
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.ATA == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return TFSR_EL2;
elsif PSTATE.EL == EL3 then
    return TFSR_EL2;
```

MSR TFSR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0110	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.ATA == '0' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.ATA == '0' then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            TFSR_EL1 = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.ATA == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        TFSR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    TFSR_EL2 = X[t];
```

MRS <Xt>, TFSR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0110	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1> == '01' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.ATA == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.ATA == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x190];
    else
        return TFSR_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.ATA == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        return TFSR_EL2;
    else
        return TFSR_EL1;
elsif PSTATE.EL == EL3 then
    return TFSR_EL1;
```

MSR TFSR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0110	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1> == '01' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.ATA == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.ATA == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x190] = X[t];
    else
        TFSR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.ATA == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        TFSR_EL2 = X[t];
    else
        TFSR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    TFSR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.126 TFSR_EL3, Tag Fault Status Register (EL3)

The TFSR_EL3 characteristics are:

Purpose

Holds accumulated Tag Check Faults occurring in EL3 that are not taken precisely.

Configurations

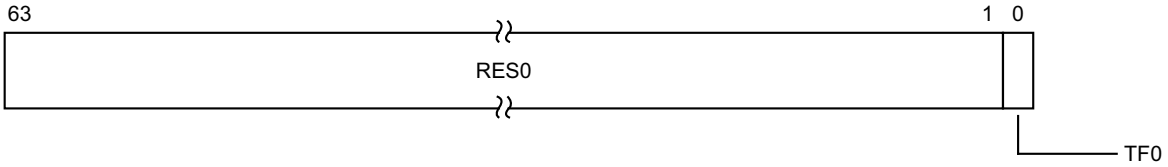
This register is present only when ARMv8.5-MemTag is implemented and ID_AA64PFR1_EL1.MTE != 0b0001. Otherwise, direct accesses to TFSR_EL3 are UNDEFINED.

Attributes

TFSR_EL3 is a 64-bit register.

Field descriptions

The TFSR_EL3 bit assignments are:



Bits [63:1]

Reserved, RES0.

TF0, bit [0]

Tag Check Fault. Asynchronously set to 1 when a Tag Check Fault using a virtual address with bit[55] == `0b0` occurs.

This field resets to an architecturally UNKNOWN value.

Accessing the TFSR_EL3

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, TFSR_EL3

op0	op1	CRn	CRm	op2
0b11	0b110	0b0101	0b0110	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    return TFSR_EL3;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR TFSR_EL3, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b110	0b0101	0b0110	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    TFSR_EL3 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.127 TPIDR_EL0, EL0 Read/Write Software Thread ID Register

The TPIDR_EL0 characteristics are:

Purpose

Provides a location where software executing at EL0 can store thread identifying information, for OS management purposes.

The PE makes no use of this register.

Configurations

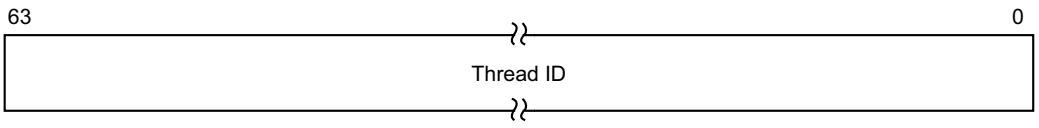
AArch64 System register TPIDR_EL0[31:0] is architecturally mapped to AArch32 System register TPIDRURW[31:0].

Attributes

TPIDR_EL0 is a 64-bit register.

Field descriptions

The TPIDR_EL0 bit assignments are:



Bits [63:0]

Thread ID. Thread identifying information stored by software running at this Exception level.

This field resets to an architecturally UNKNOWN value.

Accessing the TPIDR_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, TPIDR_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b010

```
if PSTATE.EL == EL0 then
    if EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HFGRTR_EL2.TPIDR_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return TPIDR_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.TPIDR_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return TPIDR_EL0;
elsif PSTATE.EL == EL2 then
    return TPIDR_EL0;
elsif PSTATE.EL == EL3 then
    return TPIDR_EL0;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR TPIDR_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b010

```
if PSTATE.EL == EL0 then
    if EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HFGWTR_EL2.TPIDR_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        TPIDR_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.TPIDR_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        TPIDR_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    TPIDR_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    TPIDR_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.128 TPIDR_EL1, EL1 Software Thread ID Register

The TPIDR_EL1 characteristics are:

Purpose

Provides a location where software executing at EL1 can store thread identifying information, for OS management purposes.

The PE makes no use of this register.

Configurations

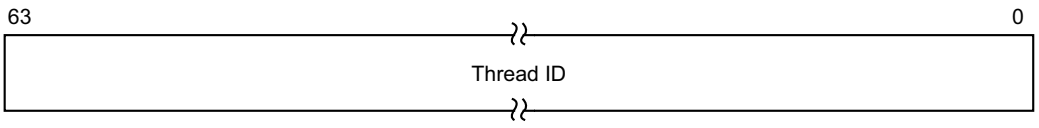
AArch64 System register TPIDR_EL1[31:0] is architecturally mapped to AArch32 System register TPIDRPRW[31:0].

Attributes

TPIDR_EL1 is a 64-bit register.

Field descriptions

The TPIDR_EL1 bit assignments are:



Bits [63:0]

Thread ID. Thread identifying information stored by software running at this Exception level.

This field resets to an architecturally UNKNOWN value.

Accessing the TPIDR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, TPIDR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.TPIDR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return TPIDR_EL1;
elsif PSTATE.EL == EL2 then
    return TPIDR_EL1;
elsif PSTATE.EL == EL3 then
    return TPIDR_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR TPIDR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.TPIDR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        TPIDR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    TPIDR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    TPIDR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.129 TPIDR_EL2, EL2 Software Thread ID Register

The TPIDR_EL2 characteristics are:

Purpose

Provides a location where software executing at EL2 can store thread identifying information, for OS management purposes.

The PE makes no use of this register.

Configurations

AArch64 System register TPIDR_EL2[31:0] is architecturally mapped to AArch32 System register HTPIDR[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

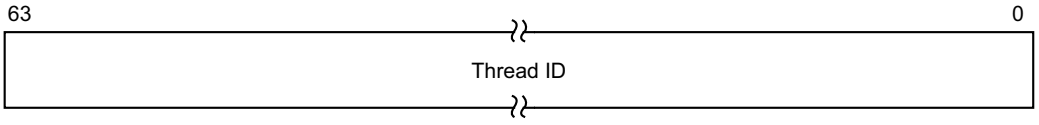
This register has no effect if EL2 is not enabled in the current Security state.

Attributes

TPIDR_EL2 is a 64-bit register.

Field descriptions

The TPIDR_EL2 bit assignments are:



Bits [63:0]

Thread ID. Thread identifying information stored by software running at this Exception level.

This field resets to an architecturally UNKNOWN value.

Accessing the TPIDR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, TPIDR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1101	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        return NVMem[0x090];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return TPIDR_EL2;
elsif PSTATE.EL == EL3 then
    return TPIDR_EL2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR TPIDR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1101	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NVMem[0x090] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TPIDR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    TPIDR_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.130 TPIDR_EL3, EL3 Software Thread ID Register

The TPIDR_EL3 characteristics are:

Purpose

Provides a location where software executing at EL3 can store thread identifying information, for OS management purposes.

The PE makes no use of this register.

Configurations

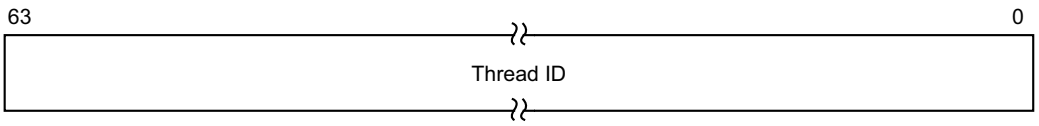
This register is present only when EL3 is implemented. Otherwise, direct accesses to TPIDR_EL3 are UNDEFINED.

Attributes

TPIDR_EL3 is a 64-bit register.

Field descriptions

The TPIDR_EL3 bit assignments are:



Bits [63:0]

Thread ID. Thread identifying information stored by software running at this Exception level.

This field resets to an architecturally UNKNOWN value.

Accessing the TPIDR_EL3

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, TPIDR_EL3

op0	op1	CRn	CRm	op2
0b11	0b110	0b1101	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    return TPIDR_EL3;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR TPIDR_EL3, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b110	0b1101	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    TPIDR_EL3 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.131 TPIDRRO_EL0, EL0 Read-Only Software Thread ID Register

The TPIDRRO_EL0 characteristics are:

Purpose

Provides a location where software executing at EL1 or higher can store thread identifying information that is visible to software executing at EL0, for OS management purposes.

The PE makes no use of this register.

Configurations

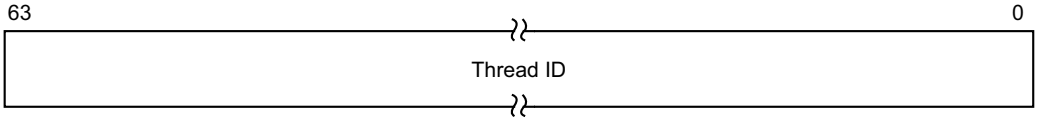
AArch64 System register TPIDRRO_EL0[31:0] is architecturally mapped to AArch32 System register TPIDRURO[31:0].

Attributes

TPIDRRO_EL0 is a 64-bit register.

Field descriptions

The TPIDRRO_EL0 bit assignments are:



Bits [63:0]

Thread ID. Thread identifying information stored by software running at this Exception level.

Accessing the TPIDRRO_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, TPIDRRO_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b011

```
if PSTATE.EL == EL0 then
    if EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HFGRTR_EL2.TPIDRRO_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return TPIDRRO_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.TPIDRRO_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return TPIDRRO_EL0;
elsif PSTATE.EL == EL2 then
    return TPIDRRO_EL0;
elsif PSTATE.EL == EL3 then
    return TPIDRRO_EL0;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR TPIDRRO_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.TPIDRRO_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        TPIDRRO_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    TPIDRRO_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    TPIDRRO_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.132 TTBR0_EL1, Translation Table Base Register 0 (EL1)

The TTBR0_EL1 characteristics are:

Purpose

Holds the base address of the translation table for the initial lookup for stage 1 of the translation of an address from the lower VA range in the EL1&0 translation regime, and other information for this translation regime.

Configurations

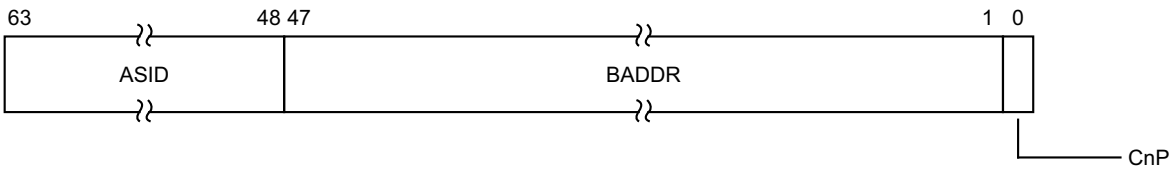
AArch64 System register TTBR0_EL1[63:0] is architecturally mapped to AArch32 System register TTBR0[63:0].

Attributes

TTBR0_EL1 is a 64-bit register.

Field descriptions

The TTBR0_EL1 bit assignments are:



ASID, bits [63:48]

An ASID for the translation table base address. The TCR_EL1.A1 field selects either TTBR0_EL1.ASID or TTBR1_EL1.ASID.

If the implementation has only 8 bits of ASID, then the upper 8 bits of this field are RES0.

This field resets to an architecturally UNKNOWN value.

BADDR, bits [47:1]

Translation table base address, A[47:x] or A[51:x], bits[47:1].

Note

- Translation table base addresses of 52 bits, A[51:x], are supported only in an implementation that includes ARMv8.2-LPA and is using the 64KB translation granule.
- A translation table must be aligned to the size of the table, except that when using a translation table base address larger than 48 bits the minimum alignment of a table containing fewer than eight entries is 64 bytes.

In an implementation that includes ARMv8.2-LPA, if the value of TCR_EL1.IPS is `0b110`, then:

- Register bits[47:z] hold bits[47:z] of the stage 1 translation table base address, where z is determined as follows:
  - If x >= 6 then z=x.
  - Otherwise, z=6.
- Register bits[5:2] hold bits[51:48] of the stage 1 translation table base address.
- When z>x register bits[(z-1):x] are RES0, and bits[(z-1):x] of the translation table base address are zero.
- When x>6 register bits[(x-1):6] are RES0.
- Register bit[1] is RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Bits[5:2] of the stage 1 translation table base address are zero.
- In an implementation that includes ARMv8.2-TTCNP bit[0] of the stage 1 translation table base address is zero.

Note

- In an implementation that includes ARMv8.2-LPA a TCR_EL1.IPS value of `0b110`, that selects an IPA size of 52 bits, is permitted only when using the 64KB translation granule.
- When the value of ID_AA64MMFR0_EL1.PARange indicates that the implementation does not support a 52 bit PA size, if a translation table lookup uses this register with the 64KB translation granule when the value of TCR_EL1.IPS is `0b110` and the value of register bits[5:2] is nonzero, an Address size fault is generated.

If the Effective value of TCR_EL1.IPS is not `0b110` then:

- Register bits[47:x] hold bits[47:x] of the stage 1 translation table base address.
- Register bits[(x-1):1] are RES0.
- If the implementation supports 52-bit PAs and IPAs, then bits[51:48] of the translation table base addresses used in this stage of translation are `0b0000`.

Note

This definition applies:

- To an implementation that includes ARMv8.2-LPA and is using a translation granule smaller than 64KB.
- To any implementation that does not include ARMv8.2-LPA.

If any TTBR0_EL1[47:0] bit that is defined as RES0 has the value 1 when a translation table walk is performed using TTBR0_EL1, then the translation table base address might be misaligned, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Bits[x-1:0] of the translation table base address are treated as if all the bits are zero. The value read back from the corresponding register bits is either the value written to the register or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of TCR_EL1.T0SZ, the stage of translation, and the translation granule size.

This field resets to an architecturally UNKNOWN value.

CnP, bit [0]

When ARMv8.2-TTCNP is implemented:

Common not Private. This bit indicates whether each entry that is pointed to by TTBR0_EL1 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of TTBR0_EL1.CnP is 1.

`0b0` The translation table entries pointed to by TTBR0_EL1, for the current translation regime and ASID, are permitted to differ from corresponding entries for TTBR0_EL1 for other PEs in the Inner Shareable domain. This is not affected by:

- The value of TTBR0_EL1.CnP on those other PEs.
- The value of the current ASID.
- If EL2 is implemented and enabled in the current Security state, the value of the current VMID.

`0b1` The translation table entries pointed to by TTBR0_EL1 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of TTBR0_EL1.CnP is 1 and all of the following apply:

- The translation table entries are pointed to by TTBR0_EL1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The translation tables relate to the same translation regime.
- The ASID is the same as the current ASID.
- If EL2 is implemented and enabled in the current Security state, the value of the current VMID.

This field is permitted to be cached in a TLB.

When a TLB combines entries from stage 1 translation and stage 2 translation into a single entry, that entry can only be shared between different PEs if the value of the CnP bit is 1 for both stage 1 and stage 2.

Note

If the value of the TTBR0_EL1.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those TTBR0_EL1s do not point to the same translation table entries when the other conditions specified for the case when the value of CnP is 1 apply, then the results of translations are CONSTRAINED UNPREDICTABLE, see *CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values* on page K1-7856.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing the TTBR0_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic TTBR0_EL1 or TTBR0_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, TTBR0_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.TTBR0_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x200];
    else
        return TTBR0_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return TTBR0_EL2;
    else
        return TTBR0_EL1;
elsif PSTATE.EL == EL3 then
    return TTBR0_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR TTBR0_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.TTBR0_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x200] = X[t];
    else
        TTBR0_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        TTBR0_EL2 = X[t];
    else
        TTBR0_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    TTBR0_EL1 = X[t];
```

MRS <Xt>, TTBR0_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0010	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        return NVMem[0x200];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return TTBR0_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        return TTBR0_EL1;
    else
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR TTBR0_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0010	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        NVMem[0x200] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        TTBR0_EL1 = X[t];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        TTBR0_EL1 = X[t];
    else
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.133 TTBR0_EL2, Translation Table Base Register 0 (EL2)

The TTBR0_EL2 characteristics are:

Purpose

When HCR_EL2.E2H is 0, holds the base address of the translation table for the initial lookup for stage 1 of an address translation in the EL2 translation regime, and other information for this translation regime.

When HCR_EL2.E2H is 1, holds the base address of the translation table for the initial lookup for stage 1 of the translation of an address from the lower VA range in the EL2&0 translation regime, and other information for this translation regime.

Configurations

AArch64 System register TTBR0_EL2[47:1] is architecturally mapped to AArch32 System register HTTBR[47:1].

If EL2 is not implemented, this register is RES0 from EL3.

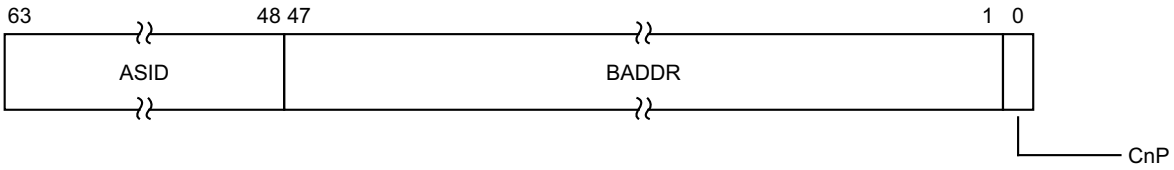
This register has no effect if EL2 is not enabled in the current Security state.

Attributes

TTBR0_EL2 is a 64-bit register.

Field descriptions

The TTBR0_EL2 bit assignments are:



ASID, bits [63:48]

When ARMv8.1-VHE is implemented:

When HCR_EL2.E2H is 0, this field is RES0.

When HCR_EL2.E2H is 1, it holds an ASID for the translation table base address. The TCR_EL2.A1 field selects either TTBR0_EL2.ASID or TTBR1_EL2.ASID.

If the implementation has only 8 bits of ASID, then the upper 8 bits of this field are RES0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

BADDR, bits [47:1]

Translation table base address, A[47:x] or A[51:x], bits[47:1].

Note

- Translation table base addresses of 52 bits, A[51:x], are supported only in an implementation that includes ARMv8.2-LPA and is using the 64KB translation granule.
- A translation table must be aligned to the size of the table, except that when using a translation table base address larger than 48 bits the minimum alignment of a table containing fewer than eight entries is 64 bytes.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In an implementation that includes ARMv8.2-LPA, if the value of TCR_EL2.{I}PS is `0b110`, then:

- Register bits[47:z] hold bits[47:z] of the stage 1 translation table base address, where z is determined as follows:
  - If x >= 6 then z=x.
  - Otherwise, z=6.
- Register bits[5:2] hold bits[51:48] of the stage 1 translation table base address.
- When z>x register bits[(z-1):x] are RES0, and bits[(z-1):x] of the translation table base address are zero.
- When x>6 register bits[(x-1):6] are RES0.
- Register bit[1] is RES0.
- Bits[5:2] of the stage 1 translation table base address are zero.
- In an implementation that includes ARMv8.2-TTCNP bit[0] of the stage 1 translation table base address is zero.

Note

In an implementation that includes ARMv8.2-LPA:

- A TCR_EL2.{I}PS value of `0b110`, that selects an OA size of 52 bits, is permitted only when using the 64KB translation granule.
- The OA size is specified by:
  - The value of TCR_EL2.PS when the value of HCR_EL2.E2H is 0.
  - The value of TCR_EL2.IPS when the value of HCR_EL2.E2H is 1.

When the value of ID_AA64MMFR0_EL1.PARange indicates that the implementation does not support a 52 bit PA size, if a translation table lookup uses this register with the 64KB translation granule when the value of TCR_EL2.{I}PS is `0b110` and the value of register bits[5:2] is nonzero, an Address size fault is generated.

If the Effective value of TCR_EL2.{I}PS is not `0b110` then:

- Register bits[47:x] hold bits[47:x] of the stage 1 translation table base address.
- Register bits[(x-1):1] are RES0.
- If the implementation supports 52-bit PAs and IPAs, then bits[51:48] of the translation table base addresses used in this stage of translation are `0b0000`.

Note

This definition applies:

- To an implementation that includes ARMv8.2-LPA and is using a translation granule smaller than 64KB.
- To any implementation that does not include ARMv8.2-LPA.

If any TTBR0_EL2[47:0] bit that is defined as RES0 has the value 1 when a translation table walk is performed using TTBR0_EL2, then the translation table base address might be misaligned, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Bits[x-1:0] of the translation table base address are treated as if all the bits are zero. The value read back from the corresponding register bits is either the value written to the register or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of TCR_EL2.T0SZ, the stage of translation, and the translation granule size.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CnP, bit [0]

When ARMv8.2-TTCNP is implemented:

Common not Private. This bit indicates whether each entry that is pointed to by TTBR0_EL2 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of TTBR0_EL2.CnP is 1.

`0b0` The translation table entries pointed to by TTBR0_EL2 for the current translation regime, and ASID if applicable, are permitted to differ from corresponding entries for TTBR0_EL2 for other PEs in the Inner Shareable domain. This is not affected by:

- The value of TTBR0_EL2.CnP on those other PEs.
- When the current translation regime is the EL2&0 regime, the value of the current ASID.

`0b1` The translation table entries pointed to by TTBR0_EL2 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of TTBR0_EL2.CnP is 1 and all of the following apply:

- The translation table entries are pointed to by TTBR0_EL2.
- The translation tables relate to the same translation regime.
- If that translation regime is the EL2&0 regime, the ASID is the same as the current ASID.

This field is permitted to be cached in a TLB.

Note

If the value of the TTBR0_EL2.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those TTBR0_EL2s do not point to the same translation table entries when the other conditions specified for the case when the value of CnP is 1 apply, then the results of translations are CONSTRAINED UNPREDICTABLE, see *CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values* on page K1-7856.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing the TTBR0_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic TTBR0_EL2 or TTBR0_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, TTBR0_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return TTBR0_EL2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    return TTBR0_EL2;
```

MSR TTBR0_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TTBR0_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    TTBR0_EL2 = X[t];
```

MRS <Xt>, TTBR0_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.TTBR0_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x200];
    else
        return TTBR0_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return TTBR0_EL2;
    else
        return TTBR0_EL1;
elsif PSTATE.EL == EL3 then
    return TTBR0_EL1;
```

MSR TTBR0_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.TTBR0_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x200] = X[t];
    else
        TTBR0_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        TTBR0_EL2 = X[t];
    else
        TTBR0_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    TTBR0_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.134 TTBR0_EL3, Translation Table Base Register 0 (EL3)

The TTBR0_EL3 characteristics are:

Purpose

Holds the base address of the translation table for the initial lookup for stage 1 of an address translation in the EL3 translation regime, and other information for this translation regime.

Configurations

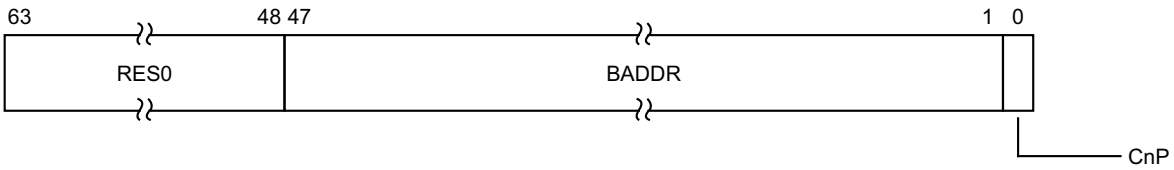
This register is present only when EL3 is implemented. Otherwise, direct accesses to TTBR0_EL3 are UNDEFINED.

Attributes

TTBR0_EL3 is a 64-bit register.

Field descriptions

The TTBR0_EL3 bit assignments are:



Bits [63:48]

Reserved, RES0.

BADDR, bits [47:1]

Translation table base address, A[47:x] or A[51:x].

Note

- Translation table base addresses of 52 bits, A[51:x], are supported only in an implementation that includes ARMv8.2-LPA and is using the 64KB translation granule.
- A translation table must be aligned to the size of the table, except that when using translation table base address larger than 48 bits the minimum alignment of a table containing fewer than eight entries is 64 bytes.

In an implementation that includes ARMv8.2-LPA, if the value of TCR_EL3.PS is `0b110` then:

- Register bits[47:z] hold bits[47:z] of the stage 1 translation table base address, where z is determined as follows:
  - If x >= 6 then z=x.
  - Otherwise, z=6.
- Register bits[5:2] hold bits[51:48] of the stage 1 translation table base address.
- When z>x register bits[(z-1):x] are RES0, and bits[(z-1):x] of the translation table base address are zero.
- When x>6 register bits[(x-1):6] are RES0.
- Register bit[1] is RES0.
- Bits[5:2] of the stage 1 translation table base address are zero.
- In an implementation that includes ARMv8.2-TTCNP bit[0] of the stage 1 translation table base address is zero.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

- In an implementation that includes ARMv8.2-LPA a TCR_EL3.PS value of `0b110`, that selects a PA size of 52 bits, is permitted only when using the 64KB translation granule.
- When the value of ID_AA64MMFR0_EL1.PARange indicates that the implementation does not support a 52 bit PA size, if a translation table lookup uses this register with the 64KB translation granule when the value of TCR_EL3.PS is `0b110` and the value of register bits[5:2] is nonzero, an Address size fault is generated.

If the Effective value of TCR_EL3.PS is not `0b110` then:

- Register bits[47:x] hold bits[47:x] of the stage 1 translation table base address.
- Register bits[(x-1):1] are RES0.
- If the implementation supports 52-bit PAs and IPAs, then bits[51:48] of the translation table base addresses used in this stage of translation are `0b0000`.

Note

This definition applies:

- To an implementation that includes ARMv8.2-LPA and is using a translation granule smaller than 64KB.
- To any implementation that does not include ARMv8.2-LPA.

If any TTBR0_EL3[47:0] bit that is defined as RES0 has the value 1 when a translation table walk is performed using TTBR0_EL3, then the translation table base address might be misaligned, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Bits[x-1:0] of the translation table base address are treated as if all the bits are zero. The value read back from the corresponding register bits is either the value written to the register or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of TCR_EL3.T0SZ, the stage of translation, and the translation granule size.

This field resets to an architecturally UNKNOWN value.

CnP, bit [0]

When ARMv8.2-TTCNP is implemented:

Common not Private. This bit indicates whether each entry that is pointed to by TTBR0_EL3 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of TTBR0_EL3.CnP is 1.

`0b0` The translation table entries pointed to by TTBR0_EL3, for the current translation regime, are permitted to differ from corresponding entries for TTBR0_EL3 for other PEs in the Inner Shareable domain. This is not affected by the value of TTBR0_EL3.CnP on those other PEs.

`0b1` The translation table entries pointed to by TTBR0_EL3 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of TTBR0_EL3.CnP is 1 and the translation table entries are pointed to by TTBR0_EL3.

This field is permitted to be cached in a TLB.

Note

If the value of the TTBR0_EL3.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those TTBR0_EL3s do not point to the same translation table entries the results of translations using TTBR0_EL3 are CONSTRAINED UNPREDICTABLE, see *CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values* on page K1-7856.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

Reserved, RES0.

Accessing the TTBR0_EL3

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, TTBR0_EL3

op0	op1	CRn	CRm	op2
0b11	0b110	0b0010	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    return TTBR0_EL3;
```

MSR TTBR0_EL3, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b110	0b0010	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    TTBR0_EL3 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.135 TTBR1_EL1, Translation Table Base Register 1 (EL1)

The TTBR1_EL1 characteristics are:

Purpose

Holds the base address of the translation table for the initial lookup for stage 1 of the translation of an address from the higher VA range in the EL1&0 stage 1 translation regime, and other information for this translation regime.

Configurations

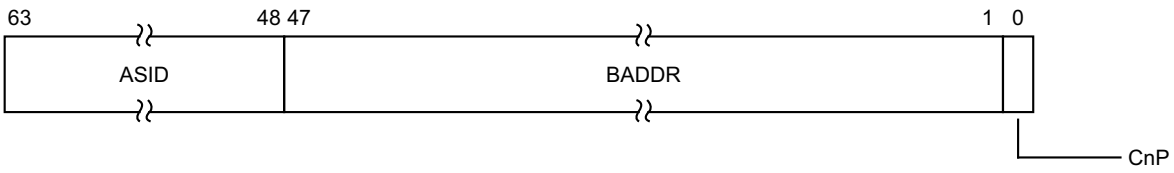
AArch64 System register TTBR1_EL1[63:0] is architecturally mapped to AArch32 System register TTBR1[63:0].

Attributes

TTBR1_EL1 is a 64-bit register.

Field descriptions

The TTBR1_EL1 bit assignments are:



ASID, bits [63:48]

An ASID for the translation table base address. The TCR_EL1.A1 field selects either TTBR0_EL1.ASID or TTBR1_EL1.ASID.

If the implementation has only 8 bits of ASID, then the upper 8 bits of this field are RES0.

This field resets to an architecturally UNKNOWN value.

BADDR, bits [47:1]

Translation table base address, A[47:x] or A[51:x], bits[47:1].

Note

- Translation table base addresses of 52 bits, A[51:x], are supported only in an implementation that includes ARMv8.2-LPA and is using the 64KB translation granule.
- A translation table must be aligned to the size of the table, except that when using a translation table base address larger than 48 bits the minimum alignment of a table containing fewer than eight entries is 64 bytes.

In an implementation that includes ARMv8.2-LPA, if the value of TCR_EL1.IPS is `0b110`, then:

- Register bits[47:z] hold bits[47:z] of the stage 1 translation table base address, where z is determined as follows:
  - If x >= 6 then z=x.
  - Otherwise, z=6.
- Register bits[5:2] hold bits[51:48] of the stage 1 translation table base address.
- When z>x register bits[(z-1):x] are RES0, and bits[(z-1):x] of the translation table base address are zero.
- When x>6 register bits[(x-1):6] are RES0.
- Register bit[1] is RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Bits[5:2] of the stage 1 translation table base address are zero.
- In an implementation that includes ARMv8.2-TTCNP bit[0] of the stage 1 translation table base address is zero.

Note

- In an implementation that includes ARMv8.2-LPA a TCR_EL1.IPS value of `0b110`, that selects an IPA size of 52 bits, is permitted only when using the 64KB translation granule.
- When the value of ID_AA64MMFR0_EL1.PARange indicates that the implementation does not support a 52 bit PA size, if a translation table lookup uses this register with the 64KB translation granule when the value of TCR_EL1.IPS is `0b110` and the value of register bits[5:2] is nonzero, an Address size fault is generated.

If the Effective value of TCR_EL1.IPS is not `0b110` then:

- Register bits[47:x] hold bits[47:x] of the stage 1 translation table base address.
- Register bits[(x-1):1] are RES0.
- If the implementation supports 52-bit PAs and IPAs, then bits[51:48] of the translation table base addresses used in this stage of translation are `0b0000`.

Note

This definition applies:

- To an implementation that includes ARMv8.2-LPA and is using a translation granule smaller than 64KB.
- To any implementation that does not include ARMv8.2-LPA.

If any TTBR1_EL1[47:0] bit that is defined as RES0 has the value 1 when a translation table walk is performed using TTBR1_EL1, then the translation table base address might be misaligned, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Bits[x-1:0] of the translation table base address are treated as if all the bits are zero. The value read back from the corresponding register bits is either the value written to the register or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of TCR_EL1.T1SZ, the stage of translation, and the translation granule size.

This field resets to an architecturally UNKNOWN value.

CnP, bit [0]

When ARMv8.2-TTCNP is implemented:

Common not Private. This bit indicates whether each entry that is pointed to by TBR1_EL1 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of TTBR1_EL1.CnP is 1.

`0b0` The translation table entries pointed to by TTBR1_EL1, for the current translation regime and ASID, are permitted to differ from corresponding entries for TTBR1_EL1 for other PEs in the Inner Shareable domain. This is not affected by:

- The value of TTBR1_EL1.CnP on those other PEs.
- The value of the current ASID.
- If EL2 is implemented and enabled in the current Security state, the value of the current VMID.

`0b1` The translation table entries pointed to by TTBR1_EL1 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of TTBR1_EL1.CnP is 1 and all of the following apply:

- The translation table entries are pointed to by TTBR1_EL1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The translation tables relate to the same translation regime.
- The ASID is the same as the current ASID.
- If EL2 is implemented and enabled in the current Security state, the value of the current VMID.

This field is permitted to be cached in a TLB.

When a TLB combines entries from stage 1 translation and stage 2 translation into a single entry, that entry can only be shared between different PEs if the value of the CnP bit is 1 for both stage 1 and stage 2.

Note

If the value of the TTBR1_EL1.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those TTBR1_EL1s do not point to the same translation table entries when the other conditions specified for the case when the value of CnP is 1 apply, then the results of translations are CONSTRAINED UNPREDICTABLE, see *CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values* on page K1-7856.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing the TTBR1_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic TTBR1_EL1 or TTBR1_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, TTBR1_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.TTBR1_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x210];
    else
        return TTBR1_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return TTBR1_EL2;
    else
        return TTBR1_EL1;
elsif PSTATE.EL == EL3 then
    return TTBR1_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR TTBR1_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.TTBR1_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x210] = X[t];
    else
        TTBR1_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        TTBR1_EL2 = X[t];
    else
        TTBR1_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    TTBR1_EL1 = X[t];
```

MRS <Xt>, TTBR1_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0010	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        return NVMem[0x210];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return TTBR1_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        return TTBR1_EL1;
    else
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR TTBR1_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0010	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        NVMem[0x210] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        TTBR1_EL1 = X[t];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        TTBR1_EL1 = X[t];
    else
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.136 TTBR1_EL2, Translation Table Base Register 1 (EL2)

The TTBR1_EL2 characteristics are:

Purpose

When HCR_EL2.E2H is 1, holds the base address of the translation table for the initial lookup for stage 1 of the translation of an address from the higher VA range in the EL2&0 translation regime, and other information for this translation regime.

Note

When HCR_EL2.E2H is 0, the contents of this register are ignored by the PE, except for a direct read or write of the register.

Configurations

This register is present only when ARMv8.1-VHE is implemented. Otherwise, direct accesses to TTBR1_EL2 are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

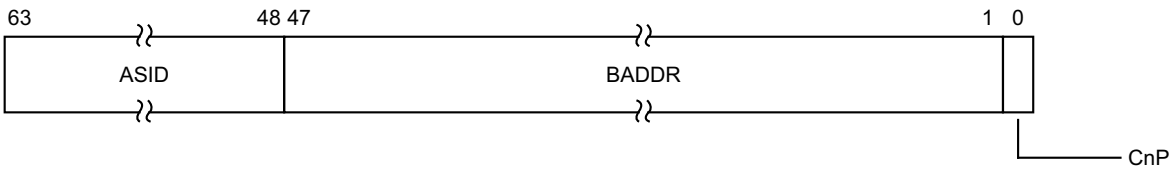
This register has no effect if EL2 is not enabled in the current Security state.

Attributes

TTBR1_EL2 is a 64-bit register.

Field descriptions

The TTBR1_EL2 bit assignments are:



ASID, bits [63:48]

An ASID for the translation table base address. The TCR_EL2.A1 field selects either TTBR0_EL2.ASID or TTBR1_EL2.ASID.

If the implementation has only 8 bits of ASID, then the upper 8 bits of this field are RES0.

This field resets to an architecturally UNKNOWN value.

BADDR, bits [47:1]

Translation table base address, A[47:x] or A[51:x], bits[47:1].

Note

- Translation table base addresses of 52 bits, A[51:x], are supported only in an implementation that includes ARMv8.2-LPA and is using the 64KB translation granule.
- A translation table must be aligned to the size of the table, except that when using a translation table base address larger than 48 bits the minimum alignment of a table containing fewer than eight entries is 64 bytes.

In an implementation that includes ARMv8.2-LPA, if the value of TCR_EL2.{I}PS is `0b110`, then:

- Register bits[47:z] hold bits[47:z] of the stage 1 translation table base address, where z is determined as follows:
  - If x >= 6 then z=x.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— Otherwise, z=6.

- Register bits[5:2] hold bits[51:48] of the stage 1 translation table base address.
- When z>x register bits[(z-1):x] are RES0, and bits[(z-1):x] of the translation table base address are zero.
- When x>6 register bits[(x-1):6] are RES0.
- Register bit[1] is RES0.
- Bits[5:2] of the stage 1 translation table base address are zero.
- In an implementation that includes ARMv8.2-TTCNP bit[0] of the stage 1 translation table base address is zero.

Note

- In an implementation that includes ARMv8.2-LPA a TCR_EL2.IPS value of `0b110`, that selects an OA size of 52 bits, is permitted only when using the 64KB translation granule.
- When the value of ID_AA64MMFR0_EL1.PARange indicates that the implementation does not support a 52 bit PA size, if a translation table lookup uses this register with the 64KB translation granule when the value of TCR_EL2.IPS is `0b110` and the value of register bits[5:2] is nonzero, an Address size fault is generated.

If the Effective value of TCR_EL2.IPS is not `0b110` then:

- Register bits[47:x] hold bits[47:x] of the stage 1 translation table base address.
- Register bits[(x-1):1] are RES0.
- If the implementation supports 52-bit PAs and IPAs, then bits[51:48] of the translation table base addresses used in this stage of translation are `0b0000`.

Note

This definition applies:

- To an implementation that includes ARMv8.2-LPA and is using a translation granule smaller than 64KB.
- To any implementation that does not include ARMv8.2-LPA.

If any TTBR1_EL2[47:0] bit that is defined as RES0 has the value 1 when a translation table walk is performed using TTBR1_EL2, then the translation table base address might be misaligned, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Bits[x-1:0] of the translation table base address are treated as if all the bits are zero. The value read back from the corresponding register bits is either the value written to the register or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of TCR_EL2.T1SZ, the stage of translation, and the translation granule size.

This field resets to an architecturally UNKNOWN value.

CnP, bit [0]

When ARMv8.2-TTCNP is implemented:

Common not Private. This bit indicates whether each entry that is pointed to by TBR1_EL2 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of TTBR1_EL2.CnP is 1.

`0b0` The translation table entries pointed to by TTBR1_EL2 for the current ASID are permitted to differ from corresponding entries for TTBR1_EL2 for other PEs in the Inner Shareable domain. This is not affected by:

- The value of TTBR1_EL2.CnP on those other PEs.
- The value of the current ASID.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` The translation table entries pointed to by TTBR1_EL2 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of TTBR1_EL2.CnP is 1 and all of the following apply:

- The translation table entries are pointed to by TTBR1_EL2.
- The ASID is the same as the current ASID.

This field is permitted to be cached in a TLB.

Note

- TTBR1_EL2 is accessible only when the value of HCR_EL2.E2H is 1, meaning the current translation regime is the EL2&0 regime.
- If the value of the TTBR1_EL2.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those TTBR1_EL2s do not point to the same translation table entries when the other conditions specified for the case when the value of CnP is 1 apply, then the results of translations are CONSTRAINED UNPREDICTABLE, see *CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values* on page K1-7856.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing the TTBR1_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic TTBR1_EL2 or TTBR1_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, TTBR1_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return TTBR1_EL2;
elsif PSTATE.EL == EL3 then
    return TTBR1_EL2;
```

MSR TTBR1_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TTBR1_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    TTBR1_EL2 = X[t];
```

MRS <Xt>, TTBR1_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.TTBR1_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x210];
    else
        return TTBR1_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return TTBR1_EL2;
    else
        return TTBR1_EL1;
elsif PSTATE.EL == EL3 then
    return TTBR1_EL1;
```

MSR TTBR1_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.TTBR1_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x210] = X[t];
    else
        TTBR1_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        TTBR1_EL2 = X[t];
    else
        TTBR1_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    TTBR1_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.137 VBAR_EL1, Vector Base Address Register (EL1)

The VBAR_EL1 characteristics are:

Purpose

Holds the vector base address for any exception that is taken to EL1.

Configurations

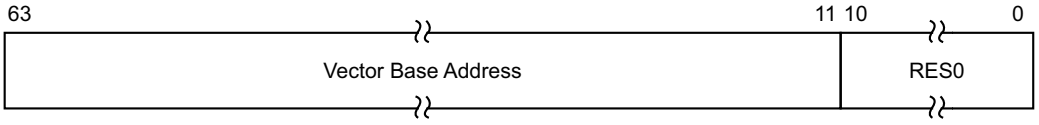
AArch64 System register VBAR_EL1[31:0] is architecturally mapped to AArch32 System register VBAR[31:0].

Attributes

VBAR_EL1 is a 64-bit register.

Field descriptions

The VBAR_EL1 bit assignments are:



Bits [63:11]

Vector Base Address. Base address of the exception vectors for exceptions taken to EL1.

If the implementation does not support ARMv8.2-LVA, then:

- If tagged addresses are being used, bits [55:48] of VBAR_EL1 must be the same or else the use of the vector address will result in a recursive exception.
- If tagged addresses are not being used, bits [63:48] of VBAR_EL1 must be the same or else the use of the vector address will result in a recursive exception.

If the implementation supports ARMv8.2-LVA, then:

- If tagged addresses are being used, bits [55:52] of VBAR_EL1 must be the same or else the use of the vector address will result in a recursive exception.
- If tagged addresses are not being used, bits [63:52] of VBAR_EL1 must be the same or else the use of the vector address will result in a recursive exception.

This field resets to an architecturally UNKNOWN value.

Bits [10:0]

Reserved, RES0.

Accessing the VBAR_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic VBAR_EL1 or VBAR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, VBAR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1> == '01' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.VBAR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x250];
    else
        return VBAR_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return VBAR_EL2;
    else
        return VBAR_EL1;
elsif PSTATE.EL == EL3 then
    return VBAR_EL1;
```

MSR VBAR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1> == '01' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.VBAR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x250] = X[t];
    else
        VBAR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        VBAR_EL2 = X[t];
    else
        VBAR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    VBAR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, VBAR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b1100	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        return NVMem[0x250];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return VBAR_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        return VBAR_EL1;
    else
        UNDEFINED;
```

MSR VBAR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b1100	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        NVMem[0x250] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        VBAR_EL1 = X[t];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        VBAR_EL1 = X[t];
    else
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.138 VBAR_EL2, Vector Base Address Register (EL2)

The VBAR_EL2 characteristics are:

Purpose

Holds the vector base address for any exception that is taken to EL2.

Configurations

AArch64 System register VBAR_EL2[31:0] is architecturally mapped to AArch32 System register HVBAR[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

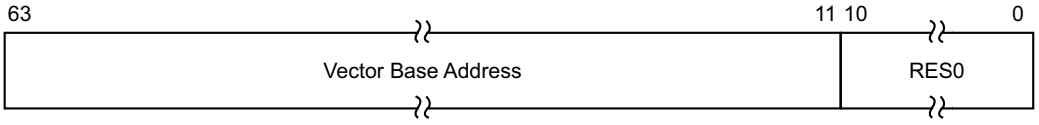
This register has no effect if EL2 is not enabled in the current Security state.

Attributes

VBAR_EL2 is a 64-bit register.

Field descriptions

The VBAR_EL2 bit assignments are:



Bits [63:11]

Vector Base Address. Base address of the exception vectors for exceptions taken to EL2.

If the implementation does not support ARMv8.2-LVA, then:

- If tagged addresses are being used, bits [55:48] of VBAR_EL2 must be 0 or else the use of the vector address will result in a recursive exception.
- If tagged addresses are not being used, bits [63:48] of VBAR_EL2 must be 0 or else the use of the vector address will result in a recursive exception.

If the implementation supports ARMv8.2-LVA, then:

- If tagged addresses are being used, bits [55:52] of VBAR_EL2 must be the same or else the use of the vector address will result in a recursive exception.
- If tagged addresses are not being used, bits [63:52] of VBAR_EL2 must be the same or else the use of the vector address will result in a recursive exception.

This field resets to an architecturally UNKNOWN value.

Bits [10:0]

Reserved, RES0.

Accessing the VBAR_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic VBAR_EL2 or VBAR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, VBAR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1100	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return VBAR_EL2;
elsif PSTATE.EL == EL3 then
    return VBAR_EL2;
```

MSR VBAR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1100	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    VBAR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    VBAR_EL2 = X[t];
```

MRS <Xt>, VBAR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1> == '01' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.VBAR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x250];
    else
        return VBAR_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return VBAR_EL2;
    else
        return VBAR_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    return VBAR_EL1;
```

MSR VBAR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1> == '01' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.VBAR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x250] = X[t];
    else
        VBAR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        VBAR_EL2 = X[t];
    else
        VBAR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    VBAR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.139 VBAR_EL3, Vector Base Address Register (EL3)

The VBAR_EL3 characteristics are:

Purpose

Holds the vector base address for any exception that is taken to EL3.

Configurations

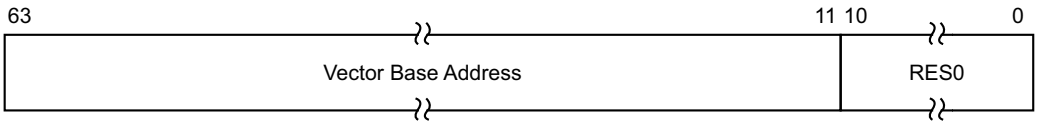
This register is present only when EL3 is implemented. Otherwise, direct accesses to VBAR_EL3 are UNDEFINED.

Attributes

VBAR_EL3 is a 64-bit register.

Field descriptions

The VBAR_EL3 bit assignments are:



Bits [63:11]

Vector Base Address. Base address of the exception vectors for exceptions taken to EL3.

If the implementation does not support ARMv8.2-LVA, then:

- If tagged addresses are being used, bits [55:48] of VBAR_EL3 must be 0 or else the use of the vector address will result in a recursive exception.
- If tagged addresses are not being used, bits [63:48] of VBAR_EL3 must be 0 or else the use of the vector address will result in a recursive exception.

If the implementation supports ARMv8.2-LVA, then:

- If tagged addresses are being used, bits [55:52] of VBAR_EL3 must be the same or else the use of the vector address will result in a recursive exception.
- If tagged addresses are not being used, bits [63:52] of VBAR_EL3 must be the same or else the use of the vector address will result in a recursive exception.

This field resets to an architecturally UNKNOWN value.

Bits [10:0]

Reserved, RES0.

Accessing the VBAR_EL3

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, VBAR_EL3

op0	op1	CRn	CRm	op2
0b11	0b110	0b1100	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    return VBAR_EL3;
```

MSR VBAR_EL3, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b110	0b1100	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    VBAR_EL3 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.140 VMPIDR_EL2, Virtualization Multiprocessor ID Register

The VMPIDR_EL2 characteristics are:

Purpose

Holds the value of the Virtualization Multiprocessor ID. This is the value returned by EL1 reads of MPIDR_EL1.

Configurations

AArch64 System register VMPIDR_EL2[31:0] is architecturally mapped to AArch32 System register VMPIDR[31:0].

If EL2 is not implemented, reads of this register return the value of the MPIDR_EL1 and writes to the register are ignored.

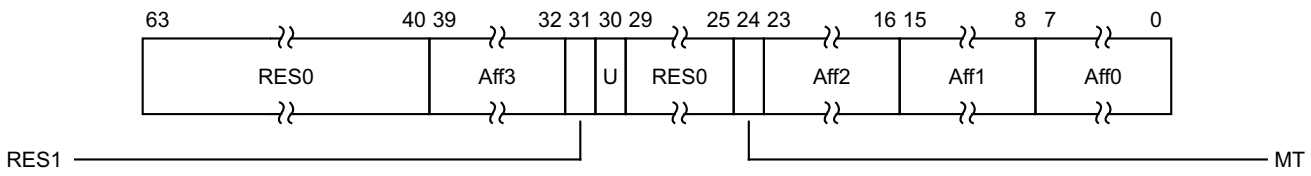
This register has no effect if EL2 is not enabled in the current Security state.

Attributes

VMPIDR_EL2 is a 64-bit register.

Field descriptions

The VMPIDR_EL2 bit assignments are:



Bits [63:40]

Reserved, RES0.

Aff3, bits [39:32]

Affinity level 3. See the description of VMPIDR_EL2.Aff0 for more information.

Aff3 is not supported in AArch32 state.

This field resets to an architecturally UNKNOWN value.

Bit [31]

Reserved, RES1.

U, bit [30]

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system.

`0b0` Processor is part of a multiprocessor system.

`0b1` Processor is part of a uniprocessor system.

This field resets to an architecturally UNKNOWN value.

Bits [29:25]

Reserved, RES0.

MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using a multithreading type approach. See the description of VMPIDR_EL2.Aff0 for more information about affinity levels.

`0b0` Performance of PEs at the lowest affinity level is largely independent.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` Performance of PEs at the lowest affinity level is very interdependent.

This field resets to an architecturally UNKNOWN value.

Aff2, bits [23:16]

Affinity level 2. See the description of VMPIDR_EL2.Aff0 for more information.

This field resets to an architecturally UNKNOWN value.

Aff1, bits [15:8]

Affinity level 1. See the description of VMPIDR_EL2.Aff0 for more information.

This field resets to an architecturally UNKNOWN value.

Aff0, bits [7:0]

Affinity level 0. This is the affinity level that is most significant for determining PE behavior. Higher affinity levels are increasingly less significant in determining PE behavior. The assigned value of the MPIDR.{Aff2, Aff1, Aff0} or MPIDR_EL1.{Aff3, Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

This field resets to an architecturally UNKNOWN value.

Accessing the VMPIDR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, VMPIDR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0000	0b0000	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        return NVMem[0x050];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return VMPIDR_EL2;
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        return MPIDR_EL1;
    else
        return VMPIDR_EL2;
```

MSR VMPIDR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0000	0b0000	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NVMem[0x050] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    VMPIDR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        //no operation
    else
        VMPIDR_EL2 = X[t];
```

MRS <Xt>, MPIDR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0000	0b101

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.MPIDR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) then
        return VMPIDR_EL2;
    else
        return MPIDR_EL1;
elsif PSTATE.EL == EL2 then
    return MPIDR_EL1;
elsif PSTATE.EL == EL3 then
    return MPIDR_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.141 VNCR_EL2, Virtual Nested Control Register

The VNCR_EL2 characteristics are:

Purpose

When ARMv8.4-NV is implemented, holds the base address that is used to define the memory location that is accessed by transformed reads and writes of System registers.

Configurations

This register is present only when ARMv8.4-NV is implemented. Otherwise, direct accesses to VNCR_EL2 are UNDEFINED.

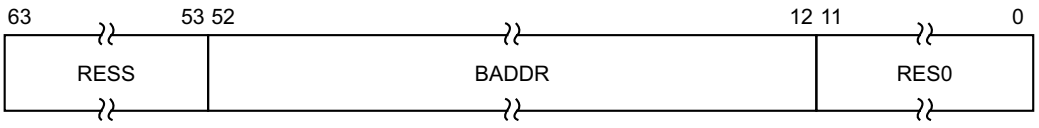
This register has no effect if EL2 is not enabled in the current Security state.

Attributes

VNCR_EL2 is a 64-bit register.

Field descriptions

The VNCR_EL2 bit assignments are:



RESS, bits [63:53]

Reserved, Sign extended. If the bits marked as RESS do not all have the same value, then there is a CONSTRAINED UNPREDICTABLE choice between:

- Generating an EL2 translation regime Translation abort on use of the VNCR_EL2 register.
- Bits[63:49] of VNCR_EL2 are treated as the same value as bit[48] for all purposes other than reading back the register.
- Bits[63:49] of VNCR_EL2 are treated as the same value as bit[48] for all purposes.
- If the virtual address space for EL2 supports more than 48 bits, bits[63:53] of VNCR_EL2 are treated as the same value as bit[52] for all purposes other than reading back the register.
- If the virtual address space for EL2 supports more than 48 bits, bits[63:53] of VNCR_EL2 are treated as the same value as bit[52].

Where the EL2 translation regime has upper and lower address ranges, bit[52] is used to select between those address ranges to determine if the address space supports more than 48 bits.

BADDR, bits [52:12]

Base Address. If the virtual address space for EL2 does not support more than 48 bits, then bits [52:49] are RESS.

When a register read/write is transformed to be a Load or Store, the address of the load/store is to SignOffset(VNCR_EL2.BADDR:Offset<11:0>, 64).

This field resets to an architecturally UNKNOWN value.

Bits [11:0]

Reserved, RES0.

Accessing the VNCR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, VNCR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        return NVMem[0x0B0];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return VNCR_EL2;
elsif PSTATE.EL == EL3 then
    return VNCR_EL2;
```

MSR VNCR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NVMem[0x0B0] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    VNCR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    VNCR_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.142 VPIDR_EL2, Virtualization Processor ID Register

The VPIDR_EL2 characteristics are:

Purpose

Holds the value of the Virtualization Processor ID. This is the value returned by EL1 reads of MIDR_EL1.

Configurations

AArch64 System register VPIDR_EL2[31:0] is architecturally mapped to AArch32 System register VPIDR[31:0].

If EL2 is not implemented, reads of this register return the value of the MIDR_EL1 and writes to the register are ignored.

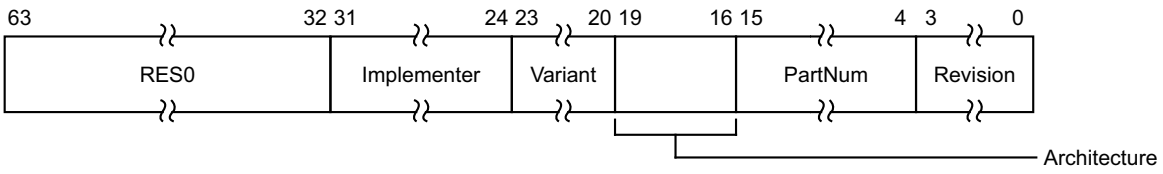
This register has no effect if EL2 is not enabled in the current Security state.

Attributes

VPIDR_EL2 is a 64-bit register.

Field descriptions

The VPIDR_EL2 bit assignments are:



Bits [63:32]

Reserved, RES0.

Implementer, bits [31:24]

The Implementer code. This field must hold an implementer code that has been assigned by Arm. Assigned codes include the following:

Hex representation	ASCII representation	Implementer
0x41	A	Arm Limited
0x42	B	Broadcom Corporation
0x43	C	Cavium Inc.
0x44	D	Digital Equipment Corporation
0x49	I	Infineon Technologies AG
0x4D	M	Motorola or Freescale Semiconductor Inc.
0x4E	N	NVIDIA Corporation
0x50	P	Applied Micro Circuits Corporation
0x51	Q	Qualcomm Inc.
0x56	V	Marvell International Ltd.
0x69	i	Intel Corporation

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Arm can assign codes that are not published in this manual. All values not assigned by Arm are reserved and must not be used.

This field resets to an architecturally UNKNOWN value.

Variant, bits [23:20]

An IMPLEMENTATION DEFINED variant number. Typically, this field is used to distinguish between different product variants, or major revisions of a product.

This field resets to an architecturally UNKNOWN value.

Architecture, bits [19:16]

The permitted values of this field are:

`0b0001` Armv4.

`0b0010` Armv4T.

`0b0011` Armv5 (obsolete).

`0b0100` Armv5T.

`0b0101` Armv5TE.

`0b0110` Armv5TEJ.

`0b0111` Armv6.

`0b1111` Architectural features are individually identified in the ID_* registers, see *ID registers* on page K14-8060.

All other values are reserved.

This field resets to an architecturally UNKNOWN value.

PartNum, bits [15:4]

An IMPLEMENTATION DEFINED primary part number for the device.

On processors implemented by Arm, if the top four bits of the primary part number are `0x0` or `0x7`, the variant and architecture are encoded differently.

This field resets to an architecturally UNKNOWN value.

Revision, bits [3:0]

An IMPLEMENTATION DEFINED revision number for the device.

This field resets to an architecturally UNKNOWN value.

Accessing the VPIDR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, VPIDR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0000	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        return NVMem[0x088];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return VPIDR_EL2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        return MIDR_EL1;
    else
        return VPIDR_EL2;
```

MSR VPIDR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0000	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NVMem[0x088] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    VPIDR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        //no operation
    else
        VPIDR_EL2 = X[t];
```

MRS <Xt>, MIDR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0000	0b000

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("ARMv8.4-IDST") then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.MIDR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) then
        return VPIDR_EL2;
    else
        return MIDR_EL1;
elsif PSTATE.EL == EL2 then
    return MIDR_EL1;
elsif PSTATE.EL == EL3 then
    return MIDR_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.143 VSTCR_EL2, Virtualization Secure Translation Control Register

The VSTCR_EL2 characteristics are:

Purpose

The control register for stage 2 of the Secure EL1&0 translation regime.

Configurations

This register is present only when ARMv8.4-SecEL2 is implemented. Otherwise, direct accesses to VSTCR_EL2 are UNDEFINED.

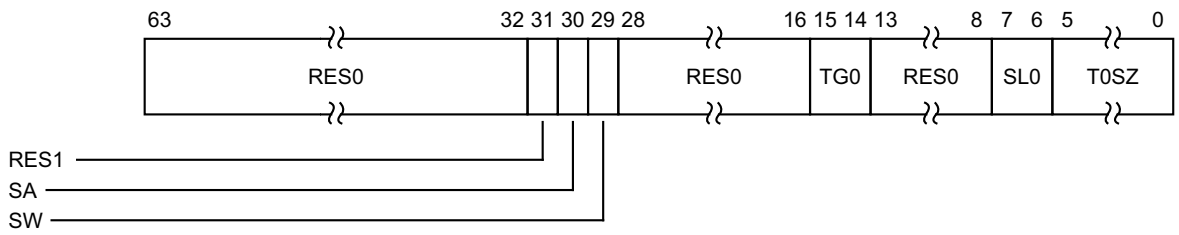
This register has no effect if EL2 is not enabled in the current Security state.

Attributes

VSTCR_EL2 is a 64-bit register.

Field descriptions

The VSTCR_EL2 bit assignments are:



Any of the bits in VSTCR_EL2 are permitted to be cached in a TLB.

Bits [63:32]

Reserved, RES0.

Bit [31]

Reserved, RES1.

SA, bit [30]

Secure stage 2 translation output address space.

`0b0` All stage 2 translations for the Secure IPA space access the Secure PA space.

`0b1` All stage 2 translations for the Secure IPA space access the Non-secure PA space.

When the value of VSTCR_EL2.SW is 1, this bit behaves as 1 for all purposes other than reading back the value of the bit.

This field resets to an architecturally UNKNOWN value.

SW, bit [29]

Secure stage 2 translation address space.

`0b0` All stage 2 translation table walks for the Secure IPA space are to the Secure PA space.

`0b1` All stage 2 translation table walks for the Secure IPA space are to the Non-secure PA space.

This field resets to an architecturally UNKNOWN value.

Bits [28:16]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TG0, bits [15:14]

Secure stage 2 granule size for VSTTBR_EL2.

`0b00` 4KB.

`0b01` 64KB.

`0b10` 16KB.

Other values are reserved.

If ARMv8.5-GTG is implemented, ID_AA64MMFR0_EL1.{TGran4_2, TGran16_2, TGran64_2} indicate which granule sizes are supported for Level 2 translation.

If ARMv8.5-GTG is not implemented, ID_AA64MMFR0_EL1.{TGran4, TGran16, TGran64} indicate which granule sizes are supported.

If the value is programmed to either a reserved value, or a size that has not been implemented, then for all purposes other than read back from this register, the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

This field resets to an architecturally UNKNOWN value.

Bits [13:8]

Reserved, RES0.

SL0, bits [7:6]

When ARMv8.4-TTST is implemented:

Starting level of the Secure stage 2 translation lookup, controlled by VSTCR_EL2. The meaning of this field depends on the value of VSTCR_EL2.TG0.

`0b00` If VSTCR_EL2.TG0 is `0b00` (4KB granule), start at level 2. If VSTCR_EL2.TG0 is `0b10` (16KB granule) or `0b01` (64KB granule), start at level 3.

`0b01` If VSTCR_EL2.TG0 is `0b00` (4KB granule), start at level 1. If VSTCR_EL2.TG0 is `0b10` (16KB granule) or `0b01` (64KB granule), start at level 2.

`0b10` If VSTCR_EL2.TG0 is `0b00` (4KB granule), start at level 0. If VSTCR_EL2.TG0 is `0b10` (16KB granule) or `0b01` (64KB granule), start at level 1.

`0b11` If VSTCR_EL2.TG0 is `0b00` (4KB granule), start at level 3.

All other values are reserved. If this field is programmed to a reserved value, or to a value that is not consistent with the programming of VSTCR_EL2.T0SZ, then a stage 2 level 0 Translation fault is generated.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Starting level of the Secure stage 2 translation lookup, controlled by VSTCR_EL2. The meaning of this field depends on the value of VSTCR_EL2.TG0.

`0b00` If VSTCR_EL2.TG0 is `0b00` (4KB granule), start at level 2. If VSTCR_EL2.TG0 is `0b10` (16KB granule) or `0b01` (64KB granule), start at level 3.

`0b01` If VSTCR_EL2.TG0 is `0b00` (4KB granule), start at level 1. If VSTCR_EL2.TG0 is `0b10` (16KB granule) or `0b01` (64KB granule), start at level 2.

`0b10` If VSTCR_EL2.TG0 is `0b00` (4KB granule), start at level 0. If VSTCR_EL2.TG0 is `0b10` (16KB granule) or `0b01` (64KB granule), start at level 1.

All other values are reserved. If this field is programmed to a reserved value, or to a value that is not consistent with the programming of VSTCR_EL2.T0SZ, then a stage 2 level 0 Translation fault is generated.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T0SZ, bits [5:0]

The size offset of the memory region addressed by VSTTBR_EL2. The region size is $2^{(64-T0SZ)}$ bytes.

The maximum and minimum possible values for this field depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.

If this field is programmed to a value that is not consistent with the programming of SL0, then a stage 2 level 0 Translation fault is generated.

This field resets to an architecturally UNKNOWN value.

Accessing the VSTCR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, VSTCR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0110	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && SCR_EL3.NS == '1' then
        UNDEFINED;
    elsif EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        return NVMem[0x048];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && SCR_EL3.NS == '1' then
        UNDEFINED;
    else
        return VSTCR_EL2;
elsif PSTATE.EL == EL3 then
    if SCR_EL3.EEL2 == '0' then
        UNDEFINED;
    else
        return VSTCR_EL2;
```

MSR VSTCR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0110	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && SCR_EL3.NS == '1' then
        UNDEFINED;
    elsif EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NVMem[0x048] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && SCR_EL3.NS == '1' then
        UNDEFINED;
    else
        VSTCR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    if SCR_EL3.EEL2 == '0' then
        UNDEFINED;
    else
        VSTCR_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.144 VSTTBR_EL2, Virtualization Secure Translation Table Base Register

The VSTTBR_EL2 characteristics are:

Purpose

The base register for stage 2 of the Secure EL1&0 translation regime. Holds the base address of the translation table for the initial lookup for stage 2 of an address translation in the Secure EL1&0 translation regime, and other information for this translation stage.

Configurations

This register is present only when ARMv8.4-SecEL2 is implemented. Otherwise, direct accesses to VSTTBR_EL2 are UNDEFINED.

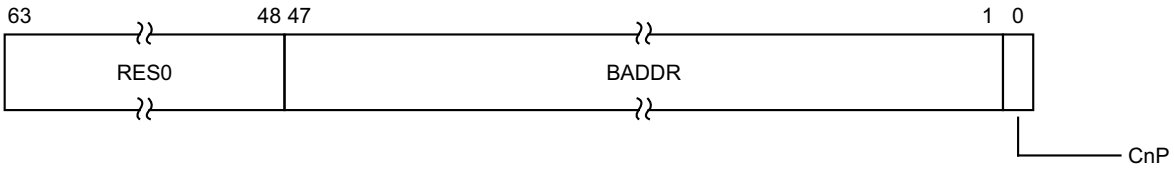
This register has no effect if EL2 is not enabled in the current Security state.

Attributes

VSTTBR_EL2 is a 64-bit register.

Field descriptions

The VSTTBR_EL2 bit assignments are:



Any of the bits in VSTTBR_EL2 are permitted to be cached in a TLB.

Bits [63:48]

Reserved, RES0.

BADDR, bits [47:1]

Translation table base address, A[47:x] or A[51:x].

Note

- Translation table base addresses of 52 bits, A[51:x], are supported only in an implementation that includes ARMv8.2-LPA and is using the 64KB translation granule.
- A translation table must be aligned to the size of the table, except that when using a translation table base address larger than 48 bits the minimum alignment of a table containing fewer than eight entries is 64 bytes.

If the value of VTCR_EL2.PS is `0b110`, then:

- Register bits[47:z] hold bits[47:z] of the stage 1 translation table base address, where z is determined as follows:
  - If x >= 6 then z=x.
  - Otherwise, z=6.
- Register bits[5:2] hold bits[51:48] of the stage 1 translation table base address.
- When z>x register bits[(z-1):x] are RES0, and bits[(z-1):x] of the translation table base address are zero.
- When x>6 register bits[(x-1):6] are RES0.
- Register bit[1] is RES0.
- Bits[5:2] of the stage 1 translation table base address are zero.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

When the value of ID_AA64MMFR0_EL1.PARange indicates that the implementation does not support a 52 bit PA size, if a translation table lookup uses this register with the 64KB translation granule when the value of VTCR_EL2.PS is `0b110` and the value of register bits[5:2] is nonzero, an Address size fault is generated.

If the Effective value of VTCR_EL2.PS is not `0b110` then:

- Register bits[47:x] hold bits[47:x] of the stage 1 translation table base address.
- Register bits[(x-1):1] are RES0.
- If the implementation supports 52-bit PAs and IPAs then bits[51:48] of the translation table base addresses used in this stage of translation are `0b0000`.

If any VSTTBR_EL2[47:1] bit that is defined as RES0 has the value 1 when a translation table walk is performed using VSTTBR_EL2, then the translation table base address might be misaligned, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Bits[x-1:0] of the translation table base address are treated as if all the bits are zero. The value read back from the corresponding register bits is either the value written to the register or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of VSTCR_EL2.T0SZ, the stage of translation, and the translation granule size.

This field resets to an architecturally UNKNOWN value.

CnP, bit [0]

Common not Private, for stage 2 of the Secure EL1&0 translation regime. In an implementation that includes ARMv8.2-TTCNP, indicates whether each entry that is pointed to by VSTTBR_EL2 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of VSTTBR_EL2.CnP is 1.

`0b0` The translation table entries pointed to by VSTTBR_EL2 are permitted to differ from the entries for VSTTBR_EL2 for other PEs in the Inner Shareable domain. This is not affected by the value of the current VMID.

`0b1` The translation table entries pointed to by VSTTBR_EL2 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of VSTTBR_EL2.CnP is 1 and the VMID is the same as the current VMID.

Note

If the value of VSTTBR_EL2.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those VSTTBR_EL2s do not point to the same translation table entries when using the current VMID, then the results of translations using VSTTBR_EL2 are CONSTRAINED UNPREDICTABLE, see CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values on page K1-6254.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to an architecturally UNKNOWN value.

Accessing the VSTTBR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, VSTTBR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0110	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && SCR_EL3.NS == '1' then
        UNDEFINED;
    elsif EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        return NVMem[0x030];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && SCR_EL3.NS == '1' then
        UNDEFINED;
    else
        return VSTTBR_EL2;
elsif PSTATE.EL == EL3 then
    if SCR_EL3.EEL2 == '0' then
        UNDEFINED;
    else
        return VSTTBR_EL2;
```

MSR VSTTBR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0110	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && SCR_EL3.NS == '1' then
        UNDEFINED;
    elsif EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NVMem[0x030] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && SCR_EL3.NS == '1' then
        UNDEFINED;
    else
        VSTTBR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    if SCR_EL3.EEL2 == '0' then
        UNDEFINED;
    else
        VSTTBR_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.145 VTCR_EL2, Virtualization Translation Control Register

The VTCR_EL2 characteristics are:

Purpose

The control register for stage 2 of the EL1&0 translation regime.

Configurations

AArch64 System register VTCR_EL2[31:0] is architecturally mapped to AArch32 System register VTCR[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

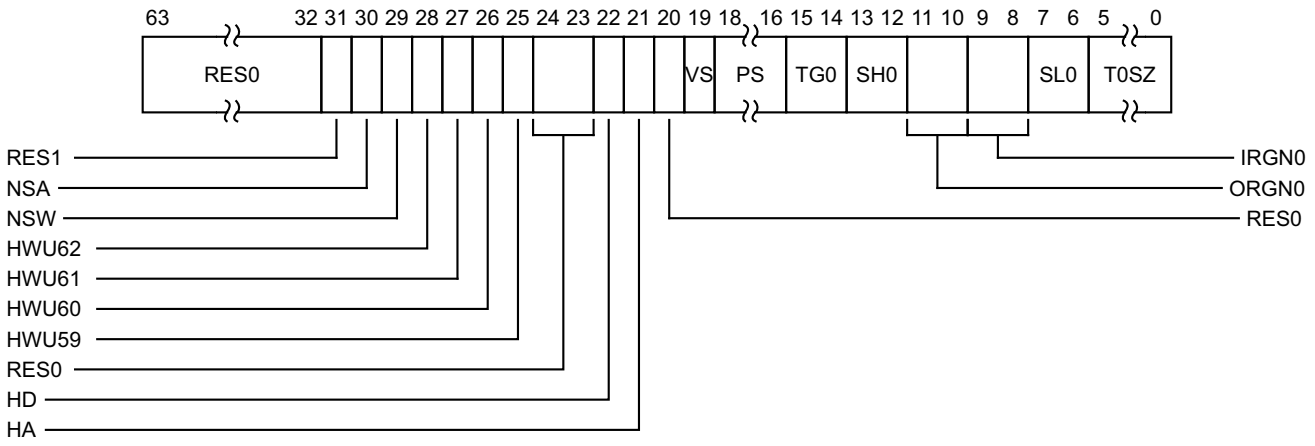
This register has no effect if EL2 is not enabled in the current Security state.

Attributes

VTCR_EL2 is a 64-bit register.

Field descriptions

The VTCR_EL2 bit assignments are:



Any of the bits in VTCR_EL2 are permitted to be cached in a TLB.

Bits [63:32]

Reserved, RES0.

Bit [31]

Reserved, RES1.

NSA, bit [30]

When ARMv8.4-SecEL2 is implemented:

Non-secure stage 2 translation output address space.

`0b0` All stage 2 translations for the Non-secure IPA space of the Secure EL1&0 translation regime access the Secure PA space.

`0b1` All stage 2 translations for the Non-secure IPA space of the Secure EL1&0 translation regime access the Non-secure PA space.

This bit behaves as 1 for all purposes other than reading back the value of the bit when one of the following is true:

- The PE is executing in Non-secure state.
- The value of VTCR_EL2.NSW is 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The value of VSTCR_EL2.SA is 1.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSW, bit [29]

When ARMv8.4-SecEL2 is implemented:

Non-secure stage 2 translation table address space.

`0b0` All stage 2 translation table walks for the Non-secure IPA space of the Secure EL1&0 translation regime are to the Secure PA space.

`0b1` All stage 2 translation table walks for the Non-secure IPA space of the Secure EL1&0 translation regime are to the Non-secure PA space.

When the PE is executing in Non-secure state, this bit behaves as 1 for all purposes other than reading back the value of the bit.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU62, bit [28]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 2 translation table Block or Page entry.

`0b0` Bit[62] of each stage 2 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` Bit[62] of each stage 2 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU61, bit [27]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 2 translation table Block or Page entry.

`0b0` Bit[61] of each stage 2 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` Bit[61] of each stage 2 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU60, bit [26]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 2 translation table Block or Page entry.

`0b0` Bit[60] of each stage 2 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` Bit[60] of each stage 2 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU59, bit [25]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 2 translation table Block or Page entry.

`0b0` Bit[59] of each stage 2 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` Bit[59] of each stage 2 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [24:23]

Reserved, RES0.

HD, bit [22]

When ARMv8.1-TTHM is implemented:

Hardware management of dirty state in stage 2 translations when EL2 is enabled in the current Security state.

`0b0` Stage 2 hardware management of dirty state disabled.

`0b1` Stage 2 hardware management of dirty state enabled, only if the VTCR_EL2.HA bit is also set to 1.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HA, bit [21]

When ARMv8.1-TTHM is implemented:

Hardware Access flag update in Non-secure and Secure stage 2 translations when EL2 is enabled in the current Security state.

`0b0` Stage 2 Access flag update disabled.

`0b1` Stage 2 Access flag update enabled.

Otherwise:

Reserved, RES0.

Bit [20]

Reserved, RES0.

VS, bit [19]

When ARMv8.1-VMID16 is implemented:

VMID Size.

`0b0` 8 bit - the upper 8 bits of VTTBR_EL2 and VSTTBR_EL2 are ignored by the hardware, and treated as if they are all zeros, for every purpose except when reading back the register.

`0b1` 16 bit - the upper 8 bits of VTTBR_EL2 and VSTTBR_EL2 are used for allocation and matching in the TLB.

If the implementation only supports an 8-bit VMID, this field is RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PS, bits [18:16]

Physical address Size for the Second Stage of translation.

`0b000` 32 bits, 4GB.

`0b001` 36 bits, 64GB.

`0b010` 40 bits, 1TB.

`0b011` 42 bits, 4TB.

`0b100` 44 bits, 16TB.

`0b101` 48 bits, 256TB.

`0b110` 52 bits, 4PB.

Other values are reserved.

The reserved values behave in the same way as the `0b101` or `0b110` encoding, but software must not rely on this property as the behavior of the reserved values might change in a future revision of the architecture.

The value `0b110` is permitted only if ARMv8.2-LPA is implemented and the translation granule size is 64KB.

In an implementation that supports 52-bit PAs, if the value of this field is not `0b110` or a value treated as `0b110`, then bits[51:48] of every translation table base address for the stage of translation controlled by VTCR_EL2 are `0b0000`.

This field resets to an architecturally UNKNOWN value.

TG0, bits [15:14]

Granule size for the VTTBR_EL2.

`0b00` 4KB.

`0b01` 64KB.

`0b10` 16KB.

Other values are reserved.

If ARMv8.5-GTG is implemented, ID_AA64MMFR0_EL1.{TGran4_2, TGran16_2, TGran64_2} indicate which granule sizes are supported for Level 2 translation.

If ARMv8.5-GTG is not implemented, ID_AA64MMFR0_EL1.{TGran4, TGran16, TGran64} indicate which granule sizes are supported.

If the value is programmed to either a reserved value, or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

This field resets to an architecturally UNKNOWN value.

SH0, bits [13:12]

Shareability attribute for memory associated with translation table walks using VTTBR_EL2 or VSTTBR_EL2.

`0b00` Non-shareable.

`0b10` Outer Shareable.

`0b11` Inner Shareable.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-7871.

This field resets to an architecturally UNKNOWN value.

ORGN0, bits [11:10]

Outer cacheability attribute for memory associated with translation table walks using VTTBR_EL2 or VSTTBR_EL2.

`0b00` Normal memory, Outer Non-cacheable.

`0b01` Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.

`0b10` Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.

`0b11` Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.

This field resets to an architecturally UNKNOWN value.

IRGN0, bits [9:8]

Inner cacheability attribute for memory associated with translation table walks using VTTBR_EL2 or VSTTBR_EL2.

`0b00` Normal memory, Inner Non-cacheable.

`0b01` Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.

`0b10` Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.

`0b11` Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

This field resets to an architecturally UNKNOWN value.

SL0, bits [7:6]

When ARMv8.4-TTST is implemented:

Starting level of the Secure stage 2 translation lookup, controlled by VTCR_EL2. The meaning of this field depends on the value of VTCR_EL2.TG0.

`0b00` If VTCR_EL2.TG0 is `0b00` (4KB granule), start at level 2. If VTCR_EL2.TG0 is `0b10` (16KB granule) or `0b01` (64KB granule), start at level 3.

`0b01` If VTCR_EL2.TG0 is `0b00` (4KB granule), start at level 1. If VTCR_EL2.TG0 is `0b10` (16KB granule) or `0b01` (64KB granule), start at level 2.

`0b10` If VTCR_EL2.TG0 is `0b00` (4KB granule), start at level 0. If VTCR_EL2.TG0 is `0b10` (16KB granule) or `0b01` (64KB granule), start at level 1.

`0b11` If VTCR_EL2.TG0 is `0b00` (4KB granule), start at level 3.

All other values are reserved. If this field is programmed to a reserved value, or to a value that is not consistent with the programming of VTCR_EL2.T0SZ, then a stage 2 level 0 Translation fault is generated.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Starting level of the Secure stage 2 translation lookup, controlled by VTCR_EL2. The meaning of this field depends on the value of VTCR_EL2.TG0.

`0b00` If VTCR_EL2.TG0 is `0b00` (4KB granule), start at level 2. If VTCR_EL2.TG0 is `0b10` (16KB granule) or `0b01` (64KB granule), start at level 3.

`0b01` If VTCR_EL2.TG0 is `0b00` (4KB granule), start at level 1. If VTCR_EL2.TG0 is `0b10` (16KB granule) or `0b01` (64KB granule), start at level 2.

`0b10` If VTCR_EL2.TG0 is `0b00` (4KB granule), start at level 0. If VTCR_EL2.TG0 is `0b10` (16KB granule) or `0b01` (64KB granule), start at level 1.

All other values are reserved. If this field is programmed to a reserved value, or to a value that is not consistent with the programming of VTCR_EL2.T0SZ, then a stage 2 level 0 Translation fault is generated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

T0SZ, bits [5:0]

The size offset of the memory region addressed by VTTBR_EL2. The region size is $2^{(64-T0SZ)}$ bytes.

The maximum and minimum possible values for T0SZ depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.

If this field is programmed to a value that is not consistent with the programming of SL0 then a stage 2 level 0 Translation fault is generated.

This field resets to an architecturally UNKNOWN value.

Accessing the VTCR_EL2

Any of the bits in VTCR_EL2 are permitted to be cached in a TLB.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, VTCR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0001	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        return NVMem[0x040];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return VTCR_EL2;
elsif PSTATE.EL == EL3 then
    return VTCR_EL2;
```

MSR VTCR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0001	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NVMem[0x040] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    VTCR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    VTCR_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.2.146 VTTBR_EL2, Virtualization Translation Table Base Register

The VTTBR_EL2 characteristics are:

Purpose

Holds the base address of the translation table for the initial lookup for stage 2 of an address translation in the EL1&0 translation regime, and other information for this translation regime.

Configurations

AArch64 System register VTTBR_EL2[63:0] is architecturally mapped to AArch32 System register VTTBR[63:0].

If EL2 is not implemented, this register is RES0 from EL3.

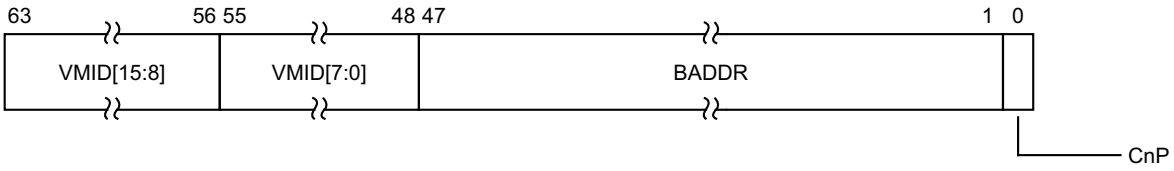
This register has no effect if EL2 is not enabled in the current Security state.

Attributes

VTTBR_EL2 is a 64-bit register.

Field descriptions

The VTTBR_EL2 bit assignments are:



VMID[15:8], bits [63:56]

When ARMv8.1-VMID16 is implemented and VTCR_EL2.VS == 1:

Extension to VMID[7:0]. See VTTBR_EL2.VMID[7:0] for more details.

If EL2 is using AArch32, or if the implementation has an 8-bit VMID, this field is RES0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

VMID[7:0], bits [55:48]

The VMID for the translation table.

The VMID is 8 bits when any of the following are true:

- EL2 is using AArch32.
- The VTCR_EL2.VS is 0.
- ARMv8.1-VMID16 is not implemented.

This field resets to an architecturally UNKNOWN value.

BADDR, bits [47:1]

Translation table base address, A[47:x] or A[51:x], bits[47:1].

Note

- Translation table base addresses of 52 bits, A[51:x], are supported only in an implementation that includes ARMv8.2-LPA and is using the 64KB translation granule.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- A translation table must be aligned to the size of the table, except that when using a translation table base address larger than 48 bits the minimum alignment of a table containing fewer than eight entries is 64 bytes.

In an implementation that includes ARMv8.2-LPA, if the value of VTCR_EL2.PS is `0b110`, then:

- Register bits[47:z] hold bits[47:z] of the stage 1 translation table base address, where z is determined as follows:
  - If x >= 6 then z=x.
  - Otherwise, z=6.
- Register bits[5:2] hold bits[51:48] of the stage 1 translation table base address.
- When z>x register bits[(z-1):x] are RES0, and bits[(z-1):x] of the translation table base address are zero.
- When x>6 register bits[(x-1):6] are RES0.
- Register bit[1] is RES0.
- Bits[5:2] of the stage 1 translation table base address are zero.
- In an implementation that includes ARMv8.2-TTCNP, bit[0] of the stage 1 translation table base address is zero.

Note

- In an implementation that includes ARMv8.2-LPA a VTCR_EL2.PS value of `0b110`, that selects a PA size of 52 bits, is permitted only when using the 64KB translation granule.
- When the value of ID_AA64MMFR0_EL1.PARange indicates that the implementation does not support a 52 bit PA size, if a translation table lookup uses this register with the 64KB translation granule when the value of VTCR_EL2.PS is `0b110` and the value of register bits[5:2] is nonzero, an Address size fault is generated.

If the Effective value of VTCR_EL2.PS is not `0b110` then:

- Register bits[47:x] hold bits[47:x] of the stage 1 translation table base address.
- Register bits[(x-1):1] are RES0.
- If the implementation supports 52-bit PAs and IPAs then bits[51:48] of the translation table base addresses used in this stage of translation are `0b0000`.

Note

This definition applies:

- To an implementation that includes ARMv8.2-LPA and is using a translation granule smaller than 64KB.
- To any implementation that does not include ARMv8.2-LPA.

If any VTTBR_EL2[47:0] bit that is defined as RES0 has the value 1 when a translation table walk is performed using VTTBR_EL2, then the translation table base address might be misaligned, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Bits[x-1:0] of the translation table base address are treated as if all the bits are zero. The value read back from the corresponding register bits is either the value written to the register or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of VTCR_EL2.T0SZ, the stage of translation, and the translation granule size.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CnP, bit [0]

When ARMv8.2-TTCNP is implemented:

Common not Private. This bit indicates whether each entry that is pointed to by VTTBR_EL2 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of VTTBR_EL2.CnP is 1.

`0b0` The translation table entries pointed to by VTTBR_EL2 are permitted to differ from the entries for VTTBR_EL2 for other PEs in the Inner Shareable domain. This is not affected by the value of the current VMID.

`0b1` The translation table entries pointed to by VTTBR_EL2 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of VTTBR_EL2.CnP is 1 and the VMID is the same as the current VMID.

This field is permitted to be cached in a TLB.

Note

If the value of VTTBR_EL2.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those VTTBR_EL2s do not point to the same translation table entries when using the current VMID then the results of translations using VTTBR_EL2 are CONSTRAINED UNPREDICTABLE, see *CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values* on page K1-7856.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing the VTTBR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, VTTBR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        return NVMem[0x020];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return VTTBR_EL2;
elsif PSTATE.EL == EL3 then
    return VTTBR_EL2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR VTTBR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NVMem[0x020] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    VTTBR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    VTTBR_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.3 Debug registers

This section lists the Debug System registers in AArch64 state, in alphabetic order:

- The principal encoding space for debug registers is `op0==0b10`, `op1==`{0, 3, 4}. *Instructions for accessing debug System registers* on page D12-2851 summarizes the registers in this encoding space and lists them in order of their encodings.
- In addition, the following registers in the `op0==0b11` encoding space are classified as Debug registers:
  - DLR_EL0.
  - DSPSR_EL0.
  - MDCR_EL2.
  - MDCR_EL3.
  - SDER32_EL3.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.3.1 DBGAUTHSTATUS_EL1, Debug Authentication Status register

The DBGAUTHSTATUS_EL1 characteristics are:

Purpose

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for debug.

Configurations

AArch64 System register DBGAUTHSTATUS_EL1[31:0] is architecturally mapped to AArch32 System register DBGAUTHSTATUS[31:0].

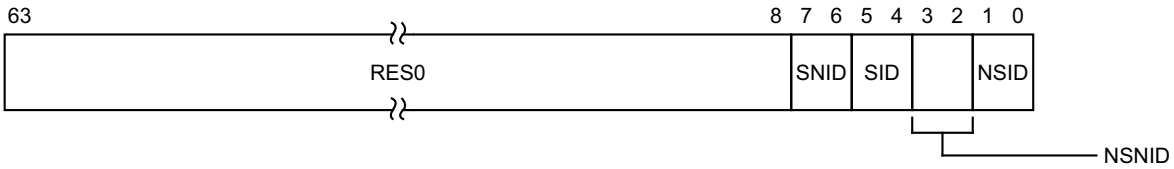
AArch64 System register DBGAUTHSTATUS_EL1[31:0] is architecturally mapped to External register DBGAUTHSTATUS_EL1[31:0].

Attributes

DBGAUTHSTATUS_EL1 is a 64-bit register.

Field descriptions

The DBGAUTHSTATUS_EL1 bit assignments are:



Bits [63:8]

Reserved, RES0.

SNID, bits [7:6]

When ARMv8.4-Debug is implemented:

Secure non-invasive debug.

This field has the same value as DBGAUTHSTATUS_EL1.SID.

Otherwise:

Secure non-invasive debug.

`0b00` Not implemented. EL3 is not implemented and the Effective value of SCR_EL3.NS is 1.

`0b10` Implemented and disabled. ExternalSecureNoninvasiveDebugEnabled() == FALSE.

`0b11` Implemented and enabled. ExternalSecureNoninvasiveDebugEnabled() == TRUE.

All other values are reserved.

SID, bits [5:4]

Secure invasive debug.

`0b00` Not implemented. EL3 is not implemented and the Effective value of SCR_EL3.NS is 1.

`0b10` Implemented and disabled. ExternalSecureInvasiveDebugEnabled() == FALSE.

`0b11` Implemented and enabled. ExternalSecureInvasiveDebugEnabled() == TRUE.

All other values are reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

NSNID, bits [3:2]

When ARMv8.4-Debug is implemented:

Non-secure non-invasive debug.

`0b00` Not implemented. EL3 is not implemented and the Effective value of SCR_EL3.NS is 0

`0b11` Implemented and enabled. EL3 is implemented or the Effective value of SCR_EL3.NS is 1.

All other values are reserved.

Otherwise:

Non-secure non-invasive debug.

`0b00` Not implemented. EL3 is not implemented and the Effective value of SCR_EL3.NS is 0.

`0b10` Implemented and disabled. ExternalNoninvasiveDebugEnabled() == FALSE.

`0b11` Implemented and enabled. ExternalNoninvasiveDebugEnabled() == TRUE.

All other values are reserved.

NSID, bits [1:0]

Non-secure invasive debug.

`0b00` Not implemented. EL3 is not implemented and the Effective value of SCR_EL3.NS is 0.

`0b10` Implemented and disabled. ExternalInvasiveDebugEnabled() == FALSE.

`0b11` Implemented and enabled. ExternalInvasiveDebugEnabled() == TRUE.

All other values are reserved.

Accessing the DBGAUTHSTATUS_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, DBGAUTHSTATUS_EL1

op0	op1	CRn	CRm	op2
0b10	0b000	0b0111	0b1110	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.DBGAUTHSTATUS_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return DBGAUTHSTATUS_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return DBGAUTHSTATUS_EL1;
elsif PSTATE.EL == EL3 then
    return DBGAUTHSTATUS_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.3.2 DBGBCR<n>_EL1, Debug Breakpoint Control Registers, n = 0 - 15

The DBGBCR<n>_EL1 characteristics are:

Purpose

Holds control information for a breakpoint. Forms breakpoint n together with value register DBGBVR<n>_EL1.

Configurations

AArch64 System register DBGBCR<n>_EL1[31:0] is architecturally mapped to AArch32 System register DBGBCR<n>[31:0].

AArch64 System register DBGBCR<n>_EL1[31:0] is architecturally mapped to External register DBGBCR<n>_EL1[31:0].

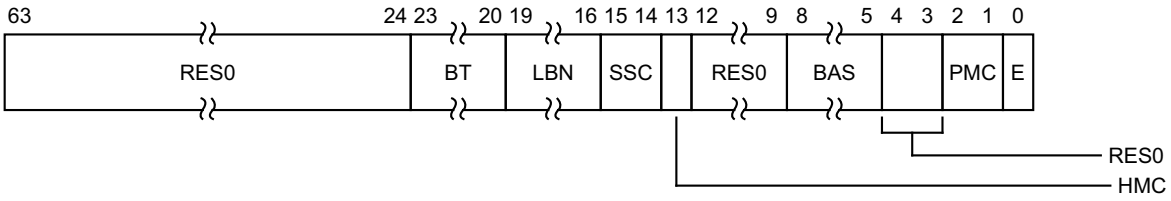
If breakpoint n is not implemented then accesses to this register are UNDEFINED.

Attributes

DBGBCR<n>_EL1 is a 64-bit register.

Field descriptions

The DBGBCR<n>_EL1 bit assignments are:



Bits [63:24]

Reserved, RES0.

BT, bits [23:20]

Breakpoint Type. Possible values are:

`0b0000` Unlinked instruction address match. DBGBVR<n>_EL1 is the address of an instruction.

`0b0001` As `0b0000`, but linked to a Context matching breakpoint.

`0b0010` Unlinked Context ID match. When ARMv8.1-VHE is implemented, EL2 is using AArch64, and the Effective value of HCR_EL2.E2H is 1, if either the PE is executing at EL0 with HCR_EL2.TGE set to 1 or the PE is executing at EL2, then DBGBVR<n>_EL1.ContextID must match the CONTEXTIDR_EL2 value. Otherwise, DBGBVR<n>_EL1.ContextID must match the CONTEXTIDR_EL1 value

`0b0011` As `0b0010`, with linking enabled.

`0b0110` Unlinked CONTEXTIDR_EL1 match. DBGBVR<n>_EL1.ContextID is a Context ID compared against CONTEXTIDR_EL1.

`0b0111` As `0b0110`, with linking enabled.

`0b1000` Unlinked VMID match. DBGBVR<n>_EL1.VMID is a VMID compared against VTTBR_EL2.VMID.

`0b1001` As `0b1000`, with linking enabled.

`0b1010` Unlinked VMID and Context ID match. DBGBVR<n>_EL1.ContextID is a Context ID compared against CONTEXTIDR_EL1, and DBGBVR<n>_EL1.VMID is a VMID compared against VTTBR_EL2.VMID.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1011` As `0b1010`, with linking enabled.

`0b1100` Unlinked CONTEXTIDR_EL2 match. DBGBVR<n>_EL1.ContextID2 is a Context ID compared against CONTEXTIDR_EL2.

`0b1101` As `0b1100`, with linking enabled.

`0b1110` Unlinked Full Context ID match. DBGBVR<n>_EL1.ContextID is compared against CONTEXTIDR_EL1, and DBGBVR<n>_EL1.ContextID2 is compared against CONTEXTIDR_EL2.

`0b1111` As `0b1110`, with linking enabled.

All other values are reserved. Constraints on breakpoint programming mean other values are reserved under some conditions.

For more information on the operation of the SSC, HMC, and PMC fields, and on the effect of programming this field to a reserved value, see *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page D2-2448 and *Reserved DBGBCR<n>_EL1.BT values* on page D2-2453.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

LBN, bits [19:16]

Linked breakpoint number. For Linked address matching breakpoints, this specifies the index of the Context-matching breakpoint linked to.

For all other breakpoint types this field is ignored and reads of the register return an UNKNOWN value.

This field is ignored when the value of DBGBCR<n>_EL1.E is 0.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

SSC, bits [15:14]

Security state control. Determines the Security states under which a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the HMC and PMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields.

For more information on the operation of the SSC, HMC, and PMC fields, and the effect of programming the fields to a reserved set of values, see *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page D2-2448 and *Reserved DBGBCR<n>_EL1.{SSC, HMC, PMC} values* on page D2-2454.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

HMC, bit [13]

Higher mode control. Determines the debug perspective for deciding when a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the SSC and PMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information see the SSC, bits [15:14] description.

For more information on the operation of the SSC, HMC, and PMC fields, see *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page D2-2448.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

Bits [12:9]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

BAS, bits [8:5]

When AArch32 is supported at any Exception level:

Byte address select. Defines which half-words an address-matching breakpoint matches, regardless of the instruction set and Execution state.

The permitted values depend on the breakpoint type.

For Address match breakpoints, the permitted values are:

BAS	Match instruction at	Constraint for debuggers
`0b0011`	DBGBVR<n>_EL1	Use for T32 instructions
`0b1100`	DBGBVR<n>_EL1 + 2	Use for T32 instructions
`0b1111`	DBGBVR<n>_EL1	Use for A64 and A32 instructions

All other values are reserved. For more information, see *Reserved DBGBCR<n>_EL1.BAS values* on page D2-2455.

For more information on using the BAS field in address match breakpoints, see *Using the BAS field in Address Match breakpoints* on page G2-5843.

For Context matching breakpoints, this field is RES1 and ignored.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

Otherwise:

Reserved, RES1.

Bits [4:3]

Reserved, RES0.

PMC, bits [2:1]

Privilege mode control. Determines the Exception level or levels at which a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the SSC and HMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information see the DBGBCR<n>_EL1.SSC description.

For more information on the operation of the SSC, HMC, and PMC fields, see *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page D2-2448.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

E, bit [0]

Enable breakpoint DBGBVR<n>_EL1. Possible values are:

`0b0` Breakpoint disabled.

`0b1` Breakpoint enabled.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

Accessing the DBGBCR<n>_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, DBGBCR<n>_EL1

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	n[3:0]	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.DBGBCRn_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif !ELUsingAArch32(EL1) && OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        return DBGBCR_EL1[UInt(CRm<3:0>)];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif !ELUsingAArch32(EL1) && OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        return DBGBCR_EL1[UInt(CRm<3:0>)];
elsif PSTATE.EL == EL3 then
    if !ELUsingAArch32(EL1) && OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        return DBGBCR_EL1[UInt(CRm<3:0>)];
```

MSR DBGBCR<n>_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	n[3:0]	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.DBGBCRn_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif !ELUsingAArch32(EL1) && OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGBCR_EL1[UInt(CRm<3:0>)] = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif !ELUsingAArch32(EL1) && OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGBCR_EL1[UInt(CRm<3:0>)] = X[t];
elsif PSTATE.EL == EL3 then
    if !ELUsingAArch32(EL1) && OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
else
    DBGBCR_EL1[UInt(CRm<3:0>)] = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.3.3 DBGBVR<n>_EL1, Debug Breakpoint Value Registers, n = 0 - 15

The DBGBVR<n>_EL1 characteristics are:

Purpose

Holds a virtual address, or a VMID and/or a context ID, for use in breakpoint matching. Forms breakpoint n together with control register DBGBCR<n>_EL1.

Configurations

AArch64 System register DBGBVR<n>_EL1[31:0] is architecturally mapped to AArch32 System register DBGBVR<n>[31:0].

If the breakpoint is context-aware and EL2 is implemented then AArch64 System register DBGBVR<n>_EL1[63:32] is architecturally mapped to AArch32 System register DBGBXVR<n>[31:0]. Otherwise there is no System register access to DBGBVR<n>_EL1[63:32] from AArch32 state.

AArch64 System register DBGBVR<n>_EL1[63:0] is architecturally mapped to External register DBGBVR<n>_EL1[63:0].

If breakpoint n is not implemented then accesses to this register are UNDEFINED.

Attributes

How this register is interpreted depends on the value of DBGBCR<n>_EL1.BT.

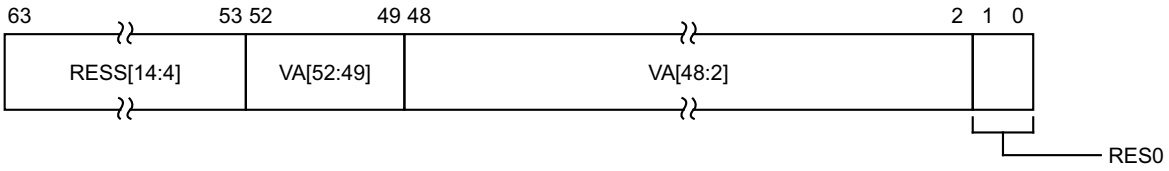
- When DBGBCR<n>_EL1.BT is `0b000x`, this register holds a virtual address.
- When DBGBCR<n>_EL1.BT is `0b001x`, `0b011x`, or `0b110x`, this register holds a Context ID.
- When DBGBCR<n>_EL1.BT is `0b100x`, this register holds a VMID.
- When DBGBCR<n>_EL1.BT is `0b101x`, this register holds a VMID and a Context ID.
- When DBGBCR<n>_EL1.BT is `0b111x`, this register holds two Context ID values.

For other values of DBGBCR<n>_EL1.BT, this register is RES0.

Field descriptions

The DBGBVR<n>_EL1 bit assignments are:

When DBGBCR<n>_EL1.BT == 0b000x:



RESS[14:4], bits [63:53]

Reserved, Sign extended. Software must treat this field as RES0 if the most significant bit of VA is 0 or RES0, and as RES1 if the most significant bit of VA is 1.

It is IMPLEMENTATION DEFINED whether:

- Reads return the value of the most significant bit of the VA for every bit in this field.
- Reads return the last value written.

The PE ignores this field.

VA[52:49], bits [52:49]

When ARMv8.2-LVA is implemented:

Extension to VA[48:2]. See VA[48:2] for more details.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

Otherwise:

Extension to RESS[14:4]. See RESS[14:4] for more details.

VA[48:2], bits [48:2]

Bits[48:2] of the address value for comparison.

When ARMv8.2-LVA is implemented, VA[52:49] forms the upper part of the address value. Otherwise, VA[52:49] are RESS.

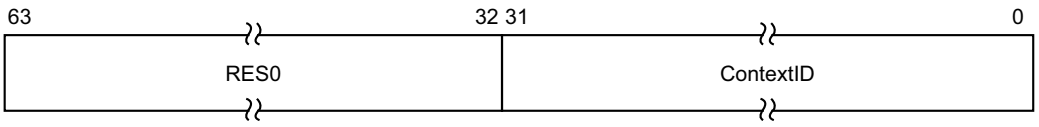
The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

Bits [1:0]

Reserved, RES0.

When DBGBCR<n>_EL1.BT == 0b001x:



Bits [63:32]

Reserved, RES0.

ContextID, bits [31:0]

Context ID value for comparison.

The value is compared against CONTEXTIDR_EL2 when ARMv8.1-VHE is implemented, HCR_EL2.E2H is 1, and either:

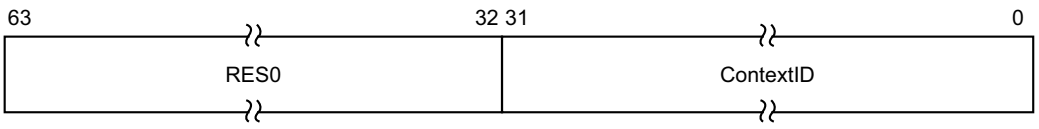
- The PE is executing at EL2.
- HCR_EL2.TGE is 1, the PE is executing at EL0, and EL2 is enabled in the current Security state.

Otherwise, the value is compared against CONTEXTIDR_EL1.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

When DBGBCR<n>_EL1.BT == 0b011x:



Bits [63:32]

Reserved, RES0.

ContextID, bits [31:0]

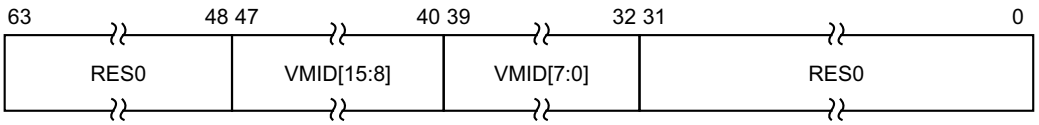
Context ID value for comparison against CONTEXTIDR_EL1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

When DBGBCR<n>_EL1.BT == 0b100x and EL2 is implemented:



Bits [63:48]

Reserved, RES0.

VMID[15:8], bits [47:40]

When ARMv8.1-VMID16 is implemented and VTCR_EL2.VS == 1:

Extension to VMID[7:0]. See DBGBVR<n>_EL1.VMID[7:0] for more details.

If EL2 is using AArch32, this field is RES0.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

Otherwise:

Reserved, RES0.

VMID[7:0], bits [39:32]

VMID value for comparison.

The VMID is 8 bits when any of the following are true:

- EL2 is using AArch32.
- VTCR_EL2.VS is 0.
- ARMv8.1-VMID16 is not implemented.

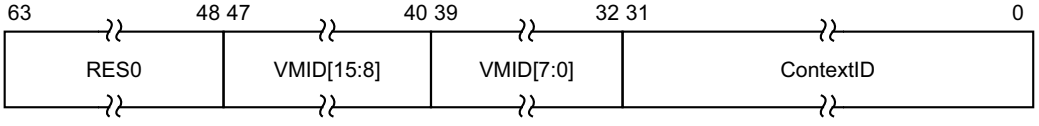
The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

Bits [31:0]

Reserved, RES0.

When DBGBCR<n>_EL1.BT == 0b101x and EL2 is implemented:



Bits [63:48]

Reserved, RES0.

VMID[15:8], bits [47:40]

When ARMv8.1-VMID16 is implemented and VTCR_EL2.VS == 1:

Extension to VMID[7:0]. See DBGBVR<n>_EL1.VMID[7:0] for more details.

If EL2 is using AArch32, or if the implementation has an 8-bit VMID, this field is RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

Otherwise:

Reserved, RES0.

VMID[7:0], bits [39:32]

VMID value for comparison.

The VMID is 8 bits when any of the following are true:

- EL2 is using AArch32.
- VTCR_EL2.VS is 0.
- ARMv8.1-VMID16 is not implemented.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

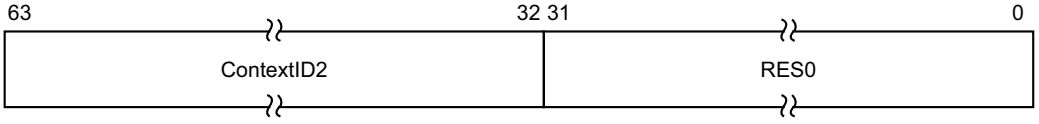
ContextID, bits [31:0]

Context ID value for comparison against CONTEXTIDR_EL1.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

When DBGBCR<n>_EL1.BT == 0b110x, EL2 is implemented and (ARMv8.1-VHE is implemented or ARMv8.2-Debug is implemented):



ContextID2, bits [63:32]

Context ID value for comparison against CONTEXTIDR_EL2.

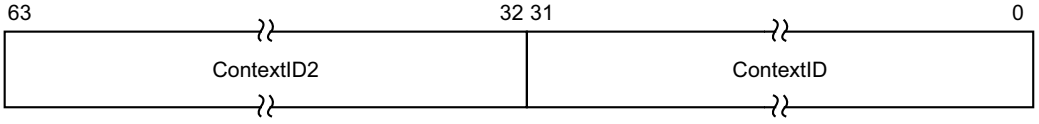
The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

Bits [31:0]

Reserved, RES0.

When DBGBCR<n>_EL1.BT == 0b111x, EL2 is implemented and (ARMv8.1-VHE is implemented or ARMv8.2-Debug is implemented):



ContextID2, bits [63:32]

Context ID value for comparison against CONTEXTIDR_EL2.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- On a Warm reset, the value of this field is unchanged.

ContextID, bits [31:0]

Context ID value for comparison against CONTEXTIDR_EL1.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

Accessing the DBGBVR<n>_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, DBGBVR<n>_EL1

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	n[3:0]	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.DBGBVRn_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif !ELUsingAArch32(EL1) && OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        return DBGBVR_EL1[UInt(CRm<3:0>)];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif !ELUsingAArch32(EL1) && OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        return DBGBVR_EL1[UInt(CRm<3:0>)];
elsif PSTATE.EL == EL3 then
    if !ELUsingAArch32(EL1) && OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        return DBGBVR_EL1[UInt(CRm<3:0>)];
```

MSR DBGBVR<n>_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	n[3:0]	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.DBGBVRn_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif !ELUsingAArch32(EL1) && OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGBVR_EL1[UInt(CRm<3:0>)] = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif !ELUsingAArch32(EL1) && OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGBVR_EL1[UInt(CRm<3:0>)] = X[t];
elsif PSTATE.EL == EL3 then
    if !ELUsingAArch32(EL1) && OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGBVR_EL1[UInt(CRm<3:0>)] = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.3.4 DBGCLAIMCLR_EL1, Debug CLAIM Tag Clear register

The DBGCLAIMCLR_EL1 characteristics are:

Purpose

Used by software to read the values of the CLAIM tag bits, and to clear CLAIM tag bits to 0.

The architecture does not define any functionality for the CLAIM tag bits.

Note

CLAIM tags are typically used for communication between the debugger and target software.

Used in conjunction with the DBGCLAIMSET_EL1 register.

Configurations

AArch64 System register DBGCLAIMCLR_EL1[31:0] is architecturally mapped to AArch32 System register DBGCLAIMCLR[31:0].

AArch64 System register DBGCLAIMCLR_EL1[31:0] is architecturally mapped to External register DBGCLAIMCLR_EL1[31:0].

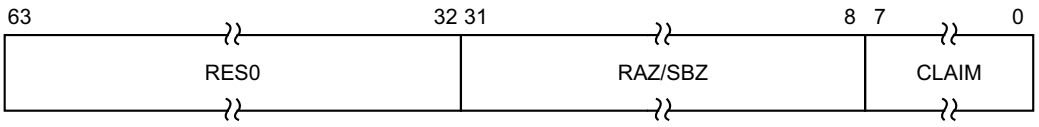
An implementation must include eight CLAIM tag bits.

Attributes

DBGCLAIMCLR_EL1 is a 64-bit register.

Field descriptions

The DBGCLAIMCLR_EL1 bit assignments are:



Bits [63:32]

Reserved, RES0.

Bits [31:8]

Reserved, RAZ/SBZ. Software can rely on these bits reading as zero, and must use a should-be-zero policy on writes. Implementations must ignore writes.

CLAIM, bits [7:0]

Read or clear CLAIM tag bits. Reading this field returns the current value of the CLAIM tag bits.

Writing a 1 to one of these bits clears the corresponding CLAIM tag bit to 0. This is an indirect write to the CLAIM tag bits. A single write operation can clear multiple CLAIM tag bits to 0.

Writing 0 to one of these bits has no effect.

The following resets apply:

- On a Cold reset, this field resets to `0`.
- On a Warm reset, the value of this field is unchanged.

Accessing the DBGCLAIMCLR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, DBGCLAIMCLR_EL1

op0	op1	CRn	CRm	op2
0b10	0b000	0b0111	0b1001	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.DBGCLAIM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return DBGCLAIMCLR_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return DBGCLAIMCLR_EL1;
elsif PSTATE.EL == EL3 then
    return DBGCLAIMCLR_EL1;
```

MSR DBGCLAIMCLR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b000	0b0111	0b1001	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.DBGCLAIM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        DBGCLAIMCLR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        DBGCLAIMCLR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    DBGCLAIMCLR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.3.5 DBGCLAIMSET_EL1, Debug CLAIM Tag Set register

The DBGCLAIMSET_EL1 characteristics are:

Purpose

Used by software to set the CLAIM tag bits to 1.

The architecture does not define any functionality for the CLAIM tag bits.

Note

CLAIM tags are typically used for communication between the debugger and target software.

Used in conjunction with the DBGCLAIMCLR_EL1 register.

Configurations

AArch64 System register DBGCLAIMSET_EL1[31:0] is architecturally mapped to AArch32 System register DBGCLAIMSET[31:0].

AArch64 System register DBGCLAIMSET_EL1[31:0] is architecturally mapped to External register DBGCLAIMSET_EL1[31:0].

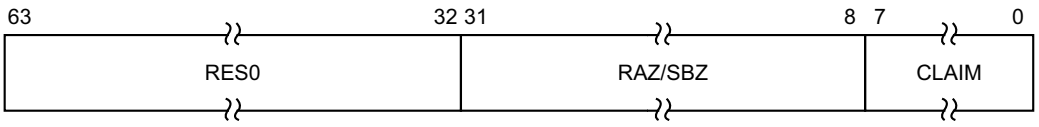
An implementation must include eight CLAIM tag bits.

Attributes

DBGCLAIMSET_EL1 is a 64-bit register.

Field descriptions

The DBGCLAIMSET_EL1 bit assignments are:



Bits [63:32]

Reserved, RES0.

Bits [31:8]

Reserved, RAZ/SBZ. Software can rely on these bits reading as zero, and must use a should-be-zero policy on writes. Implementations must ignore writes.

CLAIM, bits [7:0]

Set CLAIM tag bits.

This field is RAO.

Writing a 1 to one of these bits sets the corresponding CLAIM tag bit to 1. This is an indirect write to the CLAIM tag bits. A single write operation can set multiple CLAIM tag bits to 1.

Writing 0 to one of these bits has no effect.

Accessing the DBGCLAIMSET_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, DBGCLAIMSET_EL1

op0	op1	CRn	CRm	op2
0b10	0b000	0b0111	0b1000	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.DBGCLAIM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return DBGCLAIMSET_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return DBGCLAIMSET_EL1;
elsif PSTATE.EL == EL3 then
    return DBGCLAIMSET_EL1;
```

MSR DBGCLAIMSET_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b000	0b0111	0b1000	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.DBGCLAIM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        DBGCLAIMSET_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        DBGCLAIMSET_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    DBGCLAIMSET_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.3.6 DBGDTR_EL0, Debug Data Transfer Register, half-duplex

The DBGDTR_EL0 characteristics are:

Purpose

Transfers 64 bits of data between the PE and an external debugger. Can transfer both ways using only a single register.

Configurations

AArch64 System register DBGDTR_EL0[63:32] is architecturally mapped to AArch32 System register DBGDTRRXint[31:0] when written.

AArch64 System register DBGDTR_EL0[63:32] is architecturally mapped to External register DBGDTRRX_EL0[31:0] when written.

AArch64 System register DBGDTR_EL0[63:32] is architecturally mapped to AArch64 System register DBGDTRRX_EL0[31:0] when written.

AArch64 System register DBGDTR_EL0[31:0] is architecturally mapped to AArch32 System register DBGDTRTXint[31:0] when written.

AArch64 System register DBGDTR_EL0[31:0] is architecturally mapped to External register DBGDTRTX_EL0[31:0] when written.

AArch64 System register DBGDTR_EL0[31:0] is architecturally mapped to AArch64 System register DBGDTRTX_EL0[31:0] when written.

AArch64 System register DBGDTR_EL0[63:32] is architecturally mapped to AArch32 System register DBGDTRTXint[31:0] when read.

AArch64 System register DBGDTR_EL0[63:32] is architecturally mapped to External register DBGDTRTX_EL0[31:0] when read.

AArch64 System register DBGDTR_EL0[63:32] is architecturally mapped to AArch64 System register DBGDTRTX_EL0[31:0] when read.

AArch64 System register DBGDTR_EL0[31:0] is architecturally mapped to AArch32 System register DBGDTRRXint[31:0] when read.

AArch64 System register DBGDTR_EL0[31:0] is architecturally mapped to External register DBGDTRRX_EL0[31:0] when read.

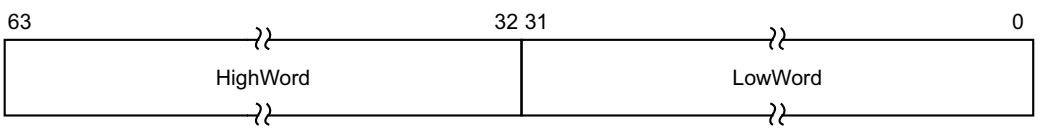
AArch64 System register DBGDTR_EL0[31:0] is architecturally mapped to AArch64 System register DBGDTRRX_EL0[31:0] when read.

Attributes

DBGDTR_EL0 is a 64-bit register.

Field descriptions

The DBGDTR_EL0 bit assignments are:



HighWord, bits [63:32]

Writes to this register set DTRRX to the value in this field and do not change RXfull.

Reads of this register:

- If RXfull is set to 1, return the last value written to DTRTX.
- If RXfull is set to 0, return an UNKNOWN value.

After the read, RXfull is cleared to 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

LowWord, bits [31:0]

Writes to this register set DTRTX to the value in this field and set TXfull to 1.

Reads of this register:

- If RXfull is set to 1, return the last value written to DTRRX.
- If RXfull is set to 0, return an UNKNOWN value.

After the read, RXfull is cleared to 0.

Accessing the DBGDTR_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, DBGDTR_EL0

op0	op1	CRn	CRm	op2
0b10	0b011	0b0000	0b0100	0b000

```
if Halted() then
    return DBGDTR_EL0;
elsif PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && MDSCR_EL1.TDCC == '1' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (HCR_EL2.TGE == '1' || MDCR_EL2.<TDE,TDA> != '00') then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return DBGDTR_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return DBGDTR_EL0;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return DBGDTR_EL0;
elsif PSTATE.EL == EL3 then
    return DBGDTR_EL0;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR DBGDTR_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b011	0b0000	0b0100	0b000

```
if Halted() then
    DBGDTR_EL0 = X[t];
elsif PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && MDSCR_EL1.TDCC == '1' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (HCR_EL2.TGE == '1' || MDCR_EL2.<TDE,TDA> != '00') then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        DBGDTR_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        DBGDTR_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        DBGDTR_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    DBGDTR_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.3.7 DBGDTRRX_EL0, Debug Data Transfer Register, Receive

The DBGDTRRX_EL0 characteristics are:

Purpose

Transfers data from an external debugger to the PE. For example, it is used by a debugger transferring commands and data to a debug target. See DBGDTR_EL0 for additional architectural mappings. It is a component of the Debug Communications Channel.

Configurations

AArch64 System register DBGDTRRX_EL0[31:0] is architecturally mapped to AArch32 System register DBGDTRRXint[31:0].

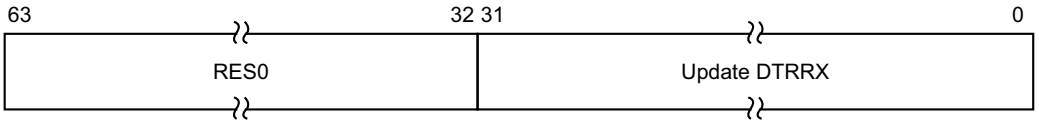
AArch64 System register DBGDTRRX_EL0[31:0] is architecturally mapped to External register DBGDTRRX_EL0[31:0].

Attributes

DBGDTRRX_EL0 is a 64-bit register.

Field descriptions

The DBGDTRRX_EL0 bit assignments are:



Bits [63:32]

Reserved, RES0.

Bits [31:0]

Update DTRRX.

Reads of this register:

- If RXfull is set to 1, return the last value written to DTRRX.
- If RXfull is set to 0, return an UNKNOWN value.

After the read, RXfull is cleared to 0.

For the full behavior of the Debug Communications Channel, see Chapter H4 *The Debug Communication Channel and Instruction Transfer Register*.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

Accessing the DBGDTRRX_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, DBGDTRRX_EL0

op0	op1	CRn	CRm	op2
0b10	0b011	0b0000	0b0101	0b000

```
if Halted() then
    return DBGDTRRX_EL0;
elsif PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && MDSCR_EL1.TDCC == '1' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (HCR_EL2.TGE == '1' || MDCR_EL2.<TDE,TDA> != '00') then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return DBGDTRRX_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return DBGDTRRX_EL0;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return DBGDTRRX_EL0;
elsif PSTATE.EL == EL3 then
    return DBGDTRRX_EL0;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.3.8 DBGDTRTX_EL0, Debug Data Transfer Register, Transmit

The DBGDTRTX_EL0 characteristics are:

Purpose

Transfers data from the PE to an external debugger. For example, it is used by a debug target to transfer data to the debugger. See DBGDTR_EL0 for additional architectural mappings. It is a component of the Debug Communication Channel.

Configurations

AArch64 System register DBGDTRTX_EL0[31:0] is architecturally mapped to AArch32 System register DBGDTRTXint[31:0].

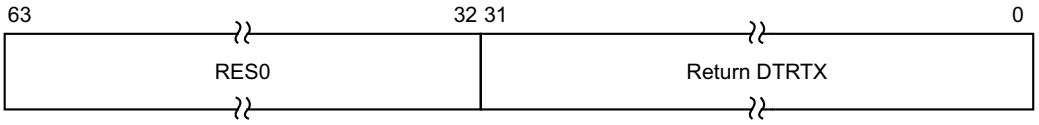
AArch64 System register DBGDTRTX_EL0[31:0] is architecturally mapped to External register DBGDTRTX_EL0[31:0].

Attributes

DBGDTRTX_EL0 is a 64-bit register.

Field descriptions

The DBGDTRTX_EL0 bit assignments are:



Bits [63:32]

Reserved, RES0.

Bits [31:0]

Return DTRTX.

Writes to this register:

- If TXfull is set to 1, set DTRRX and DTRTX to UNKNOWN.
- If TXfull is set to 0, update the value in DTRTX.

After the write, TXfull is set to 1.

For the full behavior of the Debug Communications Channel, see Chapter H4 *The Debug Communication Channel and Instruction Transfer Register*.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

Accessing the DBGDTRTX_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR DBGDTRTX_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b011	0b0000	0b0101	0b000

```
if Halted() then
    DBGDTRTX_EL0 = X[t];
elsif PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && MDSCR_EL1.TDCC == '1' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (HCR_EL2.TGE == '1' || MDCR_EL2.<TDE,TDA> != '00') then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        DBGDTRTX_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        DBGDTRTX_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        DBGDTRTX_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    DBGDTRTX_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.3.9 DBGPRCR_EL1, Debug Power Control Register

The DBGPRCR_EL1 characteristics are:

Purpose

Controls behavior of the PE on powerdown request.

Configurations

AArch64 System register DBGPRCR_EL1[31:0] is architecturally mapped to AArch32 System register DBGPRCR[31:0].

Bit [0] of this register is mapped to EDPRCR.CORENPDRQ, bit [0] of the external view of this register.

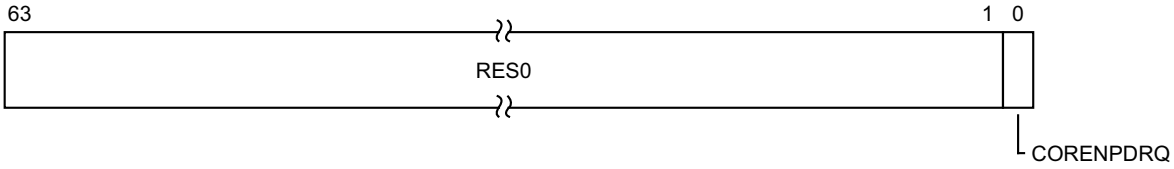
The other bits in these registers are not mapped to each other.

Attributes

DBGPRCR_EL1 is a 64-bit register.

Field descriptions

The DBGPRCR_EL1 bit assignments are:



Bits [63:1]

Reserved, RES0.

CORENPDRQ, bit [0]

When ARMv8.3-DoPD is implemented:

Core no powerdown request. Requests emulation of powerdown.

This request is typically passed to an external power controller. This means that whether a request causes power up is dependent on the IMPLEMENTATION DEFINED nature of the system. The power controller must not allow the Core power domain to switch off while this bit is 1.

`0b0` If the system responds to a powerdown request, it powers down Core power domain.

`0b1` If the system responds to a powerdown request, it does not powerdown the Core power domain, but instead emulates a powerdown of that domain.

In an implementation that includes the recommended external debug interface, this bit drives the DBGNOPWRDWN signal.

It is IMPLEMENTATION DEFINED whether this bit is reset to its Cold reset value on exit from an IMPLEMENTATION DEFINED software-visible retention state. For more information about retention states see *Core power domain power states* on page H6-7030.

Note

Writes to this bit are not prohibited by the IMPLEMENTATION DEFINED authentication interface. This means that a debugger can request emulation of powerdown regardless of whether invasive debug is permitted.

The following resets apply:

- On a Cold reset, this field is set to 1 if the powerup request is implemented and the powerup request has been asserted, and is set to 0 otherwise.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- On a Warm reset, the value of this field is unchanged.

Otherwise:

Core no powerdown request. Requests emulation of powerdown.

This request is typically passed to an external power controller. This means that whether a request causes power up is dependent on the IMPLEMENTATION DEFINED nature of the system. The power controller must not allow the Core power domain to switch off while this bit is 1.

`0b0` If the system responds to a powerdown request, it powers down Core power domain.

`0b1` If the system responds to a powerdown request, it does not powerdown the Core power domain, but instead emulates a powerdown of that domain.

In an implementation that includes the recommended external debug interface, this bit drives the DBGNOPWRDWN signal.

It is IMPLEMENTATION DEFINED whether this bit is reset to the value of EDPRCR.COREPURQ on exit from an IMPLEMENTATION DEFINED software-visible retention state. For more information about retention states see *Core power domain power states* on page H6-7030.

Note

Writes to this bit are not prohibited by the IMPLEMENTATION DEFINED authentication interface. This means that a debugger can request emulation of powerdown regardless of whether invasive debug is permitted.

The following resets apply:

- On a Cold reset, this field resets to the value in EDPRCR.COREPURQ.
- On a Warm reset, the value of this field is unchanged.

Accessing the DBGPRCR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, DBGPRCR_EL1

op0	op1	CRn	CRm	op2
0b10	0b000	0b0001	0b0100	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.DBGPRCR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDOSA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return DBGPRCR_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return DBGPRCR_EL1;
elsif PSTATE.EL == EL3 then
    return DBGPRCR_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR DBGPRCR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b000	0b0001	0b0100	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.DBGPRCR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDOSA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        DBGPRCR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        DBGPRCR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    DBGPRCR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.3.10 DBGVCR32_EL2, Debug Vector Catch Register

The DBGVCR32_EL2 characteristics are:

Purpose

Allows access to the AArch32 register DBGVCR from AArch64 state only. Its value has no effect on execution in AArch64 state.

Configurations

AArch64 System register DBGVCR32_EL2[31:0] is architecturally mapped to AArch32 System register DBGVCR[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DBGVCR32_EL2 are UNDEFINED.

If EL2 is not implemented but EL3 is implemented, and EL1 is capable of using AArch32, then this register is not RES0.

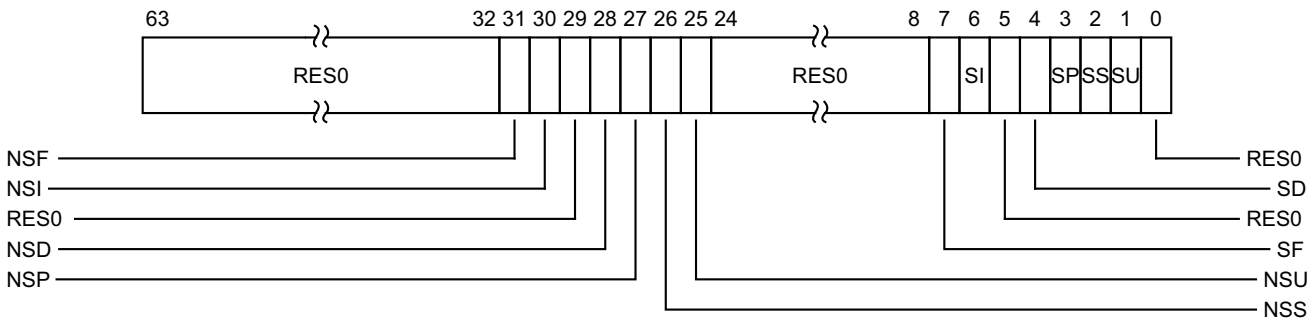
Attributes

DBGVCR32_EL2 is a 64-bit register.

Field descriptions

The DBGVCR32_EL2 bit assignments are:

When EL3 is implemented:



Bits [63:32]

Reserved, RES0.

NSF, bit [31]

FIQ vector catch enable in Non-secure state.

The exception vector offset is `0x1C`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

NSI, bit [30]

IRQ vector catch enable in Non-secure state.

The exception vector offset is `0x18`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [29]

Reserved, RES0.

NSD, bit [28]

Data Abort vector catch enable in Non-secure state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The exception vector offset is `0x10`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

NSP, bit [27]

Prefetch Abort vector catch enable in Non-secure state.

The exception vector offset is `0x0C`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

NSS, bit [26]

Supervisor Call (SVC) vector catch enable in Non-secure state.

The exception vector offset is `0x08`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

NSU, bit [25]

Undefined Instruction vector catch enable in Non-secure state.

The exception vector offset is `0x04`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [24:8]

Reserved, RES0.

SF, bit [7]

FIQ vector catch enable in Secure state.

The exception vector offset is `0x1C`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

SI, bit [6]

IRQ vector catch enable in Secure state.

The exception vector offset is `0x18`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

SD, bit [4]

Data Abort vector catch enable in Secure state.

The exception vector offset is `0x10`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

SP, bit [3]

Prefetch Abort vector catch enable in Secure state.

The exception vector offset is `0x0C`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

SS, bit [2]

Supervisor Call (SVC) vector catch enable in Secure state.

The exception vector offset is `0x08`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

SU, bit [1]

Undefined Instruction vector catch enable in Secure state.

The exception vector offset is `0x04`.

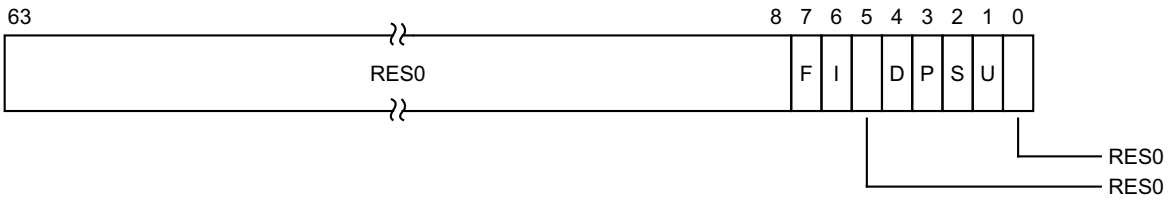
Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [0]

Reserved, RES0.

When EL3 is not implemented:



Bits [63:8]

Reserved, RES0.

F, bit [7]

FIQ vector catch enable.

The exception vector offset is `0x1C`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [6]

IRQ vector catch enable.

The exception vector offset is `0x18`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

D, bit [4]

Data Abort vector catch enable.

The exception vector offset is `0x10`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

P, bit [3]

Prefetch Abort vector catch enable.

The exception vector offset `0x0C`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

S, bit [2]

Supervisor Call (SVC) vector catch enable.

The exception vector offset is `0x08`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

U, bit [1]

Undefined Instruction vector catch enable.

The exception vector offset is `0x04`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [0]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the DBGVCR32_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, DBGVCR32_EL2

op0	op1	CRn	CRm	op2
0b10	0b100	0b0000	0b0111	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return DBGVCR32_EL2;
elsif PSTATE.EL == EL3 then
    return DBGVCR32_EL2;
```

MSR DBGVCR32_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b100	0b0000	0b0111	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        DBGVCR32_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    DBGVCR32_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.3.11 DBGWCR<n>_EL1, Debug Watchpoint Control Registers, n = 0 - 15

The DBGWCR<n>_EL1 characteristics are:

Purpose

Holds control information for a watchpoint. Forms watchpoint n together with value register DBGWVR<n>_EL1.

Configurations

AArch64 System register DBGWCR<n>_EL1[31:0] is architecturally mapped to AArch32 System register DBGWCR<n>[31:0].

AArch64 System register DBGWCR<n>_EL1[31:0] is architecturally mapped to External register DBGWCR<n>_EL1[31:0].

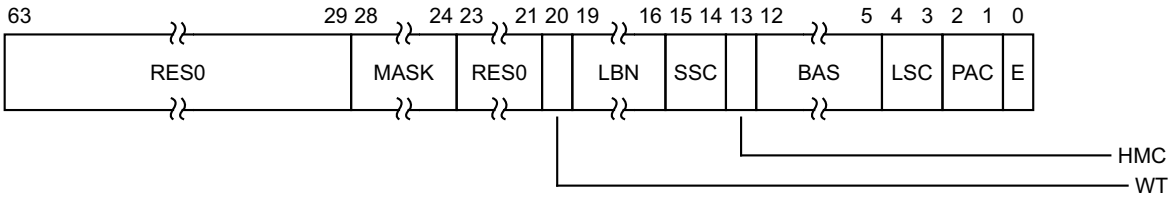
If watchpoint n is not implemented then accesses to this register are UNDEFINED.

Attributes

DBGWCR<n>_EL1 is a 64-bit register.

Field descriptions

The DBGWCR<n>_EL1 bit assignments are:



Bits [63:29]

Reserved, RES0.

MASK, bits [28:24]

Address mask. Only objects up to 2GB can be watched using a single mask.

`0b00000` No mask.

`0b00001` Reserved.

`0b00010` Reserved.

If programmed with a reserved value, a watchpoint must behave as if either:

- MASK has been programmed with a defined value, which might be 0 (no mask), other than for a direct read of DBGWCRn_EL1.
- The watchpoint is disabled.

Software must not rely on this property because the behavior of reserved values might change in a future revision of the architecture.

Other values mask the corresponding number of address bits, from `0b00011` masking 3 address bits (`0x00000007` mask for address) to `0b11111` masking 31 address bits (`0x7FFFFFFF` mask for address).

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

Bits [23:21]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

WT, bit [20]

Watchpoint type. Possible values are:

`0b0` Unlinked data address match.

`0b1` Linked data address match.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

LBN, bits [19:16]

Linked breakpoint number. For Linked data address watchpoints, this specifies the index of the Context-matching breakpoint linked to.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

SSC, bits [15:14]

Security state control. Determines the Security states under which a Watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the HMC and PAC fields.

For more information, see *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page D2-2448, and *Reserved DBGBCR<n>_EL1.{SSC, HMC, PMC} values* on page D2-2454.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

HMC, bit [13]

Higher mode control. Determines the debug perspective for deciding when a Watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the SSC and PAC fields.

For more information on the operation of the SSC, HMC, and PAC fields, see *Execution conditions for which a watchpoint generates Watchpoint exceptions* on page D2-2459.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

BAS, bits [12:5]

Byte address select. Each bit of this field selects whether a byte from within the word or double-word addressed by DBGWVR<n>_EL1 is being watched.

BAS	Description
xxxxxxx1	Match byte at DBGWVR<n>_EL1
xxxxxx1x	Match byte at DBGWVR<n>_EL1 + 1
xxxxx1xx	Match byte at DBGWVR<n>_EL1 + 2
xxxx1xxx	Match byte at DBGWVR<n>_EL1 + 3

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In cases where DBGWVR<n>_EL1 addresses a double-word:

BAS	Description, if DBGWVR<n>_EL1[2] == 0
xxx1xxxx	Match byte at DBGWVR<n>_EL1 + 4
xx1xxxxx	Match byte at DBGWVR<n>_EL1 + 5
x1xxxxxx	Match byte at DBGWVR<n>_EL1 + 6
1xxxxxxx	Match byte at DBGWVR<n>_EL1 + 7

If DBGWVR<n>_EL1[2] == 1, only BAS[3:0] are used and BAS[7:4] are ignored. Arm deprecates setting DBGWVR<n>_EL1[2] == 1.

The valid values for BAS are non-zero binary numbers all of whose set bits are contiguous. All other values are reserved and must not be used by software. See *Reserved DBGWCR<n>_EL1.BAS values* on page D2-2468.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

LSC, bits [4:3]

Load/store control. This field enables watchpoint matching on the type of access being made. Possible values of this field are:

`0b01` Match instructions that load from a watchpointed address.

`0b10` Match instructions that store to a watchpointed address.

`0b11` Match instructions that load from or store to a watchpointed address.

All other values are reserved, but must behave as if the watchpoint is disabled. Software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

PAC, bits [2:1]

Privilege of access control. Determines the Exception level or levels at which a Watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the SSC and HMC fields.

For more information on the operation of the SSC, HMC, and PAC fields, see *Execution conditions for which a watchpoint generates Watchpoint exceptions* on page D2-2459.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

E, bit [0]

Enable watchpoint n. Possible values are:

`0b0` Watchpoint disabled.

`0b1` Watchpoint enabled.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the DBGWCR<n>_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, DBGWCR<n>_EL1

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	n[3:0]	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.DBGWCRn_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif !ELUsingAArch32(EL1) && OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        return DBGWCR_EL1[UInt(CRm<3:0>)];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif !ELUsingAArch32(EL1) && OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        return DBGWCR_EL1[UInt(CRm<3:0>)];
elsif PSTATE.EL == EL3 then
    if !ELUsingAArch32(EL1) && OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        return DBGWCR_EL1[UInt(CRm<3:0>)];
```

MSR DBGWCR<n>_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	n[3:0]	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.DBGWCRn_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif !ELUsingAArch32(EL1) && OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGWCR_EL1[UInt(CRm<3:0>)] = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif !ELUsingAArch32(EL1) && OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        else
            DBGWCR_EL1[UInt(CRm<3:0>)] = X[t];
    elsif PSTATE.EL == EL3 then
        if !ELUsingAArch32(EL1) && OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
            Halt(DebugHalt_SoftwareAccess);
        else
            DBGWCR_EL1[UInt(CRm<3:0>)] = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.3.12 DBGWVR<n>_EL1, Debug Watchpoint Value Registers, n = 0 - 15

The DBGWVR<n>_EL1 characteristics are:

Purpose

Holds a data address value for use in watchpoint matching. Forms watchpoint n together with control register DBGWCR<n>_EL1.

Configurations

AArch64 System register DBGWVR<n>_EL1[31:0] is architecturally mapped to AArch32 System register DBGWVR<n>[31:0].

AArch64 System register DBGWVR<n>_EL1[63:0] is architecturally mapped to External register DBGWVR<n>_EL1[63:0].

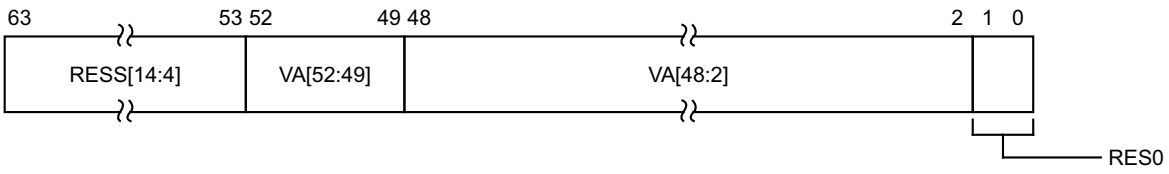
If watchpoint n is not implemented then accesses to this register are UNDEFINED.

Attributes

DBGWVR<n>_EL1 is a 64-bit register.

Field descriptions

The DBGWVR<n>_EL1 bit assignments are:



RESS[14:4], bits [63:53]

Reserved, Sign extended. Hardware and software must treat this field as RES0 if the most significant bit of VA is 0 or RES0, and as RES1 if the most significant bit of VA is 1.

Hardware always ignores the value of these bits and it is IMPLEMENTATION DEFINED whether:

- The bits are hardwired to a copy of the most significant bit of VA, meaning writes to these bits are ignored, and reads to the bits always return the hardwired value.
- The value in those bits can be written, and reads will return the last value written. The value held in those bits is ignored by hardware.

VA[52:49], bits [52:49]

When ARMv8.2-LVA is implemented:

Extension to VA[48:2]. See VA[48:2] for more details.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

Otherwise:

Extension to RESS[14:4]. See RESS[14:4] for more details.

VA[48:2], bits [48:2]

Bits[48:2] of the address value for comparison.

When ARMv8.2-LVA is implemented, VA[52:49] forms the upper part of the address value. Otherwise, VA[52:49] are RESS.

Arm deprecates setting DBGWVR<n>_EL1[2] == 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

Bits [1:0]

Reserved, RES0.

Accessing the DBGWVR<n>_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, DBGWVR<n>_EL1

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	n[3:0]	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.DBGWVRn_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif !ELUsingAArch32(EL1) && OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        return DBGWVR_EL1[UInt(CRm<3:0>)];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif !ELUsingAArch32(EL1) && OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        return DBGWVR_EL1[UInt(CRm<3:0>)];
elsif PSTATE.EL == EL3 then
    if !ELUsingAArch32(EL1) && OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        return DBGWVR_EL1[UInt(CRm<3:0>)];
```

MSR DBGWVR<n>_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	n[3:0]	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.DBGWVRn_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif !ELUsingAArch32(EL1) && OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGWVR_EL1[UInt(CRm<3:0>)] = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif !ELUsingAArch32(EL1) && OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGWVR_EL1[UInt(CRm<3:0>)] = X[t];
elsif PSTATE.EL == EL3 then
    if !ELUsingAArch32(EL1) && OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGWVR_EL1[UInt(CRm<3:0>)] = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.3.13 DLR_EL0, Debug Link Register

The DLR_EL0 characteristics are:

Purpose

In Debug state, holds the address to restart from.

Configurations

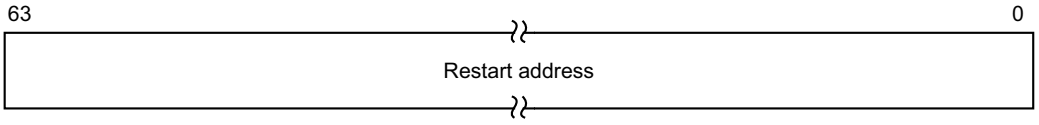
AArch64 System register DLR_EL0[31:0] is architecturally mapped to AArch32 System register DLR[31:0].

Attributes

DLR_EL0 is a 64-bit register.

Field descriptions

The DLR_EL0 bit assignments are:



Bits [63:0]

Restart address.

Accessing the DLR_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, DLR_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0101	0b001

```
if !Halted() then
    UNDEFINED;
else
    return DLR_EL0;
```

MSR DLR_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0101	0b001

```
if !Halted() then
    UNDEFINED;
else
    DLR_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.3.14 DSPSR_EL0, Debug Saved Program Status Register

The DSPSR_EL0 characteristics are:

Purpose

Holds the saved process state for Debug state. On entering Debug state, PSTATE information is written to this register. On exiting Debug state, values are copied from this register to PSTATE.

Configurations

AArch64 System register DSPSR_EL0[31:0] is architecturally mapped to AArch32 System register DSPSR[31:0].

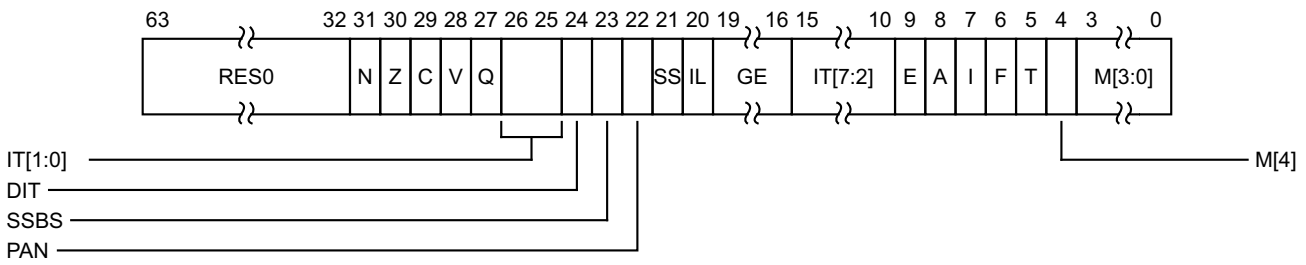
Attributes

DSPSR_EL0 is a 64-bit register.

Field descriptions

The DSPSR_EL0 bit assignments are:

When AArch32 is supported at any Exception level and exiting Debug state to AArch32 state:



Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Copied to PSTATE.N on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Copied to PSTATE.Z on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Copied to PSTATE.C on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Copied to PSTATE.V on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Copied to PSTATE.Q on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

IT[1:0], bits [26:25]

If-Then. Copied to PSTATE.IT[1:0] on exiting Debug state.

On exiting Debug state DSPSR_EL0.IT must contain a value that is valid for the instruction being returned to.

This field resets to an architecturally UNKNOWN value.

DIT, bit [24]

When ARMv8.4-DIT is implemented:

Data Independent Timing. Copied to PSTATE.DIT on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SSBS, bit [23]

When ARMv8.0-SSBS is implemented:

Speculative Store Bypass. Copied to PSTATE.SSBS on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When ARMv8.1-PAN is implemented:

Privileged Access Never. Copied to PSTATE.PAN on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SS, bit [21]

Software Step. Copied to PSTATE.SS on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Copied to PSTATE.IL on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Copied to PSTATE.GE on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

IT[7:2], bits [15:10]

If-Then. Copied to PSTATE.IT[7:2] on exiting Debug state.

DSPSR_EL0.IT must contain a value that is valid for the instruction being returned to.

This field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Copied to PSTATE.E on exiting Debug state.

If the implementation does not support big-endian operation, DSPSR_EL0.E is RES0. If the implementation does not support little-endian operation, DSPSR_EL0.E is RES1. On exiting Debug state, if the implementation does not support big-endian operation at the Exception level being returned to, DSPSR_EL0.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, DSPSR_EL0.E is RES1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

A, bit [8]

SError interrupt mask. Copied to PSTATE.A on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Copied to PSTATE.I on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Copied to PSTATE.F on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Copied to PSTATE.T on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

M[4], bit [4]

Execution state. Copied to PSTATE.nRW on exiting Debug state.

`0b1` AArch32 execution state.

This field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]

AArch32 Mode. Copied to PSTATE.M[3:0] on exiting Debug state.

`0b0000` User.

`0b0001` FIQ.

`0b0010` IRQ.

`0b0011` Supervisor.

`0b0110` Monitor.

`0b0111` Abort.

`0b1010` Hyp.

`0b1011` Undefined.

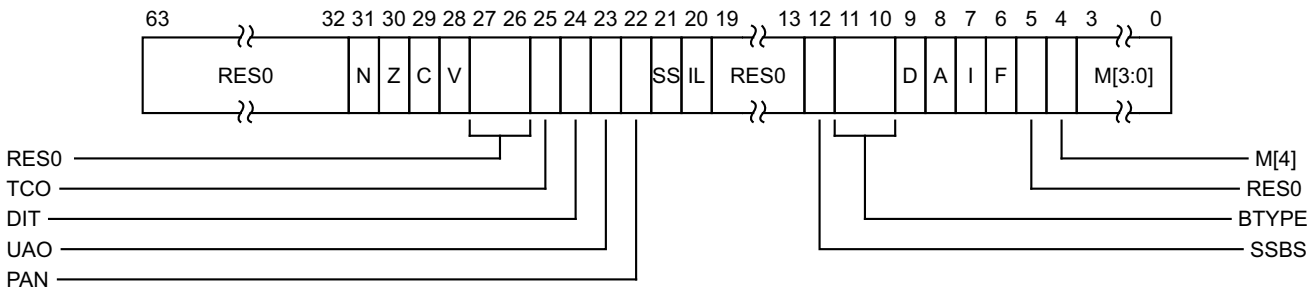
`0b1111` System.

Other values are reserved. If DSPSR_EL0.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, exiting Debug state is an illegal return event, as described in *Illegal return events from AArch64 state* on page D1-2351.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When AArch64 is supported at any Exception level and entering Debug state from AArch64 state and exiting Debug state to AArch64 state:



Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on entering Debug state, and copied to PSTATE.N on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on entering Debug state, and copied to PSTATE.Z on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on entering Debug state, and copied to PSTATE.C on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on entering Debug state, and copied to PSTATE.V on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

Bits [27:26]

Reserved, RES0.

TCO, bit [25]

When ARMv8.5-MemTag is implemented:

Tag Check Override. Set to the value of PSTATE.TCO on entering Debug state, and copied to PSTATE.TCO on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [24]

When ARMv8.4-DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on entering Debug state, and copied to PSTATE.DIT on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

Reserved, RES0.

UAO, bit [23]

When ARMv8.2-UAO is implemented:

User Access Override. Set to the value of PSTATE.UAO on entering Debug state, and copied to PSTATE.UAO on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When ARMv8.1-PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on entering Debug state, and copied to PSTATE.PAN on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SS, bit [21]

Software Step. Set to the value of PSTATE.SS on entering Debug state, and conditionally copied to PSTATE.SS on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on entering Debug state, and copied to PSTATE.IL on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

Bits [19:13]

Reserved, RES0.

SSBS, bit [12]

When ARMv8.0-SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on entering Debug state, and copied to PSTATE.SSBS on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

BTYPE, bits [11:10]

When ARMv8.5-BTI is implemented:

Branch Type Indicator. Set to the value of PSTATE.BTYPE on entering Debug state, and copied to PSTATE.BTYPE on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

D, bit [9]

Debug exception mask. Set to the value of PSTATE.D on entering Debug state, and copied to PSTATE.D on exiting Debug state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

A, bit [8]

SError interrupt mask. Set to the value of PSTATE.A on entering Debug state, and copied to PSTATE.A on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on entering Debug state, and copied to PSTATE.I on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on entering Debug state, and copied to PSTATE.F on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

M[4], bit [4]

Execution state. Set to `0b0`, the value of PSTATE.nRW, on entering Debug state from AArch64 state, and copied to PSTATE.nRW on exiting Debug state.

`0b0` AArch64 execution state.

This field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]

AArch64 Exception level and selected Stack Pointer.

`0b0000` EL0t.

`0b0100` EL1t.

`0b0101` EL1h.

`0b1000` EL2t.

`0b1001` EL2h.

`0b1100` EL3t.

`0b1101` EL3h.

Other values are reserved. If DSPSR_EL0.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, exiting Debug state is an illegal return event, as described in *Illegal return events from AArch64 state* on page D1-2351.

The bits in this field are interpreted as follows:

- M[3:2] is set to the value of PSTATE.EL on entering Debug state and copied to PSTATE.EL on exiting Debug state.
- M[1] is unused and is 0 for all non-reserved values.
- M[0] is set to the value of PSTATE.SP on entering Debug state and copied to PSTATE.SP on exiting Debug state

This field resets to an architecturally UNKNOWN value.

Accessing the DSPSR_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, DSPSR_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0101	0b000

```
if !Halted() then
    UNDEFINED;
else
    return DSPSR_EL0;
```

MSR DSPSR_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0101	0b000

```
if !Halted() then
    UNDEFINED;
else
    DSPSR_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.3.15 MDCCINT_EL1, Monitor DCC Interrupt Enable Register

The MDCCINT_EL1 characteristics are:

Purpose

Enables interrupt requests to be signaled based on the DCC status flags.

Configurations

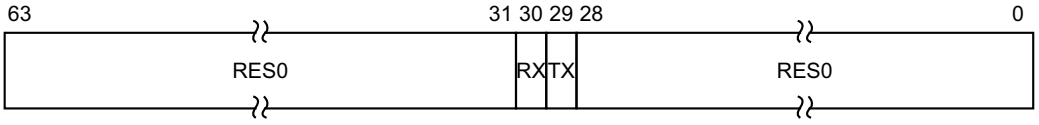
AArch64 System register MDCCINT_EL1[31:0] is architecturally mapped to AArch32 System register DBGDCCINT[31:0].

Attributes

MDCCINT_EL1 is a 64-bit register.

Field descriptions

The MDCCINT_EL1 bit assignments are:



Bits [63:31]

Reserved, RES0.

RX, bit [30]

DCC interrupt request enable control for DTRRX. Enables a common **COMMIRQ** interrupt request to be signaled based on the DCC status flags.

`0b0` No interrupt request generated by DTRRX.

`0b1` Interrupt request will be generated on RXfull == 1.

If legacy **COMMRX** and **COMMTX** signals are implemented, then these are not affected by the value of this bit.

On a Warm reset, this field resets to `0`.

TX, bit [29]

DCC interrupt request enable control for DTRTX. Enables a common **COMMIRQ** interrupt request to be signaled based on the DCC status flags.

`0b0` No interrupt request generated by DTRTX.

`0b1` Interrupt request will be generated on TXfull == 0.

If legacy **COMMRX** and **COMMTX** signals are implemented, then these are not affected by the value of this bit.

On a Warm reset, this field resets to `0`.

Bits [28:0]

Reserved, RES0.

Accessing the MDCCINT_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, MDCCINT_EL1

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return MDCCINT_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return MDCCINT_EL1;
elsif PSTATE.EL == EL3 then
    return MDCCINT_EL1;
```

MSR MDCCINT_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        MDCCINT_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        MDCCINT_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    MDCCINT_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.3.16 MDCCSR_EL0, Monitor DCC Status Register

The MDCCSR_EL0 characteristics are:

Purpose

Read-only register containing control status flags for the DCC.

Configurations

AArch64 System register MDCCSR_EL0[30:29] is architecturally mapped to External register EDSCR[30:29].

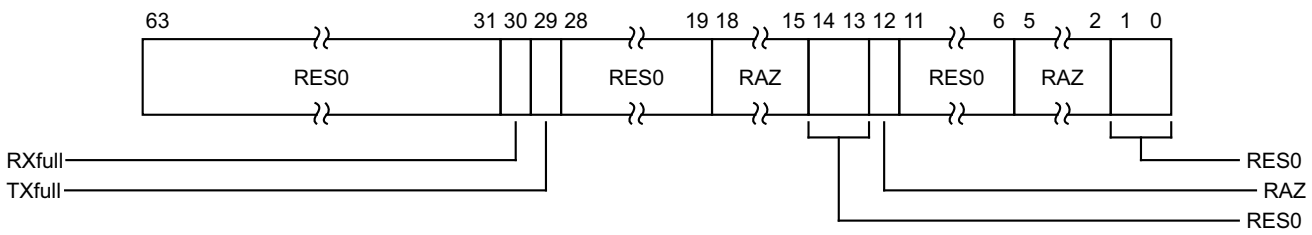
AArch64 System register MDCCSR_EL0[30:29] is architecturally mapped to AArch32 System register DBGDSCRint[30:29].

Attributes

MDCCSR_EL0 is a 64-bit register.

Field descriptions

The MDCCSR_EL0 bit assignments are:



Bits [63:31]

Reserved, RES0.

RXfull, bit [30]

DTRRX full. Read-only view of the equivalent bit in the EDSCR.

TXfull, bit [29]

DTRTX full. Read-only view of the equivalent bit in the EDSCR.

Bits [28:19]

Reserved, RES0.

Bits [18:15]

Reserved, RAZ.

Bits [14:13]

Reserved, RES0.

Bit [12]

Reserved, RAZ.

Bits [11:6]

Reserved, RES0.

Bits [5:2]

Reserved, RAZ.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bits [1:0]

Reserved, RES0.

Accessing the MDCCSR_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, MDCCSR_EL0

op0	op1	CRn	CRm	op2
0b10	0b011	0b0000	0b0001	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && MDSCR_EL1.TDCC == '1' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (HCR_EL2.TGE == '1' || MDCR_EL2.<TDE,TDA> != '00') then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return MDCCSR_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return MDCCSR_EL0;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return MDCCSR_EL0;
elsif PSTATE.EL == EL3 then
    return MDCCSR_EL0;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.3.17 MDCR_EL2, Monitor Debug Configuration Register (EL2)

The MDCR_EL2 characteristics are:

Purpose

Provides EL2 configuration options for self-hosted debug and the Performance Monitors Extension.

Configurations

AArch64 System register MDCR_EL2[31:0] is architecturally mapped to AArch32 System register HDCR[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

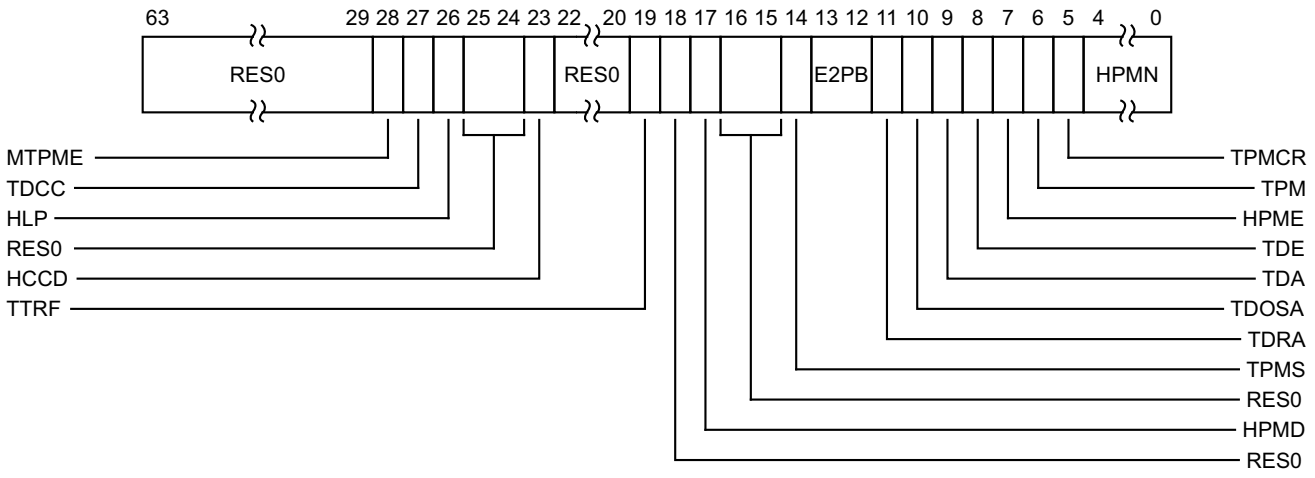
This register has no effect if EL2 is not enabled in the current Security state.

Attributes

MDCR_EL2 is a 64-bit register.

Field descriptions

The MDCR_EL2 bit assignments are:



Bits [63:29]

Reserved, RES0.

MTPME, bit [28]

When ARMv8.6-MTPMU is implemented and EL3 is not implemented:

Multi-threaded PMU Enable. Enables use of the PMEVTYPER<n>_EL0.MT bits.

`0b0` ARMv8.6-MTPMU is disabled. The Effective value of PMEVTYPER<n>_EL0.MT is zero.

`0b1` PMEVTYPER<n>_EL0.MT bits not affected by this bit.

If ARMv8.6-MTPMU is disabled for any other PE in the system that has the same level 1 Affinity as the PE, it is IMPLEMENTATION DEFINED whether the PE behaves as if this bit is 0.

On a Cold reset, this field resets to `1`.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TDCC, bit [27]

When ARMv8.6-FGT is implemented:

Trap DCC. Traps use of the Debug Comms Channel at EL1 and EL0 to EL2.

`0b0` This control does not cause any register accesses to be trapped.

`0b1` If EL2 is implemented and enabled in the current Security state, accesses to the DCC registers at EL1 and EL0 generate a Trap exception to EL2, unless the access also generates a higher priority exception.

Traps on the DCC data transfer registers are ignored when the PE is in Debug state.

The DCC registers trapped by this control are:

AArch64: OSDTRRX_EL1, OSDTRTX_EL1, MDCCSR_EL0, MDCCINT_EL1, and, when the PE is in Non-debug state, DBGDTR_EL0, DBGDTRRX_EL0, and DBGDTRTX_EL0.

AArch32: DBGDTRRXext, DBGDTRTXext, DBGDSCRint, DBGDCCINT, and, when the PE is in Non-debug state, DBGDTRRXint and DBGDTRTXint.

The traps are reported with EC syndrome value:

- `0x05` for trapped AArch32 `MRC` and `MCR` accesses with `coproc` == `0b1110`.
- `0x06` for trapped AArch32 `LDC` to DBGDTRTXint and `STC` from DBGDTRRXint.
- `0x18` for trapped AArch64 `MRS` and `MSR` accesses.

When the PE is in Debug state, MDCR_EL2.TDCC does not trap any accesses to:

AArch64: DBGDTR_EL0, DBGDTRRX_EL0, and DBGDTRTX_EL0.

AArch32: DBGDTRRXint and DBGDTRTXint.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HLP, bit [26]

When ARMv8.5-PMU is implemented:

Hypervisor Long event counter enable. Determines when unsigned overflow is recorded by a counter overflow bit.

`0b0` Event counter overflow on increment that causes unsigned overflow of PMEVCNTR<n>_EL0[31:0].

`0b1` Event counter overflow on increment that causes unsigned overflow of PMEVCNTR<n>_EL0[63:0].

If MDCR_EL2.HPMN is less than PMCR_EL0.N or PMCR.N, this bit affects the operation of event counters in the range [MDCR_EL2.HPMN..(PMCR_EL0.N-1)] or [MDCR_EL2.HPMN..(PMCR.N-1)]. Otherwise this bit has no effect on the operation of the event counters.

Note

The effect of MDCR_EL2.HPMN on the operation of this bit always applies if EL2 is implemented, at all Exception levels including EL2 and EL3, and regardless of whether EL2 is enabled in the current Security state.

For more information see the description of the MDCR_EL2.HPMN field.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [25:24]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

HCCD, bit [23]

When ARMv8.5-PMU is implemented:

Hypervisor Cycle Counter Disable. Prohibits PMCCNTR_EL0 from counting at EL2.

`0b0` Cycle counting by PMCCNTR_EL0 is not affected by this bit.

`0b1` Cycle counting by PMCCNTR_EL0 is prohibited at EL2.

This bit does not affect the CPU_CYCLES event or any other event that counts cycles.

On a Warm reset, this field resets to `0`.

Otherwise:

Reserved, RES0.

Bits [22:20]

Reserved, RES0.

TTRF, bit [19]

When ARMv8.4-Trace is implemented:

Traps use of the Trace Filter Control registers at EL1 to EL2, as follows:

- Access to TRFCR_EL1 is trapped to EL2, reported using EC syndrome value `0x18`.
- Access to TRFCR is trapped to EL2, reported using EC syndrome value `0x03`.

`0b0` Accesses to TRFCR_EL1 and TRFCR at EL1 are not affected by this control.

`0b1` Accesses to TRFCR_EL1 and TRFCR at EL1 generate a trap exception to EL2 when EL2 is enabled in the current Security state.

Otherwise:

Reserved, RES0.

Bit [18]

Reserved, RES0.

HPMD, bit [17]

When ARMv8.1-PMU is implemented:

Guest Performance Monitors Disable. This control prohibits event counting at EL2.

`0b0` Event counting allowed at EL2.

`0b1` Event counting prohibited at EL2.

If ARMv8.2-Debug is not implemented, event counting is prohibited unless enabled by the IMPLEMENTATION DEFINED authentication interface ExternalSecureNoninvasiveDebugEnabled().

This control applies only to:

- The event counters in the range [0..(MDCR_EL2.HPMN-1)].
- If PMCR_EL0.DP is set to 1, PMCCNTR_EL0.

The other event counters are unaffected, and when PMCR_EL0.DP is set to 0, PMCCNTR_EL0 is unaffected.

On a Warm reset, this field resets to `0`.

Otherwise:

Reserved, RES0.

Bits [16:15]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TPMS, bit [14]

When SPE is implemented:

Trap Performance Monitor Sampling. When EL2 is enabled in the current Security state, this field controls access to Statistical Profiling control registers from EL1.

`0b0` Do not trap Statistical Profiling controls to EL2.

`0b1` Accesses to Statistical Profiling controls at EL1 generate a Trap exception to EL2 when EL2 is enabled in the current Security state.

The Statistical Profiling control registers trapped by this control are: PMSCR_EL1, PMSEVFR_EL1, PMSFCR_EL1, PMSICR_EL1, PMSIDR_EL1, PMSIRR_EL1, and PMSLATFR_EL1.

Otherwise:

Reserved, RES0.

E2PB, bits [13:12]

When SPE is implemented:

EL2 Profiling Buffer. If EL2 is implemented and enabled in the Profiling Buffer owning Security state, this field controls the owning translation regime. If EL2 is implemented and enabled in the current Security state, this field controls access to Profiling Buffer control registers from EL1.

`0b00` If EL2 is implemented and enabled in the Profiling Buffer owning Security state, the Profiling Buffer uses the EL2 or EL2&0 stage 1 translation regime. Otherwise the Profiling Buffer uses the EL1&0 stage 1 translation regime.

If EL2 is implemented and enabled in the current Security state, accesses to Profiling Buffer control registers at EL1 generate a Trap exception to EL2.

`0b10` Profiling Buffer uses the EL1&0 stage 1 translation regime. If EL2 is implemented and enabled in the current Security state, accesses to Profiling Buffer control registers at EL1 generate a Trap exception to EL2.

`0b11` Profiling Buffer uses the EL1&0 stage 1 translation regime. Accesses to Profiling Buffer control registers at EL1 are not trapped to EL2.

All other values are reserved.

The Profiling Buffer control registers trapped by this control are: PMBLIMITR_EL1, PMBPTR_EL1, and PMBSR_EL1.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TDRA, bit [11]

Trap Debug ROM Address register access. Traps System register accesses to the Debug ROM registers to EL2 when EL2 is enabled in the current Security state as follows:

- If EL1 is using AArch64 state, accesses to MDRAR_EL1 are trapped to EL2, reported using EC syndrome value `0x18`.
- If EL0 or EL1 is using AArch32 state, MRC or MCR accesses to the following registers are trapped to EL2, reported using EC syndrome value `0x05` and MRRC or MCRR accesses are trapped to EL2, reported using EC syndrome value `0x0C`:
  — DBGDRAR, DBGDSAR.

`0b0` This control does not cause any instructions to be trapped.

`0b1` EL0 and EL1 System register accesses to the Debug ROM registers are trapped to EL2 when EL2 is enabled in the current Security state, unless it is trapped by DBGDSCRext.UDCCdis or MDSCR_EL1.TDCC.

This field is treated as being 1 for all purposes other than a direct read when one or more of the following are true:

- MDCR_EL2.TDE == 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- HCR_EL2.TGE == 1.

Note

EL2 does not provide traps on debug register accesses through the optional memory-mapped external debug interfaces.

System register accesses to the debug registers might have side-effects. When a System register access is trapped to EL2, no side-effects occur before the exception is taken to EL2.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

TDOSA, bit [10]

When ARMv8.0-DoubleLock is implemented:

Trap debug OS-related register access. Traps EL1 System register accesses to the powerdown debug registers to EL2, from both Execution states as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value `0x18`:
  - OSLAR_EL1, OSLSR_EL1, OSDLR_EL1, and DBGPRCR_EL1.
  - Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this bit.
- In AArch32 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value `0x05`:
  - DBGOSLSR, DBGOSLAR, DBGOSDLR, and DBGPRCR.
  - Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this bit.

`0b0` This control does not cause any instructions to be trapped.

`0b1` EL1 System register accesses to the powerdown debug registers are trapped to EL2 when EL2 is enabled in the current Security state.

Note

These registers are not accessible at EL0.

This field is treated as being 1 for all purposes other than a direct read when one or more of the following are true:

- MDCR_EL2.TDE == 1.
- HCR_EL2.TGE == 1.

System register accesses to the debug registers might have side-effects. When a System register access is trapped to EL2, no side-effects occur before the exception is taken to EL2.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Trap debug OS-related register access. Traps EL1 System register accesses to the powerdown debug registers to EL2, from both Execution states as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value `0x18`:
  - OSLAR_EL1, OSLSR_EL1, and DBGPRCR_EL1.
  - Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this bit.
- In AArch32 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value `0x05`:
  - DBGOSLSR, DBGOSLAR, and DBGPRCR.
  - Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

It is IMPLEMENTATION DEFINED whether accesses to OSDLR_EL1 are trapped.

It is IMPLEMENTATION DEFINED whether accesses to DBGOSDLR are trapped.

`0b0` This control does not cause any instructions to be trapped.

`0b1` EL1 System register accesses to the powerdown debug registers are trapped to EL2 when EL2 is enabled in the current Security state.

Note

These registers are not accessible at EL0.

This field is treated as being 1 for all purposes other than a direct read when one or more of the following are true:

- MDCR_EL2.TDE == 1.
- HCR_EL2.TGE == 1.

Note

EL2 does not provide traps on debug register accesses through the optional memory-mapped external debug interfaces.

System register accesses to the debug registers might have side-effects. When a System register access is trapped to EL2, no side-effects occur before the exception is taken to EL2.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

TDA, bit [9]

Trap Debug Access. Traps EL0 and EL1 System register accesses to debug System registers that are not trapped by MDCR_EL2.TDRA or MDCR_EL2.TDOSA, as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2 reported using EC syndrome value `0x18`:
  - MDCCSR_EL0, MDCCINT_EL1, OSDTRRX_EL1, MDSCR_EL1, OSDTRTX_EL1, OSECCR_EL1, DBGBVR<n>_EL1, DBGBCR<n>_EL1, DBGWVR<n>_EL1, DBGWCR<n>_EL1, DBGCLAIMSET_EL1, DBGCLAIMCLR_EL1, DBGAUTHSTATUS_EL1.
  - When not in Debug state, DBGDTR_EL0, DBGDTRRX_EL0, DBGDTRTX_EL0.
- In AArch32 state, MRC or MCR accesses to the following registers are trapped to EL2, reported using EC syndrome value `0x05`.
  - DBGDIDR, DBGDSCRint, DBGDCCINT, DBGWFAR, DBGVCR, DBGDSCRext, DBGDTRTXext, DBGDTRRXext, DBGBVR<n>, DBGBCR<n>, DBGBXVR<n>, DBGWCR<n>, DBGWVR<n>, DBGCLAIMSET, DBGCLAIMCLR, DBGAUTHSTATUS, DBGDEVID, DBGDEVID1, DBGDEVID2, DBGOSECCR.
  - When not in Debug state, DBGDTRRXint and DBGDTRTXint.
- In AArch32 state, STC accesses to DBGDTRRXint and LDC accesses to DBGDTRTXint are trapped to EL2, reported using EC syndrome value `0x06`.

`0b0` This control does not cause any instructions to be trapped.

`0b1` EL0 or EL1 System register accesses to the debug registers are trapped from both Execution states to EL2 when EL2 is enabled in the current Security state, unless the access generates a higher priority exception.

Traps of AArch32 accesses to DBGDTRRXint and DBGDTRTXint are ignored in Debug state.

Traps of AArch64 accesses to DBGDTR_EL0, DBGDTRRX_EL0, and DBGDTRTX_EL0 are ignored in Debug state.

This field is treated as being 1 for all purposes other than a direct read when one or more of the following are true:

- MDCR_EL2.TDE == 1
- HCR_EL2.TGE == 1

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

On a Warm reset, this field resets to an architecturally UNKNOWN value.

TDE, bit [8]

Trap Debug exceptions.

`0b0` This control has no effect on the routing of debug exceptions, and has no effect on accesses to debug registers.

`0b1` Debug exceptions generated at EL1 or EL0 are routed to EL2 when EL2 is enabled in the current Security state. The MDCR_EL2.{TDRA, TDOSA, TDA} fields are treated as being 1 for all purposes other than returning the result of a direct read of the register.

This field is treated as being 1 for all purposes other than a direct read when HCR_EL2.TGE == 1.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

HPME, bit [7]

When PMUv3 is implemented:

[MDCR_EL2.HPMN..(N-1)] event counters enable.

`0b0` Event counters in the range [MDCR_EL2.HPMN..(PMCR_EL0.N-1)] are disabled.

`0b1` Event counters in the range [MDCR_EL2.HPMN..(PMCR_EL0.N-1)] are enabled by PMCNTENSET_EL0.

If MDCR_EL2.HPMN is less than PMCR_EL0.N or PMCR.N, the event counters in the range [MDCR_EL2.HPMN..(PMCR_EL0.N-1)] or [HDCR.HPMN..(PMCR.N-1)], are enabled and disabled by this bit. Otherwise this bit has no effect on the operation of the event counters.

Note

The effect of MDCR_EL2.HPMN on the operation of this bit applies regardless of whether EL2 is enabled in the current Security state.

For more information see the description of the HPMN field.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TPM, bit [6]

When PMUv3 is implemented:

Trap Performance Monitors accesses. Traps EL0 and EL1 accesses to all Performance Monitor registers to EL2 when EL2 is enabled in the current Security state, from both Execution states, as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value `0x18`:
  - PMCR_EL0, PMCNTENSET_EL0, PMCNTENCLR_EL0, PMOVSCLR_EL0, PMSWINC_EL0, PMSELR_EL0, PMCEID0_EL0, PMCEID1_EL0, PMCCNTR_EL0, PMXEVTYPER_EL0, PMXEVCNTR_EL0, PMUSERENR_EL0, PMINTENSET_EL1, PMINTENCLR_EL1, PMOVSSET_EL0, PMEVCNTR<n>_EL0, PMEVTYPER<n>_EL0, PMCCFILTR_EL0.
  - If ARMv8.4-PMU is implemented, PMMIR_EL1
- In AArch32 state, MRC or MCR accesses to the following registers are trapped to EL2 and reported using EC syndrome value `0x03`, MRRC or MCRR accesses are trapped to EL2 and reported using EC syndrome value `0x04`:
  - PMCR, PMCNTENSET, PMCNTENCLR, PMOVSR, PMSWINC, PMSELR, PMCEID0, PMCEID1, PMCCNTR, PMXEVTYPER, PMXEVCNTR, PMUSERENR, PMINTENSET, PMINTENCLR, PMOVSSET, PMEVCNTR<n>, PMEVTYPER<n>, PMCCFILTR.
  - If ARMv8.4-PMU is implemented, PMMIR.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— If ARMv8.1-PMU is implemented, PMCEID2, and PMCEID3.

`0b0` This control does not cause any instructions to be trapped.

`0b1` EL0 and EL1 accesses to all Performance Monitor registers are trapped to EL2 when EL2 is enabled in the current Security state.

Note

EL2 does not provide traps on Performance Monitor register accesses through the optional memory-mapped external debug interface.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TPMCR, bit [5]

When PMUv3 is implemented:

Trap PMCR_EL0 or PMCR accesses. Traps EL0 and EL1 accesses to EL2, when EL2 is enabled in the current Security state, as follows:

- In AArch64 state, accesses to PMCR_EL0 are trapped to EL2, reported using EC syndrome value `0x18`.
- In AArch32 state, accesses to PMCR are trapped to EL2, reported using EC syndrome value `0x03`.

`0b0` This control does not cause any instructions to be trapped.

`0b1` EL0 and EL1 accesses to the PMCR_EL0 or PMCR are trapped to EL2 when EL2 is enabled in the current Security state, unless it is trapped by PMUSERENR.EN or PMUSERENR_EL0.EN.

Note

EL2 does not provide traps on Performance Monitor register accesses through the optional memory-mapped external debug interface.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HPMN, bits [4:0]

When PMUv3 is implemented:

Defines the number of event counters that are accessible from EL3, EL2, EL1, and from EL0 if permitted.

If HPMN is less than PMCR_EL0.N, HPMN divides the Performance Monitors into two ranges: [0..(HPMN-1)] and [HPMN..(PMCR_EL0.N-1)].

For an event counter in the range [0..(HPMN-1)]:

- The counter is accessible from EL3, EL2, and EL1, and from EL0 if permitted by PMUSERENR_EL0 or PMUSERENR.
- If ARMv8.5-PMU is implemented, PMCR_EL0.LP or PMCR.LP determines whether the counter overflow flag is set on unsigned overflow of PMEVCNTR<n>_EL0[31:0] or PMEVCNTR<n>_EL0[63:0].
- The counter is enabled by PMCR_EL0.E or PMCR.E and bit <n> of PMCNTENSET_EL0.

Note

If HPMN is equal to PMCR_EL0.N, this applies to all event counters.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If HPMN is less than PMCR_EL0.N, for an event counter in the range [HPMN..(PMCR_EL0.N-1)]:

- The counter is accessible from EL2 and EL3.
- If ARMv8.4-SecEL2 is disabled or is not implemented, the counter is also accessible from Secure EL1, and from Secure EL0 if permitted by PMUSERENR_EL0.
- If ARMv8.5-PMU is implemented, MDCR_EL2.HLP or HDCR.HLP determines whether the counter overflow flag is set on unsigned overflow of PMEVCNTR<n>_EL0[31:0] or PMEVCNTR<n>_EL0[63:0].
- The counter is enabled by MDCR_EL2.HPME or HDCR.HPME and bit <n> of PMCNTENSET_EL0.

If this field is set to 0, or to a value larger than PMCR_EL0.N, then the following CONSTRAINED UNPREDICTABLE behaviors apply:

- The value returned by a direct read of MDCR_EL2.HPMN is UNKNOWN.
- Either:
  - An UNKNOWN number of counters are reserved for EL2 and EL3 use. That is, the PE behaves as if MDCR_EL2.HPMN is set to an UNKNOWN non-zero value less than or equal to PMCR_EL0.N.
  - All counters are reserved for EL2 and EL3 use, meaning no counters are accessible from EL1 and EL0.

On a Warm reset, this field resets to the value in PMCR_EL0.N.

Otherwise:

Reserved, RES0.

Accessing the MDCR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, MDCR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return MDCR_EL2;
elsif PSTATE.EL == EL3 then
    return MDCR_EL2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR MDCR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        MDCR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    MDCR_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.3.18 MDCR_EL3, Monitor Debug Configuration Register (EL3)

The MDCR_EL3 characteristics are:

Purpose

Provides EL3 configuration options for self-hosted debug and the Performance Monitors Extension.

Configurations

AArch64 System register MDCR_EL3[31:0] can be mapped to AArch32 System register SDCR[31:0], but this is not architecturally mandated.

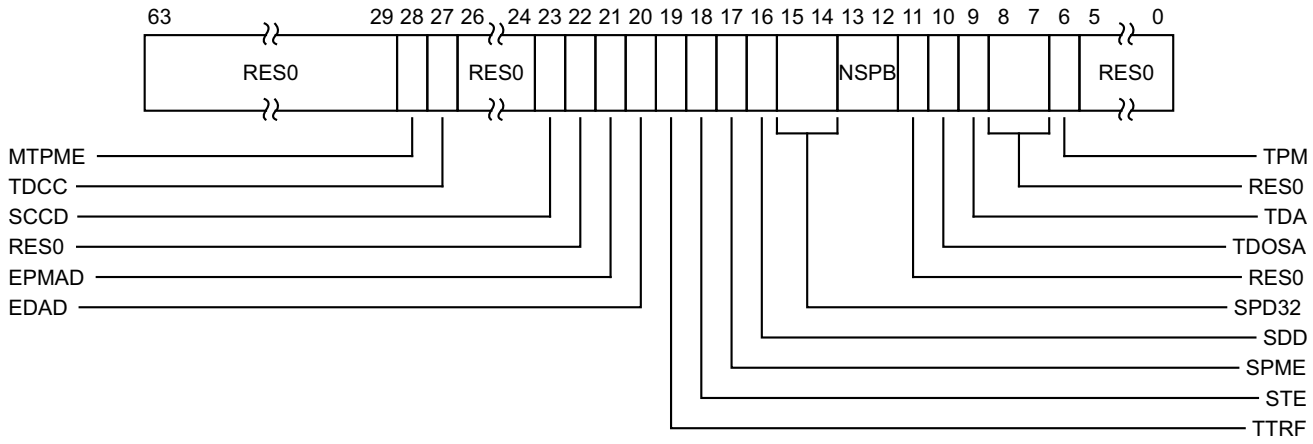
This register is present only when EL3 is implemented. Otherwise, direct accesses to MDCR_EL3 are UNDEFINED.

Attributes

MDCR_EL3 is a 64-bit register.

Field descriptions

The MDCR_EL3 bit assignments are:



Bits [63:29]

Reserved, RES0.

MTPME, bit [28]

When ARMv8.6-MTPMU is implemented:

Multi-threaded PMU Enable. Enables use of the PMEVTYPER<n>_EL0.MT bits.

`0b0` ARMv8.6-MTPMU is disabled. The Effective value of PMEVTYPER<n>_EL0.MT is zero.

`0b1` PMEVTYPER<n>_EL0.MT bits not affected by this bit.

If ARMv8.6-MTPMU is disabled for any other PE in the system that has the same level 1 Affinity as the PE, it is IMPLEMENTATION DEFINED whether the PE behaves as if this bit is 0.

On a Cold reset, this field resets to `1`.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TDCC, bit [27]

When ARMv8.6-FGT is implemented:

Trap DCC. Traps use of the Debug Comms Channel at EL2, EL1, and EL0 to EL3.

`0b0` This control does not cause any register accesses to be trapped.

`0b1` Accesses to the DCC registers at EL2, EL1, and EL0 generate a Trap exception to EL3, unless the access also generates a higher priority exception.

Traps on the DCC data transfer registers are ignored when the PE is in Debug state.

The DCC registers trapped by this control are:

AArch64: OSDTRRX_EL1, OSDTRTX_EL1, MDCCSR_EL0, MDCCINT_EL1, and, when the PE is in Non-debug state, DBGDTR_EL0, DBGDTRRX_EL0, and DBGDTRTX_EL0.

AArch32: DBGDTRRXext, DBGDTRTXext, DBGDSCRint, DBGDCCINT, and, when the PE is in Non-debug state, DBGDTRRXint and DBGDTRTXint.

The traps are reported with EC syndrome value:

- `0x05` for trapped AArch32 `MRC` and `MCR` accesses with `coproc` == `0b1110`.
- `0x06` for trapped AArch32 `LDC` to DBGDTRTXint and `STC` from DBGDTRRXint.
- `0x18` for trapped AArch64 `MRS` and `MSR` accesses.

When the PE is in Debug state, MDCR_EL3.TDCC does not trap any accesses to:

AArch64: DBGDTR_EL0, DBGDTRRX_EL0, and DBGDTRTX_EL0.

AArch32: DBGDTRRXint and DBGDTRTXint.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [26:24]

Reserved, RES0.

SCCD, bit [23]

When ARMv8.5-PMU is implemented:

Secure Cycle Counter Disable. Prohibits PMCCNTR_EL0 from counting in Secure state.

`0b0` Cycle counting by PMCCNTR_EL0 is not affected by this bit.

`0b1` Cycle counting by PMCCNTR_EL0 is prohibited in Secure state.

This bit does not affect the CPU_CYCLES event or any other event that counts cycles.

On a Warm reset, this field resets to `0`.

Otherwise:

Reserved, RES0.

Bit [22]

Reserved, RES0.

EPMAD, bit [21]

When ARMv8.4-Debug is implemented and PMUv3 is implemented:

External Performance Monitors Non-secure Access Disable. Controls Non-secure access to Performance Monitor registers by an external debugger.

`0b0` Non-secure access to Performance Monitor registers from external debugger is permitted.

`0b1` Non-secure access to Performance Monitor registers from external debugger is not permitted.

If the Performance Monitors Extension does not support external debug interface accesses this bit is RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise, if EL3 is not implemented and the Effective value of SCR_EL3.NS is `0b0`, then the Effective value of this bit is `0b1`.

On a Warm reset, this field resets to `0`.

When PMUv3 is implemented:

External Performance Monitors Access Disable. Controls access to Performance Monitor registers by an external debugger.

`0b0` Access to Performance Monitor registers from external debugger is permitted.

`0b1` Access to Performance Monitor registers from external debugger is not permitted, unless overridden by the IMPLEMENTATION DEFINED authentication interface.

If the Performance Monitors Extension does not support external debug interface accesses this bit is RES0.

Otherwise, if EL3 is not implemented and the Effective value of SCR_EL3.NS is `0b0`, then the Effective value of this bit is `0b1`.

On a Warm reset, this field resets to `0`.

Otherwise:

Reserved, RES0.

EDAD, bit [20]

When ARMv8.4-Debug is implemented:

External Debug Non-secure Access Disable. Controls Non-secure access to breakpoint, watchpoint, and OSLAR_EL1 registers by an external debugger.

`0b0` Non-secure access to debug registers from external debugger is permitted.

`0b1` Non-secure access to breakpoint and watchpoint registers, and OSLAR_EL1 from external debugger is not permitted.

If EL3 is not implemented and the Effective value of SCR_EL3.NS is `0b0`, then the Effective value of this field is `0b1`.

On a Warm reset, this field resets to `0`.

When ARMv8.2-Debug is implemented:

External Debug Access Disable. Controls access to breakpoint, watchpoint, and OSLAR_EL1 registers by an external debugger.

`0b0` Access to debug registers, and to OSLAR_EL1 from external debugger is permitted.

`0b1` Access to breakpoint and watchpoint registers, and to OSLAR_EL1 from external debugger is not permitted, unless overridden by the IMPLEMENTATION DEFINED authentication interface.

If EL3 is not implemented and the Effective value of SCR_EL3.NS is `0b0`, then the Effective value of this field is `0b1`.

On a Warm reset, this field resets to `0`.

Otherwise:

External Debug Access disable. Controls access to breakpoint, watchpoint, and optionally OSLAR_EL1 registers by an external debugger.

`0b0` Access to debug registers from external debugger is permitted.

`0b1` Access to breakpoint and watchpoint registers from an external debugger is not permitted, unless overridden by the IMPLEMENTATION DEFINED authentication interface.

It is IMPLEMENTATION DEFINED whether access to the OSLAR_EL1 register from an external debugger is permitted or not permitted.

If EL3 is not implemented and the Effective value of SCR_EL3.NS is `0b0`, then the Effective value of this field is `0b1`.

On a Warm reset, this field resets to `0`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TTRF, bit [19]

When ARMv8.4-Trace is implemented:

Trap Trace Filter controls. Traps use of the Trace Filter control registers at EL2 and EL1 to EL3.

The Trace Filter registers trapped by this control are:

- TRFCR_EL2, TRFCR_EL12, TRFCR_EL1, reported using EC syndrome value `0x18`.
- HTRFCR and TRFCR, reported using EC syndrome value `0x03`.

`0b0` Accesses to Trace Filter registers at EL2 and EL1 are not affected by this bit.

`0b1` Accesses to Trace Filter registers at EL2 and EL1 generate a Trap exception to EL3, unless the access generates a higher priority exception.

Otherwise:

Reserved, RES0.

STE, bit [18]

When ARMv8.4-Trace is implemented:

Secure Trace enable. Enables tracing in Secure state.

`0b0` Trace prohibited in Secure state unless overridden by the IMPLEMENTATION DEFINED authentication interface.

`0b1` Trace in Secure state is not affected by this bit.

This bit also controls the level of authentication required by an external debugger to enable external tracing. See *Register controls to enable self-hosted trace* on page G3-5880.

If EL3 is not implemented the Effective value of SCR_EL3.NS is `0b0`, the Effective value of this bit is `0b1`.

On a Warm reset, this field resets to `0`.

Otherwise:

Reserved, RES0.

SPME, bit [17]

When ARMv8.2-Debug is implemented and PMUv3 is implemented:

Secure Performance Monitors enable. This allows event counting in Secure state.

`0b0` Event counting prohibited in Secure state.

`0b1` Event counting in Secure state not affected by this bit.

If EL3 is not implemented the Effective value of SCR_EL3.NS is `0b0`, then the Effective value of this bit is `0b1`.

On a Warm reset, this field resets to `0`.

When PMUv3 is implemented:

Secure Performance Monitors enable. This allows event counting in Secure state.

`0b0` Event counting prohibited in Secure state, unless ExternalSecureNoninvasiveDebugEnabled() is TRUE.

`0b1` Event counting in Secure state not affected by this bit.

If EL3 is not implemented the Effective value of SCR_EL3.NS is `0b0`, then the Effective value of this bit is `0b1`.

On a Warm reset, this field resets to `0`.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

SDD, bit [16]

AArch64 Secure Self-hosted invasive debug disable. Disables Software debug exceptions in Secure state, other than Breakpoint Instruction exceptions.

`0b0` Debug exceptions in Secure state are not affected by this bit.

`0b1` Debug exceptions, other than Breakpoint Instruction exceptions, are disabled from all Exception levels in Secure state.

The SDD bit is ignored unless both of the following are true:

- The PE is in Secure state.
- The Effective value of SCR_EL3.RW is `0b1`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

SPD32, bits [15:14]

When EL1 is capable of using AArch32:

AArch32 Secure self-hosted privileged debug. Enables or disables debug exceptions from Secure EL1 using AArch32, other than Breakpoint Instruction exceptions.

`0b00` Legacy mode. Debug exceptions from Secure EL1 are enabled by the IMPLEMENTATION DEFINED authentication interface.

`0b10` Secure privileged debug disabled. Debug exceptions from Secure EL1 are disabled.

`0b11` Secure privileged debug enabled. Debug exceptions from Secure EL1 are enabled.

Other values are reserved, and have the CONSTRAINED UNPREDICTABLE behavior that they must have the same behavior as `0b00`. Software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture.

This field has no effect on Breakpoint Instruction exceptions. These are always enabled.

This field is ignored if the PE is either:

- In Non-secure state.
- In Secure state and Secure EL1 is using AArch64.

If Secure EL1 is using AArch32 then:

- If debug exceptions from Secure EL1 are enabled, then debug exceptions from Secure EL0 are also enabled.
- Otherwise, debug exceptions from Secure EL0 are enabled only if the value of SDER32_EL3.SUIDEN is `0b1`.

If EL3 is not implemented and the Effective value of SCR_EL3.NS is `0b0`, then the Effective value of this field is `0b11`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSPB, bits [13:12]

When SPE is implemented:

Non-secure Profiling Buffer. This field controls the owning translation regime and accesses to Statistical Profiling and Profiling Buffer control registers.

`0b00` Profiling Buffer uses Secure Virtual Addresses. Statistical Profiling enabled in Secure state and disabled in Non-secure state. Accesses to Statistical Profiling and Profiling Buffer control registers at EL2 and EL1 in both security states generate Trap exceptions to EL3.

`0b01` Profiling Buffer uses Secure Virtual Addresses. Statistical Profiling enabled in Secure state and disabled in Non-secure state. Accesses to Statistical Profiling and Profiling Buffer control registers at EL2 and EL1 in Non-secure state generate Trap exceptions to EL3.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b10` Profiling Buffer uses Non-secure Virtual Addresses. Statistical Profiling enabled in Non-secure state and disabled in Secure state. Accesses to Statistical Profiling and Profiling Buffer control registers at EL2 and EL1 in both security states generate Trap exceptions to EL3.

`0b11` Profiling Buffer uses Non-secure Virtual Addresses. Statistical Profiling enabled in Non-secure state and disabled in Secure state. Accesses to Statistical Profiling and Profiling Buffer control registers at EL2 and EL1 in Secure state generate Trap exceptions to EL3.

The Statistical Profiling and Profiling Buffer control registers trapped by this control are: PMBLIMITR_EL1, PMBPTR_EL1, PMBSR_EL1, PMSCR_EL1, PMSCR_EL2, PMSEVFR_EL1, PMSFCR_EL1, PMSICR_EL1, PMSIDR_EL1, PMSIRR_EL1, and PMSLATFR_EL1.

If EL3 is not implemented and the Effective value of SCR_EL3.NS is `0b1`, the Effective value of this field is `0b11`.

If EL3 is not implemented and the Effective value of SCR_EL3.NS is `0b0`, the Effective value of this field is `0b01`.

On a Warm reset, this field resets to an UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [11]

Reserved, RES0.

TDOSA, bit [10]

When ARMv8.0-DoubleLock is implemented:

Trap debug OS-related register access. Traps EL2 and EL1 System register accesses to the powerdown debug registers to EL3.

Accesses to the registers are trapped as follows:

- Accesses from AArch64 state, OSLAR_EL1, OSLSR_EL1, OSDLR_EL1, DBGPRCR_EL1 and any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this bit, are trapped to EL3 and reported using EC syndrome value `0x18`.
- Accesses using MCR or MRC to DBGOSLAR, DBGOSLSR, DBGOSDLR, and DBGPRCR, are trapped to EL3 and reported using EC syndrome value `0x05`.
- Accesses to any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this bit.

`0b0` This control does not cause any instructions to be trapped.

`0b1` EL2 and EL1 System register accesses to the powerdown debug registers are trapped to EL3, unless it is trapped by HDCR.TDOSA or MDCR_EL2.TDOSA.

Note

The powerdown debug registers are not accessible at EL0.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Trap debug OS-related register access. Traps EL2 and EL1 System register accesses to the powerdown debug registers to EL3.

The following registers are affected by this trap:

- AArch64: OSLAR_EL1, OSLSR_EL1, and DBGPRCR_EL1.
- AArch32: DBGOSLAR, DBGOSLSR, and DBGPRCR.
- AArch64 and AArch32: Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- It is IMPLEMENTATION DEFINED whether accesses to OSDLR_EL1 and DBGOSDLR are trapped.

`0b0` This control does not cause any instructions to be trapped.

`0b1` EL2 and EL1 System register accesses to the powerdown debug registers are trapped to EL3, unless it is trapped by HDCR.TDOSA or MDCR_EL2.TDOSA.

Note

The powerdown debug registers are not accessible at EL0.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

TDA, bit [9]

Trap Debug Access. Traps EL2, EL1, and EL0 System register accesses to those debug System registers that cannot be trapped using the MDCR_EL3.TDOSA field.

Accesses to the debug registers are trapped as follows:

- In AArch64 state, the following registers are trapped to EL3 and reported using EC syndrome value `0x18`:
  - DBGBVR<n>_EL1, DBGBCR<n>_EL1, DBGWVR<n>_EL1, DBGWCR<n>_EL1, DBGCLAIMSET_EL1, DBGCLAIMCLR_EL1, DBGAUTHSTATUS_EL1, DBGVCR32_EL2.
  - AArch64: MDCR_EL2, MDRAR_EL1, MDCCSR_EL0, MDCCINT_EL1, MDSCR_EL1, OSDTRRX_EL1, OSDTRTX_EL1, OSECCR_EL1.
- In AArch32 state, SDER is trapped to EL3 and reported using EC syndrome value `0x03`.
- In AArch32 state, accesses using MCR or MRC to the following registers are reported using EC syndrome value `0x05`, accesses using MCRR or MRRC are reported using EC syndrome value `0x0C`:
  - HDCR, DBGDRAR, DBGDSAR, DBGDIDR, DBGDCCINT, DBGWFAR, DBGVCR, DBGBVR<n>, DBGBCR<n>, DBGBXVR<n>, DBGWCR<n>, DBGWVR<n>.
  - DBGCLAIMSET, DBGCLAIMCLR, DBGAUTHSTATUS, DBGDEVID, DBGDEVID1, DBGDEVID2, DBGOSECCR.
- In AArch32 state, STC accesses to DBGDTRRXint and LDC accesses to DBGDTRTXint are reported using EC syndrome value `0x06`.
- When not in Debug state, the following registers are also trapped to EL3:
  - AArch64 accesses to DBGDTR_EL0, DBGDTRRX_EL0, and DBGDTRTX_EL0, reported using EC syndrome value `0x18`.
  - AArch32 accesses using MCR or MRC to DBGDTRRXint and DBGDTRTXint, reported using EC syndrome value `0x05`.

`0b0` This control does not cause any instructions to be trapped.

`0b1` EL0, EL1, and EL2 accesses to the debug registers, other than the registers that can be trapped by MDCR_EL3.TDOSA, are trapped to EL3, from both Security states and both Execution states, unless it is trapped by DBGDSCRext.UDCCdis, MDSCR_EL1.TDCC, HDCR.TDA or MDCR_EL2.TDA.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [8:7]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TPM, bit [6]

When PMUv3 is implemented:

Trap Performance Monitor register accesses. Accesses to all Performance Monitor registers from EL0, EL1 and EL2 to EL3, from both Security states and both Execution states are trapped as follows:

- In AArch64 state, accesses to the following registers are trapped to EL3 and are reported using EC syndrome value `0x18`:
  - PMCR_EL0, PMCNTENSET_EL0, PMCNTENCLR_EL0, PMOVSCLR_EL0, PMSWINC_EL0, PMSELR_EL0, PMCEID0_EL0, PMCEID1_EL0, PMCCNTR_EL0, PMXEVTYPER_EL0, PMXEVCNTR_EL0, PMUSERENR_EL0, PMINTENSET_EL1, PMINTENCLR_EL1, PMOVSSET_EL0, PMEVCNTR<n>_EL0, PMEVTYPER<n>_EL0, PMCCFILTR_EL0.
  - If ARMv8.4-PMU is implemented, PMMIR_EL1
- In AArch32 state, accesses using MCR or MRC to the following registers are reported using EC syndrome value `0x03`, accesses using MCRR or MRRC are reported using EC syndrome value `0x04`:
  - PMCR, PMCNTENSET, PMCNTENCLR, PMOVSR, PMSWINC, PMSELR, PMCEID0, PMCEID1, PMCCNTR, PMXEVTYPER, PMXEVCNTR, PMUSERENR, PMINTENSET, PMINTENCLR, PMOVSSET, PMEVCNTR<n>, PMEVTYPER<n>, PMCCFILTR.
  - If ARMv8.1-PMU is implemented, PMCEID2, and PMCEID3.
  - If ARMv8.4-PMU is implemented, PMMIR.

`0b0` This control does not cause any instructions to be trapped.

`0b1` EL2, EL1, and EL0 System register accesses to all Performance Monitor registers are trapped to EL3, unless it is trapped by HDCR.TPM or MDCR_EL2.TPM.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [5:0]

Reserved, RES0.

Accessing the MDCR_EL3

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, MDCR_EL3

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0011	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    return MDCR_EL3;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR MDCR_EL3, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0011	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    MDCR_EL3 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.3.19 MDRAR_EL1, Monitor Debug ROM Address Register

The MDRAR_EL1 characteristics are:

Purpose

Defines the base physical address of a 4KB-aligned memory-mapped debug component, usually a ROM table that locates and describes the memory-mapped debug components in the system. Armv8 deprecates any use of this register.

Configurations

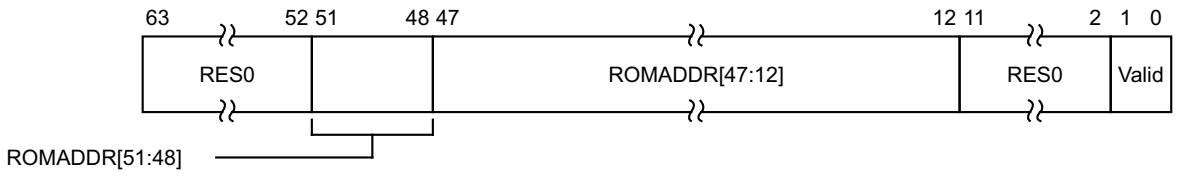
AArch64 System register MDRAR_EL1[63:0] is architecturally mapped to AArch32 System register DBGDRAR[63:0].

Attributes

MDRAR_EL1 is a 64-bit register.

Field descriptions

The MDRAR_EL1 bit assignments are:



Bits [63:52]

Reserved, RES0.

ROMADDR[51:48], bits [51:48]

When ARMv8.2-LPA is implemented:

Extension to ROMADDR[47:12]. See ROMADDR[47:12] for more details.

Otherwise:

Reserved, RES0.

ROMADDR[47:12], bits [47:12]

Bits[47:12] of the ROM table physical address.

When ARMv8.2-LPA is implemented, ROMADDR[51:48] forms the upper part of the address value. Otherwise, ROMADDR[51:48] is RES0.

If the physical address size in bits (PAsize) is less than 52, then the register bits corresponding to ROMADDR [51:PAsize] are RES0.

Bits [11:0] of the ROM table physical address are zero.

Arm strongly recommends that bits ROMADDR[(PAsize-1):32] are zero in any system that supports AArch32 at the highest implemented Exception level.

In an implementation that includes EL3, ROMADDR is an address in Non-secure memory. It is IMPLEMENTATION DEFINED whether the ROM table is also accessible in Secure memory.

Bits [11:2]

Reserved, RES0.

Valid, bits [1:0]

This field indicates whether the ROM Table address is valid. The permitted values of this field are:

`0b00` ROM Table address is not valid. Software must ignore ROMADDR.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b11` ROM Table address is valid.

Other values are reserved.

Accessing the MDRAR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, MDRAR_EL1

op0	op1	CRn	CRm	op2
0b10	0b000	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDRA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return MDRAR_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return MDRAR_EL1;
elsif PSTATE.EL == EL3 then
    return MDRAR_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.3.20 MDSCR_EL1, Monitor Debug System Control Register

The MDSCR_EL1 characteristics are:

Purpose

Main control register for the debug implementation.

Configurations

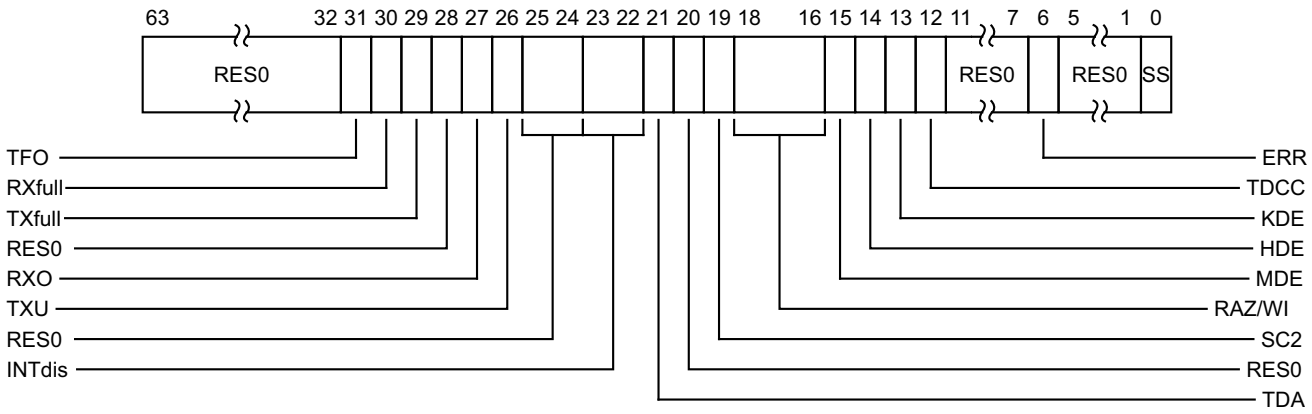
AArch64 System register MDSCR_EL1[31:0] is architecturally mapped to AArch32 System register DBGDSCRext[31:0].

Attributes

MDSCR_EL1 is a 64-bit register.

Field descriptions

The MDSCR_EL1 bit assignments are:



Bits [63:32]

Reserved, RES0.

TFO, bit [31]

When ARMv8.4-Trace is implemented:

Trace Filter override. Used for save/restore of EDSCR.TFO.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.TFO. Reads and writes of this bit are indirect accesses to EDSCR.TFO.

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == 1, access to this field is RW.
- When OSLSR_EL1.OSLK == 0, access to this field is RO.

Otherwise:

Reserved, RES0.

RXfull, bit [30]

Used for save/restore of EDSCR.RXfull.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.RXfull. Reads and writes of this bit are indirect accesses to EDSCR.RXfull.

The architected behavior of this field determines the value it returns after a reset.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == 1, access to this field is RW.
- When OSLSR_EL1.OSLK == 0, access to this field is RO.

TXfull, bit [29]

Used for save/restore of EDSCR.TXfull.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.TXfull. Reads and writes of this bit are indirect accesses to EDSCR.TXfull.

The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == 1, access to this field is RW.
- When OSLSR_EL1.OSLK == 0, access to this field is RO.

Bit [28]

Reserved, RES0.

RXO, bit [27]

Used for save/restore of EDSCR.RXO.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.RXO. Reads and writes of this bit are indirect accesses to EDSCR.RXO.

The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == 1, access to this field is RW.
- When OSLSR_EL1.OSLK == 0, access to this field is RO.

TXU, bit [26]

Used for save/restore of EDSCR.TXU.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.TXU. Reads and writes of this bit are indirect accesses to EDSCR.TXU.

The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == 1, access to this field is RW.
- When OSLSR_EL1.OSLK == 0, access to this field is RO.

Bits [25:24]

Reserved, RES0.

INTdis, bits [23:22]

Used for save/restore of EDSCR.INTdis.

When OSLSR_EL1.OSLK == 0, and software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this field holds the value of EDSCR.INTdis. Reads and writes of this field are indirect accesses to EDSCR.INTdis.

The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == 1, access to this field is RW.
- When OSLSR_EL1.OSLK == 0, access to this field is RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TDA, bit [21]

Used for save/restore of EDSCR.TDA.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.TDA. Reads and writes of this bit are indirect accesses to EDSCR.TDA.

The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == 1, access to this field is RW.
- When OSLSR_EL1.OSLK == 0, access to this field is RO.

Bit [20]

Reserved, RES0.

SC2, bit [19]

When ARMv8.0-PCSample is implemented, ARMv8.1-VHE is implemented and ARMv8.2-PCSample is not implemented:

Used for save/restore of EDSCR.SC2.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.SC2. Reads and writes of this bit are indirect accesses to EDSCR.SC2.

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == 1, access to this field is RW.
- When OSLSR_EL1.OSLK == 0, access to this field is RO.

Otherwise:

Reserved, RES0.

Bits [18:16]

Reserved, RAZ/WI.

Hardware must implement this field as RAZ/WI. Software must not rely on the register reading as zero, and must use a read-modify-write sequence to write to the register.

MDE, bit [15]

Monitor debug events. Enable Breakpoint, Watchpoint, and Vector Catch exceptions.

`0b0` Breakpoint, Watchpoint, and Vector Catch exceptions disabled.

`0b1` Breakpoint, Watchpoint, and Vector Catch exceptions enabled.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

HDE, bit [14]

Used for save/restore of EDSCR.HDE.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.HDE. Reads and writes of this bit are indirect accesses to EDSCR.HDE.

The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == 1, access to this field is RW.
- When OSLSR_EL1.OSLK == 0, access to this field is RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

KDE, bit [13]

Local (kernel) debug enable. If EL_D is using AArch64, enable debug exceptions within EL_D. Permitted values are:

`0b0` Debug exceptions, other than Breakpoint Instruction exceptions, disabled within EL_D.

`0b1` All debug exceptions enabled within EL_D.

RES0 if EL_D is using AArch32.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

TDCC, bit [12]

Traps EL0 accesses to the Debug Communication Channel (DCC) registers to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from both Execution states, as follows:

- In AArch64 state, MRS or MSR accesses to the following DCC registers are trapped, reported using EC syndrome value `0x18`:
  - MDCCSR_EL0.
  - If not in Debug state, DBGDTR_EL0, DBGDTRTX_EL0, and DBGDTRRX_EL0.
- In AArch32 state, MRC or MCR accesses to the following registers are trapped, reported using EC syndrome value `0x05`.
  - DBGDSCRint, DBGDIDR, DBGDSAR, DBGDRAR.
  - If not in Debug state, DBGDTRRXint, and DBGDTRTXint.
- In AArch32 state, LDC access to DBGDTRRXint and STC access to DBGDTRTXint are trapped, reported using EC syndrome value `0x06`.
- In AArch32 state, MRRC accesses to DBGDSAR and DBGDRAR are trapped, reported using EC syndrome value `0x0C`.

`0b0` This control does not cause any instructions to be trapped.

`0b1` EL0 using AArch64: EL0 accesses to the AArch64 DCC registers are trapped.
EL0 using AArch32: EL0 accesses to the AArch32 DCC registers are trapped.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [11:7]

Reserved, RES0.

ERR, bit [6]

Used for save/restore of EDSCR.ERR.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.ERR. Reads and writes of this bit are indirect accesses to EDSCR.ERR.

The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == 1, access to this field is RW.
- When OSLSR_EL1.OSLK == 0, access to this field is RO.

Bits [5:1]

Reserved, RES0.

SS, bit [0]

Software step control bit. If EL_D is using AArch64, enable Software step. Permitted values are:

`0b0` Software step disabled

`0b1` Software step enabled.

RES0 if EL_D is using AArch32.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the MDSCR_EL1

Individual fields within this register might have restricted accessibility when OSLSR_EL1.OSLK == 0 (the OS lock is unlocked). See the field descriptions for more detail.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, MDSCR_EL1

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0010	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.MDSCR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '1x1' then
        return NVMem[0x158];
    else
        return MDSCR_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return MDSCR_EL1;
elsif PSTATE.EL == EL3 then
    return MDSCR_EL1;
```

MSR MDSCR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0010	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.MDSCR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '1x1' then
        NVMem[0x158] = X[t];
    else
        MDSCR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        MDSCR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    MDSCR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.3.21 OSDLR_EL1, OS Double Lock Register

The OSDLR_EL1 characteristics are:

Purpose

Used to control the OS Double Lock.

Configurations

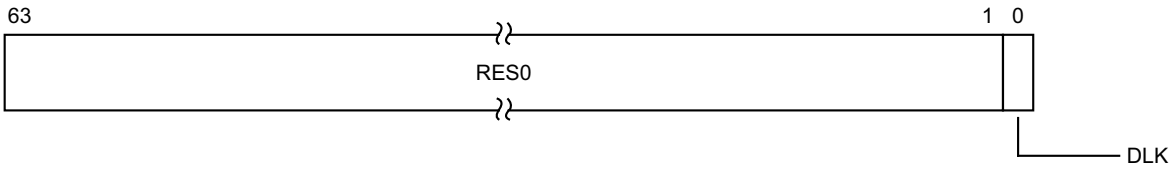
AArch64 System register OSDLR_EL1[31:0] is architecturally mapped to AArch32 System register DBGOSDLR[31:0].

Attributes

OSDLR_EL1 is a 64-bit register.

Field descriptions

The OSDLR_EL1 bit assignments are:



Bits [63:1]

Reserved, RES0.

DLK, bit [0]

When ARMv8.0-DoubleLock is implemented:

OS Double Lock control bit.

`0b0` OS Double Lock unlocked.

`0b1` OS Double Lock locked, if DBGPRCR_EL1.CORENPDRQ (Core no powerdown request) bit is set to 0 and the PE is in Non-debug state.

On a Warm reset, this field resets to `0`.

Otherwise:

Reserved, RAZ/WI.

Accessing the OSDLR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, OSDLR_EL1

op0	op1	CRn	CRm	op2
0b10	0b000	0b0001	0b0011	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
IsFeatureImplemented("ARMv8.0-DoubleLock") && HDFGRTR_EL2.OSDLR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDOSA> != '00' &&
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
(IsFeatureImplemented("ARMv8.0-DoubleLock") || boolean IMPLEMENTATION_DEFINED "Trapped by
MDCR_EL2.TDOSA") then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' &&
(IsFeatureImplemented("ARMv8.0-DoubleLock") || boolean IMPLEMENTATION_DEFINED "Trapped by
MDCR_EL3.TDOSA") then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return OSDLR_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' &&
(IsFeatureImplemented("ARMv8.0-DoubleLock") || boolean IMPLEMENTATION_DEFINED "Trapped by
MDCR_EL3.TDOSA") then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return OSDLR_EL1;
elsif PSTATE.EL == EL3 then
    return OSDLR_EL1;
```

MSR OSDLR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b000	0b0001	0b0011	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
IsFeatureImplemented("ARMv8.0-DoubleLock") && HDFGWTR_EL2.OSDLR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDOSA> != '00' &&
(IsFeatureImplemented("ARMv8.0-DoubleLock") || boolean IMPLEMENTATION_DEFINED "Trapped by
MDCR_EL2.TDOSA") then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' &&
(IsFeatureImplemented("ARMv8.0-DoubleLock") || boolean IMPLEMENTATION_DEFINED "Trapped by
MDCR_EL3.TDOSA") then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        OSDLR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' &&
(IsFeatureImplemented("ARMv8.0-DoubleLock") || boolean IMPLEMENTATION_DEFINED "Trapped by
MDCR_EL3.TDOSA") then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        OSDLR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    OSDLR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.3.22 OSDTRRX_EL1, OS Lock Data Transfer Register, Receive

The OSDTRRX_EL1 characteristics are:

Purpose

Used for save/restore of DBGDTRRX_EL0. It is a component of the Debug Communications Channel.

Configurations

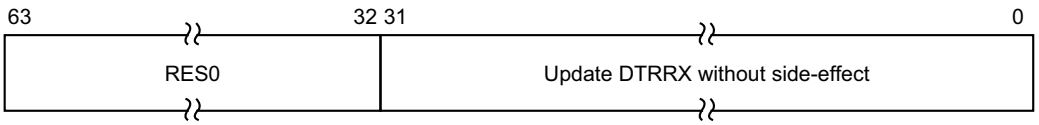
AArch64 System register OSDTRRX_EL1[31:0] is architecturally mapped to AArch32 System register DBGDTRRXext[31:0].

Attributes

OSDTRRX_EL1 is a 64-bit register.

Field descriptions

The OSDTRRX_EL1 bit assignments are:



Bits [63:32]

Reserved, RES0.

Bits [31:0]

Update DTRRX without side-effect.

Writes to this register update the value in DTRRX and do not change RXfull.

Reads of this register return the last value written to DTRRX and do not change RXfull.

For the full behavior of the Debug Communications Channel, see Chapter H4 *The Debug Communication Channel and Instruction Transfer Register*.

Accessing the OSDTRRX_EL1

Arm deprecates reads and writes of OSDTRRX_EL1 when the OS Lock is unlocked.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, OSDTRRX_EL1

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        return OSDTRRX_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return OSDTRRX_EL1;
elsif PSTATE.EL == EL3 then
    return OSDTRRX_EL1;
```

MSR OSDTRRX_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        OSDTRRX_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        OSDTRRX_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    OSDTRRX_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.3.23 OSDTRTX_EL1, OS Lock Data Transfer Register, Transmit

The OSDTRTX_EL1 characteristics are:

Purpose

Used for save/restore of DBGDTRTX_EL0. It is a component of the Debug Communications Channel.

Configurations

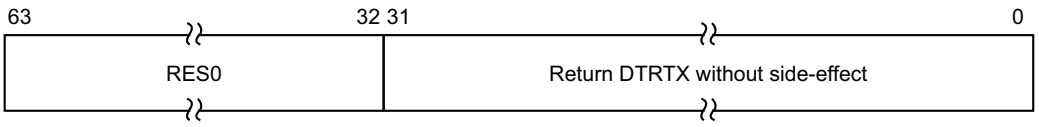
AArch64 System register OSDTRTX_EL1[31:0] is architecturally mapped to AArch32 System register DBGDTRTXext[31:0].

Attributes

OSDTRTX_EL1 is a 64-bit register.

Field descriptions

The OSDTRTX_EL1 bit assignments are:



Bits [63:32]

Reserved, RES0.

Bits [31:0]

Return DTRTX without side-effect.

Reads of this register return the value in DTRTX and do not change TXfull.

Writes of this register update the value in DTRTX and do not change TXfull.

For the full behavior of the Debug Communications Channel, see Chapter H4 *The Debug Communication Channel and Instruction Transfer Register*.

Accessing the OSDTRTX_EL1

Arm deprecates reads and writes of OSDTRTX_EL1 when the OS Lock is unlocked.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, OSDTRTX_EL1

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0011	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        return OSDTRTX_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return OSDTRTX_EL1;
elsif PSTATE.EL == EL3 then
    return OSDTRTX_EL1;
```

MSR OSDTRTX_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0011	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        OSDTRTX_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        OSDTRTX_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    OSDTRTX_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.3.24 OSECCR_EL1, OS Lock Exception Catch Control Register

The OSECCR_EL1 characteristics are:

Purpose

Provides a mechanism for an operating system to access the contents of EDECCR that are otherwise invisible to software, so it can save/restore the contents of EDECCR over powerdown on behalf of the external debugger.

Configurations

AArch64 System register OSECCR_EL1[31:0] is architecturally mapped to AArch32 System register DBGOSECCR[31:0].

AArch64 System register OSECCR_EL1[31:0] is architecturally mapped to External register EDECCR[31:0].

If OSLSR_EL1.OSLK == 0, then OSECCR_EL1 returns an UNKNOWN value on reads and ignores writes.

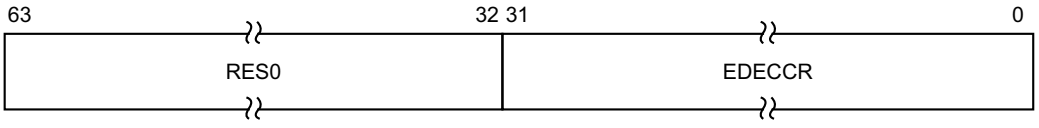
Attributes

OSECCR_EL1 is a 64-bit register.

Field descriptions

The OSECCR_EL1 bit assignments are:

When OSLSR_EL1.OSLK == 1:



Bits [63:32]

Reserved, RES0.

EDECCR, bits [31:0]

Used for save/restore to EDECCR over powerdown.

Reads or writes to this field are indirect accesses to EDECCR.

Accessing the OSECCR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, OSECCR_EL1

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0110	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.OSECCR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return OSECCR_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return OSECCR_EL1;
elsif PSTATE.EL == EL3 then
    return OSECCR_EL1;
```

MSR OSECCR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0110	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.OSECCR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        OSECCR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        OSECCR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    OSECCR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.3.25 OSLAR_EL1, OS Lock Access Register

The OSLAR_EL1 characteristics are:

Purpose

Used to lock or unlock the OS Lock.

Configurations

AArch64 System register OSLAR_EL1[31:0] is architecturally mapped to AArch32 System register DBGOSLAR[31:0].

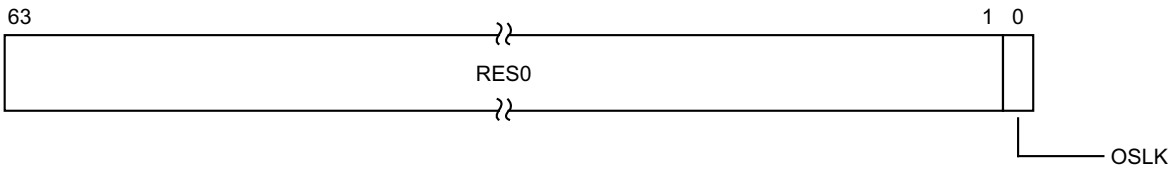
AArch64 System register OSLAR_EL1[31:0] is architecturally mapped to External register OSLAR_EL1[31:0].

Attributes

OSLAR_EL1 is a 64-bit register.

Field descriptions

The OSLAR_EL1 bit assignments are:



Bits [63:1]

Reserved, RES0.

OSLK, bit [0]

On writes to OSLAR_EL1, bit[0] is copied to the OS Lock.

Use OSLSR_EL1.OSLK to check the current status of the lock.

Accessing the OSLAR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MSR OSLAR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b000	0b0001	0b0000	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.OSLAR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDOSA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        OSLAR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            OSLAR_EL1 = X[t];
    elsif PSTATE.EL == EL3 then
        OSLAR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.3.26 OSLSR_EL1, OS Lock Status Register

The OSLSR_EL1 characteristics are:

Purpose

Provides the status of the OS Lock.

Configurations

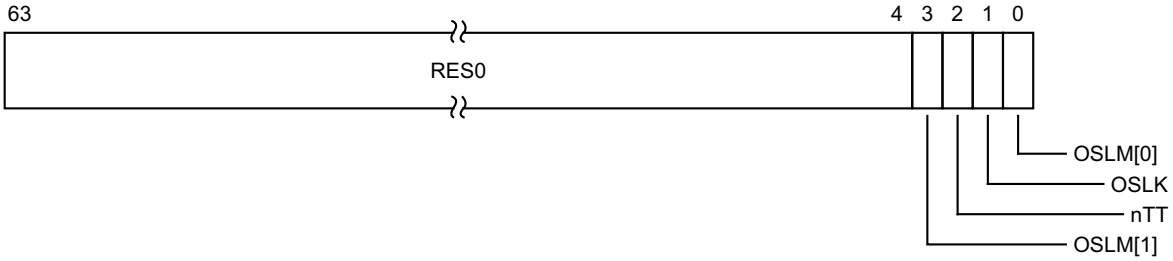
AArch64 System register OSLSR_EL1[31:0] is architecturally mapped to AArch32 System register DBGOSLSR[31:0].

Attributes

OSLSR_EL1 is a 64-bit register.

Field descriptions

The OSLSR_EL1 bit assignments are:



Bits [63:4]

Reserved, RES0.

OSLM[1], bit [3]

This field is bit[1] of OSLM[1:0].

OS lock model implemented. Identifies the form of OS save and restore mechanism implemented.

`0b00` OS Lock not implemented.

`0b10` OS Lock implemented.

All other values are reserved. In an Armv8 implementation the value `0b00` is not permitted.

The OSLM field is split as follows:

- OSLM[1] is OSLSR_EL1[3].
- OSLM[0] is OSLSR_EL1[0].

nTT, bit [2]

Not 32-bit access. This bit is always RAZ. It indicates that a 32-bit access is needed to write the key to the OS Lock Access Register.

OSLK, bit [1]

OS Lock Status.

`0b0` OS Lock unlocked.

`0b1` OS Lock locked.

The OS Lock is locked and unlocked by writing to the OS Lock Access Register.

The following resets apply:

- On a Cold reset, this field resets to `1`.
- On a Warm reset, the value of this field is unchanged.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

OSLM[0], bit [0]

This field is bit[0] of OSLM[1:0].

See OSLM[1] for the field description.

Accessing the OSLSR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, OSLSR_EL1

op0	op1	CRn	CRm	op2
0b10	0b000	0b0001	0b0001	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.OSLSR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDOSA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return OSLSR_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return OSLSR_EL1;
elsif PSTATE.EL == EL3 then
    return OSLSR_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.3.27 SDER32_EL2, AArch32 Secure Debug Enable Register

The SDER32_EL2 characteristics are:

Purpose

Allows access to the AArch32 register SDER from Secure EL2 and EL3 only.

Configurations

This register is present only when ARMv8.4-SecEL2 is implemented and EL1 supports AArch32. Otherwise, direct accesses to SDER32_EL2 are UNDEFINED.

This register is ignored by the PE when when one or more of the following are true:

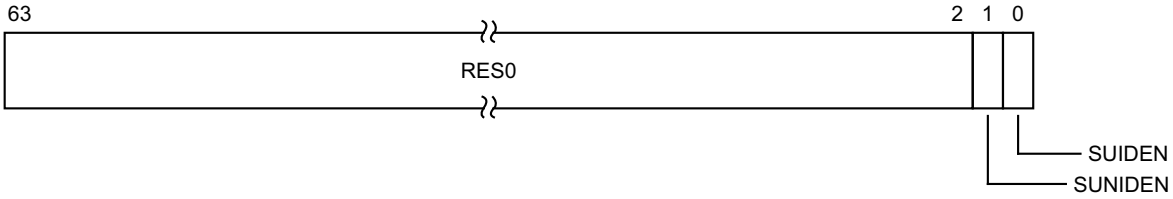
- The PE is in Non-secure state.
- EL1 is using AArch64.

Attributes

SDER32_EL2 is a 64-bit register.

Field descriptions

The SDER32_EL2 bit assignments are:



Bits [63:2]

Reserved, RES0.

SUNIDEN, bit [1]

Secure User Non-Invasive Debug Enable.

`0b0` This bit does not affect Performance Monitors event counting at Secure EL0.

`0b1` If EL1 is using AArch32, Performance Monitors event counting is allowed in Secure EL0.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

SUIDEN, bit [0]

Secure User Invasive Debug Enable.

`0b0` This bit does not affect the generation of debug exceptions at Secure EL0.

`0b1` If EL1 is using AArch32, debug exceptions from Secure EL0 are enabled.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the SDER32_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, SDER32_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0011	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return SDER32_EL2;
elsif PSTATE.EL == EL3 then
    return SDER32_EL2;
```

MSR SDER32_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0011	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        SDER32_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    SDER32_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.3.28 SDER32_EL3, AArch32 Secure Debug Enable Register

The SDER32_EL3 characteristics are:

Purpose

Allows access to the AArch32 register SDER from AArch64 state only. Its value has no effect on execution in AArch64 state.

Configurations

AArch64 System register SDER32_EL3[31:0] is architecturally mapped to AArch32 System register SDER[31:0].

This register is present only when EL3 is implemented and EL1 supports AArch32. Otherwise, direct accesses to SDER32_EL3 are UNDEFINED.

This register is ignored by the PE when when one or more of the following are true:

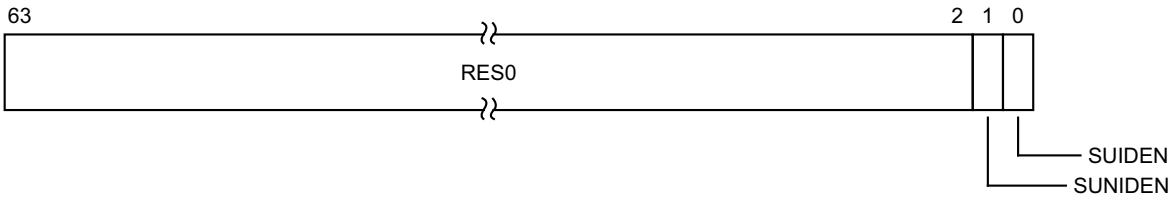
- The PE is in Non-secure state.
- EL1 is using AArch64.

Attributes

SDER32_EL3 is a 64-bit register.

Field descriptions

The SDER32_EL3 bit assignments are:



Bits [63:2]

Reserved, RES0.

SUNIDEN, bit [1]

Secure User Non-Invasive Debug Enable.

`0b0` This bit does not affect Performance Monitors event counting at Secure EL0.

`0b1` If EL1 is using AArch32, Performance Monitors event counting is allowed in Secure EL0.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

SUIDEN, bit [0]

Secure User Invasive Debug Enable.

`0b0` This bit does not affect the generation of debug exceptions at Secure EL0.

`0b1` If EL1 is using AArch32, debug exceptions from Secure EL0 are enabled.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the SDER32_EL3

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, SDER32_EL3

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    return SDER32_EL3;
```

MSR SDER32_EL3, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    SDER32_EL3 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.3.29 TRFCR_EL1, Trace Filter Control Register (EL1)

The TRFCR_EL1 characteristics are:

Purpose

Provides EL1 controls for Trace.

Configurations

AArch64 System register TRFCR_EL1[31:0] is architecturally mapped to AArch32 System register TRFCR[31:0].

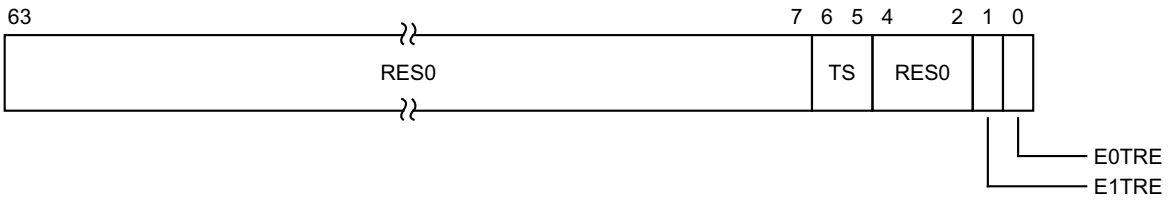
This register is present only when ARMv8.4-Trace is implemented. Otherwise, direct accesses to TRFCR_EL1 are UNDEFINED.

Attributes

TRFCR_EL1 is a 64-bit register.

Field descriptions

The TRFCR_EL1 bit assignments are:



Bits [63:7]

Reserved, RES0.

TS, bits [6:5]

Timestamp Control

`0b01` Virtual timestamp. The traced timestamp is the physical counter value, minus the value of CNTVOFF_EL2.

`0b10` **When ARMv8.6-ECV is implemented** Guest Physical timestamp. The traced timestamp is the physical counter value, minus the value of CNTPOFF_EL2.

`0b11` Physical timestamp. The traced timestamp is the physical counter value.

All other values are reserved

This field is ignored if any of the following are true:

- `SelfHostedTraceEnabled()` == FALSE.
- EL2 is implemented and TRFCR_EL2.TS != `0b00`.

If ARMv8.6-ECV is implemented, and EL2 is implemented and enabled in the current Security state, the physical counter uses a fixed physical offset of zero if any of the following are true:

- CNTHCTL_EL2.ECV is 0.
- SCR_EL3.ECVEn is 0.
- HCR_EL2.{E2H, TGE} is {1, 1}.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [4:2]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

E1TRE, bit [1]

EL1 Trace Enable.

`0b0` Trace is prohibited at EL1.

`0b1` Trace is allowed at EL1.

This field is ignored if `SelfHostedTraceEnabled()` == FALSE.

On a Warm reset, this field resets to `0`.

E0TRE, bit [0]

EL0 Trace Enable.

`0b0` Trace is prohibited at EL0.

`0b1` Trace is allowed at EL0.

This field is ignored if any of the following are true:

- `SelfHostedTraceEnabled()` == FALSE.
- EL2 is implemented and enabled in the current Security state and HCR_EL2.TGE == 1.

On a Warm reset, this field resets to `0`.

Accessing the TRFCR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, TRFCR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TTRF == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TTRF == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x880];
    else
        return TRFCR_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TTRF == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        return TRFCR_EL2;
    else
        return TRFCR_EL1;
elsif PSTATE.EL == EL3 then
    return TRFCR_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR TRFCR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.TRFCR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TTRF == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TTRF == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x880] = X[t];
    else
        TRFCR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TTRF == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        TRFCR_EL2 = X[t];
    else
        TRFCR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    TRFCR_EL1 = X[t];
```

MRS <Xt>, TRFCR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        return NVMem[0x880];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TTRF == '1' then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            return TRFCR_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        return TRFCR_EL1;
    else
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR TRFCR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        NVMem[0x880] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TTRF == '1' then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            TRFCR_EL1 = X[t];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        TRFCR_EL1 = X[t];
    else
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.3.30 TRFCR_EL2, Trace Filter Control Register (EL2)

The TRFCR_EL2 characteristics are:

Purpose

Provides EL2 controls for Trace.

Configurations

AArch64 System register TRFCR_EL2[31:0] is architecturally mapped to AArch32 System register HTRFCR[31:0].

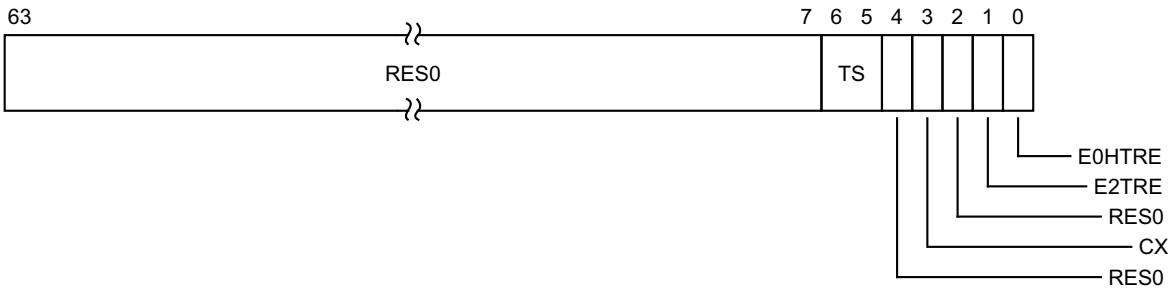
This register is present only when ARMv8.4-Trace is implemented. Otherwise, direct accesses to TRFCR_EL2 are UNDEFINED.

Attributes

TRFCR_EL2 is a 64-bit register.

Field descriptions

The TRFCR_EL2 bit assignments are:



Bits [63:7]

Reserved, RES0.

TS, bits [6:5]

Timestamp Control. Controls which timebase is used for trace timestamps.

`0b00` Timestamp controlled by TRFCR_EL1.TS or TRFCR.TS.

`0b01` Virtual timestamp. The traced timestamp is the physical counter value, minus the value of CNTVOFF_EL2.

`0b10` **When ARMv8.6-ECV is implemented** Guest Physical timestamp. The traced timestamp is the physical counter value, minus the value of CNTPOFF_EL2.

`0b11` Physical timestamp. The traced timestamp is the physical counter value.

All other values are reserved.

This field is ignored if `SelfHostedTraceEnabled()` == FALSE.

If ARMv8.6-ECV is implemented, and EL2 is implemented and enabled in the current Security state, the physical counter uses a fixed physical offset of zero if any of the following are true:

- CNTHCTL_EL2.ECV is 0.
- SCR_EL3.ECVEn is 0.
- HCR_EL2.{E2H, TGE} is {1, 1}.

On a Warm reset, this field resets to `0`.

Bit [4]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CX, bit [3]

CONTEXTIDR_EL2 and VMID trace enable.

`0b0` CONTEXTIDR_EL2 and VMID trace prohibited.

`0b1` CONTEXTIDR_EL2 and VMID trace allowed.

This field is ignored if `SelfHostedTraceEnabled()` == FALSE.

On a Warm reset, this field resets to `0`.

Bit [2]

Reserved, RES0.

E2TRE, bit [1]

EL2 Trace Enable.

`0b0` Trace is prohibited at EL2.

`0b1` Trace is allowed at EL2.

This field is ignored if `SelfHostedTraceEnabled()` == FALSE.

On a Warm reset, this field resets to `0`.

E0HTRE, bit [0]

EL0 Trace Enable.

`0b0` Trace is prohibited at EL0 when HCR_EL2.TGE == 1.

`0b1` Trace is allowed at EL0 when HCR_EL2.TGE == 1.

This field is ignored if any of the following are true:

- `SelfHostedTraceEnabled()` == FALSE.
- EL2 is disabled in the current security state.
- HCR_EL2.TGE == 0.

On a Warm reset, this field resets to `0`.

Accessing the TRFCR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, TRFCR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TTRF == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return TRFCR_EL2;
elsif PSTATE.EL == EL3 then
    return TRFCR_EL2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR TRFCR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TTRF == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        TRFCR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    TRFCR_EL2 = X[t];
```

MRS <Xt>, TRFCR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TTRF == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TTRF == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x880];
    else
        return TRFCR_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TTRF == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        return TRFCR_EL2;
    else
        return TRFCR_EL1;
elsif PSTATE.EL == EL3 then
    return TRFCR_EL1;
```

MSR TRFCR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.TRFCR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TTRF == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TTRF == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x880] = X[t];
    else
        TRFCR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TTRF == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        TRFCR_EL2 = X[t];
    else
        TRFCR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    TRFCR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.4 Performance Monitors registers

This section lists the Performance Monitoring registers in AArch64.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.4.1 PMCCFILTR_EL0, Performance Monitors Cycle Count Filter Register

The PMCCFILTR_EL0 characteristics are:

Purpose

Determines the modes in which the Cycle Counter, PMCCNTR_EL0, increments.

Configurations

AArch64 System register PMCCFILTR_EL0[31:0] is architecturally mapped to AArch32 System register PMCCFILTR[31:0].

AArch64 System register PMCCFILTR_EL0[31:0] is architecturally mapped to External register PMCCFILTR_EL0[31:0].

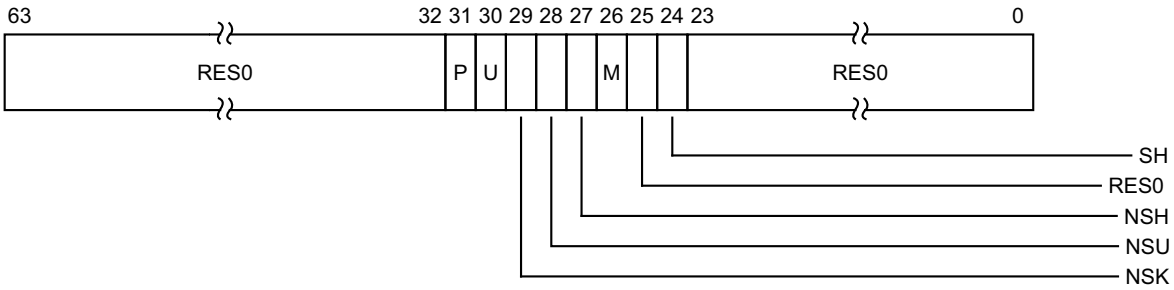
This register is present only when PMUv3 is implemented. Otherwise, direct accesses to PMCCFILTR_EL0 are UNDEFINED.

Attributes

PMCCFILTR_EL0 is a 64-bit register.

Field descriptions

The PMCCFILTR_EL0 bit assignments are:



Bits [63:32]

Reserved, RES0.

P, bit [31]

Privileged filtering bit. Controls counting in EL1.

If EL3 is implemented, then counting in Non-secure EL1 is further controlled by the PMCCFILTR_EL0.NSK bit.

`0b0` Count cycles in EL1.

`0b1` Do not count cycles in EL1.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

U, bit [30]

User filtering bit. Controls counting in EL0.

If EL3 is implemented, then counting in Non-secure EL0 is further controlled by the PMCCFILTR_EL0.NSU bit.

`0b0` Count cycles in EL0.

`0b1` Do not count cycles in EL0.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

NSK, bit [29]

When EL3 is implemented:

Non-secure EL1 (kernel) modes filtering bit. Controls counting in Non-secure EL1.

If the value of this bit is equal to the value of the PMCCFILTR_EL0.P bit, cycles in Non-secure EL1 are counted.

Otherwise, cycles in Non-secure EL1 are not counted.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSU, bit [28]

When EL3 is implemented:

Non-secure EL0 (Unprivileged) filtering bit. Controls counting in Non-secure EL0.

If the value of this bit is equal to the value of the PMCCFILTR_EL0.U bit, cycles in Non-secure EL0 are counted.

Otherwise, cycles in Non-secure EL0 are not counted.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSH, bit [27]

When EL2 is implemented:

EL2 (Hypervisor) filtering bit. Controls counting in EL2.

If Secure EL2 is implemented, counting in Secure EL2 is further controlled by the PMCCFILTR_EL0.SH bit.

`0b0` Do not count cycles in EL2.

`0b1` Count cycles in EL2.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

M, bit [26]

When EL3 is implemented:

Secure EL3 filtering bit.

If the value of this bit is equal to the value of the PMCCFILTR_EL0.P bit, cycles in Secure EL3 are counted.

Otherwise, cycles in Secure EL3 are not counted.

Most applications can ignore this field and set its value to 0.

Note

This field is not visible in the AArch32 PMCCFILTR System register.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [25]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

SH, bit [24]

When ARMv8.4-SecEL2 is implemented:

Secure EL2 filtering.

If the value of this bit is not equal to the value of the PMCCFILTR_EL0.NSH bit, cycles in Secure EL2 are counted.

Otherwise, cycles in Secure EL2 are not counted.

If Secure EL2 is not implemented or is disabled, this field is RES0.

Note

This field is not visible in the AArch32 PMCCFILTR System register.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [23:0]

Reserved, RES0.

Accessing the PMCCFILTR_EL0

PMCCFILTR_EL0 can also be accessed by using PMXEVTYPER_EL0 with PMSELR_EL0.SEL set to `0b11111`.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PMCCFILTR_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b1111	0b111

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMCCFILTR_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMCCFILTR_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.PMCCFILTR_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMCCFILTR_EL0;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        return PMCCFILTR_EL0;
elsif PSTATE.EL == EL3 then
    return PMCCFILTR_EL0;
```

MSR PMCCFILTR_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b1111	0b111

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMCCFILTR_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMCCFILTR_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.PMCCFILTR_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMCCFILTR_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMCCFILTR_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    PMCCFILTR_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.4.2 PMCCNTR_EL0, Performance Monitors Cycle Count Register

The PMCCNTR_EL0 characteristics are:

Purpose

Holds the value of the processor Cycle Counter, CCNT, that counts processor clock cycles. See *Time as measured by the Performance Monitors cycle counter* on page D7-2702 for more information.

PMCCFILTR_EL0 determines the modes and states in which the PMCCNTR_EL0 can increment.

Configurations

AArch64 System register PMCCNTR_EL0[63:0] is architecturally mapped to AArch32 System register PMCCNTR[63:0].

AArch64 System register PMCCNTR_EL0[63:0] is architecturally mapped to External register PMCCNTR_EL0[63:0].

This register is present only when PMUv3 is implemented. Otherwise, direct accesses to PMCCNTR_EL0 are UNDEFINED.

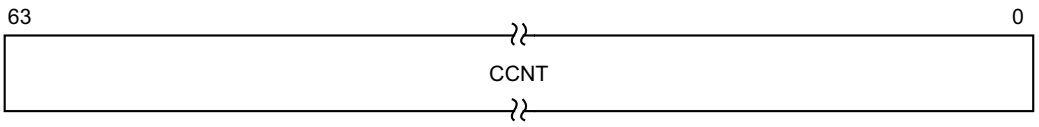
All counters are subject to any changes in clock frequency, including clock stopping caused by the WFI and WFE instructions. This means that it is CONSTRAINED UNPREDICTABLE whether or not PMCCNTR_EL0 continues to increment when clocks are stopped by WFI and WFE instructions.

Attributes

PMCCNTR_EL0 is a 64-bit register.

Field descriptions

The PMCCNTR_EL0 bit assignments are:



CCNT, bits [63:0]

Cycle count. Depending on the values of PMCR_EL0.{LC,D}, this field increments in one of the following ways:

- Every processor clock cycle.
- Every 64th processor clock cycle.

Writing 1 to PMCR_EL0.C sets this field to 0.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMCCNTR_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PMCCNTR_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1101	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.<CR,EN> == '00' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMCCNTR_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMCCNTR_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.PMCCNTR_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMCCNTR_EL0;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMCCNTR_EL0;
elsif PSTATE.EL == EL3 then
    return PMCCNTR_EL0;
```

MSR PMCCNTR_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1101	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMCCNTR_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMCCNTR_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.PMCCNTR_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMCCNTR_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMCCNTR_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    PMCCNTR_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.4.3 PMCEID0_EL0, Performance Monitors Common Event Identification register 0

The PMCEID0_EL0 characteristics are:

Purpose

Defines which common architectural events and common microarchitectural events are implemented, or counted, using PMU events in the ranges `0x0000` to `0x001F` and `0x4000` to `0x401F`.

When the value of a bit in the register is 1 the corresponding common event is implemented and counted.

Note

Arm recommends that, if a common event is never counted, the value of the corresponding register bit is 0.

For more information about the common events and the use of the PMCEID<n>_EL0 registers see *The PMU event number space and common events* on page D7-2721.

Configurations

AArch64 System register PMCEID0_EL0[31:0] is architecturally mapped to AArch32 System register PMCEID0[31:0].

AArch64 System register PMCEID0_EL0[63:32] is architecturally mapped to AArch32 System register PMCEID2[31:0].

AArch64 System register PMCEID0_EL0[31:0] is architecturally mapped to External register PMCEID0[31:0].

AArch64 System register PMCEID0_EL0[63:32] is architecturally mapped to External register PMCEID2[31:0].

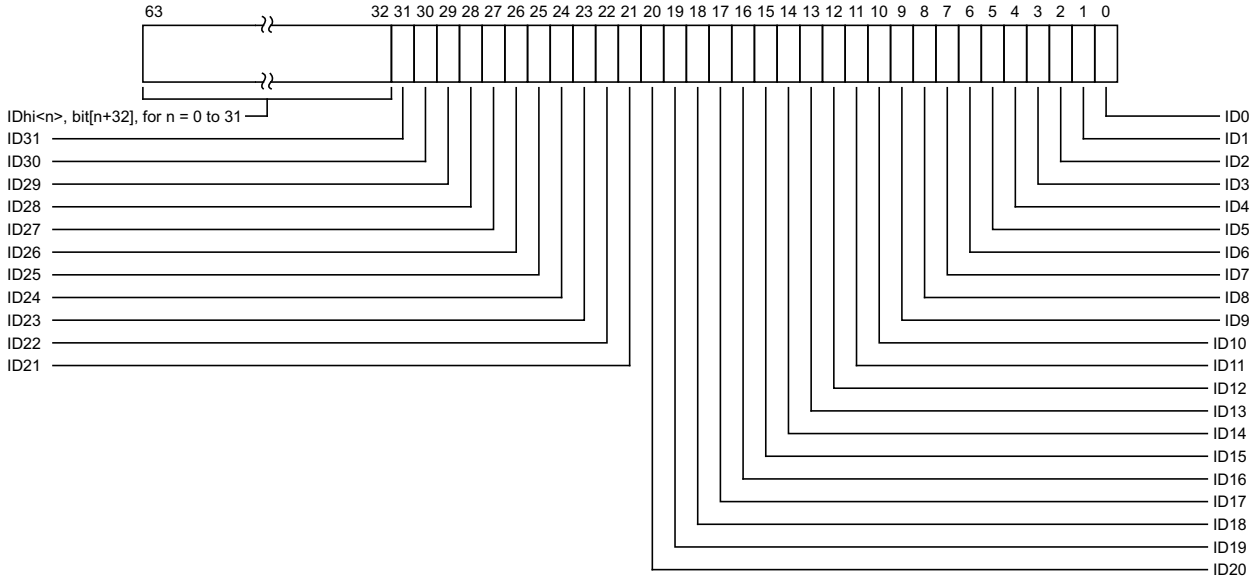
This register is present only when PMUv3 is implemented. Otherwise, direct accesses to PMCEID0_EL0 are UNDEFINED.

Attributes

PMCEID0_EL0 is a 64-bit register.

Field descriptions

The PMCEID0_EL0 bit assignments are:



Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

IDhi<n>, bit [n+32], for n = 0 to 31

When ARMv8.1-PMU is implemented:

IDhi[n] corresponds to common event (`0x4000` + n).

For each bit:

`0b0` The common event is not implemented, or not counted.

`0b1` The common event is implemented.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n>_EL0 registers of that earlier version of the PMU architecture.

Otherwise:

Reserved, RES0.

ID<n>, bit [n], for n = 0 to 31

ID[n] corresponds to common event n.

For each bit:

`0b0` The common event is not implemented, or not counted.

`0b1` The common event is implemented.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n>_EL0 registers of that earlier version of the PMU architecture.

Accessing the PMCEID0_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PMCEID0_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b110

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMCEID0_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMCEID0_EL0;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMCEID0_EL0;
elsif PSTATE.EL == EL3 then
    return PMCEID0_EL0;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.4.4 PMCEID1_EL0, Performance Monitors Common Event Identification register 1

The PMCEID1_EL0 characteristics are:

Purpose

Defines which common architectural events and common microarchitectural events are implemented, or counted, using PMU events in the ranges `0x0020` to `0x003F` and `0x4020` to `0x403F`.

When the value of a bit in the register is 1 the corresponding common event is implemented and counted.

Note

Arm recommends that, if a common event is never counted, the value of the corresponding register bit is 0.

For more information about the common events and the use of the PMCEID<n>_EL0 registers see *The PMU event number space and common events* on page D7-2721.

Configurations

AArch64 System register PMCEID1_EL0[31:0] is architecturally mapped to AArch32 System register PMCEID1[31:0].

AArch64 System register PMCEID1_EL0[63:32] is architecturally mapped to AArch32 System register PMCEID3[31:0].

AArch64 System register PMCEID1_EL0[31:0] is architecturally mapped to External register PMCEID1[31:0].

AArch64 System register PMCEID1_EL0[63:32] is architecturally mapped to External register PMCEID3[31:0].

This register is present only when PMUv3 is implemented. Otherwise, direct accesses to PMCEID1_EL0 are UNDEFINED.

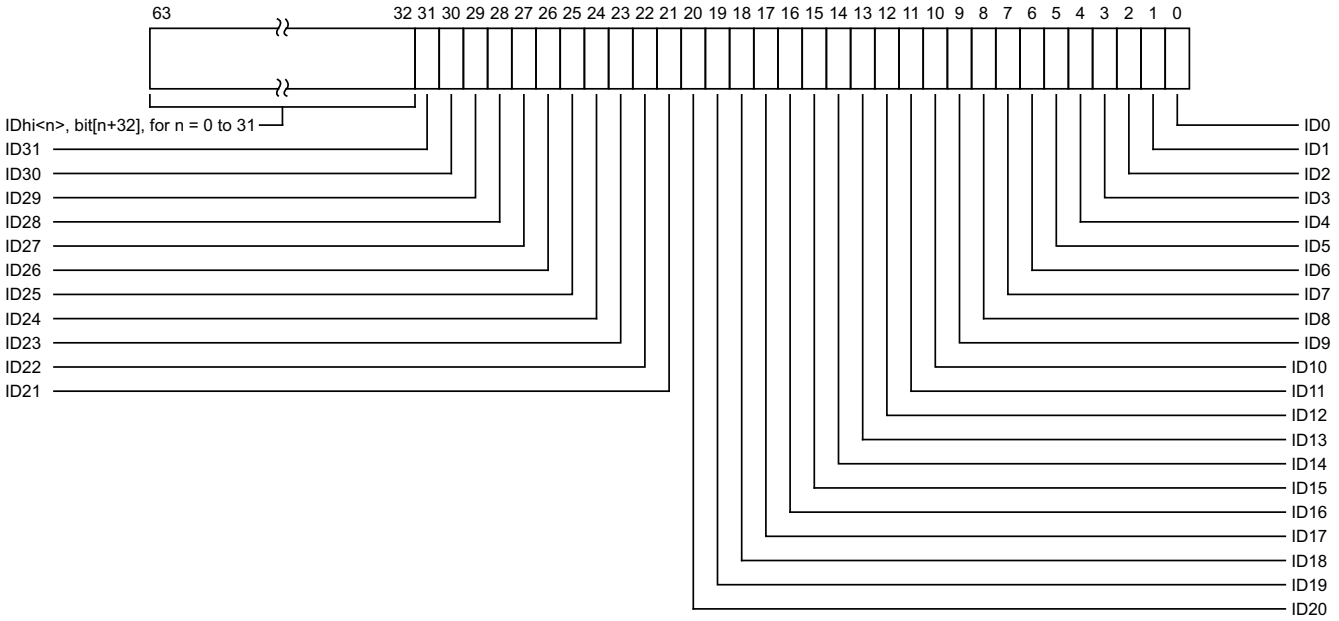
Attributes

PMCEID1_EL0 is a 64-bit register.

Field descriptions

The PMCEID1_EL0 bit assignments are:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta



IDhi<n>, bit [n+32], for n = 0 to 31

When ARMv8.1-PMU is implemented:

IDhi[n] corresponds to common event (`0x4020` + n).

For each bit:

`0b0` The common event is not implemented, or not counted.

`0b1` The common event is implemented.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n>_EL0 registers of that earlier version of the PMU architecture.

Otherwise:

Reserved, RES0.

ID<n>, bit [n], for n = 0 to 31

ID[n] corresponds to common event (`0x0020` + n).

For each bit:

`0b0` The common event is not implemented, or not counted.

`0b1` The common event is implemented.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n>_EL0 registers of that earlier version of the PMU architecture.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the PMCEID1_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PMCEID1_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b111

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMCEID1_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMCEID1_EL0;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMCEID1_EL0;
elsif PSTATE.EL == EL3 then
    return PMCEID1_EL0;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.4.5 PMCNTENCLR_EL0, Performance Monitors Count Enable Clear register

The PMCNTENCLR_EL0 characteristics are:

Purpose

Disables the Cycle Count Register, PMCCNTR_EL0, and any implemented event counters PMEVCNTR<n>. Reading this register shows which counters are enabled.

Configurations

AArch64 System register PMCNTENCLR_EL0[31:0] is architecturally mapped to AArch32 System register PMCNTENCLR[31:0].

AArch64 System register PMCNTENCLR_EL0[31:0] is architecturally mapped to External register PMCNTENCLR_EL0[31:0].

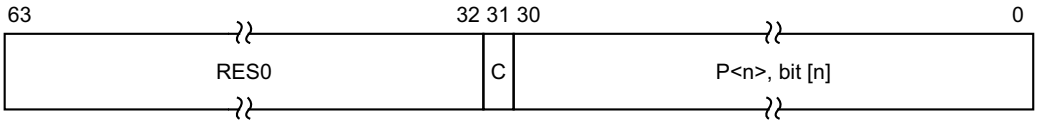
This register is present only when PMUv3 is implemented. Otherwise, direct accesses to PMCNTENCLR_EL0 are UNDEFINED.

Attributes

PMCNTENCLR_EL0 is a 64-bit register.

Field descriptions

The PMCNTENCLR_EL0 bit assignments are:



Bits [63:32]

Reserved, RES0.

C, bit [31]

PMCCNTR_EL0 disable bit. Disables the cycle counter register. Possible values are:

`0b0` When read, means the cycle counter is disabled. When written, has no effect.

`0b1` When read, means the cycle counter is enabled. When written, disables the cycle counter.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

P<n>, bit [n], for n = 0 to 30

Event counter disable bit for PMEVCNTR<n>_EL0.

If N is less than 31, then bits [30:N] are RAZ/WI. When EL2 is implemented and enabled in the current Security state, in EL1 and EL0, N is the value in MDCR_EL2.HPMN. Otherwise, N is the value in PMCR_EL0.N.

`0b0` When read, means that PMEVCNTR<n>_EL0 is disabled. When written, has no effect.

`0b1` When read, means that PMEVCNTR<n>_EL0 is enabled. When written, disables PMEVCNTR<n>_EL0.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMCNTENCLR_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, PMCNTENCLR_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b010

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMCNTEN == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMCNTENCLR_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.PMCNTEN == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMCNTENCLR_EL0;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMCNTENCLR_EL0;
elsif PSTATE.EL == EL3 then
    return PMCNTENCLR_EL0;
```

MSR PMCNTENCLR_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b010

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMCNTEN == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMCNTENCLR_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.PMCNTEN == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMCNTENCLR_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMCNTENCLR_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    PMCNTENCLR_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.4.6 PMCNTENSET_EL0, Performance Monitors Count Enable Set register

The PMCNTENSET_EL0 characteristics are:

Purpose

Enables the Cycle Count Register, PMCCNTR_EL0, and any implemented event counters PMEVCNTR<n>. Reading this register shows which counters are enabled.

Configurations

AArch64 System register PMCNTENSET_EL0[31:0] is architecturally mapped to AArch32 System register PMCNTENSET[31:0].

AArch64 System register PMCNTENSET_EL0[31:0] is architecturally mapped to External register PMCNTENSET_EL0[31:0].

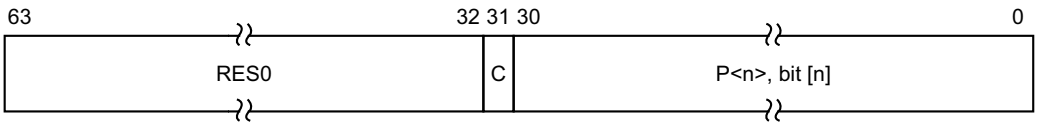
This register is present only when PMUv3 is implemented. Otherwise, direct accesses to PMCNTENSET_EL0 are UNDEFINED.

Attributes

PMCNTENSET_EL0 is a 64-bit register.

Field descriptions

The PMCNTENSET_EL0 bit assignments are:



Bits [63:32]

Reserved, RES0.

C, bit [31]

PMCCNTR_EL0 enable bit. Enables the cycle counter register. Possible values are:

`0b0` When read, means the cycle counter is disabled. When written, has no effect.

`0b1` When read, means the cycle counter is enabled. When written, enables the cycle counter.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

P<n>, bit [n], for n = 0 to 30

Event counter enable bit for PMEVCNTR<n>_EL0.

If N is less than 31, then bits [30:N] are RAZ/WI. When EL2 is implemented and enabled in the current Security state, in EL1 and EL0, N is the value in MDCR_EL2.HPMN. Otherwise, N is the value in PMCR_EL0.N.

`0b0` When read, means that PMEVCNTR<n>_EL0 is disabled. When written, has no effect.

`0b1` When read, means that PMEVCNTR<n>_EL0 event counter is enabled. When written, enables PMEVCNTR<n>_EL0.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMCNTENSET_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, PMCNTENSET_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMCNTEN == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMCNTENSET_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.PMCNTEN == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMCNTENSET_EL0;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMCNTENSET_EL0;
elsif PSTATE.EL == EL3 then
    return PMCNTENSET_EL0;
```

MSR PMCNTENSET_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMCNTEN == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMCNTENSET_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.PMCNTEN == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMCNTENSET_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMCNTENSET_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    PMCNTENSET_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.4.7 PMCR_EL0, Performance Monitors Control Register

The PMCR_EL0 characteristics are:

Purpose

Provides details of the Performance Monitors implementation, including the number of counters implemented, and configures and controls the counters.

Configurations

AArch64 System register PMCR_EL0[31:0] is architecturally mapped to AArch32 System register PMCR[31:0].

AArch64 System register PMCR_EL0[7:0] is architecturally mapped to External register PMCR_EL0[7:0].

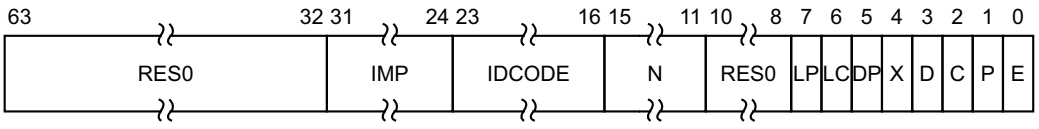
This register is present only when PMUv3 is implemented. Otherwise, direct accesses to PMCR_EL0 are UNDEFINED.

Attributes

PMCR_EL0 is a 64-bit register.

Field descriptions

The PMCR_EL0 bit assignments are:



Bits [63:32]

Reserved, RES0.

IMP, bits [31:24]

Implementer code. This field has an IMPLEMENTATION DEFINED value.

If this field is zero, then PMCR_EL0.IDCODE is RES0 and software must use the MIDR_EL1 to identify the PE.

Otherwise, this field and PMCR_EL0.IDCODE identifies the PMU implementation to software. The implementer codes are allocated by Arm. A non-zero value has the same interpretation as MIDR_EL1.Implementer.

Access to this field is RO.

IDCODE, bits [23:16]

When PMCR_EL0.IMP != 0x00:

Identification code. This field has an IMPLEMENTATION DEFINED value.

Each implementer must maintain a list of identification codes that are specific to the implementer. A specific implementation is identified by the combination of the implementer code and the identification code.

Access to this field is RO.

Otherwise:

Reserved, RES0.

N, bits [15:11]

Indicates the number of event counters implemented. This value is in the range of `0b00000`-`0b11111`. If the value is `0b00000` then only PMCCNTR_EL0 is implemented. If the value is `0b11111` PMCCNTR_EL0 and 31 event counters are implemented.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When EL2 is implemented and enabled for the current Security state, reads of this field from EL1 and EL0 return the value of MDCR_EL2.HPMN.

Access to this field is RO.

Bits [10:8]

Reserved, RES0.

LP, bit [7]

When ARMv8.5-PMU is implemented:

Long event counter enable. Determines when unsigned overflow is recorded by a counter overflow bit.

`0b0` Event counter overflow on increment that causes unsigned overflow of PMEVCNTR<n>_EL0[31:0].

`0b1` Event counter overflow on increment that causes unsigned overflow of PMEVCNTR<n>_EL0[63:0].

If EL2 is implemented and MDCR_EL2.HPMN or HDCR.HPMN is less than PMCR_EL0.N, this bit does not affect the operation of event counters in the range [HDCR.HPMN..(PMCR_EL0.N-1)] or [MDCR_EL2.HPMN..(PMCR_EL0.N-1)].

Note

The effect of MDCR_EL2.HPMN or HDCR.HPMN on the operation of this bit always applies if EL2 is implemented, at all Exception levels including EL2 and EL3, and regardless of whether EL2 is enabled in the current Security state. For more information, see the description of MDCR_EL2.HPMN or HDCR.HPMN.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

LC, bit [6]

When AArch32 is supported at any Exception level:

Long cycle counter enable. Determines when unsigned overflow is recorded by the cycle counter overflow bit.

`0b0` Cycle counter overflow on increment that causes unsigned overflow of PMCCNTR_EL0[31:0].

`0b1` Cycle counter overflow on increment that causes unsigned overflow of PMCCNTR_EL0[63:0].

Arm deprecates use of PMCR_EL0.LC = 0.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

DP, bit [5]

When EL3 is implemented or (ARMv8.1-PMU is implemented and EL2 is implemented):

Disable cycle counter when event counting is prohibited.

`0b0` Cycle counting by PMCCNTR_EL0 is not affected by this bit.

`0b1` When event counting for counters in the range [0..(MDCR_EL2.HPMN-1)] is prohibited, cycle counting by PMCCNTR_EL0 is disabled.

For more information about the interaction between the Performance Monitors and EL3, see *Prohibiting event counting* on page D7-2709.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

Reserved, RES0.

X, bit [4]

When the implementation includes a PMU event export bus:

Enable export of events in an IMPLEMENTATION DEFINED PMU event export bus.

`0b0` Do not export events.

`0b1` Export events where not prohibited.

This field enables the exporting of events over an IMPLEMENTATION DEFINED PMU event export bus to another device, for example to an OPTIONAL PE trace unit.

No events are exported when counting is prohibited.

This field does not affect the generation of Performance Monitors overflow interrupt requests or signaling to a cross-trigger interface (CTI) that can be implemented as signals exported from the PE.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

D, bit [3]

When AArch32 is supported at any Exception level:

Clock divider.

`0b0` When enabled, PMCCNTR_EL0 counts every clock cycle.

`0b1` When enabled, PMCCNTR_EL0 counts once every 64 clock cycles.

If PMCR_EL0.LC == 1, this bit is ignored and the cycle counter counts every clock cycle.

Arm deprecates use of PMCR_EL0.D = 1.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

C, bit [2]

Cycle counter reset. The effects of writing to this bit are:

`0b0` No action.

`0b1` Reset PMCCNTR_EL0 to zero.

This bit is always RAZ.

Note

Resetting PMCCNTR_EL0 does not change the cycle counter overflow bit.

The value of PMCR_EL0.LC is ignored, and bits [63:0] of all affected event counters are reset.

Access to this field is WO.

P, bit [1]

Event counter reset. The effects of writing to this bit are:

`0b0` No action.

`0b1` Reset all event counters accessible in the current Exception level, not including PMCCNTR_EL0, to zero.

This bit is always RAZ.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In EL0 and EL1:

- If EL2 is implemented and enabled in the current Security state, and MDCR_EL2.HPMN is less than PMCR_EL0.N, a write of 1 to this bit does not reset event counters in the range [MDCR_EL2.HPMN..(PMCR_EL0.N-1)].
- If EL2 is not implemented, EL2 is disabled in the current Security state, or MDCR_EL2.HPMN equals PMCR_EL0.N, a write of 1 to this bit resets all the event counters.

In EL2 and EL3, a write of 1 to this bit resets all the event counters.

Note

Resetting the event counters does not change the event counter overflow bits.

If ARMv8.5-PMU is implemented, the values of MDCR_EL2.HLP and PMCR_EL0.LP are ignored, and bits [63:0] of all affected event counters are reset.

Access to this field is WO.

E, bit [0]

Enable.

`0b0` All event counters in the range [0..(PMN-1)] and PMCCNTR_EL0, are disabled.

`0b1` All event counters in the range [0..(PMN-1)] and PMCCNTR_EL0, are enabled by PMCNTENSET_EL0.

If EL2 is implemented, then:

- If EL2 is using AArch32, PMN is HDCR.HPMN.
- If EL2 is using AArch64, PMN is MDCR_EL2.HPMN.
- If PMN is less than PMCR_EL0.N, this bit does not affect the operation of event counters in the range [PMN..(PMCR_EL0.N-1)].

If EL2 is not implemented, PMN is PMCR_EL0.N.

Note

The effect of MDCR_EL2.HPMN or HDCR.HPMN on the operation of this bit always applies if EL2 is implemented, at all Exception levels including EL2 and EL3, and regardless of whether EL2 is enabled in the current Security state. For more information, see the description of MDCR_EL2.HPMN or HDCR.HPMN.

On a Warm reset, this field resets to `0`.

Accessing the PMCR_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PMCR_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPMCR == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMCR_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPMCR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMCR_EL0;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMCR_EL0;
elsif PSTATE.EL == EL3 then
    return PMCR_EL0;
```

MSR PMCR_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMCR_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPMCR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMCR_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.PMCR_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPMCR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMCR_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMCR_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    PMCR_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.4.8 PMEVCNTR<n>_EL0, Performance Monitors Event Count Registers, n = 0 - 30

The PMEVCNTR<n>_EL0 characteristics are:

Purpose

Holds event counter n, which counts events, where n is 0 to 30.

Configurations

AArch64 System register PMEVCNTR<n>_EL0[31:0] is architecturally mapped to AArch32 System register PMEVCNTR<n>[31:0].

AArch64 System register PMEVCNTR<n>_EL0[31:0] is architecturally mapped to External register PMEVCNTR<n>_EL0[31:0].

This register is present only when PMUv3 is implemented. Otherwise, direct accesses to PMEVCNTR<n>_EL0 are UNDEFINED.

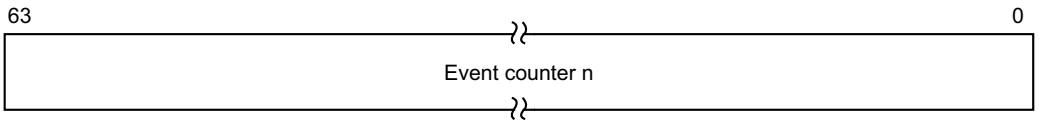
Attributes

PMEVCNTR<n>_EL0 is a 64-bit register.

Field descriptions

The PMEVCNTR<n>_EL0 bit assignments are:

When ARMv8.5-PMU is implemented:

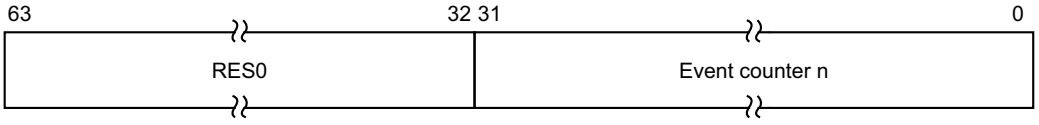


Bits [63:0]

Event counter n. Value of event counter n, where n is the number of this register and is a number from 0 to 30.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:



Bits [63:32]

Reserved, RES0.

Bits [31:0]

Event counter n. Value of event counter n, where n is the number of this register and is a number from 0 to 30.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMEVCNTR<n>_EL0

PMEVCNTR<n>_EL0 can also be accessed by using PMXEVCNTR_EL0 with PMSELR_EL0.SEL set to the value of <n>.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If ARMv8.6-FGT is implemented and <n> is greater than or equal to the number of accessible counters, then the behavior of permitted reads and writes of PMEVCNTR<n>_EL0 is as follows:

- If <n> is an unimplemented event counter, the access is UNDEFINED.
- Otherwise, the access is trapped to EL2.

If ARMv8.6-FGT is not implemented and <n> is greater than or equal to the number of accessible counters, then reads and writes of PMEVCNTR<n>_EL0 are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.
- If EL2 is implemented and enabled in the current Security state, and <n> is less than the number of implemented counters, accesses from EL1 or permitted accesses from EL0 are trapped to EL2.

Note

In EL0, an access is permitted if it is enabled by PMUSERENR_EL0.{ER,EN}.

If EL2 is implemented and enabled in the current Security state, in EL1 and EL0, MDCR_EL2.HPMN identifies the number of accessible counters. Otherwise, the number of accessible counters is the number of implemented counters. See MDCR_EL2.HPMN for more details.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PMEVCNTR<n>_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b10:n[4:3]	n[2:0]

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.<ER,EN> == '00' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMEVCNTRn_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMEVCNTR_EL0[UInt(CRm<1:0>:op2<2:0>)];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.PMEVCNTRn_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMEVCNTR_EL0[UInt(CRm<1:0>:op2<2:0>)];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        return PMEVCNTR_EL0[UInt(CRm<1:0>:op2<2:0>)];
elsif PSTATE.EL == EL3 then
    return PMEVCNTR_EL0[UInt(CRm<1:0>:op2<2:0>)];
```

MSR PMEVCNTR<n>_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b10:n[4:3]	n[2:0]

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMEVCNTRn_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMEVCNTR_EL0[UInt(CRm<1:0>:op2<2:0>)] = X[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.PMEVCNTRn_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMEVCNTR_EL0[UInt(CRm<1:0>:op2<2:0>)] = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMEVCNTR_EL0[UInt(CRm<1:0>:op2<2:0>)] = X[t];
elsif PSTATE.EL == EL3 then
    PMEVCNTR_EL0[UInt(CRm<1:0>:op2<2:0>)] = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.4.9 PMEVTYPER<n>_EL0, Performance Monitors Event Type Registers, n = 0 - 30

The PMEVTYPER<n>_EL0 characteristics are:

Purpose

Configures event counter n, where n is 0 to 30.

Configurations

AArch64 System register PMEVTYPER<n>_EL0[31:0] is architecturally mapped to AArch32 System register PMEVTYPER<n>[31:0].

AArch64 System register PMEVTYPER<n>_EL0[31:0] is architecturally mapped to External register PMEVTYPER<n>_EL0[31:0].

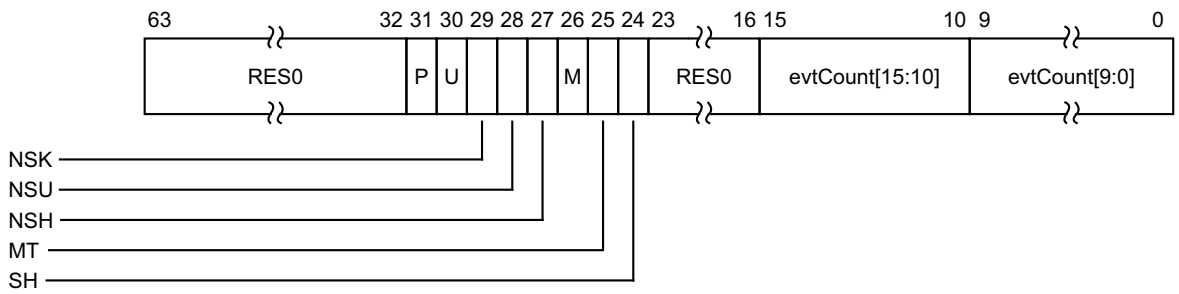
This register is present only when PMUv3 is implemented. Otherwise, direct accesses to PMEVTYPER<n>_EL0 are UNDEFINED.

Attributes

PMEVTYPER<n>_EL0 is a 64-bit register.

Field descriptions

The PMEVTYPER<n>_EL0 bit assignments are:



Bits [63:32]

Reserved, RES0.

P, bit [31]

Privileged filtering bit. Controls counting in EL1.

If EL3 is implemented, then counting in Non-secure EL1 is further controlled by the PMEVTYPER<n>_EL0.NSK bit.

`0b0` Count events in EL1.

`0b1` Do not count events in EL1.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

U, bit [30]

User filtering bit. Controls counting in EL0.

If EL3 is implemented, then counting in Non-secure EL0 is further controlled by the PMEVTYPER<n>_EL0.NSU bit.

`0b0` Count events in EL0.

`0b1` Do not count events in EL0.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

NSK, bit [29]

When EL3 is implemented:

Non-secure EL1 (kernel) modes filtering bit. Controls counting in Non-secure EL1.

If the value of this bit is equal to the value of the PMEVTYPER<n>_EL0.P bit, events in Non-secure EL1 are counted.

Otherwise, events in Non-secure EL1 are not counted.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSU, bit [28]

When EL3 is implemented:

Non-secure EL0 (Unprivileged) filtering bit. Controls counting in Non-secure EL0.

If the value of this bit is equal to the value of the PMEVTYPER<n>_EL0.U bit, events in Non-secure EL0 are counted.

Otherwise, events in Non-secure EL0 are not counted.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSH, bit [27]

When EL2 is implemented:

EL2 (Hypervisor) filtering bit. Controls counting in EL2.

If Secure EL2 is implemented, and EL3 is implemented, counting in Secure EL2 is further controlled by the PMEVTYPER<n>_EL0.SH bit.

`0b0` Do not count events in EL2.

`0b1` Count events in EL2.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

M, bit [26]

When EL3 is implemented:

Secure EL3 filtering bit.

If the value of this bit is equal to the value of the PMEVTYPER<n>_EL0.P bit, events in Secure EL3 are counted.

Otherwise, events in Secure EL3 are not counted.

Most applications can ignore this field and set its value to `0b0`.

Note

This field is not visible in the AArch32 PMEVTYPER<n> System register.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MT, bit [25]

When (ARMv8.6-MTPMU is implemented and enabled) or an IMPLEMENTATION DEFINED multi-threaded PMU Extension is implemented:

Multithreading.

`0b0` Count events only on controlling PE.

`0b1` Count events from any PE with the same affinity at level 1 and above as this PE.

Note

- When the lowest level of affinity consists of logical PEs that are implemented using a multi-threading type approach, an implementation is described as multi-threaded. That is, the performance of PEs at the lowest affinity level is highly interdependent.
- Events from a different thread of a multithreaded implementation are not Attributable to the thread counting the event.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SH, bit [24]

When ARMv8.4-SecEL2 is implemented and EL3 is implemented:

Secure EL2 filtering.

If the value of this bit is not equal to the value of the PMEVTYPER<n>_EL0.NSH bit, events in Secure EL2 are counted.

Otherwise, events in Secure EL2 are not counted.

Note

This field is not visible in the AArch32 PMEVTYPER<n> System register.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [23:16]

Reserved, RES0.

evtCount[15:10], bits [15:10]

When ARMv8.1-PMU is implemented:

Extension to evtCount[9:0]. See evtCount[9:0] for more details.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

evtCount[9:0], bits [9:0]

Event to count. The event number of the event that is counted by event counter PMEVCNTR<n>_EL0.

Software must program this field with an event that is supported by the PE being programmed.

The ranges of event numbers allocated to each type of event are shown in Table D7-7 on page D7-2721.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If evtCount is programmed to an event that is reserved or not supported by the PE, the behavior depends on the value written:

- For the range `0x0000` to `0x003F`, no events are counted, and the value returned by a direct or external read of the evtCount field is the value written to the field.
- If 16-bit evtCount is implemented, for the range `0x4000` to `0x403F`, no events are counted, and the value returned by a direct or external read of the evtCount field is the value written to the field.
- For IMPLEMENTATION DEFINED events, it is UNPREDICTABLE what event, if any, is counted, and the value returned by a direct or external read of the evtCount field is UNKNOWN.

Note

UNPREDICTABLE means the event must not expose privileged information.

Arm recommends that the behavior across a family of implementations is defined such that if a given implementation does not include an event from a set of common IMPLEMENTATION DEFINED events, then no event is counted and the value read back on evtCount is the value written.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMEVTYPER<n>_EL0

PMEVTYPER<n>_EL0 can also be accessed by using PMXEVTYPER_EL0 with PMSELR_EL0.SEL set to n.

If ARMv8.6-FGT is implemented and <n> is greater than or equal to the number of accessible counters, then the behavior of permitted reads and writes of PMEVTYPER<n>_EL0 is as follows:

- If <n> is an unimplemented event counter, the access is UNDEFINED.
- Otherwise, the access is trapped to EL2.

If ARMv8.6-FGT is not implemented and <n> is greater than or equal to the number of accessible counters, then reads and writes of PMEVTYPER<n>_EL0 are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a `NOP`.
- If EL2 is implemented and enabled in the current Security state, and <n> is less than the number of implemented counters, accesses from EL1 or permitted accesses from EL0 are trapped to EL2.

Note

In EL0, an access is permitted if it is enabled by PMUSERENR_EL0.EN.

If EL2 is implemented and enabled in the current Security state, in EL1 and EL0, MDCR_EL2.HPMN identifies the number of accessible counters. Otherwise, the number of accessible counters is the number of implemented counters. See MDCR_EL2.HPMN for more details.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, PMEVTYPER<n>_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b11:n[4:3]	n[2:0]

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMEVTYPERn_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMEVTYPER_EL0[UInt(CRm<1:0>:op2<2:0>)];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.PMEVTYPERn_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMEVTYPER_EL0[UInt(CRm<1:0>:op2<2:0>)];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMEVTYPER_EL0[UInt(CRm<1:0>:op2<2:0>)];
elsif PSTATE.EL == EL3 then
    return PMEVTYPER_EL0[UInt(CRm<1:0>:op2<2:0>)];
```

MSR PMEVTYPER<n>_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b11:n[4:3]	n[2:0]

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMEVTYPERn_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMEVTYPER_EL0[UInt(CRm<1:0>:op2<2:0>)] = X[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.PMEVTYPERn_EL0 == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMEVTYPER_EL0[UInt(CRm<1:0>:op2<2:0>)] = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMEVTYPER_EL0[UInt(CRm<1:0>:op2<2:0>)] = X[t];
elsif PSTATE.EL == EL3 then
    PMEVTYPER_EL0[UInt(CRm<1:0>:op2<2:0>)] = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.4.10 PMINTENCLR_EL1, Performance Monitors Interrupt Enable Clear register

The PMINTENCLR_EL1 characteristics are:

Purpose

Disables the generation of interrupt requests on overflows from the Cycle Count Register, PMCCNTR_EL0, and the event counters PMEVCNTR<n>_EL0. Reading the register shows which overflow interrupt requests are enabled.

Configurations

AArch64 System register PMINTENCLR_EL1[31:0] is architecturally mapped to AArch32 System register PMINTENCLR[31:0].

AArch64 System register PMINTENCLR_EL1[31:0] is architecturally mapped to External register PMINTENCLR_EL1[31:0].

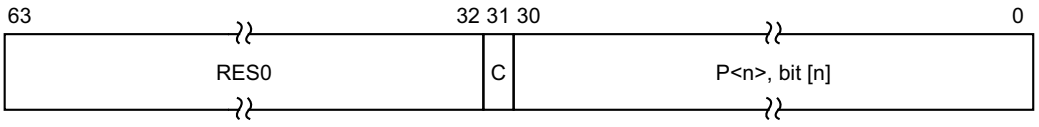
This register is present only when PMUv3 is implemented. Otherwise, direct accesses to PMINTENCLR_EL1 are UNDEFINED.

Attributes

PMINTENCLR_EL1 is a 64-bit register.

Field descriptions

The PMINTENCLR_EL1 bit assignments are:



Bits [63:32]

Reserved, RES0.

C, bit [31]

PMCCNTR_EL0 overflow interrupt request disable bit. Possible values are:

`0b0` When read, means the cycle counter overflow interrupt request is disabled. When written, has no effect.

`0b1` When read, means the cycle counter overflow interrupt request is enabled. When written, disables the cycle count overflow interrupt request.

P<n>, bit [n], for n = 0 to 30

Event counter overflow interrupt request disable bit for PMEVCNTR<n>_EL0.

If N is less than 31, then bits [30:N] are RAZ/WI. When EL2 is implemented and enabled in the current Security state, in EL1, N is the value in MDCR_EL2.HPMN. Otherwise, N is the value in PMCR_EL0.N.

`0b0` When read, means that the PMEVCNTR<n>_EL0 event counter interrupt request is disabled. When written, has no effect.

`0b1` When read, means that the PMEVCNTR<n>_EL0 event counter interrupt request is enabled. When written, disables the PMEVCNTR<n>_EL0 interrupt request.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMINTENCLR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, PMINTENCLR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1110	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.PMINTEN == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMINTENCLR_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMINTENCLR_EL1;
elsif PSTATE.EL == EL3 then
    return PMINTENCLR_EL1;
```

MSR PMINTENCLR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1110	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.PMINTEN == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMINTENCLR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMINTENCLR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    PMINTENCLR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.4.11 PMINTENSET_EL1, Performance Monitors Interrupt Enable Set register

The PMINTENSET_EL1 characteristics are:

Purpose

Enables the generation of interrupt requests on overflows from the Cycle Count Register, PMCCNTR_EL0, and the event counters PMEVCNTR<n>_EL0. Reading the register shows which overflow interrupt requests are enabled.

Configurations

AArch64 System register PMINTENSET_EL1[31:0] is architecturally mapped to AArch32 System register PMINTENSET[31:0].

AArch64 System register PMINTENSET_EL1[31:0] is architecturally mapped to External register PMINTENSET_EL1[31:0].

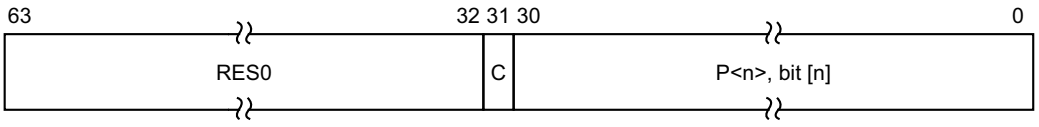
This register is present only when PMUv3 is implemented. Otherwise, direct accesses to PMINTENSET_EL1 are UNDEFINED.

Attributes

PMINTENSET_EL1 is a 64-bit register.

Field descriptions

The PMINTENSET_EL1 bit assignments are:



Bits [63:32]

Reserved, RES0.

C, bit [31]

PMCCNTR_EL0 overflow interrupt request enable bit. Possible values are:

`0b0` When read, means the cycle counter overflow interrupt request is disabled. When written, has no effect.

`0b1` When read, means the cycle counter overflow interrupt request is enabled. When written, enables the cycle count overflow interrupt request.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

P<n>, bit [n], for n = 0 to 30

Event counter overflow interrupt request enable bit for PMEVCNTR<n>_EL0.

If N is less than 31, then bits [30:N] are RAZ/WI. When EL2 is implemented and enabled in the current Security state, in EL1, N is the value in MDCR_EL2.HPMN. Otherwise, N is the value in PMCR_EL0.N.

`0b0` When read, means that the PMEVCNTR<n>_EL0 event counter interrupt request is disabled. When written, has no effect.

`0b1` When read, means that the PMEVCNTR<n>_EL0 event counter interrupt request is enabled. When written, enables the PMEVCNTR<n>_EL0 interrupt request.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMINTENSET_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, PMINTENSET_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1110	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.PMINTEN == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMINTENSET_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMINTENSET_EL1;
elsif PSTATE.EL == EL3 then
    return PMINTENSET_EL1;
```

MSR PMINTENSET_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1110	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.PMINTEN == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMINTENSET_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMINTENSET_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    PMINTENSET_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.4.12 PMMIR_EL1, Performance Monitors Machine Identification Register

The PMMIR_EL1 characteristics are:

Purpose

Describes Performance Monitors parameters specific to the implementation to software.

Configurations

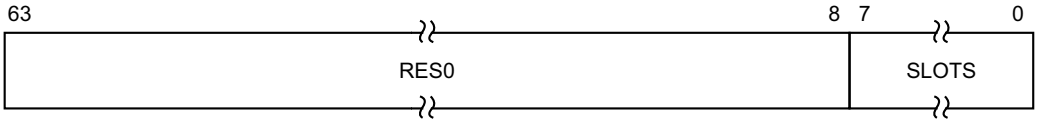
This register is present only when ARMv8.4-PMU is implemented. Otherwise, direct accesses to PMMIR_EL1 are UNDEFINED.

Attributes

PMMIR_EL1 is a 64-bit register.

Field descriptions

The PMMIR_EL1 bit assignments are:



Bits [63:8]

Reserved, RES0.

SLOTS, bits [7:0]

Operation width. The largest value by which the STALL_SLOT event might increment by in a single cycle. If the STALL_SLOT event is not implemented, this field might read as zero.

Accessing the PMMIR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PMMIR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1110	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.PMMIR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMMIR_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMMIR_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    return PMMIR_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.4.13 PMOVSCLR_EL0, Performance Monitors Overflow Flag Status Clear Register

The PMOVSCLR_EL0 characteristics are:

Purpose

Contains the state of the overflow bit for the Cycle Count Register, PMCCNTR_EL0, and each of the implemented event counters PMEVCNTR<n>. Writing to this register clears these bits.

Configurations

AArch64 System register PMOVSCLR_EL0[31:0] is architecturally mapped to AArch32 System register PMOVSR[31:0].

AArch64 System register PMOVSCLR_EL0[31:0] is architecturally mapped to External register PMOVSCLR_EL0[31:0].

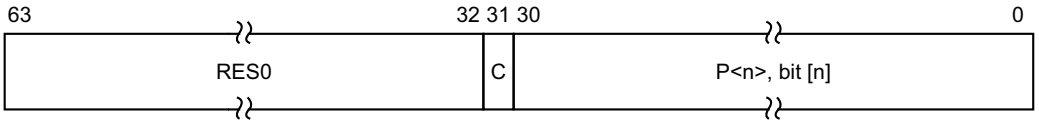
This register is present only when PMUv3 is implemented. Otherwise, direct accesses to PMOVSCLR_EL0 are UNDEFINED.

Attributes

PMOVSCLR_EL0 is a 64-bit register.

Field descriptions

The PMOVSCLR_EL0 bit assignments are:



Bits [63:32]

Reserved, RES0.

C, bit [31]

Cycle counter overflow clear bit.

`0b0` When read, means the cycle counter has not overflowed since this bit was last cleared. When written, has no effect.

`0b1` When read, means the cycle counter has overflowed since this bit was last cleared. When written, clears the cycle counter overflow bit to 0.

PMCR_EL0.LC controls whether an overflow is detected from unsigned overflow of PMCCNTR_EL0[31:0] or unsigned overflow of PMCCNTR_EL0[63:0].

On a Warm reset, this field resets to an architecturally UNKNOWN value.

P<n>, bit [n], for n = 0 to 30

Event counter overflow clear bit for PMEVCNTR<n>_EL0.

If N is less than 31, then bits [30:N] are RAZ/WI. When EL2 is implemented and enabled in the current Security state, in EL1 and EL0, N is the value in MDCR_EL2.HPMN. Otherwise, N is the value in PMCR_EL0.N.

`0b0` When read, means that PMEVCNTR<n>_EL0 has not overflowed since this bit was last cleared. When written, has no effect.

`0b1` When read, means that PMEVCNTR<n>_EL0 has overflowed since this bit was last cleared. When written, clears the PMEVCNTR<n>_EL0 overflow bit to 0.

If ARMv8.5-PMU is implemented, MDCR_EL2.HLP and PMCR_EL0.LP control whether an overflow is detected from unsigned overflow of PMEVCNTR<n>_EL0[31:0] or unsigned overflow of PMEVCNTR<n>_EL0[63:0].

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the PMOVSCLR_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PMOVSCLR_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b011

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMOVS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMOVSCLR_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.PMOVS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMOVSCLR_EL0;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMOVSCLR_EL0;
elsif PSTATE.EL == EL3 then
    return PMOVSCLR_EL0;
```

MSR PMOVSCLR_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b011

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMOVS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        PMOVSCLR_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.PMOVS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMOVSCLR_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMOVSCLR_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    PMOVSCLR_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.4.14 PMOVSSET_EL0, Performance Monitors Overflow Flag Status Set register

The PMOVSSET_EL0 characteristics are:

Purpose

Sets the state of the overflow bit for the Cycle Count Register, PMCCNTR_EL0, and each of the implemented event counters PMEVCNTR<n>.

Configurations

AArch64 System register PMOVSSET_EL0[31:0] is architecturally mapped to AArch32 System register PMOVSSET[31:0].

AArch64 System register PMOVSSET_EL0[31:0] is architecturally mapped to External register PMOVSSET_EL0[31:0].

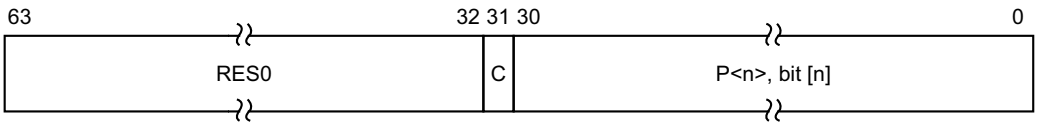
This register is present only when PMUv3 is implemented. Otherwise, direct accesses to PMOVSSET_EL0 are UNDEFINED.

Attributes

PMOVSSET_EL0 is a 64-bit register.

Field descriptions

The PMOVSSET_EL0 bit assignments are:



Bits [63:32]

Reserved, RES0.

C, bit [31]

Cycle counter overflow set bit.

`0b0` When read, means the cycle counter has not overflowed since this bit was last cleared. When written, has no effect.

`0b1` When read, means the cycle counter has overflowed since this bit was last cleared. When written, sets the cycle counter overflow bit to 1.

PMCR_EL0.LC controls whether an overflow is detected from unsigned overflow of PMCCNTR_EL0[31:0] or unsigned overflow of PMCCNTR_EL0[63:0].

On a Warm reset, this field resets to an architecturally UNKNOWN value.

P<n>, bit [n], for n = 0 to 30

Event counter overflow set bit for PMEVCNTR<n>_EL0.

If N is less than 31, then bits [30:N] are RAZ/WI. When EL2 is implemented and enabled in the current Security state, in EL1 and EL0, N is the value in MDCR_EL2.HPMN. Otherwise, N is the value in PMCR_EL0.N.

`0b0` When read, means that PMEVCNTR<n>_EL0 has not overflowed since this bit was last cleared. When written, has no effect.

`0b1` When read, means that PMEVCNTR<n>_EL0 has overflowed since this bit was last cleared. When written, sets the PMEVCNTR<n>_EL0 overflow bit to 1.

If ARMv8.5-PMU is implemented, MDCR_EL2.HLP and PMCR_EL0.LP control whether an overflow is detected from unsigned overflow of PMEVCNTR<n>_EL0[31:0] or unsigned overflow of PMEVCNTR<n>_EL0[63:0].

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the PMOVSSET_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PMOVSSET_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1110	0b011

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMOVS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMOVSSET_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.PMOVS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMOVSSET_EL0;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMOVSSET_EL0;
elsif PSTATE.EL == EL3 then
    return PMOVSSET_EL0;
```

MSR PMOVSSET_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1110	0b011

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMOVS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        PMOVSSET_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.PMOVS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMOVSSET_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMOVSSET_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    PMOVSSET_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.4.15 PMSELR_EL0, Performance Monitors Event Counter Selection Register

The PMSELR_EL0 characteristics are:

Purpose

Selects the current event counter PMEVCNTR<n>_EL0 or the cycle counter, CCNT.

PMSELR_EL0 is used in conjunction with PMXEVTYPER_EL0 to determine the event that increments a selected event counter, and the modes and states in which the selected counter increments.

It is also used in conjunction with PMXEVCNTR_EL0, to determine the value of a selected event counter.

Configurations

AArch64 System register PMSELR_EL0[31:0] is architecturally mapped to AArch32 System register PMSELR[31:0].

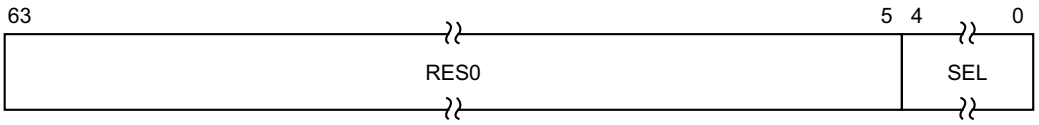
This register is present only when PMUv3 is implemented. Otherwise, direct accesses to PMSELR_EL0 are UNDEFINED.

Attributes

PMSELR_EL0 is a 64-bit register.

Field descriptions

The PMSELR_EL0 bit assignments are:



Bits [63:5]

Reserved, RES0.

SEL, bits [4:0]

Selects event counter, PMEVCNTR<n>_EL0, where n is the value held in this field. This value identifies which event counter is accessed when a subsequent access to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 occurs.

This field can take any value from 0 (`0b00000`) to (PMCR.N)-1, or 31 (`0b11111`).

When PMSELR_EL0.SEL is `0b11111`, it selects the cycle counter and:

- A read of the PMXEVTYPER_EL0 returns the value of PMCCFILTR_EL0.
- A write of the PMXEVTYPER_EL0 writes to PMCCFILTR_EL0.
- A read or write of PMXEVCNTR_EL0 has CONSTRAINED UNPREDICTABLE effects. See PMXEVCNTR_EL0 for more details.

For details of the results of accesses to the event counters, see PMXEVTYPER_EL0 and PMXEVCNTR_EL0.

For information about the number of counters accessible at each Exception level, see MDCR_EL2.HPMN.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMSELR_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, PMSELR_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b101

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.<ER,EN> == '00' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMSELR_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMSELR_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.PMSELR_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMSELR_EL0;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMSELR_EL0;
elsif PSTATE.EL == EL3 then
    return PMSELR_EL0;
```

MSR PMSELR_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b101

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.<ER,EN> == '00' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMSELR_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMSELR_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.PMSELR_EL0 == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMSELR_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMSELR_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    PMSELR_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.4.16 PMSWINC_EL0, Performance Monitors Software Increment register

The PMSWINC_EL0 characteristics are:

Purpose

Increments a counter that is configured to count the Software increment event, event `0x00`. For more information, see SW_INCR.

Configurations

AArch64 System register PMSWINC_EL0[31:0] is architecturally mapped to AArch32 System register PMSWINC[31:0].

AArch64 System register PMSWINC_EL0[31:0] is architecturally mapped to External register PMSWINC_EL0[31:0].

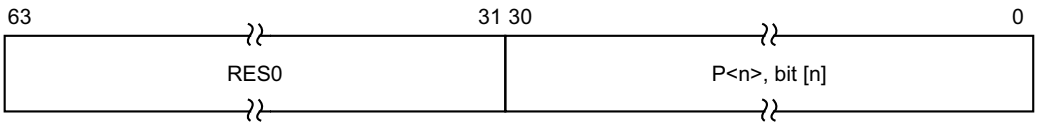
This register is present only when PMUv3 is implemented. Otherwise, direct accesses to PMSWINC_EL0 are UNDEFINED.

Attributes

PMSWINC_EL0 is a 64-bit register.

Field descriptions

The PMSWINC_EL0 bit assignments are:



Bits [63:31]

Reserved, RES0.

P<n>, bit [n], for n = 0 to 30

Event counter software increment bit for PMEVCNTR<n>_EL0.

If N is less than 31, then bits [30:N] are WI. When EL2 is implemented and enabled in the current Security state, in EL1 and EL0, N is the value in MDCR_EL2.HPMN. Otherwise, N is the value in PMCR_EL0.N.

`0b0` No action. The write to this bit is ignored.

`0b1` If PMEVCNTR<n>_EL0 is enabled and configured to count the software increment event, increments PMEVCNTR<n>_EL0 by 1. If PMEVCNTR<n>_EL0 is disabled, or not configured to count the software increment event, the write to this bit is ignored.

Accessing the PMSWINC_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

MSR PMSWINC_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b100

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.<SW,EN> == '00' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMSWINC_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMSWINC_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.PMSWINC_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMSWINC_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMSWINC_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    PMSWINC_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.4.17 PMUSERENR_EL0, Performance Monitors User Enable Register

The PMUSERENR_EL0 characteristics are:

Purpose

Enables or disables EL0 access to the Performance Monitors.

Configurations

AArch64 System register PMUSERENR_EL0[31:0] is architecturally mapped to AArch32 System register PMUSERENR[31:0].

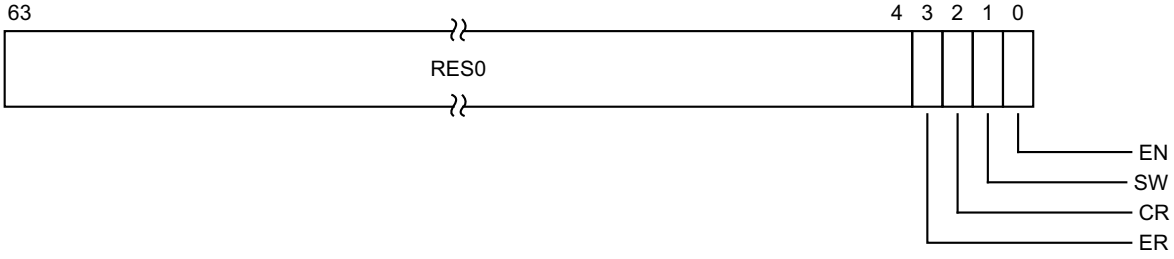
This register is present only when PMUv3 is implemented. Otherwise, direct accesses to PMUSERENR_EL0 are UNDEFINED.

Attributes

PMUSERENR_EL0 is a 64-bit register.

Field descriptions

The PMUSERENR_EL0 bit assignments are:



Bits [63:4]

Reserved, RES0.

ER, bit [3]

Event counter Read. Traps EL0 access to event counters to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1.

In AArch64 state, trapped accesses are reported using EC syndrome value `0x18`.

In AArch32 state, trapped accesses are reported using EC syndrome value `0x03`.

`0b0` EL0 using AArch64: EL0 reads of the PMXEVCNTR_EL0 and PMEVCNTR<n>_EL0, and EL0 read/write accesses to the PMSELR_EL0, are trapped if PMUSERENR_EL0.EN is also 0.

EL0 using AArch32: EL0 reads of the PMXEVCNTR and PMEVCNTR<n>, and EL0 read/write accesses to the PMSELR, are trapped if PMUSERENR_EL0.EN is also 0.

`0b1` Overrides PMUSERENR_EL0.EN and enables RO access to PMXEVCNTR_EL0 and PMEVCNTR<n>_EL0, and RW access to PMSELR_EL0 and PMSELR at EL0.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

CR, bit [2]

Cycle counter Read. Traps EL0 access to cycle counter reads to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1.

In AArch64 state, trapped accesses are reported using EC syndrome value `0x18`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In AArch32 state, trapped MRC accesses are reported using EC syndrome value `0x03`, trapped MRRC accesses are reported using EC syndrome value `0x04`.

`0b0` EL0 using AArch64: EL0 read accesses to the PMCCNTR_EL0 are trapped if PMUSERENR_EL0.EN is also 0.

EL0 using AArch32: EL0 read accesses to the PMCCNTR are trapped if PMUSERENR_EL0.EN is also 0.

`0b1` Overrides PMUSERENR_EL0.EN and enables access to PMCCNTR_EL0 and PMCCNTR at EL0.

SW, bit [1]

Traps Software Increment writes to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1.

In AArch64 state, trapped accesses are reported using EC syndrome value `0x18`.

In AArch32 state, trapped accesses are reported using EC syndrome value `0x03`.

`0b0` EL0 using AArch64: EL0 writes to the PMSWINC_EL0 are trapped if PMUSERENR_EL0.EN is also 0.

EL0 using AArch32: EL0 writes to the PMSWINC are trapped if PMUSERENR_EL0.EN is also 0.

`0b1` Overrides PMUSERENR_EL0.EN and enables access to:

- PMSWINC_EL0 at EL0.
- PMSWINC at EL0.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

EN, bit [0]

Traps EL0 accesses to the Performance Monitor registers to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from both Execution states as follows:

- In AArch64 state, MRS or MSR accesses to the following registers are reported using EC syndrome value `0x18`:
  - PMCR_EL0, PMOVSCLR_EL0, PMSELR_EL0, PMCEID0_EL0, PMCEID1_EL0, PMCCNTR_EL0, PMXEVTYPER_EL0, PMXEVCNTR_EL0, PMCNTENSET_EL0, PMCNTENSET_EL0, PMOVSSET_EL0, PMEVCNTR<n>_EL0, PMEVTYPER<n>_EL0, PMCCFILTR_EL0.
  - PMSWINC_EL0, MSR accesses only.
  - If ARMv8.4-PMU is implemented, PMMIR_EL1
- In AArch32 state, MRC or MCR accesses to the following registers are reported using EC syndrome value `0x03`, MRRC or MCRR accesses are trapped to EL2 and reported using EC syndrome value `0x04`:
  - PMCR, PMOVSR, PMSELR, PMCEID0, PMCEID1, PMCCNTR, PMXEVTYPER, PMXEVCNTR, PMCNTENSET, PMCNTENCLR, PMOVSSET, PMEVCNTR<n>, PMEVTYPER<n>, PMCCFILTR.
  - PMSWINC, MCR accesses only.
  - If ARMv8.4-PMU is implemented, PMMIR.
  - If ARMv8.1-PMU is implemented, in AArch32 state, PMCEID2, and PMCEID3.
- In AArch32 state, MRRC or MCRR accesses to PMCCNTR are reported using EC syndrome value `0x04`.

`0b0` While at EL0, accesses to the specified registers at EL0 are trapped, unless overridden by one of PMUSERENR_EL0.{ER, CR, SW}.

`0b1` While at EL0, software can access all of the specified registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

The EL0 access is trapped only if the corresponding EL1 access is permitted. If PMUSERENR_EL0.EN is 0, write access to PMSWINC_EL0 and PMSWINC from EL0 are trapped, but read access is UNDEFINED.

The affected registers do not include PMINTENSET_EL1, PMINTENCLR_EL1, PMINTENSET and PMINTENCLR.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMUSERENR_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PMUSERENR_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1110	0b000

```
if PSTATE.EL == EL0 then
    if EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMUSERENR_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMUSERENR_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.PMUSERENR_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMUSERENR_EL0;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMUSERENR_EL0;
elsif PSTATE.EL == EL3 then
    return PMUSERENR_EL0;
```

MSR PMUSERENR_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1110	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.PMUSERENR_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMUSERENR_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMUSERENR_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    PMUSERENR_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.4.18 PMXEVCNTR_EL0, Performance Monitors Selected Event Count Register

The PMXEVCNTR_EL0 characteristics are:

Purpose

Reads or writes the value of the selected event counter, PMEVCNTR<n>_EL0. PMSELR_EL0.SEL determines which event counter is selected.

Configurations

AArch64 System register PMXEVCNTR_EL0[31:0] is architecturally mapped to AArch32 System register PMXEVCNTR[31:0].

This register is present only when PMUv3 is implemented. Otherwise, direct accesses to PMXEVCNTR_EL0 are UNDEFINED.

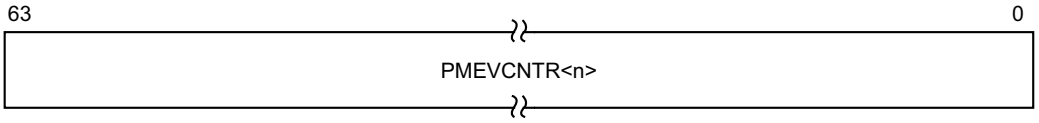
Attributes

PMXEVCNTR_EL0 is a 64-bit register.

Field descriptions

The PMXEVCNTR_EL0 bit assignments are:

When ARMv8.5-PMU is implemented:

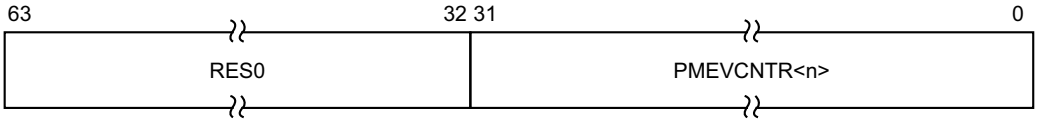


PMEVCNTR<n>, bits [63:0]

Value of the selected event counter, PMEVCNTR<n>_EL0, where n is the value stored in PMSELR_EL0.SEL.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:



Bits [63:32]

Reserved, RES0.

PMEVCNTR<n>, bits [31:0]

Value of the selected event counter, PMEVCNTR<n>_EL0, where n is the value stored in PMSELR_EL0.SEL.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMXEVCNTR_EL0

If ARMv8.6-FGT is implemented and PMSELR_EL0.SEL is greater than or equal to the number of accessible counters, then the behavior of permitted reads and writes of PMXEVCNTR_EL0 is as follows:

- If PMSELR_EL0.SEL selects an unimplemented event counter, the access is UNDEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Otherwise, the access is trapped to EL2.

If ARMv8.6-FGT is not implemented and PMSELR_EL0.SEL is greater than or equal to the number of accessible counters, then reads and writes of PMXEVCNTR_EL0 are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP
- Accesses to the register behave as if PMSELR_EL0.SEL has an UNKNOWN value less than the number of counters accessible at the current Exception level and Security state.
- If EL2 is implemented and enabled in the current Security state, and PMSELR_EL0.SEL is less than the number of implemented counters, accesses from EL1 or permitted accesses from EL0 are trapped to EL2.

Note

In EL0, an access is permitted if it is enabled by PMUSERENR_EL0.{ER,EN}.

If EL2 is implemented and enabled in the current Security state, in EL1 and EL0, MDCR_EL2.HPMN identifies the number of accessible counters. Otherwise, the number of accessible counters is the number of implemented counters. See MDCR_EL2.HPMN for more details.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PMXEVCNTR_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1101	0b010

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.<ER,EN> == '00' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMEVCNTRn_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMXEVCNTR_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.PMEVCNTRn_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMXEVCNTR_EL0;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMXEVCNTR_EL0;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    return PMXEVCNTR_EL0;
```

MSR PMXEVCNTR_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1101	0b010

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMEVCNTRn_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMXEVCNTR_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.PMEVCNTRn_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMXEVCNTR_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMXEVCNTR_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    PMXEVCNTR_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.4.19 PMXEVTYPER_EL0, Performance Monitors Selected Event Type Register

The PMXEVTYPER_EL0 characteristics are:

Purpose

When PMSELR_EL0.SEL selects an event counter, this accesses a PMEVTYPER<n>_EL0 register. When PMSELR_EL0.SEL selects the cycle counter, this accesses PMCCFILTR_EL0.

Configurations

AArch64 System register PMXEVTYPER_EL0[31:0] is architecturally mapped to AArch32 System register PMXEVTYPER[31:0].

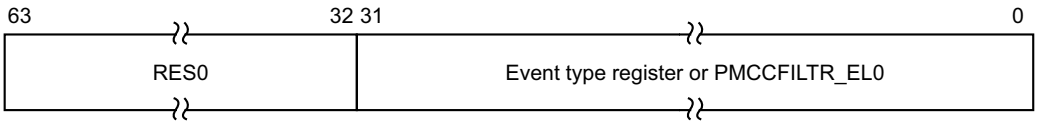
This register is present only when PMUv3 is implemented. Otherwise, direct accesses to PMXEVTYPER_EL0 are UNDEFINED.

Attributes

PMXEVTYPER_EL0 is a 64-bit register.

Field descriptions

The PMXEVTYPER_EL0 bit assignments are:



Bits [63:32]

Reserved, RES0.

Bits [31:0]

When PMSELR_EL0.SEL == 31, this register accesses PMCCFILTR_EL0.

Otherwise, this register accesses PMEVTYPER<n>_EL0 where n is the value in PMSELR_EL0.SEL.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMXEVTYPER_EL0

If ARMv8.6-FGT is implemented, and PMSELR_EL0.SEL is not 31 and is greater than or equal to the number of accessible counters, then the behavior of permitted reads and writes of PMXEVTYPER_EL0 is as follows:

- If PMSELR_EL0.SEL selects an unimplemented event counter, the access is UNDEFINED.
- Otherwise, the access is trapped to EL2.

If ARMv8.6-FGT is not implemented, and PMSELR_EL0.SEL is not 31 and is greater than or equal to the number of accessible counters, then reads and writes of PMXEVTYPER_EL0 are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a `NOP`
- Accesses to the register behave as if PMSELR_EL0.SEL has an UNKNOWN value less than the number of counters accessible at the current Exception level and Security state.
- Accesses to the register behave as if PMSELR_EL0.SEL is 31.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If EL2 is implemented and enabled in the current Security state, PMSELR_EL0 is less than the number of implemented counters, accesses from EL1 or permitted accesses from EL0 are trapped to EL2.

Note

In EL0, an access is permitted if it is enabled by PMUSERENR_EL0.EN.

If EL2 is implemented and enabled in the current Security state, in EL1 and EL0, MDCR_EL2.HPMN identifies the number of accessible counters. Otherwise, the number of accessible counters is the number of implemented counters. See MDCR_EL2.HPMN for more details.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PMXEVTYPER_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1101	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMEVTYPERn_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMXEVTYPER_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.PMEVTYPERn_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMXEVTYPER_EL0;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMXEVTYPER_EL0;
elsif PSTATE.EL == EL3 then
    return PMXEVTYPER_EL0;
```

MSR PMXEVTYPER_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1101	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMEVTYPERn_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMXEVTYPER_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.PMEVTYPERn_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMXEVTYPER_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMXEVTYPER_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    PMXEVTYPER_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.5 Activity Monitors registers

This section lists the Activity Monitors registers in AArch64.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.5.1 AMCFGR_EL0, Activity Monitors Configuration Register

The AMCFGR_EL0 characteristics are:

Purpose

Global configuration register for the activity monitors.

Provides information on supported features, the number of counter groups implemented, the total number of activity monitor event counters implemented, and the size of the counters. AMCFGR_EL0 is applicable to both the architected and the auxiliary counter groups.

Configurations

AArch64 System register AMCFGR_EL0[31:0] is architecturally mapped to AArch32 System register AMCFGR[31:0].

AArch64 System register AMCFGR_EL0[31:0] is architecturally mapped to External register AMCFGR[31:0].

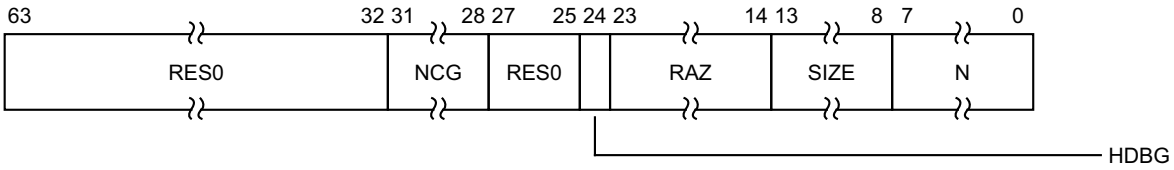
This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMCFGR_EL0 are UNDEFINED.

Attributes

AMCFGR_EL0 is a 64-bit register.

Field descriptions

The AMCFGR_EL0 bit assignments are:



Bits [63:32]

Reserved, RES0.

NCG, bits [31:28]

Defines the number of counter groups.

The number of implemented counter groups is defined as [AMCFGR_EL0.NCG + 1].

If the number of implemented auxiliary activity monitor event counters is zero, this field has a value of `0b0000`. Otherwise, this field has a value of `0b0001`.

Bits [27:25]

Reserved, RES0.

HDBG, bit [24]

Halt-on-debug supported.

From Armv8, this feature must be supported, and so this bit is `0b1`.

`0b0` AMCR_EL0.HDBG is RES0.

`0b1` AMCR_EL0.HDBG is read/write.

Bits [23:14]

Reserved, RAZ.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

SIZE, bits [13:8]

Defines the size of activity monitor event counters.

The size of the activity monitor event counters implemented by the activity monitors Extension is defined as [AMCFGR_EL0.SIZE + 1].

From Armv8, the counters are 64-bit, and so this field is `0b111111`.

Note

Software also uses this field to determine the spacing of counters in the memory-map. From Armv8, the counters are at doubleword-aligned addresses.

N, bits [7:0]

Defines the number of activity monitor event counters.

The total number of counters implemented in all groups by the Activity Monitors Extension is defined as [AMCFGR_EL0.N + 1].

Accessing the AMCFGR_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, AMCFGR_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0010	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && AMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMCFGR_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMCFGR_EL0;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMCFGR_EL0;
elsif PSTATE.EL == EL3 then
    return AMCFGR_EL0;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.5.2 AMCG1IDR_EL0, Activity Monitors Counter Group 1 Identification Register

The AMCG1IDR_EL0 characteristics are:

Purpose

Defines which auxiliary counters are implemented, and which of them have a corresponding virtual offset register, AMEVCNTVOFF1<n>_EL2 implemented.

Configurations

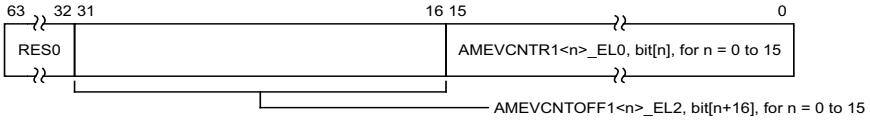
This register is present only when ARMv8.6-AMU is implemented. Otherwise, direct accesses to AMCG1IDR_EL0 are UNDEFINED.

Attributes

AMCG1IDR_EL0 is a 64-bit register.

Field descriptions

The AMCG1IDR_EL0 bit assignments are:



Bits [63:32]

Reserved, RES0.

AMEVCNTOFF1<n>_EL2, bit [n+16], for n = 0 to 15

Indicates which implemented auxiliary counters have a corresponding virtual offset register, AMEVCNTVOFF1<n>_EL2 implemented.

`0b0` When read, mean that AMEVCNTR1<n>_EL0 does not have an offset, or is not implemented.

`0b1` When read, means the offset AMEVCNTVOFF1<n>_EL2 is implemented for AMEVCNTR1<n>_EL0.

AMEVCNTR1<n>_EL0, bit [n], for n = 0 to 15

Indicates which auxiliary counters AMEVCNTR1<n>_EL0 are implemented.

`0b0` When read, means that AMEVCNTR1<n>_EL0 is not implemented.

`0b1` When read, means that AMEVCNTR1<n>_EL0 is implemented.

Accessing the AMCG1IDR_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, AMCG1IDR_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0010	0b110

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && AMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMCG1IDR_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMCG1IDR_EL0;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMCG1IDR_EL0;
elsif PSTATE.EL == EL3 then
    return AMCG1IDR_EL0;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.5.3 AMCGCR_EL0, Activity Monitors Counter Group Configuration Register

The AMCGCR_EL0 characteristics are:

Purpose

Provides information on the number of activity monitor event counters implemented within each counter group.

Configurations

AArch64 System register AMCGCR_EL0[31:0] is architecturally mapped to AArch32 System register AMCGCR[31:0].

AArch64 System register AMCGCR_EL0[31:0] is architecturally mapped to External register AMCGCR[31:0].

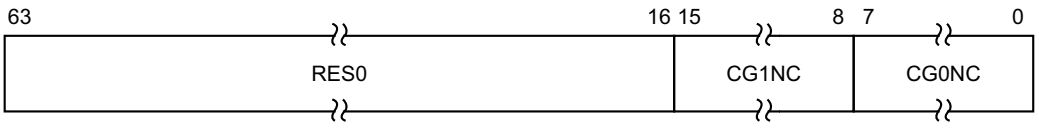
This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMCGCR_EL0 are UNDEFINED.

Attributes

AMCGCR_EL0 is a 64-bit register.

Field descriptions

The AMCGCR_EL0 bit assignments are:



Bits [63:16]

Reserved, RES0.

CG1NC, bits [15:8]

Counter Group 1 Number of Counters. The number of counters in the auxiliary counter group.

In AMUv1, the permitted range of values is `0x0` to `0x10`.

CG0NC, bits [7:0]

Counter Group 0 Number of Counters. The number of counters in the architected counter group.

In AMUv1, the value of this field is `0x4`.

Accessing the AMCGCR_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, AMCGCR_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0010	0b010

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && AMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMCGCR_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMCGCR_EL0;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMCGCR_EL0;
elsif PSTATE.EL == EL3 then
    return AMCGCR_EL0;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.5.4 AMCNTENCLR0_EL0, Activity Monitors Count Enable Clear Register 0

The AMCNTENCLR0_EL0 characteristics are:

Purpose

Disable control bits for the architected activity monitors event counters, AMEVCNTR0<n>_EL0.

Configurations

AArch64 System register AMCNTENCLR0_EL0[31:0] is architecturally mapped to AArch32 System register AMCNTENCLR0[31:0].

AArch64 System register AMCNTENCLR0_EL0[31:0] is architecturally mapped to External register AMCNTENCLR0[31:0].

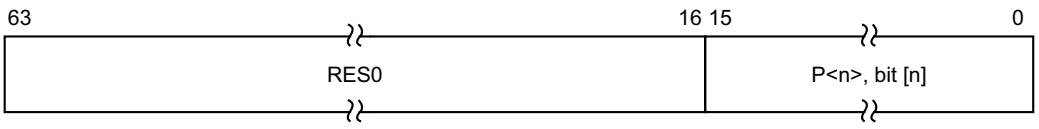
This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMCNTENCLR0_EL0 are UNDEFINED.

Attributes

AMCNTENCLR0_EL0 is a 64-bit register.

Field descriptions

The AMCNTENCLR0_EL0 bit assignments are:



Bits [63:16]

Reserved, RES0.

P<n>, bit [n], for n = 0 to 15

Activity monitor event counter disable bit for AMEVCNTR0<n>_EL0.

Bits [31:16] are RES0. Bits [15:N] are RAZ/WI. N is the value in AMCGCR_EL0.CG0NC.

Possible values of each bit are:

`0b0` When read, means that AMEVCNTR0<n>_EL0 is disabled. When written, has no effect.

`0b1` When read, means that AMEVCNTR0<n>_EL0 is enabled. When written, disables AMEVCNTR0<n>_EL0.

On a Cold reset, this field resets to `0`.

Accessing the AMCNTENCLR0_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, AMCNTENCLR0_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0010	0b100

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && AMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HAFGRTR_EL2.AMCNTEN0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMCNTENCLR0_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HAFGRTR_EL2.AMCNTEN0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMCNTENCLR0_EL0;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMCNTENCLR0_EL0;
elsif PSTATE.EL == EL3 then
    return AMCNTENCLR0_EL0;
```

MSR AMCNTENCLR0_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0010	0b100

```
if IsHighestEL(PSTATE.EL) then
    AMCNTENCLR0_EL0 = X[t];
else
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.5.5 AMCNTENCLR1_EL0, Activity Monitors Count Enable Clear Register 1

The AMCNTENCLR1_EL0 characteristics are:

Purpose

Disable control bits for the auxiliary activity monitors event counters, AMEVCNTR1<n>_EL0.

Configurations

AArch64 System register AMCNTENCLR1_EL0[31:0] is architecturally mapped to AArch32 System register AMCNTENCLR1[31:0].

AArch64 System register AMCNTENCLR1_EL0[31:0] is architecturally mapped to External register AMCNTENCLR1[31:0].

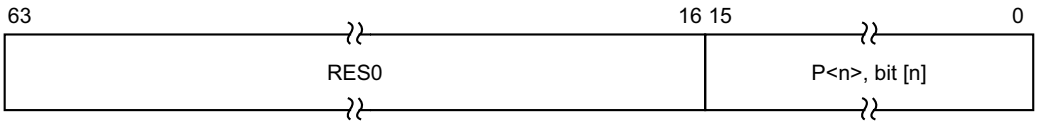
This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMCNTENCLR1_EL0 are UNDEFINED.

Attributes

AMCNTENCLR1_EL0 is a 64-bit register.

Field descriptions

The AMCNTENCLR1_EL0 bit assignments are:



Bits [63:16]

Reserved, RES0.

P<n>, bit [n], for n = 0 to 15

Activity monitor event counter disable bit for AMEVCNTR1<n>_EL0.

Bits [31:16] are RES0. Bits [15:N] are RAZ/WI. N is the value in AMCGCR_EL0.CG1NC.

Possible values of each bit are:

`0b0` When read, means that AMEVCNTR1<n>_EL0 is disabled. When written, has no effect.

`0b1` When read, means that AMEVCNTR1<n>_EL0 is enabled. When written, disables AMEVCNTR1<n>_EL0.

On a Cold reset, this field resets to `0`.

Accessing the AMCNTENCLR1_EL0

If the number of auxiliary activity monitor event counters implemented is zero, reads and writes of AMCNTENCLR1_EL0 are UNDEFINED.

Note

The number of auxiliary activity monitor event counters implemented is zero exactly when AMCFGR_EL0.NCG == `0b0000`.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, AMCNTENCLR1_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0011	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && AMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HAFGRTR_EL2.AMCNTEN1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMCNTENCLR1_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HAFGRTR_EL2.AMCNTEN1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMCNTENCLR1_EL0;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMCNTENCLR1_EL0;
elsif PSTATE.EL == EL3 then
    return AMCNTENCLR1_EL0;
```

MSR AMCNTENCLR1_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0011	0b000

```
if IsHighestEL(PSTATE.EL) then
    AMCNTENCLR1_EL0 = X[t];
else
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.5.6 AMCNTENSET0_EL0, Activity Monitors Count Enable Set Register 0

The AMCNTENSET0_EL0 characteristics are:

Purpose

Enable control bits for the architected activity monitors event counters, AMEVCNTR0<n>_EL0.

Configurations

AArch64 System register AMCNTENSET0_EL0[31:0] is architecturally mapped to AArch32 System register AMCNTENSET0[31:0].

AArch64 System register AMCNTENSET0_EL0[31:0] is architecturally mapped to External register AMCNTENSET0[31:0].

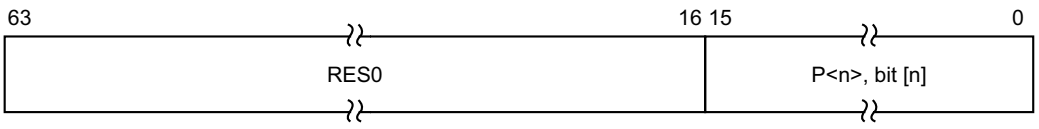
This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMCNTENSET0_EL0 are UNDEFINED.

Attributes

AMCNTENSET0_EL0 is a 64-bit register.

Field descriptions

The AMCNTENSET0_EL0 bit assignments are:



Bits [63:16]

Reserved, RES0.

P<n>, bit [n], for n = 0 to 15

Activity monitor event counter enable bit for AMEVCNTR0<n>_EL0.

Bits [31:16] are RES0. Bits [15:N] are RAZ/WI. N is the value in AMCGCR_EL0.CG0NC.

Possible values of each bit are:

`0b0` When read, means that AMEVCNTR0<n>_EL0 is disabled. When written, has no effect.

`0b1` When read, means that AMEVCNTR0<n>_EL0 is enabled. When written, enables AMEVCNTR0<n>_EL0.

On a Cold reset, this field resets to `0`.

Accessing the AMCNTENSET0_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, AMCNTENSET0_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0010	0b101

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && AMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HAFGRTR_EL2.AMCNTEN0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMCNTENSET0_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HAFGRTR_EL2.AMCNTEN0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMCNTENSET0_EL0;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMCNTENSET0_EL0;
elsif PSTATE.EL == EL3 then
    return AMCNTENSET0_EL0;
```

MSR AMCNTENSET0_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0010	0b101

```
if IsHighestEL(PSTATE.EL) then
    AMCNTENSET0_EL0 = X[t];
else
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.5.7 AMCNTENSET1_EL0, Activity Monitors Count Enable Set Register 1

The AMCNTENSET1_EL0 characteristics are:

Purpose

Enable control bits for the auxiliary activity monitors event counters, AMEVCNTR1<n>_EL0.

Configurations

AArch64 System register AMCNTENSET1_EL0[31:0] is architecturally mapped to AArch32 System register AMCNTENSET1[31:0].

AArch64 System register AMCNTENSET1_EL0[31:0] is architecturally mapped to External register AMCNTENSET1[31:0].

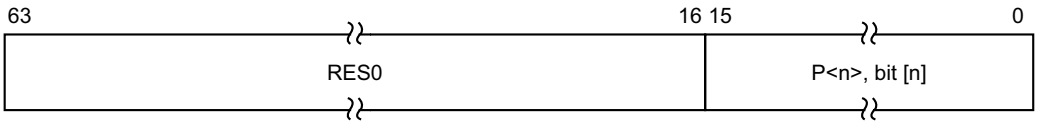
This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMCNTENSET1_EL0 are UNDEFINED.

Attributes

AMCNTENSET1_EL0 is a 64-bit register.

Field descriptions

The AMCNTENSET1_EL0 bit assignments are:



Bits [63:16]

Reserved, RES0.

P<n>, bit [n], for n = 0 to 15

Activity monitor event counter enable bit for AMEVCNTR1<n>_EL0.

Bits [31:16] are RES0. Bits [15:N] are RAZ/WI. N is the value in AMCGCR_EL0.CG1NC.

Possible values of each bit are:

`0b0` When read, means that AMEVCNTR1<n>_EL0 is disabled. When written, has no effect.

`0b1` When read, means that AMEVCNTR1<n>_EL0 is enabled. When written, enables AMEVCNTR1<n>_EL0.

On a Cold reset, this field resets to `0`.

Accessing the AMCNTENSET1_EL0

If the number of auxiliary activity monitor event counters implemented is zero, reads and writes of AMCNTENSET1_EL0 are UNDEFINED.

Note

The number of auxiliary activity monitor counters implemented is zero when AMCFGR_EL0.NCG == `0b0000`.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, AMCNTENSET1_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0011	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && AMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HAFGRTR_EL2.AMCNTEN1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMCNTENSET1_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HAFGRTR_EL2.AMCNTEN1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMCNTENSET1_EL0;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMCNTENSET1_EL0;
elsif PSTATE.EL == EL3 then
    return AMCNTENSET1_EL0;
```

MSR AMCNTENSET1_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0011	0b001

```
if IsHighestEL(PSTATE.EL) then
    AMCNTENSET1_EL0 = X[t];
else
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.5.8 AMCR_EL0, Activity Monitors Control Register

The AMCR_EL0 characteristics are:

Purpose

Global control register for the activity monitors implementation. AMCR_EL0 is applicable to both the architected and the auxiliary counter groups.

Configurations

AArch64 System register AMCR_EL0[31:0] is architecturally mapped to AArch32 System register AMCR[31:0].

AArch64 System register AMCR_EL0[31:0] is architecturally mapped to External register AMCR[31:0].

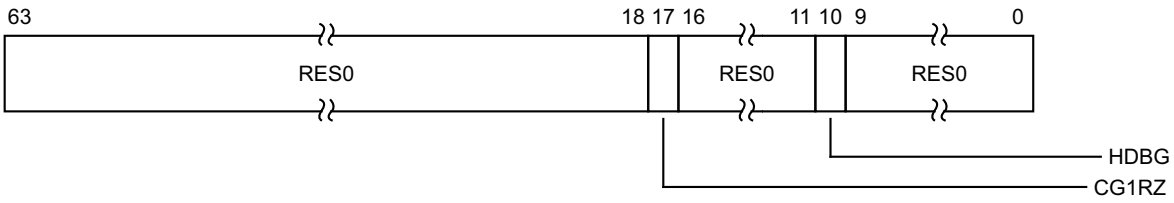
This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMCR_EL0 are UNDEFINED.

Attributes

AMCR_EL0 is a 64-bit register.

Field descriptions

The AMCR_EL0 bit assignments are:



Bits [63:18]

Reserved, RES0.

CG1RZ, bit [17]

When ARMv8.6-AMU is implemented:

Counter Group 1 Read Zero.

`0b0` System register reads of AMEVCNTR1<n>_EL0 return the event count at all implemented and enabled Exception levels.

`0b1` If the current Exception level is the highest implemented Exception level, system register reads of AMEVCNTR1<n>_EL0 return the event count. Otherwise, reads of AMEVCNTR1<n>_EL0 return a zero value.

Note

Reads from the memory-mapped view are unaffected by this field.

Otherwise:

Reserved, RES0.

Bits [16:11]

Reserved, RES0.

HDBG, bit [10]

This bit controls whether activity monitor counting is halted when the PE is halted in Debug state.

`0b0` Activity monitors do not halt counting when the PE is halted in Debug state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` Activity monitors halt counting when the PE is halted in Debug state.

Bits [9:0]

Reserved, RES0.

Accessing the AMCR_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, AMCR_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0010	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && AMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMCR_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMCR_EL0;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMCR_EL0;
elsif PSTATE.EL == EL3 then
    return AMCR_EL0;
```

MSR AMCR_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0010	0b000

```
if IsHighestEL(PSTATE.EL) then
    AMCR_EL0 = X[t];
else
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.5.9 AMEVCNTR0<n>_EL0, Activity Monitors Event Counter Registers 0, n = 0 - 15

The AMEVCNTR0<n>_EL0 characteristics are:

Purpose

Provides access to the architected activity monitor event counters.

Configurations

AArch64 System register AMEVCNTR0<n>_EL0[63:0] is architecturally mapped to AArch32 System register AMEVCNTR0<n>[63:0].

AArch64 System register AMEVCNTR0<n>_EL0[63:0] is architecturally mapped to External register AMEVCNTR0<n>[63:0].

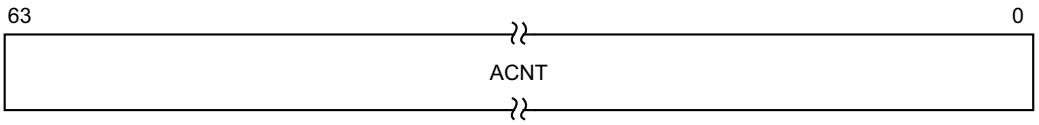
This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMEVCNTR0<n>_EL0 are UNDEFINED.

Attributes

AMEVCNTR0<n>_EL0 is a 64-bit register.

Field descriptions

The AMEVCNTR0<n>_EL0 bit assignments are:



ACNT, bits [63:0]

Architected activity monitor event counter n.

Value of architected activity monitor event counter n, where n is the number of this register and is a number from 0 to 15.

If ARMv8.6-AMU is implemented, HCR_EL2.AMVOFFEN is 1, SCR_EL3.AMVOFFEN is 1, HCR_EL2.{E2H, TGE} is not {1,1}, and EL2 is implemented in the current Security state, access to these registers at EL0 or EL1 return (PCount<63:0> - AMEVCNTVOFF0<n>_EL2<63:0>).

PCount is the physical count returned when AMEVCNTR0<n>_EL0 is read from EL2 or EL3.

If the counter is enabled, writes to this register have UNPREDICTABLE results.

On a Cold reset, this field resets to `0`.

Accessing the AMEVCNTR0<n>_EL0

If <n> is greater than or equal to the number of architected activity monitor event counters, reads and writes of AMEVCNTR0<n>_EL0 are UNDEFINED.

Note

AMCGCR_EL0.CG0NC identifies the number of architected activity monitor event counters.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, AMEVCNTR0<n>_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b010:n[3]	n[2:0]

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && AMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HAFGRTR_EL2.AMEVCNTR0<n>_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMEVCNTR0_EL0[UInt(CRm<0>:op2<2:0>)];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HAFGRTR_EL2.AMEVCNTR0<n>_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMEVCNTR0_EL0[UInt(CRm<0>:op2<2:0>)];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMEVCNTR0_EL0[UInt(CRm<0>:op2<2:0>)];
elsif PSTATE.EL == EL3 then
    return AMEVCNTR0_EL0[UInt(CRm<0>:op2<2:0>)];
```

MSR AMEVCNTR0<n>_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b010:n[3]	n[2:0]

```
if IsHighestEL(PSTATE.EL) then
    AMEVCNTR0_EL0[UInt(CRm<0>:op2<2:0>)] = X[t];
else
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.5.10 AMEVCNTR1<n>_EL0, Activity Monitors Event Counter Registers 1, n = 0 - 15

The AMEVCNTR1<n>_EL0 characteristics are:

Purpose

Provides access to the auxiliary activity monitor event counters.

Configurations

AArch64 System register AMEVCNTR1<n>_EL0[63:0] is architecturally mapped to AArch32 System register AMEVCNTR1<n>[63:0].

AArch64 System register AMEVCNTR1<n>_EL0[63:0] is architecturally mapped to External register AMEVCNTR1<n>[63:0].

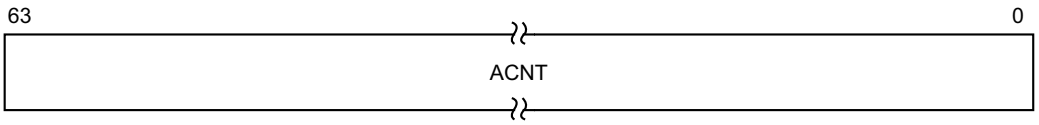
This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMEVCNTR1<n>_EL0 are UNDEFINED.

Attributes

AMEVCNTR1<n>_EL0 is a 64-bit register.

Field descriptions

The AMEVCNTR1<n>_EL0 bit assignments are:



ACNT, bits [63:0]

Auxiliary activity monitor event counter n.

Value of auxiliary activity monitor event counter n, where n is the number of this register and is a number from 0 to 15.

If ARMv8.6-AMU is implemented, HCR_EL2.AMVOFFEN is 1, SCR_EL3.AMVOFFEN is 1, HCR_EL2.{E2H, TGE} is not {1,1}, EL2 is implemented in the current Security state, and AMCR_EL0.CG1RZ is 0, reads to these registers at EL0 or EL1 return (PCount<63:0> - AMEVCNTVOFF1<n>_EL2<63:0>).

PCount is the physical count returned when AMEVCNTR1<n>_EL0 is read from EL2 or EL3.

If the counter is enabled, writes to this register have UNPREDICTABLE results.

On a Cold reset, this field resets to `0`.

Accessing the AMEVCNTR1<n>_EL0

If <n> is greater than or equal to the number of auxiliary activity monitor event counters, reads and writes of AMEVCNTR1<n>_EL0 are UNDEFINED.

Note

AMCGCR_EL0.CG1NC identifies the number of auxiliary activity monitor event counters.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, AMEVCNTR1<n>_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b110:n[3]	n[2:0]

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && AMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HAFGRTR_EL2.AMEVCNTR1<n>_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif !HighestELUsingAArch32() && AMCR_EL0.CG1RZ == '1' then
        Zeros();
    else
        return AMEVCNTR1_EL0[UInt(CRm<0>:op2<2:0>)];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HAFGRTR_EL2.AMEVCNTR1<n>_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif !IsHighestEL(PSTATE.EL) && !HighestELUsingAArch32() && AMCR_EL0.CG1RZ == '1' then
        Zeros();
    else
        return AMEVCNTR1_EL0[UInt(CRm<0>:op2<2:0>)];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif !IsHighestEL(PSTATE.EL) && !HighestELUsingAArch32() && AMCR_EL0.CG1RZ == '1' then
        Zeros();
    else
        return AMEVCNTR1_EL0[UInt(CRm<0>:op2<2:0>)];
elsif PSTATE.EL == EL3 then
    return AMEVCNTR1_EL0[UInt(CRm<0>:op2<2:0>)];
```

MSR AMEVCNTR1<n>_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b110:n[3]	n[2:0]

```
if IsHighestEL(PSTATE.EL) then
    AMEVCNTR1_EL0[UInt(CRm<0>:op2<2:0>)] = X[t];
else
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.5.11 AMEVCNTVOFF0<n>_EL2, Activity Monitors Event Counter Virtual Offset Registers 0, n = 0 - 15

The AMEVCNTVOFF0<n>_EL2 characteristics are:

Purpose

Holds the 64-bit virtual offset for architected activity monitor events.

Configurations

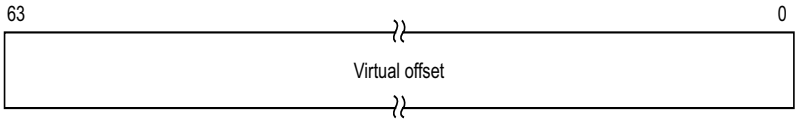
This register is present only when ARMv8.6-AMU is implemented. Otherwise, direct accesses to AMEVCNTVOFF0<n>_EL2 are UNDEFINED.

Attributes

AMEVCNTVOFF0<n>_EL2 is a 64-bit register.

Field descriptions

The AMEVCNTVOFF0<n>_EL2 bit assignments are:



Bits [63:0]

Virtual offset.

This field resets to an architecturally UNKNOWN value.

Accessing the AMEVCNTVOFF0<n>_EL2

If <n> is not 0, 2 or 3, reads and writes of AMEVCNTVOFF0<n>_EL2 are UNDEFINED.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, AMEVCNTVOFF0<n>_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1101	0b100:n[3]	n[2:0]

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        return NVMem[0xA00+8*UInt(CRm<0>:op2<2:0>)];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.AMVOFFEN == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMEVCNTVOFF0_EL2[UInt(CRm<0>:op2<2:0>)];
elsif PSTATE.EL == EL3 then
    return AMEVCNTVOFF0_EL2[UInt(CRm<0>:op2<2:0>)];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR AMEVCNTVOFF0<n>_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1101	0b100:n[3]	n[2:0]

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NVMem[0xA00+8*UInt(CRm<0>:op2<2:0>)] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AMEVCNTVOFF0_EL2[UInt(CRm<0>:op2<2:0>)] = X[t];
elsif PSTATE.EL == EL3 then
    AMEVCNTVOFF0_EL2[UInt(CRm<0>:op2<2:0>)] = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.5.12 AMEVCNTVOFF1<n>_EL2, Activity Monitors Event Counter Virtual Offset Registers 1, n = 0 - 15

The AMEVCNTVOFF1<n>_EL2 characteristics are:

Purpose

Holds the 64-bit virtual offset for auxiliary activity monitor events.

Configurations

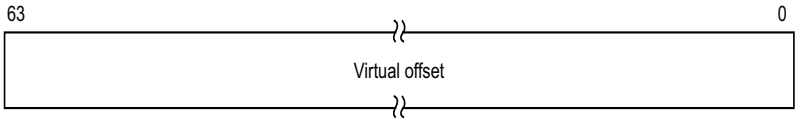
This register is present only when ARMv8.6-AMU is implemented. Otherwise, direct accesses to AMEVCNTVOFF1<n>_EL2 are UNDEFINED.

Attributes

AMEVCNTVOFF1<n>_EL2 is a 64-bit register.

Field descriptions

The AMEVCNTVOFF1<n>_EL2 bit assignments are:



Bits [63:0]

Virtual offset.

This field resets to an architecturally UNKNOWN value.

Accessing the AMEVCNTVOFF1<n>_EL2

Note

AMCG1IDR_EL0 identifies which auxiliary activity monitor event counters have a corresponding virtual offset implemented.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, AMEVCNTVOFF1<n>_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1101	0b101:n[3]	n[2:0]

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        return NVMem[0xA80+8*UInt(CRm<0>:op2<2:0>)];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.AMVOFFEN == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        return AMEVCNTVOFF1_EL2[UInt(CRm<0>:op2<2:0>)];
elsif PSTATE.EL == EL3 then
    return AMEVCNTVOFF1_EL2[UInt(CRm<0>:op2<2:0>)];
```

MSR AMEVCNTVOFF1<n>_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1101	0b101:n[3]	n[2:0]

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NVMem[0xA80+8*UInt(CRm<0>:op2<2:0>)] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AMEVCNTVOFF1_EL2[UInt(CRm<0>:op2<2:0>)] = X[t];
elsif PSTATE.EL == EL3 then
    AMEVCNTVOFF1_EL2[UInt(CRm<0>:op2<2:0>)] = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.5.13 AMEVTYPER0<n>_EL0, Activity Monitors Event Type Registers 0, n = 0 - 15

The AMEVTYPER0<n>_EL0 characteristics are:

Purpose

Provides information on the events that an architected activity monitor event counter AMEVCNTR0<n>_EL0 counts.

Configurations

AArch64 System register AMEVTYPER0<n>_EL0[31:0] is architecturally mapped to AArch32 System register AMEVTYPER0<n>[31:0].

AArch64 System register AMEVTYPER0<n>_EL0[31:0] is architecturally mapped to External register AMEVTYPER0<n>[31:0].

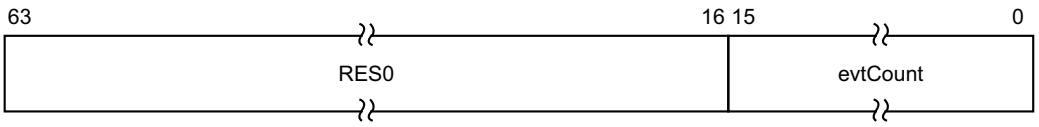
This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMEVTYPER0<n>_EL0 are UNDEFINED.

Attributes

AMEVTYPER0<n>_EL0 is a 64-bit register.

Field descriptions

The AMEVTYPER0<n>_EL0 bit assignments are:



Bits [63:16]

Reserved, RES0.

evtCount, bits [15:0]

Event to count. The event number of the event that is counted by the architected activity monitor event counter AMEVCNTR0<n>_EL0. The value of this field is architecturally mandated for each architected counter.

The following table shows the mapping between required event numbers and the corresponding counters:

`0x0011` **When n == 0** Processor frequency cycles

`0x4004` **When n == 1** Constant frequency cycles

`0x0008` **When n == 2** Instructions retired

`0x4005` **When n == 3** Memory stall cycles

Accessing the AMEVTYPER0<n>_EL0

If <n> is greater than or equal to the number of architected activity monitor event counters, reads and writes of AMEVTYPER0<n>_EL0 are UNDEFINED.

Note

AMCGCR_EL0.CG0NC identifies the number of architected activity monitor event counters.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, AMEVTYPER0<n>_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b011:n[3]	n[2:0]

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && AMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMEVTYPER0_EL0[UInt(CRm<0>:op2<2:0>)];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMEVTYPER0_EL0[UInt(CRm<0>:op2<2:0>)];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMEVTYPER0_EL0[UInt(CRm<0>:op2<2:0>)];
elsif PSTATE.EL == EL3 then
    return AMEVTYPER0_EL0[UInt(CRm<0>:op2<2:0>)];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.5.14 AMEVTYPER1<n>_EL0, Activity Monitors Event Type Registers 1, n = 0 - 15

The AMEVTYPER1<n>_EL0 characteristics are:

Purpose

Provides information on the events that an auxiliary activity monitor event counter AMEVCNTR1<n>_EL0 counts.

Configurations

AArch64 System register AMEVTYPER1<n>_EL0[31:0] is architecturally mapped to AArch32 System register AMEVTYPER1<n>[31:0].

AArch64 System register AMEVTYPER1<n>_EL0[31:0] is architecturally mapped to External register AMEVTYPER1<n>[31:0].

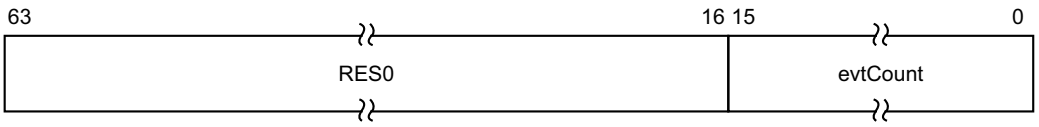
This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMEVTYPER1<n>_EL0 are UNDEFINED.

Attributes

AMEVTYPER1<n>_EL0 is a 64-bit register.

Field descriptions

The AMEVTYPER1<n>_EL0 bit assignments are:



Bits [63:16]

Reserved, RES0.

evtCount, bits [15:0]

Event to count. The event number of the event that is counted by the auxiliary activity monitor event counter AMEVCNTR1<n>_EL0.

It is IMPLEMENTATION DEFINED what values are supported by each counter.

If software writes a value to this field which is not supported by the corresponding counter AMEVCNTR1<n>_EL0, then:

- It is UNPREDICTABLE which event will be counted.
- The value read back is UNKNOWN.

The event counted by AMEVCNTR1<n>_EL0 might be fixed at implementation. In this case, the field is read-only and writes are UNDEFINED.

If the corresponding counter AMEVCNTR1<n>_EL0 is enabled, writes to this register have UNPREDICTABLE results.

Accessing the AMEVTYPER1<n>_EL0

If <n> is greater than or equal to the number of auxiliary activity monitor event counters, reads and writes of AMEVTYPER1<n>_EL0 are UNDEFINED.

Note

AMCGCR_EL0.CG1NC identifies the number of auxiliary activity monitor event counters.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, AMEVTYPER1<n>_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b111:n[3]	n[2:0]

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && AMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HAFGRTR_EL2.AMEVTYPER1<n>_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMEVTYPER1_EL0[UInt(CRm<0>:op2<2:0>)];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HAFGRTR_EL2.AMEVTYPER1<n>_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMEVTYPER1_EL0[UInt(CRm<0>:op2<2:0>)];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMEVTYPER1_EL0[UInt(CRm<0>:op2<2:0>)];
elsif PSTATE.EL == EL3 then
    return AMEVTYPER1_EL0[UInt(CRm<0>:op2<2:0>)];
```

MSR AMEVTYPER1<n>_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b111:n[3]	n[2:0]

```
if IsHighestEL(PSTATE.EL) then
    AMEVTYPER1_EL0[UInt(CRm<0>:op2<2:0>)] = X[t];
else
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.5.15 AMUSERENR_EL0, Activity Monitors User Enable Register

The AMUSERENR_EL0 characteristics are:

Purpose

Global user enable register for the activity monitors. Enables or disables EL0 access to the activity monitors. AMUSERENR_EL0 is applicable to both the architected and the auxiliary counter groups.

Configurations

AArch64 System register AMUSERENR_EL0[31:0] is architecturally mapped to AArch32 System register AMUSERENR[31:0].

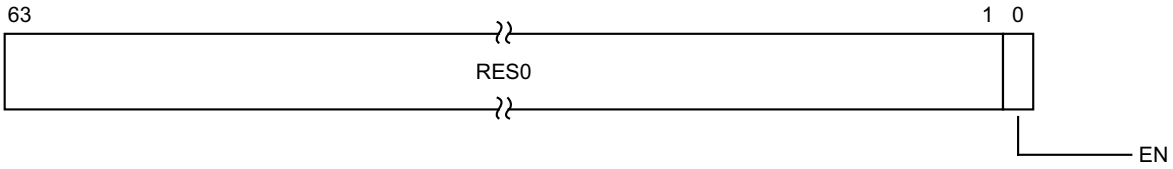
This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMUSERENR_EL0 are UNDEFINED.

Attributes

AMUSERENR_EL0 is a 64-bit register.

Field descriptions

The AMUSERENR_EL0 bit assignments are:



Bits [63:1]

Reserved, RES0.

EN, bit [0]

Traps EL0 accesses to the activity monitors registers to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, as follows:

- In AArch64 state, accesses to the following registers are trapped, reported using EC syndrome value `0x18`:
  - AMCFGR_EL0, AMCGCR_EL0, AMCNTENCLR0_EL0, AMCNTENCLR1_EL0, AMCNTENSET0_EL0, AMCNTENSET1_EL0, AMCR_EL0, AMEVCNTR0<n>_EL0, AMEVCNTR1<n>_EL0, AMEVTYPER0<n>_EL0, and AMEVTYPER1<n>_EL0.
- In AArch32 state, MRC and MCR accesses to the following registers are trapped and reported using EC syndrome value `0x03`, MRRC and MCRR accesses are trapped and reported using EC syndrome value `0x04`:
  - AMCFGR, AMCGCR, AMCNTENCLR0, AMCNTENCLR1, AMCNTENSET0, AMCNTENSET1, AMCR, AMEVCNTR0<n>, AMEVCNTR1<n>, AMEVTYPER0<n>, and AMEVTYPER1<n>.

`0b0` EL0 accesses to the activity monitors registers are trapped.

`0b1` This control does not cause any instructions to be trapped. Software can access all activity monitor registers at EL0.

Note

- AMUSERENR_EL0 can always be read at EL0 and is not governed by this bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the AMUSERENR_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, AMUSERENR_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0010	0b011

```
if PSTATE.EL == EL0 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMUSERENR_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMUSERENR_EL0;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return AMUSERENR_EL0;
elsif PSTATE.EL == EL3 then
    return AMUSERENR_EL0;
```

MSR AMUSERENR_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0010	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        AMUSERENR_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        AMUSERENR_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    AMUSERENR_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.6 Statistical Profiling Extension registers

This section lists the Statistical Profiling Extension registers in AArch64.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.6.1 PMBIDR_EL1, Profiling Buffer ID Register

The PMBIDR_EL1 characteristics are:

Purpose

Provides information to software as to whether the buffer can be programmed at the current Exception level.

Configurations

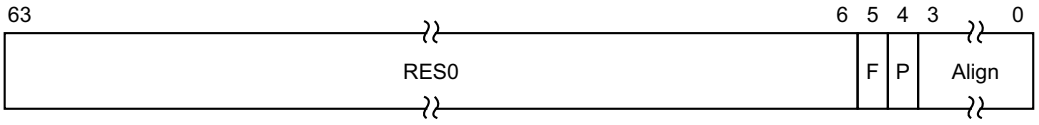
This register is present only when SPE is implemented. Otherwise, direct accesses to PMBIDR_EL1 are UNDEFINED.

Attributes

PMBIDR_EL1 is a 64-bit register.

Field descriptions

The PMBIDR_EL1 bit assignments are:



Bits [63:6]

Reserved, RES0.

F, bit [5]

Flag updates. Defines whether the address translation performed by the Profiling Buffer manages the Access Flag and dirty state.

`0b0` Hardware management of the Access Flag and dirty state for accesses made by the Statistical Profiling Extension is always disabled for all translation stages.

`0b1` Hardware management for the Access Flag and dirty state for accesses made by the Statistical Profiling Extension is controlled in the same way as explicit memory accesses in the owning translation regime.

If hardware management of the Access Flag is disabled for a stage of translation, an access to Page or Block with the Access flag bit not set in the descriptor will generate an Access Flag fault.

If hardware management of the dirty state is disabled for a stage of translation, an access to a Page or Block will ignore the Dirty Bit Modifier in the descriptor might generate a Permission fault, depending on the values of the access permission bits in the descriptor.

P, bit [4]

Programming not allowed. The Profiling Buffer is owned by a higher Exception level or the other Security state.

`0b0` Profiling Buffer is owned by the current or a lower Exception level in the current Security state.

`0b1` Profiling Buffer is owned by a higher Exception level or the other Security state.

The value read from this field depends on the current Exception level and the Effective values of MDCR_EL3.NSPB and MDCR_EL2.E2PB:

- If EL3 is implemented, and either MDCR_EL3.NSPB == `0b00` or MDCR_EL3.NSPB == `0b01`, this bit reads as one from:
  - Non-secure EL1.
  - Non-secure EL2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— If Secure EL2 is implemented and enabled, and MDCR_EL2.E2PB == `0b00`, Secure EL1.

- If EL3 is implemented, and either MDCR_EL3.NSPB == `0b10` or MDCR_EL3.NSPB == `0b11`, this bit reads as one from:
  — Secure EL1.
  — If Secure EL2 is implemented, Secure EL2.
  — If EL2 is implemented and MDCR_EL2.E2PB == `0b00`, Non-secure EL1.
- If EL3 is not implemented, EL2 is implemented, and MDCR_EL2.E2PB == `0b00`, this bit reads as one from EL1.
- Otherwise, this bit reads as zero.

Align, bits [3:0]

Defines the minimum alignment constraint for PMBPTR_EL1. If this field is non-zero, then the PE must pad every record up to a multiple of this size.

`0b0000` Byte

`0b0001` Halfword.

`0b0010` Word.

`0b0011` Doubleword.

`0b0100` 16 Bytes.

`0b0101` 32 Bytes.

`0b0110` 64 Bytes.

`0b0111` 128 Bytes.

`0b1000` 256 Bytes.

`0b1001` 512 Bytes.

`0b1010` 1KB.

`0b1011` 2KB.

For more information, see *Restrictions on the current write pointer* on page D9-2799.

Accessing the PMBIDR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PMBIDR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1010	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    return PMBIDR_EL1;
elsif PSTATE.EL == EL2 then
    return PMBIDR_EL1;
elsif PSTATE.EL == EL3 then
    return PMBIDR_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.6.2 PMBLIMITR_EL1, Profiling Buffer Limit Address Register

The PMBLIMITR_EL1 characteristics are:

Purpose

Defines the upper limit for the profiling buffer, and enables the profiling buffer

Configurations

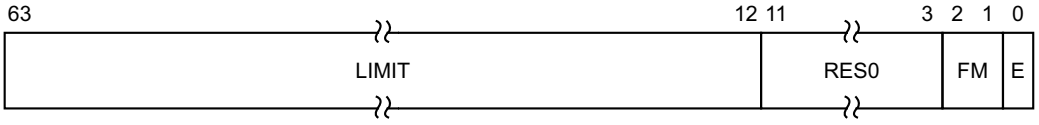
This register is present only when SPE is implemented. Otherwise, direct accesses to PMBLIMITR_EL1 are UNDEFINED.

Attributes

PMBLIMITR_EL1 is a 64-bit register.

Field descriptions

The PMBLIMITR_EL1 bit assignments are:



LIMIT, bits [63:12]

Limit address. PMBLIMITR_EL1.LIMIT:Zeros(12) is the address of the first byte in memory after the last byte in the profiling buffer. If the smallest implemented translation granule is not 4KB, then bits[N-1:12] are RES0, where N is the IMPLEMENTATION DEFINED value, Log_2(smallest implemented translation granule).

This field resets to an architecturally UNKNOWN value.

Bits [11:3]

Reserved, RES0.

FM, bits [2:1]

Fill mode

`0b00` Stop collection and raise maintenance interrupt on buffer fill.

All other values are reserved. If this field is programmed with a reserved value, the PE behaves as if this field has a defined value, other than for a direct read of the register. Software must not rely on the behavior of reserved values, as they might change in a future version of the architecture.

This field resets to an architecturally UNKNOWN value.

E, bit [0]

Profiling Buffer enable

`0b0` All output is discarded.

`0b1` Profiling buffer enabled.

On a Warm reset, this field resets to `0`.

Accessing the PMBLIMITR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, PMBLIMITR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.PMBLIMITR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.E2PB == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '1x1' then
        return NVMem[0x800];
    else
        return PMBLIMITR_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMBLIMITR_EL1;
elsif PSTATE.EL == EL3 then
    return PMBLIMITR_EL1;
```

MSR PMBLIMITR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.PMBLIMITR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.E2PB == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '1x1' then
        NVMem[0x800] = X[t];
    else
        PMBLIMITR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMBLIMITR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    PMBLIMITR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.6.3 PMBPTR_EL1, Profiling Buffer Write Pointer Register

The PMBPTR_EL1 characteristics are:

Purpose

Defines the current write pointer for the profiling buffer.

Configurations

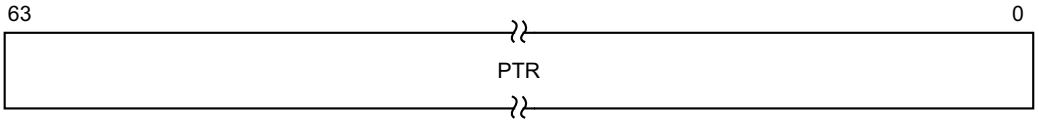
This register is present only when SPE is implemented. Otherwise, direct accesses to PMBPTR_EL1 are UNDEFINED.

Attributes

PMBPTR_EL1 is a 64-bit register.

Field descriptions

The PMBPTR_EL1 bit assignments are:



PTR, bits [63:0]

Current write address. Defines the virtual address of the next entry to be written to the buffer.

The architecture places restrictions on the values software can write to the pointer. For more information see *Restrictions on the current write pointer* on page D9-2799.

Note

As a result, an implementation might treat some of bits[M:0], where M is defined by PMBIDR_EL1.Align, as RES0.

On a management interrupt, PMBPTR_EL1 is frozen.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMBPTR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PMBPTR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.PMBPTR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.E2PB == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '1x1' then
        return NVMem[0x810];
    else
        return PMBPTR_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMBPTR_EL1;
elsif PSTATE.EL == EL3 then
    return PMBPTR_EL1;
```

MSR PMBPTR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.PMBPTR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.E2PB == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '1x1' then
        NVMem[0x810] = X[t];
    else
        PMBPTR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMBPTR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    PMBPTR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.6.4 PMBSR_EL1, Profiling Buffer Status/syndrome Register

The PMBSR_EL1 characteristics are:

Purpose

Provides syndrome information to software when the buffer is disabled because the management interrupt has been raised.

Configurations

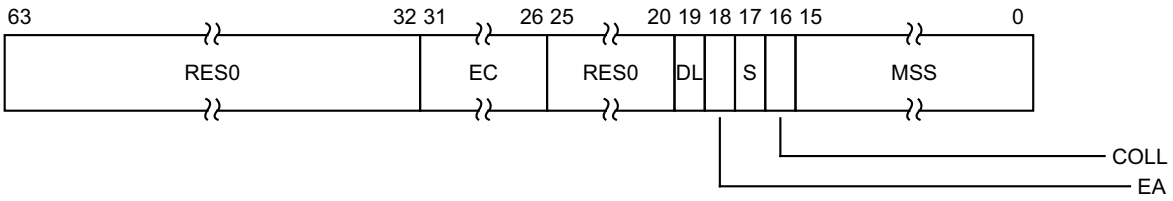
This register is present only when SPE is implemented. Otherwise, direct accesses to PMBSR_EL1 are UNDEFINED.

Attributes

PMBSR_EL1 is a 64-bit register.

Field descriptions

The PMBSR_EL1 bit assignments are:



Bits [63:32]

Reserved, RES0.

EC, bits [31:26]

Exception class

Top-level description of the cause of the buffer management event

EC == 0b000000

Other buffer management event. All buffer management events other than those described by other defined Exception class codes.

See MSS encoding for other buffer management events.

EC == 0b100100

Stage 1 Data Abort on write to Profiling Buffer.

See MSS encoding for stage 1 or stage 2 Data Aborts on write to buffer.

EC == 0b100101

Stage 2 Data Abort on write to Profiling Buffer.

See MSS encoding for stage 1 or stage 2 Data Aborts on write to buffer.

All other values are reserved. Reserved values might be defined in a future version of the architecture.

Writing a reserved value to this field will make the value of this field UNKNOWN. Values that are not supported act as reserved values when writing to this register.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [25:20]

Reserved, RES0.

DL, bit [19]

Partial record lost.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Following a buffer management event other than an asynchronous External abort, indicates whether the last record written to the Profiling Buffer is complete.

`0b0` PMBPTR_EL1 points to the first byte after the last complete record written to the Profiling Buffer.

`0b1` Part of a record was lost because of a buffer management event or synchronous External abort. PMBPTR_EL1 might not point to the first byte after the last complete record written to the buffer, and so restarting collection might result in a data record stream that software cannot parse. All records prior to the last record have been written to the buffer.

When the buffer management event was because of an asynchronous external abort, this bit is set to 1 and software must not assume that any valid data has been written to the Profiling Buffer.

This bit is RES0 if the PE never sets this bit as a result of a buffer management event caused by an asynchronous External abort.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

EA, bit [18]

External abort.

`0b0` An external abort has not been asserted.

`0b1` An external abort has been asserted and detected by the Statistical Profiling Extension.

This bit is RES0 if the PE never sets this bit as the result of an External abort.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

S, bit [17]

Service

`0b0` PMBIRQ is not asserted.

`0b1` PMBIRQ is asserted. All profiling data has either been written to the buffer or discarded.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

COLL, bit [16]

Collision detected.

`0b0` No collision events detected.

`0b1` At least one collision event was recorded.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

MSS, bits [15:0]

Management Event Specific Syndrome.

Contains syndrome specific to the management event.

The syndrome contents for each management event are described in the following sections.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

MSS encoding for stage 1 or stage 2 Data Aborts on write to buffer

15 … 6	5 … 0
RES0	FSC

Bits [15:6]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

FSC, bits [5:0]

Fault status code

`0b000000` Address size fault, level 0 of translation or translation table base register.

`0b000001` Address size fault, level 1.

`0b000010` Address size fault, level 2.

`0b000011` Address size fault, level 3.

`0b000100` Translation fault, level 0.

`0b000101` Translation fault, level 1.

`0b000110` Translation fault, level 2.

`0b000111` Translation fault, level 3.

`0b001001` Access flag fault, level 1.

`0b001010` Access flag fault, level 2.

`0b001011` Access flag fault, level 3.

`0b001101` Permission fault, level 1.

`0b001110` Permission fault, level 2.

`0b001111` Permission fault, level 3.

`0b010000` Synchronous External abort, not on translation table walk or hardware update of translation table.

`0b010001` Asynchronous External abort.

`0b010100` Synchronous External abort, on translation table walk or hardware update of translation table, level 0.

`0b010101` Synchronous External abort, on translation table walk or hardware update of translation table, level 1.

`0b010110` Synchronous External abort, on translation table walk or hardware update of translation table, level 2.

`0b010111` Synchronous External abort, on translation table walk or hardware update of translation table, level 3.

`0b100001` Alignment fault.

`0b110000` TLB conflict abort.

`0b110001` **When ARMv8.1-TTHM is implemented** Unsupported atomic hardware update fault.

All other values are reserved.

It is IMPLEMENTATION DEFINED whether each of the Access Flag fault, asynchronous External abort and synchronous External abort, Alignment fault, and TLB Conflict abort values can be generated by the PE. For more information see *Faults and watchpoints* on page D9-2804.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

MSS encoding for other buffer management events

15 … 6	5 … 0
RES0	BSC

Bits [15:6]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

BSC, bits [5:0]

Buffer status code

`0b000000` Buffer not filled

`0b000001` Buffer filled

All other values are reserved. Reserved values might be defined in a future version of the architecture.

Writing a reserved value to this field will make the value of this field UNKNOWN. Values that are not supported act as reserved values when writing to this register.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMBSR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PMBSR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1010	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.PMBSR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.E2PB == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '1x1' then
        return NVMem[0x820];
    else
        return PMBSR_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMBSR_EL1;
elsif PSTATE.EL == EL3 then
    return PMBSR_EL1;
```

MSR PMBSR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1010	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.PMBSR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.E2PB == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '1x1' then
        NVMem[0x820] = X[t];
    else
        PMBSR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMBSR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    PMBSR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.6.5 PMSCR_EL1, Statistical Profiling Control Register (EL1)

The PMSCR_EL1 characteristics are:

Purpose

Provides EL1 controls for Statistical Profiling

Configurations

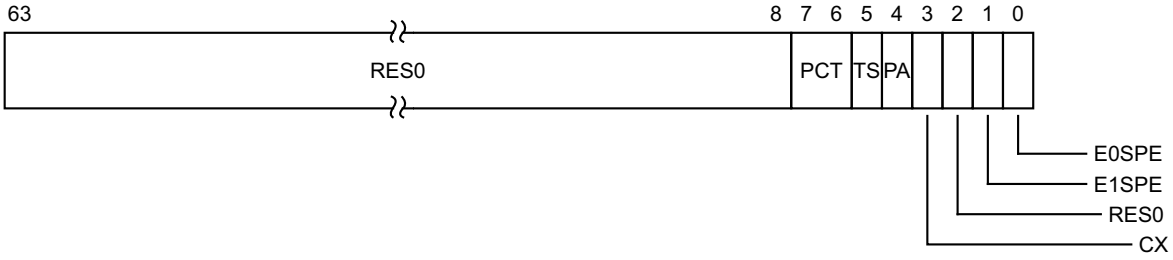
This register is present only when SPE is implemented. Otherwise, direct accesses to PMSCR_EL1 are UNDEFINED.

Attributes

PMSCR_EL1 is a 64-bit register.

Field descriptions

The PMSCR_EL1 bit assignments are:



Bits [63:8]

Reserved, RES0.

PCT, bits [7:6]

When EL2 is implemented:

Physical Timestamp. If timestamp sampling is enabled and the Profiling Buffer is owned by EL1, requests which timestamp counter value is collected.

If ARMv8.6-ECV is implemented, this is a two bit field as shown. Otherwise, bit[7] is RES0.

`0b00` Virtual counter, CNTVCT_EL0, is collected.

`0b01` Physical counter, CNTPCT_EL0, is collected.

`0b11` **When ARMv8.6-ECV is implemented** Physical counter, CNTPCT_EL0, minus CNTPOFF_EL2 is collected.

If the Profiling Buffer owning Exception level is EL2, this field is ignored.

If the Profiling Buffer owning Exception level is EL1, this field is combined with PMSCR_EL2.PCT to determine which timestamp counter value is collected. For more information, see *Controlling the data that is collected* on page D9-2795.

When EL2 is implemented and enabled in the current Security state, the physical counter uses a fixed physical offset of zero if either of the following are true:

- CNTHCTL_EL2.ECV is 0.
- SCR_EL3.ECVEn is 0.
- HCR_EL2.{E2H, TGE} is {1, 1}.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

Physical Timestamp. Reserved. This field reads as `0b01` and ignores writes. Software should treat this field as UNK/SBZP.

When EL2 is not implemented, the Effective values of CNTVOFF_EL2 and CNTPOFF_EL2 are zero, meaning the virtual counter and physical counter have the same value.

TS, bit [5]

Timestamp enable.

`0b0` Timestamp sampling disabled.

`0b1` Timestamp sampling enabled.

This bit is ignored by the PE if EL2 is implemented and the Profiling Buffer is owned by EL2. For more information, see *Controlling the data that is collected* on page D9-2795.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

PA, bit [4]

Physical Address sample enable.

`0b0` Physical addresses are not collected.

`0b1` Physical addresses are collected.

If EL2 is implemented:

- If the Profiling Buffer is owned by EL1, this bit is combined with PMSCR_EL2.PA to determine which address is collected. For more information, see *Controlling the data that is collected* on page D9-2795.
- If the Profiling Buffer is owned by EL2, this bit is ignored by the PE.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

CX, bit [3]

CONTEXTIDR_EL1 sample enable.

`0b0` CONTEXTIDR_EL1 is not collected.

`0b1` CONTEXTIDR_EL1 is collected.

If EL2 is implemented and enabled in the current Security state when an operation is sampled:

- If the PE is at EL2, this bit is ignored by the PE.
- If HCR_EL2.TGE == 1, this bit is ignored by the PE.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [2]

Reserved, RES0.

E1SPE, bit [1]

EL1 Statistical Profiling Enable.

`0b0` Sampling disabled at EL1.

`0b1` Sampling enabled at EL1.

If EL2 is implemented and enabled in the current Security state, this bit is ignored by the PE when HCR_EL2.TGE == 1.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

E0SPE, bit [0]

EL0 Statistical Profiling Enable. Controls sampling at EL0 when HCR_EL2.TGE == 0 or if EL2 is disabled or not implemented.

`0b0` Sampling disabled at EL0.

`0b1` Sampling enabled at EL0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If EL2 is implemented and enabled in the current Security state, this bit is ignored by the PE when HCR_EL2.TGE == 1.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMSCR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PMSCR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.PMSCR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPMS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x828];
    else
        return PMSCR_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        return PMSCR_EL2;
    else
        return PMSCR_EL1;
elsif PSTATE.EL == EL3 then
    return PMSCR_EL1;
```

MSR PMSCR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.PMSCR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPMS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        NVMem[0x828] = X[t];
    else
        PMSCR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        PMSCR_EL2 = X[t];
    else
        PMSCR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    PMSCR_EL1 = X[t];
```

MRS <Xt>, PMSCR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b1001	0b1001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        return NVMem[0x828];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
            AArch64.SystemAccessTrap(EL3, 0x18);
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            return PMSCR_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        return PMSCR_EL1;
    else
        UNDEFINED;
```

MSR PMSCR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b1001	0b1001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        NVMem[0x828] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        if HCR_EL2.E2H == '1' then
            if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
                AArch64.SystemAccessTrap(EL3, 0x18);
            elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
                AArch64.SystemAccessTrap(EL3, 0x18);
            else
                PMSCR_EL1 = X[t];
        else
            UNDEFINED;
    elsif PSTATE.EL == EL3 then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
            PMSCR_EL1 = X[t];
        else
            UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.6.6 PMSCR_EL2, Statistical Profiling Control Register (EL2)

The PMSCR_EL2 characteristics are:

Purpose

Provides EL2 controls for Statistical Profiling

Configurations

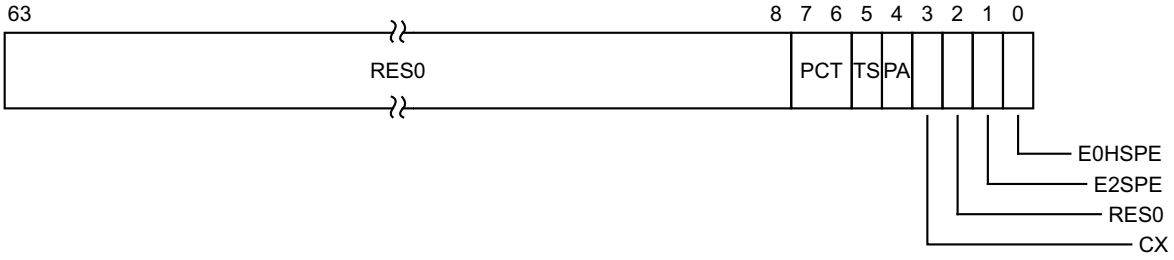
This register is present only when SPE is implemented. Otherwise, direct accesses to PMSCR_EL2 are UNDEFINED.

Attributes

PMSCR_EL2 is a 64-bit register.

Field descriptions

The PMSCR_EL2 bit assignments are:



Bits [63:8]

Reserved, RES0.

PCT, bits [7:6]

Physical Timestamp. If timestamp sampling is enabled, determines which counter is collected. The behavior depends on the Profiling Buffer owning Exception level.

If ARMv8.6-ECV is implemented, this is a two bit field as shown. Otherwise, bit[7] is RES0.

`0b00` Virtual counter, CNTVCT_EL0, is collected.

`0b01` If the Profiling Buffer owning Exception level is EL2: Physical counter, CNTPCT_EL0, is collected.

If the Profiling Buffer owning Exception level is EL1: Timestamp value is selected by PMSCR_EL1.PCT.

`0b11` **When ARMv8.6-ECV is implemented** If the Profiling Buffer owning Exception level is EL2: Physical counter, CNTPCT_EL0, minus CNTPOFF_EL2 is collected.

If the Profiling Buffer owning Exception level is EL1: If PMSCR_EL1.PCT == `0b00`, the virtual counter, CNTVCT_EL0 is collected. Otherwise, the physical counter, CNTPCT_EL0, minus CNTPOFF_EL2 is collected.

When EL2 is implemented and enabled in the current Security state, the physical counter uses a fixed physical offset of zero if either of the following are true:

- CNTHCTL_EL2.ECV is 0.
- SCR_EL3.ECVEn is 0.
- HCR_EL2.{E2H, TGE} is {1, 1}.

If EL2 is not implemented or EL2 is disabled in the current Security state, the PE behaves as if this field is set to `0b01`, other than for a direct read of the register.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TS, bit [5]

Timestamp Enable.

`0b0` Timestamp sampling disabled.

`0b1` Timestamp sampling enabled.

This bit is ignored by the PE when any of the following are true:

- The Profiling Buffer owning Exception level is EL1.
- In Secure state, and either ARMv8.4-SecEL2 is not implemented or Secure EL2 is disabled.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

PA, bit [4]

Physical Address Sample Enable.

`0b0` Physical addresses are not collected.

`0b1` Physical addresses are collected.

If the Profiling Buffer owning Exception level is EL1, and EL2 is enabled in the current Security state, this bit is combined with PMSCR_EL1.PA to determine which address is collected.

If EL2 is not implemented or EL2 is disabled in the current Security state, the PE ignores the value of this bit and behaves as if this bit is set to 1, other than for a direct read of the register.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

CX, bit [3]

CONTEXTIDR_EL2 Sample Enable.

`0b0` CONTEXTIDR_EL2 is not collected.

`0b1` CONTEXTIDR_EL2 is collected.

If EL2 is not implemented or EL2 is disabled in the current Security state, the PE ignores the value of this bit.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [2]

Reserved, RES0.

E2SPE, bit [1]

EL2 Statistical Profiling Enable.

`0b0` Sampling disabled at EL2.

`0b1` Sampling enabled at EL2.

This bit is RES0 if MDCR_EL2.E2PB != `0b00`.

If EL2 is disabled in the current Security state, this bit is ignored by the PE.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

E0HSPE, bit [0]

EL0 Statistical Profiling Enable.

`0b0` Sampling disabled at EL0.

`0b1` Sampling enabled at EL0.

If MDCR_EL2.E2PB != `0b00`, this bit is RES0.

If EL2 is implemented and enabled in the current Security state, this bit is ignored by the PE when HCR_EL2.TGE == 0.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the PMSCR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PMSCR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1001	0b1001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMSCR_EL2;
elsif PSTATE.EL == EL3 then
    return PMSCR_EL2;
```

MSR PMSCR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1001	0b1001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMSCR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    PMSCR_EL2 = X[t];
```

MRS <Xt>, PMSCR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.PMSCR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPMS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x828];
    else
        return PMSCR_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        return PMSCR_EL2;
    else
        return PMSCR_EL1;
elsif PSTATE.EL == EL3 then
    return PMSCR_EL1;
```

MSR PMSCR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.PMSCR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPMS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x828] = X[t];
    else
        PMSCR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        PMSCR_EL2 = X[t];
    else
        PMSCR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    PMSCR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.6.7 PMSEVFR_EL1, Sampling Event Filter Register

The PMSEVFR_EL1 characteristics are:

Purpose

Controls sample filtering by events. The overall filter is the logical AND of these filters. For example, if E[3] and E[5] are both set to 1, only samples that have both event 3 (Level 1 unified or data cache refill) and event 5 set (TLB walk) are recorded

Configurations

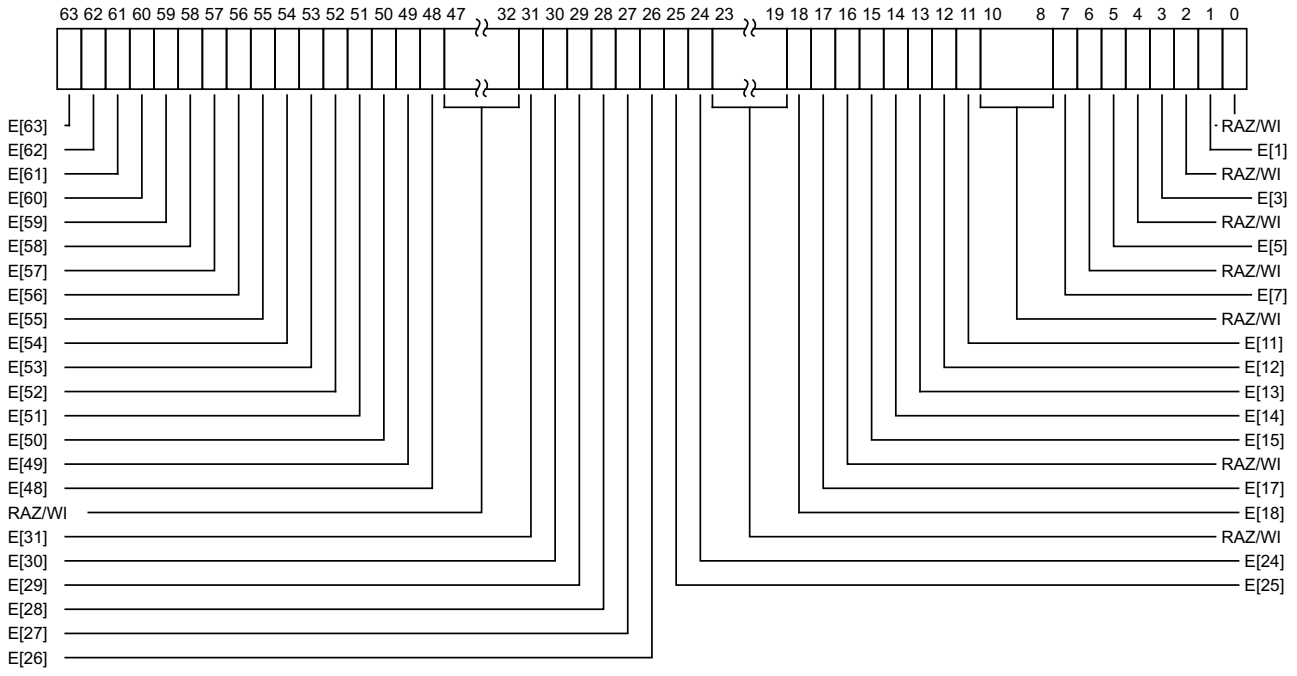
This register is present only when SPE is implemented. Otherwise, direct accesses to PMSEVFR_EL1 are UNDEFINED.

Attributes

PMSEVFR_EL1 is a 64-bit register.

Field descriptions

The PMSEVFR_EL1 bit assignments are:



E[63], bit [63]

E[63] is the event filter for event 63. If event 63 is not implemented, or filtering on event 63 is not supported, the corresponding bit is RAZ/WI.

`0b0` Event 63 is ignored.

`0b1` Do not record samples that have event 63 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, if the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

This field is ignored by the PE when PMSFCR_EL1.FE == 0

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

E[62], bit [62]

E[62] is the event filter for event 62. If event 62 is not implemented, or filtering on event 62 is not supported, the corresponding bit is RAZ/WI.

`0b0` Event 62 is ignored.

`0b1` Do not record samples that have event 62 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, if the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

This field is ignored by the PE when PMSFCR_EL1.FE == 0

On a Warm reset, this field resets to an architecturally UNKNOWN value.

E[61], bit [61]

E[61] is the event filter for event 61. If event 61 is not implemented, or filtering on event 61 is not supported, the corresponding bit is RAZ/WI.

`0b0` Event 61 is ignored.

`0b1` Do not record samples that have event 61 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, if the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

This field is ignored by the PE when PMSFCR_EL1.FE == 0

On a Warm reset, this field resets to an architecturally UNKNOWN value.

E[60], bit [60]

E[60] is the event filter for event 60. If event 60 is not implemented, or filtering on event 60 is not supported, the corresponding bit is RAZ/WI.

`0b0` Event 60 is ignored.

`0b1` Do not record samples that have event 60 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, if the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

This field is ignored by the PE when PMSFCR_EL1.FE == 0

On a Warm reset, this field resets to an architecturally UNKNOWN value.

E[59], bit [59]

E[59] is the event filter for event 59. If event 59 is not implemented, or filtering on event 59 is not supported, the corresponding bit is RAZ/WI.

`0b0` Event 59 is ignored.

`0b1` Do not record samples that have event 59 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, if the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

This field is ignored by the PE when PMSFCR_EL1.FE == 0

On a Warm reset, this field resets to an architecturally UNKNOWN value.

E[58], bit [58]

E[58] is the event filter for event 58. If event 58 is not implemented, or filtering on event 58 is not supported, the corresponding bit is RAZ/WI.

`0b0` Event 58 is ignored.

`0b1` Do not record samples that have event 58 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, if the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

This field is ignored by the PE when PMSFCR_EL1.FE == 0

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

E[57], bit [57]

E[57] is the event filter for event 57. If event 57 is not implemented, or filtering on event 57 is not supported, the corresponding bit is RAZ/WI.

`0b0` Event 57 is ignored.

`0b1` Do not record samples that have event 57 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, if the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

This field is ignored by the PE when PMSFCR_EL1.FE == 0

On a Warm reset, this field resets to an architecturally UNKNOWN value.

E[56], bit [56]

E[56] is the event filter for event 56. If event 56 is not implemented, or filtering on event 56 is not supported, the corresponding bit is RAZ/WI.

`0b0` Event 56 is ignored.

`0b1` Do not record samples that have event 56 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, if the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

This field is ignored by the PE when PMSFCR_EL1.FE == 0

On a Warm reset, this field resets to an architecturally UNKNOWN value.

E[55], bit [55]

E[55] is the event filter for event 55. If event 55 is not implemented, or filtering on event 55 is not supported, the corresponding bit is RAZ/WI.

`0b0` Event 55 is ignored.

`0b1` Do not record samples that have event 55 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, if the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

This field is ignored by the PE when PMSFCR_EL1.FE == 0

On a Warm reset, this field resets to an architecturally UNKNOWN value.

E[54], bit [54]

E[54] is the event filter for event 54. If event 54 is not implemented, or filtering on event 54 is not supported, the corresponding bit is RAZ/WI.

`0b0` Event 54 is ignored.

`0b1` Do not record samples that have event 54 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, if the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

This field is ignored by the PE when PMSFCR_EL1.FE == 0

On a Warm reset, this field resets to an architecturally UNKNOWN value.

E[53], bit [53]

E[53] is the event filter for event 53. If event 53 is not implemented, or filtering on event 53 is not supported, the corresponding bit is RAZ/WI.

`0b0` Event 53 is ignored.

`0b1` Do not record samples that have event 53 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, if the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

This field is ignored by the PE when PMSFCR_EL1.FE == 0

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

E[52], bit [52]

E[52] is the event filter for event 52. If event 52 is not implemented, or filtering on event 52 is not supported, the corresponding bit is RAZ/WI.

`0b0` Event 52 is ignored.

`0b1` Do not record samples that have event 52 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, if the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

This field is ignored by the PE when PMSFCR_EL1.FE == 0

On a Warm reset, this field resets to an architecturally UNKNOWN value.

E[51], bit [51]

E[51] is the event filter for event 51. If event 51 is not implemented, or filtering on event 51 is not supported, the corresponding bit is RAZ/WI.

`0b0` Event 51 is ignored.

`0b1` Do not record samples that have event 51 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, if the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

This field is ignored by the PE when PMSFCR_EL1.FE == 0

On a Warm reset, this field resets to an architecturally UNKNOWN value.

E[50], bit [50]

E[50] is the event filter for event 50. If event 50 is not implemented, or filtering on event 50 is not supported, the corresponding bit is RAZ/WI.

`0b0` Event 50 is ignored.

`0b1` Do not record samples that have event 50 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, if the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

This field is ignored by the PE when PMSFCR_EL1.FE == 0

On a Warm reset, this field resets to an architecturally UNKNOWN value.

E[49], bit [49]

E[49] is the event filter for event 49. If event 49 is not implemented, or filtering on event 49 is not supported, the corresponding bit is RAZ/WI.

`0b0` Event 49 is ignored.

`0b1` Do not record samples that have event 49 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, if the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

This field is ignored by the PE when PMSFCR_EL1.FE == 0

On a Warm reset, this field resets to an architecturally UNKNOWN value.

E[48], bit [48]

E[48] is the event filter for event 48. If event 48 is not implemented, or filtering on event 48 is not supported, the corresponding bit is RAZ/WI.

`0b0` Event 48 is ignored.

`0b1` Do not record samples that have event 48 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, if the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

This field is ignored by the PE when PMSFCR_EL1.FE == 0

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bits [47:32]

Reserved, RAZ/WI.

E[31], bit [31]

E[31] is the event filter for event 31. If event 31 is not implemented, or filtering on event 31 is not supported, the corresponding bit is RAZ/WI.

`0b0` Event 31 is ignored.

`0b1` Do not record samples that have event 31 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, if the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

This field is ignored by the PE when PMSFCR_EL1.FE == 0

On a Warm reset, this field resets to an architecturally UNKNOWN value.

E[30], bit [30]

E[30] is the event filter for event 30. If event 30 is not implemented, or filtering on event 30 is not supported, the corresponding bit is RAZ/WI.

`0b0` Event 30 is ignored.

`0b1` Do not record samples that have event 30 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, if the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

This field is ignored by the PE when PMSFCR_EL1.FE == 0

On a Warm reset, this field resets to an architecturally UNKNOWN value.

E[29], bit [29]

E[29] is the event filter for event 29. If event 29 is not implemented, or filtering on event 29 is not supported, the corresponding bit is RAZ/WI.

`0b0` Event 29 is ignored.

`0b1` Do not record samples that have event 29 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, if the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

This field is ignored by the PE when PMSFCR_EL1.FE == 0

On a Warm reset, this field resets to an architecturally UNKNOWN value.

E[28], bit [28]

E[28] is the event filter for event 28. If event 28 is not implemented, or filtering on event 28 is not supported, the corresponding bit is RAZ/WI.

`0b0` Event 28 is ignored.

`0b1` Do not record samples that have event 28 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, if the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

This field is ignored by the PE when PMSFCR_EL1.FE == 0

On a Warm reset, this field resets to an architecturally UNKNOWN value.

E[27], bit [27]

E[27] is the event filter for event 27. If event 27 is not implemented, or filtering on event 27 is not supported, the corresponding bit is RAZ/WI.

`0b0` Event 27 is ignored.

`0b1` Do not record samples that have event 27 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, if the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field is ignored by the PE when PMSFCR_EL1.FE == 0

On a Warm reset, this field resets to an architecturally UNKNOWN value.

E[26], bit [26]

E[26] is the event filter for event 26. If event 26 is not implemented, or filtering on event 26 is not supported, the corresponding bit is RAZ/WI.

`0b0` Event 26 is ignored.

`0b1` Do not record samples that have event 26 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, if the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

This field is ignored by the PE when PMSFCR_EL1.FE == 0

On a Warm reset, this field resets to an architecturally UNKNOWN value.

E[25], bit [25]

E[25] is the event filter for event 25. If event 25 is not implemented, or filtering on event 25 is not supported, the corresponding bit is RAZ/WI.

`0b0` Event 25 is ignored.

`0b1` Do not record samples that have event 25 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, if the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

This field is ignored by the PE when PMSFCR_EL1.FE == 0

On a Warm reset, this field resets to an architecturally UNKNOWN value.

E[24], bit [24]

E[24] is the event filter for event 24. If event 24 is not implemented, or filtering on event 24 is not supported, the corresponding bit is RAZ/WI.

`0b0` Event 24 is ignored.

`0b1` Do not record samples that have event 24 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, if the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

This field is ignored by the PE when PMSFCR_EL1.FE == 0

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [23:19]

Reserved, RAZ/WI.

E[18], bit [18]

When ARMv8.3-SPE is implemented and SVE is implemented:

Empty predicate.

`0b0` Empty predicate event is ignored.

`0b1` Do not record samples that have the Empty predicate event == 0.

This bit is ignored by the PE when PMSFCR_EL1.FE == 0.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[17], bit [17]

When ARMv8.3-SPE is implemented and SVE is implemented:

Partial predicate.

`0b0` Partial predicate event is ignored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` Do not record samples that have the Partial predicate event == 0.

This bit is ignored by the PE when PMSFCR_EL1.FE == 0.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

Bit [16]

Reserved, RAZ/WI.

E[15], bit [15]

E[15] is the event filter for event 15. If event 15 is not implemented, or filtering on event 15 is not supported, the corresponding bit is RAZ/WI.

`0b0` Event 15 is ignored.

`0b1` Do not record samples that have event 15 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, if the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

This field is ignored by the PE when PMSFCR_EL1.FE == 0

On a Warm reset, this field resets to an architecturally UNKNOWN value.

E[14], bit [14]

E[14] is the event filter for event 14. If event 14 is not implemented, or filtering on event 14 is not supported, the corresponding bit is RAZ/WI.

`0b0` Event 14 is ignored.

`0b1` Do not record samples that have event 14 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, if the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

This field is ignored by the PE when PMSFCR_EL1.FE == 0

On a Warm reset, this field resets to an architecturally UNKNOWN value.

E[13], bit [13]

E[13] is the event filter for event 13. If event 13 is not implemented, or filtering on event 13 is not supported, the corresponding bit is RAZ/WI.

`0b0` Event 13 is ignored.

`0b1` Do not record samples that have event 13 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, if the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

This field is ignored by the PE when PMSFCR_EL1.FE == 0

On a Warm reset, this field resets to an architecturally UNKNOWN value.

E[12], bit [12]

E[12] is the event filter for event 12. If event 12 is not implemented, or filtering on event 12 is not supported, the corresponding bit is RAZ/WI.

`0b0` Event 12 is ignored.

`0b1` Do not record samples that have event 12 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, if the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

This field is ignored by the PE when PMSFCR_EL1.FE == 0

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

E[11], bit [11]

When ARMv8.3-SPE is implemented:

Alignment.

`0b0` Alignment event is ignored.

`0b1` Do not record samples that have the Alignment event == 0.

This bit is ignored by the PE when PMSFCR_EL1.FE == 0.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

Bits [10:8]

Reserved, RAZ/WI.

E[7], bit [7]

Mispredicted.

`0b0` Mispredicted event is ignored.

`0b1` Do not record samples that have the Mispredicted event == 0.

This bit is ignored by the PE when PMSFCR_EL1.FE == 0.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [6]

Reserved, RAZ/WI.

E[5], bit [5]

TLB walk.

`0b0` TLB walk event is ignored.

`0b1` Do not record samples that have the TLB walk event == 0.

This bit is ignored by the PE when PMSFCR_EL1.FE == 0.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [4]

Reserved, RAZ/WI.

E[3], bit [3]

Level 1 data or unified cache refill.

`0b0` Level 1 data or unified cache refill event is ignored.

`0b1` Do not record samples that have the Level 1 data or unified cache refill event == 0.

This bit is ignored by the PE when PMSFCR_EL1.FE == 0.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [2]

Reserved, RAZ/WI.

E[1], bit [1]

When the PE supports sampling of speculative instructions:

Architecturally retired.

When the PE supports sampling of speculative instructions:

`0b0` Architecturally retired event is ignored.

`0b1` Do not record samples that have the Architecturally retired event == 0.

This bit is ignored by the PE when PMSFCR_EL1.FE == 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the PE does not support the sampling of speculative instructions, or always discards the sample record for speculative instructions, this bit reads as an UNKNOWN value and the PE ignores its value.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, UNKNOWN.

Bit [0]

Reserved, RAZ/WI.

Accessing the PMSEVFR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PMSEVFR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1001	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.PMSEVFR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPMS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '1x1' then
        return NVMem[0x830];
    else
        return PMSEVFR_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMSEVFR_EL1;
elsif PSTATE.EL == EL3 then
    return PMSEVFR_EL1;
```

MSR PMSEVFR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1001	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.PMSEVFR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPMS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '1x1' then
        NVMem[0x830] = X[t];
    else
        PMSEVFR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMSEVFR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    PMSEVFR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.6.8 PMSFCR_EL1, Sampling Filter Control Register

The PMSFCR_EL1 characteristics are:

Purpose

Controls sample filtering. The filter is the logical AND of the FL, FT and FE bits. For example, if FE == 1 and FT == 1 only samples including the selected operation types and the selected events will be recorded

Configurations

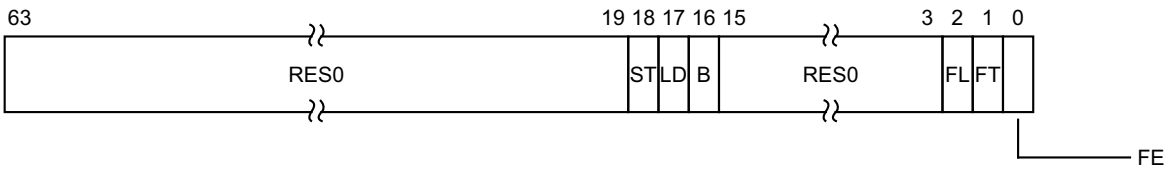
This register is present only when SPE is implemented. Otherwise, direct accesses to PMSFCR_EL1 are UNDEFINED.

Attributes

PMSFCR_EL1 is a 64-bit register.

Field descriptions

The PMSFCR_EL1 bit assignments are:



Bits [63:19]

Reserved, RES0.

ST, bit [18]

Store filter enable

`0b0` Do not record store operations

`0b1` Record all store operations, including vector stores and all atomic operations

This bit is ignored by the PE when PMSFCR_EL1.FT == 0.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

LD, bit [17]

Load filter enable

`0b0` Do not record load operations

`0b1` Record all load operations, including vector loads and atomic operations that return data

This bit is ignored by the PE when PMSFCR_EL1.FT == 0.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

B, bit [16]

Branch filter enable

`0b0` Do not record branch and exception return operations

`0b1` Record all branch and exception return operations

This bit is ignored by the PE when PMSFCR_EL1.FT == 0.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [15:3]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

FL, bit [2]

Filter by latency

`0b0` Latency filtering disabled

`0b1` Latency filtering enabled. Samples with a total latency less than PMSLATFR_EL1.MINLAT will not be recorded

If this field is set to 1 and PMSLATFR_EL1.MINLAT is set to zero, it is CONSTRAINED UNPREDICTABLE whether no samples are recorded or the PE behaves as if PMSFCR_EL1.FL is set to 0

On a Warm reset, this field resets to an architecturally UNKNOWN value.

FT, bit [1]

Filter by operation type. The filter is the logical OR of the ST, LD and B bits. For example, if LD and ST are both set, both load and store operations are recorded

`0b0` Type filtering disabled

`0b1` Type filtering enabled. Samples not one of the selected operation types will not be recorded

If this field is set to 1 and the PMSFCR_EL1.{ST, LD, B} bits are all set to zero, it is CONSTRAINED UNPREDICTABLE whether no samples are recorded or the PE behaves as if PMSFCR_EL1.FT is set to 0

On a Warm reset, this field resets to an architecturally UNKNOWN value.

FE, bit [0]

Filter by event

`0b0` Event filtering disabled

`0b1` Event filtering enabled. Samples not including the events selected by PMSEVFR_EL1 will not be recorded

If this field is set to 1 and PMSEVFR_EL1 is set to zero, it is CONSTRAINED UNPREDICTABLE whether no samples are recorded or the PE behaves as if PMSFCR_EL1.FE is set to 0

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMSFCR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PMSFCR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1001	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.PMSFCR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPMS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMSFCR_EL1;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMSFCR_EL1;
elsif PSTATE.EL == EL3 then
    return PMSFCR_EL1;
```

MSR PMSFCR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1001	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.PMSFCR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPMS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMSFCR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMSFCR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    PMSFCR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.6.9 PMSICR_EL1, Sampling Interval Counter Register

The PMSICR_EL1 characteristics are:

Purpose

Software must write zero to PMSICR_EL1 before enabling sample profiling for a sampling session. Software must then treat PMSICR_EL1 as an opaque, 64-bit, read/write register used for context switches only.

Configurations

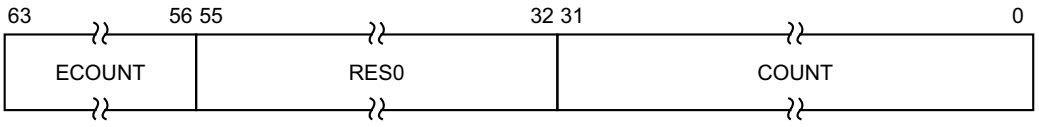
This register is present only when SPE is implemented. Otherwise, direct accesses to PMSICR_EL1 are UNDEFINED.

Attributes

The value of PMSICR_EL1 does not change whilst profiling is disabled.

Field descriptions

The PMSICR_EL1 bit assignments are:



ECOUNT, bits [63:56]

When PMSIDR_EL1.ERnd == 0b1:

Secondary sample interval counter.

When PMSIDR_EL1.ERnd is 1, this field provides the secondary counter used after the primary counter reaches zero. Whilst the secondary counter is nonzero and profiling is enabled, the secondary counter decrements by 1 for each member of the sample population. The primary counter also continues to decrement since it is also nonzero. When the secondary counter reaches zero, a member of the sampling population is selected for sampling.

When PMSIDR_EL1.ERnd is 0, this field is RES1.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [55:32]

Reserved, RES0.

COUNT, bits [31:0]

Primary sample interval counter

Provides the primary counter used for sampling.

The primary counter is reloaded when the value of this register is zero and the PE moves from a state or Exception level where profiling is disabled to a state or Exception level where profiling is enabled

Whilst the primary counter is nonzero and sampling is enabled, the primary counter decrements by 1 for each member of the sample population

When the counter reaches zero, the behavior depends on the values of PMSIDR_EL1.ERnd and PMSIRR_EL1.RND

- If PMSIRR_EL1.RND == 0 or PMSIDR_EL1.ERnd == 0:
  - A member of the sampling population is selected for sampling
  - The primary counter is reloaded

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If PMSIRR_EL1.RND == 1 and PMSIDR_EL1.ERnd == 1:
  - The secondary counter is set to a random or pseudorandom value in the range `0x00` to `0xFF`
  - The primary counter is reloaded

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMSICR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PMSICR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1001	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.PMSICR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPMS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '1x1' then
        return NVMem[0x838];
    else
        return PMSICR_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMSICR_EL1;
elsif PSTATE.EL == EL3 then
    return PMSICR_EL1;
```

MSR PMSICR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1001	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.PMSICR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPMS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '1x1' then
        NVMem[0x838] = X[t];
    else
        PMSICR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMSICR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    PMSICR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.6.10 PMSIDR_EL1, Sampling Profiling ID Register

The PMSIDR_EL1 characteristics are:

Purpose

Describes the Statistical Profiling implementation to software

Configurations

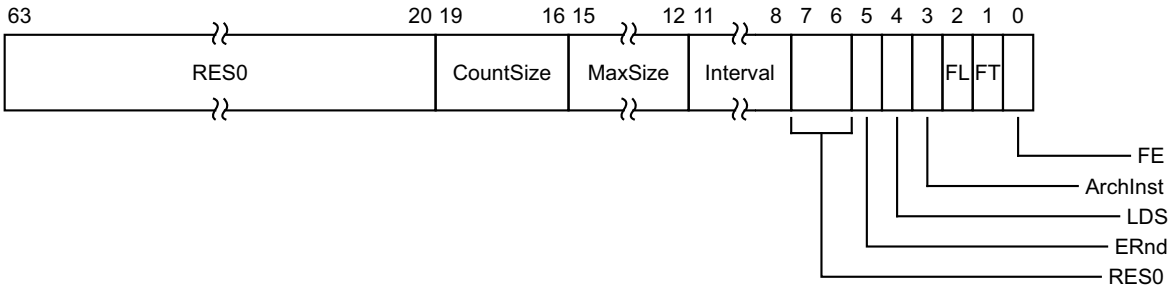
This register is present only when SPE is implemented. Otherwise, direct accesses to PMSIDR_EL1 are UNDEFINED.

Attributes

PMSIDR_EL1 is a 64-bit register.

Field descriptions

The PMSIDR_EL1 bit assignments are:



Bits [63:20]

Reserved, RES0.

CountSize, bits [19:16]

Defines the size of the counters

`0b0010` 12-bit saturating counters

All other values are reserved. Reserved values might be defined in a future version of the architecture.

MaxSize, bits [15:12]

Defines the largest size for a single record, rounded up to a power-of-two. If this is the same as the minimum alignment (PMBIDR_EL1.Align), then each record is exactly this size

`0b0100` 16 bytes

`0b0101` 32 bytes

`0b0110` 64 bytes

`0b0111` 128 bytes

`0b1000` 256 bytes

`0b1001` 512 bytes

`0b1010` 1024 bytes

`0b1011` 2KB

All other values are reserved. Reserved values might be defined in a future version of the architecture.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Interval, bits [11:8]

Recommended minimum sampling interval. This provides guidance from the implementer to the smallest minimum sampling interval, N.

`0b0000`	256
`0b0010`	512
`0b0011`	768
`0b0100`	1,024
`0b0101`	1,536
`0b0110`	2,048
`0b0111`	3,072
`0b1000`	4,096

All other values are reserved. Reserved values might be defined in a future version of the architecture.

Bits [7:6]

Reserved, RES0.

ERnd, bit [5]

Defines how the random number generator is used in determining the interval between samples, when enabled by PMSIRR_EL1.RND.

`0b0`	The random number is added at the start of the interval, and the sample is taken and a new interval started when the combined interval expires.
`0b1`	The random number is added and the new interval started after the interval programmed in PMSIRR_EL1.INTERVAL expires, and the sample is taken when the random interval expires.

LDS, bit [4]

Data source indicator for sampled load instructions

`0b0`	Loaded data source not implemented
`0b1`	Loaded data source implemented

ArchInst, bit [3]

Architectural instruction profiling

`0b0`	Micro-op sampling implemented
`0b1`	Architecture instruction sampling implemented

FL, bit [2]

Filtering by latency. This bit is RAO.

FT, bit [1]

Filtering by operation type. This bit is RAO.

FE, bit [0]

Filtering by events. This bit is RAO.

Accessing the PMSIDR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, PMSIDR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1001	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.PMSIDR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPMS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMSIDR_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMSIDR_EL1;
elsif PSTATE.EL == EL3 then
    return PMSIDR_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.6.11 PMSIRR_EL1, Sampling Interval Reload Register

The PMSIRR_EL1 characteristics are:

Purpose

Defines the interval between samples

Configurations

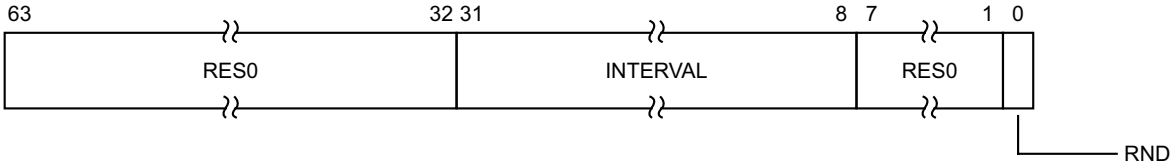
This register is present only when SPE is implemented. Otherwise, direct accesses to PMSIRR_EL1 are UNDEFINED.

Attributes

PMSIRR_EL1 is a 64-bit register.

Field descriptions

The PMSIRR_EL1 bit assignments are:



Bits [63:32]

Reserved, RES0.

INTERVAL, bits [31:8]

Bits [31:8] of the PMSICR_EL1 interval counter reload value. Software must set this to a non-zero value. If software sets this to zero, an UNKNOWN sampling interval is used. Software should set this to a value greater than the minimum indicated by PMSIDR_EL1.Interval

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [7:1]

Reserved, RES0.

RND, bit [0]

Controls randomization of the sampling interval

`0b0` Disable randomization of sampling interval

`0b1` Add (pseudo-)random jitter to sampling interval

The random number generator is not architected.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMSIRR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, PMSIRR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1001	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.PMSIRR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPMS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '1x1' then
        return NVMem[0x840];
    else
        return PMSIRR_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMSIRR_EL1;
elsif PSTATE.EL == EL3 then
    return PMSIRR_EL1;
```

MSR PMSIRR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1001	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.PMSIRR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPMS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '1x1' then
        NVMem[0x840] = X[t];
    else
        PMSIRR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMSIRR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    PMSIRR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.6.12 PMSLATFR_EL1, Sampling Latency Filter Register

The PMSLATFR_EL1 characteristics are:

Purpose

Controls sample filtering by latency

Configurations

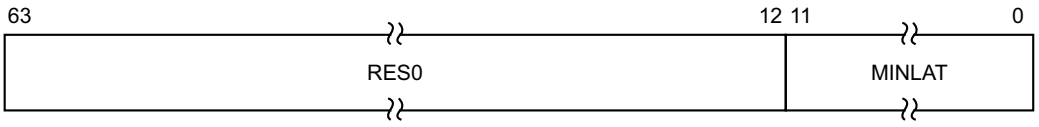
This register is present only when SPE is implemented. Otherwise, direct accesses to PMSLATFR_EL1 are UNDEFINED.

Attributes

PMSLATFR_EL1 is a 64-bit register.

Field descriptions

The PMSLATFR_EL1 bit assignments are:



Bits [63:12]

Reserved, RES0.

MINLAT, bits [11:0]

Minimum latency. When PMSFCR_EL1.FL == 1, defines the minimum total latency for filtered operations. Samples with a total latency less than MINLAT will not be recorded

This field is ignored by the PE when PMSFCR_EL1.FL == 0.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMSLATFR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PMSLATFR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1001	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGRTR_EL2.PMSLATFR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPMS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '1x1' then
        return NVMem[0x848];
    else
        return PMSLATFR_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMSLATFR_EL1;
elsif PSTATE.EL == EL3 then
    return PMSLATFR_EL1;
```

MSR PMSLATFR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1001	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWTR_EL2.PMSLATFR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPMS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '1x1' then
        NVMem[0x848] = X[t];
    else
        PMSLATFR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMSLATFR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    PMSLATFR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.7 RAS registers

This section lists the RAS Extension registers in AArch64.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.7.1 DISR_EL1, Deferred Interrupt Status Register

The DISR_EL1 characteristics are:

Purpose

Records that an SError interrupt has been consumed by an ESB instruction.

Configurations

AArch64 System register DISR_EL1[31:0] is architecturally mapped to AArch32 System register DISR[31:0].

This register is present only when RAS is implemented. Otherwise, direct accesses to DISR_EL1 are UNDEFINED.

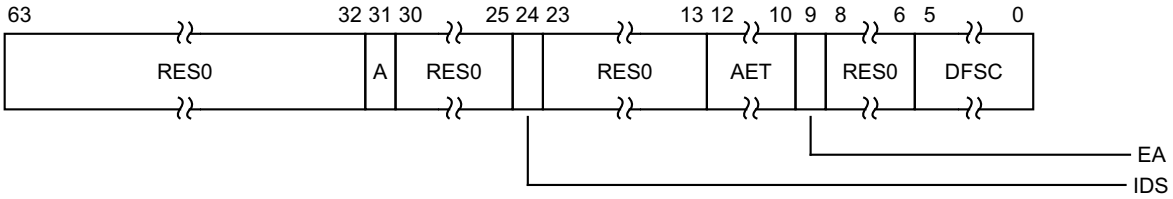
Attributes

DISR_EL1 is a 64-bit register.

Field descriptions

The DISR_EL1 bit assignments are:

When DISR_EL1.IDS == 0:



Bits [63:32]

Reserved, RES0.

A, bit [31]

Set to 1 when an ESB instruction defers an asynchronous SError interrupt. If the implementation does not include any sources of SError interrupt that can be synchronized by an Error Synchronization Barrier, then this bit is RES0.

This field resets to an architecturally UNKNOWN value.

Bits [30:25]

Reserved, RES0.

IDS, bit [24]

Indicates the deferred SError interrupt type.

`0b0` Deferred error uses architecturally-defined format.

This field resets to an architecturally UNKNOWN value.

Bits [23:13]

Reserved, RES0.

AET, bits [12:10]

Asynchronous Error Type. See the description of ESR_ELx.AET for an SError interrupt.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

EA, bit [9]

External abort Type. See the description of ESR_ELx.EA for an SError interrupt.

This field resets to an architecturally UNKNOWN value.

Bits [8:6]

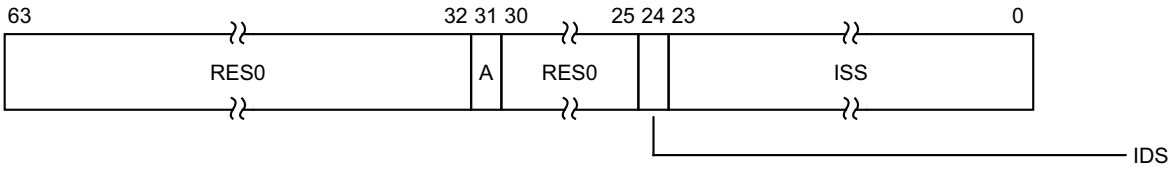
Reserved, RES0.

DFSC, bits [5:0]

Fault Status Code. See the description of ESR_ELx.DFSC for an SError interrupt.

This field resets to an architecturally UNKNOWN value.

When DISR_EL1.IDS == 1:



Bits [63:32]

Reserved, RES0.

A, bit [31]

Set to 1 when an ESB instruction defers an asynchronous SError interrupt. If the implementation does not include any sources of SError interrupt that can be synchronized by an Error Synchronization Barrier, then this bit is RES0.

This field resets to an architecturally UNKNOWN value.

Bits [30:25]

Reserved, RES0.

IDS, bit [24]

Indicates the deferred SError interrupt type.

`0b1` Deferred error uses IMPLEMENTATION DEFINED format.

This field resets to an architecturally UNKNOWN value.

ISS, bits [23:0]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED syndrome. See the description of ESR_ELx[23:0] for an SError interrupt.

This field resets to an architecturally UNKNOWN value.

The following subsections describe each ISS format.

Accessing the DISR_EL1

An indirect write to DISR_EL1 made by an ESB instruction does not require an explicit synchronization operation for the value that is written to be observed by a direct read of DISR_EL1 occurring in program order after the ESB instruction.

DISR_EL1 is RAZ/WI if EL3 is implemented, the PE is in Non-debug state, SCR_EL3.EA == 1, and any of the following apply:

- At EL2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- At EL1 and ((SCR_EL3.NS == 0 && SCR_EL3.EEL2 == 0) || HCR_EL2.AMO == 0).

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, DISR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.AMO == '1' then
        return VDISR_EL2;
    else
        return DISR_EL1;
elsif PSTATE.EL == EL2 then
    return DISR_EL1;
elsif PSTATE.EL == EL3 then
    return DISR_EL1;
```

MSR DISR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.AMO == '1' then
        VDISR_EL2 = X[t];
    else
        DISR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    DISR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    DISR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.7.2 ERRIDR_EL1, Error Record ID Register

The ERRIDR_EL1 characteristics are:

Purpose

Defines the highest numbered index of the error records that can be accessed through the Error Record System registers.

Configurations

AArch64 System register ERRIDR_EL1[31:0] is architecturally mapped to AArch32 System register ERRIDR[31:0].

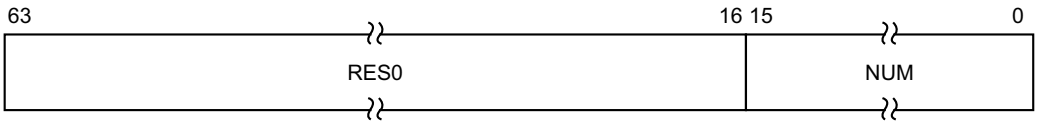
This register is present only when RAS is implemented. Otherwise, direct accesses to ERRIDR_EL1 are UNDEFINED.

Attributes

ERRIDR_EL1 is a 64-bit register.

Field descriptions

The ERRIDR_EL1 bit assignments are:



Bits [63:16]

Reserved, RES0.

NUM, bits [15:0]

Highest numbered index of the records that can be accessed through the Error Record System registers plus one. Zero indicates no records can be accessed through the Error Record System registers.

Each implemented record is owned by a node. A node might own multiple records.

Accessing the ERRIDR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ERRIDR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.ERRIDR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return ERRIDR_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return ERRIDR_EL1;
elsif PSTATE.EL == EL3 then
    return ERRIDR_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.7.3 ERRSELR_EL1, Error Record Select Register

The ERRSELR_EL1 characteristics are:

Purpose

Selects an error record to be accessed through the Error Record System registers.

Configurations

AArch64 System register ERRSELR_EL1[31:0] is architecturally mapped to AArch32 System register ERRSELR[31:0].

This register is present only when RAS is implemented. Otherwise, direct accesses to ERRSELR_EL1 are UNDEFINED.

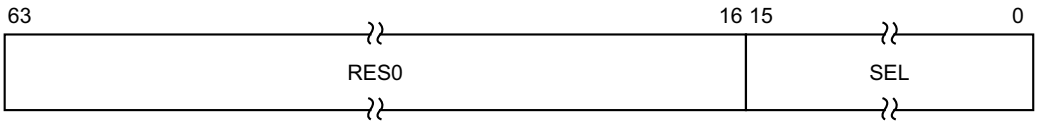
If ERRIDR_EL1 indicates that zero error records are implemented, then it is IMPLEMENTATION DEFINED whether ERRSELR_EL1 is UNDEFINED or RES0.

Attributes

ERRSELR_EL1 is a 64-bit register.

Field descriptions

The ERRSELR_EL1 bit assignments are:



Bits [63:16]

Reserved, RES0.

SEL, bits [15:0]

Selects the error record accessed through the ERX registers.

For example, if ERRSELR_EL1.SEL is set to `0x0004`, then direct reads and writes of ERXSTATUS_EL1 access ERR4STATUS.

If ERRSELR_EL1.SEL is set to a value greater than or equal to ERRIDR_EL1.NUM, then all of the following apply:

- The value read back from ERRSELR_EL1.SEL is UNKNOWN.
- One of the following occurs:
  - An UNKNOWN error record is selected.
  - The ERX*_EL1 registers are RAZ/WI.
  - ERX*_EL1 register reads and writes are NOPs.
  - ERX*_EL1 register reads and writes are UNDEFINED.

This field resets to an architecturally UNKNOWN value.

Accessing the ERRSELR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, ERRSELR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0011	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.ERRSELR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return ERRSELR_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return ERRSELR_EL1;
elsif PSTATE.EL == EL3 then
    return ERRSELR_EL1;
```

MSR ERRSELR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0011	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.ERRSELR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        ERRSELR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        ERRSELR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    ERRSELR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.7.4 ERXADDR_EL1, Selected Error Record Address Register

The ERXADDR_EL1 characteristics are:

Purpose

Accesses ERR<n>ADDR for the error record <n> selected by ERRSELR_EL1.SEL.

For details of this, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*.

Configurations

AArch64 System register ERXADDR_EL1[31:0] is architecturally mapped to AArch32 System register ERXADDR[31:0].

AArch64 System register ERXADDR_EL1[63:32] is architecturally mapped to AArch32 System register ERXADDR2[31:0].

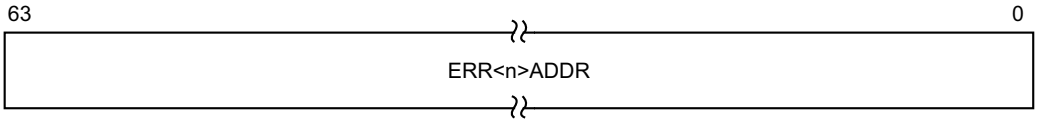
This register is present only when RAS is implemented. Otherwise, direct accesses to ERXADDR_EL1 are UNDEFINED.

Attributes

ERXADDR_EL1 is a 64-bit register.

Field descriptions

The ERXADDR_EL1 bit assignments are:



Bits [63:0]

ERXADDR_EL1 accesses ERR<n>ADDR, where <n> is the value in ERRSELR_EL1.SEL.

Accessing the ERXADDR_EL1

If ERRIDR_EL1.NUM == `0x0000` or ERRSELR_EL1.SEL is set to a value greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXADDR_EL1 is RAZ/WI.
- Direct reads and writes of ERXADDR_EL1 are NOPs.
- Direct reads and writes of ERXADDR_EL1 are UNDEFINED.

ERR<n>ADDR describes additional constraints that also apply when ERR<n>ADDR is accessed through ERXADDR_EL1.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, ERXADDR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.ERXADDR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return ERXADDR_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return ERXADDR_EL1;
elsif PSTATE.EL == EL3 then
    return ERXADDR_EL1;
```

MSR ERXADDR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.ERXADDR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        ERXADDR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        ERXADDR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    ERXADDR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.7.5 ERXCTLR_EL1, Selected Error Record Control Register

The ERXCTLR_EL1 characteristics are:

Purpose

Accesses ERR<n>CTLR for the error record <n> selected by ERRSELR_EL1.SEL.

For details of this, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*.

Configurations

AArch64 System register ERXCTLR_EL1[31:0] is architecturally mapped to AArch32 System register ERXCTLR[31:0].

AArch64 System register ERXCTLR_EL1[63:32] is architecturally mapped to AArch32 System register ERXCTLR2[31:0].

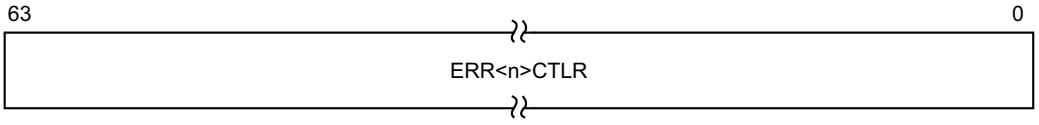
This register is present only when RAS is implemented. Otherwise, direct accesses to ERXCTLR_EL1 are UNDEFINED.

Attributes

ERXCTLR_EL1 is a 64-bit register.

Field descriptions

The ERXCTLR_EL1 bit assignments are:



Bits [63:0]

ERXCTLR_EL1 accesses ERR<n>CTLR, where <n> is the value in ERRSELR_EL1.SEL.

Accessing the ERXCTLR_EL1

If ERRIDR_EL1.NUM == `0x0000` or ERRSELR_EL1.SEL is set to a value greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXCTLR_EL1 is RAZ/WI.
- Direct reads and writes of ERXCTLR_EL1 are NOPs.
- Direct reads and writes of ERXCTLR_EL1 are UNDEFINED.

If ERRSELR_EL1.SEL is not the index of the first error record owned by a node, then ERR<n>CTLR is not present, meaning reads and writes of ERXCTLR_EL1 are RES0.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, ERXCTLR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.ERXCTLR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return ERXCTLR_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return ERXCTLR_EL1;
elsif PSTATE.EL == EL3 then
    return ERXCTLR_EL1;
```

MSR ERXCTLR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.ERXCTLR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        ERXCTLR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        ERXCTLR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    ERXCTLR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.7.6 ERXFR_EL1, Selected Error Record Feature Register

The ERXFR_EL1 characteristics are:

Purpose

Accesses ERR<n>FR for the error record <n> selected by ERRSELR_EL1.SEL.

For details of this, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*.

Configurations

AArch64 System register ERXFR_EL1[31:0] is architecturally mapped to AArch32 System register ERXFR[31:0].

AArch64 System register ERXFR_EL1[63:32] is architecturally mapped to AArch32 System register ERXFR2[31:0].

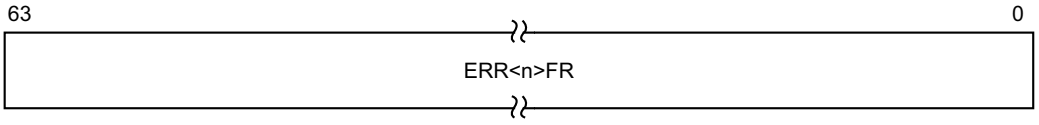
This register is present only when RAS is implemented. Otherwise, direct accesses to ERXFR_EL1 are UNDEFINED.

Attributes

ERXFR_EL1 is a 64-bit register.

Field descriptions

The ERXFR_EL1 bit assignments are:



Bits [63:0]

ERXFR_EL1 accesses ERR<n>FR, where <n> is the value in ERRSELR_EL1.SEL.

Accessing the ERXFR_EL1

If ERRIDR_EL1.NUM == `0x0000` or ERRSELR_EL1.SEL is set to a value greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXFR_EL1 is RAZ.
- Direct reads of ERXFR_EL1 are NOPs.
- Direct reads of ERXFR_EL1 are UNDEFINED.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ERXFR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.ERXFR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return ERXFR_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return ERXFR_EL1;
elsif PSTATE.EL == EL3 then
    return ERXFR_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.7.7 ERXMISC0_EL1, Selected Error Record Miscellaneous Register 0

The ERXMISC0_EL1 characteristics are:

Purpose

Accesses ERR<n>MISC0 for the error record <n> selected by ERRSELR_EL1.SEL.

For details of this, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*.

Configurations

AArch64 System register ERXMISC0_EL1[31:0] is architecturally mapped to AArch32 System register ERXMISC0[31:0].

AArch64 System register ERXMISC0_EL1[63:32] is architecturally mapped to AArch32 System register ERXMISC1[31:0].

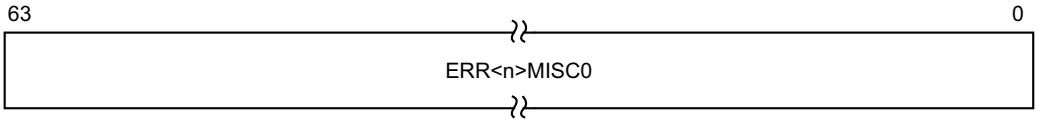
This register is present only when RAS is implemented. Otherwise, direct accesses to ERXMISC0_EL1 are UNDEFINED.

Attributes

ERXMISC0_EL1 is a 64-bit register.

Field descriptions

The ERXMISC0_EL1 bit assignments are:



Bits [63:0]

ERXMISC0_EL1 accesses ERR<n>MISC0, where <n> is the value in ERRSELR_EL1.SEL.

Accessing the ERXMISC0_EL1

If ERRIDR_EL1.NUM == `0x0000` or ERRSELR_EL1.SEL is set to a value greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC0_EL1 is RAZ/WI.
- Direct reads and writes of ERXMISC0_EL1 are NOPs.
- Direct reads and writes of ERXMISC0_EL1 are UNDEFINED.

ERR<n>MISC0 describes additional constraints that also apply when ERR<n>MISC0 is accessed through ERXMISC0_EL1.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, ERXMISC0_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0101	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.ERXMISCn_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return ERXMISC0_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return ERXMISC0_EL1;
elsif PSTATE.EL == EL3 then
    return ERXMISC0_EL1;
```

MSR ERXMISC0_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0101	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.ERXMISCn_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        ERXMISC0_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        ERXMISC0_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    ERXMISC0_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.7.8 ERXMISC1_EL1, Selected Error Record Miscellaneous Register 1

The ERXMISC1_EL1 characteristics are:

Purpose

Accesses ERR<n>MISC1 for the error record <n> selected by ERRSELR_EL1.SEL.

For details of this, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*.

Configurations

AArch64 System register ERXMISC1_EL1[31:0] is architecturally mapped to AArch32 System register ERXMISC2[31:0].

AArch64 System register ERXMISC1_EL1[63:32] is architecturally mapped to AArch32 System register ERXMISC3[31:0].

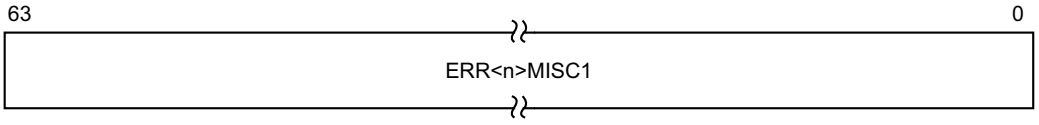
This register is present only when RAS is implemented. Otherwise, direct accesses to ERXMISC1_EL1 are UNDEFINED.

Attributes

ERXMISC1_EL1 is a 64-bit register.

Field descriptions

The ERXMISC1_EL1 bit assignments are:



Bits [63:0]

ERXMISC1_EL1 accesses ERR<n>MISC1, where <n> is the value in ERRSELR_EL1.SEL.

Accessing the ERXMISC1_EL1

If ERRIDR_EL1.NUM == `0x0000` or ERRSELR_EL1.SEL is set to a value greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC1_EL1 is RAZ/WI.
- Direct reads and writes of ERXMISC1_EL1 are NOPs.
- Direct reads and writes of ERXMISC1_EL1 are UNDEFINED.

ERR<n>MISC1 describes additional constraints that also apply when ERR<n>MISC1 is accessed through ERXMISC1_EL1.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, ERXMISC1_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0101	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.ERXMISCn_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return ERXMISC1_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return ERXMISC1_EL1;
elsif PSTATE.EL == EL3 then
    return ERXMISC1_EL1;
```

MSR ERXMISC1_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0101	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.ERXMISCn_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        ERXMISC1_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        ERXMISC1_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    ERXMISC1_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.7.9 ERXMISC2_EL1, Selected Error Record Miscellaneous Register 2

The ERXMISC2_EL1 characteristics are:

Purpose

Accesses ERR<n>MISC2 for the error record <n> selected by ERRSELR_EL1.SEL.

For details of this, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*.

Configurations

AArch64 System register ERXMISC2_EL1[31:0] is architecturally mapped to AArch32 System register ERXMISC4[31:0].

AArch64 System register ERXMISC2_EL1[63:32] is architecturally mapped to AArch32 System register ERXMISC5[31:0].

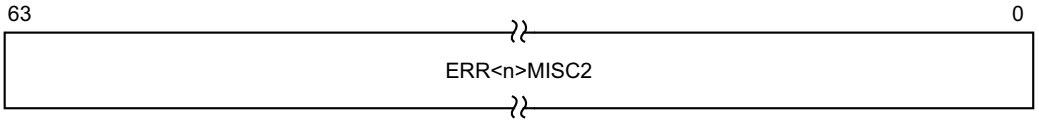
This register is present only when ARMv8.4-RAS is implemented. Otherwise, direct accesses to ERXMISC2_EL1 are UNDEFINED.

Attributes

ERXMISC2_EL1 is a 64-bit register.

Field descriptions

The ERXMISC2_EL1 bit assignments are:



Bits [63:0]

ERXMISC2_EL1 accesses ERR<n>MISC2, where <n> is the value in ERRSELR_EL1.SEL.

Accessing the ERXMISC2_EL1

If ERRIDR_EL1.NUM == `0x0000` or ERRSELR_EL1.SEL is set to a value greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC2_EL1 is RAZ/WI.
- Direct reads and writes of ERXMISC2_EL1 are NOPs.
- Direct reads and writes of ERXMISC2_EL1 are UNDEFINED.

ERR<n>MISC2 describes additional constraints that also apply when ERR<n>MISC2 is accessed through ERXMISC2_EL1.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, ERXMISC2_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0101	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.ERXMISCn_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return ERXMISC2_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return ERXMISC2_EL1;
elsif PSTATE.EL == EL3 then
    return ERXMISC2_EL1;
```

MSR ERXMISC2_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0101	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.ERXMISCn_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        ERXMISC2_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        ERXMISC2_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    ERXMISC2_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.7.10 ERXMISC3_EL1, Selected Error Record Miscellaneous Register 3

The ERXMISC3_EL1 characteristics are:

Purpose

Accesses ERR<n>MISC3 for the error record <n> selected by ERRSELR_EL1.SEL.

For details of this, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*.

Configurations

AArch64 System register ERXMISC3_EL1[31:0] is architecturally mapped to AArch32 System register ERXMISC6[31:0].

AArch64 System register ERXMISC3_EL1[63:32] is architecturally mapped to AArch32 System register ERXMISC7[31:0].

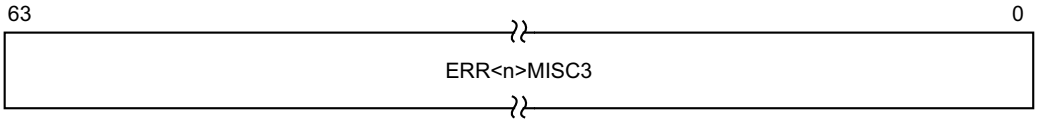
This register is present only when ARMv8.4-RAS is implemented. Otherwise, direct accesses to ERXMISC3_EL1 are UNDEFINED.

Attributes

ERXMISC3_EL1 is a 64-bit register.

Field descriptions

The ERXMISC3_EL1 bit assignments are:



Bits [63:0]

ERXMISC3_EL1 accesses ERR<n>MISC3, where <n> is the value in ERRSELR_EL1.SEL.

Accessing the ERXMISC3_EL1

If ERRIDR_EL1.NUM == `0x0000` or ERRSELR_EL1.SEL is set to a value greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC3_EL1 is RAZ/WI.
- Direct reads and writes of ERXMISC3_EL1 are NOPs.
- Direct reads and writes of ERXMISC3_EL1 are UNDEFINED.

ERR<n>MISC3 describes additional constraints that also apply when ERR<n>MISC3 is accessed through ERXMISC3_EL1.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, ERXMISC3_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0101	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.ERXMISCn_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return ERXMISC3_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return ERXMISC3_EL1;
elsif PSTATE.EL == EL3 then
    return ERXMISC3_EL1;
```

MSR ERXMISC3_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0101	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.ERXMISCn_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        ERXMISC3_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        ERXMISC3_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    ERXMISC3_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.7.11 ERXPFGCDN_EL1, Selected Pseudo-fault Generation Countdown register

The ERXPFGCDN_EL1 characteristics are:

Purpose

Accesses ERR<n>PFGCDN for the error record <n> selected by ERRSELR_EL1.SEL.

For details of this, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*.

Configurations

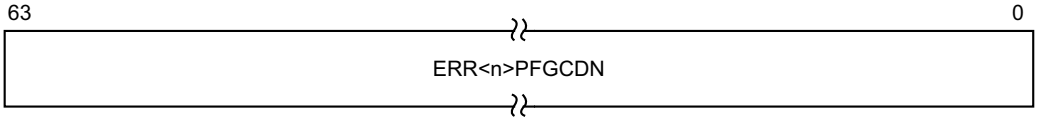
This register is present only when ARMv8.4-RAS is implemented. Otherwise, direct accesses to ERXPFGCDN_EL1 are UNDEFINED.

Attributes

ERXPFGCDN_EL1 is a 64-bit register.

Field descriptions

The ERXPFGCDN_EL1 bit assignments are:



Bits [63:0]

ERXPFGCDN_EL1 accesses ERR<n>PFGCDN, where <n> is the value in ERRSELR_EL1.SEL.

Accessing the ERXPFGCDN_EL1

If ERRIDR_EL1.NUM == `0x0000` or ERRSELR_EL1.SEL is set to a value greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXPFGCDN_EL1 is RAZ/WI.
- Direct reads and writes of ERXPFGCDN_EL1 are NOPs.
- Direct reads and writes of ERXPFGCDN_EL1 are UNDEFINED.

If ERRSELR_EL1.SEL selects an error record owned by a node that does not implement the RAS Common Fault Injection Model Extension, then one of the following occurs:

- ERXPFGCDN_EL1 is RAZ/WI.
- Direct reads and writes of ERXPFGCDN_EL1 are NOPs.
- Direct reads and writes of ERXPFGCDN_EL1 are UNDEFINED.

Note

A node does not implement the RAS Common Fault Injection Model Extension when ERR<n>FR.INJ == `0b00`. <q> is the index of the first error record owned by the same node as error record <n>, where <n> is the value in ERRSELR_EL1.SEL. If the node owns a single record, then q = n.

If ERRSELR_EL1.SEL is not the index of the first error record owned by a node, then ERR<n>PFGCDN is not present, meaning reads and writes of ERXPFGCDN_EL1 are RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ERR<n>PFGCDN describes additional constraints that also apply when ERR<n>PFGCDN is accessed through ERXPFGCDN_EL1.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ERXPFGCDN_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.FIEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.ERXPFGCDN_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.FIEN == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return ERXPFGCDN_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.FIEN == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return ERXPFGCDN_EL1;
elsif PSTATE.EL == EL3 then
    return ERXPFGCDN_EL1;
```

MSR ERXPFGCDN_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.FIEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.ERXPFGCDN_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.FIEN == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        ERXPFGCDN_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.FIEN == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        ERXPFGCDN_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    ERXPFGCDN_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.7.12 ERXPFGCTL_EL1, Selected Pseudo-fault Generation Control register

The ERXPFGCTL_EL1 characteristics are:

Purpose

Accesses ERR<n>PFGCTL for the error record <n> selected by ERRSELR_EL1.SEL.

For details of this, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*.

Configurations

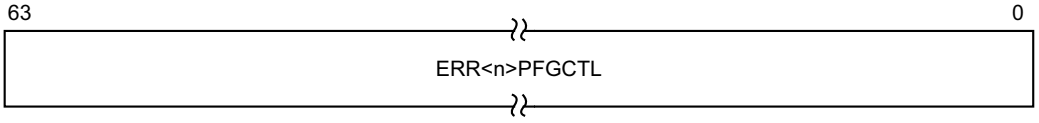
This register is present only when ARMv8.4-RAS is implemented. Otherwise, direct accesses to ERXPFGCTL_EL1 are UNDEFINED.

Attributes

ERXPFGCTL_EL1 is a 64-bit register.

Field descriptions

The ERXPFGCTL_EL1 bit assignments are:



Bits [63:0]

ERXPFGCTL_EL1 accesses ERR<n>PFGCTL, where <n> is the value in ERRSELR_EL1.SEL.

Accessing the ERXPFGCTL_EL1

If ERRIDR_EL1.NUM == `0x0000` or ERRSELR_EL1.SEL is set to a value greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXPFGCTL_EL1 is RAZ/WI.
- Direct reads and writes of ERXPFGCTL_EL1 are NOPs.
- Direct reads and writes of ERXPFGCTL_EL1 are UNDEFINED.

If ERRSELR_EL1.SEL selects an error record owned by a node that does not implement the RAS Common Fault Injection Model Extension, then one of the following occurs:

- ERXPFGCTL_EL1 is RAZ/WI.
- Direct reads and writes of ERXPFGCTL_EL1 are NOPs.
- Direct reads and writes of ERXPFGCTL_EL1 are UNDEFINED.

Note

A node does not implement the RAS Common Fault Injection Model Extension when ERR<n>FR.INJ == `0b00`. <q> is the index of the first error record owned by the same node as error record <n>, where <n> is the value in ERRSELR_EL1.SEL. If the node owns a single record, then q = n.

If ERRSELR_EL1.SEL is not the index of the first error record owned by a node, then ERR<n>PFGCTL is not present, meaning reads and writes of ERXPFGCTL_EL1 are RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ERR<n>PFGCTL describes additional constraints that also apply when ERR<n>PFGCTL is accessed through ERXPFGCTL_EL1.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ERXPFGCTL_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.FIEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.ERXPFGCTL_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.FIEN == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return ERXPFGCTL_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.FIEN == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return ERXPFGCTL_EL1;
elsif PSTATE.EL == EL3 then
    return ERXPFGCTL_EL1;
```

MSR ERXPFGCTL_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.FIEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.ERXPFGCTL_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.FIEN == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        ERXPFGCTL_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.FIEN == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        ERXPFGCTL_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    ERXPFGCTL_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.7.13 ERXPFGF_EL1, Selected Pseudo-fault Generation Feature register

The ERXPFGF_EL1 characteristics are:

Purpose

Accesses ERR<n>PFGF for the error record <n> selected by ERRSELR_EL1.SEL.

For details of this, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*.

Configurations

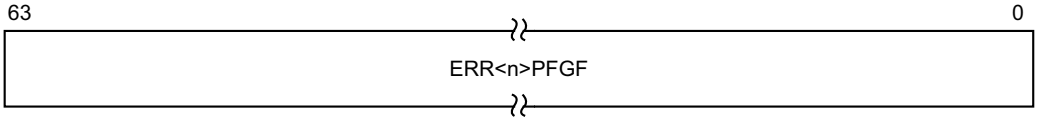
This register is present only when ARMv8.4-RAS is implemented. Otherwise, direct accesses to ERXPFGF_EL1 are UNDEFINED.

Attributes

ERXPFGF_EL1 is a 64-bit register.

Field descriptions

The ERXPFGF_EL1 bit assignments are:



Bits [63:0]

ERXPFGF_EL1 accesses ERR<n>PFGF, where <n> is the value in ERRSELR_EL1.SEL.

Accessing the ERXPFGF_EL1

If ERRIDR_EL1.NUM == `0x0000` or ERRSELR_EL1.SEL is set to a value greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXPFGF_EL1 is RAZ.
- Direct reads of ERXPFGF_EL1 are NOPs.
- Direct reads of ERXPFGF_EL1 are UNDEFINED.

If ERRSELR_EL1.SEL selects an error record owned by a node that does not implement the RAS Common Fault Injection Model Extension, then one of the following occurs:

- ERXPFGF_EL1 is RAZ.
- Direct reads of ERXPFGF_EL1 are NOPs.
- Direct reads of ERXPFGF_EL1 are UNDEFINED.

Note

A node does not implement the RAS Common Fault Injection Model Extension when ERR<n>FR.INJ == `0b00`. <q> is the index of the first error record owned by the same node as error record <n>, where <n> is the value in ERRSELR_EL1.SEL. If the node owns a single record, then q = n.

If ERRSELR_EL1.SEL is not the index of the first error record owned by a node, then ERR<n>PFGF is not present, meaning reads of ERXPFGF_EL1 are RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ERR<n>PFGF describes additional constraints that also apply when ERR<n>PFGF is accessed through ERXPFGF_EL1.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ERXPFGF_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.FIEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.ERXPFGF_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.FIEN == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return ERXPFGF_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.FIEN == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return ERXPFGF_EL1;
elsif PSTATE.EL == EL3 then
    return ERXPFGF_EL1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.7.14 ERXSTATUS_EL1, Selected Error Record Primary Status Register

The ERXSTATUS_EL1 characteristics are:

Purpose

Accesses ERR<n>STATUS for the error record <n> selected by ERRSELR_EL1.SEL.

For details of this, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*.

Configurations

AArch64 System register ERXSTATUS_EL1[31:0] is architecturally mapped to AArch32 System register ERXSTATUS[31:0].

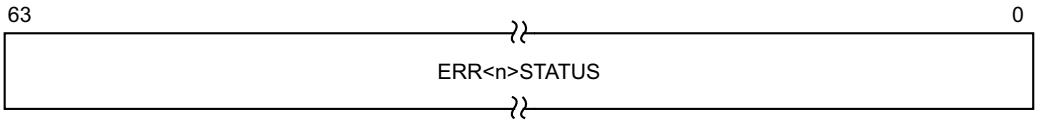
This register is present only when RAS is implemented. Otherwise, direct accesses to ERXSTATUS_EL1 are UNDEFINED.

Attributes

ERXSTATUS_EL1 is a 64-bit register.

Field descriptions

The ERXSTATUS_EL1 bit assignments are:



Bits [63:0]

ERXSTATUS_EL1 accesses ERR<n>STATUS, where <n> is the value in ERRSELR_EL1.SEL.

Accessing the ERXSTATUS_EL1

If ERRIDR_EL1.NUM == `0x0000` or ERRSELR_EL1.SEL is set to a value greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXSTATUS_EL1 is RAZ/WI.
- Direct reads and writes of ERXSTATUS_EL1 are NOPs.
- Direct reads and writes of ERXSTATUS_EL1 are UNDEFINED.

ERR<n>STATUS describes additional constraints that also apply when ERR<n>STATUS is accessed through ERXSTATUS_EL1.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ERXSTATUS_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.ERXSTATUS_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return ERXSTATUS_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return ERXSTATUS_EL1;
elsif PSTATE.EL == EL3 then
    return ERXSTATUS_EL1;
```

MSR ERXSTATUS_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGWTR_EL2.ERXSTATUS_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        ERXSTATUS_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        ERXSTATUS_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    ERXSTATUS_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.7.15 VDISR_EL2, Virtual Deferred Interrupt Status Register

The VDISR_EL2 characteristics are:

Purpose

Records that a virtual SError interrupt has been consumed by an ESB instruction executed at EL1.

An indirect write to VDISR_EL2 made by an ESB instruction does not require an explicit synchronization operation for the value written to be observed by a direct read of DISR_EL1 or DISR occurring in program order after the ESB instruction.

Configurations

AArch64 System register VDISR_EL2[31:0] is architecturally mapped to AArch32 System register VDISR[31:0].

This register is present only when RAS is implemented. Otherwise, direct accesses to VDISR_EL2 are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

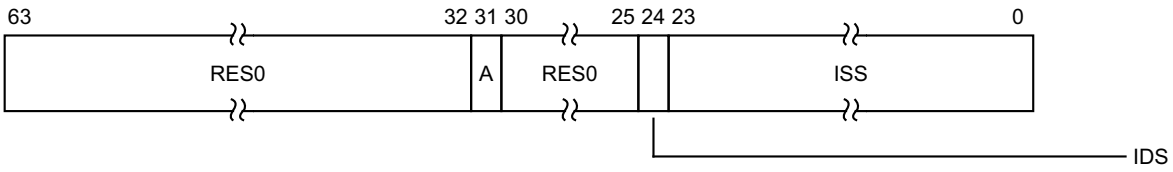
Attributes

VDISR_EL2 is a 64-bit register.

Field descriptions

The VDISR_EL2 bit assignments are:

When EL1 is using AArch64:



Bits [63:32]

Reserved, RES0.

A, bit [31]

Set to 1 when an ESB instruction defers a virtual SError interrupt.

This field resets to an architecturally UNKNOWN value.

Bits [30:25]

Reserved, RES0.

IDS, bit [24]

The value copied from VSESR_EL2.IDS.

This field resets to an architecturally UNKNOWN value.

ISS, bits [23:0]

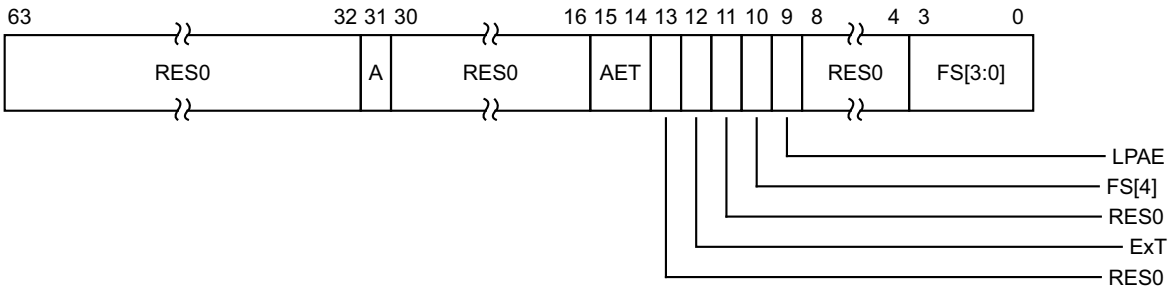
The value copied from VSESR_EL2.ISS.

This field resets to an architecturally UNKNOWN value.

The following subsections describe each ISS format.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When EL1 is using AArch32 and VDISR_EL2.LPAE == 0:



Bits [63:32]

Reserved, RES0.

A, bit [31]

Set to 1 when an `ESB` instruction defers a virtual SError interrupt.

This field resets to an architecturally UNKNOWN value.

Bits [30:16]

Reserved, RES0.

AET, bits [15:14]

The value copied from VSESR_EL2.AET.

This field resets to an architecturally UNKNOWN value.

Bit [13]

Reserved, RES0.

ExT, bit [12]

The value copied from VSESR_EL2.ExT.

This field resets to an architecturally UNKNOWN value.

Bit [11]

Reserved, RES0.

FS[4], bit [10]

This field is bit[4] of FS[4:0].

Fault status code. Set to `0b10110` when an `ESB` instruction defers a virtual SError interrupt.

`0b10110` Asynchronous SError interrupt.

All other values are reserved.

The FS field is split as follows:

- FS[4] is VDISR_EL2[10].
- FS[3:0] is VDISR_EL2[3:0].

This field resets to an architecturally UNKNOWN value.

LPAE, bit [9]

Format.

Set to TTBCR.EAE when an `ESB` instruction defers a virtual SError interrupt.

`0b0` Using the Short-descriptor translation table format.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bits [8:4]

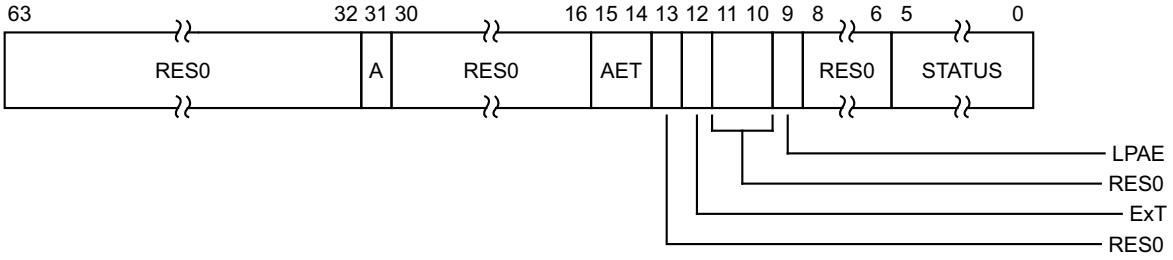
Reserved, RES0.

FS[3:0], bits [3:0]

This field is bits[3:0] of FS[4:0].

See FS[4] for the field description.

When EL1 is using AArch32 and VDISR_EL2.LPAE == 1:



Bits [63:32]

Reserved, RES0.

A, bit [31]

Set to 1 when an `ESB` instruction defers a virtual SError interrupt.

This field resets to an architecturally UNKNOWN value.

Bits [30:16]

Reserved, RES0.

AET, bits [15:14]

The value copied from VSESR_EL2.AET.

This field resets to an architecturally UNKNOWN value.

Bit [13]

Reserved, RES0.

ExT, bit [12]

The value copied from VSESR_EL2.ExT.

This field resets to an architecturally UNKNOWN value.

Bits [11:10]

Reserved, RES0.

LPAE, bit [9]

Format.

Set to TTBCR.EAE when an `ESB` instruction defers a virtual SError interrupt.

`0b1` Using the Long-descriptor translation table format.

This field resets to an architecturally UNKNOWN value.

Bits [8:6]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

STATUS, bits [5:0]

Fault status code. Set to `0b010001` when an `ESB` instruction defers a virtual SError interrupt.

`0b010001` Asynchronous SError interrupt.

All other values are reserved.

This field resets to an architecturally UNKNOWN value.

Accessing the VDISR_EL2

An indirect write to VDISR_EL2 made by an `ESB` instruction does not require an explicit synchronization operation for the value that is written to be observed by a direct read of DISR_EL1 or DISR occurring in program order after the `ESB` instruction.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, VDISR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1100	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        return NVMem[0x500];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return VDISR_EL2;
elsif PSTATE.EL == EL3 then
    return VDISR_EL2;
```

MSR VDISR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1100	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NVMem[0x500] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    VDISR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    VDISR_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, DISR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.AMO == '1' then
        return VDISR_EL2;
    else
        return DISR_EL1;
elsif PSTATE.EL == EL2 then
    return DISR_EL1;
elsif PSTATE.EL == EL3 then
    return DISR_EL1;
```

MSR DISR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.AMO == '1' then
        VDISR_EL2 = X[t];
    else
        DISR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    DISR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    DISR_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.7.16 VSESR_EL2, Virtual SError Exception Syndrome Register

The VSESR_EL2 characteristics are:

Purpose

Provides the syndrome value reported to software on taking a virtual SError interrupt exception to EL1, or on executing an ESB instruction at EL1.

When the virtual SError interrupt is taken to EL1 using AArch64, then the syndrome value is reported in ESR_EL1.

When the virtual SError interrupt is taken to EL1 using AArch32, then the syndrome value is reported in DFSR.{AET, ExT} and the remainder of DFSR is set as defined by VMSAv8-32. For more information, see Chapter G5 *The AArch32 Virtual Memory System Architecture*.

When the virtual SError interrupt is deferred by an ESB instruction, then the syndrome value is written to VDISR_EL2.

Configurations

AArch64 System register VSESR_EL2[31:0] is architecturally mapped to AArch32 System register VDFSR[31:0].

This register is present only when RAS is implemented. Otherwise, direct accesses to VSESR_EL2 are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

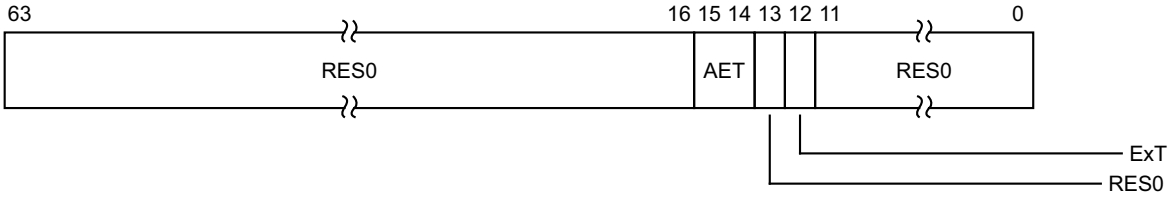
Attributes

VSESR_EL2 is a 64-bit register.

Field descriptions

The VSESR_EL2 bit assignments are:

When EL1 is using AArch32:



Bits [63:16]

Reserved, RES0.

AET, bits [15:14]

When a virtual SError interrupt is taken to EL1 using AArch32, DFSR[15:4] is set to VSESR_EL2.AET.

When a virtual SError interrupt is deferred by an ESB instruction, VDISR_EL2[15:4] is set to VSESR_EL2.AET.

This field resets to an architecturally UNKNOWN value.

Bit [13]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ExT, bit [12]

When a virtual SError interrupt is taken to EL1 using AArch32, DFSR[12] is set to VSESR_EL2.ExT.

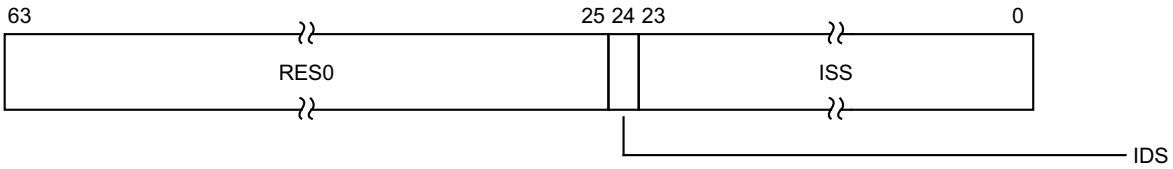
When a virtual SError interrupt is deferred by an ESB instruction, VDISR_EL2[12] is set to VSESR_EL2.ExT.

This field resets to an architecturally UNKNOWN value.

Bits [11:0]

Reserved, RES0.

When EL1 is using AArch64:



Bits [63:25]

Reserved, RES0.

IDS, bit [24]

When a virtual SError interrupt is taken to EL1 using AArch64, ESR_EL1[24] is set to VSESR_EL2.IDS.

When a virtual SError interrupt is deferred by an ESB instruction, VDISR_EL2[24] is set to VSESR_EL2.IDS.

This field resets to an architecturally UNKNOWN value.

ISS, bits [23:0]

When a virtual SError interrupt is taken to EL1 using AArch64, ESR_EL1[23:0] is set to VSESR_EL2.ISS.

When a virtual SError interrupt is deferred by an ESB instruction, VDISR_EL2[23:0] is set to VSESR_EL2.ISS.

This field resets to an architecturally UNKNOWN value.

The following subsections describe each ISS format.

Accessing the VSESR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, VSESR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0010	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        return NVMem[0x508];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return VSESR_EL2;
elsif PSTATE.EL == EL3 then
    return VSESR_EL2;
```

MSR VSESR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0010	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NVMem[0x508] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    VSESR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    VSESR_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.8 Generic Timer registers

This section lists the Generic Timer registers in AArch64.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.8.1 CNTFRQ_EL0, Counter-timer Frequency register

The CNTFRQ_EL0 characteristics are:

Purpose

This register is provided so that software can discover the frequency of the system counter. It must be programmed with this value as part of system initialization. The value of the register is not interpreted by hardware.

Configurations

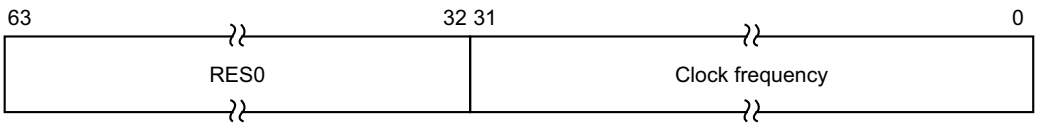
AArch64 System register CNTFRQ_EL0[31:0] is architecturally mapped to AArch32 System register CNTFRQ[31:0].

Attributes

CNTFRQ_EL0 is a 64-bit register.

Field descriptions

The CNTFRQ_EL0 bit assignments are:



Bits [63:32]

Reserved, RES0.

Bits [31:0]

Clock frequency. Indicates the system counter clock frequency, in Hz.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTFRQ_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, CNTFRQ_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') &&
CNTKCTL_EL1.<EL0PCTEN,EL0VCTEN> == '00' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' &&
CNTHCTL_EL2.<EL0PCTEN,EL0VCTEN> == '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return CNTFRQ_EL0;
elsif PSTATE.EL == EL1 then
    return CNTFRQ_EL0;
elsif PSTATE.EL == EL2 then
    return CNTFRQ_EL0;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    return CNTFRQ_EL0;
```

MSR CNTFRQ_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0000	0b000

```
if IsHighestEL(PSTATE.EL) then
    CNTFRQ_EL0 = X[t];
else
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.8.2 CNTHCTL_EL2, Counter-timer Hypervisor Control register

The CNTHCTL_EL2 characteristics are:

Purpose

Controls the generation of an event stream from the physical counter, and access from EL1 to the physical counter and the EL1 physical timer.

Configurations

AArch64 System register CNTHCTL_EL2[31:0] is architecturally mapped to AArch32 System register CNTHCTL[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

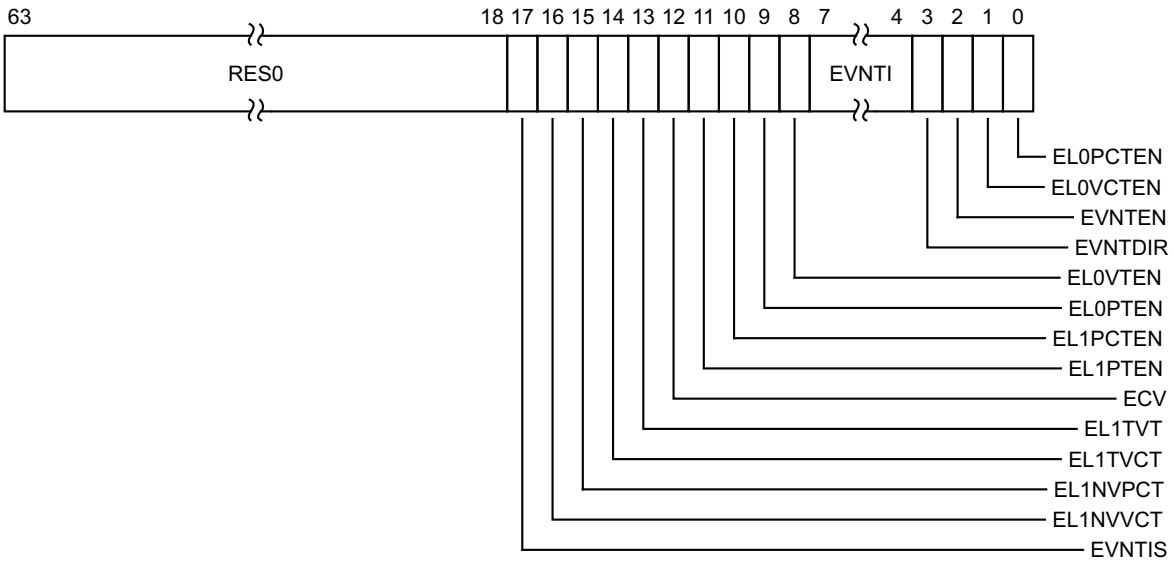
Attributes

CNTHCTL_EL2 is a 64-bit register.

Field descriptions

The CNTHCTL_EL2 bit assignments are:

When ARMv8.1-VHE is implemented and HCR_EL2.E2H == 1:



Bits [63:18]

Reserved, RES0.

EVNTIS, bit [17]

When ARMv8.6-ECV is implemented:

Controls the scale of the generation of the event stream.

`0b0` The CNTHCTL_EL2.EVNTI field applies to CNTPCT_EL0[15:0].

`0b1` The CNTHCTL_EL2.EVNTI field applies to CNTPCT_EL0[23:8].

This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

Reserved, RES0.

EL1NVVCT, bit [16]

When ARMv8.6-ECV is implemented:

Traps EL1 accesses to the specified EL1 virtual timer registers using the EL02 descriptors to EL2, when EL2 is enabled for the current Security state.

`0b0` This control does not cause any instructions to be trapped.

`0b1` If ((HCR_EL2.E2H==1 && HCR_EL2.TGE==1) || HCR_EL2.NV2==0 || HCR_EL2.NV1==1 || HCR_EL2.NV==0), this control does not cause any instructions to be trapped.

If ((HCR_EL2.E2H==0 || HCR_EL2.TGE==0) && HCR_EL2.NV2==1 && HCR_EL2.NV1==0 && HCR_EL2.NV==1), then EL1 accesses to CNTV_CTL_EL02 and CNTV_CVAL_EL02 are trapped to EL2.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 0 other than for the purpose of a direct read.

This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EL1NVPCT, bit [15]

When ARMv8.6-ECV is implemented:

Traps EL1 accesses to the specified EL1 physical timer registers using the EL02 descriptors to EL2, when EL2 is enabled for the current Security state.

`0b0` This control does not cause any instructions to be trapped.

`0b1` If ((HCR_EL2.E2H==1 && HCR_EL2.TGE==1) || HCR_EL2.NV2==0 || HCR_EL2.NV1==1 || HCR_EL2.NV==0), this control does not cause any instructions to be trapped.

If (HCR_EL2.E2H==0 || HCR_EL2.TGE==0) && HCR_EL2.NV2==1 && HCR_EL2.NV1==0 && HCR_EL2.NV==1, then EL1 accesses to CNTP_CTL_EL02 and CNTP_CVAL_EL02, are trapped to EL2.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 0 other than for the purpose of a direct read.

This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EL1TVCT, bit [14]

When ARMv8.6-ECV is implemented:

Traps EL0 and EL1 accesses to the EL1 virtual counter registers to EL2, when EL2 is enabled for the current Security state.

`0b0` This control does not cause any instructions to be trapped.

`0b1` If HCR_EL2.{E2H, TGE} is {1, 1}, this control does not cause any instructions to be trapped.

If HCR_EL2.E2H is 0 or HCR_EL2.TGE is 0, then:

- In AArch64 state, traps EL0 and EL1 accesses to CNTVCT_EL0 to EL2, unless they are trapped by CNTKCTL_EL1.EL0VCTEN.
- In AArch32 state, traps EL0 and EL1 accesses to CNTVCT to EL2, unless they are trapped by CNTKCTL_EL1.EL0VCTEN or CNTKCTL.PL0VCTEN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 0 other than for the purpose of a direct read.

This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EL1TVT, bit [13]

When ARMv8.6-ECV is implemented:

Traps EL0 and EL1 accesses to the EL1 virtual timer registers to EL2, when EL2 is enabled for the current Security state.

`0b0` This control does not cause any instructions to be trapped.

`0b1` If HCR_EL2.{E2H, TGE} is {1, 1}, this control does not cause any instructions to be trapped.

If HCR_EL2.E2H is 0 or HCR_EL2.TGE is 0, then:

- In AArch64 state, traps EL0 and EL1 accesses to CNTV_CTL_EL0, CNTV_CVAL_EL0, and CNTV_TVAL_EL0 to EL2, unless they are trapped by CNTKCTL_EL1.EL0VTEN.
- In AArch32 state, traps EL0 and EL1 accesses to CNTV_CTL, CNTV_CVAL, and CNTV_TVAL to EL2, unless they are trapped by CNTKCTL_EL1.EL0VTEN or CNTKCTL.PL0VTEN.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 0 other than for the purpose of a direct read.

This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ECV, bit [12]

When ARMv8.6-ECV is implemented:

Enables the Enhanced Counter Virtualization functionality registers.

`0b0` Enhanced Counter Virtualization functionality is disabled.

`0b1` When HCR_EL2.{E2H, TGE} == {1, 1} or SCR_EL3.{NS, EEL2} == {0, 0}, then Enhanced Counter Virtualization functionality is disabled.

When SCR_EL3.NS or SCR_EL3.EEL2 are 1, and HCR_EL2.E2H or HCR_EL2.TGE are 0, then Enhanced Counter Virtualization functionality is enabled when EL2 is enabled for the current Security state. This means that:

- An MRS to CNTPCT_EL0 from either EL0 or EL1 that is not trapped will return the value (PCount<63:0> - CNTPOFF_EL2<63:0>).
- The EL1 physical timer interrupt is triggered when ((PCount<63:0> - CNTPOFF_EL2<63:0>) - PCVal<63:0>) is greater than or equal to 0. PCount<63:0> is the physical count returned when CNTPCT_EL0 is read from EL2 or EL3. PCVal<63:0> is the EL1 physical timer compare value for this timer.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

EL1PTEN, bit [11]

When HCR_EL2.TGE is 0, traps EL0 and EL1 accesses to the E1 physical timer registers to EL2 when EL2 is enabled in the current Security state.

`0b0` From AArch64 state: EL0 and EL1 accesses to the CNTP_CTL_EL0, CNTP_CVAL_EL0, and CNTP_TVAL_EL0 are trapped to EL2 when EL2 is enabled in the current Security state, unless they are trapped by CNTKCTL_EL1.EL0PTEN.

From AArch32 state: EL0 and EL1 accesses to the CNTP_CTL, CNTP_CVAL, and CNTP_TVAL are trapped to EL2 when EL2 is enabled in the current Security state, unless they are trapped by CNTKCTL_EL1.EL0PTEN or CNTKCTL.PL0PTEN.

`0b1` This control does not cause any instructions to be trapped.

When HCR_EL2.TGE is 1, this control does not cause any instructions to be trapped.

This field resets to an architecturally UNKNOWN value.

EL1PCTEN, bit [10]

When HCR_EL2.TGE is 0, traps EL0 and EL1 accesses to the EL1 physical counter register to EL2 when EL2 is enabled in the current Security state, as follows:

- In AArch64 state, accesses to CNTPCT_EL0 are trapped to EL2, reported using EC syndrome value `0x18`.
- In AArch32 state, MRRC or MCRR accesses to CNTPCT are trapped to EL2, reported using EC syndrome value `0x04`.

`0b0` From AArch64 state: EL0 and EL1 accesses to the CNTPCT_EL0 are trapped to EL2 when EL2 is enabled in the current Security state, unless they are trapped by CNTKCTL_EL1.EL0PCTEN.

From AArch32 state: EL0 and EL1 accesses to the CNTPCT are trapped to EL2 when EL2 is enabled in the current Security state, unless they are trapped by CNTKCTL_EL1.EL0PCTEN or CNTKCTL.PL0PCTEN.

`0b1` This control does not cause any instructions to be trapped.

When HCR_EL2.TGE is 1, this control does not cause any instructions to be trapped.

This field resets to an architecturally UNKNOWN value.

EL0PTEN, bit [9]

When HCR_EL2.TGE is 0, this control does not cause any instructions to be trapped.

When HCR_EL2.TGE is 1, traps EL0 accesses to the physical timer registers to EL2.

`0b0` EL0 using AArch64: EL0 accesses to the CNTP_CTL_EL0, CNTP_CVAL_EL0, and CNTP_TVAL_EL0 registers are trapped to EL2.

EL0 using AArch32: EL0 accesses to the CNTP_CTL, CNTP_CVAL and CNTP_TVAL registers are trapped to EL2.

`0b1` This control does not cause any instructions to be trapped.

This field resets to an architecturally UNKNOWN value.

EL0VTEN, bit [8]

When HCR_EL2.TGE is 0, this control does not cause any instructions to be trapped.

When HCR_EL2.TGE is 1, traps EL0 accesses to the virtual timer registers to EL2.

`0b0` EL0 using AArch64: EL0 accesses to the CNTV_CTL_EL0, CNTV_CVAL_EL0, and CNTV_TVAL_EL0 registers are trapped to EL2.

EL0 using AArch32: EL0 accesses to the CNTV_CTL, CNTV_CVAL, and CNTV_TVAL registers are trapped to EL2.

`0b1` This control does not cause any instructions to be trapped.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

EVNTI, bits [7:4]

Selects which bit (0 to 15) of the counter register CNTPCT_EL0 is the trigger for the event stream generated from that counter, when that stream is enabled.

If ARMv8.6-ECV is implemented, and CNTHCTL_EL2.EVNTIS is 1, this selects which bit (8 to 23) of the counter register CNTPCT_EL0 is the trigger.

This field resets to an architecturally UNKNOWN value.

EVNTDIR, bit [3]

Controls which transition of the counter register CNTPCT_EL0 trigger bit, defined by EVNTI, generates an event when the event stream is enabled:

`0b0` A 0 to 1 transition of the trigger bit triggers an event.

`0b1` A 1 to 0 transition of the trigger bit triggers an event.

This field resets to an architecturally UNKNOWN value.

EVNTEN, bit [2]

Enables the generation of an event stream from the counter register CNTPCT_EL0:

`0b0` Disables the event stream.

`0b1` Enables the event stream.

This field resets to an architecturally UNKNOWN value.

EL0VCTEN, bit [1]

When HCR_EL2.TGE is 0, this control does not cause any instructions to be trapped.

When HCR_EL2.TGE is 1, traps EL0 accesses to the frequency register and virtual counter register to EL2.

`0b0` EL0 using AArch64: EL0 accesses to the CNTVCT_EL0 are trapped to EL2.

EL0 using AArch64: EL0 accesses to the CNTFRQ_EL0 register are trapped to EL2, if CNTHCTL_EL2.EL0PCTEN is also 0.

EL0 using AArch32: EL0 accesses to the CNTVCT are trapped to EL2.

EL0 using AArch32: EL0 accesses to the CNTFRQ register are trapped to EL2, if CNTHCTL.EL0PCTEN is also 0.

`0b1` This control does not cause any instructions to be trapped.

This field resets to an architecturally UNKNOWN value.

EL0PCTEN, bit [0]

When HCR_EL2.TGE is 0, this control does not cause any instructions to be trapped.

When HCR_EL2.TGE is 1, traps EL0 accesses to the frequency register and physical counter register to EL2.

`0b0` EL0 using AArch64: EL0 accesses to the CNTPCT_EL0 are trapped to EL2.

EL0 using AArch64: EL0 accesses to the CNTFRQ_EL0 register are trapped to EL2, if CNTHCTL_EL2.EL0VCTEN is also 0.

EL0 using AArch32: EL0 accesses to the CNTPCT are trapped to EL2.

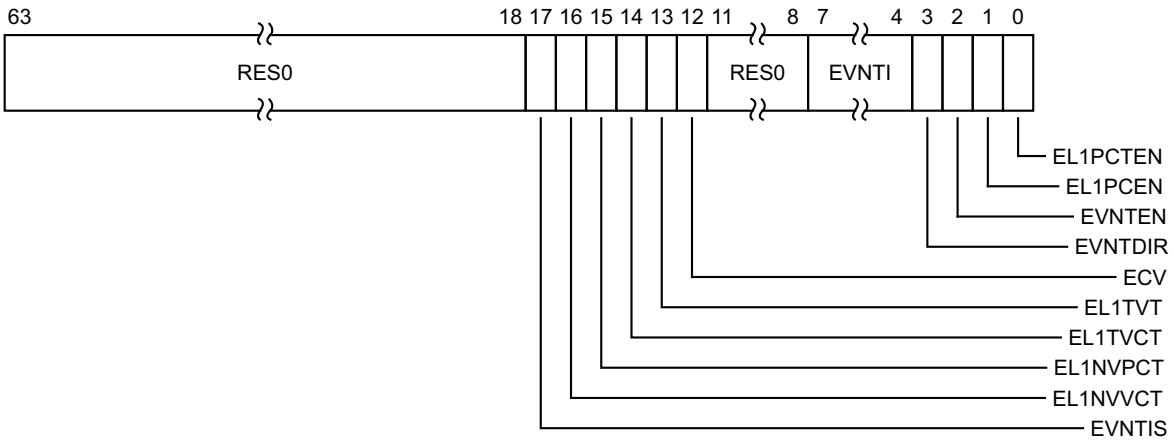
EL0 using AArch32: EL0 accesses to the CNTFRQ and register are trapped to EL2, if CNTHCTL_EL2.EL0VCTEN is also 0.

`0b1` This control does not cause any instructions to be trapped.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:



This format applies in all Armv8.0 implementations, and it also contains a description of the behavior when EL3 is implemented and EL2 is not implemented.

Bits [63:18]

Reserved, RES0.

EVNTIS, bit [17]

Controls the scale of the generation of the event stream.

`0b0` The CNTHCTL_EL2.EVNTI field applies to CNTPCT_EL0[15:0].

`0b1` The CNTHCTL_EL2.EVNTI field applies to CNTPCT_EL0[23:8].

This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.

This field resets to an architecturally UNKNOWN value.

EL1NVVCT, bit [16]

Traps EL1 accesses to the specified EL1 virtual timer registers using the EL02 descriptors to EL2, when EL2 is enabled for the current Security state.

`0b0` This control does not cause any instructions to be trapped.

`0b1` If ((HCR_EL2.E2H==1 && HCR_EL2.TGE==1) || HCR_EL2.NV2==0 || HCR_EL2.NV1==1 || HCR_EL2.NV==0), this control does not cause any instructions to be trapped.

If ((HCR_EL2.E2H==0 || HCR_EL2.TGE==0) && HCR_EL2.NV2==1 && HCR_EL2.NV1==0 && HCR_EL2.NV==1), then EL1 accesses to CNTV_CTL_EL02 and CNTV_CVAL_EL02 are trapped to EL2.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 0 other than for the purpose of a direct read.

This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.

This field resets to an architecturally UNKNOWN value.

EL1NVPCT, bit [15]

Traps EL1 accesses to the specified EL1 physical timer registers using the EL02 descriptors to EL2, when EL2 is enabled for the current Security state.

`0b0` This control does not cause any instructions to be trapped.

`0b1` If ((HCR_EL2.E2H==1 && HCR_EL2.TGE==1) || HCR_EL2.NV2==0 || HCR_EL2.NV1==1 || HCR_EL2.NV==0), this control does not cause any instructions to be trapped.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If (HCR_EL2.E2H==0 || HCR_EL2.TGE==0) && HCR_EL2.NV2==1 && HCR_EL2.NV1==0 && HCR_EL2.NV==1, then EL1 accesses to CNTP_CTL_EL02 and CNTP_CVAL_EL02, are trapped to EL2.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 0 other than for the purpose of a direct read.

This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.

This field resets to an architecturally UNKNOWN value.

EL1TVCT, bit [14]

Traps EL0 and EL1 accesses to the EL1 virtual counter registers to EL2, when EL2 is enabled for the current Security state.

`0b0` This control does not cause any instructions to be trapped.

`0b1` If HCR_EL2.{E2H, TGE} is {1, 1}, this control does not cause any instructions to be trapped.

If HCR_EL2.E2H is 0 or HCR_EL2.TGE is 0, then:

In AArch64 state, traps EL0 and EL1 accesses to CNTVCT_EL0 to EL2, unless they are trapped by CNTKCTL_EL1.EL0VCTEN. In AArch32 state, traps EL0 and EL1 accesses to CNTVCT to EL2, unless they are trapped by CNTKCTL_EL1.EL0VCTEN or CNTKCTL.PL0VCTEN.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 0 other than for the purpose of a direct read.

This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.

This field resets to an architecturally UNKNOWN value.

EL1TVT, bit [13]

Traps EL0 and EL1 accesses to the EL1 virtual timer registers to EL2, when EL2 is enabled for the current Security state.

`0b0` This control does not cause any instructions to be trapped.

`0b1` If HCR_EL2.{E2H, TGE} is {1, 1}, this control does not cause any instructions to be trapped.

If HCR_EL2.E2H is 0 or HCR_EL2.TGE is 0, then:

- In AArch64 state, traps EL0 and EL1 accesses to CNTV_CTL_EL0, CNTV_CVAL_EL0, and CNTV_TVAL_EL0 to EL2, unless they are trapped by CNTKCTL_EL1.EL0VTEN.
- In AArch32 state, traps EL0 and EL1 accesses to CNTV_CTL, CNTV_CVAL, and CNTV_TVAL to EL2, unless they are trapped by CNTKCTL_EL1.EL0VTEN or CNTKCTL.PL0VTEN.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 0 other than for the purpose of a direct read.

This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.

This field resets to an architecturally UNKNOWN value.

ECV, bit [12]

Enables the Enhanced Counter Virtualization functionality registers.

`0b0` Enhanced Counter Virtualization functionality is disabled.

`0b1` When HCR_EL2.{E2H, TGE} == {1, 1} or SCR_EL3.{NS, EEL2} == {0, 0}, then Enhanced Counter Virtualization functionality is disabled.

When SCR_EL3.NS or SCR_EL3.EEL2 are 1, and HCR_EL2.E2H or HCR_EL2.TGE are 0, then Enhanced Counter Virtualization functionality is enabled when EL2 is enabled for the current Security state. This means that:

- An MRS to CNTPCT_EL0 from either EL0 or EL1 that is not trapped will return the value (PCount<63:0> - CNTPOFF_EL2<63:0>).

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The EL1 physical timer interrupt is triggered when ((PCount<63:0> - CNTPOFF_EL2<63:0>) - PCVal<63:0>) is greater than or equal to 0. PCount is the physical count returned when CNTPCT_EL0 is read from EL2 or EL3. PCVal<63:0> is the EL1 physical timer compare value for this timer.

This field resets to an architecturally UNKNOWN value.

Bits [11:8]

Reserved, RES0.

EVNTI, bits [7:4]

Selects which bit (0 to 15) of the counter register CNTPCT_EL0 is the trigger for the event stream generated from that counter, when that stream is enabled.

If ARMv8.6-ECV is implemented, and CNTHCTL_EL2.EVNTIS is 1, this selects which bit (8 to 23) of the counter register CNTPCT_EL0 is the trigger.

This field resets to an architecturally UNKNOWN value.

EVNTDIR, bit [3]

Controls which transition of the counter register CNTPCT_EL0 trigger bit, defined by EVNTI, generates an event when the event stream is enabled:

`0b0` A 0 to 1 transition of the trigger bit triggers an event.

`0b1` A 1 to 0 transition of the trigger bit triggers an event.

This field resets to an architecturally UNKNOWN value.

EVNTEN, bit [2]

Enables the generation of an event stream from the counter register CNTPCT_EL0:

`0b0` Disables the event stream.

`0b1` Enables the event stream.

This field resets to `0`.

EL1PCEN, bit [1]

Traps EL0 and EL1 accesses to the EL1 physical timer registers to EL2 when EL2 is enabled in the current Security state, as follows:

- In AArch64 state, accesses to CNTP_CTL_EL0, CNTP_CVAL_EL0, CNTP_TVAL_EL0 are trapped to EL2, reported using EC syndrome value `0x18`.
- In AArch32 state, MRC or MCR accesses to the following registers are trapped to EL2 reported using EC syndrome value `0x3` and MRRC and MCRR accesses are trapped to EL2, reported using EC syndrome value `0x04`:
  - CNTP_CTL, CNTP_CVAL, CNTP_TVAL.

`0b0` From AArch64 state: EL0 and EL1 accesses to the CNTP_CTL_EL0, CNTP_CVAL_EL0, and CNTP_TVAL_EL0 are trapped to EL2 when EL2 is enabled in the current Security state, unless they are trapped by CNTKCTL_EL1.EL0PTEN.

From AArch32 state: EL0 and EL1 accesses to the CNTP_CTL, CNTP_CVAL, and CNTP_TVAL are trapped to EL2 when EL2 is enabled in the current Security state, unless they are trapped by CNTKCTL_EL1.EL0PTEN or CNTKCTL.PL0PTEN.

`0b1` This control does not cause any instructions to be trapped.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 1 other than for the purpose of a direct read.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

EL1PCTEN, bit [0]

Traps EL0 and EL1 accesses to the EL1 physical counter register to EL2 when EL2 is enabled in the current Security state, as follows:

- In AArch64 state, accesses to CNTPCT_EL0 are trapped to EL2, reported using EC syndrome value `0x18`.
- In AArch32 state, MRRC or MCRR accesses to CNTPCT are trapped to EL2, reported using EC syndrome value `0x04`.

`0b0` From AArch64 state: EL0 and EL1 accesses to the CNTPCT_EL0 are trapped to EL2 when EL2 is enabled in the current Security state, unless they are trapped by CNTKCTL_EL1.EL0PCTEN.

From AArch32 state: EL0 and EL1 accesses to the CNTPCT are trapped to EL2 when EL2 is enabled in the current Security state, unless they are trapped by CNTKCTL_EL1.EL0PCTEN or CNTKCTL.PL0PCTEN.

`0b1` This control does not cause any instructions to be trapped.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 1 other than for the purpose of a direct read.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTHCTL_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic CNTHCTL_EL2 or CNTKCTL_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, CNTHCTL_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return CNTHCTL_EL2;
elsif PSTATE.EL == EL3 then
    return CNTHCTL_EL2;
```

MSR CNTHCTL_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    CNTHCTL_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    CNTHCTL_EL2 = X[t];
```

MRS <Xt>, CNTKCTL_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1110	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    return CNTKCTL_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return CNTHCTL_EL2;
    else
        return CNTKCTL_EL1;
elsif PSTATE.EL == EL3 then
    return CNTKCTL_EL1;
```

MSR CNTKCTL_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1110	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    CNTKCTL_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        CNTHCTL_EL2 = X[t];
    else
        CNTKCTL_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    CNTKCTL_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.8.3 CNTHP_CTL_EL2, Counter-timer Hypervisor Physical Timer Control register

The CNTHP_CTL_EL2 characteristics are:

Purpose

Control register for the EL2 physical timer.

Configurations

AArch64 System register CNTHP_CTL_EL2[31:0] is architecturally mapped to AArch32 System register CNTHP_CTL[31:0].

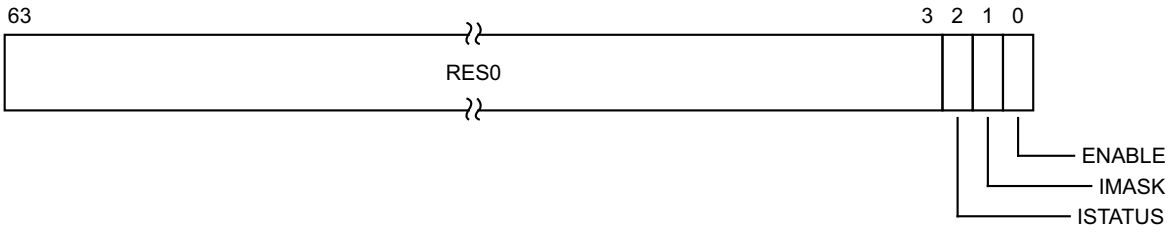
If EL2 is not implemented, this register is RES0 from EL3.

Attributes

CNTHP_CTL_EL2 is a 64-bit register.

Field descriptions

The CNTHP_CTL_EL2 bit assignments are:



Bits [63:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

`0b0` Timer condition is not met.

`0b1` Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

For more information see *Operation of the CompareValue views of the timers* on page D11-2847 and *Operation of the TimerValue views of the timers* on page D11-2848.

This bit is read-only.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

`0b0` Timer interrupt is not masked by the IMASK bit.

`0b1` Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

This field resets to an architecturally UNKNOWN value.

ENABLE, bit [0]

Enables the timer. Permitted values are:

`0b0` Timer disabled.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTHP_TVAL_EL2 continues to count down.

Note

Disabling the output signal might be a power-saving option.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTHP_CTL_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic CNTHP_CTL_EL2 or CNTP_CTL_EL0 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, CNTHP_CTL_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return CNTHP_CTL_EL2;
elsif PSTATE.EL == EL3 then
    return CNTHP_CTL_EL2;
```

MSR CNTHP_CTL_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    CNTHP_CTL_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    CNTHP_CTL_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, CNTP_CTL_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHPS_CTL_EL2;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHP_CTL_EL2;
    else
        return CNTP_CTL_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x180];
    else
        return CNTP_CTL_EL0;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHPS_CTL_EL2;
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        return CNTHP_CTL_EL2;
    else
        return CNTP_CTL_EL0;
elsif PSTATE.EL == EL3 then
    return CNTP_CTL_EL0;
```

MSR CNTP_CTL_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHPS_CTL_EL2 = X[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHP_CTL_EL2 = X[t];
    else
        CNTP_CTL_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x180] = X[t];
    else
        CNTP_CTL_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHPS_CTL_EL2 = X[t];
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        CNTHP_CTL_EL2 = X[t];
    else
        CNTP_CTL_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    CNTP_CTL_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.8.4 CNTHP_CVAL_EL2, Counter-timer Physical Timer CompareValue register (EL2)

The CNTHP_CVAL_EL2 characteristics are:

Purpose

Holds the compare value for the EL2 physical timer.

Configurations

AArch64 System register CNTHP_CVAL_EL2[63:0] is architecturally mapped to AArch32 System register CNTHP_CVAL[63:0].

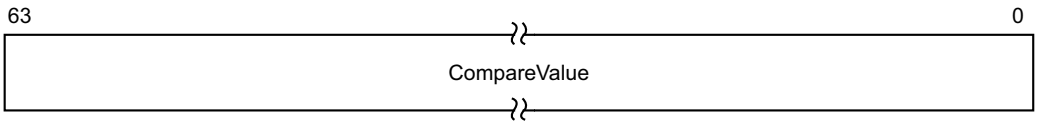
If EL2 is not implemented, this register is RES0 from EL3.

Attributes

CNTHP_CVAL_EL2 is a 64-bit register.

Field descriptions

The CNTHP_CVAL_EL2 bit assignments are:



CompareValue, bits [63:0]

Holds the EL2 physical timer CompareValue.

When CNTHP_CTL_EL2.ENABLE is 1, the timer condition is met when (CNTPCT_EL0 - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTHP_CTL_EL2.ISTATUS is set to 1.
- If CNTHP_CTL_EL2.IMASK is 0, an interrupt is generated.

When CNTHP_CTL_EL2.ENABLE is 0, the timer condition is not met, but CNTPCT_EL0 continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTHP_CVAL_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic CNTHP_CVAL_EL2 or CNTP_CVAL_EL0 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, CNTHP_CVAL_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0010	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return CNTHP_CVAL_EL2;
elsif PSTATE.EL == EL3 then
    return CNTHP_CVAL_EL2;
```

MSR CNTHP_CVAL_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0010	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    CNTHP_CVAL_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    CNTHP_CVAL_EL2 = X[t];
```

MRS <Xt>, CNTP_CVAL_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b010

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHPS_CVAL_EL2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHP_CVAL_EL2;
    else
        return CNTP_CVAL_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x178];
    else
        return CNTP_CVAL_EL0;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHPS_CVAL_EL2;
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        return CNTHP_CVAL_EL2;
    else
        return CNTP_CVAL_EL0;
elsif PSTATE.EL == EL3 then
    return CNTP_CVAL_EL0;
```

MSR CNTP_CVAL_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b010

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHPS_CVAL_EL2 = X[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHP_CVAL_EL2 = X[t];
    else
        CNTP_CVAL_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x178] = X[t];
    else
        CNTP_CVAL_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHPS_CVAL_EL2 = X[t];
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        CNTHP_CVAL_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        CNTP_CVAL_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    CNTP_CVAL_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.8.5 CNTHP_TVAL_EL2, Counter-timer Physical Timer TimerValue register (EL2)

The CNTHP_TVAL_EL2 characteristics are:

Purpose

Holds the timer value for the EL2 physical timer.

Configurations

AArch64 System register CNTHP_TVAL_EL2[31:0] is architecturally mapped to AArch32 System register CNTHP_TVAL[31:0].

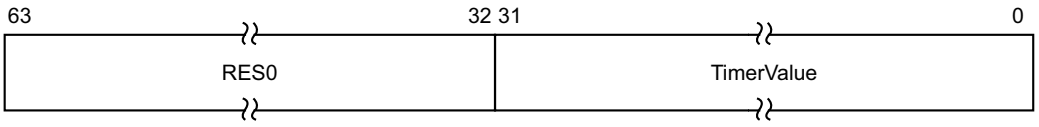
If EL2 is not implemented, this register is RES0 from EL3.

Attributes

CNTHP_TVAL_EL2 is a 64-bit register.

Field descriptions

The CNTHP_TVAL_EL2 bit assignments are:



Bits [63:32]

Reserved, RES0.

TimerValue, bits [31:0]

The TimerValue view of the EL2 physical timer.

On a read of this register:

- If CNTHP_CTL_EL2.ENABLE is 0, the value returned is UNKNOWN.
- If CNTHP_CTL_EL2.ENABLE is 1, the value returned is (CNTHP_CVAL_EL2 - CNTPCT_EL0).

On a write of this register, CNTHP_CVAL_EL2 is set to (CNTPCT_EL0 + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTHP_CTL_EL2.ENABLE is 1, the timer condition is met when (CNTPCT_EL0 - CNTHP_CVAL_EL2) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTHP_CTL_EL2.ISTATUS is set to 1.
- If CNTHP_CTL_EL2.IMASK is 0, an interrupt is generated.

When CNTHP_CTL_EL2.ENABLE is 0, the timer condition is not met, but CNTPCT_EL0 continues to count, so the TimerValue view appears to continue to count down.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTHP_TVAL_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic CNTHP_TVAL_EL2 or CNTP_TVAL_EL0 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, CNTHP_TVAL_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return CNTHP_TVAL_EL2;
elsif PSTATE.EL == EL3 then
    return CNTHP_TVAL_EL2;
```

MSR CNTHP_TVAL_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    CNTHP_TVAL_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    CNTHP_TVAL_EL2 = X[t];
```

MRS <Xt>, CNTP_TVAL_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHPS_TVAL_EL2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHP_TVAL_EL2;
    else
        return CNTP_TVAL_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return CNTP_TVAL_EL0;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHPS_TVAL_EL2;
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        return CNTHP_TVAL_EL2;
    else
        return CNTP_TVAL_EL0;
elsif PSTATE.EL == EL3 then
    return CNTP_TVAL_EL0;
```

MSR CNTP_TVAL_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHPS_TVAL_EL2 = X[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHP_TVAL_EL2 = X[t];
    else
        CNTP_TVAL_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        CNTP_TVAL_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHPS_TVAL_EL2 = X[t];
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        CNTHP_TVAL_EL2 = X[t];
    else
        CNTP_TVAL_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    CNTP_TVAL_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.8.6 CNTHPS_CTL_EL2, Counter-timer Secure Physical Timer Control register (EL2)

The CNTHPS_CTL_EL2 characteristics are:

Purpose

Control register for the Secure EL2 physical timer.

Configurations

AArch64 System register CNTHPS_CTL_EL2[31:0] is architecturally mapped to AArch32 System register CNTHPS_CTL[31:0].

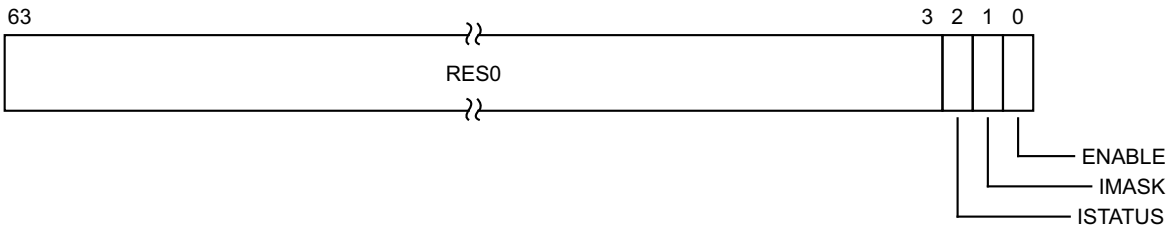
This register is present only when ARMv8.4-SecEL2 is implemented. Otherwise, direct accesses to CNTHPS_CTL_EL2 are UNDEFINED.

Attributes

CNTHPS_CTL_EL2 is a 64-bit register.

Field descriptions

The CNTHPS_CTL_EL2 bit assignments are:



Bits [63:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

`0b0` Timer condition is not met.

`0b1` Timer condition is met.

When the value of the CNTHPS_CTL_EL2.ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the CNTHPS_CTL_EL2.ENABLE bit is 0, the ISTATUS field is UNKNOWN.

For more information see *Operation of the CompareValue views of the timers* on page D11-2847 and *Operation of the TimerValue views of the timers* on page D11-2848.

This bit is read-only.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

`0b0` Timer interrupt is not masked by the IMASK bit.

`0b1` Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ENABLE, bit [0]

Enables the timer. Permitted values are:

`0b0` Timer disabled.

`0b1` Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTHPS_TVAL_EL2 continues to count down.

Note

Disabling the output signal might be a power-saving option.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTHPS_CTL_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, CNTHPS_CTL_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0101	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && SCR_EL3.NS == '1' then
        UNDEFINED;
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && SCR_EL3.NS == '1' then
        UNDEFINED;
    else
        return CNTHPS_CTL_EL2;
elsif PSTATE.EL == EL3 then
    if SCR_EL3.EEL2 == '0' then
        UNDEFINED;
    else
        return CNTHPS_CTL_EL2;
```

MSR CNTHPS_CTL_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0101	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && SCR_EL3.NS == '1' then
        UNDEFINED;
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if HaveEL(EL3) && SCR_EL3.NS == '1' then
        UNDEFINED;
    else
        CNTHPS_CTL_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    if SCR_EL3.EEL2 == '0' then
        UNDEFINED;
    else
        CNTHPS_CTL_EL2 = X[t];
```

MRS <Xt>, CNTP_CTL_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHPS_CTL_EL2;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHP_CTL_EL2;
    else
        return CNTP_CTL_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x180];
    else
        return CNTP_CTL_EL0;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHPS_CTL_EL2;
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        return CNTHP_CTL_EL2;
    else
        return CNTP_CTL_EL0;
elsif PSTATE.EL == EL3 then
    return CNTP_CTL_EL0;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR CNTP_CTL_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHPS_CTL_EL2 = X[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHP_CTL_EL2 = X[t];
    else
        CNTP_CTL_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x180] = X[t];
    else
        CNTP_CTL_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHPS_CTL_EL2 = X[t];
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        CNTHP_CTL_EL2 = X[t];
    else
        CNTP_CTL_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    CNTP_CTL_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.8.7 CNTHPS_CVAL_EL2, Counter-timer Secure Physical Timer CompareValue register (EL2)

The CNTHPS_CVAL_EL2 characteristics are:

Purpose

Holds the compare value for the Secure EL2 physical timer.

Configurations

AArch64 System register CNTHPS_CVAL_EL2[31:0] is architecturally mapped to AArch32 System register CNTHPS_CVAL[31:0].

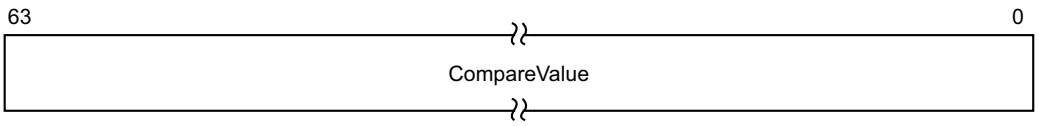
This register is present only when ARMv8.4-SecEL2 is implemented. Otherwise, direct accesses to CNTHPS_CVAL_EL2 are UNDEFINED.

Attributes

CNTHPS_CVAL_EL2 is a 64-bit register.

Field descriptions

The CNTHPS_CVAL_EL2 bit assignments are:



CompareValue, bits [63:0]

Holds the EL2 physical timer CompareValue.

When CNTHPS_CTL_EL2.ENABLE is 1, the timer condition is met when (CNTPCT_EL0 - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTHPS_CTL_EL2.ISTATUS is set to 1.
- If CNTHPS_CTL_EL2.IMASK is 0, an interrupt is generated.

When CNTHPS_CTL_EL2.ENABLE is 0, the timer condition is not met, but CNTPCT_EL0 continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTHPS_CVAL_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, CNTHPS_CVAL_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0101	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && SCR_EL3.NS == '1' then
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && SCR_EL3.NS == '1' then
        UNDEFINED;
    else
        return CNTHPS_CVAL_EL2;
elsif PSTATE.EL == EL3 then
    if SCR_EL3.EEL2 == '0' then
        UNDEFINED;
    else
        return CNTHPS_CVAL_EL2;
```

MSR CNTHPS_CVAL_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0101	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && SCR_EL3.NS == '1' then
        UNDEFINED;
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && SCR_EL3.NS == '1' then
        UNDEFINED;
    else
        CNTHPS_CVAL_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    if SCR_EL3.EEL2 == '0' then
        UNDEFINED;
    else
        CNTHPS_CVAL_EL2 = X[t];
```

MRS <Xt>, CNTP_CVAL_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b010

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHPS_CVAL_EL2;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHP_CVAL_EL2;
    else
        return CNTP_CVAL_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x178];
    else
        return CNTP_CVAL_EL0;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHPS_CVAL_EL2;
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        return CNTHP_CVAL_EL2;
    else
        return CNTP_CVAL_EL0;
elsif PSTATE.EL == EL3 then
    return CNTP_CVAL_EL0;
```

MSR CNTP_CVAL_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b010

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHPS_CVAL_EL2 = X[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHP_CVAL_EL2 = X[t];
    else
        CNTP_CVAL_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x178] = X[t];
    else
        CNTP_CVAL_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        CNTHPS_CVAL_EL2 = X[t];
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        CNTHP_CVAL_EL2 = X[t];
    else
        CNTP_CVAL_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    CNTP_CVAL_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.8.8 CNTHPS_TVAL_EL2, Counter-timer Secure Physical Timer TimerValue register (EL2)

The CNTHPS_TVAL_EL2 characteristics are:

Purpose

Holds the timer value for the Secure EL2 physical timer.

Configurations

AArch64 System register CNTHPS_TVAL_EL2[31:0] is architecturally mapped to AArch32 System register CNTHPS_TVAL[31:0].

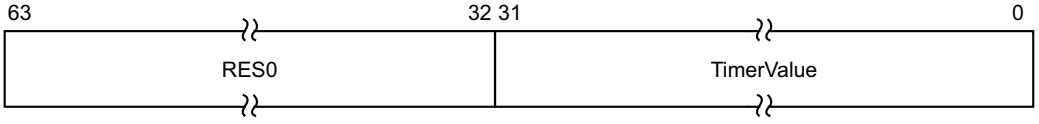
This register is present only when ARMv8.4-SecEL2 is implemented. Otherwise, direct accesses to CNTHPS_TVAL_EL2 are UNDEFINED.

Attributes

CNTHPS_TVAL_EL2 is a 64-bit register.

Field descriptions

The CNTHPS_TVAL_EL2 bit assignments are:



Bits [63:32]

Reserved, RES0.

TimerValue, bits [31:0]

The TimerValue view of the EL2 physical timer.

On a read of this register:

- If CNTHPS_CTL_EL2.ENABLE is 0, the value returned is UNKNOWN.
- If CNTHPS_CTL_EL2.ENABLE is 1, the value returned is (CNTHPS_CVAL_EL2 - CNTPCT_EL0).

On a write of this register, CNTHPS_CVAL_EL2 is set to (CNTPCT_EL0 + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTHPS_CTL_EL2.ENABLE is 1, the timer condition is met when (CNTPCT_EL0 - CNTHPS_CVAL_EL2) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTHPS_CTL_EL2.ISTATUS is set to 1.
- If CNTHPS_CTL_EL2.IMASK is 0, an interrupt is generated.

When CNTHPS_CTL_EL2.ENABLE is 0, the timer condition is not met, but CNTPCT_EL0 continues to count, so the TimerValue view appears to continue to count down.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTHPS_TVAL_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, CNTHPS_TVAL_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0101	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && SCR_EL3.NS == '1' then
        UNDEFINED;
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && SCR_EL3.NS == '1' then
        UNDEFINED;
    else
        return CNTHPS_TVAL_EL2;
elsif PSTATE.EL == EL3 then
    if SCR_EL3.EEL2 == '0' then
        UNDEFINED;
    else
        return CNTHPS_TVAL_EL2;
```

MSR CNTHPS_TVAL_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0101	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && SCR_EL3.NS == '1' then
        UNDEFINED;
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && SCR_EL3.NS == '1' then
        UNDEFINED;
    else
        CNTHPS_TVAL_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    if SCR_EL3.EEL2 == '0' then
        UNDEFINED;
    else
        CNTHPS_TVAL_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, CNTP_TVAL_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHPS_TVAL_EL2;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHP_TVAL_EL2;
    else
        return CNTP_TVAL_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return CNTP_TVAL_EL0;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHPS_TVAL_EL2;
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        return CNTHP_TVAL_EL2;
    else
        return CNTP_TVAL_EL0;
elsif PSTATE.EL == EL3 then
    return CNTP_TVAL_EL0;
```

MSR CNTP_TVAL_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHPS_TVAL_EL2 = X[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHP_TVAL_EL2 = X[t];
    else
        CNTP_TVAL_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        CNTP_TVAL_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHPS_TVAL_EL2 = X[t];
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        CNTHP_TVAL_EL2 = X[t];
    else
        CNTP_TVAL_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    CNTP_TVAL_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.8.9 CNTHV_CTL_EL2, Counter-timer Virtual Timer Control register (EL2)

The CNTHV_CTL_EL2 characteristics are:

Purpose

Control register for the EL2 virtual timer.

Configurations

AArch64 System register CNTHV_CTL_EL2[31:0] is architecturally mapped to AArch32 System register CNTHV_CTL[31:0].

This register is present only when ARMv8.1-VHE is implemented. Otherwise, direct accesses to CNTHV_CTL_EL2 are UNDEFINED.

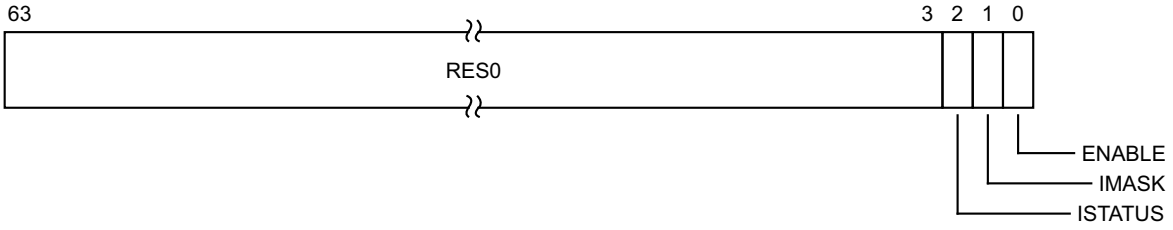
If EL2 is not implemented, this register is RES0 from EL3.

Attributes

CNTHV_CTL_EL2 is a 64-bit register.

Field descriptions

The CNTHV_CTL_EL2 bit assignments are:



Bits [63:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

`0b0` Timer condition is not met.

`0b1` Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

For more information see *Operation of the CompareValue views of the timers* on page D11-2847 and *Operation of the TimerValue views of the timers* on page D11-2848.

This bit is read-only.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

`0b0` Timer interrupt is not masked by the IMASK bit.

`0b1` Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ENABLE, bit [0]

Enables the timer. Permitted values are:

`0b0` Timer disabled.

`0b1` Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTHV_TVAL_EL2 continues to count down.

Note

Disabling the output signal might be a power-saving option.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTHV_CTL_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic CNTHV_CTL_EL2 or CNTV_CTL_EL0 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, CNTHV_CTL_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0011	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return CNTHV_CTL_EL2;
elsif PSTATE.EL == EL3 then
    return CNTHV_CTL_EL2;
```

MSR CNTHV_CTL_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0011	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    CNTHV_CTL_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    CNTHV_CTL_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, CNTV_CTL_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHVS_CTL_EL2;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHV_CTL_EL2;
    else
        return CNTV_CTL_EL0;
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x170];
    else
        return CNTV_CTL_EL0;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHVS_CTL_EL2;
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        return CNTHV_CTL_EL2;
    else
        return CNTV_CTL_EL0;
elsif PSTATE.EL == EL3 then
    return CNTV_CTL_EL0;
```

MSR CNTV_CTL_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHVS_CTL_EL2 = X[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHV_CTL_EL2 = X[t];
    else
        CNTV_CTL_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x170] = X[t];
    else
        CNTV_CTL_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHVS_CTL_EL2 = X[t];
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        CNTHV_CTL_EL2 = X[t];
    else
        CNTV_CTL_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    CNTV_CTL_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.8.10 CNTHV_CVAL_EL2, Counter-timer Virtual Timer CompareValue register (EL2)

The CNTHV_CVAL_EL2 characteristics are:

Purpose

Holds the compare value for the EL2 virtual timer.

Configurations

AArch64 System register CNTHV_CVAL_EL2[63:0] is architecturally mapped to AArch32 System register CNTHV_CVAL[63:0].

This register is present only when ARMv8.1-VHE is implemented. Otherwise, direct accesses to CNTHV_CVAL_EL2 are UNDEFINED.

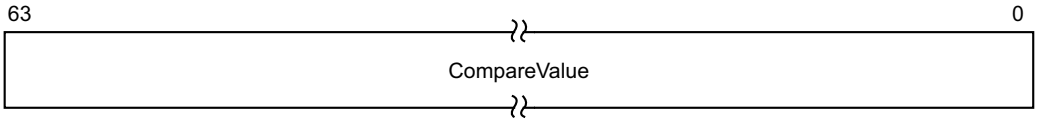
If EL2 is not implemented, this register is RES0 from EL3.

Attributes

CNTHV_CVAL_EL2 is a 64-bit register.

Field descriptions

The CNTHV_CVAL_EL2 bit assignments are:



CompareValue, bits [63:0]

Holds the EL2 virtual timer CompareValue.

When CNTHV_CTL_EL2.ENABLE is 1, the timer condition is met when (CNTVCT_EL0 - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTHV_CTL_EL2.ISTATUS is set to 1.
- If CNTHV_CTL_EL2.IMASK is 0, an interrupt is generated.

When CNTHV_CTL_EL2.ENABLE is 0, the timer condition is not met, but CNTVCT_EL0 continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTHV_CVAL_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic CNTHV_CVAL_EL2 or CNTV_CVAL_EL0 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, CNTHV_CVAL_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0011	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return CNTHV_CVAL_EL2;
elsif PSTATE.EL == EL3 then
    return CNTHV_CVAL_EL2;
```

MSR CNTHV_CVAL_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0011	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    CNTHV_CVAL_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    CNTHV_CVAL_EL2 = X[t];
```

MRS <Xt>, CNTV_CVAL_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b010

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHVS_CVAL_EL2;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHV_CVAL_EL2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        return CNTV_CVAL_EL0;
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x168];
    else
        return CNTV_CVAL_EL0;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHVS_CVAL_EL2;
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        return CNTHV_CVAL_EL2;
    else
        return CNTV_CVAL_EL0;
elsif PSTATE.EL == EL3 then
    return CNTV_CVAL_EL0;
```

MSR CNTV_CVAL_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b010

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHVS_CVAL_EL2 = X[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHV_CVAL_EL2 = X[t];
    else
        CNTV_CVAL_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x168] = X[t];
    else
        CNTV_CVAL_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHVS_CVAL_EL2 = X[t];
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        CNTHV_CVAL_EL2 = X[t];
    else
        CNTV_CVAL_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    CNTV_CVAL_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.8.11 CNTHV_TVAL_EL2, Counter-timer Virtual Timer TimerValue Register (EL2)

The CNTHV_TVAL_EL2 characteristics are:

Purpose

Holds the timer value for the EL2 virtual timer.

Configurations

AArch64 System register CNTHV_TVAL_EL2[31:0] is architecturally mapped to AArch32 System register CNTHV_TVAL[31:0].

This register is present only when ARMv8.1-VHE is implemented. Otherwise, direct accesses to CNTHV_TVAL_EL2 are UNDEFINED.

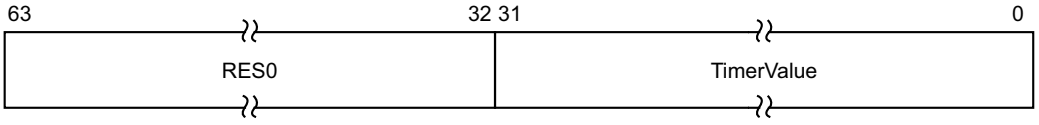
If EL2 is not implemented, this register is RES0 from EL3.

Attributes

CNTHV_TVAL_EL2 is a 64-bit register.

Field descriptions

The CNTHV_TVAL_EL2 bit assignments are:



Bits [63:32]

Reserved, RES0.

TimerValue, bits [31:0]

The TimerValue view of the EL2 virtual timer.

On a read of this register:

- If CNTHV_CTL_EL2.ENABLE is 0, the value returned is UNKNOWN.
- If CNTHV_CTL_EL2.ENABLE is 1, the value returned is (CNTHV_CVAL_EL2 - CNTVCT_EL0).

On a write of this register, CNTHV_CVAL_EL2 is set to (CNTVCT_EL0 + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTHV_CTL_EL2.ENABLE is 1, the timer condition is met when (CNTVCT_EL0 - CNTHV_CVAL_EL2) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTHV_CTL_EL2.ISTATUS is set to 1.
- If CNTHV_CTL_EL2.IMASK is 0, an interrupt is generated.

When CNTHV_CTL_EL2.ENABLE is 0, the timer condition is not met, but CNTVCT_EL0 continues to count, so the TimerValue view appears to continue to count down.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTHV_TVAL_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic CNTHV_TVAL_EL2 or CNTV_TVAL_EL0 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, CNTHV_TVAL_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return CNTHV_TVAL_EL2;
elsif PSTATE.EL == EL3 then
    return CNTHV_TVAL_EL2;
```

MSR CNTHV_TVAL_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    CNTHV_TVAL_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    CNTHV_TVAL_EL2 = X[t];
```

MRS <Xt>, CNTV_TVAL_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHVS_TVAL_EL2;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHV_TVAL_EL2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        return CNTV_TVAL_EL0;
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return CNTV_TVAL_EL0;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHVS_TVAL_EL2;
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        return CNTHV_TVAL_EL2;
    else
        return CNTV_TVAL_EL0;
elsif PSTATE.EL == EL3 then
    return CNTV_TVAL_EL0;
```

MSR CNTV_TVAL_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHVS_TVAL_EL2 = X[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHV_TVAL_EL2 = X[t];
    else
        CNTV_TVAL_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        CNTV_TVAL_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHVS_TVAL_EL2 = X[t];
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        CNTHV_TVAL_EL2 = X[t];
    else
        CNTV_TVAL_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    CNTV_TVAL_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.8.12 CNTHVS_CTL_EL2, Counter-timer Secure Virtual Timer Control register (EL2)

The CNTHVS_CTL_EL2 characteristics are:

Purpose

Control register for the Secure EL2 virtual timer.

Configurations

AArch64 System register CNTHVS_CTL_EL2[31:0] is architecturally mapped to AArch32 System register CNTHVS_CTL[31:0].

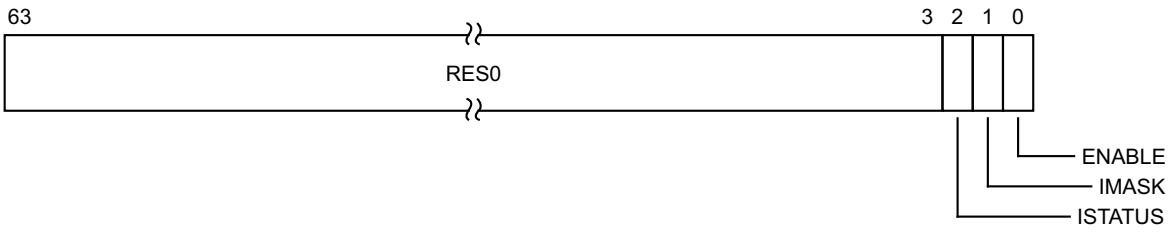
This register is present only when ARMv8.4-SecEL2 is implemented. Otherwise, direct accesses to CNTHVS_CTL_EL2 are UNDEFINED.

Attributes

CNTHVS_CTL_EL2 is a 64-bit register.

Field descriptions

The CNTHVS_CTL_EL2 bit assignments are:



Bits [63:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

`0b0` Timer condition is not met.

`0b1` Timer condition is met.

When the value of the CNTHVS_CTL_EL2.ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

For more information see *Operation of the CompareValue views of the timers* on page D11-2847 and *Operation of the TimerValue views of the timers* on page D11-2848.

This bit is read-only.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

`0b0` Timer interrupt is not masked by the IMASK bit.

`0b1` Timer interrupt is masked by the IMASK bit.

For more information, see the description of the CNTHVS_CTL_EL2.ISTATUS bit.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ENABLE, bit [0]

Enables the timer. Permitted values are:

`0b0` Timer disabled.

`0b1` Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTHVS_TVAL_EL2 continues to count down.

Note

Disabling the output signal might be a power-saving option.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTHVS_CTL_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, CNTHVS_CTL_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0100	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && SCR_EL3.NS == '1' then
        UNDEFINED;
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && SCR_EL3.NS == '1' then
        UNDEFINED;
    else
        return CNTHVS_CTL_EL2;
elsif PSTATE.EL == EL3 then
    if SCR_EL3.EEL2 == '0' then
        UNDEFINED;
    else
        return CNTHVS_CTL_EL2;
```

MSR CNTHVS_CTL_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0100	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && SCR_EL3.NS == '1' then
        UNDEFINED;
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if HaveEL(EL3) && SCR_EL3.NS == '1' then
        UNDEFINED;
    else
        CNTHVS_CTL_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    if SCR_EL3.EEL2 == '0' then
        UNDEFINED;
    else
        CNTHVS_CTL_EL2 = X[t];
```

MRS <Xt>, CNTV_CTL_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHVS_CTL_EL2;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHV_CTL_EL2;
    else
        return CNTV_CTL_EL0;
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x170];
    else
        return CNTV_CTL_EL0;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHVS_CTL_EL2;
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        return CNTHV_CTL_EL2;
    else
        return CNTV_CTL_EL0;
elsif PSTATE.EL == EL3 then
    return CNTV_CTL_EL0;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR CNTV_CTL_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHVS_CTL_EL2 = X[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHV_CTL_EL2 = X[t];
    else
        CNTV_CTL_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x170] = X[t];
    else
        CNTV_CTL_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHVS_CTL_EL2 = X[t];
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        CNTHV_CTL_EL2 = X[t];
    else
        CNTV_CTL_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    CNTV_CTL_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.8.13 CNTHVS_CVAL_EL2, Counter-timer Secure Virtual Timer CompareValue register (EL2)

The CNTHVS_CVAL_EL2 characteristics are:

Purpose

Holds the compare value for the Secure EL2 virtual timer.

Configurations

AArch64 System register CNTHVS_CVAL_EL2[63:0] is architecturally mapped to AArch32 System register CNTHVS_CVAL[63:0].

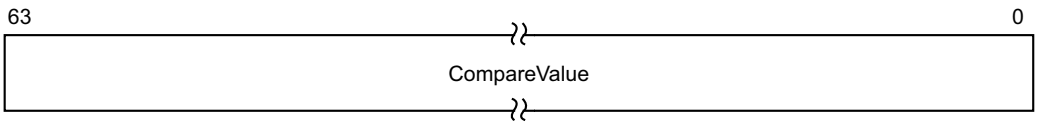
This register is present only when ARMv8.4-SecEL2 is implemented. Otherwise, direct accesses to CNTHVS_CVAL_EL2 are UNDEFINED.

Attributes

CNTHVS_CVAL_EL2 is a 64-bit register.

Field descriptions

The CNTHVS_CVAL_EL2 bit assignments are:



CompareValue, bits [63:0]

Holds the Secure EL2 virtual timer CompareValue.

When CNTHVS_CTL_EL2.ENABLE is 1, the timer condition is met when (CNTVCT_EL0 - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTHVS_CTL_EL2.ISTATUS is set to 1.
- If CNTHVS_CTL_EL2.IMASK is 0, an interrupt is generated.

When CNTHVS_CTL_EL2.ENABLE is 0, the timer condition is not met, but CNTVCT_EL0 continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTHVS_CVAL_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, CNTHVS_CVAL_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0100	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && SCR_EL3.NS == '1' then
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && SCR_EL3.NS == '1' then
        UNDEFINED;
    else
        return CNTHVS_CVAL_EL2;
elsif PSTATE.EL == EL3 then
    if SCR_EL3.EEL2 == '0' then
        UNDEFINED;
    else
        return CNTHVS_CVAL_EL2;
```

MSR CNTHVS_CVAL_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0100	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && SCR_EL3.NS == '1' then
        UNDEFINED;
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && SCR_EL3.NS == '1' then
        UNDEFINED;
    else
        CNTHVS_CVAL_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    if SCR_EL3.EEL2 == '0' then
        UNDEFINED;
    else
        CNTHVS_CVAL_EL2 = X[t];
```

MRS <Xt>, CNTV_CVAL_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b010

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        return CNTHVS_CVAL_EL2;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHV_CVAL_EL2;
    else
        return CNTV_CVAL_EL0;
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x168];
    else
        return CNTV_CVAL_EL0;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHVS_CVAL_EL2;
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        return CNTHV_CVAL_EL2;
    else
        return CNTV_CVAL_EL0;
elsif PSTATE.EL == EL3 then
    return CNTV_CVAL_EL0;
```

MSR CNTV_CVAL_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b010

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHVS_CVAL_EL2 = X[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHV_CVAL_EL2 = X[t];
    else
        CNTV_CVAL_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x168] = X[t];
    else
        CNTV_CVAL_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHVS_CVAL_EL2 = X[t];
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        CNTHV_CVAL_EL2 = X[t];
    else
        CNTV_CVAL_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    CNTV_CVAL_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.8.14 CNTHVS_TVAL_EL2, Counter-timer Secure Virtual Timer TimerValue register (EL2)

The CNTHVS_TVAL_EL2 characteristics are:

Purpose

Holds the timer value for the Secure EL2 virtual timer.

Configurations

AArch64 System register CNTHVS_TVAL_EL2[31:0] is architecturally mapped to AArch32 System register CNTHVS_TVAL[31:0].

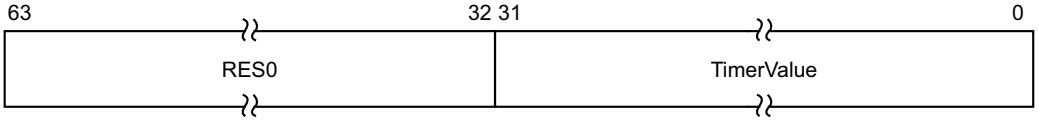
This register is present only when ARMv8.4-SecEL2 is implemented. Otherwise, direct accesses to CNTHVS_TVAL_EL2 are UNDEFINED.

Attributes

CNTHVS_TVAL_EL2 is a 64-bit register.

Field descriptions

The CNTHVS_TVAL_EL2 bit assignments are:



Bits [63:32]

Reserved, RES0.

TimerValue, bits [31:0]

The TimerValue view of the EL2 virtual timer.

On a read of this register:

- If CNTHVS_CTL_EL2.ENABLE is 0, the value returned is UNKNOWN.
- If CNTHVS_CTL_EL2.ENABLE is 1, the value returned is (CNTHVS_CVAL_EL2 - CNTVCT_EL0).

On a write of this register, CNTHVS_CVAL_EL2 is set to (CNTVCT_EL0 + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTHVS_CTL_EL2.ENABLE is 1, the timer condition is met when ((CNTVCT_EL0 - CNTHVS_CVAL_EL2) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTHVS_CTL_EL2.ISTATUS is set to 1.
- If CNTHVS_CTL_EL2.IMASK is 0, an interrupt is generated.

When CNTHVS_CTL_EL2.ENABLE is 0, the timer condition is not met, but CNTVCT_EL0 continues to count, so the TimerValue view appears to continue to count down.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTHVS_TVAL_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, CNTHVS_TVAL_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0100	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && SCR_EL3.NS == '1' then
        UNDEFINED;
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && SCR_EL3.NS == '1' then
        UNDEFINED;
    else
        return CNTHVS_TVAL_EL2;
elsif PSTATE.EL == EL3 then
    if SCR_EL3.EEL2 == '0' then
        UNDEFINED;
    else
        return CNTHVS_TVAL_EL2;
```

MSR CNTHVS_TVAL_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0100	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && SCR_EL3.NS == '1' then
        UNDEFINED;
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && SCR_EL3.NS == '1' then
        UNDEFINED;
    else
        CNTHVS_TVAL_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    if SCR_EL3.EEL2 == '0' then
        UNDEFINED;
    else
        CNTHVS_TVAL_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, CNTV_TVAL_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHVS_TVAL_EL2;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHV_TVAL_EL2;
    else
        return CNTV_TVAL_EL0;
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return CNTV_TVAL_EL0;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHVS_TVAL_EL2;
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        return CNTHV_TVAL_EL2;
    else
        return CNTV_TVAL_EL0;
elsif PSTATE.EL == EL3 then
    return CNTV_TVAL_EL0;
```

MSR CNTV_TVAL_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        CNTHVS_TVAL_EL2 = X[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHV_TVAL_EL2 = X[t];
    else
        CNTV_TVAL_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        CNTV_TVAL_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHVS_TVAL_EL2 = X[t];
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        CNTHV_TVAL_EL2 = X[t];
    else
        CNTV_TVAL_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    CNTV_TVAL_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.8.15 CNTKCTL_EL1, Counter-timer Kernel Control register

The CNTKCTL_EL1 characteristics are:

Purpose

When ARMv8.1-VHE is not implemented, or when HCR_EL2.{E2H, TGE} is not {1, 1}, this register controls the generation of an event stream from the virtual counter, and access from EL0 to the physical counter, virtual counter, EL1 physical timers, and the virtual timer.

When ARMv8.1-VHE is implemented and HCR_EL2.{E2H, TGE} is {1, 1}, this register does not cause any event stream from the virtual counter to be generated, and does not control access to the counters and timers. The access to counters and timers at EL0 is controlled by CNTHCTL_EL2.

Configurations

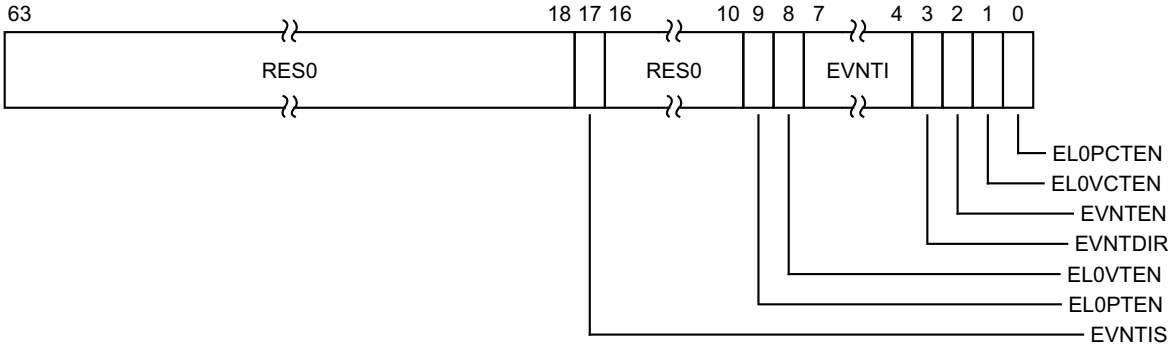
AArch64 System register CNTKCTL_EL1[31:0] is architecturally mapped to AArch32 System register CNTKCTL[31:0].

Attributes

CNTKCTL_EL1 is a 64-bit register.

Field descriptions

The CNTKCTL_EL1 bit assignments are:



Bits [63:18]

Reserved, RES0.

EVNTIS, bit [17]

When ARMv8.6-ECV is implemented:

Controls the scale of the generation of the event stream.

`0b0` The CNTKCTL_EL1.EVNTI field applies to CNTVCT_EL0[15:0].

`0b1` The CNTKCTL_EL1.EVNTI field applies to CNTVCT_EL0[23:8].

This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [16:10]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

EL0PTEN, bit [9]

Traps EL0 accesses to the physical timer registers to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, as follows:

- In AArch64 state, the following registers are trapped, reported using EC syndrome value `0x18`:
  - CNTP_CTL_EL0, CNTP_CVAL_EL0, and CNTP_TVAL_EL0.
- In AArch32 state, MRC and MCR accesses to the following registers are trapped, reported using EC syndrome value `0x03`, MRRC and MCRR accesses are trapped, reported using EC syndrome value `0x04`:
  - CNTP_CTL, CNTP_CVAL, CNTP_TVAL.

`0b0` EL0 accesses to the physical timer registers are trapped to EL1.

`0b1` This control does not cause any instructions to be trapped.

When ARMv8.1-VHE is implemented and HCR_EL2.{E2H, TGE} is {1, 1}, this control does not cause any instructions to be trapped.

This field resets to an architecturally UNKNOWN value.

EL0VTEN, bit [8]

Traps EL0 accesses to the virtual timer registers to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, as follows:

- In AArch64 state, accesses to the following registers are trapped, reported using EC syndrome value `0x18`:
  - CNTV_CTL_EL0, CNTV_CVAL_EL0, and CNTV_TVAL_EL0.
- In AArch32 state, MRC and MCR accesses to the following registers are trapped and reported using EC syndrome value `0x03`, MRRC and MCRR accesses are trapped using EC syndrome value `0x04`:
  - CNTV_CTL, CNTV_CVAL, and CNTV_TVAL.

`0b0` EL0 accesses to the virtual timer registers are trapped.

`0b1` This control does not cause any instructions to be trapped.

When ARMv8.1-VHE is implemented and HCR_EL2.{E2H, TGE} is {1, 1}, this control does not cause any instructions to be trapped.

This field resets to an architecturally UNKNOWN value.

EVNTI, bits [7:4]

Selects which bit (0 to 15) of the counter register CNTVCT_EL0 is the trigger for the event stream generated from that counter, when that stream is enabled.

If ARMv8.6-ECV is implemented, and CNTKCTL_EL1.EVNTIS is 1, this selects which bit (8 to 23) of the counter register CNTVCT_EL0 is the trigger.

This field resets to an architecturally UNKNOWN value.

EVNTDIR, bit [3]

Controls which transition of the counter register CNTVCT_EL0 trigger bit, defined by EVNTI, generates an event when the event stream is enabled:

`0b0` A 0 to 1 transition of the trigger bit triggers an event.

`0b1` A 1 to 0 transition of the trigger bit triggers an event.

This field resets to an architecturally UNKNOWN value.

EVNTEN, bit [2]

When ARMv8.1-VHE is not implemented, or when HCR_EL2.{E2H, TGE} is not {1, 1}, enables the generation of an event stream from the counter register CNTVCT_EL0:

`0b0` Disables the event stream.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` Enables the event stream.

When ARMv8.1-VHE is implemented and HCR_EL2.{E2H, TGE} is {1, 1}, this control does not enable the event stream.

This field resets to `0`.

EL0VCTEN, bit [1]

Traps EL0 accesses to the frequency register and virtual counter register to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, as follows:

- In AArch64 state, accesses to the following registers are trapped and reported using EC syndrome value `0x18`:
  - CNTVCT_EL0 and if CNTKCTL_EL1.EL0PCTEN is 0, CNTFRQ_EL0.
- In AArch32 state, MRC and MCR accesses to the following registers are trapped and reported using EC syndrome value `0x03`, MRRC and MCRR accesses are trapped and reported using EC syndrome value `0x04`:
  - CNTVCT and if CNTKCTL_EL1.EL0PCTEN is 0, CNTFRQ.

`0b0` EL0 accesses to the frequency register and virtual counter registers are trapped.

`0b1` This control does not cause any instructions to be trapped.

When ARMv8.1-VHE is implemented and HCR_EL2.{E2H, TGE} is {1, 1}, this control does not cause any instructions to be trapped.

This field resets to an architecturally UNKNOWN value.

EL0PCTEN, bit [0]

Traps EL0 accesses to the frequency register and physical counter register to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, as follows:

- In AArch64 state, the following registers are trapped, reported using EC syndrome value `0x18`:
  - CNTPCT_EL0 and if CNTKCTL_EL1.EL0VCTEN is 0, CNTFRQ_EL0.
- In AArch32 state, MCR or MRC accesses the following registers are trapped, reported using EC syndrome value `0x03`, MCRR or MRRC accesses are trapped and reported using EC syndrome value `0x04`:
  - CNTPCT and if CNTKCTL_EL1.EL0VCTEN is 0, CNTFRQ.

`0b0` EL0 accesses to the frequency register and physical counter register are trapped.

`0b1` This control does not cause any instructions to be trapped.

When ARMv8.1-VHE is implemented and HCR_EL2.{E2H, TGE} is {1, 1}, this control does not cause any instructions to be trapped.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTKCTL_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic CNTKCTL_EL1 or CNTKCTL_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, CNTKCTL_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1110	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    return CNTKCTL_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return CNTHCTL_EL2;
    else
        return CNTKCTL_EL1;
elsif PSTATE.EL == EL3 then
    return CNTKCTL_EL1;
```

MSR CNTKCTL_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1110	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    CNTKCTL_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        CNTHCTL_EL2 = X[t];
    else
        CNTKCTL_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    CNTKCTL_EL1 = X[t];
```

MRS <Xt>, CNTKCTL_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b1110	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return CNTKCTL_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        return CNTKCTL_EL1;
    else
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR CNTKCTL_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b1110	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        CNTKCTL_EL1 = X[t];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        CNTKCTL_EL1 = X[t];
    else
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.8.16 CNTP_CTL_EL0, Counter-timer Physical Timer Control register

The CNTP_CTL_EL0 characteristics are:

Purpose

Control register for the EL1 physical timer.

Configurations

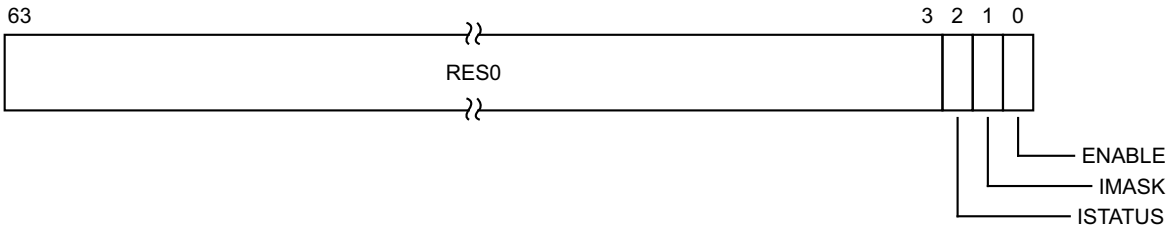
AArch64 System register CNTP_CTL_EL0[31:0] is architecturally mapped to AArch32 System register CNTP_CTL[31:0].

Attributes

CNTP_CTL_EL0 is a 64-bit register.

Field descriptions

The CNTP_CTL_EL0 bit assignments are:



Bits [63:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

`0b0` Timer condition is not met.

`0b1` Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

For more information see *Operation of the CompareValue views of the timers* on page D11-2847 and *Operation of the TimerValue views of the timers* on page D11-2848.

This bit is read-only.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

`0b0` Timer interrupt is not masked by the IMASK bit.

`0b1` Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

This field resets to an architecturally UNKNOWN value.

ENABLE, bit [0]

Enables the timer. Permitted values are:

`0b0` Timer disabled.

`0b1` Timer enabled.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTP_TVAL_EL0 continues to count down.

Note

Disabling the output signal might be a power-saving option.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTP_CTL_EL0

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic CNTP_CTL_EL0 or CNTP_CTL_EL02 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, CNTP_CTL_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHPS_CTL_EL2;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHP_CTL_EL2;
    else
        return CNTP_CTL_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x180];
    else
        return CNTP_CTL_EL0;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHPS_CTL_EL2;
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        return CNTHP_CTL_EL2;
    else
        return CNTP_CTL_EL0;
elsif PSTATE.EL == EL3 then
    return CNTP_CTL_EL0;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR CNTP_CTL_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHPS_CTL_EL2 = X[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHP_CTL_EL2 = X[t];
    else
        CNTP_CTL_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x180] = X[t];
    else
        CNTP_CTL_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHPS_CTL_EL2 = X[t];
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        CNTHP_CTL_EL2 = X[t];
    else
        CNTP_CTL_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    CNTP_CTL_EL0 = X[t];
```

MRS <Xt>, CNTP_CTL_EL02

op0	op1	CRn	CRm	op2
0b11	0b101	0b1110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        if EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1NVPCT == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            return NVMem[0x180];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return CNTP_CTL_EL0;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        return CNTP_CTL_EL0;
    else
        UNDEFINED;
```

MSR CNTP_CTL_EL02, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b1110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        if EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1NVPCT == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            NVMem[0x180] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        CNTP_CTL_EL0 = X[t];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        CNTP_CTL_EL0 = X[t];
    else
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.8.17 CNTP_CVAL_EL0, Counter-timer Physical Timer CompareValue register

The CNTP_CVAL_EL0 characteristics are:

Purpose

Holds the compare value for the EL1 physical timer.

Configurations

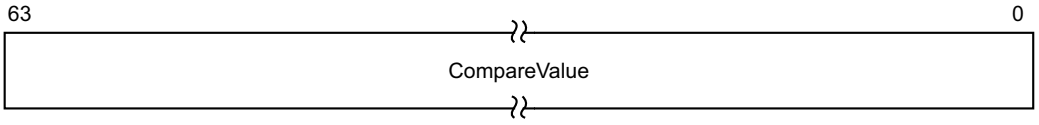
AArch64 System register CNTP_CVAL_EL0[63:0] is architecturally mapped to AArch32 System register CNTP_CVAL[63:0].

Attributes

CNTP_CVAL_EL0 is a 64-bit register.

Field descriptions

The CNTP_CVAL_EL0 bit assignments are:



CompareValue, bits [63:0]

Holds the EL1 physical timer CompareValue.

When CNTP_CTL_EL0.ENABLE is 1, the timer condition is met when (CNTPCT_EL0 - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTP_CTL_EL0.ISTATUS is set to 1.
- If CNTP_CTL_EL0.IMASK is 0, an interrupt is generated.

When CNTP_CTL_EL0.ENABLE is 0, the timer condition is not met, but CNTPCT_EL0 continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTP_CVAL_EL0

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic CNTP_CVAL_EL0 or CNTP_CVAL_EL02 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, CNTP_CVAL_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b010

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHPS_CVAL_EL2;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHP_CVAL_EL2;
    else
        return CNTP_CVAL_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x178];
    else
        return CNTP_CVAL_EL0;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHPS_CVAL_EL2;
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        return CNTHP_CVAL_EL2;
    else
        return CNTP_CVAL_EL0;
elsif PSTATE.EL == EL3 then
    return CNTP_CVAL_EL0;
```

MSR CNTP_CVAL_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b010

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHPS_CVAL_EL2 = X[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHP_CVAL_EL2 = X[t];
    else
        CNTP_CVAL_EL0 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x178] = X[t];
    else
        CNTP_CVAL_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHPS_CVAL_EL2 = X[t];
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        CNTHP_CVAL_EL2 = X[t];
    else
        CNTP_CVAL_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    CNTP_CVAL_EL0 = X[t];
```

MRS <Xt>, CNTP_CVAL_EL02

op0	op1	CRn	CRm	op2
0b11	0b101	0b1110	0b0010	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        if EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1NVPCT == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            return NVMem[0x178];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return CNTP_CVAL_EL0;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        return CNTP_CVAL_EL0;
    else
        UNDEFINED;
```

MSR CNTP_CVAL_EL02, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b1110	0b0010	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        if EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1NVPCT == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            NVMem[0x178] = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        CNTP_CVAL_EL0 = X[t];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        CNTP_CVAL_EL0 = X[t];
    else
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.8.18 CNTP_TVAL_EL0, Counter-timer Physical Timer TimerValue register

The CNTP_TVAL_EL0 characteristics are:

Purpose

Holds the timer value for the EL1 physical timer.

Configurations

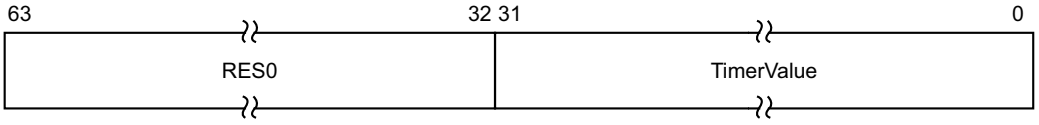
AArch64 System register CNTP_TVAL_EL0[31:0] is architecturally mapped to AArch32 System register CNTP_TVAL[31:0].

Attributes

CNTP_TVAL_EL0 is a 64-bit register.

Field descriptions

The CNTP_TVAL_EL0 bit assignments are:



Bits [63:32]

Reserved, RES0.

TimerValue, bits [31:0]

The TimerValue view of the EL1 physical timer.

On a read of this register:

- If CNTP_CTL_EL0.ENABLE is 0, the value returned is UNKNOWN.
- If CNTP_CTL_EL0.ENABLE is 1, the value returned is (CNTP_CVAL_EL0 - CNTPCT_EL0).

On a write of this register, CNTP_CVAL_EL0 is set to (CNTPCT_EL0 + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTP_CTL_EL0.ENABLE is 1, the timer condition is met when (CNTPCT_EL0 - CNTP_CVAL_EL0) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTP_CTL_EL0.ISTATUS is set to 1.
- If CNTP_CTL_EL0.IMASK is 0, an interrupt is generated.

When CNTP_CTL_EL0.ENABLE is 0, the timer condition is not met, but CNTPCT_EL0 continues to count, so the TimerValue view appears to continue to count down.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTP_TVAL_EL0

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic CNTP_TVAL_EL0 or CNTP_TVAL_EL02 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, CNTP_TVAL_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHPS_TVAL_EL2;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHP_TVAL_EL2;
    else
        return CNTP_TVAL_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return CNTP_TVAL_EL0;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHPS_TVAL_EL2;
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        return CNTHP_TVAL_EL2;
    else
        return CNTP_TVAL_EL0;
elsif PSTATE.EL == EL3 then
    return CNTP_TVAL_EL0;
```

MSR CNTP_TVAL_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHPS_TVAL_EL2 = X[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHP_TVAL_EL2 = X[t];
    else
        CNTP_TVAL_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        CNTP_TVAL_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHPS_TVAL_EL2 = X[t];
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        CNTHP_TVAL_EL2 = X[t];
    else
        CNTP_TVAL_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    CNTP_TVAL_EL0 = X[t];
```

MRS <Xt>, CNTP_TVAL_EL02

op0	op1	CRn	CRm	op2
0b11	0b101	0b1110	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return CNTP_TVAL_EL0;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        return CNTP_TVAL_EL0;
    else
        UNDEFINED;
```

MSR CNTP_TVAL_EL02, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b1110	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        else
            UNDEFINED;
    elsif PSTATE.EL == EL2 then
        if HCR_EL2.E2H == '1' then
            CNTP_TVAL_EL0 = X[t];
        else
            UNDEFINED;
    elsif PSTATE.EL == EL3 then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
            CNTP_TVAL_EL0 = X[t];
        else
            UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.8.19 CNTPCTSS_EL0, Counter-timer Self-Synchronized Physical Count register

The CNTPCTSS_EL0 characteristics are:

Purpose

Holds the self-synchronized view of the 64-bit physical count value.

Configurations

AArch64 System register CNTPCTSS_EL0[63:0] is architecturally mapped to AArch32 System register CNTPCTSS[63:0].

This register is present only when ARMv8.6-ECV is implemented. Otherwise, direct accesses to CNTPCTSS_EL0 are UNDEFINED.

All reads to the CNTPCTSS_EL0 occur in program order relative to reads to CNTPCT_EL0 or CNTPCTSS_EL0.

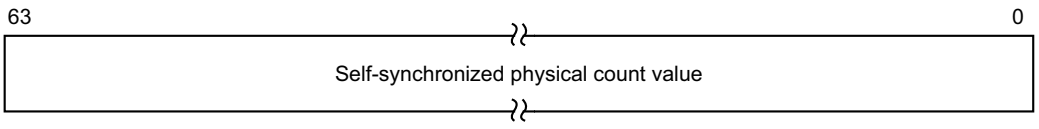
This register is a self-synchronised view of the CNTPCT_EL0 counter, and cannot be read speculatively.

Attributes

CNTPCTSS_EL0 is a 64-bit register.

Field descriptions

The CNTPCTSS_EL0 bit assignments are:



Bits [63:0]

Self-synchronized physical count value.

Accessing the CNTPCTSS_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, CNTPCTSS_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0000	0b101

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PCTEN ==
'0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PCTEN ==
'0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PCTEN ==
'0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        return CNTPCTSS_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CNTHCTL_EL2.EL1PCTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return CNTPCTSS_EL0;
elsif PSTATE.EL == EL2 then
    return CNTPCTSS_EL0;
elsif PSTATE.EL == EL3 then
    return CNTPCTSS_EL0;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.8.20 CNTPCT_EL0, Counter-timer Physical Count register

The CNTPCT_EL0 characteristics are:

Purpose

Holds the 64-bit physical count value.

Configurations

AArch64 System register CNTPCT_EL0[63:0] is architecturally mapped to AArch32 System register CNTPCT[63:0].

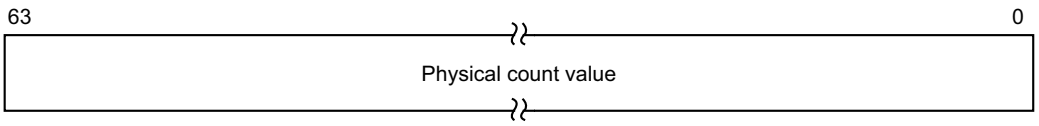
All reads to the CNTPCT_EL0 occur in program order relative to reads to CNTPCTSS_EL0 or CNTPCT_EL0.

Attributes

CNTPCT_EL0 is a 64-bit register.

Field descriptions

The CNTPCT_EL0 bit assignments are:



Bits [63:0]

Physical count value.

Reads of CNTPCT_EL0 from EL0 or EL1 return (PCount<63:0> - CNTPOFF_EL2<63:0>) if the access is not trapped, and all of the following are true:

- CNTHCTL_EL2.ECV is 1.
- HCR_EL2.{E2H, TGE} is not {1, 1}.

Where PCount<63:0> is the physical count returned when CNTPCT_EL0 is read from EL2 or EL3.

Accessing the CNTPCT_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, CNTPCT_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0000	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PCTEN ==
'0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PCTEN ==
'0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PCTEN ==
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
'0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return CNTPCT_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CNTHCTL_EL2.EL1PCTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return CNTPCT_EL0;
elsif PSTATE.EL == EL2 then
    return CNTPCT_EL0;
elsif PSTATE.EL == EL3 then
    return CNTPCT_EL0;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.8.21 CNTPS_CTL_EL1, Counter-timer Physical Secure Timer Control register

The CNTPS_CTL_EL1 characteristics are:

Purpose

Control register for the secure physical timer, usually accessible at EL3 but configurably accessible at EL1 in Secure state.

Configurations

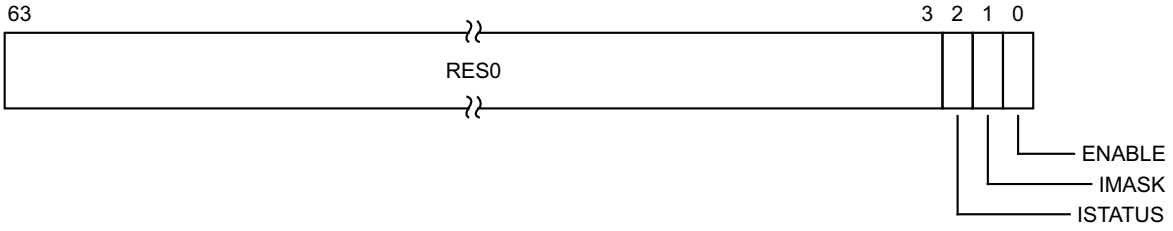
There are no configuration notes.

Attributes

CNTPS_CTL_EL1 is a 64-bit register.

Field descriptions

The CNTPS_CTL_EL1 bit assignments are:



Bits [63:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

`0b0` Timer condition is not met.

`0b1` Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

For more information see *Operation of the CompareValue views of the timers* on page D11-2847 and *Operation of the TimerValue views of the timers* on page D11-2848.

This bit is read-only.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

`0b0` Timer interrupt is not masked by the IMASK bit.

`0b1` Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

This field resets to an architecturally UNKNOWN value.

ENABLE, bit [0]

Enables the timer. Permitted values are:

`0b0` Timer disabled.

`0b1` Timer enabled.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTPS_TVAL_EL1 continues to count down.

Note

Disabling the output signal might be a power-saving option.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTPS_CTL_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, CNTPS_CTL_EL1

op0	op1	CRn	CRm	op2
0b11	0b111	0b1110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && SCR_EL3.NS == '0' then
        if SCR_EL3.EEL2 == '1' then
            UNDEFINED;
        elsif SCR_EL3.ST == '0' then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            return CNTPS_CTL_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    return CNTPS_CTL_EL1;
```

MSR CNTPS_CTL_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b111	0b1110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && SCR_EL3.NS == '0' then
        if SCR_EL3.EEL2 == '1' then
            UNDEFINED;
        elsif SCR_EL3.ST == '0' then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            CNTPS_CTL_EL1 = X[t];
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    CNTPS_CTL_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.8.22 CNTPOFF_EL2, Counter-timer Physical Offset register

The CNTPOFF_EL2 characteristics are:

Purpose

Holds the 64-bit physical offset. This is the offset for the AArch64 physical timers and counters when Enhanced Counter Virtualization is enabled.

Configurations

This register is present only when ARMv8.6-ECV is implemented. Otherwise, direct accesses to CNTPOFF_EL2 are UNDEFINED.

The offsetting of the timers and counters based on EL2 using AArch64 apply at:

- EL1 when EL1 is using AArch64 or AArch32.
- EL0 when EL0 is using AArch64 or AArch32.

When EL2 is implemented and enabled in the current Security state, the physical counter uses a fixed physical offset of zero if either of the following are true:

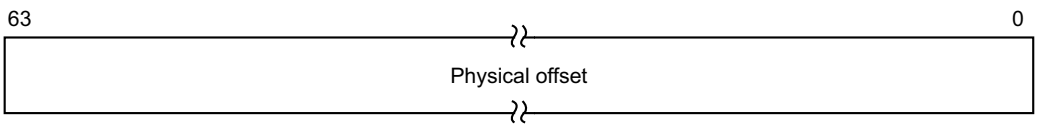
- CNTHCTL_EL2.ECV is 0.
- SCR_EL3.ECVEn is 0.
- HCR_EL2.{E2H, TGE} is {1, 1}.

Attributes

CNTPOFF_EL2 is a 64-bit register.

Field descriptions

The CNTPOFF_EL2 bit assignments are:



Bits [63:0]

Physical offset.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTPOFF_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, CNTPOFF_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0000	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        return NVMem[0x1A8];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.ECVEn == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return CNTPOFF_EL2;
elsif PSTATE.EL == EL3 then
    return CNTPOFF_EL2;
```

MSR CNTPOFF_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0000	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NVMem[0x1A8] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.ECVEn == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        CNTPOFF_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    CNTPOFF_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.8.23 CNTPS_CVAL_EL1, Counter-timer Physical Secure Timer CompareValue register

The CNTPS_CVAL_EL1 characteristics are:

Purpose

Holds the compare value for the secure physical timer, usually accessible at EL3 but configurably accessible at EL1 in Secure state.

Configurations

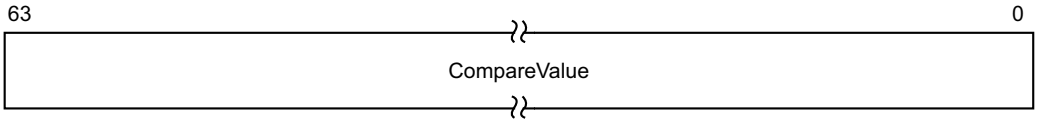
There are no configuration notes.

Attributes

CNTPS_CVAL_EL1 is a 64-bit register.

Field descriptions

The CNTPS_CVAL_EL1 bit assignments are:



CompareValue, bits [63:0]

Holds the secure physical timer CompareValue.

When CNTPS_CTL_EL1.ENABLE is 1, the timer condition is met when (CNTPCT_EL0 - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTPS_CTL_EL1.ISTATUS is set to 1.
- If CNTPS_CTL_EL1.IMASK is 0, an interrupt is generated.

When CNTPS_CTL_EL1.ENABLE is 0, the timer condition is not met, but CNTPCT_EL0 continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTPS_CVAL_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, CNTPS_CVAL_EL1

op0	op1	CRn	CRm	op2
0b11	0b111	0b1110	0b0010	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && SCR_EL3.NS == '0' then
        if SCR_EL3.EEL2 == '1' then
            UNDEFINED;
        elsif SCR_EL3.ST == '0' then
            AArch64.SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        else
            return CNTPS_CVAL_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    return CNTPS_CVAL_EL1;
```

MSR CNTPS_CVAL_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b111	0b1110	0b0010	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && SCR_EL3.NS == '0' then
        if SCR_EL3.EEL2 == '1' then
            UNDEFINED;
        elsif SCR_EL3.ST == '0' then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            CNTPS_CVAL_EL1 = X[t];
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    CNTPS_CVAL_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.8.24 CNTPS_TVAL_EL1, Counter-timer Physical Secure Timer TimerValue register

The CNTPS_TVAL_EL1 characteristics are:

Purpose

Holds the timer value for the secure physical timer, usually accessible at EL3 but configurably accessible at EL1 in Secure state.

Configurations

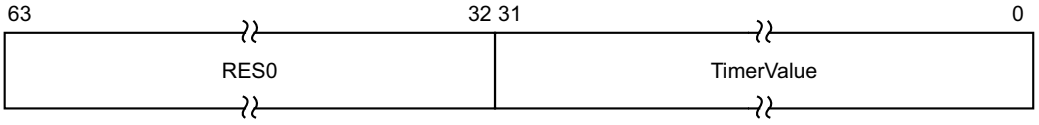
There are no configuration notes.

Attributes

CNTPS_TVAL_EL1 is a 64-bit register.

Field descriptions

The CNTPS_TVAL_EL1 bit assignments are:



Bits [63:32]

Reserved, RES0.

TimerValue, bits [31:0]

The TimerValue view of the secure physical timer.

On a read of this register:

- If CNTPS_CTL_EL1.ENABLE is 0, the value returned is UNKNOWN.
- If CNTPS_CTL_EL1.ENABLE is 1, the value returned is (CNTPS_CVAL_EL1 - CNTPCT_EL0).

On a write of this register, CNTPS_CVAL_EL1 is set to (CNTPCT_EL0 + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTPS_CTL_EL1.ENABLE is 1, the timer condition is met when (CNTPCT_EL0 - CNTPS_CVAL_EL1) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTPS_CTL_EL1.ISTATUS is set to 1.
- If CNTPS_CTL_EL1.IMASK is 0, an interrupt is generated.

When CNTPS_CTL_EL1.ENABLE is 0, the timer condition is not met, but CNTPCT_EL0 continues to count, so the TimerValue view appears to continue to count down.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTPS_TVAL_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, CNTPS_TVAL_EL1

op0	op1	CRn	CRm	op2
0b11	0b111	0b1110	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && SCR_EL3.NS == '0' then
        if SCR_EL3.EEL2 == '1' then
            UNDEFINED;
        elsif SCR_EL3.ST == '0' then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            return CNTPS_TVAL_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    return CNTPS_TVAL_EL1;
```

MSR CNTPS_TVAL_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b111	0b1110	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && SCR_EL3.NS == '0' then
        if SCR_EL3.EEL2 == '1' then
            UNDEFINED;
        elsif SCR_EL3.ST == '0' then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            CNTPS_TVAL_EL1 = X[t];
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    CNTPS_TVAL_EL1 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.8.25 CNTV_CTL_EL0, Counter-timer Virtual Timer Control register

The CNTV_CTL_EL0 characteristics are:

Purpose

Control register for the virtual timer.

Configurations

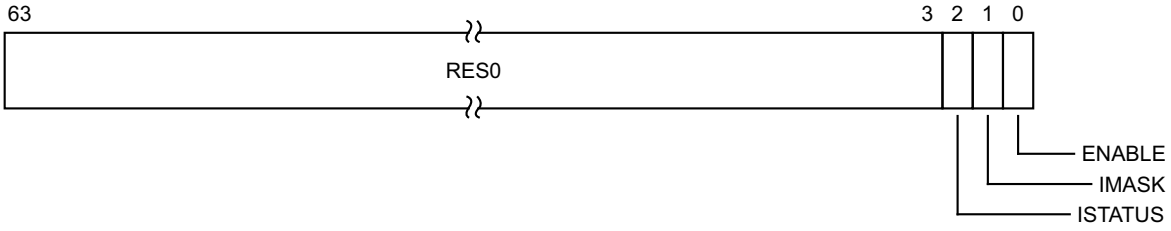
AArch64 System register CNTV_CTL_EL0[31:0] is architecturally mapped to AArch32 System register CNTV_CTL[31:0].

Attributes

CNTV_CTL_EL0 is a 64-bit register.

Field descriptions

The CNTV_CTL_EL0 bit assignments are:



Bits [63:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

`0b0` Timer condition is not met.

`0b1` Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

For more information see *Operation of the CompareValue views of the timers* on page D11-2847 and *Operation of the TimerValue views of the timers* on page D11-2848.

This bit is read-only.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

`0b0` Timer interrupt is not masked by the IMASK bit.

`0b1` Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

This field resets to an architecturally UNKNOWN value.

ENABLE, bit [0]

Enables the timer. Permitted values are:

`0b0` Timer disabled.

`0b1` Timer enabled.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTV_TVAL_EL0 continues to count down.

Note

Disabling the output signal might be a power-saving option.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTV_CTL_EL0

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic CNTV_CTL_EL0 or CNTV_CTL_EL02 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, CNTV_CTL_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHVS_CTL_EL2;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHV_CTL_EL2;
    else
        return CNTV_CTL_EL0;
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x170];
    else
        return CNTV_CTL_EL0;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHVS_CTL_EL2;
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        return CNTHV_CTL_EL2;
    else
        return CNTV_CTL_EL0;
elsif PSTATE.EL == EL3 then
    return CNTV_CTL_EL0;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MSR CNTV_CTL_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHVS_CTL_EL2 = X[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHV_CTL_EL2 = X[t];
    else
        CNTV_CTL_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x170] = X[t];
    else
        CNTV_CTL_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHVS_CTL_EL2 = X[t];
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        CNTHV_CTL_EL2 = X[t];
    else
        CNTV_CTL_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    CNTV_CTL_EL0 = X[t];
```

MRS <Xt>, CNTV_CTL_EL02

op0	op1	CRn	CRm	op2
0b11	0b101	0b1110	0b0011	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        if EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1NVVCT == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            return NVMem[0x170];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if HCR_EL2.E2H == '1' then
        return CNTV_CTL_EL0;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        return CNTV_CTL_EL0;
    else
        UNDEFINED;
```

MSR CNTV_CTL_EL02, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b1110	0b0011	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        if EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1NVVCT == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            NVMem[0x170] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        CNTV_CTL_EL0 = X[t];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        CNTV_CTL_EL0 = X[t];
    else
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.8.26 CNTV_CVAL_EL0, Counter-timer Virtual Timer CompareValue register

The CNTV_CVAL_EL0 characteristics are:

Purpose

Holds the compare value for the virtual timer.

Configurations

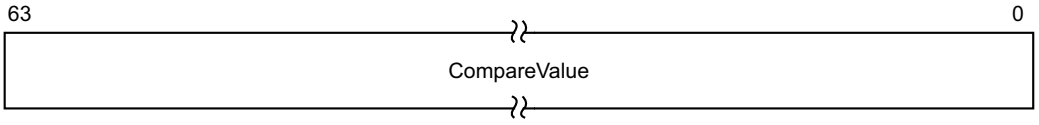
AArch64 System register CNTV_CVAL_EL0[63:0] is architecturally mapped to AArch32 System register CNTV_CVAL[63:0].

Attributes

CNTV_CVAL_EL0 is a 64-bit register.

Field descriptions

The CNTV_CVAL_EL0 bit assignments are:



CompareValue, bits [63:0]

Holds the EL1 virtual timer CompareValue.

When CNTV_CTL_EL0.ENABLE is 1, the timer condition is met when (CNTVCT_EL0 - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTV_CTL_EL0.ISTATUS is set to 1.
- If CNTV_CTL_EL0.IMASK is 0, an interrupt is generated.

When CNTV_CTL_EL0.ENABLE is 0, the timer condition is not met, but CNTVCT_EL0 continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTV_CVAL_EL0

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic CNTV_CVAL_EL0 or CNTV_CVAL_EL02 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, CNTV_CVAL_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b010

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHVS_CVAL_EL2;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHV_CVAL_EL2;
    else
        return CNTV_CVAL_EL0;
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
        return NVMem[0x168];
    else
        return CNTV_CVAL_EL0;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHVS_CVAL_EL2;
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        return CNTHV_CVAL_EL2;
    else
        return CNTV_CVAL_EL0;
elsif PSTATE.EL == EL3 then
    return CNTV_CVAL_EL0;
```

MSR CNTV_CVAL_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b010

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHVS_CVAL_EL2 = X[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHV_CVAL_EL2 = X[t];
    else
        CNTV_CVAL_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<NV2,NV1,NV> == '111' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        NVMem[0x168] = X[t];
    else
        CNTV_CVAL_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHVS_CVAL_EL2 = X[t];
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        CNTHV_CVAL_EL2 = X[t];
    else
        CNTV_CVAL_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    CNTV_CVAL_EL0 = X[t];
```

MRS <Xt>, CNTV_CVAL_EL02

op0	op1	CRn	CRm	op2
0b11	0b101	0b1110	0b0011	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        if EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1NVVCT == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            return NVMem[0x168];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return CNTV_CVAL_EL0;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        return CNTV_CVAL_EL0;
    else
        UNDEFINED;
```

MSR CNTV_CVAL_EL02, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b1110	0b0011	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        if EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1NVVCT == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            NVMem[0x168] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            CNTV_CVAL_EL0 = X[t];
        else
            UNDEFINED;
    elsif PSTATE.EL == EL3 then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
            CNTV_CVAL_EL0 = X[t];
        else
            UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.8.27 CNTV_TVAL_EL0, Counter-timer Virtual Timer TimerValue register

The CNTV_TVAL_EL0 characteristics are:

Purpose

Holds the timer value for the EL1 virtual timer.

Configurations

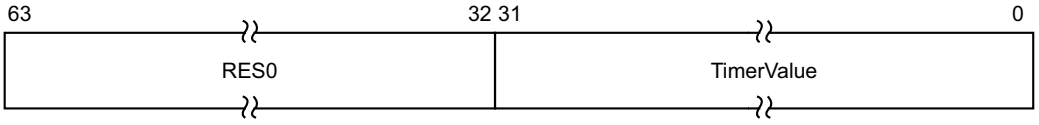
AArch64 System register CNTV_TVAL_EL0[31:0] is architecturally mapped to AArch32 System register CNTV_TVAL[31:0].

Attributes

CNTV_TVAL_EL0 is a 64-bit register.

Field descriptions

The CNTV_TVAL_EL0 bit assignments are:



Bits [63:32]

Reserved, RES0.

TimerValue, bits [31:0]

The TimerValue view of the EL1 virtual timer.

On a read of this register:

- If CNTV_CTL_EL0.ENABLE is 0, the value returned is UNKNOWN.
- If CNTV_CTL_EL0.ENABLE is 1, the value returned is (CNTV_CVAL_EL0 - CNTVCT_EL0).

On a write of this register, CNTV_CVAL_EL0 is set to (CNTVCT_EL0 + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTV_CTL_EL0.ENABLE is 1, the timer condition is met when (CNTVCT_EL0 - CNTV_CVAL_EL0) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTV_CTL_EL0.ISTATUS is set to 1.
- If CNTV_CTL_EL0.IMASK is 0, an interrupt is generated.

When CNTV_CTL_EL0.ENABLE is 0, the timer condition is not met, but CNTVCT_EL0 continues to count, so the TimerValue view appears to continue to count down.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTV_TVAL_EL0

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic CNTV_TVAL_EL0 or CNTV_TVAL_EL02 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, CNTV_TVAL_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHVS_TVAL_EL2;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHV_TVAL_EL2;
    else
        return CNTV_TVAL_EL0;
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return CNTV_TVAL_EL0;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHVS_TVAL_EL2;
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        return CNTHV_TVAL_EL2;
    else
        return CNTV_TVAL_EL0;
elsif PSTATE.EL == EL3 then
    return CNTV_TVAL_EL0;
```

MSR CNTV_TVAL_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        CNTHVS_TVAL_EL2 = X[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHV_TVAL_EL2 = X[t];
    else
        CNTV_TVAL_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        CNTV_TVAL_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHVS_TVAL_EL2 = X[t];
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        CNTHV_TVAL_EL2 = X[t];
    else
        CNTV_TVAL_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    CNTV_TVAL_EL0 = X[t];
```

MRS <Xt>, CNTV_TVAL_EL02

op0	op1	CRn	CRm	op2
0b11	0b101	0b1110	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return CNTV_TVAL_EL0;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        return CNTV_TVAL_EL0;
    else
        UNDEFINED;
```

MSR CNTV_TVAL_EL02, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b1110	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        CNTV_TVAL_EL0 = X[t];
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        CNTV_TVAL_EL0 = X[t];
    else
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.8.28 CNTVCTSS_EL0, Counter-timer Self-Synchronized Virtual Count register

The CNTVCTSS_EL0 characteristics are:

Purpose

Holds the 64-bit virtual count value. The virtual count value is equal to the physical count value visible in CNTPCT_EL0 minus the virtual offset visible in CNTVOFF_EL2.

Configurations

AArch64 System register CNTVCTSS_EL0[63:0] is architecturally mapped to AArch32 System register CNTVCTSS[63:0].

This register is present only when ARMv8.6-ECV is implemented. Otherwise, direct accesses to CNTVCTSS_EL0 are UNDEFINED.

All reads to the CNTVCTSS_EL0 occur in program order relative to reads to CNTVCT_EL0 or CNTVCTSS_EL0.

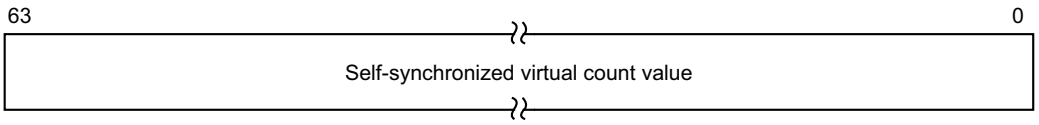
This register is a self-synchronised view of the CNTVCT_EL0 counter, and cannot be read speculatively.

Attributes

CNTVCTSS_EL0 is a 64-bit register.

Field descriptions

The CNTVCTSS_EL0 bit assignments are:



Bits [63:0]

Self-synchronized virtual count value.

Accessing the CNTVCTSS_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, CNTVCTSS_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0000	0b110

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VCTEN ==
'0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VCTEN ==
'0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTHCTL_EL2.EL1TVCT ==
'1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        return CNTVCTSS_EL0;
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTHCTL_EL2.EL1TVCT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return CNTVCTSS_EL0;
elsif PSTATE.EL == EL2 then
    return CNTVCTSS_EL0;
elsif PSTATE.EL == EL3 then
    return CNTVCTSS_EL0;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.8.29 CNTVCT_EL0, Counter-timer Virtual Count register

The CNTVCT_EL0 characteristics are:

Purpose

Holds the 64-bit virtual count value. The virtual count value is equal to the physical count value minus the virtual offset visible in CNTVOFF_EL2.

Configurations

AArch64 System register CNTVCT_EL0[63:0] is architecturally mapped to AArch32 System register CNTVCT[63:0].

The value of this register is the same as the value of CNTPCT_EL0 in the following conditions:

- When EL2 is not implemented.
- When EL2 is implemented, HCR_EL2.E2H is 1, and this register is read from EL2.
- When EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} is {1, 1}, and this register is read from EL0 or EL2.

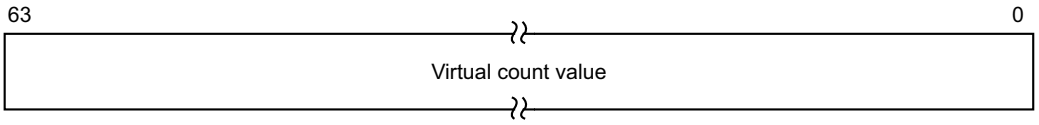
All reads to the CNTVCT_EL0 occur in program order relative to reads to CNTVCTSS_EL0 or CNTVCT_EL0.

Attributes

CNTVCT_EL0 is a 64-bit register.

Field descriptions

The CNTVCT_EL0 bit assignments are:



Bits [63:0]

Virtual count value.

Accessing the CNTVCT_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, CNTVCT_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0000	0b010

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VCTEN ==
'0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VCTEN ==
'0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTHCTL_EL2.EL1TVCT ==
'1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return CNTVCT_EL0;
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTHCTL_EL2.EL1TVCT == '1'
then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return CNTVCT_EL0;
elsif PSTATE.EL == EL2 then
    return CNTVCT_EL0;
elsif PSTATE.EL == EL3 then
    return CNTVCT_EL0;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D13.8.30 CNTVOFF_EL2, Counter-timer Virtual Offset register

The CNTVOFF_EL2 characteristics are:

Purpose

Holds the 64-bit virtual offset. This is the offset between the physical count value visible in CNTPCT_EL0 and the virtual count value visible in CNTVCT_EL0.

Configurations

AArch64 System register CNTVOFF_EL2[63:0] is architecturally mapped to AArch32 System register CNTVOFF[63:0].

If EL2 is not implemented, this register is RES0 from EL3 and the virtual counter uses a fixed virtual offset of zero.

Note

When EL2 is implemented and enabled in the current Security state, and is using AArch64, the virtual counter uses a fixed virtual offset of zero in the following situations:

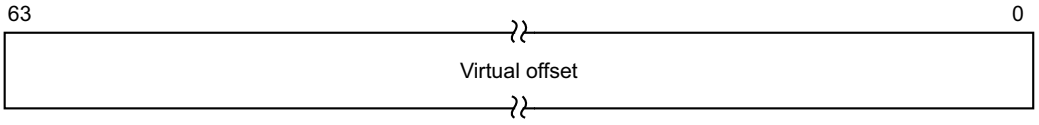
- HCR_EL2.E2H is 1, and CNTVCT_EL0 is read from EL2.
- HCR_EL2.{E2H, TGE} is {1, 1}, and either:
  - CNTVCT_EL0 is read from EL0 or EL2.
  - CNTVCT is read from EL0.

Attributes

CNTVOFF_EL2 is a 64-bit register.

Field descriptions

The CNTVOFF_EL2 bit assignments are:



Bits [63:0]

Virtual offset.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTVOFF_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS <Xt>, CNTVOFF_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0000	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        return NVMem[0x060];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return CNTVOFF_EL2;
elsif PSTATE.EL == EL3 then
    return CNTVOFF_EL2;
```

MSR CNTVOFF_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0000	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NVMem[0x060] = X[t];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    CNTVOFF_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    CNTVOFF_EL2 = X[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Part E

The AArch32 Application Level Architecture

Chapter E1
The AArch32 Application Level Programmers' Model

This chapter gives an Application level description of the programmers' model for software executing in AArch32 state. This means it describes execution in EL0 when EL0 is using AArch32. It contains the following sections:

- *About the Application level programmers' model* on page E1-3948.
- *The Application level programmers' model in AArch32 state* on page E1-3949.
- *Advanced SIMD and floating-point instructions* on page E1-3960.
- *About the AArch32 System register interface* on page E1-3971.
- *Exceptions* on page E1-3972.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

E1.1 About the Application level programmers' model

This chapter contains the programmers' model information required for the development of applications that will execute in AArch32 state.

The information in this chapter is distinct from the system information required to service and support application execution under an operating system, or higher level of system software. However, some knowledge of that system information is needed to put the Application level programmers' model into context.

Depending on the implementation, the architecture supports multiple levels of execution privilege. These privilege levels are indicated by different *Exception levels* that number upwards from EL0, where EL0 corresponds to the lowest privilege level and is often described as *unprivileged*. The Application level programmers' model is the programmers' model for software executing at EL0. For more information see *Armv8 architectural concepts* on page A1-37.

System software determines the Exception level, and therefore the level of privilege, at which application software runs. When an operating system supports execution at both EL1 and EL0, an application usually runs unprivileged. This has the following effects:

- It means that the operating system can allocate system resources to an application in a unique or shared manner.
- It provides a degree of protection from other processes, and so helps protect the operating system from malfunctioning software.

This chapter indicates where some System level understanding is helpful, and if appropriate it gives a reference to the System level description.

Application level software is generally unaware of its Security state, and of any virtualization. For more information, see *The Armv8-A security model* on page G1-5679 and *The effect of implementing EL2 on the Exception model* on page G1-5684.

Note

- When an implementation includes EL3, application and operating system software normally executes in Non-secure state.
- Older documentation, describing implementations or architecture versions that support only two privilege levels, often refers to execution at EL1 as *privileged* execution.
- In this manual, the terms *CONSTRAINED UNPREDICTABLE*, *IMPLEMENTATION DEFINED*. *OPTIONAL*, *RES0*, *RES1*, *UNDEFINED*, *UNKNOWN*, and *UNPREDICTABLE* have Arm-specific meanings, as defined in the *Glossary*. In body text, these terms are shown in SMALL CAPS, for example IMPLEMENTATION DEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

E1.2 The Application level programmers' model in AArch32 state

The following sections give more information about the Application level programmers' model in AArch32 state:

- *Instruction sets, arithmetic operations, and register files*.
- *Core data types and arithmetic in AArch32 state*.
- *The general-purpose registers, and the PC, in AArch32 state* on page E1-3951.
- *Process state, PSTATE* on page E1-3953.
- *Jazelle support* on page E1-3959.

E1.2.1 Instruction sets, arithmetic operations, and register files

The A32 and T32 instruction sets both provide a wide range of integer arithmetic and logical operations, that operate on a register file of sixteen 32-bit registers, that are comprised of the AArch32 general-purpose registers and the PC. As described in *The general-purpose registers, and the PC, in AArch32 state* on page E1-3951, these registers include the registers SP (R13) and LR (R14), which have specialized uses. *Core data types and arithmetic in AArch32 state* gives more information about these operations.

In addition, an implementation that implements the T32 and A32 instruction sets includes both:

- Scalar floating-point instructions.
- The Advanced SIMD vector instructions.

Floating-point and vector instructions operate on a separate common register file, described in *The SIMD and floating-point register file* on page E1-3960. *Advanced SIMD and floating-point instructions* on page E1-3960 gives more information about these instructions.

E1.2.2 Core data types and arithmetic in AArch32 state

When executing in AArch32 state, a PE supports the following data types in memory:

Byte 8 bits.

Halfword 16 bits.

Word 32 bits.

Doubleword 64 bits.

PE registers are 32 bits in size. The instruction sets provide instructions that use the following data types for data held in registers:

- 32-bit pointers.
- Unsigned or signed 32-bit integers.
- Unsigned 16-bit or 8-bit integers, held in zero-extended form.
- Signed 16-bit or 8-bit integers, held in sign-extended form.
- Two 16-bit integers packed into a register.
- Four 8-bit integers packed into a register.
- Unsigned or signed 64-bit integers held in two registers.

Load and store operations can transfer bytes, halfwords, or words to and from memory. Loads of bytes or halfwords zero-extend or sign-extend the data as it is loaded, as specified in the appropriate load instruction.

The instruction sets include load and store operations that transfer two or more words to and from memory. Software can load and store doublewords using these instructions.

Note

For information about the atomicity of memory accesses see *Atomicity in the Arm architecture* on page E2-3976.

When any of the data types is described as *unsigned*, the N-bit data value represents a non-negative integer in the range 0 to 2^N-1, using normal binary format.

When any of these types is described as *signed*, the N-bit data value represents an integer in the range $-2^{(N-1)}$ to $+2^{(N-1)}-1$, using two's complement format.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The instructions that operate on packed halfwords or bytes include some multiply instructions that use only one of two halfwords, and SIMD instructions that perform parallel addition or subtraction on all of the halfwords or bytes.

Note

These SIMD instructions operate on values held in the general-purpose registers, and must not be confused with the Advanced SIMD instructions that operate on a separate register file that provides registers of up 128 bits.

Direct instruction support for 64-bit integers is limited, and most 64-bit operations require sequences of two or more instructions to synthesize them.

Integer arithmetic

The instruction set provides a wide range of operations on the values in registers, including bitwise logical operations, shifts, additions, subtractions, multiplications, and divisions. The pseudocode described in Appendix K13 *Arm Pseudocode Definition* defines these operations, usually in one of three ways:

- By direct use of the pseudocode operators and built-in functions defined in *Operators* on page K13-8018.
- By use of pseudocode helper functions defined in the main text. See Appendix K12 *Pseudocode Index*.
- By a sequence of the form:
  1. Use of the `SInt()`, `UInt()`, and `Int()` built-in functions defined in *Converting bitstrings to integers* on page K13-8030 to convert the bitstring contents of the instruction operands to the unbounded integers that they represent as two's complement or unsigned integers.
  2. Use of mathematical operators, built-in functions and helper functions on those unbounded integers to calculate other such integers.
  3. Use of either the bitstring extraction operator defined in *Bitstring concatenation and slicing* on page K13-8019 or of the saturation helper functions described in *Pseudocode description of saturation* on page E1-3951 to convert an unbounded integer result into a bitstring result that can be written to a register.

Shift and rotate operations

The following types of shift and rotate operations are used in instructions:

Logical Shift Left

The `LSL()` pseudocode function moves each bit of a bitstring left by a specified number of bits. Zeros are shifted in at the right end of the bitstring. Bits that are shifted off the left end of the bitstring are discarded, except that the last such bit can be produced as a carry output.

Logical Shift Right

The `LSR()` pseudocode function moves each bit of a bitstring right by a specified number of bits. Zeros are shifted in at the left end of the bitstring. Bits that are shifted off the right end of the bitstring are discarded, except that the last such bit can be produced as a carry output.

Arithmetic Shift Right

The `ASR()` pseudocode function moves each bit of a bitstring right by a specified number of bits. Copies of the leftmost bit are shifted in at the left end of the bitstring. Bits that are shifted off the right end of the bitstring are discarded, except that the last such bit can be produced as a carry output.

Rotate Right The `ROR()` pseudocode function moves each bit of a bitstring right by a specified number of bits. Each bit that is shifted off the right end of the bitstring is re-introduced at the left end. The last bit shifted off the right end of the bitstring can be produced as a carry output.

Rotate Right with Extend

The `RRX()` pseudocode function moves each bit of a bitstring right by one bit. A carry input is shifted in at the left end of the bitstring. The bit shifted off the right end of the bitstring can be produced as a carry output.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Pseudocode description of addition and subtraction

In pseudocode, addition and subtraction can be performed on any combination of unbounded integers and bitstrings, provided that if they are performed on two bitstrings, the bitstrings must be identical in length. The result is another unbounded integer if both operands are unbounded integers, and a bitstring of the same length as the bitstring operand or operands otherwise. For the definition of these operations, see *Addition and subtraction* on page K13-8020.

The main addition and subtraction instructions can produce status information about both unsigned carry and signed overflow conditions. When necessary, multi-word additions and subtractions can be synthesized from this status information. In pseudocode the `AddWithCarry()` function provides an addition with a carry input and a set of output Condition flags including carry output and overflow:

An important property of the `AddWithCarry()` function is that if:

```
(result, nzcv) = AddWithCarry(x, NOT(y), carry_in)
```

Then:

- If `carry_in == '1'`, then `result == x-y` with:
  - `nzcv<0> == '1'` if signed overflow occurred during the subtraction.
  - `nzcv<1> == '1'` if unsigned borrow did not occur during the subtraction, that is, if x≥y.
- If `carry_in == '0'`, then `result == x-y-1` with:
  - `nzcv<0> == '1'` if signed overflow occurred during the subtraction.
  - `nzcv<1> == '1'` if unsigned borrow did not occur during the subtraction, that is, if x≥y.

Taken together, this means that the `carry_in` and `nzcv<1>` output in `AddWithCarry()` calls can act as NOT borrow flags for subtractions as well as carry flags for additions.

Pseudocode description of saturation

Some instructions perform *saturating arithmetic*, that is, if the result of the arithmetic overflows the destination signed or unsigned N-bit integer range, the result produced is the largest or smallest value in that range, rather than wrapping around modulo 2^N. This is supported in pseudocode by:

- The `SignedSatQ()` and `UnsignedSatQ()` functions when an operation requires, in addition to the saturated result, a Boolean argument that indicates whether saturation occurred.
- The `SignedSat()` and `UnsignedSat()` functions when only the saturated result is required.

`SatQ(i, N, unsigned)` returns either `UnsignedSatQ(i, N)` or `SignedSatQ(i, N)` depending on the value of its third argument, and `Sat(i, N, unsigned)` returns either `UnsignedSat(i, N)` or `SignedSat(i, N)` depending on the value of its third argument.

E1.2.3 The general-purpose registers, and the PC, in AArch32 state

In the AArch32 Application level view, a PE has:

- Fifteen general-purpose 32-bit registers, R0 to R14, of which R13 and R14 have alternative names reflecting how they are, or can be, used:
  - R13 is usually identified as SP.
  - R14 is usually identified as LR.
- The PC (*program counter*), that can be described as R15.

The specialized uses of the SP (R13), LR (R14), and PC (R15) are:

SP, the stack pointer

The PE uses SP as a pointer to the active stack.

In the T32 instruction set, some instructions cannot access SP. Instructions that can access SP can use SP as a general-purpose register.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The A32 instruction set provides more general access to SP, and it can be used as a general-purpose register.

Note

Using SP for any purpose other than as a stack pointer might break the requirements of operating systems, debuggers, and other software systems, causing them to malfunction.

Software can refer to SP as R13.

LR, the link register

The link register can be used to hold return link information, and some cases described in this manual require this use of the LR. When software does not require the LR for linking, it can use it for other purposes. Software can refer to LR as R14.

PC, the program counter

- When executing an A32 instruction, PC reads as the address of the current instruction plus 8.
- When executing a T32 instruction, PC reads as the address of the current instruction plus 4.
- Writing an address to PC causes a branch to that address.

Most T32 instructions cannot access PC.

The A32 instruction set provides more general access to the PC, and many A32 instructions can use the PC as a general-purpose register. However, Arm deprecates the use of PC for any purpose other than as the program counter. See *Writing to the PC* for more information.

Software can refer to PC as R15.

See *AArch32 general-purpose registers, the PC, and the Special-purpose registers* on page G1-5691 for the system level view of these registers.

Note

In general, Arm strongly recommends using the names SP, LR and PC instead of R13, R14 and R15. However, sometimes it is simpler to use the R13-R15 names when referring to a group of registers. For example, it is simpler to refer to *registers R8 to R15*, rather than to *registers R8 to R12, the SP, LR and PC*. These two descriptions of the group of registers have exactly the same meaning.

Writing to the PC

In the A32 and T32 instruction sets, many data-processing instructions can write to the PC. Writes to the PC are handled as follows:

- In T32 state, the following 16-bit T32 instruction encodings branch to the value written to the PC:
  - Encoding T2 of *ADD, ADDS (register)* on page F5-4256.
  - Encoding T1 of *MOV, MOVS (register)* on page F5-4517.

  The value written to the PC is forced to be halfword-aligned by ignoring its least significant bit, treating that bit as being 0.

- The `B`, `BL`, `CBNZ`, `CBZ`, `CHKA`, `HB`, `HBL`, `HBLP`, `HBP`, `TBB`, and `TBH` instructions remain in the same instruction set state and branch to the value written to the PC.

  The definition of each of these instructions ensures that the value written to the PC is correctly aligned for the current instruction set state.

- The `BLX` (immediate) instruction switches between A32 and T32 states and branches to the value written to the PC. Its definition ensures that the value written to the PC is correctly aligned for the new instruction set state.

- The following instructions write a value to the PC, treating that value as an interworking address to branch to, with low-order bits that determine the new instruction set state:
  - `BLX` (register), `BX`, and `BXJ`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— LDR instructions with <Rt> equal to the PC.

— POP and all forms of LDM except LDM (exception return), when the register list includes the PC.

— In A32 state only, ADC, ADD, ADR, AND, ASR (immediate), BIC, EOR, LSL (immediate), LSR (immediate), MOV, MVN, ORR, ROR (immediate), RRX, RSB, RSC, SBC, and SUB instructions with <Rd> equal to the PC and without flag-setting specified.

For details of how an interworking address specifies the new instruction set state and instruction address, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.

Note

The register-shifted register instructions, that are available only in the A32 instruction set and are summarized in *Data-processing register (register shift)* on page F4-4191, are CONSTRAINED UNPREDICTABLE if they attempt to write to the PC, see *Using R15* on page K1-7831.

- Some instructions are treated as exception return instructions, and write both the PC and the CPSR. For more information, including which instructions are exception return instructions, see *Exception return to an Exception level using AArch32* on page G1-5725.
- Some instructions cause an exception, and the exception handler address is written to the PC as part of the exception entry.

Pseudocode description of operations on the AArch32 general-purpose registers and the PC

In pseudocode, the uses of the R[] function, with an index parameter n, are:

- Reading or writing R0-R12, SP, and LR, using n = 0-12, 13, and 14 respectively.
- Reading the PC, using n = 15.

Pseudocode description of general-purpose register and PC operations on page G1-5693 describes accesses to these registers.

Descriptions of A32 store instructions that store the PC value use the PCStoreValue() pseudocode function to specify the PC value stored by the instruction.

Writing an address to the PC causes either a simple branch to that address or an *interworking* branch that also selects the instruction set to execute after the branch. A simple branch is performed by the BranchWritePC() function.

An interworking branch is performed by the BXWritePC() function.

The LoadWritePC() and ALUWritePC() functions are used for two cases where the behavior was systematically modified between architecture versions.

E1.2.4 Process state, PSTATE

Process state or PSTATE is an abstraction of process state information. All of the instruction sets provide instructions that operate on elements of PSTATE.

Note

In this chapter, references to PSTATE link to the more appropriate of:

- The Application-level view of PSTATE given in this section.
- The System-level description in *Process state, PSTATE* on page G1-5695.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following PSTATE information is accessible at EL0:

The Condition flags

Flag-setting instructions set these. They are:

N Negative Condition flag. If the result of the instruction is regarded as a two's complement signed integer, the PE sets this to:

- 1 if the result is negative.
- 0 if the result is positive or zero.

Z Zero Condition flag. Set to:

- 1 if the result of the instruction is zero.
- 0 otherwise.

A result of zero often indicates an equal result from a comparison.

C Carry Condition flag. Set to:

- 1 if the instruction results in a carry condition, for example an unsigned overflow that is the result of an addition.
- 0 otherwise.

V Overflow Condition flag. Set to:

- 1 if the instruction results in an overflow condition, for example a signed overflow that is the result of an addition.
- 0 otherwise.

Conditional instructions test the N, Z, C, and V Condition flags, combining them with the *Condition code* for the instruction, to determine whether the instruction must be executed. In this way, execution of the instruction is conditional on the result of a previous operation. For more information about conditional execution, see *Conditional execution* on page F2-4081.

The overflow or saturation flag

Q Some instructions can set this. For those instructions that can, the PE:

- Sets it to 1 if the instruction indicates overflow or saturation.
- Leaves it unchanged otherwise.

For more information, see *Pseudocode description of saturation* on page E1-3951.

The greater than or equal flags

GE[3:0] The instructions described in *Parallel addition and subtraction instructions* on page F1-4046 update these to indicate the results from individual bytes or halfwords of the operation. These flags can control a later `SEL` instruction. For more information, see *SEL* on page F5-4674.

PSTATE also contains *PE state controls*. There is no direct access to these from application level instructions, but they can be changed by side-effects of application level instructions. They are:

Instruction set state

J, T The current instruction set state, as shown in Table E1-1. In Armv8, the J bit is RES0, see the Note in this section.

Table E1-1 PSTATE.{J, T} encoding

J	T	Instruction set state
0	0	A32
0	1	T32

A32 The PE is executing the A32 instruction set, summarized in Chapter F4 *A32 Instruction Set Encoding*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T32 The PE is executing the T32 instruction set, summarized in Chapter F3 *T32 Instruction Set Encoding*.

Note

Encoding with J==1 before Armv8, Jazelle and T32EE states

In previous versions of the Arm architecture, the encoding {1, 0} selected Jazelle state, and encoding {1, 1} selected T32EE state. Armv8 does not support either of these states, and these are encodings for unimplemented instruction set states, see *Unimplemented instruction sets* on page G1-5701.

Armv8 AArch32 state requires a Trivial Jazelle implementation, see *Trivial implementation of the Jazelle extension* on page G1-5701.

The IT block state

IT[7:0] The If-Then controls for the T32 `IT` instruction, that applies to the *IT block* of instructions that immediately follow the `IT` instruction. See *IT* on page F5-4378 for a description of the `IT` instruction and its associated `IT` block.

For more information about the use of PSTATE.IT see *Use of PSTATE.IT* on page E1-3957.

Endianness mapping

E For data accesses, controls the endianness:

0 Little-endian.

1 Big-endian.

If an implementation does not provide:

- Big-endian support for data accesses, this bit is RES0.
- Little-endian support for data accesses, this bit is RES1.

Instruction fetches are always little-endian, and ignore PSTATE.E.

Timing control bits

DIT *Data Independent Timing* (DIT) bit. For more information, see *About the DIT bit* on page E1-3959.

This bit is implemented only when ARMv8.4-DIT is implemented.

On a reset to AArch32 state, this bit is set to 0.

Accessing PSTATE fields at EL0

The following sections describe which PSTATE fields can be directly accessed at EL0, and how they can be accessed:

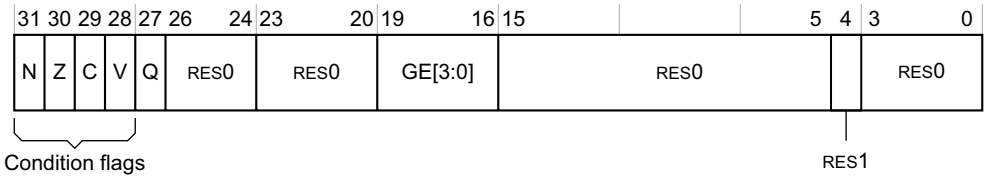
- *The Application Program Status Register, APSR*.
- *The SETEND instruction* on page E1-3956.

The Application Program Status Register, APSR

At EL0, some PSTATE fields can be accessed using the Special-purpose *Application Program Status Register* (APSR). The APSR can be directly read using the MRS instruction, and directly written using the MSR (register) and MSR (immediate) instructions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The APSR bit assignments are:



N, Z, C, V, bits [31:28]

The PSTATE Condition flags.

Q, bit [27] The PSTATE overflow or saturation flag.

Bits[26:24] Reserved, RES0. Software can use MSR instructions that write the top byte of the APSR without using a read-modify-write sequence. If it does this, it must write zeros to bits[26:24].

Bits[23:20, 15:0]

Reserved bits that are allocated to system features, or are available for future expansion. Unprivileged execution ignores writes to fields that are accessible only at EL1 or higher. However, application level software that writes to the APSR must treat reserved bits as *Do-Not-Modify* (DNM) bits. For more information about the reserved bits, see *The Current Program Status Register, CPSR* on page G1-5697.

These bits are UNKNOWN on a Read, and it is permitted that, on a read of APSR:

- Bit[22] returns the value of PSTATE.PAN.
- Bit[9] returns the value of PSTATE.E.
- Bits[8:6] return the value of PSTATE.{A,I,F}, the mask bits.
- Bits[4:0] return the value of PSTATE.M[4:0]. Bit[4] is RES1 indicating that the PE is in AArch32 state.

Note

This is an exception to the general rule that an UNKNOWN field must not return information that cannot be obtained, at the current Privilege level, by an architected mechanism.

GE[3:0], bits [19:16]

The PSTATE greater than or equal flags.

The other PSTATE fields cannot be accessed by using the APSR.

The system level alias for the APSR is the CPSR. The CPSR is a superset of the APSR. See *The Current Program Status Register, CPSR* on page G1-5697.

Writes to the PSTATE fields have side-effects on various aspects of PE operation. All of these side-effects, except side-effects on memory accesses associated with fetching instructions, are synchronous to the APSR write. This means they are guaranteed:

- Not to be visible to earlier instructions in the execution stream.
- To be visible to later instructions in the execution stream.

The SETEND instruction

The A32 and T32 instruction sets both include an instruction to manipulate PSTATE.E:

`SETEND BE` Sets PSTATE.E to 1, for big-endian operation.

`SETEND LE` Sets PSTATE.E to 0, for little-endian operation.

The `SETEND` instruction is unconditional. For more information, see *SETEND* on page F5-4676. Arm deprecates use of the `SETEND` instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Use of PSTATE.IT

PSTATE.IT provides the If-Then controls for the T32 `IT` instruction, that applies to the *IT block* of instructions that immediately follow the `IT` instruction.

PSTATE.IT divides into two subfields:

IT[7:5] Holds the *base condition* for the current IT block. The base condition is the top three bits of the Condition code specified by the `<firstcond>` field of the `IT` instruction.

IT[4:0] Encodes:

- Implicitly, the size of the IT block. This is the number of instructions that are to be conditionally executed. The size of the block is indicated by the position of the least significant 1 in this field, as shown in Table E1-2.
- For each instruction in the IT block, the least significant bit of the Condition code. This is encoded in the IT block entries that Table E1-2 shows as N*x*.

Note

Changing the least significant bit of a Condition code from 0 to 1 has the effect of inverting the Condition code.

Both subfields are all zeros when no IT block is active.

When an `IT` instruction is executed, PSTATE.IT is set according to the `<firstcond>` field of the instruction and the *Then* and *Else* (T and E) parameters in the instruction, see *IT* on page F5-4378. This means that, on executing an IT instruction, the initial state of PSTATE.IT depends on the number of instructions in the IT block, as Table E1-2 shows:

Table E1-2 Initial state of PSTATE.IT on executing an `IT` instruction

Number of instructions in IT block	PSTATE.IT bits[a]						Notes
	[7:5]	[4]	[3]	[2]	[1]	[0]	
4	cond_base	N1	N2	N3	N4	1	-
3	cond_base	N1	N2	N3	1	0	-
2	cond_base	N1	N2	1	0	0	-
1	cond_base	N1	1	0	0	0	-
Not executing an `IT` instruction	000	0	0	0	0	0	No IT block is active

a. Combinations of the IT bits not shown in this table are reserved.

In Table E1-2, N1 refers to the first instruction in the IT block, and N2, N3, and N4 refer to the second, third, and fourth instructions in the IT block if they are present,

When permitted, an instruction in an IT block is conditional, see *Conditional instructions* on page F1-4037 and *Conditional execution* on page F2-4081. The Condition code used is the current value of IT[7:4]. When an instruction in an IT block completes its execution normally, PSTATE.IT[4:0] is left-shifted by one bit, so that PSTATE[4] always relates to the next instruction to be executed. Table E1-3 on page E1-3958 shows how PSTATE.IT during the execution of an `IT` instruction with four instructions in the IT block.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table E1-3 Updates to PSTATE.IT when executing an IT instruction with a four-instruction IT block

IT block instruction being executed	PSTATE.IT bits						Notes
	[7:5]	[4]	[3]	[2]	[1]	[0]	
First	cond_base	N1	N2	N3	N4	1	-
Second	cond_base	N2	N3	N4	1	0	-
Third	cond_base	N3	N4	1	0	0	-
Fourth	cond_base	N4	1	0	0	0	-
Not executing an IT instruction	000	0	0	0	0	0	No IT block is active

A few instructions, for example BKPT, cannot be conditional and therefore are always executed ignoring the current value of PSTATE.IT.

For details of what happens if an instruction in an IT block takes an exception, see *Overview of exception entry* on page G1-5710.

An instruction that might complete its normal execution by branching is only permitted in an IT block as the last instruction in the block. This means that normal execution of the instruction always results in PSTATE.IT advancing to execution where no IT block is active.

For performance reasons, Armv8 deprecates the use of IT other than with a single 16-bit T32 instruction from a specified subset of the 16-bit T32 instructions, see *Partial deprecation of IT* on page F1-4054. In addition, implementations can provide a set of ITD control fields, SCTLR.ITD, SCTLR_EL1.ITD, and HSCTLR.ITD, to disable these deprecated uses, making them UNDEFINED. When an implementation includes ITD control fields, *Changes to an ITD control by an instruction in an IT block* describes the permitted CONSTRAINED UNPREDICTABLE behaviors if an instruction in an IT block changes the value of an ITD control to disable the use of the IT instruction.

On a branch or an exception return, if PSTATE.IT is set to a value that is not consistent with the instruction stream being branched to or returned to, then instruction execution is CONSTRAINED UNPREDICTABLE.

PSTATE.IT affects instruction execution only in T32 state. In A32 state, PSTATE.IT must be 0b00000000, otherwise the behavior is CONSTRAINED UNPREDICTABLE.

For more information see *CONSTRAINED UNPREDICTABLE behavior associated with IT instructions and PSTATE.IT* on page K1-7832.

Changes to an ITD control by an instruction in an IT block

In an implementation that includes SCTLR.ITD, SCTLR_EL1.ITD, and HSCTLR.ITD controls, if an instruction in an IT block changes an ITD control so that the IT instruction using the IT block would be disabled, then one of the following behaviors applies:

- The change to the ITD field, once synchronized, has no effect on the execution of instructions in the current IT block, but applies only to any subsequent execution of an IT instruction to which the control applies.
- Synchronizing the change to the ITD field guarantees that all bits of PSTATE.IT are cleared to 0.

In addition, after the change to the ITD field has been synchronized, any remaining instructions in the IT block that would be made UNDEFINED by the new value of ITD are either:

- Executed normally.
- Treated as UNDEFINED.

The choice between the options described in this section is determined by the implementation, and any choice can vary between different changes to an ITD control by an instruction in an IT block.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Pseudocode description of PSTATE PE state fields

The pseudocode function `CurrentInstrSet()` returns the current instruction set. The pseudocode function `SelectInstrSet()` selects a new instruction set.

PSTATE.IT advances after normal execution of an IT block instruction. This is described by the `AArch32.ITAdvance()` pseudocode function.

The pseudocode function `InITBlock()` tests whether the current instruction is in an IT block. The pseudocode function `LastInITBlock()` tests whether the current instruction is the last instruction in an IT block.

The `BigEndian()` pseudocode function tests whether big-endian data memory accesses are currently selected.

E1.2.5 About the DIT bit

When the value of CPSR.DIT is 1:

- The instructions listed in CPSR are required to have;
  - Timing which is independent of the values of the data supplied in any of its registers, and the values of the NZCV flags.
  - Responses to asynchronous exceptions which do not vary based on the values supplied in any of their registers, or the values of the NZCV flags.
- All loads and stores have their timing insensitive to the value of the data being loaded or stored.

Note

When the value of CPSR.DIT is 0, the architecture makes no statement about the timing properties of any instructions.

A corresponding DIT bit is added to PSTATE in AArch64 state, and to CPSR in AArch32 state.

When an exception is taken from AArch32 state to AArch32 state, CPSR.DIT is copied to SPSR.DIT.

When an exception is taken from AArch32 state to AArch64 state, CPSR.DIT is copied to SPSR_ELx.DIT.

When an exception returns to AArch32 state from AArch32 state, SPSR.DIT is copied to CPSR.DIT.

When an exception returns to AArch32 state from AArch64 state, SPSR_ELx.DIT is copied to CPSR.DIT.

CPSR.DIT bit can be written using an `MSR` instruction at any exception level in AArch32 state, and read using an `MRS` instruction at any exception level.

E1.2.6 Jazelle support

Armv8 requires AArch32 state to include a trivial implementation of the Jazelle extension, as described in *Trivial implementation of the Jazelle extension* on page G1-5701.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

E1.3 Advanced SIMD and floating-point instructions

In general, Armv8 requires implementation of Advanced SIMD and floating-point instructions in the T32 and A32 instruction sets, but see *Implications of not including Advanced SIMD and floating-point support* on page E1-3966.

The Advanced SIMD instructions perform packed *Single Instruction Multiple Data* (SIMD) operations, either integer or single-precision floating-point. The floating-point instructions perform single-precision or double-precision scalar floating-point operations. When ARMv8.2-FP16 is implemented, half-precision floating-point can also be used for data processing.

These instructions permit *floating-point exceptions*, such as Overflow or Divide by Zero, to be handled without trapping. When handled in this way, a floating-point exception causes a cumulative status register bit to be set to 1 and a default result to be produced by the operation. Armv8 also optionally supports the trapping of floating-point exceptions, see *Floating-point exceptions and exception traps* on page G1-5776. For more information about floating-point exceptions see *Floating-point exceptions and exception traps* on page E1-3963.

The Advanced SIMD and floating-point instructions also provide the following conversion functions:

- Between half-precision floating-point and single-precision floating point, in both directions.
- From double-precision, floating point to a signed single precision integer.
- When ARMv8.2-AA32BF16 is implemented, between single-precision floating-point and BFloat16 floating-point.

Some Advanced SIMD instructions support polynomial arithmetic over {0, 1}, as described in *Polynomial arithmetic over {0, 1}* on page A1-50.

For system level information about the Advanced SIMD and Floating-point implementation see *Advanced SIMD and floating-point support* on page G1-5772.

The following sections give more information about the Advanced SIMD and floating-point instructions:

- *Floating-point standards, and terminology* on page A1-54.
- *The SIMD and floating-point register file*.
- *Data types supported by the Advanced SIMD implementation* on page E1-3962.
- *Advanced SIMD and floating-point System registers* on page E1-3962.
- *Floating-point data types and arithmetic* on page E1-3962.
- *Floating-point exceptions and exception traps* on page E1-3963.
- *Controls of Advanced SIMD operation that do not apply to floating-point operation* on page E1-3966.
- *Implications of not including Advanced SIMD and floating-point support* on page E1-3966.
- *Pseudocode description of floating-point operations* on page E1-3967.

E1.3.1 The SIMD and floating-point register file

The Advanced SIMD and floating-point instructions use the same register file, that comprises 32 registers. This is distinct from the register file that holds the general-purpose registers and the PC.

The Advanced SIMD and floating-point views of the register file are different. The following sections describe these different views. Figure E1-1 on page E1-3961 shows the views of the register file, and the way the word, doubleword, and quadword registers overlap.

Advanced SIMD views of the register file

Advanced SIMD can view this register file as:

- Sixteen 128-bit quadword registers, `Q0-Q15`.
- Thirty-two 64-bit doubleword registers, `D0-D31`.

These views can be used simultaneously. For example, a program might hold 64-bit vectors in `D0` and `D1` and a 128-bit vector in `Q1`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Floating-point views of the register file

The Advanced SIMD and floating-point register file consists of thirty-two doubleword registers, that can be viewed as:

- Thirty-two 64-bit doubleword registers, `D0-D31`. This view is also available to Advanced SIMD instructions.
- Thirty-two 32-bit single word registers, `S0-S31`. Only half of the set is accessible in this view.

Note

In AArch32 state, half-precision floating point values are always represented using the bottom 16 bits of a single word register, `S0-S31`. When a half-precision value is written to a single word register, the top 16 bits of that register are set to 0.

The two views can be used simultaneously.

SIMD and Floating-point register file mapping onto registers

Figure E1-1 shows the different views of the SIMD and floating-point register file, and the relationship between them.

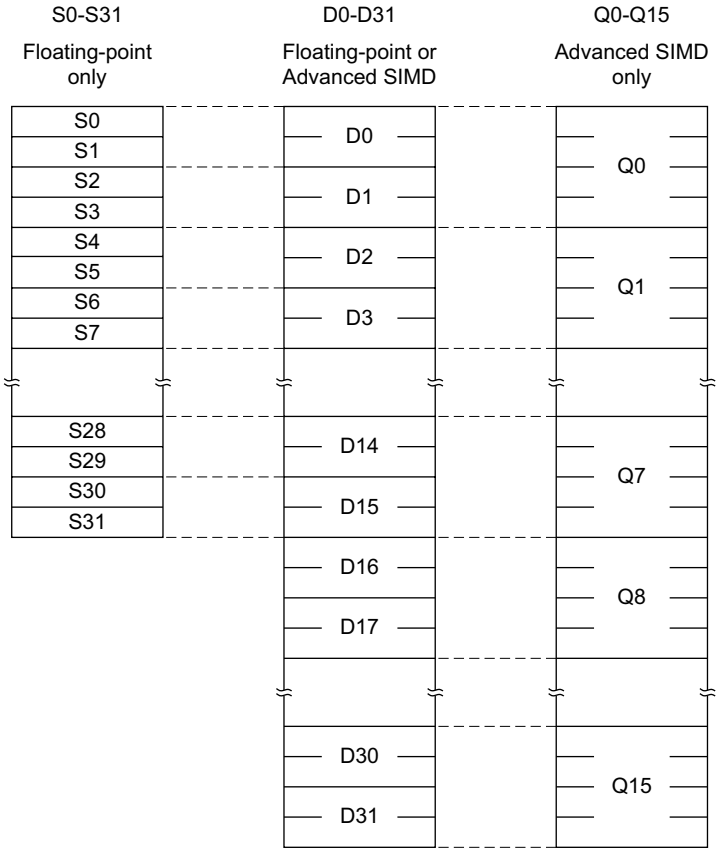


Figure E1-1 SIMD and floating-point register file, AArch32 operation

The mapping between the registers is as follows:

- `S<2n>` maps to the least significant half of `D<n>`.
- `S<2n+1>` maps to the most significant half of `D<n>`.
- `D<2n>` maps to the least significant half of `Q<n>`.
- `D<2n+1>` maps to the most significant half of `Q<n>`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For example, software can access the least significant half of the elements of a vector in `Q6` by referring to `D12`, and the most significant half of the elements by referring to `D13`.

Pseudocode description of the SIMD and Floating-point register file

The functions `_Dclone`, `S[]`, and `D[]` provide the `S0-S31`, `D0-D31`, and `Q0-Q15` views of the Advanced SIMD and floating-point registers:

The `Din[]` function returns a Doubleword register from the `_Dclone[]` copy of the SIMD and Floating-point register file, and the `Qin[]` function returns a Quadword register from that register file.

Note

The `CheckAdvSIMDEnabled()` function copies the `D[]` register file to `_Dclone[]`, see *Pseudocode description of enabling SIMD and floating-point functionality* on page G1-5811.

E1.3.2 Data types supported by the Advanced SIMD implementation

Advanced SIMD instructions can operate on integer and floating-point data, and the implementation defines a set of data types that support the required data formats. *Vector formats in AArch32 state* on page A1-42 describes these formats.

Advanced SIMD vectors

In an implementation that includes support for Advanced SIMD operation, a register can hold one or more packed elements, all of the same size and type. The combination of a register and a data type describes a vector of elements. The vector is considered to be an array of elements of the data type specified in the instruction. The number of elements in the vector is implied by the size of the data elements and the size of the register.

Vector indices are in the range 0 to (number of elements – 1). An index of 0 refers to the least significant end of the vector. In *Vector formats in AArch32 state* on page A1-42, Figure A1-3 on page A1-44 shows the Advanced SIMD vector formats.

Pseudocode description of Advanced SIMD vectors

The pseudocode function `Elem[]` accesses the element of a specified index and size in a vector.

E1.3.3 Advanced SIMD and floating-point System registers

The Advanced SIMD and floating-point instructions have a shared register space for System registers. The only register in this space that is accessible at the Application level is the FPSCR.

Writes to the FPSCR can have side-effects on various aspects of PE operation. All of these side-effects are synchronous to the FPSCR write. This means they are guaranteed not to be visible to earlier instructions in the execution stream, and they are guaranteed to be visible to later instructions in the execution stream.

See *Advanced SIMD and floating-point System registers* on page G1-5774 for the system level view of the registers.

These registers can be described as the *SIMD and floating-point System registers*.

E1.3.4 Floating-point data types and arithmetic

The T32 and A32 floating-point instructions support single-precision (32-bit) and double-precision (64-bit) data types and arithmetic as defined by the IEEE 754 floating-point standard. They also support the half-precision (16-bit) floating-point data type for data storage, by supporting conversions between single-precision and half-precision data types. When ARMv8.2-FP16 is implemented, it also supports the half-precision floating-point data type for data processing operations. When ARMv8.2-AA32BF16 is implemented, it also supports the BFloat16 floating-point storage format.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Arm standard floating-point arithmetic means IEEE 754 floating-point arithmetic with the restrictions described in *Advanced SIMD and floating-point support* on page A1-52, including supporting only the input and output values described in *Arm standard floating-point input and output values* on page A1-54.

The AArch32 Advanced SIMD instructions support single-precision and, when ARMv8.2-FP16 is implemented, half-precision Arm standard floating-point arithmetic.

The following sections describe the Advanced SIMD and floating-point formats:

- *Half-precision floating-point formats* on page A1-44.
- *Single-precision floating-point format* on page A1-46.
- *Double-precision floating-point format* on page A1-47.
- *BFloat16 floating-point format* on page A1-48.

The following sections describe features of Advanced SIMD and floating-point processing:

- *Flush-to-zero* on page A1-55.
- *NaN handling and the Default NaN* on page A1-56.

E1.3.5 Floating-point exceptions and exception traps

Arm Advanced SIMD and floating-point instructions record the following floating-point exceptions in the FPSCR cumulative bits, unless the floating-point exception is trapped and generates an exception:

FPSCR.IOC Invalid Operation. The bit is set to 1 if the result of an operation has no mathematical value or cannot be represented. Cases include, for example:

- (infinity) × 0.
- (+infinity) + (–infinity).

These tests are made after flush-to-zero processing. For example, if flush-to-zero mode is selected, multiplying a denormalized number and an infinity is treated as (0 × infinity), and causes an Invalid Operation floating-point exception.

IOC is also set on any floating-point operation with one or more signaling NaNs as operands, except for negation and absolute value, as described in *Floating-point negation and absolute value* on page E1-3967.

FPSCR.DZC Divide by Zero. The bit is set to 1 if a divide operation has a zero divisor and a dividend that is not zero, an infinity or a NaN. These tests are made after flush-to-zero processing, so if flush-to-zero processing is selected, a denormalized dividend is treated as zero and prevents Divide by Zero from occurring, and a denormalized divisor is treated as zero and causes Divide by Zero to occur if the dividend is a normalized number.

For the reciprocal and reciprocal square root estimate functions the dividend is assumed to be +1.0. This means that a zero or denormalized operand to these functions sets the DZC bit.

FPSCR.OFC Overflow. The bit is set to 1 if the absolute value of the result of an operation, produced after rounding, is greater than the maximum positive normalized number for the destination precision.

FPSCR.UFC Underflow. The bit is set to 1 if the absolute value of the result of an operation, produced before rounding, is less than the minimum positive normalized number for the destination precision, and the rounded result is inexact.

The criteria for the Underflow floating-point exception to occur are different in Flush-to-zero mode. For details, see *Flush-to-zero* on page A1-55.

FPSCR.IXC Inexact. The bit is set to 1 if the result of an operation is not equivalent to the value that would be produced if the operation were performed with unbounded precision and exponent range.

The criteria for the Inexact floating-point exception to occur are different in Flush-to-zero mode. For details, see *Flush-to-zero* on page A1-55.

FPSCR.IDC Input Denormal. The bit is set to 1 if a denormalized input operand is replaced in the computation by a zero, as described in *Flush-to-zero* on page A1-55.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For Advanced SIMD instructions, and for floating-point instructions when floating-point exception trapping is not supported, these are non-trapping exceptions and the data-processing instructions do not generate any trapped exceptions.

For floating-point instructions when floating-point exception trapping is supported:

- The floating-point exceptions can be trapped, by setting trap enable bits in the FPSCR, see *Floating-point exceptions and exception traps* on page G1-5776, and:
  - When a trap is not enabled the corresponding floating-point exception updates the corresponding FPSCR cumulative bit does not generate an exception.
  - When a trap is enabled the corresponding floating-point exception does not update the FPSCR, but generates an exception. In this case, bits in the FPEXC indicate which floating-point exceptions have occurred.
- The definition of the Underflow floating-point exception is different in the trapped and cumulative exception cases. In the trapped case the definition is:
  - The trapped Underflow floating-point exception occurs if the absolute value of the result of an operation, produced before rounding, is less than the minimum positive normalized number for the destination precision, regardless of whether the rounded result is inexact.
- As with cumulative exceptions, higher priority trapped exceptions can prevent lower priority exceptions from occurring, as described in *Combinations of floating-point exceptions* on page E1-3965.
- For Invalid Operation floating-point exceptions, for details of which quiet NaN is produced as the default result see *NaN handling and the Default NaN* on page A1-56.
- For Overflow floating-point exceptions, the sign bit of the default result is determined normally for the overflowing operation.
- For Divide by Zero floating-point exceptions, the sign bit of the default result is determined normally for a division. This means it is the exclusive OR of the sign bits of the two operands.

Table E1-4 shows the results of untrapped floating-point exceptions. That table uses the following abbreviations:

MaxNorm The maximum normalized number of the destination precision.

RM Round towards Minus Infinity mode, as defined in the IEEE 754 standard.

RN Round to Nearest mode, as defined in the IEEE 754 standard.

RP Round towards Plus Infinity mode, as defined in the IEEE 754 standard.

RZ Round towards Zero mode, as defined in the IEEE 754 standard.

For more information about the IEEE 754 descriptions of the rounding modes see *Floating-point standards, and terminology* on page A1-54.

Table E1-4 Results of untrapped floating-point exceptions

Exception type	Default result for positive sign		Default result for negative sign	
IOC, Invalid Operation	Quiet NaN		Quiet NaN	
DZC, Divide by Zero	+infinity		-infinity	
OFC, Overflow	RN, RP: RM, RZ:	+infinity +MaxNorm	RN, RM: RP, RZ:	-infinity -MaxNorm
UFC, Underflow	Normal rounded result		Normal rounded result	
IXC, Inexact	Normal rounded result		Normal rounded result	
IDC, Input Denormal	Normal rounded result		Normal rounded result	

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Combinations of floating-point exceptions

Many pseudocode functions perform *floating-point operations*, including `FixedToFP()`, `FPAdd()`, `FPCompare()`, `FPCompareEQ()`, `FPCompareGE()`, `FPCompareGT()`, `FPDiv()`, `FPMax()`, `FPMin()`, `FPMul()`, `FPMulAdd()`, `FPRecipEstimate()`, `FPRecipStep()`, `FPRSqrtEstimate()`, `FPRSqrtStep()`, `FPSqrt()`, `FPSub()`, and `FPToFixed()`. All of these operations can generate floating-point exceptions.

Note

`FPAbs()` and `FPNeg()` are not classified as *floating-point operations* because:

- They cannot generate floating-point exceptions.
- The floating-point operation behavior described in the following sections does not apply to them:
  - *Flush-to-zero* on page A1-55.
  - *NaN handling and the Default NaN* on page A1-56.

More than one exception can occur on the same operation. The only combinations of floating-point exceptions that can occur are:

- Overflow with Inexact.
- Underflow with Inexact.
- Input Denormal with other floating-point exceptions.

The priority order of these floating-point exceptions is that the Inexact exception is treated as lowest priority, and the Input Denormal exception is treated as highest priority.

When none of the floating-point exceptions caused by an operation is trapped, any floating-point exception that occurs causes the associated cumulative bit in the FPSCR to be set.

When one or more floating-point exceptions caused by an operation is trapped, the behavior of the instruction depends on the priority of the exceptions:

- If the higher priority floating-point exception is trapped, its trap handler is called. It is IMPLEMENTATION DEFINED whether any information about the lower priority floating-point exception is provided.

  Note

  Information about the lower priority floating-point exception might be provided in:

  - The FPEXC, if the exception generated by the trap is taken to an Exception level that is using AArch32.
  - The ESR_ELx.ISS field, if the exception generated by the trap is taken to an Exception level that is using AArch64.

  However, information might be provided in another IMPLEMENTATION DEFINED way, for example using an IMPLEMENTATION DEFINED register.

  Apart from this, the lower priority floating-point exception is ignored in this case.

- If the higher priority floating-point exception is untrapped, its cumulative bit is set to 1 and its default result is evaluated. Then the lower priority floating-point exception is handled normally, using this default result.

Some floating-point instructions specify more than one floating-point operation, as indicated by the pseudocode descriptions of the instruction. In such cases, a floating-point exception on one operation is treated as higher priority than a floating-point exception on another operation if the occurrence of the second floating-point exception depends on the result of the first operation. Otherwise, it is CONSTRAINED UNPREDICTABLE which floating-point exception is treated as higher priority.

For example, a `VMLA.F32` instruction specifies a floating-point multiplication followed by a floating-point addition. The addition can generate Overflow, Underflow and Inexact floating-point exceptions, all of which depend on both operands to the addition and so are treated as lower priority than any floating-point exception on the multiplication. The same applies to Invalid Operation floating-point exceptions on the addition caused by adding opposite-signed infinities. The addition can also generate an Input Denormal floating-point exception, caused by the addend being a denormalized number while in Flush-to-zero mode. It is CONSTRAINED UNPREDICTABLE which of an Input Denormal floating-point exception on the addition and a floating-point exception on the multiplication is treated as

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

higher priority, because the occurrence of the Input Denormal floating-point exception does not depend on the result of the multiplication. The same applies to an Invalid Operation floating-point exception on the addition caused by the addend being a signaling NaN.

Note

The VFMA instruction performs a vector addition and a vector multiplication as a single operation. The VFMS instruction performs a vector subtraction and a vector multiplication as a single operation.

E1.3.6 Controls of Advanced SIMD operation that do not apply to floating-point operation

Armv7 permitted implementation of either, both, or neither of the Advanced SIMD and floating-point additions to the base instruction set, and provided some controls that applied to the Advanced SIMD functionality but not to the floating-point functionality. In Armv8, Advanced SIMD functionality cannot be separated from floating-point functionality, but in AArch32 state these controls function as they did in Armv7. This means they apply only to the following instructions and instruction encodings:

- All instructions with encodings defined in:
  - *Advanced SIMD data-processing* on page F3-4137, for the T32 instruction set.
  - *Advanced SIMD data-processing* on page F4-4225, for the A32 instruction set.
- All instructions with encodings defined in:
  - *Advanced SIMD element or structure load/store* on page F3-4160, for the T32 instruction set.
  - *Advanced SIMD element or structure load/store* on page F4-4237, for the A32 instruction set.
- The form of the VDUP instruction described in *VDUP (general-purpose register)* on page F6-5155.
- The byte and halfword forms of the VMOV instructions described in each of:
  - *VMOV (general-purpose register to scalar)* on page F6-5331.
  - *VMOV (scalar to general-purpose register)* on page F6-5335.

The controls of this functionality are:

- The CPACR.ASEDIS field.
- The HCPTR.TASE field.

In an implementation that supports Advanced SIMD functionality, support for each of these controls is optional:

- If the CPACR.ASEDIS control is not supported then the CPACR.ASEDIS field is RAZ/WI. This is equivalent to the control permitting the execution of Advanced SIMD instructions at EL1 and EL0.
- If the HCPTR.TASE control is not supported then the HCPTR.TASE field is RAZ/WI. This means the HCPTR does not provide a control that can trap Non-secure execution of Advanced SIMD instructions to Hyp mode.

E1.3.7 Implications of not including Advanced SIMD and floating-point support

In general, Armv8 requires the inclusion of the Advanced SIMD and floating-point instructions in all instruction sets. Exceptionally, for implementation targeting specialized markets, Arm might produce or license an Armv8-A implementation that does not provide any support for Advanced SIMD and floating-point instructions. In such an implementation, in AArch32 state:

- Each of the CPACR.{cp10, cp11} fields is RES0.
- The CPACR.ASEDIS bit is RES0.
- Each of the HCPTR.{TASE, TCP10, TCP11} fields is RES1.
- Each of the NSACR.{NSASEDIS, cp10, cp11} fields is RES0.
- The FPEXC register is UNDEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

E1.3.8 Pseudocode description of floating-point operations

The following subsections contain pseudocode definitions of the floating-point functionality supported by the Armv8 architecture:

- *Generation of specific floating-point values*.
- *Floating-point negation and absolute value*.
- *Floating-point value unpacking*.
- *Floating-point exception and NaN handling*.
- *Floating-point rounding* on page E1-3968.
- *Selection of Arm standard floating-point arithmetic* on page E1-3968.
- *Floating-point comparisons* on page E1-3968.
- *Floating-point maximum and minimum* on page E1-3968.
- *Floating-point addition and subtraction* on page E1-3968.
- *Floating-point multiplication and division* on page E1-3968.
- *Floating-point fused multiply-add* on page E1-3968.
- *Floating-point reciprocal estimate and step* on page E1-3969.
- *Floating-point square root* on page E1-3969.
- *Floating-point reciprocal square root estimate and step* on page E1-3969.
- *Floating-point conversions* on page E1-3970.

Generation of specific floating-point values

The following pseudocode functions generate specific floating-point values. The `sign` argument is `'0'` for the positive version and `'1'` for the negative version:

- `FPInfinity()`.
- `FPMaxNormal()`.
- `FPZero()`.
- `FPTwo()`.
- `FPThree()`.
- `FPDefaultNaN()`.

Floating-point negation and absolute value

The floating-point negation and absolute value operations only affect the sign bit. They do not treat NaN operands specially, nor denormalized number operands when flush-to-zero is selected.

The floating-point negation operation is described by the pseudocode function `FPNeg()`. The floating-point absolute value operation is described by the pseudocode function `FPAbs()`.

Floating-point value unpacking

The `FPUnpack()` function determines the type of a floating-point number, defined by `FPType{}`, and its numerical value. It also does flush-to-zero processing on input operands.

Floating-point exception and NaN handling

The `FPProcessException()` procedure checks whether a floating-point exception is trapped, and handles it accordingly. The floating-point exception types are defined by `FPExc{}`.

The `FPProcessNaN()` function processes a NaN operand, producing the correct result value and generating an Invalid Operation floating-point exception if necessary. The `FPProcessNaNs()` function performs the standard NaN processing for a two-operand operation. The `FPProcessNaNs3()` function performs the standard NaN processing for a three-operand operation.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Floating-point rounding

The `FPRound()` function rounds and encodes a floating-point result to a specified destination format. This includes processing Overflow, Underflow and Inexact floating-point exceptions and performing flush-to-zero processing on result values.

Selection of Arm standard floating-point arithmetic

The `StandardFPSCRValue()` function returns the FPSCR value that selects Arm standard floating-point arithmetic. Most of the arithmetic functions have a Boolean `fpscr_controlled` argument that is `TRUE` for Floating-point operations and `FALSE` for Advanced SIMD operations, and that selects between using the real FPSCR value and this value.

Floating-point comparisons

The `FPCompare()` function compares two floating-point numbers, producing a {N, Z, C, V} Condition flags result as shown in Table E1-5:

Table E1-5 Effect of a Floating-point comparison on the Condition flags

Comparison result	N	Z	C	V
Equal	0	1	1	0
Less than	1	0	0	0
Greater than	0	0	1	0
Unordered	0	0	1	1

This result defines the operation of the `VCMP` floating-point instruction. The `VCMP` instruction writes these flag values in the FPSCR. After using a `VMRS` instruction to transfer them to the APSR, they can control conditional execution as shown in Table F2-1 on page F2-4081.

The `FPCompareEQ()`, `FPCompareGE()`, and `FPCompareGT()` functions describe the operation of Advanced SIMD instructions that perform floating-point comparisons.

Floating-point maximum and minimum

The `FPMax()` function returns the maximum of two floating-point numbers. The `FPMin()` function returns the minimum of two floating-point numbers.

Floating-point addition and subtraction

The `FPAdd()` function adds two floating-point numbers. The `FPSub()` function subtracts one floating-point number from another floating-point number.

Floating-point multiplication and division

The `FPMul()` function multiplies two floating-point numbers. The `FPDiv()` function divides one floating-point number by another floating-point number.

Floating-point fused multiply-add

The `FPMulAdd()` function performs a floating-point fused multiply-add.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Floating-point reciprocal estimate and step

The Advanced SIMD implementation includes instructions that support Newton-Raphson calculation of the reciprocal of a number.

The `VRECPE` instruction produces the initial estimate of the reciprocal. It uses the pseudocode functions:

- `FPRecipEstimate()`.
- `UnsignedRecipEstimate()`.

Table E1-6 shows the results where input values are out of range.

Table E1-6 VRECPE results for out of range inputs

Number type	Input Vm[i]	Result Vd[i]
Integer	<= `0x7FFFFFFF`	`0xFFFFFFFF`
Floating-point	NaN	Default NaN
Floating-point	±0 or denormalized number	±infinity [a]
Floating-point	±infinity	±0
Floating-point	Absolute value >= 2^{126}	±0

a. FPSCR.DZC is set to 1

The Newton-Raphson iteration:

$x_{n+1} = x_n(2-dx_n)$

converges to $(1/d)$ if x_0 is the result of `VRECPE` applied to d.

The `VRECPS` instruction performs a (2 - op1×op2) calculation and can be used with a multiplication to perform a step of this iteration. The functionality of this instruction is defined by the `FPRecipStep()` pseudocode function.

Table E1-7 shows the results where input values are out of range.

Table E1-7 VRECPS results for out of range inputs

Input Vn[i]	Input Vm[i]	Result Vd[i]
Any NaN	-	Default NaN
-	Any NaN	Default NaN
±0.0 or denormalized number	±infinity	2.0
±infinity	±0.0 or denormalized number	2.0

Floating-point square root

The `FPSqrt()` function returns the square root of a floating-point number.

Floating-point reciprocal square root estimate and step

The Advanced SIMD implementation includes instructions that support Newton-Raphson calculation of the reciprocal of the square root of a number.

The `VRSQRTE` instruction produces the initial estimate of the reciprocal of the square root. It uses the pseudocode functions:

- `FPRSqrtEstimate()`.
- `UnsignedRSqrtEstimate()`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table E1-8 shows the results where input values are out of range.

Table E1-8 VRSQRTE results for out of range inputs

Number type	Input Vm[i]	Result Vd[i]
Integer	<= `0x3FFFFFFF`	`0xFFFFFFFF`
Floating-point	NaN, –(normalized number), –infinity	Default NaN
Floating-point	–0 or –(denormalized number)	– infinity [a]
Floating-point	+0 or +(denormalized number)	+infinity [a]
Floating-point	+infinity	+0

a. FPSCR.DZC is set to 1.

The Newton-Raphson iteration:

$x_{n+1} = x_n(3-dx_n^2)/2$

converges to $(1/\sqrt{d})$ if x_0 is the result of `VRSQRTE` applied to d.

The `VRSQRTS` instruction performs a (3 – op1×op2)/2 calculation and can be used with two multiplications to perform a step of this iteration. The functionality of this instruction is defined by the `FPRSqrtStep()` pseudocode function.

Table E1-9 shows the results where input values are out of range.

Table E1-9 VRSQRTS results for out of range inputs

Input Vn[i]	Input Vm[i]	Result Vd[i]
Any NaN	-	Default NaN
-	Any NaN	Default NaN
±0.0 or denormalized number	±infinity	1.5
±infinity	±0.0 or denormalized number	1.5

`FPRSqrtStep()` calls the `FPHalvedSub()` pseudocode function.

Floating-point conversions

The `FPConvert()` pseudocode function performs conversions between half-precision, single-precision, and double-precision floating-point numbers.

The `FPToFixed()` and `FixedToFP()` functions perform conversions between floating-point numbers and integers or fixed-point numbers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

E1.4 About the AArch32 System register interface

AArch32 state provides a System register encoding space, that is indexed by the parameter set {`coproc`, `opc1`, `CRn`, `CRm`, `opc2`}, and a set of System register access instructions. This encoding space is used for:

- System registers.
- System instructions, for:
  - Cache and branch predictor maintenance.
  - Address translation.
  - TLB maintenance.

In Armv8, this encoding space uses only the `coproc` values `0b111x`.

Note

The encoding space with `coproc` values `0b101x` is redefined to provide Advanced SIMD and floating-point functionality.

In Armv8:

- The (`coproc==0b1111`) encodings provide system control functionality, by providing access to System registers and System instructions. This includes architecture and feature identification, as well as control, status information and configuration support.

  The following sections give a general description of these encodings:
  - *About the System registers for VMSAv8-32* on page G5-6052.
  - *VMSAv8-32 organization of registers in the (coproc==0b1111) encoding space* on page G7-6078.
  - *Functional grouping of VMSAv8-32 System registers* on page G5-6058.

  These encodings also provide:
  - The Performance Monitor registers. For more information, see Chapter D7 *The Performance Monitors Extension*.

    The Activity Monitor registers. For more information, see Chapter D8 *The Activity Monitors Extension*.
- The (`coproc==0b1110`) encodings provide access to additional registers, that support:
  - Debug, see Chapter G2 *AArch32 Self-hosted Debug*.
  - The Jazelle identification registers, see *Jazelle support* on page E1-3959.

UNPREDICTABLE, CONSTRAINED UNPREDICTABLE, and UNDEFINED behavior for AArch32 System register accesses on page G8-6097 gives information more information about permitted accesses to the System registers in AArch32 state.

Most functionality in the (`coproc==0b111x`) encoding space cannot be accessed by software executing at EL0. This manual clearly identifies those functions that can be accessed at EL0.

For more information:

- About this encoding space, including the naming of the parameters that index the space, see *The AArch32 System register interface* on page G1-5769.
- About the System interface access instructions, see *System register access instructions* on page F1-4057.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

E1.5 Exceptions

The Arm architecture uses the following terms to describe various types of exceptional condition:

Exceptions In the Arm architecture, an *exception* causes entry to EL1, EL2, or EL3. If the Exception level that is entered is using AArch32, it also causes entry to the PE mode in which the exception must be taken. A software handler for the exception is then executed.

Note

The term *floating-point exception* does not use this meaning of *exception*. This term is described later in this list.

Exceptions include:

- Reset.
- Interrupts.
- Memory system aborts.
- Undefined instructions.
- Supervisor calls (SVCs), Secure Monitor calls (SMCs), and hypervisor calls (HVCs).
- Debug exceptions.

Most details of exception handling are not visible to application level software, and are described in *Handling exceptions that are taken to an Exception level using AArch32* on page G1-5703. In an Armv8 implementation that includes all the Exception levels, aspects that are visible to application level software are:

- The `SVC` instruction causes a Supervisor Call exception. This provides a mechanism for unprivileged software to make a call to the operating system, or other system component that is accessible only at EL1.
- The `SMC` instruction causes a Secure Monitor Call exception, but only if software execution is at EL1 or higher. Unprivileged software can only cause a Secure Monitor Call exception by methods defined by the operating system, or by another component of the software system that executes at EL1 or higher.
- The `HVC` instruction causes a Hypervisor Call exception, but only if software execution is at EL1 or higher. Unprivileged software can only cause a Hypervisor Call exception by methods defined by the hypervisor, or by another component of the software system that executes at EL1 or higher.
- The `BKPT` instruction causes a Breakpoint Instruction exception, that is taken as a Prefetch Abort exception. This provides a mechanism for a debugger to insert breakpoints into unprivileged software, or for unprivileged software to make a call into a debugger that is accessible at EL1.
- The `WFI` (Wait for Interrupt) instruction provides a hint that nothing needs to be done until an interrupt or another WFI wake-up event occurs, see *Wait For Interrupt* on page G1-5767. This means the hardware might enter a low-power state until the wake-up event occurs.
- The `WFE` (Wait for Event) instruction provides a hint that nothing needs to be done until either an `SEV` instruction generates an event, or another WFE wake-up event occurs, see *Wait For Event and Send Event* on page G1-5764. This means the hardware might enter a low-power state until the wake-up event occurs.

Floating-point exceptions

These relate to exceptional conditions encountered during floating-point arithmetic, such as Divide by Zero or Overflow. For more information see:

- *Floating-point exceptions and exception traps* on page E1-3963.
- The FPEXC and FPSCR register descriptions.
- ANSI/IEEE Std. 754, *IEEE Standard for Binary Floating-Point Arithmetic*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter E2
The AArch32 Application Level Memory Model

This chapter gives an application level description of the memory model for software executing in AArch32 state. This means it describes the memory model for execution in EL0 when EL0 is using AArch32 in the following sections:

- *About the Arm memory model* on page E2-3974.
- *Atomicity in the Arm architecture* on page E2-3976.
- *Definition of the Armv8 memory model* on page E2-3980.
- *Ordering of translation table walks* on page E2-3998.
- *Caches and memory hierarchy* on page E2-3999.
- *Alignment support* on page E2-4004.
- *Endian support* on page E2-4006.
- *Memory types and attributes* on page E2-4010.
- *Mismatched memory attributes* on page E2-4020.
- *Synchronization and semaphores* on page E2-4023

Note

In this chapter, System register names usually link to the description of the register in Chapter G8 *AArch32 System Register Descriptions*, for example SCTLR.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

E2.1 About the Arm memory model

The Arm architecture is a weakly ordered memory architecture that permits the observation and completion of memory accesses in a different order from the program order. The following sections of this chapter provide the complete definition of the Armv8 memory model, this introduction is not intended to contradict the definition found in those sections. In general, the basic principles of the Armv8 memory model are:

- To provide a memory model that has similar weaknesses to those found in the memory models used by high-level programming languages such as C or Java. For example, by permitting independent memory accesses to be re-ordered as seen by other observers.
- To avoid the requirement for multi-copy atomicity in the majority of memory types.
- The provision of instructions and memory barriers to compensate for the lack of multi-copy atomicity in the cases where it would be needed.
- The use of address, data, and control dependencies in the creation of order so as to avoid having excessive numbers of barriers or other explicit instructions in common situations where some order is required by the programmer or the compiler.

This section contains:

- *Address space*.
- *Memory type overview*.

E2.1.1 Address space

Address calculations are performed using 32-bit registers. Supervisory software determines the valid address range.

Attempting to access an address that is not valid generates an MMU fault.

Address calculations are performed modulo 2^{32}.

The result of an address calculation is UNKNOWN if it overflows or underflows the 32-bit address range A[31:0].

Memory accesses use the `MemA[]`, `MemO[]`, `MemU[]`, and `MemU_unpriv[]` pseudocode functions:

- The `MemA[]` function makes an aligned access of the required type.
- The `MemO[]` function makes an ordered access of the required type.
- The `MemU[]` function makes an unaligned access of the required type
- The `MemU_unpriv[]` function makes an unaligned, unprivileged access of the required type.

Each of these functions calls `Mem_with_type[]` function, that specifies the required access. This calls `AArch32.MemSingle[]`, which performs an atomic, little-endian read of `size` bytes.

The `AccType` enumeration defines the different access types.

Note

- Chapter G4 *The AArch32 System Level Memory Model* and Chapter G5 *The AArch32 Virtual Memory System Architecture* include descriptions of memory system features that are transparent to the application, including memory access, address translation, memory maintenance instructions, and alignment checking and the associated fault handling. These chapters also reference pseudocode descriptions of these operations.
- For references to the pseudocode that relates to memory accesses, see *Basic memory access* on page G4-5917, *Unaligned memory access* on page G4-5918, and *Aligned memory access* on page G4-5917.

E2.1.2 Memory type overview

Armv8 provides the following mutually-exclusive memory types:

Normal This is generally used for bulk memory operations, both read-write and read-only operations.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Device The Arm architecture forbids speculative reads of any type of Device memory. This means Device memory types are suitable attributes for read-sensitive locations.

Locations of the memory map that are assigned to peripherals are usually assigned the Device memory attribute.

Device memory has additional attributes that have the following effects:

- They prevent aggregation of reads and writes, maintaining the number and size of the specified memory accesses. See *Gathering* on page E2-4016.
- They preserve the access order and synchronization requirements, both for accesses to a single peripheral and where there is a synchronization requirement on the observability of one or more memory write and read accesses. See *Reordering* on page E2-4017
- They indicate whether a write can be acknowledged other than at the end point. See *Early Write Acknowledgement* on page E2-4018.

For more information on Normal memory and Device memory, see *Memory types and attributes* on page E2-4010.

Note

Earlier versions of the Arm architecture defined a single Device memory type and a Strongly-Ordered memory type. A Note in *Device memory* on page E2-4014 describes how these memory types map onto the Armv8 memory types.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

E2.2 Atomicity in the Arm architecture

Atomicity is a feature of memory accesses, described as *atomic* accesses. The Arm architecture description refers to two types of atomicity, *single-copy atomicity* and *multi-copy atomicity*. In the Armv8 architecture, the atomicity requirements for memory accesses depend on the memory type, and whether the access is explicit or implicit. For more information see:

- *Requirements for single-copy atomicity*.
- *Properties of single-copy atomic accesses* on page E2-3977.
- *Multi-copy atomicity* on page E2-3977.
- *Requirements for multi-copy atomicity* on page E2-3978.
- *Concurrent modification and execution of instructions* on page E2-3978.

For more information about the memory types, see *Memory type overview* on page E2-3974.

E2.2.1 Requirements for single-copy atomicity

In AArch32 state, the single-copy atomic PE accesses are:

- All byte accesses.
- All halfword accesses to halfword-aligned locations.
- All word accesses to word-aligned locations.
- Memory accesses caused by `LDREXD` and `STREXD` instructions to doubleword-aligned locations.

`LDM`, `LDC`, `LDRD`, `STM`, `STC`, `STRD`, `PUSH`, `POP`, `RFE`, `SRS`, `VLDM`, `VLDR`, `VSTM`, and `VSTR` instructions are executed as a sequence of word-aligned word accesses. Each 32-bit word access is guaranteed to be single-copy atomic. The architecture does not require subsequences of two or more word accesses from the sequence to be single-copy atomic.

`LDRD` and `STRD` accesses to 64-bit aligned locations are 64-bit single-copy atomic as seen by translation table walks and accesses to translation tables.

Note

This requirement has been added to avoid the need for complex measures to avoid atomicity issues when changing translation table entries, without creating a requirement that all locations in the memory system are 64-bit single-copy atomic. This addition means:

- The system designer must ensure that all writable memory locations that might be used to hold translations, such as bulk SDRAM, can be accessed with 64-bit single-copy atomicity.
- Software must ensure that translation tables are not held in memory locations that cannot meet this atomicity requirement, such as peripherals that are typically accessed using a narrow bus.

This requirement places no burden on read-only memory locations for which reads have no side effects, since it is impossible to detect the size of memory accesses to such locations.

Advanced SIMD element and structure loads and stores are executed as a sequence of accesses of the element or structure size. The architecture requires the element accesses to be single-copy atomic if and only if both:

- The element size is 32 bits, or smaller.
- The elements are naturally aligned.

Accesses to 64-bit elements or structures that are 32-bit aligned are executed as a sequence of 32-bit accesses, each of which is single-copy atomic. The architecture does not require subsequences of two or more 32-bit accesses from the sequence to be single-copy atomic.

When an access is not single-copy atomic by the rules described in this section, it is executed as a sequence of one or more accesses that aggregate to the size of the original access. Each of the accesses in this sequence is single-copy atomic, at least at the byte level.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

In this section, the terms *before the write operation* and *after the write operation* mean before or after the write operation has had its effect on the coherence order of the bytes of the memory location accessed by the write operation.

If, according to these rules, an instruction is executed as a sequence of accesses, a synchronous Data Abort exception or Debug state entry can be taken during that sequence. This causes execution of the instruction to be abandoned. See *Data Abort exception* on page G1-5749 and, when ARMv8.2-LSMAOC is implemented, *Taking an interrupt or other exception during a multiple-register load or store* on page G1-5737.

If the synchronous Data Abort exception is returned from using the preferred return address, the instruction that generated the sequence of accesses is re-executed and so any access that was performed before the exception was taken is repeated. This also applies to an exit from Debug state.

Note

The exception behavior for these multiple access instructions means they are not suitable for use for writes to memory for the purpose of software synchronization.

For implicit accesses:

- Cache linefills and evictions have no effect on the single-copy atomicity of explicit transactions or instruction fetches.
- Instruction fetches are single-copy atomic:
  - At 32-bit granularity in A32 state.
  - At 16-bit granularity in T32 state.

  Concurrent modification and execution of instructions on page E2-3978 describes additional constraints on the behavior of instruction fetches.
- Translation table walks are performed using accesses that are single-copy atomic:
  - At 32-bit granularity when using Short-descriptor format translation tables.
  - At 64-bit granularity when using Long-descriptor format translation tables.

E2.2.2 Properties of single-copy atomic accesses

A memory access instruction that is single-copy atomic has the following properties:

1. For a pair of overlapping single-copy atomic store instructions, all of the overlapping writes generated by one of the stores are Coherence-after the corresponding overlapping writes generated by the other store.
2. For a single-copy atomic load instruction L_1 that overlaps a single-copy atomic store instruction S_2, if one of the overlapping reads generated by L_1 Reads-from one of the overlapping writes generated by S_2, then none of the overlapping writes generated by S_2 are Coherence-after the corresponding overlapping reads generated by L_1.

For more information, see *Definition of the Armv8 memory model* on page E2-3980.

E2.2.3 Multi-copy atomicity

In a multiprocessing system, writes to a memory location are *multi-copy atomic* if the following conditions are both true:

- All writes to the same location are *serialized*, meaning they are observed in the same order by all observers, although some observers might not observe all of the writes.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- A read of a location does not return the value of a write until all observers observe that write.

Note

Writes that are not coherent are not multi-copy atomic.

E2.2.4 Requirements for multi-copy atomicity

For Normal memory, writes are not required to be multi-copy atomic.

For Device memory, writes are not required to be multi-copy atomic.

The Armv8 memory model is Other-multi-copy atomic. For more information, see *Ordering constraints* on page E2-3984.

E2.2.5 Concurrent modification and execution of instructions

The Armv8 architecture limits the set of instructions that can be executed by one thread of execution as they are being modified by another thread of execution without requiring explicit synchronization.

Concurrent modification and execution of instructions can lead to the resulting instruction performing any behavior that can be achieved by executing any sequence of instructions that can be executed from the same Exception level, except where the instruction before modification and the instruction after modification are:

- When executing the A32 instruction set, a `B`, `BKPT`, `BL`, `HVC`, `ISB`, `NOP`, `SMC`, or `SVC` instruction.
- When executing the T32 instruction set. a 16-bit `B`, `BKPT`, `BLX`, `BX`, `NOP`, or `SVC` instruction.

In addition, for the 32-bit T32 instructions, for which *Instruction encodings* on page F2-4076 describes the meaning of `{hw1, hw2}`:

- `hw1` of a 32-bit `BL` (immediate) instruction can be concurrently modified to `hw1` of another `BL` (immediate) instruction:
  - This means that some of the most significant bits of the immediate value can be modified.
- `hw1` of a 32-bit `BLX` (immediate) instruction can be concurrently modified to `hw1` of another `BLX` immediate instruction:
  - This means that some of the most significant bits of the immediate value can be modified.
- `hw1` of a 32-bit `BL` (immediate) or `BLX` (immediate) instruction can be concurrently modified to a T32 16-bit `B`, `BX`, `BLX`, `BKPT`, or `SVC` instruction. This modification also works in reverse.
- `hw2` of a 32-bit `BL` (immediate) instruction can be concurrently modified to `hw2` of another `BL` (immediate) instruction with a different immediate:
  - This means that some bits of the immediate value, including the least significant bits, can be modified.
- `hw2` of a 32-bit `BLX` (immediate) instruction can be concurrently modified to `hw2` of another `BLX` (immediate) instruction with a different immediate:
  - This means that some bits of the immediate value, including the least significant bits, can be modified.
- `hw2` of a 32-bit `B` (immediate) instruction with a condition field can be concurrently modified to `hw2` of another 32-bit `B` (immediate) instruction with a condition field with a different immediate:
  - This means that some bits of the immediate value, including the least significant bits, can be modified.
- `hw2` of a 32-bit `B` (immediate) instruction without a condition field can be concurrently modified to `hw2` of another 32-bit `B` (immediate) instruction without a condition field:
  - This means that some bits of the immediate value, including the least significant bits, can be modified.

Note

- In the T32 instruction set:
  - The only encodings of `BKPT` and `SVC` are 16-bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— The only encoding of BL is 32-bit.

- The ISB instruction can be concurrently modified and executed in the A32 and A64 instruction sets, but not in the T32 instruction set.

For the instructions explicitly identified in this section, the architecture guarantees that, after modification of the instruction, behavior is consistent with execution of either:

- The instruction originally fetched.
- A fetch of the modified instruction.

The instructions to which this applies are the B, BL, NOP, BKPT, SVC, HVC, and SMC instructions.

For both instruction sets, if one thread of execution changes a conditional branch instruction to another conditional branch instruction, and the change affects both the condition field and the branch target, execution of the changed instruction by another thread of execution before the change is synchronized can lead to either:

- The old condition being associated with the new target address.
- The new condition being associated with the old target address.

These possibilities apply regardless of whether the condition, either before or after the change to the branch instruction, is the *always* condition.

For all other instructions, to avoid UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior, instruction modifications must be explicitly synchronized before they are executed. The required synchronization is as follows:

1. No PE must be executing an instruction when another PE is modifying that instruction.
2. To ensure that the modified instructions are observable, a PE that is writing the instructions must issue the following sequence of instructions and operations:

```
; Coherency example for self-modifying code
; Enter this code with <Rt> containing a new 32-bit instruction,
; to be held in Cacheable space at a location pointed to by Rn. Use STRH in the first
; line instead of STR for a 16-bit instruction.
STR <Rt>, [Rn]
DCCMVAU Rn              ; Clean data cache by MVA to point of unification (PoU)
DSB                     ; Ensure visibility of the data stored
ICIMVAU Rn              ; Invalidate instruction cache by VA to PoU
BPIMVA Rn               ; Invalidate branch predictor by MVA to PoU
DSB
```

Note

- The DCCMVAU operation is not required if the area of memory is either Non-cacheable or Write-Through Cacheable.
- If the contents of physical memory differ between the mappings, changing the mapping of VAs to PAs can cause the instructions to be concurrently modified by one PE and executed by another PE. If the modifications affect instructions other than those listed as being acceptable for modification, synchronization must be used to avoid UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior.

---

3. In a multiprocessor system, the ICIMVAU and BPIMVA are broadcast to all PEs within the Inner Shareable domain of the PE running this sequence. However, once the modified instructions are observable, each PE that is executing the modified instructions must issue the following instruction to ensure execution of the modified instructions:

```
ISB                             ; Synchronize fetched instruction stream
```

For more information about the required synchronization operation, see *Synchronization and coherency issues between data and instruction accesses* on page E2-4001.

For information about memory accesses caused by instruction fetches, see *Ordering constraints* on page E2-3984.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

E2.3 Definition of the Armv8 memory model

This section describes observation and ordering in the Armv8 memory model. It contains the following subsections:

- *Locations*.
- *Ordering and observability* on page E2-3981.
- *Ordering constraints* on page E2-3984.
- *Completion and endpoint ordering* on page E2-3987.
- *Ordering of instruction fetches* on page E2-3989.
- *Restrictions on the effects of speculation* on page E2-3989.
- *Memory barriers* on page E2-3991.

For more information on endpoint ordering of memory accesses, see *Reordering* on page E2-4017.

In the Armv8 memory model, the Shareability memory attribute indicates the degree to which hardware must ensure memory coherency between a set of observers, see *Memory types and attributes* on page E2-4010.

The Armv8 architecture defines additional memory attributes and associated behaviors, which are defined in the system level section of this manual. See:

- Chapter G4 *The AArch32 System Level Memory Model*.
- Chapter G5 *The AArch32 Virtual Memory System Architecture*.

See also *Mismatched memory attributes* on page E2-4020.

E2.3.1 Locations

The Armv8 memory model provides a set of definitions that are used to construct conditions on the permitted sequences of accesses to memory. The Armv8 memory model defines:

- The ordering of observation of memory accesses between different observers.
- The ordering of arrival of memory accesses arriving at an endpoint.
- The mechanisms to control the ordering of observation of memory accesses and the arrival of memory accesses at an endpoint.

Locations, Memory effects, and Observers

The Armv8 memory model provides the following definition of a *Location* in memory:

Location

A *Location* refers to a single byte in memory.

As part of its execution an instruction might generate a Memory effect. Observers in the system might observe the Memory effects of that instruction on a Location. The Armv8 memory model provides the following definitions of a Memory effect and an Observer:

Memory effect

The *Memory effects* of an instruction are the read, write, or barrier effects of that instruction. For an instruction that accesses memory:

- A read effect is generated for each Location that is read by the instruction.
- A write effect is generated for each Location that is written by the instruction.

An instruction can generate both read and write effects.

The memory effects of an instruction I_1 are said to appear in program order before the Memory effects of instruction I_2 if and only if I_1 occurs before I_2 in program order.

For the purposes of describing the Armv8 memory model, all read and write effects access only Normal memory locations in a Common Shareability Domain. Where this section refers to a read, write, or memory barrier without any qualification, then it is referring to the corresponding Memory effect.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Observer

An *Observer* refers to either a processing element, or some other memory accessing agent that can generate reads from or writes to memory.

Common Shareability Domain

A *Common Shareability Domain* for a program is the smallest Shareability domain that contains all of the active Observers of the Memory effects generated by a program.

E2.3.2 Ordering and observability

The Armv8 memory model permits reordering of memory accesses. This section defines the permitted reordering of memory accesses using the following:

- Register value dependencies to establish order between instructions on a PE.
- Ordering constraints to establish order between accesses to a Location.

Register value dependencies

The Armv8 memory model defines the following dependencies between instructions:

Register dependency

A *Register dependency* from a first data value V_1 to a second data value V_2 exists within a PE if and only if either:

- The register, excluding the AArch64 zero register (XZR or WZR), that is used to hold V_1 is used in the calculation of V_2.
- There is a Register dependency from V_1 to a third data value V_3 and there is a register dependency from V_3 to V_2.

Register data dependency

A *Register data dependency* from a first data value V_1 to a second data value V_2 exists within a PE if and only if either:

- The register, excluding the AArch64 zero register (XZR or WZR) and the AArch32 PC, that is used to hold V_1 and is used in the calculation of V_2, and the calculation between V_1 and the V_2 does not consist of either:
  - A conditional branch whose condition is determined by V_1.
  - A conditional selection, move, or computation whose condition is determined by V_1, where the input data values for the selection, move, or computation do not have a data dependency on V_1.
- There is a Register data dependency from V_1 to a third data value V_3, and there is a Register data dependency from V_3 to V_2.

Address dependency

An *Address dependency* from a read R_1 to a subsequent read R_2 exists if and only if there is a Register data dependency from the data value that is returned by R_1 to the address used by R_2.

An *Address dependency* from a read R_1 to a subsequent write W_2 exists if and only if there is a Register dependency from the data value that is returned by R_1 to the address used by W_2.

Data dependency

A *Data dependency* from a read R_1 to a subsequent write W_2 exists if and only if there is a Register dependency from the data value returned by R_1 to the data value written by W_2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Control dependency

A *Control dependency* from a read R_1 to a subsequent instruction I_2 exists if and only if either:

- There is a Register dependency from the data value returned by R_1 to the data value used in the evaluation of a conditional branch, and I_2 is only executed as a result of one of the possible outcomes of that conditional branch.
- There is a Register dependency from the data value returned by R_1 to the data value used in the determination of a synchronous exception on an instruction I_3, and I_2 appears in program order after I_3.

Ordering and observability at a Location

Memory effects on a Location are related by the following relations:

Reads-from

A *Reads-from* relation that couples reads and writes to the same Location such that each read is paired with a single write in the program. If a read R_2 of a Location Reads-from a write W_1 to the same Location if and only if R_2 takes its data from W_1.

Note

The Reads-from relation represents a read being satisfied by a write and then returning the written data.

Local read successor

A read R_2 of a Location is the *Local read successor* of a write W_1 from the same Observer to the same Location if and only if W_1 appears in program order before R_2 and there is not a write W_3 from the same Observer to the same Location appearing in program order between W_1 and R_2.

Local write successor

A write W_2 of a Location is a *Local write successor* of a write W_1 from the same Observer to the same Location if and only if W_1 appears in program order before W_2.

Coherence order

A *Coherence order* relation for each Location in the program that provides a total order on all writes from all coherent Observers to that Location, starting with a notional write of the initial value.

Note

The Coherence order of a Location represents the order in which writes to the Location arrive at memory.

Coherence-after

A write W_2 to a Location is *Coherence-after* another write W_1 to the same Location if and only if W_2 is sequenced after W_1 in the Coherence order of the Location.

A write W_2 to a Location is Coherence-after a read R_1 of the same location if and only if R_1 Reads-from a write W_3 to the same Location and W_2 is Coherence-after W_3.

Overlapping accesses

Two Memory effect overlap if and only if they access the same Location. Two instructions overlap if and only if one or more of their generated Memory effects overlap.

Observed-by

A read or a write RW_1 from an Observer is *Observed-by* a write W_2 from a different Observer if and only if W_2 is coherence-after RW_1.

A write W_1 from an Observer is Observed-by a read R_2 from a different Observer if and only if R_2 Reads-from W_1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

The Observed-by relation only relates accesses generated by different Observers.

`DMB FULL`

A `DMB FULL` is a `DMB` with neither the `LD` or the `ST` qualifier.

Where this section refers to `DMB` without any qualification, then it is referring to all types of `DMB`. Unless a specific shareability domain is defined, a `DMB` applies to the Common Shareability Domain.

All properties that apply to `DMB` also apply to the corresponding `DSB`.

Ordering relations

In addition to the ordering relations that describe order at a location, the Armv8 memory model also provides ordering relations to describe the ordering of instructions. These are as follows:

Dependency-ordered-before

A dependency creates externally-visible order between a read and another Memory effect generated by the same Observer. A read R_1 is *Dependency-ordered-before* a read or write RW_2 from the same Observer if and only if R_1 appears in program order before RW_2 and any of the following cases apply:

- There is an Address dependency or a Data dependency from R_1 to RW_2.
- RW_2 is a write W_2 and there is a Control dependency from R_1 to W_2.
- RW_2 is a read R_2 generated by an instruction appearing in program order after an instruction I_3 that generates a Context synchronization event, and there is a Control dependency from R_1 to I_3.
- RW_2 is a write W_2 appearing in program order after a read or a write RW_3 and there is an Address dependency from R_1 to RW_3.
- RW_2 is a read R_2 that is a Local read successor of W_3 and there is an Address dependency or a Data dependency from R_1 to W_3.

Atomic-ordered-before

Load-Exclusive and Store-Exclusive instructions provide some ordering guarantees, even in the absence of dependencies. A read or a write RW_1 is *Atomic-ordered-before* a read or a write RW_2 from the same Observer if and only if RW_1 appears in program order before RW_2 and either of the following cases apply:

- RW_1 is a read R_1 and RW_2 is a write W_2 such that R_1 and W_2 are generated by an atomic instruction or a successful Load-Exclusive/Store-Exclusive instruction pair to the same Location.
- RW_1 is a write W_1 generated by an atomic instruction or a successful Store-Exclusive instruction and RW_2 is a read R_2 generated by an instruction with Acquire semantics such that R_2 is a Local read successor of W_1.

For more information, see *Synchronization and semaphores* on page E2-4023.

Barrier-ordered-before

Barrier instructions order prior Memory effects before subsequent Memory effects generated by the same Observer. A read or a write RW1 is *Barrier-ordered-before* a read or a write RW_2 from the same Observer if and only if RW1 appears in program order before RW_2 and any of the following cases apply:

- RW_1 appears in program order before a `DMB FULL` that appears in program order before RW_2.
- At least one of RW_1 and RW_2 is generated by an atomic instruction with both Acquire and Release semantics.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- RW_1 is a write W_1 generated by an instruction with Release semantics and RW_2 is a read R_2 generated by an instruction with Acquire semantics.
- RW_1 is a read R_1 and either:
  - R_1 appears in program order before a `DMB LD` that appears in program order before RW_2.
  - R_1 is generated by an instruction with Acquire semantics.
- RW_2 is a write W_2 and either:
  - RW_1 is a write W_1 appearing in program order before a `DMB ST` that appears in program order before W_2.
  - W_2 is generated by an instruction with Release semantics.

Locally-ordered-before

Dependencies, Local write successor, Load/Store-Exclusive, atomic and barrier instructions can be composed within an Observer to create externally-visible order. A read or write RW_1 is *Locally-ordered-before* a read or write RW_2 from the same Observer if and only if any of the following apply:

- RW_1 is a write W_1 and RW_2 is a write W_2 such that RW_2 is, or is generated by the same instruction as, a Local write successor of RW_1.
- RW_1 is Dependency-ordered-before RW_2.
- RW_1 is Atomic-ordered-before RW_2.
- RW_1 is Barrier-ordered-before RW_2.
- RW_1 is Locally-ordered-before a read or a write that is Locally-ordered-before RW_2.

Ordered-before

An arbitrary pair of Memory effects is ordered if it can be linked by a chain of ordered accesses consistent with external observation. A read or a write RW_1 is *Ordered-before* a read or a write RW_2 if and only if any of the following cases apply:

- RW_1 is Observed-by a read or write RW_3 that is equal to or generated by the same instruction as RW_2.
- RW_1 is Locally-ordered-before RW_2.
- RW_1 is Ordered-before a read or write that is Ordered-before RW_2.

E2.3.3 Ordering constraints

The Armv8 memory model is described as being Other-multi-copy atomic. The definition of Other-multi-copy atomic is as follows:

Other-multi-copy atomic

In an *Other-multi-copy atomic* system, it is required that a write from an Observer, if observed by a different Observer, is then observed by all other Observers that access the Location coherently. It is, however, permitted for an Observer to observe its own writes prior to making them visible to other observers in the system.

The Other-multi-copy atomic property of the Armv8 memory model is enforced by placing constraints on the possible executions of a program. Those executions that meet the constraints given by the ordering model are said to be *architecturally well-formed*. An implementation that is executing a program is only permitted to exhibit behavior consistent with an architecturally well-formed execution.

Architecturally well-formed

An *architecturally well-formed* execution must satisfy both the Internal visibility requirement and one of the following three alternative representations of the external ordering requirements:

- External visibility requirement.
- External completion requirement.
- External global completion requirement.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Internal visibility requirement

For a read or a write RW_1 that appears in program order before a read or a write RW_2 to the same Location, the internal visibility requirement requires that exactly one of the following statements is true:

- RW_2 is a write W_2 that is Coherence-after RW_1.
- RW_1 is a write W_1 and RW_2 is a read R_2 such that either:
  - — R_2 Reads-from W_1.
  - — R_2 Reads-from another write that is Coherence-after W_1.
- RW_1 and RW_2 are both reads R_1 and R_2 such that R_1 Reads-from a write W_3 and either:
  - — R_2 Reads-from W_3.
  - — R_2 Reads-from another write that is Coherence-after W_3.

Note

If a Memory effect M_1 from an Observer appears in program order before a Memory effect M_2 from the same Observer, then M_1 will be seen to occur before M_2 by that Observer.

External visibility requirement

For a read or a write RW_1 from an Observer that is Ordered-before a read or a write RW_2 from a different Observer, the external visibility constraint requires that RW_2 is not Observed-by RW_1. This means that an Architecturally well-formed execution must not exhibit a cycle in the Ordered-before relation.

Note

If a Memory effect M_1 from an Observer is Ordered-before another Memory effect M_2, from a different Observer, then M_1 will be seen to occur before M_2 by all Observers in the system.

Alternative equivalent formulations of external visibility requirement

Single-copy-atomic-ordered-before

A read R_1 is *Single-copy-atomic-ordered-before* another read R_2 if and only if all of the following statements are true:

- R_1 and R_2 are reads from the same instruction.
- R_1 does not Local read successor a write.
- R_2 Local read successor a write.

Total time order

The *Total time order* (TTO) is a total order that corresponds to the order in which memory effects complete within the system. The following effects constitute a single entry in the TTO:

- Writes from the same instruction.
- Reads from the same instruction which read from external writes.
- Writes from the same instruction which read from the same internal write.

All other reads constitute distinct entries in the TTO.

Completes-before

A read or a write RW_1 *Completes-before* a read or a write RW_2 if and only if RW_1 appears in the TTO before RW_2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Deriving Reads-from the TTO

The TTO can be used to resolve the Reads-from relation for every memory access in the system as follows:

- For a read RW_1 of a memory location by an observer, then:
  - If there is a write W_2 to the same Location from the same Observer and all of the following are true:
    - W_2 appears in program order before R_1.
    - R_1 Completes-before W_2.
    - There are no writes to the Location appearing in program order between W_2 and R_1 then R_1 reads from W_2.
- Otherwise, if there is a write W_2 to the same Location and all of the following are true:
  - W_2 Completes-before R_1.
  - There are no writes to the Location appearing in program order or in the TTO between W_2 and R_1 then R_1 Reads-from W_2.
- If no such write exists, then R_1 Reads-from the initial value of the memory location.

Deriving Coherence order from the TTO

The TTO can be used to resolve the Coherence order relation for every memory access in the system as follows:

The Coherence order of writes to a memory location is the order in which those writes appear in the TTO. The final value of each memory location is therefore determined by the final write to each Location in the TTO. If no such write exists for a given location, the final value is the initial value of that Location.

External completion requirement

A read or a write RW_1 Globally-completes-before a read or a write RW_2 if and only if any of the following statements are true:

- RW_1 is Locally-ordered-before RW_2.
- RW_1 is a read R_1 and either:
  - RW_2 is a read R_2 and R_1 is Single-copy-atomic-ordered-before R_2.
  - R_1 is Single-copy-atomic-ordered-before a read R_3, such that:
    - R_3 is Locally-ordered-before RW_2.
    - R_3 Local read successor a write W4 that is not Locally-ordered-before RW_2.

Total coherence order

The total coherence order (TCO) is a total order that corresponds to the order in which memory effects globally-complete within the system. The following effects constitute a single entry in the TCO:

- Writes from the same instruction.
- Reads from the same instruction which read from external writes.
- Writes from the same instruction which read from the same internal write.

All other reads constitute distinct entries in the TCO.

Globally-completes-before

A read or a write RW_1 *globally-completes-before* a read or a write RW_2 if and only if RW_1 appears in the TCO before RW_2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Deriving Reads-from and Coherence order from the TCO

The TCO can be used to resolve the Reads-from and Coherence order relations for every memory access in the system as follows:

- A read R_1 of a memory location by an Observer Reads-from its closest preceding write in the TCO to the same Location. If no such write exists, then R_1 Reads-from the initial value of the memory location.
- The Coherence order of writes to a memory location is the order in which those writes appear in the TCO. The final value of each memory location is therefore determined by the final write to each Location in the TCO. If no such write exists for a given Location, the final value is the initial value of that Location.

External global completion requirement

For a read or a write RW_1 that is Locally-ordered-before a read or a write RW_2, the external global completion requirement requires that RW_1 Globally-completes-before RW_2 if and only if any of the following statements are true:

- RW_1 is Locally-ordered-before RW_2 and either:
  - — RW_1 is a write.
  - — RW_1 is a read R_1 and either:
    - — R_1 does not Local read successor a write.
    - — R_1 Local read successor a write that is Locally-ordered-before RW_2.
- RW_1 is a read R_1 and either:
  - — RW_2 is a read R_2 and R_1 is Single-copy-atomic-ordered-before R_2.
  - — R_1 is Single-copy-atomic-ordered-before a read R_3, such that:
    - — R_3 is Locally-ordered-before RW_2.
    - — R_3 Local read successor a write W_4 that is not Locally-ordered-before RW_2.

E2.3.4 Completion and endpoint ordering

Interaction between Observers in a system is not restricted to communication via shared variables in coherent memory. For example, an Observer could configure an interrupt controller to raise an interrupt on another Observer as a form of message passing. These interactions typically involve an additional agent, which defines the instruction sequence that is required to establish communication links between different Observers. When these forms of interaction are used in conjunction with shared variables, a `DSB` instruction can be used to enforce ordering between them.

For all memory, the completion rules are defined as:

- A read R_1 to a Location is complete for a shareability domain when all of the following are true:
  - — Any write to the same Location by an Observer within the shareability domain will be Coherence-after R_1.
  - — Any translation table walks associated with R_1 are complete for that shareability domain.
- A write W_1 to a Location is complete for a shareability domain when all of the following are true:
  - — Any write to the same Location by an Observer within the shareability domain will be Coherence-after W_1.
  - — Any read to the same Location by an Observer within the shareability domain will either Reads-from W_1 or Reads-from a write that is Coherence-after W_1.
  - — Any translation table walks associated with the write are complete for that shareability domain.
- A translation table walk is complete for a shareability domain when the memory accesses, including the updates to translation table entries, associated with the translation table walk are complete for that shareability domain, and the TLB is updated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- A cache or branch predictor maintenance instruction is complete for a shareability domain when the memory effects of the instruction are complete for that shareability domain, and any translation table walks that arise from the instruction are complete for that shareability domain.
- A TLB invalidate instruction is complete when all memory accesses using the TLB entries that have been invalidated are complete.

The completion of any cache, branch predictor, or TLB maintenance instruction includes its completion on all PEs that are affected by both the instruction and the `DSB` operation that is required to guarantee visibility of the maintenance instruction.

Note

These completion rules mean that, for example, a cache maintenance instruction that operates by VA to the PoC completes only after memory at the PoC has been updated.

Additionally, for Device-nGnRnE memory, a read or write of a Location in a Memory-mapped peripheral that exhibits side-effects is complete only when the read or write both:

- Can begin to affect the state of the Memory-mapped peripheral.
- Can trigger all associated side-effects, whether they affect other peripheral devices, PEs, or memory.

Note

This requirement for Device-nGnRnE memory is consistent with the memory access having reached the peripheral endpoint.

Peripherals

This section defines a Memory-mapped peripheral and the total order of reads and write to a peripheral which is defined as the Peripheral coherence order:

Memory-mapped peripheral

A *Memory-mapped peripheral* occupies a memory region of IMPLEMENTATION DEFINED size and can be accessed using load and store instructions. Memory effects to a Memory-mapped peripheral can have side-effects, such as causing the peripheral to perform an action. Values that are read from addresses within a Memory-mapped peripheral might not correspond to the last data value written to those addresses. As such, Memory effects to a Memory-mapped peripheral might not appear in the Reads-from or Coherence order relations.

Peripheral coherence order

The *Peripheral coherence order* of a Memory-mapped peripheral is a total order on all reads and writes to that peripheral.

Note

The Peripheral coherence order for a Memory-mapped peripheral signifies the order in which accesses arrive at the endpoint.

For a read or a write RW_1 and a read or a write RW_2 to the same peripheral, then RW_1 will appear in the Peripheral coherence order for the peripheral before RW_2 if either of the following cases apply:

- RW_1 and RW_2 are accesses using Non-cacheable or Device attributes and RW_1 is Ordered-before RW_2.
- RW_1 and RW_2 are accesses using Device-nGnRE or Device-nGnRnE attributes and RW_1 appears in program order before RW_2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Out-of-band-ordered-before

A read or a write RW_1 is *Out-of-band-ordered-before* a read or a write RW_2 if and only if either of the following cases apply:

- RW_1 appears in program order before a `DSB` instruction that begins an IMPLEMENTATION DEFINED instruction sequence indirectly leading to the generation of RW_2.
- RW_1 is Ordered-before a read or a write RW_3 and RW_3 is Out-of-band-ordered-before RW_2.

If a Memory effect M_1 is Out-of-band-ordered-before a read or a write M_2, then M_1 is seen to occur before M_2 by all Observers.

E2.3.5 Ordering of instruction fetches

For two memory locations A and B, if A has been written to and been made coherent with the instruction fetches of the shareability domain, before an update to B by an observer in the same shareability domain, then the instruction stream of each observer in the shareability domain will not see the updated value of B without also seeing the updated value of A.

A write has been made coherent with an instruction fetch of a shareability domain when:

CTR.{DIC, IDC} == {0, 0}

The location written to has been cleaned to the *Point of unification (PoU)* from the data cache, and that clean is complete for the shareability domain. Subsequently the location has been invalidated to the *Point of unification (PoU)* from the instruction cache, and that invalidation is complete for the shareability domain.

CTR.{DIC, IDC} == {1, 0}

The location written to has been cleaned to the *Point of unification (PoU)* from the data cache, and that clean is complete for the shareability domain.

CTR.{DIC, IDC} == {0, 1}

The write is complete for the shareability domain. Subsequently the location has been invalidated to the *Point of unification (PoU)* from the instruction cache, and that invalidation is complete for the shareability domain.

CTR.{DIC, IDC} == {1, 1}

The write is complete for the shareability domain.

Note

Microarchitecturally, this means that these situations cannot both be true in an implementation:

- After delays in fetching from memory, the instruction queue can have entries written into it out of order.
- For an implementation:
  - When CTR.DIC == 0, if there is an outstanding entry in the instruction queue, then later entries in the instruction queue are not impacted by the `ICIMVAU` instructions of a different core.
  - When CTR.DIC == 1, if there is a write to the location that is held in the queue when there is an outstanding entry in the instruction queue for an older entry, then the instruction queue does not have entries invalidated from it.

E2.3.6 Restrictions on the effects of speculation

This section covers restrictions on speculation effects, including:

- *Restrictions on the effects of speculation* on page E2-3990.
- *Speculative Store Bypass Safe (SSBS)* on page E2-3990.
- *Further restrictions on the effects of speculation from Armv8.5* on page E2-3990.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Restrictions on the effects of speculation

The Arm architecture places certain restrictions on the effects of speculation. These are:

- Each load from a location using a particular VA after an exception return that is a Context synchronization event will not speculatively read an entry from earlier in the coherence order for the location being loaded from than the entry generated by the latest store to that location using the same VA before the exception exit.
- Each load from a location using a particular VA after an exception entry that is a Context synchronization event will not speculatively read an entry from earlier in the coherence order for the location being loaded from than the entry generated by the latest store to that location using the same VA before the exception entry.
- Any load from a location using a particular VA before an exception entry that is a Context synchronization event will not speculatively read data from a store to the same location using the same VA after the exception entry.
- Any load from a location using a particular VA before an exception return that is a Context synchronization event will not speculatively read data from a store to the same location using the same VA after the exception exit.
- When data is loaded under speculation with a translation fault, it cannot be used to form an address or generate condition codes to be used by instructions newer than the load in the speculative sequence.
- When data is loaded under speculation from a location without a translation for the translation regime being speculated in, the data cannot be used to form an address or generate condition codes to be used by instructions newer than the load in the speculative sequence.
- Changes to System registers must not occur speculatively in a way that can affect a speculative memory access that can cause a change to the micro-architectural state.
- Changes to Special-purpose registers can occur speculatively.

Speculative Store Bypass Safe (SSBS)

When ARMv8.0-SSBS is implemented, CPSR.SSBS is a control that can be set by software to indicate whether hardware is permitted to load or store speculatively, in a manner that could be exploited to produce a cache timing side channel using an address derived from a register value that has been loaded from memory using a load instruction that speculatively read an entry for the location being loaded from, where the entry that is speculatively read is from earlier in the coherence order than the entry generated by the latest store to that location using the same virtual address as the load instruction.

When the value of CPSR.SSBS is 0, hardware is not permitted to load or store speculatively in this way.

When the value of CPSR.SSBS is 1, hardware is permitted to load or store speculatively in this way.

Note

- If speculation is permitted, then cache timing side channels can lead to addresses being derived using reads of address values that have been speculatively loaded from memory to a register.
- Software written for architectures from Armv8.0 to Armv8.4 will set SPSR.SSBS to 0. This means that CPSR.SSBS will not set, so hardware will not be permitted to use speculative loads with outstanding memory disambiguation issues for any subsequent speculative memory accesses if there is any possibility of those subsequent memory accesses creating a cache timing side channel.

Further restrictions on the effects of speculation from Armv8.5

From Armv8.5, there are some further restrictions on the effects of speculation in addition to those in Armv8.0:

- Data loaded under speculation with a permission or domain fault cannot be used to form an address or to generate condition codes to be used by instructions newer than the load in the speculative sequence.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Any System register read under speculation to a register that is not architecturally accessible from the current Exception level cannot be used to form an address or to generate condition codes to be used by instructions newer than the load in the speculative sequence.

  Note

  As the effects of speculation are not architecturally visible, this restriction level requires that the effect of any speculation cannot give rise to side channels that will leak the values of memory locations, System registers, or Special-purpose registers to a level of privilege that would otherwise not be able to determine those values.

- For all execution prediction resources that predict address or register values, speculative execution at one hardware defined context should be separated in a hard-to-determine manner from the predictions trained in a different hardware defined context. In the case of this definition, the hardware defined context is determined by:
  - The Exception level.
  - The Security state.
  - When executing at EL1 and when EL2 is enabled in the current Security state, the VMID.
  - When executing at EL0 and using the EL1&0 translation regime, the ASID and, when EL2 is enabled in the current Security state, the VMID.
  - When executing at EL0 and using the EL2&0 translation regime, the ASID.

  Note

  - The definition of "hard-to-determine manner" is left open to implementations. Examples could include the complete separation of prediction resources, or the isolation of the predictions using a cryptographic or pseudo-random mechanism to separate each context.
  - The architecture does not require that prediction resources that simply predict the direction of a branch are separated in this way.

- Changes to System registers must not occur speculatively in a way that can affect a speculative memory access that can cause a change to the micro-architectural state.

- Changes to Special-purpose registers can occur speculatively.

E2.3.7 Memory barriers

The Arm architecture is a weakly ordered memory architecture that supports out of order completion. *Memory barrier* is the general term applied to an instruction, or sequence of instructions, that forces synchronization events by a PE with respect to retiring Load/Store instructions. The memory barriers defined by the Armv8 architecture provide a range of functionality, including:

- Ordering of Load/Store instructions.
- Completion of Load/Store instructions.
- Context synchronization.

The following subsections describe the Armv8 memory barrier instructions:

- *Instruction Synchronization Barrier (ISB)* on page E2-3992.
- *Data Memory Barrier (DMB)* on page E2-3992.
- *Data Synchronization Barrier (DSB)* on page E2-3993.
- *Speculation Barrier (SB)* on page E2-3993.
- *Consumption of Speculative Data Barrier (CSDB)* on page E2-3994.
- *Speculative Store Bypass Barrier (SSBB)* on page E2-3994.
- *Physical Speculative Store Bypass Barrier (PSSBB)* on page E2-3994.
- *Trace Synchronization Barrier (TSB CSYNC)* on page E2-3995.
- *Shareability and access limitations on the data barrier operations* on page E2-3995.
- *Load-Acquire, Store-Release* on page E2-3996.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

Depending on the required synchronization, a program might use memory barriers on their own, or it might use them in conjunction with cache maintenance and memory management instructions that in general are only available when software execution is at EL1 or higher.

The DMB and DSB memory barriers affect reads and writes to the memory system generated by Load/Store instructions and data or unified cache maintenance instructions being executed by the PE. Instruction fetches or accesses caused by a hardware translation table access are not explicit accesses.

AArch32 state also supports the legacy barrier instructions CP15DMB, CP15DSB, and CP15ISB. These instructions are executed as MCRs using the appropriate encoding, and are accessible from EL0. However, for performance reasons Arm deprecates any use of these operations, and strongly recommends that software uses the DMB, DSB, and ISB instructions described in this section instead. Optionally, an implementation can support a CP15BEN control that supervisory software can use to disable use of these instructions, meaning the corresponding MCR encodings are UNDEFINED. When the CP15BEN control is supported, setting one of the following CP15BEN fields to 0 makes execution of CP15DMB, CP15DSB, and CP15ISB UNDEFINED:

- SCTLR_EL1.CP15BEN, for execution of these instructions at EL0 using AArch32 when EL1 is using AArch64.
- SCTLR.CP15BEN, for execution of these instructions at EL0 or EL1 when EL1 is using AArch32.
- HSCTLR.CP15BEN, for execution of these instructions at EL2 when EL2 is using AArch32.

Instruction Synchronization Barrier (ISB)

An ISB instruction ensures that all instructions that come after the ISB instruction in program order are fetched from the cache or memory after the ISB instruction has completed. Using an ISB ensures that the effects of context-changing operations executed before the ISB are visible to the instructions fetched after the ISB instruction. Examples of context-changing operations that require the insertion of an ISB instruction to ensure the effects of the operation are visible to instructions fetched after the ISB instruction are:

- Completed cache and TLB maintenance instructions.
- Changes to System registers.

Any context-changing operations appearing in program order after the ISB instruction only take effect after the ISB has been executed.

The pseudocode function for the operation of an ISB is `InstructionSynchronizationBarrier()`.

See also *Memory barriers* on page G4-5919.

Data Memory Barrier (DMB)

The DMB instruction is a memory barrier instruction that ensures the relative order of memory accesses before the barrier with memory accesses after the barrier. The DMB instruction does not ensure the completion of any of the memory accesses for which it ensures relative order.

The full definition of the DMB instruction is covered formally in the *Definition of the Armv8 memory model* on page E2-3980 and this introduction to the DMB instruction is not intended to contradict that section.

The basic principle of a DMB instruction is to introduce order between memory accesses that are specified to be affected by the DMB options supplied as arguments to the DMB instruction. The DMB instruction ensures that all affected memory accesses by the PE executing the DMB instruction that appear in program order before the DMB instruction and those which originate from a different PE, to the extent required by the DMB options, which have been Observed-by the PE before the DMB instruction is executed, are Observed-by each PE, to the extent required by the DMB options, before any affected memory accesses that appear in program order after the DMB instruction are Observed-by that PE.

The use of a DMB instruction creates order between the Memory effects of instructions as described in the definition of Barrier-ordered-before.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The pseudocode function for the operation of a DMB instruction is DataMemoryBarrier().

Data Synchronization Barrier (DSB)

A DSB instruction is a memory barrier that ensures that memory accesses that occur before the DSB instruction have completed before the completion of the DSB instruction. In doing this, it acts as a stronger barrier than a DMB and all ordering that is created by a DMB with specific options is also generated by a DSB with the same options.

Execution of a DSB at EL2 ensures that any memory accesses caused by speculative translation table walks from the Non-secure PL1&0 translation regime have been observed.

For more information, see *Use of out-of-context translation regimes* on page G5-5928.

A DSB executed by a PE, PEe, completes when all of the following apply:

- All explicit memory accesses of the required access types appearing in program order before the DSB are complete for the set of observers in the required shareability domain.
- If the required access types of the DSB is reads and writes, then all cache maintenance instructions, all TLB maintenance instructions, and all PSB CYNC instructions issued by PEe before the DSB are complete for the required shareability domain.

In addition, no instruction that appears in program order after the DSB instruction can alter any state of the system or perform any part of its functionality until the DSB completes, other than:

- Being fetched from memory and decoded.
- Reading the general-purpose, SIMD and floating-point, Special-purpose, or System registers that are directly or indirectly read without causing side-effects.

The pseudocode function for the operation of a DSB is DataSynchronizationBarrier().

See also *Memory barrier instructions* on page G4-5916 and *Memory barriers* on page G4-5919.

Speculation Barrier (SB)

An SB is a memory barrier that prevents speculative execution of instructions until after the barrier has completed when those instructions could be observed through side-channels.

Until the barrier completes, the speculative execution of any instruction appearing later in the program order than the barrier:

- Cannot be performed to the extent that such speculation can be observed through side-channels as a result of control flow speculation or data value speculation.
- Can be performed when predicting that a instruction that could generate an exception does not generate an exception.

Speculative execution of an SB instruction:

- Cannot be as a result of control flow speculation.
- Cannot be as a result of data value speculation.
- Can be as a result of predicting that an instruction that could generate an exception does not generate an exception.

An SB instruction can complete when:

- It is known that it is not speculative.
- All the predicted data values generated by instructions appearing in program order before the SB instruction have their predicted values confirmed.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

The SB instruction has no effect on the use of prediction resources to predict the instruction stream that is being fetched, so long as the prediction of the instruction stream is not informed by data taken from the register outputs of the speculative execution of instructions appearing in program order after an uncompleted SB instruction.

Consumption of Speculative Data Barrier (CSDB)

The CSDB instruction is a memory barrier instruction that controls speculative execution and data value prediction. This includes:

- Data value predictions of any instructions.
- PSTATE.{N,Z,C,V} predictions of any instructions other than conditional branch instructions appearing in program order before the CSDB that have not been architecturally resolved.
- Predictions of SVE prediction state for any SVE instructions.

For purposes of the definition of CSDB, PSTATE.{N,Z,C,V} is not considered a data value. This definition permits:

- Control flow speculation before and after the CSDB.
- Speculative execution of conditional data processing instructions after the CSDB, unless they use the results of data value or PSTATE.{N,Z,C,V} predictions of instructions appearing in program order before the CSDB that have not been architecturally resolved.

Speculative Store Bypass Barrier (SSBB)

The SSBB instruction is a memory barrier that prevents speculative loads from bypassing earlier stores to the same virtual address under certain conditions.

The semantics of the Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the SSBB, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store uses the same virtual address as the load.
  - The store appears in program order before the SSBB instruction.
- When a load to a location appears in program order before the SSBB, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store uses the same virtual address as the load.
  - The store appears in program order after the SSBB instruction.

Physical Speculative Store Bypass Barrier (PSSBB)

The PSSBB instruction is a memory barrier that prevents speculative loads from bypassing earlier stores to the same physical address under certain conditions.

The semantics of the Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the PSSBB, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order before the PSSBB instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- When a load to a location appears in program order before the PSSBB, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order after the PSSBB instruction.

Note

The effect of this barrier applies to accesses to the same location even if they are accessed with different virtual addresses and from different Exception levels.

Trace Synchronization Barrier (TSB CSYNC)

The TSB CSYNC is a memory barrier instruction that preserves the relative order of memory accesses to System registers due to trace operations and other memory accesses to the same registers.

A trace operation is an operation of the PE Trace Unit generating trace for an instruction when ARMv8.4-Trace is implemented and enabled.

A TSB CSYNC is not required to execute in program order with respect to other instructions. This includes being reordered with respect to other trace instructions. One or more context synchronization events are required to ensure that TSB CSYNC is executed in the necessary order.

If trace is generated between a context synchronization event and a TSB CSYNC operation, these trace operations may be reordered with respect to the TSB CSYNC operation, and therefore may not be synchronized.

The following situations are synchronized using a TSB CSYNC:

- A direct write B to a System register is ordered after an indirect read or indirect write of the same register by a trace operation of a traced instruction A, if all of the following are true:
  - A is executed in program order before a context synchronization event C.
  - C is in program order before a TSB CSYNC operation T.
  - B is executed in program order after T.
- A direct read B of a System register is ordered after an indirect write to the same register by a trace operation of a traced instruction A if all the following are true:
  - A is executed in program order before a context synchronization event C1.
  - C1 is in program order before TSB CSYNC operation T.
  - T is executed in program order before a second context synchronization event C2.
  - B is executed in program order after C2.

A TSB CSYNC is not needed when a direct write B to a System register is ordered before an indirect read or indirect write of the same register by a trace operation of a traced instruction A, if all the following are true:

- A is executed in program order after a context synchronization event C.
- B is executed in program order before C.

The pseudocode function for the operation of a TSB CSYNC is TraceSynchronizationBarrier().

Shareability and access limitations on the data barrier operations

The DMB and DSB instructions can each take an optional limitation argument that specifies:

- The shareability domain over which the instruction must operate. This is one of:
  - Full system.
  - Outer Shareable.
  - Inner Shareable.
  - Non-shareable.
- The accesses for which the instruction operates. This is one of:
  - Read and write accesses, both before and after the barrier instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— Write accesses only, before and after the barrier instruction.
— Read accesses before the barrier instruction, and read and write accesses after the barrier instruction.

Note

This form of a DMB or DSB instruction can be described as a Load-Load/Store barrier.

For more information on whether an access is before or after a barrier instruction, see *Data Memory Barrier (DMB)* on page E2-3992 or *Data Synchronization Barrier (DSB)* on page E2-3993.

Table E2-1 shows how these options are encoded in the <option> field of the instruction.

Table E2-1 Encoding of the DMB and DSB <option> parameter

Accesses		Shareability domain			
Before the barrier	**After the barrier**	**Full system**	**Outer Shareable**	**Inner Shareable**	**Non-shareable**
Reads and writes	Reads and writes	SY	OSH	ISH	NSH
Writes	Writes	ST	OSHST	ISHST	NSHST
Reads	Reads and writes	LD	OSHLD	ISHLD	NSHLD

If no <option> is specified then the instruction operates for read and write accesses, over the full system, meaning the operation is the same as for the SY option. See the instruction descriptions for more information:

- *DMB* on page F5-4354.
- *DSB* on page F5-4357.

Note

ISB also supports an optional limitation argument that can only contain one value that corresponds to full system operation, see *ISB* on page F5-4377.

Load-Acquire, Store-Release

Armv8 provides a set of instructions with Acquire semantics for loads, and Release semantics for stores.

The full definition of the Load-Acquire instruction is covered formally in the *Definition of the Armv8 memory model* on page E2-3980 and this introduction to the Load-Acquire instruction is not intended to contradict that section.

The basic principle of a Load-Acquire instruction is to introduce order between the memory access generated by the Load-Acquire instruction and the memory accesses appearing in program order after the Load-Acquire instruction, such that the memory access generated by the Load-Acquire instruction is Observed-by each PE, to the extent that that PE is required to observe the access coherently, before any of the memory accesses appearing in program order after the Load-Acquire instruction are Observed-by that PE, to the extent that the PE is required to observe the accesses coherently.

The use of a Load-Acquire instruction creates order between the Memory effects of instructions as described in the definition of Barrier-ordered-before.

The full definition of the Store-Release instruction is covered formally in the *Definition of the Armv8 memory model* on page E2-3980 and this introduction to the Store-Release instruction is not intended to contradict that section.

The basic principle of a Store-Release instruction is to introduce order between the memory accesses generated by the PEe executing the Store-Release instruction, together with those which originate from a different PE, to the extent that the PEe is required to observe them coherently, Observed-by the PEe before executing the Store-release.

The use of a Store-Release instruction creates order between the Memory effects of instructions as described in the definition of Barrier-ordered-before.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In addition, the use of a Load-Acquire or a Store-Release instruction on accesses to a Memory-mapped peripheral introduces order between the Memory effects of the instructions that access that peripheral, as described in the definition of Peripheral coherence order.

Load-Acquire and Store-Release, other than LDAEXD and STLEXD, access only a single data element. This access is single-copy atomic. The address of the data object must be aligned to the size of the data element being accessed, otherwise the access generates an Alignment fault.

LDAEXD and STLEXD access two data elements. The address supplied to the instructions must be doubleword aligned, otherwise the access generates an Alignment fault.

A Store-Release Exclusive instruction only has the release semantics if the store is successful.

Note

- Each Load-Acquire Exclusive and Store-Release Exclusive instruction is essentially a variant of the equivalent Load-Exclusive or Store-Exclusive instruction. All usage restrictions and single-copy atomicity properties:
  - That apply to the Load-Exclusive instructions also apply to the Load-Acquire Exclusive instructions.
  - That apply to the Store-Exclusive instructions also apply to the Store-Release Exclusive instructions.
- The Load-Acquire/Store-Release instructions can remove the requirement to use the explicit DMB memory barrier instruction.

Table E2-2 summarizes the Load-Acquire/Store-release instructions.

Table E2-2 Load-Acquire/Store-Release instructions

Data type	Load-Acquire	Store-Release	Load-Acquire Exclusive	Store-Release Exclusive
32-bit word	LDA	STL	LDAEX	STLEX
16-bit halfword	LDAH	STLH	LDAEXH	STLEXH
8-bit byte	LDAB	STLB	LDAEXB	STLEXB
64-bit doubleword	-	-	LDAEXD	STLEXD

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

E2.4 Ordering of translation table walks

If ARMv8.0-ETS is implemented, and a memory access RW_1 is Ordered-before a second memory access RW_2, then RW_1 is also Ordered-before any translation table walk generated by RW_2 that generates any of the following:

- A Translation fault.
- An Address size fault.
- An Access flag fault.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

E2.5 Caches and memory hierarchy

The implementation of a memory system depends heavily on the microarchitecture and therefore many details of the memory system are IMPLEMENTATION DEFINED. Armv8 defines the application level interface to the memory system, including a hierarchical memory system with multiple levels of cache. This section describes an application level view of this system. It contains the subsections:

- *Introduction to caches*.
- *Memory hierarchy*.
- *Implication of caches for the application programmer* on page E2-4000.
- *Preloading caches* on page E2-4002.

E2.5.1 Introduction to caches

A cache is a block of high-speed memory that contains a number of entries, each consisting of:

- Main memory address information, commonly known as a *tag*.
- The associated data.

Caches increase the average speed of a memory access and take account of two principles of locality:

Spatial locality

An access to one location is likely to be followed by accesses to adjacent locations. Examples of this principle are:

- Sequential instruction execution.
- Accessing a data structure.

Temporal locality

An access to an area of memory is likely to be repeated in a short time period. An example of this principle is the execution of a software loop.

To minimize the quantity of control information stored, the spatial locality property groups several locations together under the same tag. This logical block is commonly known as a *cache line*. When data is loaded into a cache, access times for subsequent loads and stores are reduced, resulting in overall performance benefits. An access to information already in a cache is known as a *cache hit*, and other accesses are called *cache misses*.

Normally, caches are self-managing, with the updates occurring automatically. Whenever the PE accesses a cacheable memory location, the cache is checked. If the access is a cache hit, the access occurs in the cache. Otherwise, the access is made to memory. Typically, when making this access, a cache location is allocated and the cache line loaded from memory. Armv8 permits different cache topologies and access policies, provided they comply with the memory coherency model described in this manual.

Caches introduce a number of potential problems, mainly because:

- Memory accesses can occur at times other than when the programmer would expect them.
- A data item can be held in multiple physical locations.

E2.5.2 Memory hierarchy

Typically memory close to a PE has very low latency, but is limited in size and expensive to implement. Further from the PE it is common to implement larger blocks of memory but these have increased latency. To optimize overall performance, an Armv8 memory system can include multiple levels of cache in a hierarchical memory system that exploits this trade-off between size and latency. Figure E2-1 on page E2-4000 shows an example of such a system in an Armv8-A system that supports virtual addressing.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

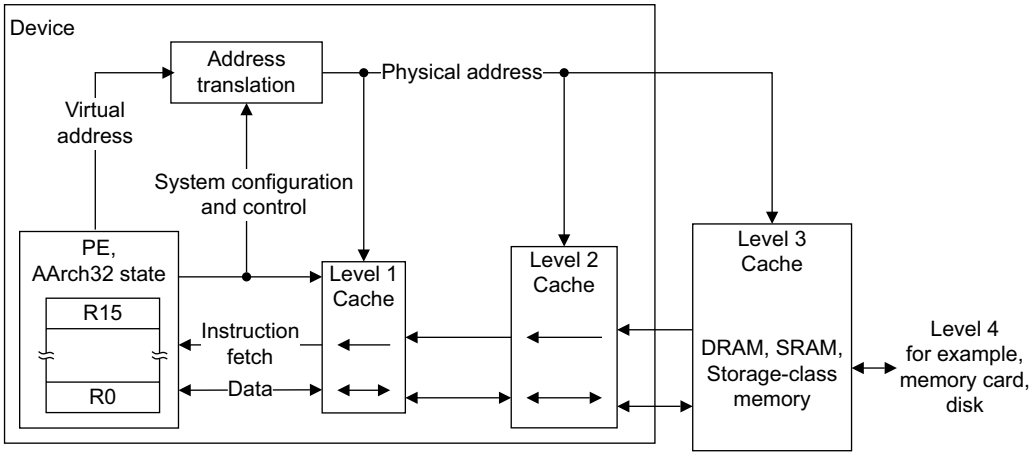


Figure E2-1 Multiple levels of cache in a memory hierarchy

Note

In this manual, in a hierarchical memory system, Level 1 refers to the level closest to the PE, as shown in Figure E2-1.

Instructions and data can be held in separate caches or in a unified cache. A cache hierarchy can have one or more levels of separate instruction and data caches, with one or more unified caches located at the levels closest to the main memory. Memory coherency for cache topologies can be defined using the conceptual points Point of Unification (PoU) and Point of Coherency (PoC). For more information, including the definitions of PoU and PoC, see *About cache maintenance in AArch32 state* on page G4-5895.

Note

Armv8 ARMv8.2-DCPoP adds architectural support for an additional conceptual point, Point of Persistence, but this support is provided only in AArch64 state. For more information see *About cache maintenance in AArch64 state* on page D4-2506.

The Cacheability and Shareability memory attributes

Cacheability and Shareability are two attributes that describe the memory hierarchy in a multiprocessing system:

Cacheability This term defines whether memory locations are allowed to be allocated into a cache or not. Cacheability is defined independently for Inner and Outer Cacheability locations.

Shareability This term defines whether memory locations are shareable between different agents in a system. Marking a memory location as shareable for a particular domain requires hardware to ensure that the location is coherent for all agents in that domain. Shareability is defined independently for Inner and Outer Shareability domains.

For more information about the Cacheability and Shareability attributes see *Memory types and attributes* on page E2-4010.

E2.5.3 Implication of caches for the application programmer

In normal operation, the caches are largely invisible to the application programmer. However they can become visible when there is a breakdown in the coherency of the caches. Such a breakdown can occur:

- When memory locations are updated by other agents in the system that do not use hardware management of coherency.
- When memory updates made from the application software must be made visible to other agents in the system, without the use of hardware management of coherency.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For example:

- In the absence of hardware management of coherency of DMA accesses, in a system with a DMA controller that reads memory locations that are held in the data cache of a PE, a breakdown of coherency occurs when the PE has written new data in the data cache, but the DMA controller reads the old data held in memory.
- In a Harvard cache implementation, where there are separate instruction and data caches, a breakdown of coherency occurs when new instruction data has been written into the data cache, but the instruction cache still contains the old instruction data.

Data coherency issues

Software can ensure the data coherency of caches in the following ways:

- By not using the caches in situations where coherency issues can arise. This can be achieved by:
  - Using Non-cacheable or, in some cases, Write-Through Cacheable memory.
  - Not enabling caches in the system.
- By using system calls to functions using cache maintenance instructions that execute at a higher Exception level.
- By using hardware coherency mechanisms to ensure the coherency of data accesses to memory for cacheable locations by observers within the different shareability domains, see *Non-shareable Normal memory* on page E2-4012 and *Shareable, Inner Shareable, and Outer Shareable Normal memory* on page E2-4011.

  Note

  The performance of these hardware coherency mechanisms is highly implementation-specific. In some implementations the mechanism suppresses the ability to cache shareable locations. In other implementations, cache coherency hardware can hold data in caches while managing coherency between observers within the shareability domains.

Synchronization and coherency issues between data and instruction accesses

How far ahead of the current point of execution instructions are fetched from is IMPLEMENTATION DEFINED. Such prefetching can be either a fixed or a dynamically varying number of instructions, and can follow any or all possible future execution paths. For all types of memory:

- The PE might have fetched the instructions from memory at any time since the last *Context synchronization event* on that PE.
- Any instructions fetched in this way might be executed multiple times, if this is required by the execution of the program, without being re-fetched from memory.

The Arm architecture does not require the hardware to ensure coherency between instruction caches and memory, even for locations of shared memory.

If software requires coherency between instruction execution and memory, it must manage this coherency using *Context synchronization event*s and cache maintenance instructions. These can only be accessed from an Exception level that is higher than EL0, and therefore require a system call, see *Exception-generating and exception-handling instructions* on page F1-4055. The following code sequence can be used for this purpose:

```
; Coherency example for data and instruction accesses within the same Inner Shareable domain.
; Enter this code with <Rt> containing a new 32-bit instruction,
; to be held in Cacheable space at a location pointed to by Rn. Use STRH in the first line
; instead of STR for a 16-bit instruction.
        STR Rt, [Rn]
        DCCMVAU Rn          ; Clean data cache by MVA to point of unification (PoU)
        DSB                 ; Ensure visibility of the data cleaned from cache
        ICIMVAU Rn          ; Invalidate instruction cache by MVA to PoU
        BPIMVA  Rn          ; Invalidate branch predictor by MVA to PoU
        DSB                 ; Ensure completion of the invalidations
        ISB                 ; Synchronize the fetched instruction stream
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A write has been made coherent with an instruction fetch of a shareability domain when:

CTR.{DIC, IDC} == {0, 0}

The location written to has been cleaned to the *Point of unification (PoU)* from the data cache, and that clean is complete for the shareability domain. Subsequently the location has been invalidated to the *Point of unification (PoU)* from the instruction cache, and that invalidation is complete for the shareability domain.

CTR.{DIC, IDC} == {1, 0}

The location written to has been cleaned to the *Point of unification (PoU)* from the data cache, and that clean is complete for the shareability domain.

CTR.{DIC, IDC} == {0, 1}

The write is complete for the shareability domain. Subsequently the location has been invalidated to the *Point of unification (PoU)* from the instruction cache, and that invalidation is complete for the shareability domain.

CTR.{DIC, IDC} == {1, 1}

The write is complete for the shareability domain.

Note

- For accesses that are Non-cacheable or Write-Through, the clean data cache instruction is not required. For accesses that are Non-cacheable, the invalidate instruction cache is not required, because in AArch32 state these accesses are not permitted to be held in an instruction cache.
- This code can be used when the thread of execution modifying the code is the same thread of execution that is executing the code. The Armv8 architecture limits the set of instructions that can be executed by one thread of execution as they are being modified by another thread of execution without requiring explicit synchronization. See *Concurrent modification and execution of instructions* on page E2-3978.

E2.5.4 Preloading caches

The Arm architecture provides the memory system hints `PLD` (Preload Data), `PLDW` (Preload Data With Intent To Write) and `PLI` (Preload Instruction) that software can use to communicate the expected use of memory locations to the hardware. The memory system can respond by taking actions that are expected to speed up the memory accesses if they occur. The effect of these memory system hints is IMPLEMENTATION DEFINED. Typically, implementations use this information to bring data or instruction locations into caches.

The Preload instructions are hints, and so implementations can treat them as `NOP`s without affecting the functional behavior of the device. The instructions cannot generate synchronous Data Abort exceptions, but the resulting memory system operations might, under exceptional circumstances, generate an asynchronous External abort, which is reported using an SError interrupt and taken using an asynchronous Data Abort exception. For more information, see *Data Abort exception* on page G1-5749.

A `PLD`, `PLDW`, or `PLI` instruction can only cause allocation to software-visible caching structures such caches or TLBs for memory locations that can be accessed, according to the permissions defined by the current translation regime or a translation regime for a higher Exception level in the current Security state, by any of:

- Reads.
- Writes.
- Instruction fetches.

A `PLD`, `PLDW`, or `PLI` instruction can access any memory location in Normal memory that can be accessed, according to the permissions defined by the current translation regime or a translation regime for a higher Exception level in the current Security state, by any of:

- Reads.
- Writes.
- Instruction fetches.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

In each case, the entire list applies to each of PLD, PLDW, and PLI.

A PLD, PLDW, or PLI instruction is guaranteed not to access any type of Device memory.

A PLI instruction must not perform any access that cannot be performed by a speculative instruction fetch by the processor. Therefore in a VMSA implementation, if all associated MMUs are disabled, a PLI instruction cannot access any memory location that cannot be accessed by instruction fetches.

The pseudocode enumeration PrefetchHint defines the prefetch hint types.

The Hint_Prefetch() pseudocode function signals to the memory system that memory accesses of the type hint to or from the specified address are likely to occur in the near future. The memory system might take some action to speed up the memory accesses when they do occur, such as preloading the specified address into one or more caches as indicated by the innermost cache level target and non-temporal hint stream.

For more information on PLD, PLI, and PLDW, see:

- *PLD, PLDW (immediate)* on page F5-4574.
- *PLD (literal)* on page F5-4576.
- *PLD, PLDW (register)* on page F5-4578.
- *PLI (immediate, literal)* on page F5-4580.
- *PLI (register)* on page F5-4583.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

E2.6 Alignment support

This section describes alignment support. It contains the following subsections:

- *Instruction alignment*.
- *Unaligned data access*.
- *Cases where unaligned accesses are CONSTRAINED UNPREDICTABLE* on page E2-4005.
- *Unaligned data access restrictions* on page E2-4005.
- *Generation of Alignment faults by Load/store multiple accesses to Device memory* on page E2-4005.

For more information about Alignment faults see *Alignment faults* on page G5-6020.

E2.6.1 Instruction alignment

A32 instructions are word-aligned.

T32 instructions are halfword-aligned.

E2.6.2 Unaligned data access

An Armv8 implementation must support unaligned data accesses to Normal memory by some load and store instructions. As Table E2-3 shows, software can control whether a misaligned access to Normal memory by one of these instructions causes an Alignment fault Data Abort exception:

- By setting SCTLR.A, for unaligned accesses from any mode other than Hyp mode.
- By setting HSCTLR.A, for unaligned accesses from Hyp mode.

Table E2-3 Alignment requirements of load/store instructions

Instructions	Alignment check	Result if check fails when: SCTLR.A or HSCTLR.A is 0	Result if check fails when: SCTLR.A or HSCTLR.A is 1
LDRB, LDREXB, LDRBT, LDRSB, LDRSBT, STRB, STREXB, STRBT, TBB	None	-	-
LDRH, LDRHT, LDRSH, LDRSHT, STRH, STRHT, TBH	Halfword	Unaligned access	Alignment fault
LDREXH, STREXH, LDAH, STLH, LDAEXH, STLEXH	Halfword	Alignment fault	Alignment fault
LDR, LDRT, STR, STRT PUSH, encodings T3 and A2 only POP, encodings T3 and A2 only	Word	Unaligned access	Alignment fault
LDREX, STREX, LDA, STL, LDAEX, STLEX	Word	Alignment fault	Alignment fault
LDREXD, STREXD, LDAEXD, STLEXD	Doubleword	Alignment fault	Alignment fault
All forms of LDM and STM, LDRD, RFE, SRS, STRD	Word	Alignment fault	Alignment fault
LDC, STC	Word	Alignment fault	Alignment fault
VLDM, VLDR, VPOP, VPUSH, VSTM, VSTR	Word	Alignment fault	Alignment fault
VLD1, VLD2, VLD3, VLD4, VST1, VST2, VST3, VST4, all with standard alignment	Element size	Unaligned access	Alignment fault
VLD1, VLD2, VLD3, VLD4, VST1, VST2, VST3, VST4, all with :<align> specified[a]	As specified by :<align>	Alignment fault	Alignment fault

a. Previous versions of this manual used @<align> to specify alignment. Both forms are supported, see Chapter F6 *T32 and A32 Advanced SIMD and Floating-point Instruction Descriptions*for more information.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

Any unaligned access to any type of Device memory generates an Alignment fault, see *Alignment faults* on page G5-6020.

E2.6.3 Cases where unaligned accesses are CONSTRAINED UNPREDICTABLE

Any load instruction that is not faulted by the alignment restrictions shown in Table E2-3 on page E2-4004 and that loads the PC has CONSTRAINED UNPREDICTABLE behavior if the address it loads from is not word-aligned, see *Loads and Stores to unaligned locations* on page K1-7832. This overrules any permitted Load/Store behavior shown in Table E2-3 on page E2-4004.

E2.6.4 Unaligned data access restrictions

The following points apply to unaligned data accesses in Armv8:

- Accesses are not guaranteed to be single-copy atomic except at the byte access level, see *Atomicity in the Arm architecture* on page E2-3976.
- Unaligned accesses typically take a number of additional cycles to complete compared to a naturally-aligned access.
- An operation that performs an unaligned access can abort on any memory access that it makes, and can abort on more than one access. This means that an unaligned access that occurs across a page boundary can generate an abort on either side of the boundary.

E2.6.5 Generation of Alignment faults by Load/store multiple accesses to Device memory

When ARMv8.2-LSMAOC is implemented and the value of the applicable nTLSMD field is 0, any memory access by an AArch32 Load Multiple or Store Multiple instruction to an address that the stage 1 translation assigns as Device-nGRE, Device-nGnRE, or Device-nGnRnE generates an Alignment fault.

The applicable nTLSMD field is the field in the SCTLR_EL1, SCTLR_EL2, HSCTLR, or SCTLR register that applies to the Exception level and Security state at which the `LDM` or `STM` instruction is executed.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

E2.7 Endian support

General description of endianness in the Arm architecture describes the relationship between endianness and memory addressing in the Arm architecture.

The following subsections then describe the endianness schemes supported by the architecture:

- *Instruction endianness*.
- *Data endianness* on page E2-4007.
- *Endianness of memory-mapped peripherals* on page E2-4008.

E2.7.1 General description of endianness in the Arm architecture

This section only describes memory addressing and the effects of endianness for data elements up to doubleword of 64 bits. However, this description can be extended to apply to larger data elements.

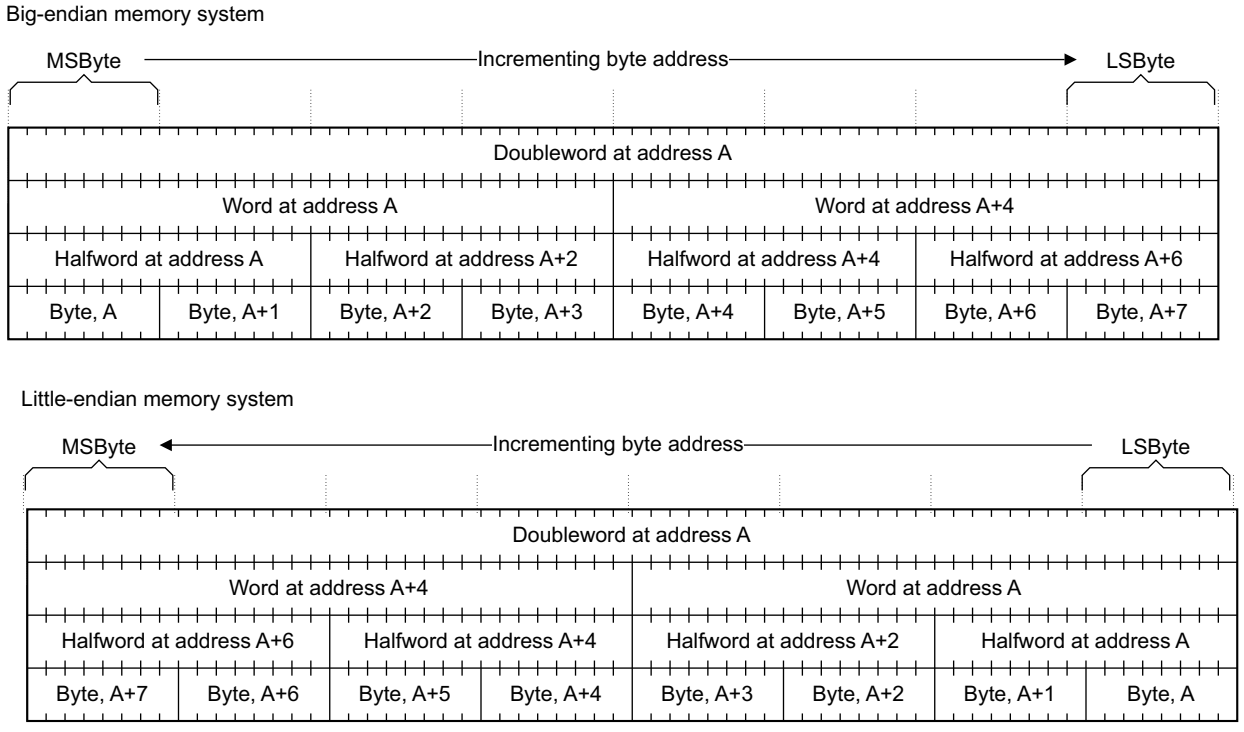
For an address A, Figure E2-2 shows, for big-endian and little-endian memory systems, the relationship between:

- The doubleword at address A.
- The words at addresses A and A+4.
- The halfwords at addresses A, A+2, A+4, and A+6.
- The bytes at addresses A, A+1, A+2, A+3, A+4, A+5, A+6, and A+7.

The terms in Figure E2-2 have the following definitions:

MSByte Most-significant byte.

LSByte Least-significant byte.



In this figure, *Byte, A+1* is an abbreviation for *Byte at address A+1*

Figure E2-2 Endianness relationships in AArch32 state

E2.7.2 Instruction endianness

In Armv8-A, the mapping of instruction memory is always little-endian.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

E2.7.3 Data endianness

The size of the data value that is loaded or stored is the size that is used for the purpose of endian conversion for floating-point, Advanced SIMD, and general-purpose register loads and stores.

Table E2-4 shows the element sizes of all the load/store instructions, for all instruction sets.

Table E2-4 Element size of load/store instructions

Instructions	Element size
LDRB, LDREXB, LDRBT, LDRSB, LDRSBT, STRB, STREXB, STRBT, TBB	Byte
LDRH, LDREXH, LDRHT, LDRSH, LDRSHT, STRH, STREXH, STRHT, TBH	Halfword
LDR, LDRT, LDREX, STR, STRT, STREX	Word
LDRD, LDREXD, STRD, STREXD	Word
All forms of LDM, PUSH, POP, RFE, SRS, all forms of STM,	Word
LDC, STC	Word
Forms of VLDM, VLDR, VPOP, VPUSH, VSTM, VSTR that transfer 32-bit Si registers	Word
Forms of VLDM, VLDR, VPOP, VPUSH, VSTM, VSTR that transfer 64-bit Di registers	Doubleword
VLD1, VLD2, VLD3, VLD4, VST1, VST2, VST3, VST4	Element size of the Advanced SIMD access

CPSR.E determines the data endianness.

The data size used for endianness conversions:

- Is the size of the data value that is loaded or stored for Advanced SIMD and floating-point register and general-purpose register loads and stores.
- Is the size of the data element that is loaded or stored for Advanced SIMD element and data structure loads and stores. For more information see *Endianness in Advanced SIMD* on page E2-4008.

Instructions to reverse bytes in registers

An application or device driver might have to interface to memory-mapped peripheral registers or shared memory structures that are not the same endianness as the internal data structures. Similarly, the endianness of the operating system might not match that of the peripheral registers or shared memory. In these cases, the PE requires an efficient method to transform explicitly the endianness of the data.

Table E2-5 shows the instructions that provide this functionality in the A32 and T32 instruction sets.

Table E2-5 Byte reversal instructions

Function	T32 / A32 instruction	Notes
Reverse bytes in whole register	REV	For use with general purpose registers.
Reverse bytes in 16-bit halfwords	REV16	For use with general purpose registers.
Reverse bytes in halfword and sign-extend	REVSH	For use with general purpose registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table E2-5 Byte reversal instructions (continued)

Function	T32 / A32 instruction	Notes
Reverse elements in doublewords, vector	`VREV64`	For use with registers in the SIMD and floating-point register file
Reverse elements in words, vector	`VREV32`	For use with registers in the SIMD and floating-point register file
Reverse elements in halfwords, vector	`VREV16`	For use with registers in the SIMD and floating-point register file

Endianness in Advanced SIMD

Advanced SIMD element Load/Store instructions transfer vectors of elements between memory and the SIMD and floating-point register file. An instruction specifies both the length of the transfer and the size of the data elements being transferred. This information is used by the PE to load and store data correctly in both big-endian and little-endian systems.

Consider, for example, the A32 or T32 instruction:

```
VLD1.16 {D0}, [R1]
```

This loads a 64-bit register with four 16-bit values. The four elements appear in the register in array order, with the lowest indexed element fetched from the lowest address. The order of bytes in the elements depends on the endianness configuration, as shown in Figure E2-3. Therefore, the order of the elements in the registers is the same regardless of the endianness configuration.

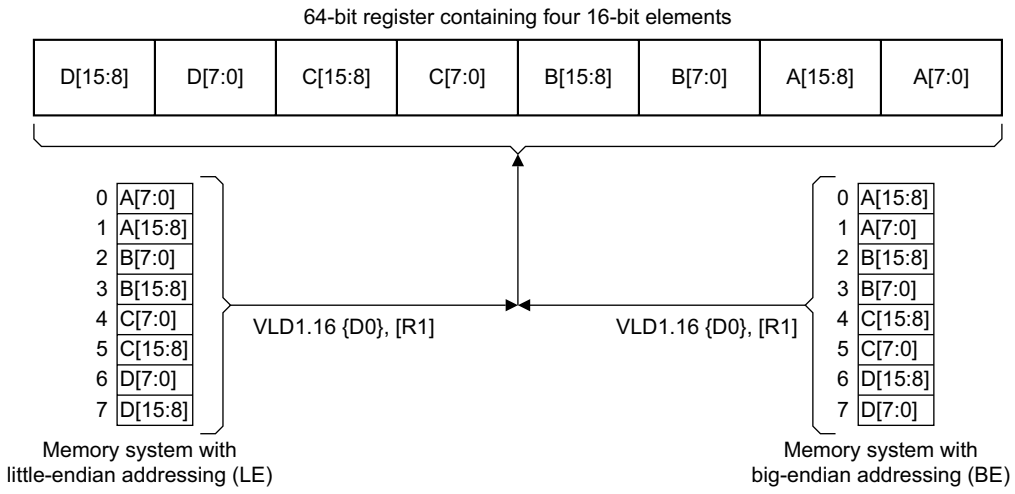


Figure E2-3 Advanced SIMD byte order example for AArch32 state

For information about the alignment of Advanced SIMD instructions see *Alignment support* on page E2-4004.

The `BigEndian()` pseudocode function determines the current endianness of the data.

The `BigEndianReverse()` pseudocode function reverses the endianness of a bitstring.

The `BigEndian()` and `BigEndianReverse()` functions are defined in Chapter J1 *Armv8 Pseudocode*.

E2.7.4 Endianness of memory-mapped peripherals

All memory-mapped peripherals defined in the Arm architecture must be little-endian.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Peripherals to which this requirement applies include:

- Memory-mapped register interfaces to a debugger, or to a cross-trigger interface, see Chapter H8 *About the External Debug Registers*.
- The memory-mapped register interface to the system level implementation of the Generic Timer, see Chapter I2 *System Level Implementation of the Generic Timer*.
- A memory-mapped register interface to the Performance Monitors, see Chapter I3 *Recommended External Interface to the Performance Monitors*.
- A memory-mapped register interface to the Activity Monitors, see Chapter I4 *Recommended External Interface to the Activity Monitors*.
- Memory-mapped register interfaces to an Arm Generic Interface Controller, see the *ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0 and version 4.0.*
- The memory-mapped register interface to an Arm trace component. See, for example, the *ARM® Embedded Trace Macrocell Architecture Specification, ETMv4.*

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

E2.8 Memory types and attributes

In Armv8 the ordering of accesses for addresses in memory, referred to as the memory order model, is defined by the memory attributes. The following sections describe this model:

- *Normal memory*.
- *Device memory* on page E2-4014.
- *Memory access restrictions* on page E2-4019.

E2.8.1 Normal memory

The Normal memory type attribute applies to most memory in a system. It indicates that the hardware is permitted by the architecture to perform *Speculative* data read accesses to these locations, regardless of the access permissions for these locations.

The Normal memory type has the following properties:

- A write to a memory location with the Normal attribute completes in finite time.
- Writes to a memory location with the Normal memory type that is either Non-cacheable or Write-through cacheable for both the Inner and Outer cacheability must reach the endpoint for that location in the memory system in finite time. Two writes to the same location, where at least one is using the Normal memory type, might be merged before they reach the endpoint unless there is an ordered-before relationship between the two writes.
- Unaligned memory accesses can access Normal memory if the system is configured to generate such accesses.
- There is no requirement for the memory system beyond the PE to be able to identify the elements accessed by multi-register Load/Store instructions. See *Multi-register loads and stores that access Normal memory* on page E2-4014.

Note

- The Normal memory attribute is appropriate for locations of memory that are idempotent, meaning that they exhibit all of the following properties:
  - Read accesses can be repeated with no side-effects.
  - Repeated read accesses return the last value written to the resource being read.
  - Read accesses can fetch additional memory locations with no side-effects.
  - Write accesses can be repeated with no side-effects if the contents of the location accessed are unchanged between the repeated writes or as the result of an exception, as described in this section.
  - Unaligned accesses can be supported.
  - Accesses can be merged before accessing the target memory system.
- Normal memory allows speculative reads and may be affected by intermediate buffering and forwarding of data. If non-idempotent memory locations are mapped as Normal memory, the following may occur:
  - Memory accesses return UNKNOWN values.
  - UNPREDICTABLE effects on memory-mapped peripherals.
- An instruction that generates a sequence of accesses as described in *Atomicity in the Arm architecture* on page E2-3976 might be abandoned as a result of an exception being taken during the sequence of accesses. On return from the exception the instruction is restarted, and therefore one or more of the memory locations might be accessed multiple times. This can result in repeated write accesses to a location that has been changed between the write accesses.

The following sections describe the other attributes for Normal memory:

- *Shareable Normal memory* on page E2-4011.
- *Non-shareable Normal memory* on page E2-4012.
- *Cacheability attributes for Normal memory* on page E2-4012.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

See also:

- *Multi-register loads and stores that access Normal memory* on page E2-4014.
- *Atomicity in the Arm architecture* on page E2-3976.
- *Memory barriers* on page E2-3991. For accesses to Normal memory, a `DMB` instruction is required to ensure the required ordering.
- *Concurrent modification and execution of instructions* on page E2-3978.

Shareable Normal memory

A Normal memory location has a Shareability attribute that is defined as one of:

- Inner Shareable.
- Outer Shareable.
- Non-shareable.

The shareability attributes define the data coherency requirements of the location, that hardware must enforce. They do not affect the coherency requirements of instruction fetches, see *Synchronization and coherency issues between data and instruction accesses* on page E2-4001.

Note

- System designers can use the Shareability attribute to specify the locations in Normal memory for which coherency must be maintained. However, software developers must not assume that specifying a memory location as Non-shareable permits software to make assumptions about the incoherency of the location between different PEs in a shared memory system. Such assumptions are not portable between different multiprocessing implementations that might use the Shareability attribute. Any multiprocessing implementation might implement caches that are shared, inherently, between different PEs.
- This architecture assumes that all PEs that use the same operating system or hypervisor are in the same Inner Shareable shareability domain.

Shareable, Inner Shareable, and Outer Shareable Normal memory

The Arm architecture abstracts the system as a series of Inner and Outer Shareability domains.

Each Inner Shareability domain contains a set of observers that are data coherent for each member of that set for data accesses with the Inner Shareable attribute made by any member of that set.

Each Outer Shareability domain contains a set of observers that are data coherent for each member of that set for data accesses with the Outer Shareable attribute made by any member of that set.

The following properties also hold:

- Each observer is only a member of a single Inner Shareability domain.
- Each observer is only a member of a single Outer Shareability domain.
- All observers in an Inner Shareability domain are always members of the same Outer Shareability domain. This means that an Inner Shareability domain is a subset of an Outer Shareability domain, although it is not required to be a proper subset.

Note

- Because all data accesses to Non-cacheable locations are data coherent to all observers, Non-cacheable locations are always treated as Outer Shareable.
- The Inner Shareable domain is expected to be the set of PEs controlled by a single hypervisor or operating system.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The details of the use of the Shareability attributes are system-specific. Example E2-1 shows how they might be used.

Example E2-1 Use of shareability attributes

In an implementation, a particular subsystem with two clusters of PEs has the requirement that:

- In each cluster, the data caches or unified caches of the PEs in the cluster are transparent for all data accesses to memory locations with the Inner Shareable attribute.
- However, between the two clusters, the caches:
  - Are not required to be coherent for data accesses that have only the Inner Shareable attribute.
  - Are coherent for data accesses that have the Outer Shareable attribute.

In this system, each cluster is in a different Shareability domain for the Inner Shareable attribute, but all components of the subsystem are in the same Shareability domain for the Outer Shareable attribute.

A system might implement two such subsystems. If the data caches or unified caches of one subsystem are not transparent to the accesses from the other subsystem, this system has two Outer Shareable Shareability domains.

Having two levels of shareability means system designers can reduce the performance and power overhead for shared memory locations that do not need to be part of the Outer Shareable Shareability domain.

For Shareable Normal memory, the Load-Exclusive and Store-Exclusive synchronization primitives take account of the possibility of accesses by more than one observer in the same Shareability domain.

Non-shareable Normal memory

For Normal memory locations, the Non-shareable attribute identifies Normal memory that is likely to be accessed only by a single PE.

A location in Normal memory with the Non-shareable attribute does not require the hardware to make data accesses by different observers coherent, unless the memory is Non-cacheable. For a Non-shareable location, if other observers share the memory system, software must use cache maintenance instructions, if the presence of caches might lead to coherency issues when communicating between the observers. This cache maintenance requirement is in addition to the barrier operations that are required to ensure memory ordering.

For Non-shareable Normal memory, it is IMPLEMENTATION DEFINED whether the Load-Exclusive and Store-Exclusive synchronization primitives take account of the possibility of accesses by more than one observer.

Cacheability attributes for Normal memory

In addition to being Outer Shareable, Inner Shareable or Non-shareable, each region of Normal memory is assigned a Cacheability attribute that is one of:

- Write-Through Cacheable.
- Write-Back Cacheable.
- Non-cacheable.

Also, for Write-Through Cacheable and Write-Back Cacheable Normal memory regions:

- A region might be assigned cache allocation hints for read and write accesses.
- It is IMPLEMENTATION DEFINED whether the cache allocation hints can have an additional attribute of Transient or Non-transient.

For more information see *Cacheability, cache allocation hints, and cache transient hints* on page G4-5892.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A memory location can be marked as having different cacheability attributes, for example when using aliases in a VA to PA mapping:

- If the attributes differ only in the cache allocation hint this does not affect the behavior of accesses to that location.
- For other cases see *Mismatched memory attributes* on page E2-4020.

The cacheability attributes provide a mechanism of coherency control with observers that lie outside the Shareability domain of a region of memory. In some cases, the use of Write-Through Cacheable or Non-cacheable regions of memory might provide a better mechanism for controlling coherency than the use of hardware coherency mechanisms or the use of cache maintenance routines. To this end, the architecture requires the following properties for Non-cacheable or Write-Through Cacheable memory:

- A completed write to a memory location that is Non-cacheable or Write-Through Cacheable for a level of cache made by an observer accessing the memory system inside the level of cache is visible to all observers accessing the memory system outside the level of cache without the need of explicit cache maintenance.
- A completed write to a memory location that is Non-cacheable for a level of cache made by an observer accessing the memory system outside the level of cache is visible to all observers accessing the memory system inside the level of cache without the need of explicit cache maintenance.

Note

Implementations can use the cache allocation hints to indicate a probable performance benefit of caching. For example, a programmer might know that a piece of memory is not going to be accessed again and would be better treated as Non-cacheable. The distinction between memory regions with attributes that differ only in the cache allocation hints exists only as a hint for performance.

For Normal memory, the Arm architecture provides cacheability attributes that are defined independently for each of two conceptual levels of cache, the *inner* and the *outer* cache. The relationship between these conceptual levels of cache and the implemented physical levels of cache is IMPLEMENTATION DEFINED, and can differ from the boundaries between the Inner and Outer Shareability domains. However:

- Inner refers to the innermost caches, meaning the caches that are closest to the PE, and always includes the lowest level of cache.
- No cache that is controlled by the Inner cacheability attributes can lie outside a cache that is controlled by the Outer cacheability attributes.
- An implementation might not have any outer cache.

Example E2-2, Example E2-3 on page E2-4014, and Example E2-4 on page E2-4014 describe the possible ways of implementing a system with three levels of cache, *level 1* (L1) to *level 3* (L3).

Note

- L1 cache is the level closest to the PE, see *Memory hierarchy* on page E2-3999.
- When managing coherency, system designs must consider both the inner and outer cacheability attributes, as well as the Shareability attributes. This is because hardware might have to manage the coherency of caches at one conceptual level, even when another conceptual level has the Non-cacheable attribute.

Example E2-2 Implementation with two inner and one outer cache levels

Implement the three levels of cache in the system, L1 to L3, with:

- The Inner cacheability attribute applied to L1 and L2 cache.
- The Outer cacheability attribute applied to L3 cache.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Example E2-3 Implementation with three inner and no outer cache levels

Implement the three levels of cache in the system, L1 to L3, with the Inner cacheability attribute applied to L1, L2, and L3 cache. Do not use the Outer cacheability attribute.

Example E2-4 Implementation with one inner and two outer cache levels

Implement the three levels of cache in the system, L1 to L3, with:

- The Inner cacheability attribute applied to L1 cache.
- The Outer cacheability attribute applied to L2 and L3 cache.

Multi-register loads and stores that access Normal memory

For all instructions that load or store more than one general-purpose register from an Exception level there is no requirement for the memory system beyond the PE to be able to identify the size of the elements accessed by these load or store instructions.

For all instructions that load or store more than one general-purpose register from an Exception level the order in which the registers are accessed is not defined by the architecture.

For all instructions that load or store one or more registers from the SIMD and floating-point register file from an Exception level there is no requirement for the memory system beyond the PE to be able to identify the size of the element accessed by these load or store instructions.

E2.8.2 Device memory

The Device memory type attributes define memory locations where an access to the location can cause side-effects, or where the value returned for a load can vary depending on the number of loads performed. Typically, the Device memory attributes are used for memory-mapped peripherals and similar locations.

The attributes for Armv8 Device memory are:

Gathering Identified as G or nG, see *Gathering* on page E2-4016.

Reordering Identified as R or nR, see *Reordering* on page E2-4017.

Early Write Acknowledgement

Identified as E or nE, see *Early Write Acknowledgement* on page E2-4018.

The Armv8 Device memory types are:

Device-nGnRnE Device non-Gathering, non-Reordering, No Early write acknowledgement.

Equivalent to the Strongly-ordered memory type in earlier versions of the architecture.

Device-nGnRE Device non-Gathering, non-Reordering, Early Write Acknowledgement.

Equivalent to the Device memory type in earlier versions of the architecture.

Device-nGRE Device non-Gathering, Reordering, Early Write Acknowledgement.

Armv8 adds this memory type to the translation table formats found in earlier versions of the architecture. The use of barriers is required to order accesses to Device-nGRE memory.

The Device-nGRE memory type is introduced into the AArch32 translation table formats when the PE is using the Long Descriptor Translation Table format.

Device-GRE Device Gathering, Reordering, Early Write Acknowledgement.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Armv8 adds this memory type to the translation table formats found in earlier versions of the architecture. Device-GRE memory has the fewest constraints. It behaves similar to Normal memory, with the restriction that speculative accesses to Device-GRE memory is forbidden.

The Device-GRE memory type is introduced into the AArch32 translation table formats when the PE is using the Long Descriptor Translation Table format.

Collectively these are referred to as *any Device memory type*. Going down the list, the memory types are described as getting *weaker*; conversely the going up the list the memory types are described as getting *stronger*.

Note

- As the list of types shows, these additional attributes are hierarchical. For example, a memory location that permits Gathering must also permit Reordering and Early Write Acknowledgement.
- The architecture does not require an implementation to distinguish between each of these memory types and Arm recognizes that not all implementations will do so. The subsection that describes each of the attributes, describes the implementation rules for the attribute.
- Earlier versions of the Arm architecture defined the following memory types:
  - Strongly-ordered memory. This is the equivalent of the Device-nGnRnE memory type.
  - Device memory. This is the equivalent of the Device-nGnRE memory type.

All of these memory types have the following properties:

- Speculative data accesses are not permitted to any memory location with any Device memory attribute. This means that each memory access to any Device memory type must be one that would be generated by a simple sequential execution of the program.

  An exception to this applies:

  - Reads generated by the Advanced SIMD and floating-point instructions can access bytes that are not explicitly accessed by the instruction if the bytes accessed are in a 16-byte window, aligned to 16-bytes, that contains at least one byte that is explicitly accessed by the instruction.

  Note

  - An instruction that generates a sequence of accesses as described in *Atomicity in the Arm architecture* on page E2-3976 might be abandoned as a result of an exception being taken during the sequence of accesses. On return from the exception the instruction is restarted, and therefore one or more of the memory locations might be accessed multiple times. This can result in repeated accesses to a location where the program only defines a single access. For this reason, Arm strongly recommends that no accesses to Device memory are performed from a single instruction that spans the boundary of a translation granule or which in some other way could lead to some of the accesses being aborted.
  - Write speculation that is visible to other observers is prohibited for all memory types.

- A write to a memory location with any Device memory type completes in finite time.
- If a value that would be returned from a read of a memory location with the Device memory type changes without an explicit write by an observer, this change must also be globally observed for all observers in the system in finite time. Such a change might occur in a peripheral location that holds status information.
- Data accesses to memory locations are coherent for all observers in the system, and correspondingly are treated as being Outer Shareable.
- A memory location with any Device memory attribute cannot be allocated into a cache.
- Writes to a memory location with any Device memory attribute must reach the endpoint for that address in the memory system in finite time. Two writes of Device memory type to the same location might be merged before they reach the endpoint, unless both writes have the non-Gathering attribute or there is an ordered-before relationship between the two writes.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- All accesses to memory with any Device memory attribute must be aligned. Any unaligned access generates an Alignment fault at the first stage of translation that defined the location as being Device.

  Note

  In the Non-secure PL1&0 translation regime in systems where HCR.TGE==1 and HCR.DC==0, any Alignment fault that results from the fact that all locations are treated as Device is a fault at the first stage of translation. This causes the value of HSR.ISS.[24] to be 0.

- Hardware does not prevent speculative instruction fetches from a memory location with any of the Device memory attributes unless the memory location is also marked as Execute-never for all Exception levels.

  Note

  This means that to prevent speculative instruction fetches from memory locations with Device memory attributes, any location that is assigned any Device memory type must also be marked as Execute-never for all Exception levels. Failure to mark a memory location with any Device memory attribute as Execute-never for all Exception levels is a programming error.

See also *Memory access restrictions* on page E2-4019.

The memory types for Translation table walks cannot be defined as any Device memory type within the TCR. For the Non-secure EL1&0 translation regime, the memory accesses made during a stage 1 translation table walk are subject to a stage 2 translation, and as a result of this second stage of translation, the accesses from the first stage translation table walk might be made to memory locations with any Device memory type. These accesses might be made speculatively. When the value of the HCR.PTW bit is 1, a stage 2 permission fault is generated if a first stage translation table walk is made to any Device memory type.

For instruction fetches, if branches cause the program counter to point to an area of memory with the Device attribute which is not marked as Execute-never for the current Exception level, an implementation can either:

- Treat the instruction fetch as if it were to a memory location with the Normal Non-cacheable attribute.
- Take a Permission fault.

Gathering

In the Device memory attribute:

G Indicates that the location has the Gathering attribute.

nG Indicates that the location does not have the Gathering attribute, meaning it is non-Gathering.

The Gathering attribute determines whether it is permissible for either:

- Multiple memory accesses of the same type, read or write, to the same memory location to be merged into a single transaction.
- Multiple memory accesses of the same type, read or write, to different memory locations to be merged into a single memory transaction on an interconnect.

For memory types with the Gathering attribute, either of these behaviors is permitted, provided that the ordering and coherency rules of the memory location are followed.

For memory types with the non-Gathering attribute, neither of these behaviors is permitted. As a result:

- The number of memory accesses that are made corresponds to the number that would be generated by a simple sequential execution of the program.
- All access occur at their programmed size, except that there is no requirement for the memory system beyond the PE to be able to identify the elements accessed by multi-register Load/Store instructions. See *Multi-register loads and stores that access Device memory* on page E2-4018.

Gathering between memory accesses separated by a memory barrier that affects those memory accesses is not permitted.

Gathering between two memory accesses generated by a Load-Acquire/Store-Release is not permitted.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A read from a memory location with the non-Gathering attribute cannot come from a cache or a buffer, but must come from the endpoint for that address in the memory system. Typically this is a peripheral or physical memory.

Note

- A read from a memory location with the Gathering attribute can come from intermediate buffering of a previous write, provided that:
  - The accesses are not separated by a `DMB` or `DSB` barrier that affects both of the accesses.
  - The accesses are not separated by other ordering constructions that require that the accesses are in order. Such a construction might be a combination of Load-Acquire and Store-Release.
  - The accesses are not generated by a Store-Release instruction.
- The Arm architecture only defines programmer visible behavior. Therefore, gathering can be performed if a programmer cannot tell whether gathering has occurred.

An implementation is permitted to perform an access with the Gathering attribute in a manner consistent with the requirements specified by the non-Gathering attribute.

An implementation is not permitted to perform an access with the non-Gathering attribute in a manner consistent with the relaxations allowed by the Gathering attribute.

Reordering

In the Device memory attribute:

R Indicates that the location has the Reordering attribute.

nR Indicates that the location does not have the Reordering attribute, meaning it is non-Reordering.

For all memory types with the non-Reordering attribute, the order of memory accesses arriving at a single peripheral of IMPLEMENTATION DEFINED size, as defined by the peripheral, must be the same order that occurs in a simple sequential execution of the program. That is, the accesses appear in program order. This ordering applies to all accesses using any of the memory types with the non-Reordering attribute. As a result, if there is a mixture of Device-nGnRE and Device-nGnRnE accesses to the same peripheral, these occur in program order. If the memory accesses are not to a peripheral, then this attribute imposes no restrictions.

Note

- The IMPLEMENTATION DEFINED size of the single peripheral is the same as applies for the ordering guarantee provided by the `DMB` instruction.
- The Arm architecture only defines programmer visible behavior. Therefore, reordering can be performed if a programmer cannot tell whether reordering has occurred.

An implementation is permitted to perform an access with the Reordering attribute in a manner consistent with the requirements specified by the non-Reordering attribute.

An additional relaxation is that an implementation is not permitted to perform an access with the non-Reordering attribute in a manner consistent with the relaxations allowed by the Reordering attribute.

The non-Reordering attribute does not require any additional ordering, other than that which applies to Normal memory, between:

- Accesses to one physical address with the non-Reordering attribute and accesses to a different physical address with the Reordering attribute.
- Access to one physical address with the non-Reordering attribute and access to a different physical address to Normal memory.
- Accesses with the non-Reordering attribute and accesses to different peripherals of IMPLEMENTATION DEFINED size.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Early Write Acknowledgement

In the Device memory attribute:

E Indicates that the location has the Early Write Acknowledgement attribute.

nE Indicates that the location has the No Early Write Acknowledgement attribute.

For memory system endpoints where the system architecture in which the PE is operating requires that acknowledgement of a write comes from the endpoint, assigning the No Early Write Acknowledgement attribute to a Device memory location guarantees that:

- Only the endpoint of the write access returns a write acknowledgement of the access.
- No earlier point in the memory system returns a write acknowledgement.

This means that a `DSB` barrier instruction, executed by the PE that performed the write to the No Early Write Acknowledgement Location, completes only after the write has reached its endpoint in the memory system.

Peripherals are an example of system endpoints that require that the acknowledgement of a write comes from the endpoint.

Note

- The Early Write Acknowledgement attribute only affects where the endpoint acknowledgement is returned from, and does not affect the ordering of arrival at the endpoint between accesses, which is determined by either the Device Reordering attribute, or the use of barriers to create order.
- The areas of the physical memory map for which write acknowledgement from the endpoint is required is outside the scope of the Arm Architecture definition and must be defined as part of the system architecture in which the PE is operating. In particular, regions of memory handled as PCIe configuration writes are expected to support write acknowledgement from the endpoint.
- Arm recognizes that not all areas of a physical memory map will be capable of supporting write acknowledgement from the endpoint. In particular, Arm expects that regions of memory handled as posted writes under PCIe will not support write acknowledgement from the endpoint.
- For maximum software compatibility, Arm strongly recommends that all peripherals for which standard software drivers expect that the use of a `DSB` instruction will determine that a write has reached its endpoint are placed in areas of the physical memory map that support write acknowledgement from the endpoint.

Multi-register loads and stores that access Device memory

For all instructions that load or store more than one general-purpose register there is no requirement for the memory system beyond the PE to be able to identify the size of the elements accessed by these load and store instructions.

For all instructions that load or store one or more registers from the SIMD and floating-point register file there is no requirement for the memory system beyond the PE to be able to identify the size of the element accessed by these load and store instructions.

For an `LDRD`, `STRD`, or `LDM` instruction with a register list that includes the PC, or an `STM` instruction with a register list that includes the PC, the order in which the registers are accessed is not defined by the architecture.

For a load or store of an Advanced SIMD element or structure, the order in which the registers are accessed is not defined by the architecture.

For a `VLDM` and `VSTM` instruction with a register list that does not include the PC, all registers are accessed in ascending address order for accesses to Device memory with the non-Reordering attribute.

For a `LDM` or `STM` instruction with a register list that does not include the PC:

- When ARMv8.2-LSMAOC is not implemented, and when ARMv8.2-LSMAOC is implemented and the value of the applicable LSMAOE field is 1, all registers are accessed in ascending address order for accesses to Device memory with the non-Reordering attribute.
- When ARMv8.2-LSMAOC is implemented and the value of the applicable LSMAOE field is 0, no memory accesses are required to be ordered.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- When ARMv8.2-LSMAOC is implemented and the value of the applicable nTLSMD field is 0, any memory access to an address that the stage 1 translation assigns as Device-nGRE, Device-nGnRE, or Device-nGnRnE generates an Alignment fault.

The applicable LSMAOE or nTLSMD field is the field in the SCTLR_EL1, SCTLR_EL2, HSCTLR, or SCTLR register that applies to the Exception level and Security state at which the `LDM` or `STM` instruction is executed.

Armv8.2 deprecates software relying on accesses to Device memory made by a single `LDM` or `STM` instruction not being reordered.

E2.8.3 Memory access restrictions

The following restrictions apply to memory accesses:

- For accesses to any two bytes, *p* and *q*, that are generated by the same instruction:
  - The bytes *p* and *q* must have the same memory type and Shareability attributes. otherwise the results are CONSTRAINED UNPREDICTABLE. For example, an `LDC`, `LDM`, `LDRD` `STC`, `STM` `or` `STRD` instruction, or an unaliged load or store that spans the boundary between Normal memory and Device memory is CONSTRAINED UNPREDICTABLE.
  - Except for possible differences in the cache allocation hints, Arm deprecates having different cacheability attributes for bytes *p* and *q*.

  For the permitted CONSTRAINED UNPREDICTABLE behavior, see *Crossing a page boundary with different memory types or Shareability attributes* on page K1-7841.

- If the accesses of an instruction that causes multiple accesses to any type of Device memory cross a 4KB address boundary then behavior is CONSTRAINED UNPREDICTABLE and *Crossing a 4KB boundary with a Device access* on page K1-7841 describes the permitted behaviors.

  Note

  - The boundary referred to is between two Device memory regions that are both of 4KB and aligned to 4KB.
  - This restriction means it is important that an access to a volatile memory device is not made using a single instruction that crosses a 4KB address boundary.
  - Arm expects this restriction to constrain the placing of volatile memory devices in the system memory map, rather than expecting a compiler to be aware of the alignment of memory accesses.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

E2.9 Mismatched memory attributes

In the Armv8 architecture mismatched memory attributes are controlled by privileged software. For more information, see Chapter G5 *The AArch32 Virtual Memory System Architecture*.

Physical memory Locations are accessed with *mismatched attributes* if all accesses to the Location do not use a common definition of all of the following attributes of that Location:

- Memory type, Device or Normal.
- Shareability.
- Cacheability, for the same level of the inner or outer cache, but excluding any cache allocation hints.

Collectively these are referred to as memory attributes.

Note

In this document, the terms *location* and *memory location* refer to any byte within the current coherency granule and are used interchangeably.

When a memory Location is accessed with mismatched attributes the only software visible effects are one or more of the following:

- Uniprocessor semantics for reads and writes to that memory Location might be lost. This means:
  - A read of the memory Location by one agent might not return the value most recently written to that memory Location by the same agent.
  - Multiple writes to the memory Location by one agent with different memory attributes might not be ordered in program order.
- There might be a loss of coherency when multiple agents attempt to access a memory Location.
- There might be a loss of properties derived from the memory type, as described in later bullets in this section.
- If all Load-Exclusive/Store-Exclusive instructions executed across all threads to access a given memory Location do not use consistent memory attributes, the Exclusives monitor state becomes UNKNOWN.
- Bytes written without the Write-Back cacheable attribute within the same Write-Back granule as bytes written with the Write-Back cacheable attribute might have their values reverted to the old values as a result of cache Write-Back.

The loss of properties associated with mismatched memory type attributes refers only to the following properties of Device memory that are additional to the properties of Normal memory:

- Prohibition of speculative read accesses.
- Prohibition on Gathering.
- Prohibition on Re-ordering.

For the following situations, when a physical memory Location is accessed with mismatched attributes, a more restrictive set of behaviors applies. The description of each situation also describes the behaviors that apply:

1. Any agent that reads that memory Location using the same common definition of the Memory type, Shareability and Cacheability attributes is guaranteed to access it coherently, to the extent required by that common definition of the memory attributes, only if all the following conditions are met:
   - All writes are performed to an alias of the memory Location that uses the same definition of the Memory type, Shareability and Cacheability attributes.
   - Either:
     - In the Non-secure PL1&0 translation regime, HCR2.MIOCNCE has a value of 0.
     - All aliases with write permission have the Inner Cacheability attribute the same as the Outer Cacheability attribute.
   - Either:
     - All writes are performed to an alias of the memory Location that has Inner Cacheability and Outer Cacheability attributes both as Non-cacheable.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— All aliases to a memory Location use a definition of the Shareability attributes that encompasses all the agents with permission to access the Location.

2. The possible software-visible effects caused by mismatched attributes for a memory Location are defined more precisely if all of the mismatched attributes define the memory Location as one of:
   - Any Device memory type.
   - Normal Inner Non-cacheable, Outer Non-cacheable memory.

   In these cases, the only permitted software-visible effects of the mismatched attributes are one or more of the following:
   - Possible loss of properties derived from the memory type when multiple agents attempt to access the memory Location.
   - Possible reordering of memory transactions to the same memory Location with different memory attributes, potentially leading to a loss of coherency or uniprocessor semantics. Any possible loss of coherency or uniprocessor semantics can be avoided by inserting `DMB` barrier instructions between accesses to the same memory Location that might use different attributes.

Where there is a loss of the uniprocessor semantics, ordering, or coherency, the following approaches can be used:

1. If the mismatched attributes for a memory Location all assign the same Shareability attribute to a Location that has a cacheable attribute, any loss of uniprocessor semantics, ordering, or coherency within a Shareability domain can be avoided by use of software cache management. To do so, software must use the techniques that are required for the software management of the ordering or coherency of cacheable Locations between agents in different shareability domains. This means:
   - Before writing to a cacheable Location not using the Write-Back attribute, software must invalidate, or clean, a Location from the caches if any agent might have written to the Location with the Write-Back attribute. This avoids the possibility of overwriting the Location with stale data.
   - After writing to a cacheable Location with the Write-Back attribute, software must clean the Location from the caches, to make the write visible to external memory.
   - Before reading the Location with a cacheable attribute, software must invalidate, or clean and invalidate, the Location from the caches, to ensure that any value held in the caches reflects the last value made visible in external memory.
   - Executing a `DMB` barrier instruction, with scope that applies to the common Shareability of the accesses, between any accesses to the same cacheable Location that use different attributes.

   Note

   In AArch32 state, cache maintenance instructions can only be accessed from an Exception level that is higher than EL0, and therefore require a system call. For information on system calls, see *Exception-generating and exception-handling instructions* on page F1-4055. For information about the AArch32 cache maintenance instructions, see *AArch32 cache and branch predictor support* on page G4-5889.

   In all cases:
   - Location refers to any byte within the current coherency granule.
   - A clean and invalidate instruction can be used instead of a clean instruction, or instead of an invalidate instruction.
   - In the sequences outlined in this section, all cache maintenance instructions and memory transactions must be completed, or ordered by the use of barrier operations, if they are not naturally ordered by the use of a common address, see *Ordering of cache and branch predictor maintenance instructions* on page G4-5907.

   Note

   With software management of coherency, race conditions can cause loss of data. A race condition occurs when different agents write simultaneously to bytes that are in the same Location, and the invalidate, write, clean sequence of one agent overlaps with the equivalent sequence of another agent. A race condition also occurs if the first operation of either sequence is a clean, rather than an invalidate.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

2. If the mismatched attributes for a Location mean that multiple cacheable accesses to the Location might be made with different Shareability attributes, then ordering and coherency are guaranteed only if:
   - Each PE that accesses the Location with a cacheable attribute performs a clean and invalidate of the Location before and after accessing that Location.
   - A `DMB` barrier with scope that covers the full Shareability of the accesses is placed between any accesses to the same memory Location that use different attributes.

   Note

   The Note in rule 1 of this list, about possible race conditions, also applies to this rule.

In addition, if multiple agents attempt to use Load-Exclusive or Store-Exclusive instructions to access a Location, and the accesses from the different agents have different memory attributes associated with the Location, the Exclusives monitor state becomes UNKNOWN.

Arm strongly recommends that software does not use mismatched attributes for aliases of the same Location. An implementation might not optimize the performance of a system that uses mismatched aliases.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

E2.10 Synchronization and semaphores

Armv8 provides non-blocking synchronization of shared memory, using *synchronization primitives*. The information in this section about memory accesses by synchronization primitives applies to accesses to both Normal and Device memory.

Note

Use of the Armv8 synchronization primitives scales for multiprocessing system designs.

Table E2-6 shows the synchronization primitives and the associated CLREX instruction.

Table E2-6 Synchronization primitives and associated instruction, T32 and A32 instruction sets

Transaction size	Additional semantics	Load-Exclusive[a]	Store-Exclusive[a]	Other[a]
Byte	-	LDREXB	STREXB	-
	Load-Acquire/Store-Release	LDAEXB	STLEXB	-
Halfword	-	LDREXH	STREXH	-
	Load-Acquire/Store-Release	LDAEXH	STLEXH	-
Word	-	LDREX	STREX	-
	Load-Acquire/Store-Release	LDAEX	STLEX	-
Doubleword	-	LDREXD	STREXD	-
	Load-Acquire/Store-Release	LDAEXD	STLEXD	-
None	Clear-Exclusive	-	-	CLREX

a. Instruction in the T32 and A32 instruction sets.

Except for the row showing the CLREX instruction, the two instructions in a single row are a Load-Exclusive/Store-Exclusive instruction pair. The model for the use of a Load-Exclusive/Store-Exclusive instruction pair accessing a non-aborting memory address *x* is:

- The Load-Exclusive instruction reads a value from memory address *x*.
- The corresponding Store-Exclusive instruction succeeds in writing back to memory address *x* only if no other observer, process, or thread has performed a more recent store to address *x*. The Store-Exclusive instruction returns a status bit that indicates whether the memory write succeeded.

A Load-Exclusive instruction marks a small block of memory for exclusive access. The size of the marked block is IMPLEMENTATION DEFINED, see *Marking and the size of the marked memory block* on page E2-4029. A Store-Exclusive instruction to any address in the marked block clears the marking.

Note

In this section, the term PE includes any observer that can generate a Load-Exclusive or a Store-Exclusive instruction.

The following sections give more information:

- *Exclusive access instructions and Non-shareable memory locations* on page E2-4024.
- *Exclusive access instructions and shareable memory locations* on page E2-4025.
- *Marking and the size of the marked memory block* on page E2-4029.
- *Context switch support* on page E2-4029.
- *Load-Exclusive and Store-Exclusive instruction usage restrictions* on page E2-4029.
- *Use of WFE and SEV instructions by spin-locks* on page E2-4032.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

E2.10.1 Exclusive access instructions and Non-shareable memory locations

For memory locations for which the Shareability attribute is Non-shareable, the exclusive access instructions rely on a *local Exclusives monitor*, or *local monitor*, that marks any address from which the PE executes a Load-Exclusive instruction. Any non-aborted attempt by the same PE to use a Store-Exclusive instruction to modify any address is guaranteed to clear the marking.

A Load-Exclusive instruction performs a load from memory, and:

- The executing PE marks the physical memory address for exclusive access.
- The local monitor of the executing PE transitions to the Exclusive Access state.

A Store-Exclusive instruction performs a conditional store to memory that depends on the state of the local monitor:

If the local monitor is in the Exclusive Access state

- If the address of the Store-Exclusive instruction is the same as the address that has been marked in the monitor by an earlier Load-Exclusive instruction, then the store occurs. Otherwise, it is IMPLEMENTATION DEFINED whether the store occurs.
- A status value is returned to a register:
  - If the store took place the status value is 0.
  - Otherwise, the status value is 1.
- The local monitor of the executing PE transitions to the Open Access state.

When an Exclusives monitor is in the Exclusive Access state the monitor is *set*.

If the local monitor is in the Open Access state

- No store takes place.
- A status value of 1 is returned to a register.
- The local monitor remains in the Open Access state.

When an Exclusives monitor is in the Exclusive Access state the monitor is *clear*.

The Store-Exclusive instruction defines the register to which the status value is returned.

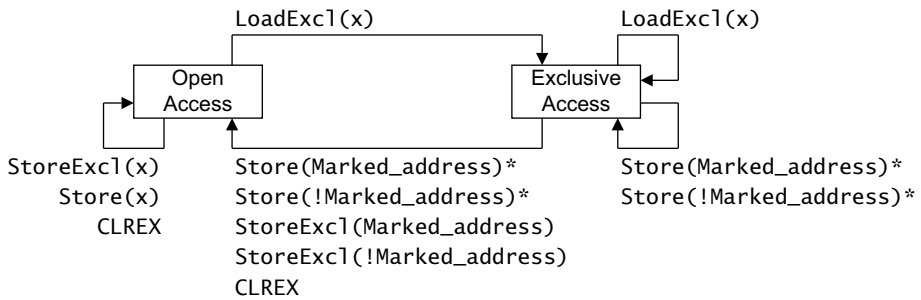
When a PE writes using any instruction other than a Store-Exclusive instruction:

- If the write is to a PA that is not marked as Exclusive Access by its local monitor and that local monitor is in the Exclusive Access state it is IMPLEMENTATION DEFINED whether the write affects the state of the local monitor.
- If the write is to a PA that is marked as Exclusive Access by its local monitor it is IMPLEMENTATION DEFINED whether the write affects the state of the local monitor.

It is IMPLEMENTATION DEFINED whether a store to a marked PA causes a mark in the local monitor to be cleared if that store is by an observer other than the one that caused the PA to be marked.

Figure E2-4 on page E2-4025 shows the state machine for the local monitor and the effect of each of the operations shown in the figure.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta



Operations marked * are possible alternative IMPLEMENTATION DEFINED options.
In the diagram: `LoadExcl` represents any Load-Exclusive instruction
`StoreExcl` represents any Store-Exclusive instruction
`Store` represents any other store instruction.

Any `LoadExcl` operation updates the marked address to the most significant bits of the address x used for the operation.

Figure E2-4 Local monitor state machine diagram

For more information about marking see *Marking and the size of the marked memory block* on page E2-4029.

Note

For the local monitor state machine, as shown in Figure E2-4:

- The IMPLEMENTATION DEFINED options for the local monitor are consistent with the local monitor being constructed so that it does not hold any PA, but instead treats any access as matching the address of the previous Load-Exclusive instruction.
- A local monitor implementation can be unaware of Load-Exclusive and Store-Exclusive instructions from other PEs.
- The architecture does not require a load instruction, by another PE, that is not a Load-Exclusive instruction, to have any effect on the local monitor.
- It is IMPLEMENTATION DEFINED whether the transition from Exclusive Access to Open Access state occurs when the `Store` or `StoreExcl` is from another observer.

Changes to the local monitor state resulting from speculative execution

The architecture permits a local monitor to transition to the Open Access state as a result of speculation, or from some other cause. This is in addition to the transitions to Open Access state caused by the architectural execution of an operation shown in Figure E2-4.

An implementation must ensure that:

- The local monitor cannot be seen to transition to the Exclusive Access state except as a result of the architectural execution of one of the operations shown in Figure E2-4.
- Any transition of the local monitor to the Open Access state not caused by the architectural execution of an operation shown in Figure E2-4 must not indefinitely delay forward progress of execution.

E2.10.2 Exclusive access instructions and shareable memory locations

In the context of this section, a shareable memory location is a memory location that has, or is treated as if it has, a Shareability attribute of Inner Shareable or Outer Shareable.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For shareable memory locations, exclusive access instructions rely on:

- A *local monitor* for each PE in the system, that marks any address from which the PE executes a Load-Exclusive. The local monitor operates as described in *Exclusive access instructions and Non-shareable memory locations* on page E2-4024, except that for shareable memory any Store-Exclusive is then subject to checking by the global monitor if it is described in that section as doing at least one of the following:
  - Updating memory.
  - Returning a status value of 0.

  The local monitor can ignore accesses from other PEs in the system.

- A *global monitor* that marks a PA as exclusive access for a particular PE. This marking is used later to determine whether a Store-Exclusive to that address that has not been failed by the local monitor can occur. Any successful write to the marked block by any other observer in the Shareability domain of the memory location is guaranteed to clear the marking. For each PE in the system, the global monitor:
  - Can hold at least one marked block.
  - Maintains a state machine for each marked block it can hold.

  Note

  For each PE, the architecture only requires global monitor support for a single marked address. Any situation that might benefit from the use of multiple marked addresses on a single PE is CONSTRAINED UNPREDICTABLE, see *Load-Exclusive and Store-Exclusive instruction usage restrictions* on page E2-4029.

Note

The global monitor can either reside in a block that is part of the hardware on which the PE executes or exist as a secondary monitor at the memory interfaces. The IMPLEMENTATION DEFINED aspects of the monitors mean that the global monitor and local monitor can be combined into a single unit, provided that the unit performs the global monitor and local monitor functions defined in this manual.

For shareable memory locations, in some implementations and for some memory types, the properties of the global monitor require functionality outside the PE. Some system implementations might not implement this functionality for all locations of memory. In particular, this can apply to:

- Any type of memory in the system implementation that does not support hardware cache coherency.
- Non-cacheable memory, or memory treated as Non-cacheable, in an implementation that does support hardware cache coherency.

In such a system, it is defined by the system:

- Whether the global monitor is implemented.
- If the global monitor is implemented, which address ranges or memory types it monitors.

Note

To support the use of the Load-Exclusive/Store-Exclusive mechanism when address translation is disabled, a system might define at least one location of memory, of at least the size of the translation granule, in the system memory map to support the global monitor for all PEs within a common Inner Shareable domain. However, this is not an architectural requirement. Therefore, architecturally-compliant software that requires mutual exclusion must not rely on using the Load-Exclusive/Store-Exclusive mechanism, and must instead use a software algorithm such as Lamport's Bakery algorithm to achieve mutual exclusion.

Because implementations can choose which memory types are treated as Non-cacheable, the only memory types for which it is architecturally guaranteed that a global Exclusives monitor is implemented are:

- Inner Shareable, Inner Write-Back, Outer Write-Back Normal memory with Read allocation hint and Write allocation hint and not transient.

- Outer Shareable, Inner Write-Back, Outer Write-Back Normal memory with Read allocation hint and Write allocation hints and not transient.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the global monitor is not implemented for an address range or memory type, then performing a Load-Exclusive/Store-Exclusive instruction to such a location has one or more of the following effects:

- The instruction generates an External abort.
- The instruction generates an IMPLEMENTATION DEFINED MMU fault. This is reported using the Fault status code of:
  - DFSR.STATUS = `0b110101` when using the Long-descriptor translation table format. The fault can also be reported in the HSR.ISS[5:0] field for exceptions to Hyp mode.
  - DFSR.FS = `0b10101` when using the Short-descriptor translation table format.

  If the IMPLEMENTATION DEFINED MMU fault is generated for the Non-secure PL1&0 translation regime then:
  - If the fault is generated because of the memory type defined in the first stage of translation, or if the second stage of translation is disabled, then this is a first stage fault and the exception is taken to EL1.
  - Otherwise, the fault is a second stage fault and the exception is taken to EL2.

  The priority of this fault is IMPLEMENTATION DEFINED.
- The instruction is treated as a `NOP`.
- The Load-Exclusive instruction is treated as if it were accessing a Non-shareable location, but the state of the local monitor becomes UNKNOWN.
- The Store-Exclusive instruction is treated as if it were accessing a Non-shareable location, but the state of the local monitor becomes UNKNOWN.
- The value held in the result register of the Store-Exclusive instruction becomes UNKNOWN.

In addition, for write transactions generated by non-PE observers that do not implement exclusive accesses or other atomic access mechanisms, the effect that writes have on the global and local monitors used by an Arm PE is IMPLEMENTATION DEFINED. The writes might not clear the global monitors of other PEs for:

- Some address ranges.
- Some memory types.

Operation of the global Exclusives monitor

A Load-Exclusive instruction from shareable memory performs a load from memory, and causes the PA of the access to be marked as exclusive access for the requesting PE. This access can also cause the exclusive access mark to be removed from any other PA that has been marked by the requesting PE.

Note

The global monitor only supports a single outstanding exclusive access to shareable memory for each PE.

A Load-Exclusive instruction by one PE has no effect on the global monitor state for any other PE.

A Store-Exclusive instruction performs a conditional store to memory:

- The store is guaranteed to succeed only if the PA accessed is marked as exclusive access for the requesting PE and both the local monitor and the global monitor state machines for the requesting PE are in the Exclusive Access state. In this case:
  - A status value of 0 is returned to a register to acknowledge the successful store.
  - The final state of the global monitor state machine for the requesting PE is IMPLEMENTATION DEFINED.
  - If the address accessed is marked for exclusive access in the global monitor state machine for any other PE then that state machine transitions to Open Access state.
- If no address is marked as exclusive access for the requesting PE, the store does not succeed:
  - A status value of 1 is returned to a register to indicate that the store failed.
  - The global monitor is not affected and remains in Open Access state for the requesting PE.
- If a different PA is marked as exclusive access for the requesting PE, it is IMPLEMENTATION DEFINED whether the store succeeds or not:
  - If the store succeeds a status value of 0 is returned to a register, otherwise a value of 1 is returned.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— If the global monitor state machine for the PE was in the Exclusive Access state before the Store-Exclusive instruction it is IMPLEMENTATION DEFINED whether that state machine transitions to the Open Access state.

The Store-Exclusive instruction defines the register to which the status value is returned.

In a shared memory system, the global monitor implements a separate state machine for each PE in the system. The state machine for accesses to shareable memory by PE(n) can respond to all the shareable memory accesses visible to it. This means it responds to:

- Accesses generated by PE(n).
- Accesses generated by the other observers in the Shareability domain of the memory location. These accesses are identified as (!n).

In a shared memory system, the global monitor implements a separate state machine for each observer that can generate a Load-Exclusive or a Store-Exclusive instruction in the system.

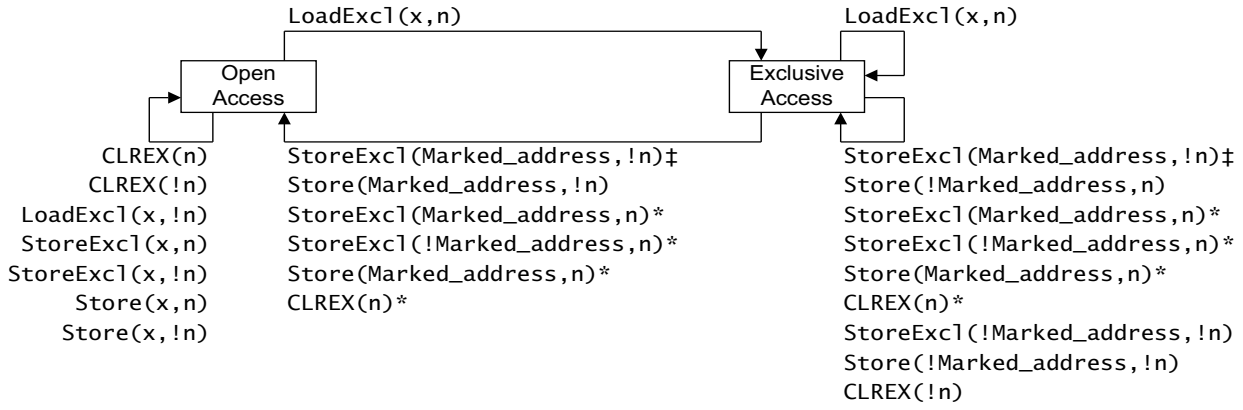
A global monitor:

- In the Exclusive Access state is *set*.
- In the Open Access state is *clear*.

Clear global monitor event

Whenever the global monitor state for a PE changes from Exclusive access to Open access, an event is generated and held in the Event register for that PE. This register is used by the Wait for Event mechanism, see *Wait For Event and Send Event* on page G1-5764.

Figure E2-5 shows the state machine for PE(n) in a global monitor.



‡`StoreExcl`(Marked_address,!n) clears the monitor only if the `StoreExcl` updates memory
Operations marked * are possible alternative IMPLEMENTATION DEFINED options.
In the diagram: `LoadExcl` represents any Load-Exclusive instruction
`StoreExcl` represents any Store-Exclusive instruction
`Store` represents any other store instruction.

Any `LoadExcl` operation updates the marked address to the most significant bits of the address x used for the operation.

Figure E2-5 Global monitor state machine diagram for PE(n) in a multiprocessor system

For more information about marking see *Marking and the size of the marked memory block* on page E2-4029.

Note

For the global monitor state machine, as shown in Figure E2-5:

- The architecture does not require a load instruction by another PE, that is not a Load-Exclusive instruction, to have any effect on the global monitor.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Whether a Store-Exclusive instruction successfully updates memory or not depends on whether the address accessed matches the marked shareable memory address for the PE issuing the Store-Exclusive instruction, and whether the local and global monitors are in the exclusive state. For this reason, Figure E2-5 on page E2-4028 only shows how the operations by (!n) cause state transitions of the state machine for PE(n).
- A Load-Exclusive instruction can only update the marked shareable memory address for the PE issuing the Load-Exclusive instruction.
- When the global monitor is in the Exclusive Access state, it is IMPLEMENTATION DEFINED whether a `CLREX` instruction causes the global monitor to transition from Exclusive Access to Open Access state.
- It is IMPLEMENTATION DEFINED:
  - Whether a modification to a Non-shareable memory location can cause a global monitor to transition from Exclusive Access to Open Access state.
  - Whether a Load-Exclusive instruction to a Non-shareable memory location can cause a global monitor to transition from Open Access to Exclusive Access state.

E2.10.3 Marking and the size of the marked memory block

When a Load-Exclusive instruction is executed, the resulting marked block ignores the least significant bits of the 64-bit memory address.

When a Load-Exclusive instruction is executed, a marked block of size 2^a bytes is created by ignoring the least significant bits of the memory address. A marked address is any address within this marked block. The size of the marked memory block is called the *Exclusives reservation granule*. The Exclusives reservation granule is IMPLEMENTATION DEFINED in the range 4 - 512 words.

Note

This definition means that the Exclusives reservation granule is:

- 4 words in an implementation where *a* is 4.
- 512 words in an implementation where *a* is 11.

For example, in an implementation where *a* is 4, a successful `LDREXB` of address `0x341B4` defines a marked block using bits[47:4] of the address. This means that the four words of memory from `0x341B0` to `0x341BF` are marked for exclusive access.

In some implementations the CTR identifies the Exclusives reservation granule, see CTR. Otherwise, software must assume that the maximum Exclusives reservation granule, 512 words, is implemented.

E2.10.4 Context switch support

An exception return clears the local monitor. As a result, performing a `CLREX` instruction as part of a context switch is not required in most situations.

Note

Context switching is not an application level operation. However, this information is included here to complete the description of the exclusive operations.

E2.10.5 Load-Exclusive and Store-Exclusive instruction usage restrictions

The Load-Exclusive and Store-Exclusive instructions are intended to work together as a pair, for example a `LDREX/STREX` pair or a `LDREXB/STREXB` pair. To support different implementations of these functions, software must follow the notes and restrictions given in this subsection.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following notes describe use of a Load-Exclusive/ Store-Exclusive instruction pair, `LoadExcl`/`StoreExcl`, to indicate the use of any of the Load-Exclusive/Store-Exclusive instruction pairs shown in Table E2-6 on page E2-4023. In this context, a `LoadExcl`/`StoreExcl` pair comprises two instructions in the same thread of execution:

- The exclusives support a single outstanding exclusive access for each PE thread that is executed. The architecture makes use of this by not requiring an address or size check as part of the `IsExclusiveLocal()` function. If the target VA of a `StoreExcl` is different from the VA of the preceding `LoadExcl` instruction in the same thread of execution, behavior can be CONSTRAINED UNPREDICTABLE with the following behavior:
  - The `StoreExcl` either passes or fails, the status value returned by the `StoreExcl` is UNKNOWN, and the states of the local and global monitors for that PE are UNKNOWN.

    Note

    This means the `StoreExcl` might pass for some instances of a `LoadExcl`/`StoreExcl` pair with mismatched addresses, and fail for other instances of a `LoadExcl`/`StoreExcl` pair with mismatched addresses.

  - The data at the address accessed by the `LoadExcl`, and at the address accessed by the `StoreExcl`, is UNKNOWN.

  This means software can rely on a `LoadExcl`/`StoreExcl` pair to eventually succeed only if the `LoadExcl` and the `StoreExcl` are executed with the same VA.

- An implementation of the Load-Exclusive and Store-Exclusive instructions can require that, in any thread of execution, the transaction size of a `StoreExcl` instruction is the same as the transaction size of the preceding `LoadExcl` instruction executed in that thread. If the transaction size of a `StoreExcl` instruction is different from the preceding `LoadExcl` instruction in the same thread of execution, behavior can be CONSTRAINED UNPREDICTABLE with the following behavior:
  - The `StoreExcl` either passes or fails, and the status value returned by the `StoreExcl` is UNKNOWN.

    Note

    This means the `StoreExcl` might pass for some instances of a `LoadExcl`/`StoreExcl` pair with mismatched transaction sizes, and fail for other instances of a `LoadExcl`/`StoreExcl` pair with mismatched transaction sizes.

  - The block of data of the size of the larger of the transaction sizes used by the `LoadExcl`/`StoreExcl` pair at the address accessed by the `LoadExcl`/`StoreExcl` pair, is UNKNOWN.

  This means software can rely on a `LoadExcl`/`StoreExcl` pair to eventually succeed only if the `LoadExcl` and the `StoreExcl` have the same transaction size.

- `LoadExcl`/`StoreExcl` loops are guaranteed to make forward progress only if, for any `LoadExcl`/`StoreExcl` loop within a single thread of execution, the software meets all of the following conditions:

  1 Between the Load-Exclusive and the Store-Exclusive, there are no explicit memory accesses, preloads, direct or indirect System register writes, address translation instructions, cache or TLB maintenance instructions, exception generating instructions, exception returns, or indirect branches.

  2 Between the Store-Exclusive returning a failing result and the retry of the corresponding Load-Exclusive:
    - There are no stores or `PLDW` instructions to any address within the Exclusives reservation granule accessed by the Store-Exclusive.
    - There are no loads or preloads to any address within the Exclusives reservation granule accessed by the Store-Exclusive that use a different VA alias to that address.
    - There are no direct or indirect System register writes, other than changes to the flag fields in the CPSR or FPSCR, caused by data processing or comparison instructions.
    - There are no direct or indirect address translation instructions, cache or TLB maintenance instructions, exception generating instructions, exception returns, or indirect branches.
    - All loads and stores are to a block of contiguous virtual memory of not more than 512 bytes in size.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The Exclusives monitor can be cleared at any time without an application-related cause, provided that such clearing is not systematically repeated so as to prevent the forward progress in finite time of at least one of the threads that is accessing the Exclusives monitor. However, it is permissible for the `LoadExcl`/`StoreExcl` loop not to make forward progress if a different thread is repeatedly doing any of the following in a tight loop:

— Performing stores to a PA covered by the Exclusives monitor.

— Prefetching with intent to write to a PA covered by the Exclusives monitor.

— Executing data cache clean, data cache invalidate, or data cache clean and invalidate instructions to a PA covered by the Exclusives monitor.

— Executing instruction cache invalidate all instructions.

— Executing instruction cache invalidate by VA instructions to a PA covered by the Exclusives monitor.

- Implementations can benefit from keeping the `LoadExcl` and `StoreExcl` operations close together in a single thread of execution. This minimizes the likelihood of the Exclusives monitor state being cleared between the `LoadExcl` instruction and the `StoreExcl` instruction. Therefore, for best performance, Arm strongly recommends a limit of 128 bytes between `LoadExcl` and `StoreExcl` instructions in a single thread of execution.

- The architecture sets an upper limit of 2048 bytes on the Exclusives reservation granule that can be marked as exclusive. For performance reasons, Arm recommends that objects that are accessed by exclusive accesses are separated by the size of the Exclusives reservation granule. This is a performance guideline rather than a functional requirement.

- After taking a Data Abort exception, the state of the Exclusives monitors is UNKNOWN.

- For the memory location accessed by a `LoadExcl`/`StoreExcl` pair, if the memory attributes for a `StoreExcl` instruction are different from the memory attributes for the preceding `LoadExcl` instruction in the same thread of execution, behavior is CONSTRAINED UNPREDICTABLE. Where this occurs because the translation of the accessed address changes between the `LoadExcl` instruction and the `StoreExcl` instruction, the CONSTRAINED UNPREDICTABLE behavior is as follows:

  — The `StoreExcl` either passes or fails, and the status value returned by the `StoreExcl` is UNKNOWN.

  Note

  This means the `StoreExcl` might pass for some instances of a `LoadExcl`/`StoreExcl` pair with changed memory attributes, and fail for other instances of a `LoadExcl`/`StoreExcl` pair with changed memory attributes.

  — The data at the address accessed by the `StoreExcl` is UNKNOWN.

  Note

  Another bullet point in this list covers the case where the memory attributes of a `LoadExcl`/`StoreExcl` pair differ as a result of using different VAs with different attributes that point to the same PA.

- The effect of a data or unified cache invalidate, clean, or clean and invalidate instruction on a local or global Exclusives monitor that is in the Exclusive Access state is CONSTRAINED UNPREDICTABLE, and the instruction might clear the monitor, or it might leave it in the Exclusive Access state. For address-based maintenance instructions, this also applies to the monitors of other PEs in the same Shareability domain as the PE executing the cache maintenance instruction, as determined by the Shareability domain of the address being maintained.

  Note

  Arm strongly recommends that implementations ensure that the use of such maintenance instructions by a PE in the Non-secure state cannot cause a denial of service on a PE in the Secure state.

- If the mapping of the VA to PA is changed between the `LoadExcl` instruction and the `StoreExcl` instruction, and the change is performed using a break-before-make sequence as described in *Using break-before-make when updating translation table entries* on page G5-5996, if the `StoreExcl` is performed after another write

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

to the same PA as the `StoreExcl`, and that other write was performed after the old translation was properly invalidated and that invalidation was properly synchronized, then the `StoreExcl` will not pass its monitor check.

Note

Arm expects that, in many implementations, either:

— The TLB invalidation will clear either the local or global monitor.

— The PA will be checked between the `LoadExcl` and `StoreExcl`.

- The Exclusive Access state for an address accessed by a PE can be lost as a result of a `PLDW` instruction to the same PA executed by another PE. This means that a very high rate of repeated `PLDW` accesses to a memory location might impede the forward progress of another PE.

Note

In the event of repeatedly-contending `LoadExcl`/`StoreExcl` instruction sequences from multiple PEs, an implementation must ensure that forward progress is made by at least one PE.

E2.10.6 Use of WFE and SEV instructions by spin-locks

Armv8 provides Wait For Event, Send Event, and Send Event Local instructions, `WFE`, `SEV`, `SEVL`, that can assist with reducing power consumption and bus contention caused by PEs repeatedly attempting to obtain a spin-lock. These instructions can be used at the application level, but a complete understanding of what they do depends on a system level understanding of exceptions. They are described in *Wait For Event and Send Event* on page G1-5764. However, in Armv8, when the global monitor for a PE changes from Exclusive Access state to Open Access state, an event is generated.

Note

This is equivalent to issuing an `SEVL` instruction on the PE for which the monitor state has changed. It removes the need for spinlock code to include an `SEV` instruction after clearing a spinlock.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Part F

The AArch32 Instruction Sets

Chapter F1
The AArch32 Instruction Sets Overview

This chapter describes the T32 and A32 instruction sets. It contains the following sections:

- *Support for instructions in different versions of the Arm architecture* on page F1-4036.
- *Unified Assembler Language* on page F1-4037.
- *Branch instructions* on page F1-4039.
- *Data-processing instructions* on page F1-4040.
- *PSTATE and banked register access instructions* on page F1-4048.
- *Load/store instructions* on page F1-4049.
- *Load/store multiple instructions* on page F1-4052.
- *Miscellaneous instructions* on page F1-4053.
- *Exception-generating and exception-handling instructions* on page F1-4055.
- *System register access instructions* on page F1-4057.
- *Advanced SIMD and floating-point load/store instructions* on page F1-4058.
- *Advanced SIMD and floating-point register transfer instructions* on page F1-4060.
- *Advanced SIMD data-processing instructions* on page F1-4061.
- *Floating-point data-processing instructions* on page F1-4072.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F1.1 Support for instructions in different versions of the Arm architecture

This manual describes the T32 and A32 instruction sets for the Armv8 architecture. Therefore, it indicates how any options or extensions in the Armv8 architecture affect the available instructions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F1.2 Unified Assembler Language

This manual uses the Arm *Unified Assembler Language* (UAL). This assembly language syntax provides a canonical form for all T32 and A32 instructions.

UAL describes the syntax for the mnemonic and the operands of each instruction. In addition, it assumes that instructions and data items can be given labels. It does not specify the syntax to be used for labels, nor what assembler directives and options are available. See your assembler documentation for these details.

Most earlier Arm assembly language mnemonics are still supported as synonyms, as described in the instruction details.

Note

Most earlier T32 assembly language mnemonics are *not* supported.

UAL includes *instruction selection* rules that specify which instruction encoding is selected when more than one can provide the required functionality. For example, both 16-bit and 32-bit encodings exist for an `ADD R0, R1, R2` instruction. The most common instruction selection rule is that, when both a 16-bit encoding and a 32-bit encoding are available, the 16-bit encoding is selected, to optimize code density.

Syntax options exist to override the normal instruction selection rules and ensure that a particular encoding is selected. These are useful when disassembling code, to ensure that subsequent assembly produces the original code, and in some other situations.

F1.2.1 Conditional instructions

For maximum portability of UAL assembly language between the T32 and A32 instruction sets, Arm recommends that:

- `IT` instructions are written before conditional instructions in the correct way for the T32 instruction set.
- When assembling to the A32 instruction set, assemblers check that any `IT` instructions are correct, but do not generate any code for them.

Although other T32 instructions are unconditional, all instructions that are made conditional by an `IT` instruction must be written with a condition. These conditions must match the conditions imposed by the `IT` instruction. For example, an `ITTEE EQ` instruction imposes the `EQ` condition on the first two following instructions, and the `NE` condition on the next two. Those four instructions must be written with `EQ`, `EQ`, `NE` and `NE` conditions respectively.

Some instructions cannot be made conditional by an `IT` instruction. Some instructions can be conditional if they are the last instruction in the IT block, but not otherwise.

The branch instruction encodings that include a Condition code field cannot be made conditional by an `IT` instruction. If the assembler syntax indicates a conditional branch that correctly matches a preceding `IT` instruction, it is assembled using a branch instruction encoding that does not include a Condition code field.

Note

For performance reasons Armv8 deprecates many uses of `IT`, see *Partial deprecation of IT* on page F1-4054. As described in that section, an implementation can include ITD controls that disable those uses of `IT`, making them UNDEFINED.

F1.2.2 Use of labels in UAL instruction syntax

The UAL syntax for some instructions includes the label of an instruction or a literal data item that is at a fixed offset from the instruction being specified. The assembler must:

1. Calculate the `PC` or `Align(PC, 4)` value of the instruction. The `PC` value of an instruction is its address plus 4 for a T32 instruction, or plus 8 for an A32 instruction. The `Align(PC, 4)` value of an instruction is its `PC` value ANDed with `0xFFFFFFFC` to force it to be word-aligned. There is no difference between the `PC` and `Align(PC, 4)` values for an A32 instruction, but there can be for a T32 instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

2. Calculate the offset from the `PC` or `Align(PC, 4)` value of the instruction to the address of the labeled instruction or literal data item.

3. Assemble a *PC-relative* encoding of the instruction, that is, one that reads its `PC` or `Align(PC, 4)` value and adds the calculated offset to form the required address.

   Note

   For instructions that can encode a subtraction operation, if the instruction cannot encode the calculated offset but can encode minus the calculated offset, the instruction encoding specifies a subtraction of minus the calculated offset.

The syntax of the following instructions includes a label:

- `B`, `BL`, and `BLX` (immediate). The assembler syntax for these instructions always specifies the label of the instruction that they branch to. Their encodings specify a sign-extended immediate offset that is added to the `PC` value of the instruction to form the target address of the branch.

- `CBNZ` and `CBZ`. The assembler syntax for these instructions always specifies the label of the instruction that they branch to. Their encodings specify a zero-extended immediate offset that is added to the `PC` value of the instruction to form the target address of the branch. They do not support backward branches.

- `LDR`, `LDRB`, `LDRD`, `LDRH`, `LDRSB`, `LDRSH`, `PLD`, `PLDW`, `PLI`, and `VLDR`. The normal assembler syntax of these load instructions can specify the label of a literal data item that is to be loaded. The encodings of these instructions specify a zero-extended immediate offset that is either added to or subtracted from the `Align(PC, 4)` value of the instruction to form the address of the data item. A few such encodings perform a fixed addition or a fixed subtraction and must only be used when that operation is required, but most contain a bit that specifies whether the offset is to be added or subtracted.

  When the assembler calculates an offset of 0 for the normal syntax of these instructions, it must assemble an encoding that adds 0 to the `Align(PC, 4)` value of the instruction. Encodings that subtract 0 from the `Align(PC, 4)` value cannot be specified by the normal syntax.

  There is an alternative syntax for these instructions that specifies the addition or subtraction and the immediate offset explicitly. In this syntax, the label is replaced by `[PC, #+/-<imm>]`, where:

  `+/-` Is `+` or omitted to specify that the immediate offset is to be added to the `Align(PC, 4)` value, or `-` if it is to be subtracted.

  `<imm>` Is the immediate offset.

  This alternative syntax makes it possible to assemble the encodings that subtract 0 from the `Align(PC, 4)` value, and to disassemble them to a syntax that can be re-assembled correctly.

- `ADR`. The normal assembler syntax for this instruction can specify the label of an instruction or literal data item whose address is to be calculated. Its encoding specifies a zero-extended immediate offset that is either added to or subtracted from the `Align(PC, 4)` value of the instruction to form the address of the data item, and some opcode bits that determine whether it is an addition or subtraction.

  When the assembler calculates an offset of 0 for the normal syntax of this instruction, it must assemble the encoding that adds 0 to the `Align(PC, 4)` value of the instruction. The encoding that subtracts 0 from the `Align(PC, 4)` value cannot be specified by the normal syntax.

  There is an alternative syntax for this instruction that specifies the addition or subtraction and the immediate value explicitly, by writing them as additions `ADD <Rd>, PC, #<imm>` or subtractions `SUB <Rd>, PC, #<imm>`. This alternative syntax makes it possible to assemble the encoding that subtracts 0 from the `Align(PC, 4)` value, and to disassemble it to a syntax that can be re-assembled correctly.

Note

Arm recommends that where possible, software avoids using:

- The alternative syntax for the `ADR`, `LDC`, `LDR`, `LDRB`, `LDRD`, `LDRH`, `LDRSB`, `LDRSH`, `PLD`, `PLI`, `PLDW`, and `VLDR` instructions.
- The encodings of these instructions that subtract 0 from the `Align(PC, 4)` value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F1.3 Branch instructions

Table F1-1 summarizes the branch instructions in the T32 and A32 instruction sets. In addition to providing for changes in the flow of execution, some branch instructions can change instruction set.

Table F1-1 Branch instructions

Instruction	See	Range, T32	Range, A32
Branch to target address	*B* on page F5-4291	±16MB	±32MB
Compare and Branch on Nonzero, Compare and Branch on Zero	*CBNZ, CBZ* on page F5-4316	0-126 bytes	[a]
Call a subroutine Call a subroutine, change instruction set[b]	*BL, BLX (immediate)* on page F5-4309	±16MB ±16MB	±32MB ±32MB
Call a subroutine, optionally change instruction set	*BLX (register)* on page F5-4311	Any	Any
Branch to target address, change instruction set	*BX* on page F5-4313	Any	Any
Change to Jazelle state	*BXJ* on page F5-4315	-	-
Table Branch (byte offsets) Table Branch (halfword offsets)	*TBB, TBH* on page F5-4861	0-510 bytes 0-131070 bytes	[a]

a. These instructions do not exist in the A32 instruction set.

b. The range is determined by the instruction set of the `BLX` instruction, not of the instruction it branches to.

Branches to loaded and calculated addresses can be performed by `LDR`, `LDM` and data-processing instructions. For details see *Load/store instructions* on page F1-4049, *Load/store multiple instructions* on page F1-4052, *Standard data-processing instructions* on page F1-4040, and *Shift instructions* on page F1-4042.

In addition to the branch instructions shown in Table F1-1:

- In the A32 instruction set, a data-processing instruction that targets the PC behaves as a branch instruction. For more information, see *Data-processing instructions* on page F1-4040.
- In the T32 and A32 instruction sets, a load instruction that targets the PC behaves as a branch instruction. For more information, see *Load/store instructions* on page F1-4049.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F1.4 Data-processing instructions

Core data-processing instructions belong to one of the following groups:

- *Standard data-processing instructions*.

  These instructions perform basic data-processing operations, and share a common format with some variations.
- *Shift instructions* on page F1-4042.
- *Multiply instructions* on page F1-4042.
- *Saturating instructions* on page F1-4044.
- *Saturating addition and subtraction instructions* on page F1-4044.
- *Packing and unpacking instructions* on page F1-4045.
- *Parallel addition and subtraction instructions* on page F1-4046.
- *Divide instructions* on page F1-4047.
- *Miscellaneous data-processing instructions* on page F1-4047.

For related Advanced SIMD and floating-point instructions see *Advanced SIMD data-processing instructions* on page F1-4061 and *Floating-point data-processing instructions* on page F1-4072.

F1.4.1 Standard data-processing instructions

These instructions generally have a destination register Rd, a first operand register Rn, and a second operand. The second operand can be another register Rm, or an immediate constant.

If the second operand is an immediate constant, it can be:

- Encoded directly in the instruction.
- A *modified immediate constant* that uses 12 bits of the instruction to encode a range of constants. T32 and A32 instructions have slightly different ranges of modified immediate constants. For more information, see *Modified immediate constants in T32 instructions* on page F2-4095 and *Modified immediate constants in A32 instructions* on page F2-4096.

If the second operand is another register, it can optionally be shifted in any of the following ways:

`LSL` Logical Shift Left by 1-31 bits.

`LSR` Logical Shift Right by 1-32 bits.

`ASR` Arithmetic Shift Right by 1-32 bits.

`ROR` Rotate Right by 1-31 bits.

`RRX` Rotate Right with Extend. For details, see *Shift and rotate operations* on page E1-3950.

In T32 code, the amount to shift by is always a constant encoded in the instruction. In A32 code, the amount to shift by is either a constant encoded in the instruction, or the value of a register, Rs.

For instructions other than `CMN`, `CMP`, `TEQ`, and `TST`, the result of the data-processing operation is placed in the destination register. In the A32 instruction set, the destination register can be the PC, causing the result to be treated as a branch address. In the T32 instruction set, this is only permitted for some 16-bit forms of the `ADD` and `MOV` instructions.

These instructions can optionally set the Condition flags, according to the result of the operation. If they do not set the flags, existing flag settings from a previous instruction are preserved.

Table F1-2 on page F1-4041 summarizes the main data-processing instructions in the T32 and A32 instruction sets. Generally, each of these instructions is described in three sections in Chapter F2 *About the T32 and A32 Instruction Descriptions*, one section for each of the following:

- `INSTRUCTION` (immediate) where the second operand is a modified immediate constant.
- `INSTRUCTION` (register) where the second operand is a register, or a register shifted by a constant.
- `INSTRUCTION` (register-shifted register) where the second operand is a register shifted by a value obtained from another register. These are only available in the A32 instruction set.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table F1-2 Standard data-processing instructions

Instruction	Mnemonic	Notes
Add with Carry	ADC	-
Add	ADD	T32 instruction set permits use of a modified immediate constant or a zero-extended 12-bit immediate constant.
Form PC-relative Address	ADR	First operand is the PC. Second operand is an immediate constant. T32 instruction set uses a zero-extended 12-bit immediate constant. Operation is an addition or a subtraction.
Bitwise AND	AND	-
Bitwise Bit Clear	BIC	-
Compare Negative	CMN	Sets flags. Like ADD but with no destination register.
Compare	CMP	Sets flags. Like SUB but with no destination register.
Bitwise Exclusive OR	EOR	-
Copy operand to destination	MOV	Has only one operand, with the same options as the second operand in most of these instructions. If the operand is a shifted register, the instruction is an LSL, LSR, ASR, or ROR instruction instead. For details, see *Shift instructions* on page F1-4042. The T32 and A32 instruction sets permit use of a modified immediate constant or a zero-extended 16-bit immediate constant.
Bitwise NOT	MVN	Has only one operand, with the same options as the second operand in most of these instructions.
Bitwise OR NOT	ORN	Not available in the A32 instruction set.
Bitwise OR	ORR	-
Reverse Subtract	RSB	Subtracts first operand from second operand. This permits subtraction from constants and shifted registers.
Reverse Subtract with Carry	RSC	Not available in the T32 instruction set.
Subtract with Carry	SBC	-
Subtract	SUB	T32 instruction set permits use of a modified immediate constant or a zero-extended 12-bit immediate constant.
Test Equivalence	TEQ	Sets flags. Like EOR but with no destination register.
Test	TST	Sets flags. Like AND but with no destination register.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F1.4.2 Shift instructions

Table F1-3 lists the shift instructions in the T32 and A32 instruction sets.

Table F1-3 Shift instructions

Instruction	See
Arithmetic Shift Right	*ASR (immediate)* on page F5-4283 *ASR (register)* on page F5-4285 *ASRS (immediate)* on page F5-4287 *ASRS (register)* on page F5-4289
Logical Shift Left	*LSL (immediate)* on page F5-4489 *LSL (register)* on page F5-4491 *LSLS (immediate)* on page F5-4493 *LSLS (register)* on page F5-4495
Logical Shift Right	*LSR (immediate)* on page F5-4497 *LSR (register)* on page F5-4499 *LSRS (immediate)* on page F5-4501 *LSRS (register)* on page F5-4503
Rotate Right	*ROR (immediate)* on page F5-4627 *ROR (register)* on page F5-4629 *RORS (immediate)* on page F5-4631 *RORS (register)* on page F5-4633
Rotate Right with Extend	*RRX* on page F5-4635 *RRXS* on page F5-4637

In the A32 instruction set only, the destination register of these instructions can be the PC, causing the result to be treated as an address to branch to.

F1.4.3 Multiply instructions

These instructions can operate on signed or unsigned quantities. In some types of operation, the results are the same whether the operands are signed or unsigned.

- Table F1-4 summarizes the multiply instructions where there is no distinction between signed and unsigned quantities.

  The least significant 32 bits of the result are used. More significant bits are discarded.

- Table F1-5 on page F1-4043 summarizes the signed multiply instructions.

- Table F1-6 on page F1-4043 summarizes the unsigned multiply instructions.

Table F1-4 General multiply instructions

Instruction	See	Operation (number of bits)
Multiply Accumulate	*MLA, MLAS* on page F5-4509	32 = 32 + 32 × 32
Multiply and Subtract	*MLS* on page F5-4511	32 = 32 – 32 × 32
Multiply	*MUL, MULS* on page F5-4547	32 = 32 × 32

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table F1-5 Signed multiply instructions

Instruction	See	Operation (number of bits)
Signed Multiply Accumulate (halfwords)	*SMLABB, SMLABT, SMLATB, SMLATT* on page F5-4696	32 = 32 + 16 × 16
Signed Multiply Accumulate Dual	*SMLAD, SMLADX* on page F5-4698	32 = 32 + 16 × 16 + 16 × 16
Signed Multiply Accumulate Long	*SMLAL, SMLALS* on page F5-4700	64 = 64 + 32 × 32
Signed Multiply Accumulate Long (halfwords)	*SMLALBB, SMLALBT, SMLALTB, SMLALTT* on page F5-4702	64 = 64 + 16 × 16
Signed Multiply Accumulate Long Dual	*SMLALD, SMLALDX* on page F5-4705	64 = 64 + 16 × 16 + 16 × 16
Signed Multiply Accumulate (word by halfword)	*SMLAWB, SMLAWT* on page F5-4707	32 = 32 + 32 × 16 [a]
Signed Multiply Subtract Dual	*SMLSD, SMLSDX* on page F5-4709	32 = 32 + 16 × 16 – 16 × 16
Signed Multiply Subtract Long Dual	*SMLSLD, SMLSLDX* on page F5-4711	64 = 64 + 16 × 16 – 16 × 16
Signed Most Significant Word Multiply Accumulate	*SMMLA, SMMLAR* on page F5-4713	32 = 32 + 32 × 32 [b]
Signed Most Significant Word Multiply Subtract	*SMMLS, SMMLSR* on page F5-4715	32 = 32 – 32 × 32 [b]
Signed Most Significant Word Multiply	*SMMUL, SMMULR* on page F5-4717	32 = 32 × 32 [b]
Signed Dual Multiply Add	*SMUAD, SMUADX* on page F5-4719	32 = 16 × 16 + 16 × 16
Signed Multiply (halfwords)	*SMULBB, SMULBT, SMULTB, SMULTT* on page F5-4721	32 = 16 × 16
Signed Multiply Long	*SMULL, SMULLS* on page F5-4723	64 = 32 × 32
Signed Multiply (word by halfword)	*SMULWB, SMULWT* on page F5-4725	32 = 32 × 16 [a]
Signed Dual Multiply Subtract	*SMUSD, SMUSDX* on page F5-4727	32 = 16 × 16 – 16 × 16

a. The most significant 32 bits of the 48-bit product are used. Less significant bits are discarded.

b. The most significant 32 bits of the 64-bit product are used. Less significant bits are discarded.

Table F1-6 Unsigned multiply instructions

Instruction	See	Operation (number of bits)
Unsigned Multiply Accumulate Accumulate Long	*UMAAL* on page F5-4901	64 = 32 + 32 + 32 × 32
Unsigned Multiply Accumulate Long	*UMLAL, UMLALS* on page F5-4903	64 = 64 + 32 × 32
Unsigned Multiply Long	*UMULL, UMULLS* on page F5-4905	64 = 32 × 32

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F1.4.4 Saturating instructions

Table F1-7 lists the saturating instructions in the T32 and A32 instruction sets. For more information, see *Pseudocode description of saturation* on page E1-3951.

Table F1-7 Saturating instructions

Instruction	See	Operation
Signed Saturate	*SSAT* on page F5-4733	Saturates optionally shifted 32-bit value to selected range
Signed Saturate 16	*SSAT16* on page F5-4735	Saturates two 16-bit values to selected range
Unsigned Saturate	*USAT* on page F5-4923	Saturates optionally shifted 32-bit value to selected range
Unsigned Saturate 16	*USAT16* on page F5-4925	Saturates two 16-bit values to selected range

F1.4.5 Saturating addition and subtraction instructions

Table F1-8 lists the saturating addition and subtraction instructions in the T32 and A32 instruction sets. For more information, see *Pseudocode description of saturation* on page E1-3951.

Table F1-8 Saturating addition and subtraction instructions

Instruction	See	Operation
Saturating Add	*QADD* on page F5-4597	Add, saturating result to the 32-bit signed integer range
Saturating Subtract	*QSUB* on page F5-4610	Subtract, saturating result to the 32-bit signed integer range
Saturating Double and Add	*QADD* on page F5-4597	Doubles one value and adds a second value, saturating the doubling and the addition to the 32-bit signed integer range
Saturating Double and Subtract	*QDSUB* on page F5-4606	Doubles one value and subtracts the result from a second value, saturating the doubling and the subtraction to the 32-bit signed integer range

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F1.4.6 Packing and unpacking instructions

Table F1-9 lists the packing and unpacking instructions in the T32 and A32 instruction sets.

Table F1-9 Packing and unpacking instructions

Instruction	See	Operation
Pack Halfword	*PKHBT, PKHTB* on page F5-4571	Combine halfwords
Signed Extend and Add Byte	*SXTAB* on page F5-4849	Extend 8 bits to 32 and add
Signed Extend and Add Byte 16	*SXTAB16* on page F5-4851	Dual extend 8 bits to 16 and add
Signed Extend and Add Halfword	*SXTAH* on page F5-4853	Extend 16 bits to 32 and add
Signed Extend Byte	*SXTB* on page F5-4855	Extend 8 bits to 32
Signed Extend Byte 16	*SXTB16* on page F5-4857	Dual extend 8 bits to 16
Signed Extend Halfword	*SXTH* on page F5-4859	Extend 16 bits to 32
Unsigned Extend and Add Byte	*UXTAB* on page F5-4933	Extend 8 bits to 32 and add
Unsigned Extend and Add Byte 16	*UXTAB16* on page F5-4935	Dual extend 8 bits to 16 and add
Unsigned Extend and Add Halfword	*UXTAH* on page F5-4937	Extend 16 bits to 32 and add
Unsigned Extend Byte	*UXTB* on page F5-4939	Extend 8 bits to 32
Unsigned Extend Byte 16	*UXTB16* on page F5-4941	Dual extend 8 bits to 16
Unsigned Extend Halfword	*UXTH* on page F5-4943	Extend 16 bits to 32

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F1.4.7 Parallel addition and subtraction instructions

These instructions perform additions and subtractions on the values of two registers and write the result to a destination register, treating the register values as sets of two halfwords or four bytes. That is, they perform SIMD additions or subtractions on the general-purpose registers.

These instructions consist of a prefix followed by a main instruction mnemonic. The prefixes are as follows:

`S` Signed arithmetic modulo 2^8 or 2^{16}.

`Q` Signed saturating arithmetic.

`SH` Signed arithmetic, halving the results.

`U` Unsigned arithmetic modulo 2^8 or 2^{16}.

`UQ` Unsigned saturating arithmetic.

`UH` Unsigned arithmetic, halving the results.

The main instruction mnemonics are as follows:

`ADD16` Adds the top halfwords of two operands to form the top halfword of the result, and the bottom halfwords of the same two operands to form the bottom halfword of the result.

`ASX` Exchanges halfwords of the second operand, and then adds top halfwords and subtracts bottom halfwords.

`SAX` Exchanges halfwords of the second operand, and then subtracts top halfwords and adds bottom halfwords.

`SUB16` Subtracts each halfword of the second operand from the corresponding halfword of the first operand to form the corresponding halfword of the result.

`ADD8` Adds each byte of the second operand to the corresponding byte of the first operand to form the corresponding byte of the result.

`SUB8` Subtracts each byte of the second operand from the corresponding byte of the first operand to form the corresponding byte of the result.

The instruction set permits all 36 combinations of prefix and main instruction operand, as Table F1-10 shows.

See also *Advanced SIMD parallel addition and subtraction* on page F1-4062.

Table F1-10 Parallel addition and subtraction instructions

Main instruction	Signed	Saturating	Signed halving	Unsigned	Unsigned saturating	Unsigned halving
`ADD16`, add, two halfwords	`SADD16`	`QADD16`	`SHADD16`	`UADD16`	`UQADD16`	`UHADD16`
`ASX`, add and subtract with exchange	`SASX`	`QASX`	`SHASX`	`UASX`	`UQASX`	`UHASX`
`SAX`, subtract and add with exchange	`SSAX`	`QSAX`	`SHSAX`	`USAX`	`UQSAX`	`UHSAX`
`SUB16`, subtract, two halfwords	`SSUB16`	`QSUB16`	`SHSUB16`	`USUB16`	`UQSUB16`	`UHSUB16`
`ADD8`, add, four bytes	`SADD8`	`QADD8`	`SHADD8`	`UADD8`	`UQADD8`	`UHADD8`
`SUB8`, subtract, four bytes	`SSUB8`	`QSUB8`	`SHSUB8`	`USUB8`	`UQSUB8`	`UHSUB8`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F1.4.8 Divide instructions

In Armv8, signed and unsigned integer divide instructions are included in both the T32 instruction set and the A32 instruction set.

For descriptions of the instructions, see:

- *SDIV* on page F5-4672.
- *UDIV* on page F5-4887.

For the `SDIV` and `UDIV` instructions, division by zero always returns a zero result.

The ID_ISAR0.Divide_instrs field indicates the level of support for these instructions. The field value of `0b0010` indicates that they are implemented in both the T32 and A32 instruction sets.

F1.4.9 Miscellaneous data-processing instructions

Table F1-11 lists the miscellaneous data-processing instructions in the T32 and A32 instruction sets. Immediate values in these instructions are simple binary numbers.

Table F1-11 Miscellaneous data-processing instructions

Instruction	See	Notes
BitField Clear	*BFC* on page F5-4294	-
BitField Insert	*BFI* on page F5-4296	-
Count Leading Zeros	*CLZ* on page F5-4318	-
Move Top	*MOVT* on page F5-4526	Moves 16-bit immediate value to top halfword. Bottom halfword unchanged.
Reverse Bits	*RBIT* on page F5-4616	-
Byte-Reverse Word	*REV* on page F5-4618	-
Byte-Reverse Packed Halfword	*REV16* on page F5-4620	-
Byte-Reverse Signed Halfword	*REVSH* on page F5-4622	-
Signed BitField Extract	*SBFX* on page F5-4670	-
Select Bytes using GE flags	*SEL* on page F5-4674	-
Unsigned BitField Extract	*UBFX* on page F5-4883	-
Unsigned Sum of Absolute Differences	*USAD8* on page F5-4919	-
Unsigned Sum of Absolute Differences and Accumulate	*USADA8* on page F5-4921	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F1.5 PSTATE and banked register access instructions

These instructions transfer PE state information to or from a general-purpose register.

F1.5.1 PSTATE access instructions

PSTATE holds process state information, see *Process state, PSTATE* on page E1-3953. In AArch32 state:

- At EL1 or higher, PSTATE is accessible using the *Current Program Status Register* (CPSR).
- At EL0, a subset of the CPSR is accessible as the *Application Program Status Register* (APSR).
- On taking an exception, the contents of the CPSR are copied to the *Saved Program Status Register* (SPSR) of the mode from which the exception is taken.

The `MRS` and `MSR` instructions move the contents of the CPSR, APSR, or the SPSR of the current mode to or from a general-purpose register, see:

- *MRS* on page F5-4532.
- *MSR (immediate)* on page F5-4542.
- *MSR (register)* on page F5-4544.

When executed at EL0, `MRS` and `MSR` instructions can only access the APSR.

The PSTATE Condition flags, PSTATE.{N, Z, C, V} are set by the execution of data-processing instructions, and can control the execution of conditional instructions. However, software can set the Condition flags explicitly using the `MSR` instruction, and can read the current state of the Condition flags explicitly using the `MRS` instruction.

In addition, at EL1 or higher, software can use the `CPS` instruction to change the PSTATE.M field and the PSTATE.{A, I, F} interrupt mask bits, see *CPS, CPSID, CPSIE* on page F5-4334.

F1.5.2 Banked register access instructions

At EL1 or higher, the `MRS` (banked register) and `MSR` (banked register) instructions move the contents of a banked general-purpose register, the SPSR, or the ELR_hyp, to or from a general-purpose register. See:

- *MRS (Banked register)* on page F5-4534.
- *MSR (Banked register)* on page F5-4538.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F1.6 Load/store instructions

Table F1-12 summarizes the general-purpose register load/store instructions in the T32 and A32 instruction sets. Some of these instructions can also operate on the PC. See also:

- *Load/store multiple instructions* on page F1-4052.
- *Synchronization and semaphores* on page E2-4023, for more information about the Load-Exclusive and Store-Exclusive instructions.
- *Load-Acquire, Store-Release* on page E2-3996, for more information about the Load-Acquire/Store-Release and Load-Acquire Exclusive/Store-Release Exclusive instructions.
- *Advanced SIMD and floating-point load/store instructions* on page F1-4058.

Load/store instructions have several options for addressing memory. For more information, see *Addressing modes* on page F1-4050.

Table F1-12 Load/store instructions

Data type	Load	Store	Unprivileged		Exclusive		Load-Acquire	Store-Release	Exclusive	
			Load	Store	Load	Store			Load-Acquire	Store-Release
32-bit word	LDR	STR	LDRT	STRT	LDREX	STREX	LDA	STL	LDAEX	STLEX
16-bit halfword	-	STRH	-	STRHT	-	STREXH	LDAH	STLH	LDAEXH	STLEXH
16-bit unsigned halfword	LDRH	-	LDRHT	-	LDREXH	-	-	-	-	-
16-bit signed halfword	LDRSH	-	LDRSHT	-	-	-	-	-	-	-
8-bit byte	-	STRB	-	STRBT	-	STREXB	LDAB	STLB	LDAEXB	STLEXB
8-bit unsigned byte	LDRB	-	LDRBT	-	LDREXB	-	-	-	-	-
8-bit signed byte	LDRSB	-	LDRSBT	-	-	-	-	-	-	-
Two 32-bit words	LDRD	STRD	-	-	-	-	-	-	-	-
64-bit doubleword	-	-	-	-	LDREXD	STREXD	-	-	LDAEXD	STLEXD

F1.6.1 Loads to the PC

The `LDR` instruction can load a value into the PC. The value loaded is treated as an interworking address, as described by the `LoadWritePC()` pseudocode function in *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.

F1.6.2 Halfword and byte loads and stores

Halfword and byte stores store the least significant halfword or byte from the register, to 16 or 8 bits of memory respectively. There is no distinction between signed and unsigned stores.

Halfword and byte loads load 16 or 8 bits from memory into the least significant halfword or byte of a register. Unsigned loads zero-extend the loaded value to 32 bits, and signed loads sign-extend the value to 32 bits.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F1.6.3 Load unprivileged and Store unprivileged

When executing at EL0, a Load unprivileged or Store unprivileged instruction operates in exactly the same way as the corresponding ordinary load or store instruction. For example, an `LDRT` instruction executes in exactly the same way as the equivalent `LDR` instruction. When executed at PL1, Load unprivileged and Store unprivileged instructions behave as they would if they were executed at EL0. For example, an `LDRT` instruction executes in exactly the way that the equivalent `LDR` instruction would execute at EL0. In particular, the instructions make unprivileged memory accesses.

Note

As described in *Security state, Exception levels, and AArch32 execution privilege* on page G1-5682, execution at PL1 describes all of the following:

- Execution at Non-secure EL1 using AArch32.
- Execution at Secure EL1 using AArch32 when EL3 is not implemented.
- Execution at Secure EL1 using AArch32 when EL3 is implemented and is using AArch64.
- Execution at Secure EL3 when EL3 is implemented and is using AArch32.

The Load unprivileged and Store unprivileged instructions are CONSTRAINED UNPREDICTABLE if executed at EL2, see *Execution of Load/Store unprivileged instructions in Hyp mode* on page K1-7851.

For more information about execution privilege, see *About access permissions* on page G5-5968.

F1.6.4 Load-Exclusive and Store-Exclusive

Load-Exclusive and Store-Exclusive instructions provide shared memory synchronization. For more information, see *Synchronization and semaphores* on page E2-4023.

F1.6.5 Load-Acquire and Store-Release

Load-Acquire and Store-Release instructions provide memory barriers. Load-Acquire Exclusive and Store-Release Exclusive instructions provide memory barriers with shared memory synchronization. For more information, see *Load-Acquire, Store-Release* on page E2-3996.

F1.6.6 Addressing modes

The address for a load or store is formed from two parts: a value from a base register, and an offset.

The base register can be any one of the general-purpose registers R0-R12, SP, or LR.

For loads, the base register can be the PC. This provides PC-relative addressing for position-independent code. Instructions marked (literal) in their title in Chapter F2 *About the T32 and A32 Instruction Descriptions* are PC-relative loads.

The offset takes one of three formats:

Immediate The offset is an unsigned number that can be added to or subtracted from the base register value. Immediate offset addressing is useful for accessing data elements that are a fixed distance from the start of the data object, such as structure fields, stack offsets, and input/output registers.

Register The offset is a value from a general-purpose register. The value can be added to, or subtracted from, the base register value. Register offsets are useful for accessing arrays or blocks of data.

Scaled register The offset is a general-purpose register, shifted by an immediate value, then added to or subtracted from the base register. This means an array index can be scaled by the size of each array element.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The offset and base register can be used in three different ways to form the memory address. The addressing modes are described as follows:

Offset The offset is added to or subtracted from the base register to form the memory address.

Pre-indexed The offset is added to or subtracted from the base register to form the memory address. The base register is then updated with this new address, to permit automatic indexing through an array or memory block.

Post-indexed The value of the base register alone is used as the memory address. The offset is then added to or subtracted from the base register. The result is stored back in the base register, to permit automatic indexing through an array or memory block.

Note

Not every variant is available for every instruction, and the range of permitted immediate values and the options for scaled registers vary from instruction to instruction. See Chapter F2 *About the T32 and A32 Instruction Descriptions* for full details for each instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F1.7 Load/store multiple instructions

Load Multiple instructions load from memory a subset, or possibly all, of the general-purpose registers and the PC.

Store Multiple instructions store to memory a subset, or possibly all, of the general-purpose registers.

The memory locations are consecutive word-aligned words. The addresses used are obtained from a base register, and can be either above or below the value in the base register. The base register can optionally be updated by the total size of the data transferred.

Table F1-13 summarizes the load/store multiple instructions in the T32 and A32 instruction sets.

Table F1-13 Load/store multiple instructions

Instruction	See
Load Multiple, Increment After or Full Descending	*LDM, LDMIA, LDMFD* on page F5-4398
Load Multiple, Decrement After or Full Ascending [a]	*LDMDA, LDMFA* on page F5-4406
Load Multiple, Decrement Before or Empty Ascending	*LDMDB, LDMEA* on page F5-4408
Load Multiple, Increment Before or Empty Descending [a]	*LDMIB, LDMED* on page F5-4411
Pop multiple registers off the stack [b]	*POP* on page F5-4585
Push multiple registers onto the stack [c]	*PUSH* on page F5-4592
Store Multiple, Increment After or Empty Ascending	*STM, STMIA, STMEA* on page F5-4765
Store Multiple, Decrement After or Empty Descending [a]	*STMDA, STMED* on page F5-4771
Store Multiple, Decrement Before or Full Descending	*STMDB, STMFD* on page F5-4773
Store Multiple, Increment Before or Full Ascending [a]	*STMIB, STMFA* on page F5-4776

a. Not available in the T32 instruction set.
b. This instruction is equivalent to an `LDM` instruction with the SP as base register, and base register updating.
c. This instruction is equivalent to an `STMDB` instruction with the SP as base register, and base register updating.

When executing at EL1, variants of the `LDM` and `STM` instructions load and store User mode registers. Another system level variant of the `LDM` instruction performs an exception return.

F1.7.1 Loads to the PC

The `LDM`, `LDMDA`, `LDMDB`, `LDMIB`, and `POP` instructions can load a value into the PC. The value loaded is treated as an interworking address, as described by the `LoadWritePC()` pseudocode function in *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F1.8 Miscellaneous instructions

Table F1-14 summarizes the miscellaneous instructions in the T32 and A32 instruction sets.

Table F1-14 Miscellaneous instructions

Instruction	See
Clear-Exclusive	*CLREX* on page F5-4317
Data Memory Barrier	*DMB* on page F5-4354
Data Synchronization Barrier	*DSB* on page F5-4357
Error Synchronization Barrier	*ESB* on page F5-4371
Instruction Synchronization Barrier	*ISB* on page F5-4377
If-Then	*IT* on page F5-4378[a]
No Operation	*NOP* on page F5-4556
Preload Data	*PLD, PLDW (immediate)* on page F5-4574 *PLD (literal)* on page F5-4576 *PLD, PLDW (register)* on page F5-4578
Preload Instruction	*PLI (immediate, literal)* on page F5-4580 *PLI (register)* on page F5-4583
Speculation Barrier	*SB* on page F5-4659
Set Endianness	*SETEND* on page F5-4676[b]
Set Privileged Access Never	*SETPAN* on page F5-4677
Send Event	*SEV* on page F5-4678
Send Event Local	*SEVL* on page F5-4680
Wait For Event	*WFE* on page F5-4945
Wait For Interrupt	*WFI* on page F5-4947
Yield	*YIELD* on page F5-4949[c]

a. For performance reasons, Arm deprecates many uses of the `IT` instruction, see *Partial deprecation of IT* on page F1-4054.

b. Arm deprecates any use of the `SETEND` instruction.

c. See also *The Yield instruction*.

Note

Previous versions of the architecture defined the `DBG` instruction, that could provide a hint to the debug system, in this group. In Armv8, this instruction executes as a `NOP`. Arm deprecates any use of the `DBG` instruction.

F1.8.1 The Yield instruction

In a *Symmetric Multithreading* (SMT) design, a thread can use the `YIELD` instruction to give a hint to the PE that it is running on. The `YIELD` hint indicates that whatever the thread is currently doing is of low importance, and so could yield. For example, the thread might be sitting in a spin-lock. A similar use might be in modifying the arbitration priority of the snoop bus in a multiprocessor (MP) system. Defining such an instruction permits binary compatibility between SMT and SMP systems.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

AArch32 state defines a YIELD instruction as a specific NOP (No Operation) hint instruction.

The YIELD instruction has no effect in a single-threaded system, but developers of such systems can use the instruction to flag its intended use on migration to a multiprocessor or multithreading system. Operating systems can use YIELD in places where a yield hint is wanted, knowing that it will be treated as a NOP if there is no implementation benefit.

F1.8.2 Partial deprecation of IT

Armv8-A deprecates some uses of the T32 IT instruction, for performance reasons. All uses of IT that apply to instructions other than a single subsequent 16-bit instruction from a restricted set are deprecated, as are explicit references to the PC within that single 16-bit instruction. This permits the non-deprecated forms of IT and subsequent instructions to be treated as a single 32-bit conditional instruction. Table F1-15 shows the restricted set of 16-bit instructions that are not deprecated when used in conjunction with IT.

Table F1-15 Non-deprecated IT 16-bit conditional instructions

Non-deprecated 16-bit instructions	Class	Notes
MOV, MVN	Move	Deprecated when Rm or Rd is the PC.
LDR, LDRB, LDRH, LDRSB, LDRSH	Load	Deprecated for PC-relative load literal forms
STR, STRB, STRH	Store	-
ADD, ADC, RSB, SBC, SUB	Add/Subtract	Deprecated for ADD SP,SP,#imm, SUB SP,SP,#imm, and when Rm, Rdn, or Rdm is the PC
CMP, CMN	Compare	Deprecated when Rm or Rn is the PC
MUL	Multiply	-
ASR, LSL, LSR, ROR	Shift	-
AND, BIC, EOR, ORR, TST	Logical	-
BX, BLX	Branch to register	Deprecated when Rm is the PC

The full Armv7 IT instruction functionality remains available in order to execute legacy T32 code. It is IMPLEMENTATION DEFINED whether an Armv8 implementation provides an ITD control, that software can use to disable the deprecated uses of the IT instruction. In an implementation that included the ITD control, setting an ITD field to 1 disables the deprecated uses of the IT instruction, making those uses of the IT instruction UNDEFINED. The ITD control fields are:

HSCTLR.ITD When EL2 is using AArch32, makes execution of the deprecated uses of the IT UNDEFINED at EL2.

SCTLR.ITD When EL1 is using AArch32, makes execution of the deprecated uses of the IT UNDEFINED at EL0 and EL1.

SCTLR_EL1.ITD

When EL1 is using AArch64, makes execution of the deprecated uses of the IT UNDEFINED at EL0 when EL0 is using AArch32.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F1.9 Exception-generating and exception-handling instructions

The following instructions are intended specifically to cause a synchronous exception to occur:

- The SVC instruction generates a Supervisor Call exception. For more information, see *Supervisor Call (SVC) exception* on page G1-5742.
- The Breakpoint instruction BKPT provides software breakpoints. For more information, see *Breakpoint Instruction exceptions* on page G2-5827.
- In an implementation that includes EL3 the SMC instruction generates a Secure Monitor Call exception. For more information, see *Secure Monitor Call (SMC) exception* on page G1-5743.
- In an implementation that includes EL2 the HVC instruction generates a Hypervisor Call exception. For more information, see *Hypervisor Call (HVC) exception* on page G1-5744.

Debug state on page F1-4056 summarizes the Debug state instructions.

For an exception taken to an EL1 mode:

- The system level variants of the SUBS and LDM instructions can perform a return from an exception.

  Note

  The variants of SUBS include MOVS. See the references to *Subtract (exception return)*, *Move (exception return)*, and *Load Multiple (exception return)* in Table F1-16 for more information.

- The SRS instruction can be used near the start of the handler, to store return information. The RFE instruction can then perform a return from the exception using the stored return information.

In an implementation that includes EL2, the ERET instruction performs a return from an exception taken to Hyp mode.

For more information, see *Exception return to an Exception level using AArch32* on page G1-5725.

Table F1-16 summarizes the instructions, in the T32 and A32 instruction sets, for generating or handling an exception. Except for BKPT and SVC, these are system level instructions.

Table F1-16 Exception-generating and exception-handling instructions

Instruction	See
Supervisor Call	*SVC* on page F5-4847
Breakpoint	*BKPT* on page F5-4307
Secure Monitor Call	*SMC* on page F5-4694
Return From Exception	*RFE, RFEDA, RFEDB, RFEIA, RFEIB* on page F5-4624
Subtract (exception return)[a]	*SUB, SUBS (immediate)* on page F5-4831[a]
Move (exception return)[a]	*MOV, MOVS (register)* on page F5-4517[a]
Hypervisor Call	*HVC* on page F5-4375
Exception Return	*ERET* on page F5-4369
Load Multiple (exception return)	*LDM (exception return)* on page F5-4402
Store Return State	*SRS, SRSDA, SRSDB, SRSIA, SRSIB* on page F5-4729

a. The A32 instruction set includes other instruction forms that can be used for an exception return, that have previously been described as variants of SUBS PC, LR. Arm deprecates any use of these instruction forms.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F1.9.1 Debug state

Table F1-17 shows the Debug state instructions that are implemented in the T32 instruction set:

Table F1-17 T32 Debug state instructions

Mnemonic	Instruction	See	Note
`DCPS`*n*	Debug switch to EL*n*	*DCPS1* on page F5-4348 *DCPS2* on page F5-4350 *DCPS3* on page F5-4352	-
`ERET`	Debug restore PE state (DRPS)	*ERET* on page F5-4369	When executed in Debug state, the T1 encoding of `ERET` performs the DRPS operation

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F1.10 System register access instructions

The System register encoding space is indexed using the parameters {`coproc`, `opc1`, `CRn`, `CRm`, `opc2`}, see *The AArch32 System register interface* on page G1-5769. This encoding space provides System registers and System instructions. In Armv8, the only permitted values of `coproc` are `0b1110` and `0b1111`, and the following instructions give access to this encoding space:

- Instructions that transfer data between general-purpose registers and System registers. See:
  - *MCR* on page F5-4505.
  - *MCRR* on page F5-4507.
  - *MRC* on page F5-4528.
  - *MRRC* on page F5-4530.
- Instructions that load or store from memory to a System register. See:
  - *LDC (immediate)* on page F5-4394.
  - *LDC (literal)* on page F5-4396.
  - *STC* on page F5-4745.

Note

The System register encoding space with `coproc==0b101x` is redefined to provide some of the Advanced SIMD and floating-point functionality. That is, to:

- Initiate a floating-point data-processing operation, see *Floating-point data-processing instructions* on page F1-4072.
- Transfer data between general-purpose registers and the Advanced SIMD and floating-point registers, see *Advanced SIMD and floating-point register transfer instructions* on page F1-4060.
- Load or store data to the Advanced SIMD and floating-point registers, see *Advanced SIMD and floating-point load/store instructions* on page F1-4058.

System register access instructions are part of the instruction stream executed by the PE, and therefore any System register access instruction that cannot be executed by the implementation causes an Undefined Instruction exception. In Armv8-A and Armv8-R, the instruction encodings in the System register access instruction encoding space are unallocated, and generate Undefined Instruction exceptions, except for:

- The instructions summarized in this section that access the `coproc==0b111x` encoding space.
- The instructions in the `coproc==0b101x` encoding space that are redefined to provide Advanced SIMD and floating-point functionality, as summarized in the Note in this section.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F1.11 Advanced SIMD and floating-point load/store instructions

Table F1-18 summarizes the SIMD and floating-point register file load/store instructions in the Advanced SIMD and floating-point instruction sets.

Advanced SIMD also provides instructions for loading and storing multiple elements, or structures of elements, see *Element and structure load/store instructions*.

Table F1-18 SIMD and floating-point register file load/store instructions

Instruction	See	Operation
Vector Load Multiple	*VLDM, VLDMDB, VLDMIA* on page F6-5259	Load 1-16 consecutive 64-bit registers, Advanced SIMD and floating-point. Load 1-16 consecutive 32-bit registers, floating-point only.
Vector Load Register	*VLDR (immediate)* on page F6-5264 *VLDR (literal)* on page F6-5267	Load one 64-bit register, Advanced SIMD and floating-point. Load one 32-bit register, floating-point only.
Vector Store Multiple	*VSTM, VSTMDB, VSTMIA* on page F6-5617	Store 1-16 consecutive 64-bit registers, Advanced SIMD and floating-point. Store 1-16 consecutive 32-bit registers, floating-point only.
Vector Store Register	*VSTR* on page F6-5622	Store one 64-bit register, Advanced SIMD and floating-point. Store one 32-bit register, floating-point only.

F1.11.1 Element and structure load/store instructions

Table F1-19 shows the element and structure load/store instructions available in the Advanced SIMD instruction set. Loading and storing structures of more than one element automatically de-interleaves or interleaves the elements, see Figure F1-1 on page F1-4059 for an example of de-interleaving. Interleaving is the inverse process.

Table F1-19 Element and structure load/store instructions

Instruction	See
Load single element	
Multiple elements	*VLD1 (multiple single elements)* on page F6-5213
To one lane	*VLD1 (single element to one lane)* on page F6-5205
To all lanes	*VLD1 (single element to all lanes)* on page F6-5210
Load 2-element structure	
Multiple structures	*VLD2 (multiple 2-element structures)* on page F6-5230
To one lane	*VLD2 (single 2-element structure to one lane)* on page F6-5221
To all lanes	*VLD2 (single 2-element structure to all lanes)* on page F6-5227
Load 3-element structure	
Multiple structures	*VLD3 (multiple 3-element structures)* on page F6-5244
To one lane	*VLD3 (single 3-element structure to one lane)* on page F6-5235
To all lanes	*VLD3 (single 3-element structure to all lanes)* on page F6-5241

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table F1-19 Element and structure load/store instructions (continued)

Instruction	See
Load 4-element structure	
Multiple structures	*VLD4 (multiple 4-element structures)* on page F6-5256
To one lane	*VLD4 (single 4-element structure to one lane)* on page F6-5247
To all lanes	*VLD4 (single 4-element structure to all lanes)* on page F6-5253
Store single element	
Multiple elements	*VST1 (multiple single elements)* on page F6-5580
From one lane	*VST1 (single element from one lane)* on page F6-5575
Store 2-element structure	
Multiple structures	*VST2 (multiple 2-element structures)* on page F6-5594
From one lane	*VST2 (single 2-element structure from one lane)* on page F6-5588
Store 3-element structure	
Multiple structures	*VST3 (multiple 3-element structures)* on page F6-5605
From one lane	*VST3 (single 3-element structure from one lane)* on page F6-5599
Store 4-element structure	
Multiple structures	*VST4 (multiple 4-element structures)* on page F6-5614
From one lane	*VST4 (single 4-element structure from one lane)* on page F6-5608

Figure F1-1 shows the de-interleaving of a `VLD3.16` (multiple 3-element structures) instruction:

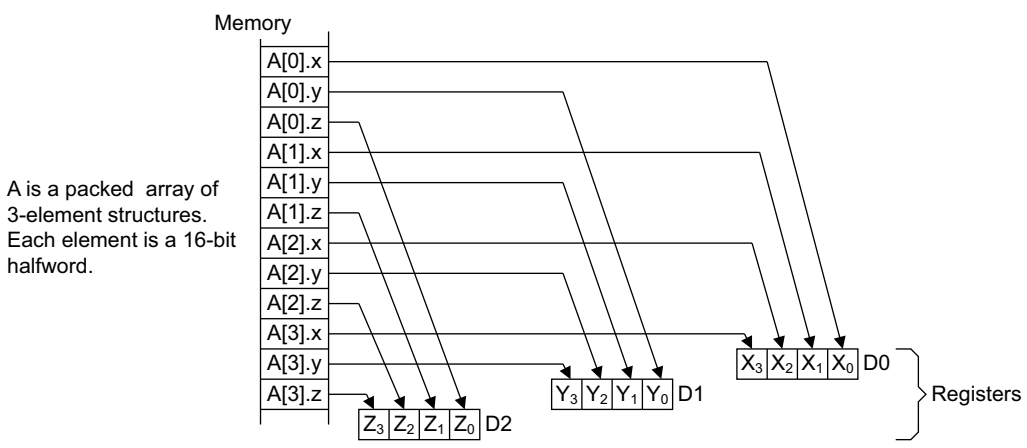


Figure F1-1 De-interleaving an array of 3-element structures

Figure F1-1 shows the `VLD3.16` instruction operating to three 64-bit registers that comprise four 16-bit elements:

- Different instructions in this group would produce similar figures, but operate on different numbers of registers. For example, `VLD4` and `VST4` instructions operate on four registers.
- Different element sizes would produce similar figures but with 8-bit or 32-bit elements.
- These instructions operate only on doubleword (64-bit) registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F1.12 Advanced SIMD and floating-point register transfer instructions

Table F1-20 summarizes the SIMD and floating-point register file transfer instructions in the Advanced SIMD and floating-point instruction sets. These instructions transfer data between the general-purpose registers and the registers in the SIMD and floating-point register file.

Advanced SIMD vectors, and single-precision and double-precision floating-point registers, are all views of the same register file. For details, see *The SIMD and floating-point register file* on page E1-3960.

Table F1-20 SIMD and floating-point register file transfer instructions

Instruction	See
Copy element from general-purpose register to every element of an Advanced SIMD vector	*VDUP (general-purpose register)* on page F6-5155
Copy byte, halfword, or word from general-purpose register to a register in the SIMD and floating-point register file	*VMOV (general-purpose register to scalar)* on page F6-5331
Copy byte, halfword, or word from a register in the SIMD and floating-point register file to a general-purpose register	*VMOV (scalar to general-purpose register)* on page F6-5335
Copy from half-precision floating-point register to general-purpose register, or from general-purpose register to half-precision floating-point register Only supported if ARMv8.2-FP16 is implemented	*VMOV (between general-purpose register and half-precision)* on page F6-5318
Copy from single-precision floating-point register to general-purpose register, or from general-purpose register to single-precision floating-point register	*VMOV (between general-purpose register and single-precision)* on page F6-5333
Copy two words from general-purpose registers to consecutive single-precision floating-point registers, or from consecutive single-precision floating-point registers to general-purpose registers	*VMOV (between two general-purpose registers and two single-precision registers)* on page F6-5337
Copy two words from general-purpose registers to a doubleword register in the SIMD and floating-point register file, or from a doubleword register in the SIMD and floating-point register file to general-purpose registers	*VMOV (between two general-purpose registers and a doubleword floating-point register)* on page F6-5316
Copy from an Advanced SIMD and floating-point System Register to a general-purpose register	*VMRS* on page F6-5346
Copy from a general-purpose register to an Advanced SIMD and floating-point System Register	*VMSR* on page F6-5349

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F1.13 Advanced SIMD data-processing instructions

Advanced SIMD data-processing instructions process registers containing vectors of elements of the same type packed together, enabling the same operation to be performed on multiple items in parallel.

Instructions operate on vectors held in 64-bit or 128-bit registers. Figure F1-2 shows an operation on two 64-bit operand vectors, generating a 64-bit vector result.

Note

Figure F1-2 and other similar figures show 64-bit vectors that consist of four 16-bit elements, and 128-bit vectors that consist of four 32-bit elements. Other element sizes produce similar figures, but with one, two, eight, or sixteen operations performed in parallel instead of four.

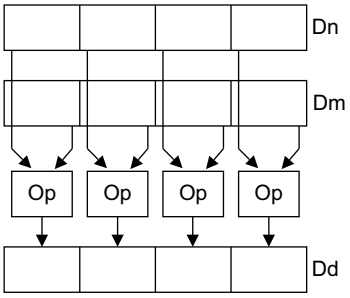


Figure F1-2 Advanced SIMD instruction operating on 64-bit registers

Many Advanced SIMD instructions have variants that produce vectors of elements double the size of the inputs. In this case, the number of elements in the result vector is the same as the number of elements in the operand vectors, but each element, and the whole vector, is double the size.

Figure F1-3 shows an example of an Advanced SIMD instruction operating on 64-bit registers, and generating a 128-bit result.

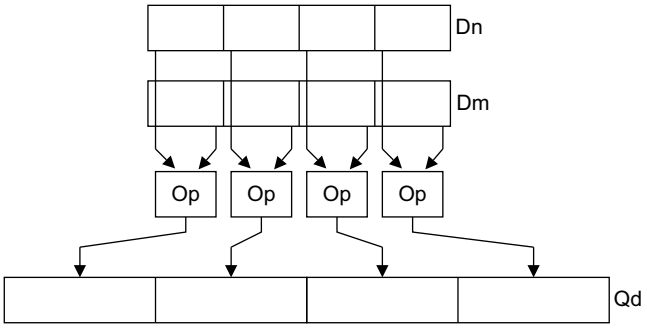


Figure F1-3 Advanced SIMD instruction producing wider result

There are also Advanced SIMD instructions that have variants that produce vectors containing elements half the size of the inputs. Figure F1-4 on page F1-4062 shows an example of an Advanced SIMD instruction operating on one 128-bit register, and generating a 64-bit result.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

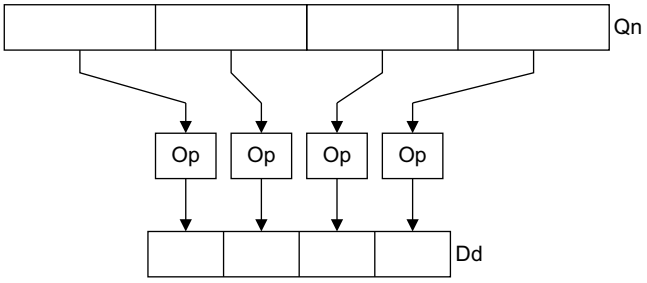


Figure F1-4 Advanced SIMD instruction producing narrower result

Some Advanced SIMD instructions do not conform to these standard patterns. Their operation patterns are described in the individual instruction descriptions.

Advanced SIMD instructions that perform floating-point arithmetic use the Arm standard floating-point arithmetic defined in *Advanced SIMD and floating-point support* on page A1-52.

The following sections summarize the Advanced SIMD data-processing instructions:

- *Advanced SIMD parallel addition and subtraction*.
- *Bitwise Advanced SIMD data-processing instructions* on page F1-4063.
- *Advanced SIMD comparison instructions* on page F1-4064.
- *Advanced SIMD shift instructions* on page F1-4065.
- *Advanced SIMD multiply instructions* on page F1-4066.
- *Advanced SIMD dot product instructions* on page F1-4067.
- *Miscellaneous Advanced SIMD data-processing instructions* on page F1-4069.
- *Advanced SIMD BFloat16 instructions* on page F1-4068.
- *Advanced SIMD matrix multiply instructions* on page F1-4068.
- *The Cryptographic Extension in AArch32 state* on page F1-4070.

F1.13.1 Advanced SIMD parallel addition and subtraction

Table F1-21 shows the Advanced SIMD parallel add and subtract instructions.

Table F1-21 Advanced SIMD parallel add and subtract instructions

Instruction	See
Vector Add	*VADD (integer)* on page F6-5019 *VADD (floating-point)* on page F6-5015
Vector Add and Narrow, returning High Half	*VADDHN* on page F6-5021
Vector Add Long	*VADDL* on page F6-5023
Vector Add Wide	*VADDW* on page F6-5025
Vector Halving Add	*VHADD* on page F6-5195
Vector Halving Subtract	*VHSUB* on page F6-5198
Vector Pairwise Add and Accumulate Long	*VPADAL* on page F6-5395
Vector Pairwise Add	*VPADD (integer)* on page F6-5400 *VPADD (floating-point)* on page F6-5398
Vector Pairwise Add Long	*VPADDL* on page F6-5402
Vector Rounding Add and Narrow, returning High Half	*VRADDHN* on page F6-5471

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table F1-21 Advanced SIMD parallel add and subtract instructions (continued)

Instruction	See
Vector Rounding Halving Add	*VRHADD* on page F6-5486
Vector Rounding Subtract and Narrow, returning High Half	*VRSUBHN* on page F6-5534
Vector Saturating Add	*VQADD* on page F6-5419
Vector Saturating Subtract	*VQSUB* on page F6-5469
Vector Subtract	*VSUB (integer)* on page F6-5629 *VSUB (floating-point)* on page F6-5625
Vector Subtract and Narrow, returning High Half	*VSUBHN* on page F6-5631
Vector Subtract Long	*VSUBL* on page F6-5633
Vector Subtract Wide	*VSUBW* on page F6-5635

F1.13.2 Bitwise Advanced SIMD data-processing instructions

Table F1-22 shows bitwise Advanced SIMD data-processing instructions. These operate on the doubleword (64-bit) or quadword (128-bit) registers in the SIMD and floating-point register file, and there is no division into vector elements.

Table F1-22 Bitwise Advanced SIMD data-processing instructions

Instruction	See
Vector Bitwise AND	*VAND (register)* on page F6-5030
Vector Bitwise Bit Clear (AND complement)	*VBIC (immediate)* on page F6-5032 *VBIC (register)* on page F6-5035
Vector Bitwise Exclusive OR	*VEOR* on page F6-5159
Vector Bitwise Insert if False	*VBIF* on page F6-5037
Vector Bitwise Insert if True	*VBIT* on page F6-5039
Vector Bitwise Move	*VMOV (immediate)* on page F6-5320 *VMOV (register)* on page F6-5327
Vector Bitwise NOT	*VMVN (immediate)* on page F6-5366 *VMVN (register)* on page F6-5370
Vector Bitwise OR	*VORR (immediate)* on page F6-5390 *VORR (register)* on page F6-5393
Vector Bitwise OR NOT	*VORN (register)* on page F6-5388
Vector Bitwise Select	*VBSL* on page F6-5041

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F1.13.3 Advanced SIMD comparison instructions

Table F1-23 shows Advanced SIMD comparison instructions.

Table F1-23 Advanced SIMD comparison instructions

Instruction	See
Vector Absolute Compare Greater Than or Equal	*VACGE* on page F6-5005
Vector Absolute Compare Greater Than	*VACGT* on page F6-5010
Vector Compare Equal	*VCEQ (register)* on page F6-5047
Vector Compare Equal to Zero	*VCEQ (immediate #0)* on page F6-5045
Vector Compare Greater Than or Equal	*VCGE (register)* on page F6-5053
Vector Compare Greater Than or Equal to Zero	*VCGE (immediate #0)* on page F6-5050
Vector Compare Greater Than	*VCGT (register)* on page F6-5060
Vector Compare Greater Than Zero	*VCGT (immediate #0)* on page F6-5057
Vector Compare Less Than or Equal to Zero	*VCLE (immediate #0)* on page F6-5064
Vector Compare Less Than Zero	*VCLT (immediate #0)* on page F6-5072
Vector Test Bits	*VTST* on page F6-5647

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F1.13.4 Advanced SIMD shift instructions

Table F1-24 lists the shift instructions in the Advanced SIMD instruction set.

Table F1-24 Advanced SIMD shift instructions

Instruction	See
Vector Saturating Rounding Shift Left	*VQRSHL* on page F6-5448
Vector Saturating Rounding Shift Right and Narrow	*VQRSHRN, VQRSHRUN* on page F6-5452
Vector Saturating Shift Left	*VQSHL (register)* on page F6-5460 *VQSHL, VQSHLU (immediate)* on page F6-5457
Vector Saturating Shift Right and Narrow	*VQSHRN, VQSHRUN* on page F6-5464
Vector Rounding Shift Left	*VRSHL* on page F6-5515
Vector Rounding Shift Right	*VRSHR* on page F6-5518
Vector Rounding Shift Right and Accumulate	*VRSRA* on page F6-5531
Vector Rounding Shift Right and Narrow	*VRSHRN* on page F6-5523
Vector Shift Left	*VSHL (immediate)* on page F6-5544 *VSHL (register)* on page F6-5547
Vector Shift Left Long	*VSHLL* on page F6-5550
Vector Shift Right	*VSHR* on page F6-5553
Vector Shift Right and Narrow	*VSHRN* on page F6-5558
Vector Shift Left and Insert	*VSLI* on page F6-5562
Vector Shift Right and Accumulate	*VSRA* on page F6-5569
Vector Shift Right and Insert	*VSRI* on page F6-5572

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F1.13.5 Advanced SIMD multiply instructions

Table F1-25 summarizes the Advanced SIMD multiply instructions.

Table F1-25 Advanced SIMD multiply instructions

Instruction	See
Vector Multiply Accumulate	*VMLA (integer)* on page F6-5292 *VMLA (floating-point)* on page F6-5288 *VMLA (by scalar)* on page F6-5294
Vector Multiply Accumulate Long	*VMLAL (integer)* on page F6-5297 *VMLAL (by scalar)* on page F6-5299
Vector Multiply Subtract	*VMLS (integer)* on page F6-5305 *VMLS (floating-point)* on page F6-5301 *VMLS (by scalar)* on page F6-5307
Vector Multiply Subtract Long	*VMLSL (integer)* on page F6-5310 *VMLSL (by scalar)* on page F6-5312
Vector Multiply	*VMUL (integer and polynomial)* on page F6-5355 *VMUL (floating-point)* on page F6-5351 *VMUL (by scalar)* on page F6-5358
Vector Multiply Long	*VMULL (integer and polynomial)* on page F6-5361 *VMULL (by scalar)* on page F6-5364
Vector Fused Multiply Accumulate	*VFMA* on page F6-5165
Vector Floating-Point Multiply-Add Long	*VFMAL (vector)* on page F6-5173 *VFMAL (by scalar)* on page F6-5176
Vector Fused Multiply Subtract	*VFMS* on page F6-5179
Vector Floating-Point Multiply-Subtract Long	*VFMSL (vector)* on page F6-5183 *VFMSL (by scalar)* on page F6-5186
Vector Saturating Doubling Multiply Accumulate Long	*VQDMLAL* on page F6-5421
Vector Saturating Doubling Multiply Subtract Long	*VQDMLSL* on page F6-5424
Vector Saturating Doubling Multiply Returning High Half	*VQDMULH* on page F6-5427
Vector Saturating Doubling Multiply Long	*VQDMULL* on page F6-5430
Vector Saturating Rounding Doubling Multiply Accumulate Returning High Half	*VQRDMLAH* on page F6-5437
Vector Saturating Rounding Doubling Multiply Subtract Returning High Half	*VQRDMLSH* on page F6-5441
Vector Saturating Rounding Doubling Multiply Returning High Half	*VQRDMULH* on page F6-5445

Advanced SIMD multiply instructions can operate on vectors of:

- 8-bit, 16-bit, or 32-bit unsigned integers.
- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit polynomials over {0, 1}. `VMUL` and `VMULL` are the only instructions that operate on polynomials. `VMULL` produces a 16-bit polynomial over {0, 1}.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Single-precision (32-bit) or half-precision (16-bit) floating-point numbers.

They can also act on one vector and one scalar.

Long instructions have doubleword (64-bit) operands, and produce quadword (128-bit) results. Other Advanced SIMD multiply instructions can have either doubleword or quadword operands, and produce results of the same size.

Floating-point multiply instructions can operate on:
- Half-precision (16-bit) floating-point numbers.
- Single-precision (32-bit) floating-point numbers.
- Double-precision (64-bit) floating-point numbers.

F1.13.6 Advanced SIMD dot product instructions

ARMv8.2-DotProd provides SIMD instructions that perform the dot product of the four 8-bit subelements of the 32-bit elements of one vector with the four 8-bit subelements of a second vector. It provides two forms of the instructions, each with signed and unsigned versions:

Vector form The dot product is calculated for each element of the first vector with the corresponding element of the second element.

Indexed form The dot product is calculated for each element of the first vector with the element of the second vector that is indicated by the index argument to the instruction.

Note

That is, a single element from the second vector is used, and a the dot product is calculated between each element of the first vector and this single element from the second vector.

Table F1-26 Advanced SIMD dot product instructions

Mnemonic	Instruction	See
VSDOT	Signed dot product (vector form)	*VSDOT (vector)* on page F6-5538
VUDOT	Unsigned dot product (vector form)	*VUDOT (vector)* on page F6-5651
VSDOT	Signed dot product (indexed form)	*VSDOT (by element)* on page F6-5536
VSUDOT	Mixed sign integer dot product by indexed quadruplet[a]	*VSUDOT (by element)* on page F6-5637
VUDOT	Unsigned dot product (indexed form)	*VUDOT (by element)* on page F6-5649
VUSDOT	Mixed sign integer dot product (vector format)[a]	*VUSDOT (vector)* on page F6-5657
	Mixed sign integer dot product by indexed quadruplet[a]	*VUSDOT (by element)* on page F6-5655

a. This instruction is only supported when ARMv8.2-AA32I8MM is implemented.

F1.13.7 Advanced SIMD complex number arithmetic instructions

ARMv8.3-CompNum provides AArch32 Advanced SIMD instructions that perform arithmetic on complex numbers held in element pairs in vector registers, where the less significant element of the pair contains the real component and the more significant element contains the imaginary component.

These instructions provide single-precision versions. If ARMv8.2-FP16 is implemented they also provide half-precision versions, otherwise the half-precision encodings are UNDEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table F1-27 shows the ARMv8.3-CompNum AArch32 Advanced SIMD instructions.

Table F1-27 Advanced SIMD complex number arithmetic instructions

Mnemonic	Instruction	See
VCADD	Floating-point complex add	*VCADD* on page F6-5043
VCMLA	Floating-point complex multiply accumulate (vector form)	*VCMLA* on page F6-5080
VCMLA	Floating-point complex multiply accumulate (indexed form)	*VCMLA (by element)* on page F6-5083

A pair of VCMLA instructions can be used to perform a complex number multiplication. In *Complex multiplication* on page K10-7958, this is demonstrated for the similar AArch64 instruction FCMLA. The usage of VCMLA in this manner is identical.

F1.13.8 Advanced SIMD BFloat16 instructions

When ARMv8.2-AA32BF16 is implemented, BFloat16 instructions are available in AArch32 state.

Table F1-28 shows the Advanced SIMD BFloat16 instructions.

Table F1-28 BFloat16 Advanced SIMD instructions

Mnemonic	Instruction	See
VDOT	BFloat16 floating-point vector dot product (vector and by scalar formats)	*VDOT (vector)* on page F6-5151 *VDOT (by element)* on page F6-5153
VMMLA	BFloat16 floating-point matrix multiply-accumulate	*VMMLA* on page F6-5314
VFMAB, VFMAT	BFloat16 floating-point widening multiply-add long (vector and by scalar formats)	*VFMAB, VFMAT (BFloat16, vector)* on page F6-5169 *VFMAB, VFMAT (BFloat16, by scalar)* on page F6-5171
VCVT	BFloat16 convert from single-precision to BF16 format	*VCVT (from single-precision to BFloat16, Advanced SIMD)* on page F6-5096

F1.13.9 Advanced SIMD matrix multiply instructions

When ARMv8.2-AA32I8MM is implemented, these instructions are available in AArch32 state. They include integer and mixed sign dot product instructions.

The matrix multiply-accumulate instructions delimit source and destination vectors into segments. Within each segment:

- The first source vector matrix is organized in row-by-row order.
- The second source vector matrix elements are organized in a column-by-column order.
- The destination vector matrix is organized in row-by-row order.

One matrix multiplication is performed per segment.

Table F1-29 shows the Advanced SIMD matrix multiply instructions.

Table F1-29 Matrix multiply Advanced SIMD instructions

Mnemonic	Instruction	See
VSMMLA	Widening 8-bit signed integer matrix multiply-accumulate into 2x2 matrix	*VSMMLA* on page F6-5565
VUMMLA	Widening 8-bit unsigned integer matrix multiply-accumulate into 2x2 matrix	*VUMMLA* on page F6-5653
VUSMMLA	Widening 8-bit mixed sign integer matrix multiply-accumulate into 2x2 matrix	*VUSMMLA* on page F6-5659

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F1.13.10 Miscellaneous Advanced SIMD data-processing instructions

Table F1-30 shows miscellaneous Advanced SIMD data-processing instructions.

Table F1-30 Miscellaneous Advanced SIMD data-processing instructions

Instruction	See
Vector Absolute Difference and Accumulate	*VABA* on page F6-4991
Vector Absolute Difference and Accumulate Long	*VABAL* on page F6-4993
Vector Absolute Difference	*VABD (integer)* on page F6-4997 *VABD (floating-point)* on page F6-4995
Vector Absolute Difference Long	*VABDL (integer)* on page F6-4999
Vector Absolute	*VABS* on page F6-5001
Vector Convert between floating-point and fixed point	*VCVT (between floating-point and fixed-point, Advanced SIMD)* on page F6-5112
Vector Convert between floating-point and integer	*VCVT (between floating-point and integer, Advanced SIMD)* on page F6-5102
Vector Convert between half-precision and single-precision	*VCVT (between half-precision and single-precision, Advanced SIMD)* on page F6-5100
Vector Count Leading Sign Bits	*VCLS* on page F6-5070
Vector Count Leading Zeros	*VCLZ* on page F6-5078
Vector Count Set Bits	*VCNT* on page F6-5094
Vector Duplicate scalar	*VDUP (scalar)* on page F6-5157
Vector Extract	*VEXT (byte elements)* on page F6-5161
Vector move Insertion	*VINS* on page F6-5201
Vector Move and Narrow	*VMOVN* on page F6-5342
Vector Move Long	*VMOVL* on page F6-5340
Vector Move extraction	*VMOVX* on page F6-5344
Vector Maximum	*VMAX (integer)* on page F6-5272 *VMAX (floating-point)* on page F6-5270
Vector Minimum	*VMIN (integer)* on page F6-5281 *VMIN (floating-point)* on page F6-5279
Vector Negate	*VNEG* on page F6-5372
Vector Pairwise Maximum	*VPMAX (integer)* on page F6-5407 *VPMAX (floating-point)* on page F6-5405
Vector Pairwise Minimum	*VPMIN (integer)* on page F6-5411 *VPMIN (floating-point)* on page F6-5409
Vector Reciprocal Estimate	*VRECPE* on page F6-5473
Vector Reciprocal Step	*VRECPS* on page F6-5475
Vector Reciprocal Square Root Estimate	*VRSQRTE* on page F6-5527

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table F1-30 Miscellaneous Advanced SIMD data-processing instructions (continued)

Instruction	See
Vector Reciprocal Square Root Step	*VRSQRTS* on page F6-5529
Vector Reverse in halfwords	*VREV16* on page F6-5477
Vector Reverse in words	*VREV32* on page F6-5480
Vector Reverse in doublewords	*VREV64* on page F6-5483
Vector Saturating Absolute	*VQABS* on page F6-5417
Vector Saturating Move and Narrow	*VQMOVN, VQMOVUN* on page F6-5433
Vector Saturating Negate	*VQNEG* on page F6-5435
Vector Swap	*VSWP* on page F6-5639
Vector Table Lookup	*VTBL, VTBX* on page F6-5641
Vector Transpose	*VTRN* on page F6-5644
Vector Unzip	*VUZP* on page F6-5661
Vector Zip	*VZIP* on page F6-5665

F1.13.11 The Cryptographic Extension in AArch32 state

The instructions provided by the optional Cryptographic Extension use the Advanced SIMD and floating-point register file. For more information about the functions they provide see:

- Announcing the Advanced Encryption Standard.
- *The Galois/Counter Mode of Operation*.
- *Announcing the Secure Hash Standard*.

Table F1-31 shows the AArch32 Cryptographic Extension instructions.

Table F1-31 AArch32 Cryptographic Extension instructions

Mnemonic	Instruction	See
`AESD`	AES single round decryption	*AESD* on page F6-4957
`AESE`	AES single round encryption	*AESE* on page F6-4959
`AESIMC`	AES inverse mix columns	*AESIMC* on page F6-4961
`AESMC`	AES mix columns	*AESMC* on page F6-4963
`VMULL`	Polynomial multiply long	*VMULL (integer and polynomial)* on page F6-5361[a]
`SHA1C`	SHA1 hash update (choose)	*SHA1C* on page F6-4971
`SHA1H`	SHA1 fixed rotate	*SHA1H* on page F6-4973
`SHA1M`	SHA1 hash update (majority)	*SHA1M* on page F6-4975
`SHA1P`	SHA1 hash update (parity)	*SHA1P* on page F6-4977
`SHA1SU0`	SHA1 schedule update 0	*SHA1SU0* on page F6-4979
`SHA1SU1`	SHA1 schedule update 1	*SHA1SU1* on page F6-4981
`SHA256H`	SHA256 hash update (part 1)	*SHA256H* on page F6-4983

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table F1-31 AArch32 Cryptographic Extension instructions (continued)

Mnemonic	Instruction	See
SHA256H2	SHA256 hash update (part 2)	*SHA256H2* on page F6-4985
SHA256SU0	SHA256 schedule update 0	*SHA256SU0* on page F6-4987
SHA256SU1	SHA256 schedule update 1	*SHA256SU1* on page F6-4989

a. The Cryptographic Extension adds the variant of the instruction that operates on two 64-bit polynomials.

See *The Armv8 Cryptographic Extension* on page A2-68 for information about the permitted implementation options for the Cryptographic Extension.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F1.14 Floating-point data-processing instructions

Table F1-32 summarizes the data-processing instructions in the floating-point instruction set. In this table, *floating-point register* means a register in the SIMD and floating-point register file. The BFloat16 floating-point instructions are provided by ARMv8.2-AA32BF16.

For details of the floating-point arithmetic used by floating-point instructions, see *Advanced SIMD and floating-point support* on page A1-52.

Table F1-32 Floating-point data-processing instructions

Instruction	See
BFloat16 convert from single-precision to BF16 format writing to bottom half of single-precision register	*VCVTB (BFloat16)* on page F6-5126
BFloat16 convert from single-precision to BF16 format writing to top half of single-precision register	*VCVTT (BFloat16)* on page F6-5147
Convert between double-precision and single-precision	*VCVT (between double-precision and single-precision)* on page F6-5098
Convert between floating-point and fixed-point	*VCVT (between floating-point and fixed-point, floating-point)* on page F6-5115
Convert between half-precision and single-precision, writing to bottom half of single-precision register	*VCVTB* on page F6-5123
Convert between half-precision and single-precision, writing to top half of single-precision register	*VCVTT* on page F6-5144
Convert from floating-point to integer	*VCVT (floating-point to integer, floating-point)* on page F6-5105
Convert from floating-point to integer using FPSCR rounding mode	*VCVTR* on page F6-5140
Convert from integer to floating-point	*VCVT (integer to floating-point, floating-point)* on page F6-5109
Floating-point Javascript convert to signed fixed-point, rounding toward zero	*VJCVT* on page F6-5203
Copy from one floating-point register to another	*VMOV (register)* on page F6-5327
Divide	*VDIV* on page F6-5149
Move immediate value to a floating-point register	*VMOV (immediate)* on page F6-5320
Square Root	*VSQRT* on page F6-5567
Vector Absolute value	*VABS* on page F6-5001
Vector Add	*VADD (floating-point)* on page F6-5015
Vector Compare with exceptions disabled	*VCMPE* on page F6-5090
Vector Compare with exceptions enabled	*VCMP* on page F6-5086
Vector Fused Multiply Accumulate	*VFMA* on page F6-5165
Vector Fused Multiply Subtract	*VFMS* on page F6-5179
Vector Fused Negate Multiply Accumulate	*VFNMA* on page F6-5189
Vector Fused Negate Multiply Subtract	*VFNMS* on page F6-5192
Vector Multiply	*VMUL (floating-point)* on page F6-5351

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table F1-32 Floating-point data-processing instructions (continued)

Instruction	See
Vector Multiply Accumulate	*VMLA (floating-point)* on page F6-5288
Vector Multiply Subtract	*VMLS (floating-point)* on page F6-5301
Vector Negate Multiply	*VNMUL* on page F6-5382
Vector Negate Multiply Accumulate	*VNMLA* on page F6-5376
Vector Negate Multiply Subtract	*VNMLS* on page F6-5379
Vector Negate, by inverting the sign bit	*VNEG* on page F6-5372
Vector Subtract	*VSUB (floating-point)* on page F6-5625

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter F2
About the T32 and A32 Instruction Descriptions

This chapter describes each instruction. It contains the following sections:

- *Format of instruction descriptions* on page F2-4076.
- *Standard assembler syntax fields* on page F2-4080.
- *Conditional execution* on page F2-4081.
- *Shifts applied to a register* on page F2-4083.
- *Memory accesses* on page F2-4085.
- *Encoding of lists of general-purpose registers and the PC* on page F2-4086.
- *General information about the T32 and A32 instruction descriptions* on page F2-4087.
- *Additional pseudocode support for instruction descriptions* on page F2-4100.
- *Additional information about Advanced SIMD and floating-point instructions* on page F2-4101.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F2.1 Format of instruction descriptions

The instruction descriptions in Chapter F5 *T32 and A32 Base Instruction Set Instruction Descriptions* and Chapter F6 *T32 and A32 Advanced SIMD and Floating-point Instruction Descriptions* normally use the following format:

- Instruction section title.
- Introduction to the instruction.
- A description of each encoding of the instruction.
- Assembler syntax.
- Pseudocode describing how the instruction operates.
- Notes, if applicable.

Each of these items is described in more detail in the following subsections.

F2.1.1 Instruction section title

The instruction section title gives the base mnemonic for the instruction or instructions described in the section. When one mnemonic has multiple forms described in separate instruction sections, this is followed by a short description of the form in parentheses. The most common use of this is to distinguish between forms of an instruction in which one of the operands is an immediate value and forms in which it is a register.

F2.1.2 Introduction to the instruction

The introduction to the instruction briefly describes the main features of the instruction. This description is not necessarily complete and is not definitive. If there is any conflict between it and the more detailed information that follows, the latter takes priority.

F2.1.3 Instruction encodings

This is a list of one or more instruction encodings. Each instruction encoding is labelled as:

- A1, A2, A3 … for the first, second, third and any additional A32 encodings.
- T1, T2, T3 … for the first, second, third and any additional T32 encodings.

Each instruction encoding description consists of:

- An assembly syntax that ensures that the assembler selects the encoding in preference to any other encoding. In some cases, multiple syntax variants are given. These are written in a `typewriter` font using the conventions described in *Assembler syntax prototype line conventions* on page F2-4078. The correct one to use can be indicated by:
  - A subheading that identifies the encodings that correspond to the syntax. See, for example, the subheading *Flag setting, rotate right with extend variant* in the description of the A1 encoding of the `ADC`, `ADCS` (register) instructions in *A1* on page F5-4246.
  - An annotation to the syntax, such as *Inside IT block* or *Outside IT block*. See, for example, the syntax descriptions of the T1 encoding of the `ADC`, `ADCS` (register) instructions in *T1* on page F5-4247.

  In other cases, the correct one to use can be determined by looking at the assembler syntax description and using it to determine which syntax corresponds to the instruction being disassembled.

  There is usually more than one syntax variant that ensures re-assembly to any particular encoding, and the exact set of syntaxes that do so usually depends on the register numbers, immediate constants and other operands to the instruction. For example, when assembling to the T32 instruction set, the syntax `AND R0, R0, R8` ensures selection of a 32-bit encoding but `AND R0, R0, R1` selects a 16-bit encoding.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For each instruction encoding belonging to a target instruction set, an assembler can use this information to determine whether it can use that encoding to encode the instruction requested by the UAL source. If multiple encodings can encode the instruction, then:

— If both a 16-bit encoding and a 32-bit encoding can encode the instruction, the architecture *prefers* the 16-bit encoding. This means the assembler must use the 16-bit encoding rather than the 32-bit encoding.

  Software can use the `.W` and `.N` qualifiers to specify the required encoding width, see *Standard assembler syntax fields* on page F2-4080.

— If multiple encodings of the same length can encode the instruction, the *Assembler syntax* subsection says which encoding is preferred, and how software can, instead, select the other encodings.

  Each encoding also documents UAL syntax that selects it in preference to any other encoding.

If no encodings of the target instruction set can encode the instruction requested by the UAL source, normally the assembler generates an error saying that the instruction is not available in that instruction set.

Note

In some cases, an instruction is available in one instruction set but not in another. The *Assembler syntax* subsection identifies many of these cases. For example, the A32 instructions with bits<31:28> == `0b1111` described in *Branch, branch with link, and block data transfer* on page F4-4206, *System register access, Advanced SIMD, floating-point, and Supervisor call* on page F4-4208, and *Unconditional instructions* on page F4-4223 cannot have a Condition code, but the equivalent T32 instructions often can, and this usually appears in the *Assembler syntax* subsection as a statement that the A32 instruction cannot be conditional.

However, some such cases are too complex to describe in the available space, so the definitive test of whether an instruction is available in a given instruction set is whether there is an available encoding for it in that instruction set.

The assembly syntax given for an encoding is therefore a suitable one for a disassembler to disassemble that encoding to. However, disassemblers might wish to use simpler syntaxes when they are suitable for the operand combination, in order to produce more readable disassembled code.

- An encoding diagram, where:

  — For a 32-bit A32 encoding diagram, the bits are numbered from 31 to 0.

  — For a 16-bit T32 encoding diagram, the bits are numbered from 15 to 0.

    This halfword can be described as `hw1` of the instruction.

  — For a 32-bit T32 encoding diagram, the bits are numbered from 15 to 0 for each halfword, as a reminder that a 32-bit T32 instruction consists of two consecutive halfwords rather than a word.

    In this case, the left-hand halfword in the diagram is identified as `hw1`, and the right-hand halfword is identified as `hw2`.

  Where instructions are stored using the standard little-endian instruction endianness:

  — The encoding diagram for an A32 instruction at address A shows, from left to right, the bytes at addresses A+3, A+2, A+1, A.

  — The encoding diagram for a 32-bit T32 instruction shows bytes in the order A+1, A for `hw1`, followed by bytes A+3, A+2 for `hw2`.

- Encoding-specific pseudocode. This is pseudocode that translates the encoding-specific instruction fields into inputs to the encoding-independent pseudocode in the *Operation* subsection, and that picks out any special cases in the encoding. For a detailed description of the pseudocode used and of the relationship between the encoding diagram, the encoding-specific pseudocode and the encoding-independent pseudocode, see Appendix K13 *Arm Pseudocode Definition*.

F2.1.4 Assembler symbols

The *Assembly symbols* describe the standard UAL syntax for the instruction.

Each syntax description consists of the following elements:

- Descriptions of all of the variable or optional fields of the syntax.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Some syntax fields are standardized across all or most instructions. *Standard assembler syntax fields* on page F2-4080 describes these fields.

By default, syntax fields that specify registers, such as `<Rd>`, `<Rn>`, or `<Rt>`, can be any of R0-R12 or LR in T32 instructions, and any of R0-R12, SP or LR in A32 instructions. These require that the encoding-specific pseudocode set the corresponding integer variable (such as `d`, `n`, or `t`) to the corresponding register number, using 0-12 for R0-R12, 13 for SP, or 14 for LR:

— Normally, software can do this by setting the corresponding field in the instruction, typically named Rd, Rn, Rt, to the binary encoding of that number.

— In the case of 16-bit T32 encodings, the field is normally of length 3, and so the encoding is only available when the assembler syntax specifies one of R0-R7. Such encodings often use a register field name like Rdn. This indicates that the encoding is only available if `<Rd>` and `<Rn>` specify the same register, and that the register number of that register is encoded in the field if they do.

The description of a syntax field that specifies a register sometimes extends or restricts the permitted range of registers or documents other differences from the default rules for such fields. Examples of extensions are permitting the use of the SP in a T32 instruction, or permitting the use of the PC, identified using register number 15.

- Where appropriate, text that briefly describes changes from the pre-UAL assembler syntax. Where present, this usually consists of an alternative pre-UAL form of the assembler mnemonic. The pre-UAL assembler syntax does not conflict with UAL. Arm recommends that it is supported, as an optional extension to UAL, so that pre-UAL assembler source files can be assembled.

Assembler syntax prototype line conventions

The following conventions are used in assembler syntax prototype lines and their subfields:

`< >`

Any item bracketed by `<` and `>` is a short description of a type of value to be supplied by the user in that position. A longer description of the item is normally supplied by subsequent text. Such items often correspond to a similarly named field in an encoding diagram for an instruction. When the correspondence only requires the binary encoding of an integer value or register number to be substituted into the instruction encoding, it is not described explicitly. For example, if the assembler syntax for an instruction contains an item `<Rn>` and the instruction encoding diagram contains a 4-bit field named Rn, the number of the register specified in the assembler syntax is encoded in binary in the instruction field.

If the correspondence between the assembler syntax item and the instruction encoding is more complex than simple binary encoding of an integer or register number, the item description indicates how it is encoded. This is often done by specifying a required output from the encoding-specific pseudocode, such as `add = TRUE`. The assembler must only use encodings that produce that output.

`{ }`

Any item bracketed by `{` and `}` is optional. A description of the item and of how its presence or absence is encoded in the instruction is normally supplied by subsequent text.

Many instructions have an optional destination register. Unless otherwise stated, if such a destination register is omitted, it is the same as the immediately following source register in the instruction syntax.

`#`

In the assembler syntax, numeric constants are normally preceded by a `#`. Some UAL instruction syntax descriptions explicitly show this `#` as optional. Any UAL assembler:

- Must treat the `#` as optional where an instruction syntax description shows it as optional.
- Can treat the `#` either as mandatory or as optional where an instruction syntax description does not show it as optional.

Note

Arm recommends that UAL assemblers treat all uses of `#` shown in this manual as optional.

spaces

Single spaces are used for clarity, to separate items. When a space is obligatory in the assembler syntax, two or more consecutive spaces are used.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

+/- This indicates an optional + or - sign. If neither is coded, + is assumed.

All other characters must be encoded precisely as they appear in the assembler syntax. Apart from { and }, the special characters described above do not appear in the basic forms of assembler instructions documented in this manual. In a few places, the { and } characters must be encoded as part of a variable item. When this happens, the long description of the variable item indicates how they must be used.

F2.1.5 Pseudocode describing how the instruction operates

The *Operation for all classes* subsection contains encoding-independent pseudocode that describes the main operation of the instruction. For a detailed description of the pseudocode used and of the relationship between the encoding diagram, the encoding-specific pseudocode and the encoding-independent pseudocode, see Appendix K13 *Arm Pseudocode Definition*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F2.2 Standard assembler syntax fields

The following assembler syntax fields are standard across all or most instructions:

`<c>` Is an optional field. It specifies the condition under which the instruction is executed. See *Conditional execution* on page F2-4081 for the range of available conditions and their encoding. If `<c>` is omitted, it defaults to *always* (`AL`).

`<q>` Specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- `.N` Meaning narrow, specifies that the assembler must select a 16-bit encoding for the instruction. If this is not possible, an assembler error is produced.
- `.W` Meaning wide, specifies that the assembler must select a 32-bit encoding for the instruction. If this is not possible, an assembler error is produced.

If neither `.W` nor `.N` is specified, the assembler can select either 16-bit or 32-bit encodings. If both are available, it must select a 16-bit encoding. In a few cases, more than one encoding of the same length can be available for an instruction. The rules for selecting between such encodings are instruction-specific and are part of the instruction description. The assembler syntax includes a mandatory `.W` qualifier, along with a note describing the cases in which it applies, where this qualifier is required to select a particular encoding for an instruction. Additional assembler syntax will describe the syntax when the conditions are not met.

Note

When assembling to the A32 instruction set, the `.N` qualifier produces an assembler error and the `.W` qualifier has no effect.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F2.3 Conditional execution

Most T32 and A32 instructions can be executed conditionally, based on the values of the APSR Condition flags. Table F2-1 lists the available conditions.

Table F2-1 Condition codes

cond	Mnemonic extension	Meaning (integer)	Meaning (floating-point) [a]	Condition flags
0000	EQ	Equal	Equal	Z == 1
0001	NE	Not equal	Not equal, or unordered	Z == 0
0010	CS [b]	Carry set	Greater than, equal, or unordered	C == 1
0011	CC [c]	Carry clear	Less than	C == 0
0100	MI	Minus, negative	Less than	N == 1
0101	PL	Plus, positive or zero	Greater than, equal, or unordered	N == 0
0110	VS	Overflow	Unordered	V == 1
0111	VC	No overflow	Not unordered	V == 0
1000	HI	Unsigned higher	Greater than, or unordered	C == 1 and Z == 0
1001	LS	Unsigned lower or same	Less than or equal	C == 0 or Z == 1
1010	GE	Signed greater than or equal	Greater than or equal	N == V
1011	LT	Signed less than	Less than, or unordered	N != V
1100	GT	Signed greater than	Greater than	Z == 0 and N == V
1101	LE	Signed less than or equal	Less than, equal, or unordered	Z == 1 or N != V
1110	None (AL) [d]	Always (unconditional)	Always (unconditional)	Any

a. Unordered means at least one NaN operand.
b. HS (unsigned higher or same) is a synonym for CS.
c. LO (unsigned lower) is a synonym for CC.
d. AL is an optional mnemonic extension for always, except in IT instructions. For details, see *IT* on page F5-4378.

In T32 instructions, the condition, if it is not AL, is normally encoded in a preceding IT instruction. For more information see *Conditional instructions* on page F1-4037 and *IT* on page F5-4378. Some conditional branch instructions do not require a preceding IT instruction, because they include a Condition code in their encoding.

For performance reasons, Armv8 deprecates the use of IT other than with a single 16-bit T32 instruction from a specified subset of the 16-bit T32 instructions, see *Partial deprecation of IT* on page F1-4054. In addition, implementations can provide a set of ITD control fields, SCTLR.ITD, SCTLR_EL1.ITD, and HSCTLR.ITD, to disable these deprecated uses, making them UNDEFINED. For more information, see:

- *Disabling or enabling PL0 and PL1 use of AArch32 deprecated functionality* on page G1-5781.
- *Disabling or enabling EL2 use of AArch32 deprecated functionality* on page G1-5790.

In A32 instructions, bits[31:28] of the instruction contain either:

- The Condition code, see *The Condition code field in A32 instruction encodings* on page F2-4082.
- 0b1111 for some A32 instructions that can only be executed unconditionally.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F2.3.1 The Condition code field in A32 instruction encodings

Every conditional A32 instruction contains a 4-bit Condition code field, the `cond` field, in bits 31 to 28:

31 30 29 28	27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
cond	

This field contains one of the values `0b0000-0b1110`, as shown in Table F2-1 on page F2-4081. Most instruction mnemonics can be extended with the letters defined in the *Mnemonic extension* on page F2-4081 column of that table.

If the *always* (`AL`) condition is specified, the instruction is executed irrespective of the value of the Condition flags. The absence of a Condition code on an instruction mnemonic implies the `AL` Condition code.

F2.3.2 Pseudocode description of conditional execution

The `AArch32.CurrentCond()` function returns a 4-bit condition specifier as follows:

- For A32 instructions, it returns bits[31:28] of the instruction.
- For the T1 and T3 encodings of the Branch instruction (see *B* on page F5-4291), it returns the 4-bit `cond` field of the encoding.
- For all other T32 instructions:
  - If `PSTATE.IT<3:0> != '0000'` it returns `PSTATE.IT<7:4>`.
  - If `PSTATE.IT<7:0> == '00000000'` it returns `'1110'`.
  - Otherwise, execution of the instruction is CONSTRAINED UNPREDICTABLE.

  For more information, see *Process state, PSTATE* on page E1-3953.

The `ConditionPassed()` function uses this condition specifier and the Condition flags to determine whether the instruction must be executed, by calling the `ConditionHolds()` function.

Chapter J1 *Armv8 Pseudocode* includes the definitions of these functions.

Undefined Instruction exception on page G1-5738 describes the handling of conditional instructions that are UNDEFINED, UNPREDICTABLE, or CONSTRAINED UNPREDICTABLE. The pseudocode in the manual, as a sequential description of the instructions, has limitations in this respect. For more information, see *Limitations of the instruction pseudocode* on page K13-8012.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F2.4 Shifts applied to a register

A32 register offset load/store word and unsigned byte instructions can apply a wide range of different constant shifts to the offset register. Both T32 and A32 data-processing instructions can apply the same range of different constant shifts to the second operand register. For details, see *Constant shifts*.

A32 data-processing instructions can apply a register-controlled shift to the second operand register.

F2.4.1 Constant shifts

These are the same in T32 and A32 instructions, except that the input bits come from different positions.

`<shift>` is an optional shift to be applied to `<Rm>`. It can be any one of:

(omitted) No shift.

`LSL #<n>` Logical shift left `<n>` bits. 1 <= `<n>` <= 31.

`LSR #<n>` Logical shift right `<n>` bits. 1 <= `<n>` <= 32.

`ASR #<n>` Arithmetic shift right `<n>` bits. 1 <= `<n>` <= 32.

`ROR #<n>` Rotate right `<n>` bits. 1 <= `<n>` <= 31.

`RRX` Rotate right one bit, with extend. Bit[0] is written to `shifter_carry_out`, bits[31:1] are shifted right one bit, and the Carry flag is shifted into bit[31].

Note

Assemblers can permit the use of some or all of `ASR #0`, `LSL #0`, `LSR #0`, and `ROR #0` to specify that no shift is to be performed. This is not standard UAL, and the encoding selected for T32 instructions might vary between UAL assemblers if it is used. To ensure disassembled code assembles to the original instructions, disassemblers must omit the shift specifier when the instruction specifies no shift.

Similarly, assemblers can permit the use of `#0` in the immediate forms of `ASR`, `LSL`, `LSR`, and `ROR` instructions to specify that no shift is to be performed, that is, that a `MOV` (register) instruction is wanted. Again, this is not standard UAL, and the encoding selected for T32 instructions might vary between UAL assemblers if it is used. To ensure disassembled code assembles to the original instructions, disassemblers must use the `MOV` (register) syntax when the instruction specifies no shift.

Encoding

The assembler encodes `<shift>` into two type bits and five immediate bits, as follows:

(omitted) type = `0b00`, immediate = 0.

`LSL #<n>` type = `0b00`, immediate = `<n>`.

`LSR #<n>` type = `0b01`.
If `<n>` < 32, immediate = `<n>`.
If `<n>` == 32, immediate = 0.

`ASR #<n>` type = `0b10`.
If `<n>` < 32, immediate = `<n>`.
If `<n>` == 32, immediate = 0.

`ROR #<n>` type = `0b11`, immediate = `<n>`.

`RRX` type = `0b11`, immediate = 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F2.4.2 Register controlled shifts

These are only available in A32 instructions.

`<type>` is the type of shift to apply to the value read from `<Rm>`. It must be one of:

`ASR`	Arithmetic shift right, encoded as type = `0b10`.
`LSL`	Logical shift left, encoded as type = `0b00`.
`LSR`	Logical shift right, encoded as type = `0b01`.
`ROR`	Rotate right, encoded as type = `0b11`.

The bottom byte of `<Rs>` contains the shift amount.

F2.4.3 Pseudocode description of instruction-specified shifts and rotates

The pseudocode enumeration `SRType{}` defines the shift types. Shift and rotate instruction decode is described by the pseudocode function:

- `DecodeImmShift()` for a constant shift.
- `DecodeRegShift()` for a register controlled shift.

Shift and rotate operations are made by the pseudocode function `Shift()`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F2.5 Memory accesses

Commonly, the following addressing modes are permitted for memory access instructions:

Offset addressing

The offset value is applied to an address obtained from the base register. The result is used as the address for the memory access. The value of the base register is unchanged.

The assembly language syntax for this mode is:

```
[<Rn>, <offset>]
```

Pre-indexed addressing

The offset value is applied to an address obtained from the base register. The result is used as the address for the memory access, and written back into the base register.

The assembly language syntax for this mode is:

```
[<Rn>, <offset>]!
```

Post-indexed addressing

The address obtained from the base register is used, unchanged, as the address for the memory access. The offset value is applied to the address, and written back into the base register.

The assembly language syntax for this mode is:

```
[<Rn>], <offset>
```

In each case, `<Rn>` is the base register. `<offset>` can be:

- An immediate constant, such as `<imm8>` or `<imm12>`.
- An index register, `<Rm>`.
- A shifted index register, such as `<Rm>, LSL #<shift>`.

For information about unaligned access, endianness, and exclusive access, see:

- *Alignment support* on page E2-4004.
- *Endian support* on page E2-4006.
- *Synchronization and semaphores* on page E2-4023.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F2.6 Encoding of lists of general-purpose registers and the PC

A number of instructions operate on lists of general-purpose registers. For some load instructions, the list of registers to be loaded can include the PC. For these instructions, the assembler syntax includes a `<registers>` field, that provides a list of the registers to be operated on, with list entries separated by commas.

The registers list is encoded in the instruction encoding. Most often, this is done using an 8-bit, 13-bit, or 16-bit `register_list` field. This section gives more information about these and other possible register list encodings.

In a `register_list` field, each bit corresponds to a single register, and if the `<registers>` field of the assembler instruction includes Rt then `register_list<t>` is set to 1, otherwise it is set to 0.

The full rules for the encoding of lists of general-purpose registers, and possibly the PC, are:

- Except for the cases listed here, 16-bit T32 encodings use an 8-bit register list, and can access only registers R0-R7.

  The exceptions to this rule are:

  — The T1 encoding of `POP` uses an 8-bit register list, and an additional bit, `P`, that corresponds to the PC. This means it can access any of R0-R7 and the PC.

  — The T1 encoding of `PUSH` uses an 8-bit register list, and an additional bit, `M`, that corresponds to the LR. This means it can access any of R0-R7 and the LR.

- 32-bit T32 encodings of load operations use a 13-bit register list, and two additional bits, `M`, corresponding to the LR, and `P`, corresponding to the PC. This means these instructions can access any of R0-R12 and the LR and PC.

- 32-bit T32 encodings of store operations use a 13-bit register list, and one additional bit, `M`, corresponding to the LR. This means these instructions can access any of R0-R12 and the LR.

- Except for the case listed here, A32 encodings use a 16-bit register list. This means these instructions can access any of R0-R12 and the SP, LR, and PC.

  The exception to this rule is:

  — The System instructions `LDM` (exception return) and `LDM` (User registers) use a 15-bit register list. This means these instructions can access any of R0-R12 and the SP and LR.

- The T3 and A2 encodings of `POP`, and the T3 and A2 encodings of `PUSH`, access a single register from the set of registers {R0-R12, LR, PC} and encode the register number in the `Rt` field.

Note

`POP` is a load operation, and `PUSH` is a store operation.

In every case, the encoding-specific pseudocode converts the register list into a 32-bit variable, `registers`, with a bit corresponding to each of the registers R0-R12, SP, LR, and PC.

Note

Some Advanced SIMD and floating-point instructions operate on lists of SIMD and floating-point registers. The assembler syntax of these instructions includes a `<list>` field that specifies the registers to be operated on, and the description of the instruction in *Alphabetical list of T32 and A32 base instruction set instructions* on page F5-4242 defines the use and encoding of this field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F2.7 General information about the T32 and A32 instruction descriptions

Chapter F3 *T32 Instruction Set Encoding* describes the T32 instruction encodings, and Chapter F4 *A32 Instruction Set Encoding* describes the A32 instruction encodings. The following subsections give more information about the descriptions of these instructions and their encodings:

- *Execution of instructions in debug state*.
- *Fixed values in AArch32 instruction and System register descriptions*.
- *UNDEFINED, UNPREDICTABLE, and CONSTRAINED UNPREDICTABLE instruction set space* on page F2-4088.
- *T32 and A32 Advanced SIMD and floating-point instruction encodings* on page F2-4089.
- *The PC and the use of 0b1111 as a register specifier in T32 and A32 instructions* on page F2-4093.
- *The SP and the use of 0b1101 as a register specifier in T32 and A32 instructions* on page F2-4094.
- *Modified immediate constants in T32 and A32 instructions* on page F2-4094.

F2.7.1 Execution of instructions in debug state

In general, except for the instructions described in *Debug state* on page F1-4056, the T32 instruction descriptions do not indicate any differences in the behavior of the instruction if it is executed in Debug state. For this information, see *Executing instructions in Debug state* on page H2-6937.

Note

- A32 instructions cannot be executed in Debug state.
- For many T32 instructions, execution is unchanged in Debug state. *Executing instructions in Debug state* on page H2-6937 identifies these instructions.

F2.7.2 Fixed values in AArch32 instruction and System register descriptions

This section summarizes the terms used to describe fixed values in AArch64 register and instruction descriptions. The Glossary gives full descriptions of these terms, and each entry in this section includes a link to the corresponding Glossary entry.

Note

In register descriptions, the meaning of some bits depends on the PE state. This affects the definitions of RES0 and RES1, as shown in the Glossary.

The following terms are used to describe bits or fields with fixed values:

RAZ Read-As-Zero. See *Read-As-Zero (RAZ)*.

In diagrams, a RAZ bit can be shown as 0.

(0), RES0 Reserved, *Should-Be-Zero (SBZ)* or *RES0*.

In instruction encoding diagrams, and sometimes in other descriptions, `(0)` indicates an SBZ bit. If the bit is set to 1, behavior is CONSTRAINED UNPREDICTABLE, and must be one of the following:

- The instruction is UNDEFINED.
- The instruction is treated as a `NOP`.
- The instruction executes as if the value of the bit was 0.
- Any destination registers of the instruction become UNKNOWN.

This notation can be expanded for fields, so a three-bit field can be shown as either `(0)(0)(0)` or as `(000)`.

In register diagrams, but not in the A64 encoding and instruction descriptions, bits or fields can be shown as RES0. See the Glossary definition of RES0 for more information.

Note

Some of the System instruction descriptions in this chapter are based on the *field description* of the input value for the instruction. These are register descriptions and therefore can include RES0 fields,

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The (0) and RES0 descriptions can be applied to bits or bitfields that are read-only, or are write-only. The Glossary definitions cover these cases.

RAO Read-As-One. See *Read-As-One (RAO)*.

In diagrams, a RAO bit can be shown as 1.

(1), RES1 Reserved, *Should-Be-One (SBO)* or *RES1*.

In instruction encoding diagrams, and sometimes in other descriptions, (1) indicates a SBO bit. If the bit is set to 0, behavior is CONSTRAINED UNPREDICTABLE, and must be one of the following:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if the value of the bit was 1.
- Any destination registers of the instruction become UNKNOWN.

This notation can be expanded for fields, so a three-bit field can be shown as either (1)(1)(1) or as (111).

In register diagrams, but not in the A64 encoding and instruction descriptions, bits or fields can be shown as RES1. See the Glossary definition of RES1 for more information.

Note

Some of the System instruction descriptions in this chapter are based on the *field description* of the input value for the instruction. These are register descriptions and therefore can include RES1 fields,

The (1) and RES1 descriptions can be applied to bits or bitfields that are read-only, or are write-only. The Glossary definitions cover these cases.

Note

In register diagrams, (0) is a synonym for RES0, and (1) is a synonym for RES1, where RES0 and RES1 are defined in the *Glossary*. However, when used in an instruction encoding diagram, (0) and (1) have the narrower definition that behavior is UNPREDICTABLE or CONSTRAINED UNPREDICTABLE if either:

- A bit marked as (0) has the value 1.
- A bit marked as (1) has the value 0.

F2.7.3 UNDEFINED, UNPREDICTABLE, and CONSTRAINED UNPREDICTABLE instruction set space

An attempt to execute an unallocated instruction results in either:

- Unpredictable behavior. The instruction is described as UNPREDICTABLE or CONSTRAINED UNPREDICTABLE.

  Armv8-A greatly reduces the architecturally UNPREDICTABLE behavior in AArch32 state. Most cases that earlier versions of the architecture describe as UNPREDICTABLE become either:

  — CONSTRAINED UNPREDICTABLE, meaning the architecture defines a limited range of permitted behaviors.

  — Fully predictable.

  For more information, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

- An Undefined Instruction exception. The instruction is described as UNDEFINED.

An instruction is UNDEFINED if it is declared as UNDEFINED in an instruction description, or in Chapter F3 *T32 Instruction Set Encoding* or Chapter F4 *A32 Instruction Set Encoding*.

An instruction is UNPREDICTABLE only if:

- It is declared as UNPREDICTABLE in an instruction description or in Chapter F3 or Chapter F4, and Appendix K1 does not redefine the behavior as CONSTRAINED UNPREDICTABLE.

- The pseudocode for that encoding does not indicate that a different special case applies, and a bit marked (0) or (1) in the encoding diagram of an instruction is not 0 or 1 respectively. In most cases, Armv8 makes these cases CONSTRAINED UNPREDICTABLE, as described in *SBZ or SBO fields T32 and A32 in instructions* on page K1-7834.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Unless otherwise specified, T32 and A32 instructions provided as part of an architectural extension, or by an optional feature of the architecture, are UNDEFINED in an implementation that does not include that extension or feature.

Note

Examples of where this rule applies are:

- The instructions provided by the Cryptographic Extension.
- The System instructions that provide access to the System registers of the OPTIONAL Performance Monitors Extension.
- The Advanced SIMD and floating-point instructions.

For more information about UNDEFINED, UNPREDICTABLE, and CONSTRAINED UNPREDICTABLE instruction behavior, see *Undefined Instruction exception* on page G1-5738.

F2.7.4 T32 and A32 Advanced SIMD and floating-point instruction encodings

The T32 and A32 encodings of Advanced SIMD and floating-point instructions that are described in Chapter F3 *T32 Instruction Set Encoding* and in Chapter F4 *A32 Instruction Set Encoding* are common to the T32 and A32 instruction sets. This means:

- The instruction groups, and the set of instructions in each group, are identical for T32 and A32.
- For each instruction:
  - Each T32 encoding is exactly equivalent to an A32 encoding.
  - There is no T32 encoding without an equivalent A32 encoding, and no A32 encoding without an equivalent T32 encoding.

Note

- In the T32 instruction sets, the Advanced SIMD and floating-point instructions have 32-bit encodings.
- In the base instruction sets, some instructions are common to the T32 and A32 instruction sets, whereas other instructions have equivalent but not identical functionality in the two instruction sets.

32-bit T32 encodings are described as two contiguous halfwords, `{hw1:hw2}`, as described in *Instruction encodings* on page F2-4076. In general:

- `hw1` of a T32 encoding maps onto bits[31:16] of an equivalent A32 encoding.
- `hw2` of a T32 encoding maps onto bits[15:0] of an equivalent A32 encoding.

However, the different structures of the T32 instruction encoding space and the A32 instruction encoding space mean that:

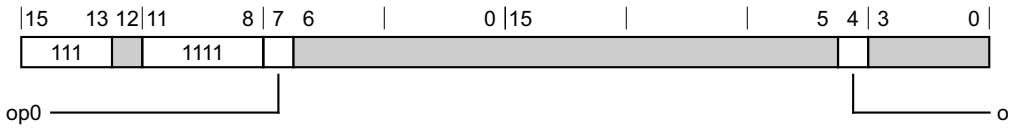
- For a given Advanced SIMD and floating-point instruction group:
  - The positions of the fields that identify the instruction, or instruction encoding, within the instruction group might differ between the T32 encodings and the A32 encodings.
  - However, the field values that identify the instruction of instruction encoding are identical for the T32 encoding and the A32 encoding.

The remainder of this section describes the equivalence of the T32 and A32 encodings for each of the Advanced SIMD and floating-point instruction groups.

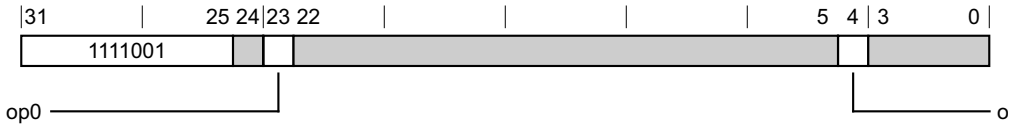
Advanced SIMD data-processing

The T32 encoding of the Advanced SIMD data-processing group is:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta



The A32 encoding of the Advanced SIMD data-processing group is:



The encodings in this group are identified by:

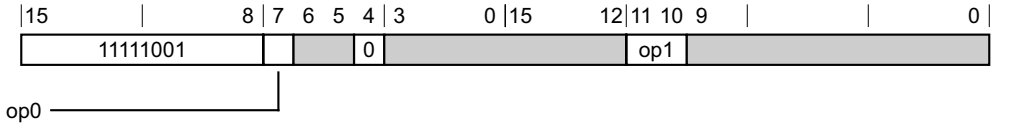
- `hw1`[15:13] of the T32 encoding is equivalent to bits[27:25] of the A32 encoding, and:
  - Has the value `0b111` in the T32 encoding.
  - Has the value `0b001` in the A32 encoding.
- `hw1`[11:8] of the T32 encoding is equivalent to bits[31:28] of the A32 encoding, and has the value `0b111`.

This table shows the equivalence of the fields that identify the instructions, or instruction encodings, within this group:

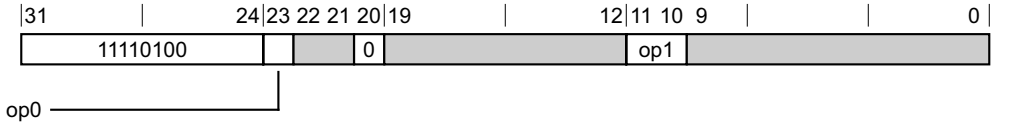
T32 encoding	A32 encoding	Field size
`op0:op1`	`op0`	2 bits
`op2`	`op1`	15 bits
`op3`	`op2`	1 bit
`op4`	`op3`	1 bit

Advanced SIMD element or structure load/store

The T32 encoding of the Advanced SIMD element or structure load/store group is:



The A32 encoding of the Advanced SIMD element or structure load/store group is:



The encodings in this group are identified by:

- `hw1`[15:12] of the T32 encoding is equivalent to bits[31:28] of the A32 encoding, and has the value `0b1111`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

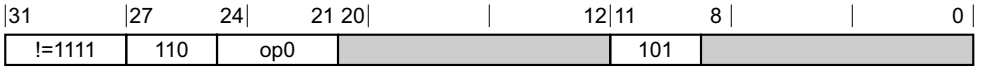
- `hw1`[11:8] of the T32 encoding is equivalent to bits[27:24] of the A32 encoding, and:
  - Has the value `0b1001` in the T32 encoding.
  - Has the value `0b0100` in the A32 encoding.
- `hw1`[4] of the T32 encoding is equivalent to bit[20] of the A32 encoding, and has the value `0b0`.

`op0`, `op1`, and `op2` are the fields that identify the instructions, or instruction encodings, within this group, and they are in equivalent positions in the T32 and A32 encodings.

Floating-point and Advanced SIMD load/store and 64-bit register moves

The T32 encoding of the Floating-point and Advanced SIMD load/store and 64-bit register moves group is:

The A32 encoding of the Floating-point and Advanced SIMD load/store and 64-bit register moves group is:



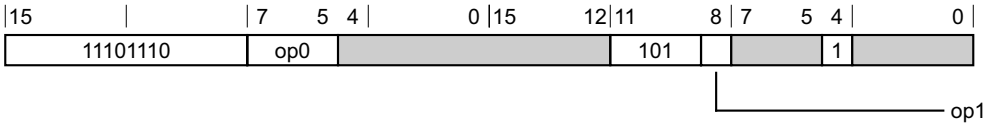
The encodings in the group are identified by:

- `hw1`[15:12] of the T32 encoding is equivalent to bits[31:28] of the A32 encoding, and:
  - Has the value `0b1110` in the T32 encoding.
  - Can have any value other than `0b1111` in the A32 encoding.
    This range of values is required because A32 instructions in this group can be executed conditionally, see *Conditional execution* on page F2-4081.
- `hw1`[11:9] of the T32 encoding is equivalent to bits[27:25] of the A32 encoding, and has the value `0b110`.
- `hw2`[11:9] of the T32 encoding is equivalent to bits[11:9] of the A32 encoding, and has the value `0b101`.

`op0` is the field that identifies the instructions, or instruction encodings, within this group, and is in equivalent positions in the T32 and A32 encodings.

Floating-point and Advanced SIMD 32-bit register moves

The T32 encoding of the Floating-point and Advanced SIMD 32-bit register moves group is:



The A32 encoding of the Advanced SIMD 32-bit register moves group is:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

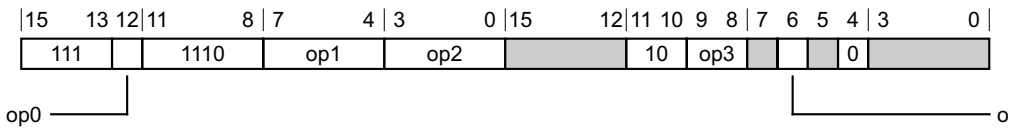
The encodings in this group are identified by:

- `hw1`[15:12] of the T32 encoding is equivalent to bits[31:28] of the A32 encoding, and:
  - Has the value `0b1110` in the T32 encoding.
  - Can have any value other than `0b1111` in the A32 encoding.
    This range of values is required because A32 instructions in this group can be executed conditionally, see *Conditional execution* on page F2-4081.
- `hw1`[11:8] of the T32 encoding is equivalent to bits[27:24] of the A32 encoding, and has the value `0b1110`.
- `hw2`[11:9] of the T32 encoding is equivalent to bits[11:9] of the A32 encoding, and has the value `0b101`.
- `hw2`[4] of the T32 encoding is equivalent to bit[4] of the A32 encoding, and has the value `0b1`.

`op0` is the field that identifies the instructions, or instruction encodings, within this group, and is in equivalent positions in the T32 and A32 encodings.

Floating-point data-processing

The T32 encoding of the Floating-point data-processing group is:



The A32 encoding of the Floating-point data-processing group is:

The encodings in this group are identified by:

- `hw1`[15:12] of the T32 encoding is equivalent to bits[31:28] of the A32 encoding, and:
  - In the T32 encoding, `hw1`[15:13] has the value `0b111`, and `hw1`[12] is the `op0` parameter used in identifying instruction encodings within this group.
  - In the A32 encoding, is the `cond` field and also implies the value of bit[28] of some A32 instruction encodings within this group, as the following table shows:

cond	Significance of bit[28] in A32 encodings
!= `0b1111`	Part of the `cond` field.
`0b1111`	Has fixed value of 1.

  The range of `cond` values other than `0b1111` is required because A32 instructions in this group can be executed conditionally, see *Conditional execution* on page F2-4081.
- `hw1`[11:8] of the T32 encoding is equivalent to bits[27:24] of the A32 encoding, and has the value `0b1110`.
- `hw2`[11:9] of the T32 encoding is equivalent to bits[11:9] of the A32 encoding, and has the value `0b101`.
- `hw2`[4] of the T32 encoding is equivalent to bit[4] of the A32 encoding, and has the value `0b0`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This table shows the equivalence of the fields that identify the instructions, or instruction encodings, within this group:

T32 encoding	A32 encoding
op0	Bit[28] of the instruction encoding is 1 when cond is 0b1111.
op1	op0
op2	op1
op3	op2

F2.7.5 The PC and the use of 0b1111 as a register specifier in T32 and A32 instructions

Restrictions on the use of PC or 0b1111 as a register specifier differ between the T32 and the A32 instruction sets, as described in:

- *T32 restrictions on the use of the PC, and use of 0b1111 as a register specifier*.
- *A32 restrictions on the use of PC or 0b1111 as a register specifier* on page F2-4094.

T32 restrictions on the use of the PC, and use of 0b1111 as a register specifier

The use of 0b1111 as a register specifier is not normally permitted in T32 instructions. When a value of 0b1111 is permitted, a variety of meanings is possible. For register reads, these meanings include:

- Read the PC value, that is, the address of the current instruction + 4. The base register of the table branch instructions TBB and TBH can be the PC. This means branch tables can be placed in memory immediately after the instruction.

  Note

  Arm deprecates use of the PC as the base register in the STC instruction.

- Read the word-aligned PC value, that is, the address of the current instruction + 4, with bits[1:0] forced to zero. The base register of LDC, LDR, LDRB, LDRD (pre-indexed, no writeback), LDRH, LDRSB, and LDRSH instructions can be the word-aligned PC. This provides PC-relative data addressing. In addition, some encodings of the ADD and SUB instructions permit their source registers to be 0b1111 for the same purpose.
- Read zero. This is done in some cases when one instruction is a special case of another, more general instruction, but with one operand zero. In these cases, the instructions are listed on separate pages, with a special case in the pseudocode for the more general instruction cross-referencing the other page.

For register writes, these meanings include:

- The PC can be specified as the destination register of an LDR instruction. This is done by encoding Rt as 0b1111. The loaded value is treated as an address, and the effect of execution is a branch to that address. Bit[0] of the loaded value selects whether to execute A32 or T32 instructions after the branch.
- Some other instructions write the PC in similar ways. An instruction can specify that the PC is written:
  - Implicitly, for example, branch instructions.
  - Explicitly by a register specifier of 0b1111, for example 16-bit MOV (register) instructions.
  - Explicitly by using a register mask, for example LDM instructions.

  The address to branch to can be:
  - A loaded value, for example, RFE.
  - A register value, for example, BX.
  - The result of a calculation, for example, TBB or TBH.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The method of choosing the instruction set used after the branch can be:

— Similar to the `LDR` case, for example, `LDM` or `BX`.

— A fixed instruction set other than the one currently being used, for example, the immediate form of `BLX`.

— Unchanged, for example, branch instructions or 16-bit `MOV` (register) instructions.

— Set from the SPSR.T bit, for `RFE` and `SUBS PC, LR, #imm8`.

- Discard the result of a calculation. This is done in some cases when one instruction is a special case of another, more general instruction, but with the result discarded. In these cases, the instructions are listed on separate pages, with a special case in the pseudocode for the more general instruction cross-referencing the other page.
- If the destination register specifier of an `LDRB`, `LDRH`, `LDRSB`, or `LDRSH` instruction is `0b1111`, the instruction is a memory hint instead of a load operation.
- If the destination register specifier of an `MRC` instruction is `0b1111`, bits[31:28] of the value transferred from the System register are written to the N, Z, C, and V Condition flags in the APSR, and bits[27:0] are discarded.

A32 restrictions on the use of PC or `0b1111` as a register specifier

In A32 instructions, the use of `0b1111` as a register specifier specifies the PC.

Many instructions are CONSTRAINED UNPREDICTABLE if they use `0b1111` as a register specifier. This is specified by pseudocode in the instruction description. Armv8-A constrains the resulting CONSTRAINED UNPREDICTABLE behavior, see *Using R15* on page K1-7831.

Note

Arm deprecates use of the PC as the base register in any store instruction.

F2.7.6 The SP and the use of `0b1101` as a register specifier in T32 and A32 instructions

In the T32 and A32 instruction sets, Arm recommends that the use of `0b1101` as a register specifier specifies the SP.

Note

- The recommendation that the register specifier `0b1101` is only used to specify the SP applies to both the T32 and the A32 instruction sets.
- Despite this recommendation, T32 instructions that can access R13, or the SP, behave predictably in Armv8. This differs from Armv7, where many uses of R13 are defined as UNPREDICTABLE. For more information about these cases, see the *ARM® Architecture Reference Manual, ARMv7-A and ARMv7-R edition*.

F2.7.7 Modified immediate constants in T32 and A32 instructions

The following sections describe the encoding of modified immediate constants:

- *Modified immediate constants in T32 instructions* on page F2-4095.
- *Modified immediate constants in A32 instructions* on page F2-4096.
- *Modified immediate constants in T32 and A32 Advanced SIMD instructions* on page F2-4097.
- *Modified immediate constants in T32 and A32 floating-point instructions* on page F2-4098.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Modified immediate constants in T32 instructions

The encoding of a modified immediate constant in a 32-bit T32 instruction is:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
					i												imm3							a	b	c	d	e	f	g	h

Table F2-2 shows the range of modified immediate constants available in T32 data-processing instructions, and their encoding in the a, b, c, d, e, f, g, h, and i bits, and the imm3 field, in the instruction.

Table F2-2 Encoding of modified immediates in T32 data-processing instructions

i:imm3:a	<const> [a]
0000x	`00000000 00000000 00000000 abcdefgh`
0001x	`00000000 abcdefgh 00000000 abcdefgh` [b]
0010x	`abcdefgh 00000000 abcdefgh 00000000` [b]
0011x	`abcdefgh abcdefgh abcdefgh abcdefgh` [b]
01000	`1bcdefgh 00000000 00000000 00000000`
01001	`01bcdefg h0000000 00000000 00000000` [c]
01010	`001bcdef gh000000 00000000 00000000`
01011	`0001bcde fgh00000 00000000 00000000` [c]
.	.
.	. 8-bit values shifted to other positions
.	.
11101	`00000000 00000000 000001bc defgh000` [c]
11110	`00000000 00000000 0000001b cdefgh00`
11111	`00000000 00000000 00000001 bcdefgh0` [c]

a. This table shows the immediate constant value in binary form, to relate `abcdefgh` to the encoding diagram. In assembly syntax, the immediate value is specified in the usual way (a decimal number by default).

b. Arm deprecates using a modified immediate with abcdefgh == 00000000, and these cases are CONSTRAINED UNPREDICTABLE, see *UNPREDICTABLE cases in immediate constants in T32 data-processing instructions* on page K1-7834.

c. Not available in A32 instructions if `h` == 1.

Note

As the footnotes to Table F2-2 show, the range of values available in T32 modified immediate constants is slightly different from the range of values available in A32 instructions. See *Modified immediate constants in A32 instructions* on page F2-4096 for the A32 values.

Carry out

A logical instruction with i:imm3:a == '00xxx' does not affect the Carry flag. Otherwise, a logical flag-setting instruction sets the Carry flag to the value of bit[31] of the modified immediate constant.

Operation of modified immediate constants, T32 instructions

For a T32 data-processing instruction, the `T32ExpandImm()` pseudocode function returns the value of the 32-bit immediate constant, calling `T32ExpandImm_C()` to evaluate the constant.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Modified immediate constants in A32 instructions

The encoding of a modified immediate constant in an A32 instruction is:

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12	11 10 9 8	7	6	5	4	3	2	1	0
	rotation	a	b	c	d	e	f	g	h

Table F2-3 shows the range of modified immediate constants available in A32 data-processing instructions, and their encoding in the a, b, c, d, e, f, g, and h bits and the rotation field in the instruction.

Table F2-3 Encoding of modified immediates in A32 processing instructions

rotation	<const> [a]
0000	`00000000 00000000 00000000 abcdefgh`
0001	`gh000000 00000000 00000000 00abcdef`
0010	`efgh0000 00000000 00000000 0000abcd`
0011	`cdefgh00 00000000 00000000 000000ab`
0100	`abcdefgh 00000000 00000000 00000000`
. . .	. . . 8-bit values shifted to other even-numbered positions
1001	`00000000 00abcdef gh000000 00000000`
. . .	. . . 8-bit values shifted to other even-numbered positions
1110	`00000000 00000000 0000abcd efgh0000`
1111	`00000000 00000000 000000ab cdefgh00`

a. This table shows the immediate constant value in binary form, to relate `abcdefgh` to the encoding diagram. In assembly syntax, the immediate value is specified in the usual way (a decimal number by default).

Note

The range of values available in A32 modified immediate constants is slightly different from the range of values available in 32-bit T32 instructions. See *Modified immediate constants in T32 instructions* on page F2-4095.

Carry out

A logical instruction with the rotation field set to `0b0000` does not affect APSR.C. Otherwise, a logical flag-setting instruction sets APSR.C to the value of bit[31] of the modified immediate constant.

Constants with multiple encodings

Some constant values have multiple possible encodings. In this case, a UAL assembler must select the encoding with the lowest unsigned value of the rotation field. This is the encoding that appears first in Table F2-3. For example, the constant `#3` must be encoded with (rotation, abcdefgh) == (`0b0000`, `0b00000011`), not (`0b0001`, `0b00001100`), (`0b0010`, `0b00110000`), or (`0b0011`, `0b11000000`).

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In particular, this means that all constants in the range 0-255 are encoded with rotation == `0b0000`, and permitted constants outside that range are encoded with rotation != `0b0000`. A flag-setting logical instruction with a modified immediate constant therefore leaves APSR.C unchanged if the constant is in the range 0-255 and sets it to the most significant bit of the constant otherwise. This matches the behavior of T32 modified immediate constants for all constants that are permitted in both the A32 and T32 instruction sets.

An alternative syntax is available for a modified immediate constant that permits the programmer to specify the encoding directly. In this syntax, `#<const>` is instead written as `#<byte>, #<rot>`, where:

`<byte>` Is the numeric value of abcdefgh, in the range 0-255.

`<rot>` Is twice the numeric value of rotation, an even number in the range 0-30.

This syntax permits all A32 data-processing instructions with modified immediate constants to be disassembled to assembler syntax that assembles to the original instruction.

This syntax also makes it possible to write variants of some flag-setting logical instructions that have different effects on APSR.C to those obtained with the normal `#<const>` syntax. For example, `ANDS R1, R2, #12, #2` has the same behavior as `ANDS R1, R2, #3` except that it sets APSR.C to 0 instead of leaving it unchanged. Such variants of flag-setting logical instructions do not have equivalents in the T32 instruction set, and Arm deprecates their use.

Operation of modified immediate constants, A32 instructions

For an A32 data-processing instruction, the `A32ExpandImm()` pseudocode function returns the value of the 32-bit immediate constant, calling `A32ExpandImm_C()` to evaluate the constant.

Modified immediate constants in T32 and A32 Advanced SIMD instructions

Table F2-4 shows the modified immediate constants available with Advanced SIMD instructions, and how they are encoded.

Table F2-4 Modified immediate values for Advanced SIMD instructions

op	cmode	Constant[a]	<dt>[b]	Notes
-	000x	`00000000 00000000 00000000 abcdefgh 00000000 00000000 00000000 abcdefgh`	I32	c
	001x	`00000000 00000000 abcdefgh 00000000 00000000 00000000 abcdefgh 00000000`	I32	c, d
	010x	`00000000 abcdefgh 00000000 00000000 00000000 abcdefgh 00000000 00000000`	I32	c, d
	011x	`abcdefgh 00000000 00000000 00000000 abcdefgh 00000000 00000000 00000000`	I32	c, d
	100x	`00000000 abcdefgh 00000000 abcdefgh 00000000 abcdefgh 00000000 abcdefgh`	I16	c
	101x	`abcdefgh 00000000 abcdefgh 00000000 abcdefgh 00000000 abcdefgh 00000000`	I16	c, d
	1100	`00000000 00000000 abcdefgh 11111111 00000000 00000000 abcdefgh 11111111`	I32	d, e
	1101	`00000000 abcdefgh 11111111 11111111 00000000 abcdefgh 11111111 11111111`	I32	d, e
0	1110	`abcdefgh abcdefgh abcdefgh abcdefgh abcdefgh abcdefgh abcdefgh abcdefgh`	I8	f
	1111	`aBbbbbbc defgh000 00000000 00000000 aBbbbbbc defgh000 00000000 00000000`	F32	f, g
1	1110	`aaaaaaaa bbbbbbbb cccccccc dddddddd eeeeeeee ffffffff gggggggg hhhhhhhh`	I64	f
	1111	UNDEFINED	-	-

a. In this table, the immediate value is shown in binary form, to relate abcdefgh to the encoding diagram. In assembler syntax, the constant is specified by a data type and a value of that type. That value is specified in the normal way (a decimal number by default) and is replicated enough times to fill the 64-bit immediate. For example, a data type of `I32` and a value of 10 specify the 64-bit constant `0x0000000A0000000A`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

b. This specifies the data type used when the instruction is disassembled. On assembly, the data type must be matched in the table if possible. Other data types are permitted as pseudo-instructions when a program is assembled, provided the 64-bit constant specified by the data type and value is available for the instruction. If a constant is available in more than one way, the first entry in this table that can produce it is used. For example, `VMOV.I64 D0, #0x8000000080000000` does not specify a 64-bit constant that is available from the I64 line of the table, but does specify one that is available from the fourth I32 line or the F32 line. It is assembled to the first of these, and therefore is disassembled as `VMOV.I32 D0, #0x80000000`.

c. This constant is available for the `VBIC`, `VMOV`, `VMVN`, and `VORR` instructions.

d. CONSTRAINED UNPREDICTABLE if `abcdefgh` == `0b00000000`, see *UNPREDICTABLE cases in immediate constants in Advanced SIMD instructions* on page K1-7835. The required behavior is that these encodings produce an immediate constant of zero.

e. This constant is available for the `VMOV` and `VMVN` instructions only.

f. This constant is available for the `VMOV` instruction only.

g. In this entry, `B` = NOT(`b`). The bit pattern represents the floating-point number $(-1)^S \times 2^{exp} \times$ mantissa, where S = `UInt(a)`, exp = `UInt(NOT(b):c:d)-3` and mantissa = `(16+UInt(e:f:g:h))/16`.

Operation of modified immediate constants, Advanced SIMD instructions

For a T32 or A32 Advanced SIMD instruction that uses a modified immediate constant, the operation described by the `AdvSIMDExpandImm()` pseudocode function returns the value of the 64-bit immediate constant.

Modified immediate constants in T32 and A32 floating-point instructions

Table F2-5 shows the immediate constants available in the `VMOV` (immediate) floating-point instruction, and Table F2-6 shows the resulting floating-point values.

Table F2-5 Floating-point modified immediate constants

Data type	imm4H	imm4L	Constant [a]
`F16`	abcd	efgh	`aBbbcdef gh000000`
`F32`	abcd	efgh	`aBbbbbbc defgh000 00000000 00000000`
`F64`	abcd	efgh	`aBbbbbbb bbcdefgh 00000000 00000000 00000000 00000000 00000000 00000000`

a. In this column, `B` = NOT(`b`). The bit pattern represents the floating-point number $(-1)^S \times 2^{exp} \times$ mantissa, where S = `UInt(a)`, exp = `UInt(NOT(b):c:d)-3` and mantissa = `(16+UInt(e:f:g:h))/16`.

Table F2-6 Floating-point constant values

efgh	bcd							
	`000`	`001`	`010`	`011`	`100`	`101`	`110`	`111`
`0000`	2.0	4.0	8.0	16.0	0.125	0.25	0.5	1.0
`0001`	2.125	4.25	8.5	17.0	0.1328125	0.265625	0.53125	1.0625
`0010`	2.25	4.5	9.0	18.0	0.140625	0.28125	0.5625	1.125
`0011`	2.375	4.75	9.5	19.0	0.1484375	0.296875	0.59375	1.1875
`0100`	2.5	5.0	10.0	20.0	0.15625	0.3125	0.625	1.25
`0101`	2.625	5.25	10.5	21.0	0.1640625	0.328125	0.65625	1.3125
`0110`	2.75	5.5	11.0	22.0	0.171875	0.34375	0.6875	1.375
`0111`	2.875	5.75	11.5	23.0	0.1796875	0.359375	0.71875	1.4375
`1000`	3.0	6.0	12.0	24.0	0.1875	0.375	0.75	1.5

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table F2-6 Floating-point constant values (continued)

efgh	bcd							
	000	001	010	011	100	101	110	111
1001	3.125	6.25	12.5	25.0	0.1953125	0.390625	0.78125	1.5625
1010	3.25	6.5	13.0	26.0	0.203125	0.40625	0.8125	1.625
1011	3.375	6.75	13.5	27.0	0.2109375	0.421875	0.84375	1.6875
1100	3.5	7.0	14.0	28.0	0.21875	0.4375	0.875	1.75
1101	3.625	7.25	14.5	29.0	0.2265625	0.453125	0.90625	1.8125
1110	3.75	7.5	15.0	30.0	0.234375	0.46875	0.9375	1.875
1111	3.875	7.75	15.5	31.0	0.2421875	0.484375	0.96875	1.9375

Operation of modified immediate constants, floating-point instructions

For a T32 or A32 floating-point instruction that uses a modified immediate constant, the operation described by the `VFPExpandImm()` pseudocode function returns the value of the immediate constant.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F2.8 Additional pseudocode support for instruction descriptions

Earlier sections of this chapter include pseudocode that describes features of the execution of A32 and T32 instructions, see:

- *Pseudocode description of conditional execution* on page F2-4082.
- *Pseudocode description of instruction-specified shifts and rotates* on page F2-4084

The following subsection gives additional pseudocode support functions for some of the instructions described in *Alphabetical list of T32 and A32 base instruction set instructions* on page F5-4242. See also *Pseudocode support for the banked register transfer instructions* on page F5-4954.

F2.8.1 Pseudocode description of operations for System register access instructions

The `AArch32.SysRegRead()` function obtains the word for an `MRC` instruction from the System register.

The `AArch32.SysRegRead64()` function obtains the two words for an `MRRC` instruction from the System register.

Note

The relative significance of the two words returned is IMPLEMENTATION DEFINED, but all uses within this manual present the two words in the order (most significant, least significant).

The `AArch32.SysRegWrite()` procedure sends the word for an `MCR` instruction to the System register.

The `AArch32.SysRegWrite64()` procedure sends the two words for an `MCRR` instruction to the System register.

Note

The relative significance of `word2` and `word1` is IMPLEMENTATION DEFINED, but all uses within this manual treat `word2` as more significant than `word1`.

F2.8.2 Pseudocode details of system calls

The `AArch32.CallSupervisor()` pseudocode function generates a Supervisor Call exception. Valid execution of the `SVC` instruction calls this function.

The `AArch32.CallHypervisor()` pseudocode function generates an HVC exception. Valid execution of the `HVC` instruction calls this function.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F2.9 Additional information about Advanced SIMD and floating-point instructions

The following subsections give additional information about the Advanced SIMD and floating-point instructions:

- *Advanced SIMD and floating-point instruction syntax*.
- *The Advanced SIMD addressing mode*.
- *Advanced SIMD instruction modifiers* on page F2-4102.
- *Advanced SIMD operand shapes* on page F2-4102.
- *Data type specifiers* on page F2-4103.
- *Register specifiers* on page F2-4104.
- *Register lists* on page F2-4105.
- *Register encoding* on page F2-4105.
- *Advanced SIMD scalars* on page F2-4106.

Note

The Advanced SIMD architecture, its associated implementations, and supporting software, are commonly referred to as NEON™ technology.

F2.9.1 Advanced SIMD and floating-point instruction syntax

Advanced SIMD and floating-point instructions use the general conventions of the T32 and A32 instruction sets.

Advanced SIMD and floating-point data-processing instructions use the following general format:

```
V{<modifier>}<operation>{<shape>}{<c>}{<q>}{.<dt>}  {<dest>,} <src1>, <src2>
```

All Advanced SIMD and floating-point instructions begin with a `V`. This distinguishes Advanced SIMD vector and floating-point instructions from scalar instructions.

The main operation is specified in the `<operation>` field. It is usually a three letter mnemonic the same as or similar to the corresponding scalar integer instruction.

The `<c>` and `<q>` fields are standard assembler syntax fields. For details, see *Standard assembler syntax fields* on page F2-4080.

F2.9.2 The Advanced SIMD addressing mode

All the element and structure load/store instructions use this addressing mode. There is a choice of three formats:

`[<Rn>{:<align>}]`

The address is contained in general-purpose register Rn.

Rn is not updated by this instruction.

Encoded as Rm = `0b1111`.

If Rn is encoded as `0b1111`, the instruction is CONSTRAINED UNPREDICTABLE.

`[<Rn>{:<align>}]!`

The address is contained in general-purpose register Rn.

Rn is updated by this instruction: `Rn = Rn + transfer_size`

Encoded as Rm = `0b1101`.

`transfer_size` is the number of bytes transferred by the instruction. This means that, after the instruction is executed, Rn points to the address in memory immediately following the last address loaded from or stored to.

If Rn is encoded as `0b1111`, the instruction is CONSTRAINED UNPREDICTABLE.

This addressing mode can also be written as:

`[<Rn>{:align}], #<transfer_size>`

However, disassembly produces the `[<Rn>{:align}]!` form.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`[<Rn>{:<align>}], <Rm>`

The address is contained in general-purpose register `<Rn>`.

Rn is updated by this instruction: `Rn = Rn + Rm`

Encoded as Rm = Rm. Rm must not be encoded as `0b1111` or `0b1101`, the PC or the SP.

If Rn is encoded as `0b1111`, the instruction is CONSTRAINED UNPREDICTABLE.

The CONSTRAINED UNPREDICTABLE behavior of encodings where Rn is 0b1111 is described in the section: *Using R15* on page K1-7831.

In all cases, `<align>` specifies an alignment, as specified by the individual instruction descriptions.

Previous versions of the manual used the `@` character for alignment. So, for example, the first format in this section was shown as `[<Rn>{@<align>}]`. Both `@` and `:` are supported. However, to ensure portability of code to assemblers that treat `@` as a comment character, `:` is preferred.

F2.9.3 Advanced SIMD instruction modifiers

The `<modifier>` field provides additional variants of some instructions. Table F2-7 provides definitions of the modifiers. Modifiers are not available for every instruction.

Table F2-7 Advanced SIMD instruction modifiers

<modifier>	Meaning
Q	The operation uses saturating arithmetic.
R	The operation performs rounding.
D	The operation doubles the result (before accumulation, if any).
H	The operation halves the result.

F2.9.4 Advanced SIMD operand shapes

The `<shape>` field provides additional variants of some instructions. Table F2-8 provides definitions of the shapes. Operand shapes are not available for every instruction.

Table F2-8 Advanced SIMD operand shapes

<shape>	Meaning	Typical register shape	
(none)	The operands and result are all the same width.	Dd, Dn, Dm	Qd, Qn, Qm
L	Long operation - result is twice the width of both operands	Qd, Dn, Dm	
N	Narrow operation - result is half the width of both operands	Dd, Qn, Qm	
W	Wide operation - result and first operand are twice the width of the second operand	Qd, Qn, Dm	

Note

- Some assemblers support a Q shape specifier, that requires all operands to be Q registers. An example of using this specifier is `VADDQ.S32 q0, q1, q2`. This is not standard UAL, and Arm recommends that programmers do not use a Q shape specifier.
- A disassembler must not generate any shape specifier not shown in Table F2-8.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F2.9.5 Data type specifiers

The `<dt>` field normally contains one data type specifier. Unless the assembler syntax description for the instruction indicates otherwise, this indicates the data type contained in:

- The second operand, if any.
- The operand, if there is no second operand.
- The result, if there are no operand registers.

The data types of the other operand and result are implied by the `<dt>` field combined with the instruction shape. For information about data type formats see *Data types supported by the Advanced SIMD implementation* on page E1-3962.

In the instruction syntax descriptions in Chapter F2 *About the T32 and A32 Instruction Descriptions*, the `<dt>` field is usually specified as a single field. However, where more convenient, it is sometimes specified as a concatenation of two fields, `<type><size>`.

Syntax flexibility

There is some flexibility in the data type specifier syntax:

- Software can specify three data types, specifying the result and both operand data types. For example:

  `VSUBW.I16.I16.S8 Q3, Q5, D0` instead of `VSUBW.S8 Q3, Q5, D0`

- Software can specify two data types, specifying the data types of the two operands. The data type of the result is implied by the instruction shape. For example:

  `VSUBW.I16.S8 Q3, Q5, D0` instead of `VSUBW.S8 Q3, Q5, D0`

- Software can specify two data types, specifying the data types of the single operand and the result. For example:

  `VMOVN.I16.I32 D0, Q1` instead of `VMOVN.I32 D0, Q1`

- Where an instruction requires a less specific data type, software can instead specify a more specific type, as shown in Table F2-9.
- Where an instruction does not require a data type, software can provide one.
- The `F32` data type can be abbreviated to `F`.
- The `F64` data type can be abbreviated to `D`.

In all cases, if software provides additional information, the additional information must match the instruction shape. Disassembly does not regenerate this additional information.

Table F2-9 Data type specification flexibility

Specified data type	Permitted more specific data types				
None	Any				
`.I<size>`	-	`.S<size>`	`.U<size>`	-	-
`.8`	`.I8`	`.S8`	`.U8`	`.P8`	-
`.16`	`.I16`	`.S16`	`.U16`	`.P16`	`.F16`
`.32`	`.I32`	`.S32`	`.U32`	-	`.F32` or `.F`
`.64`	`.I64`	`.S64`	`.U64`	-	`.F64` or `.D`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F2.9.6 Register specifiers

The `<dest>`, `<src1>`, and `<src2>` fields contain register specifiers, or in some cases scalar specifiers or register lists. Table F2-10 shows the register and scalar specifier formats that appear in the instruction descriptions.

If `<dest>` is omitted, it is the same as `<src1>`.

Table F2-10 Advanced SIMD and floating-point register specifier formats

<specifier>	Usual meaning [a]	Used in
`<Qd>`	A quadword destination register for the result vector.	Advanced SIMD
`<Qn>`	A quadword source register for the first operand vector.	Advanced SIMD
`<Qm>`	A quadword source register for the second operand vector.	Advanced SIMD
`<Dd>`	A doubleword destination register for the result vector.	Both
`<Dn>`	A doubleword source register for the first operand vector.	Both
`<Dm>`	A doubleword source register for the second operand vector.	Both
`<Sd>`	A singleword destination register for the result vector.	Floating-point
`<Sn>`	A singleword source register for the first operand vector.	Floating-point
`<Sm>`	A singleword source register for the second operand vector.	Floating-point
`<Dd[x]>`	A destination scalar for the result. Element x of vector `<Dd>`.	Advanced SIMD
`<Dn[x]>`	A source scalar for the first operand. Element x of vector `<Dn>`.	Both [b]
`<Dm[x]>`	A source scalar for the second operand. Element x of vector `<Dm>`.	Advanced SIMD
`<Rt>`	A general-purpose register, used for a source or destination address.	Both
`<Rt2>`	A general-purpose register, used for a source or destination address.	Both
`<Rn>`	A general-purpose register, used as a load or store base address.	Both
`<Rm>`	A general-purpose register, used as a post-indexed address source.	Both

a. In some instructions the roles of registers are different.

b. In the floating-point instructions, `<Dn[x]>` is used only in `VMOV` (scalar to general-purpose register), see *VMOV (scalar to general-purpose register)* on page F6-5335.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F2.9.7 Register lists

A register list is a list of register specifiers separated by commas and enclosed in brackets { and }. There are restrictions on what registers can appear in a register list. These restrictions are described in the individual instruction descriptions. Table F2-11 shows some register list formats, with examples of actual register lists corresponding to those formats.

Note

Register lists must not wrap around the end of the register bank.

Syntax flexibility

There is some flexibility in the register list syntax:

- Where a register list contains consecutive registers, they can be specified as a range, instead of listing every register, for example `{D0-D3}` instead of `{D0, D1, D2, D3}`.
- Where a register list contains an even number of consecutive doubleword registers starting with an even numbered register, it can be written as a list of quadword registers instead, for example `{Q1, Q2}` instead of `{D2-D5}`.
- Where a register list contains only one register, the enclosing braces can be omitted, for example `VLD1.8 D0, [R0]` instead of `VLD1.8 {D0}, [R0]`.

Table F2-11 Example register lists

Format	Example	Alternative
`{<Dd>}`	`{D3}`	`D3`
`{<Dd>, <Dd+1>, <Dd+2>}`	`{D3, D4, D5}`	`{D3-D5}`
`{<Dd[x]>, <Dd+2[x]}`	`{D0[3], D2[3]}`	-
`{<Dd[]>}`	`{D7[]}`	`D7[]`

F2.9.8 Register encoding

An Advanced SIMD register is either:

- *Quadword*, meaning it is 128 bits wide.
- *Doubleword*, meaning it is 64 bits wide.

Some instructions have options for either doubleword or quadword registers. This is normally encoded in Q, bit[6], as Q = 0 for doubleword operations, or Q = 1 for quadword operations.

A floating-point register is either:

- Double-precision, meaning it is 64 bits wide.
- Single-precision, meaning it is 32 bits wide.

This is encoded in the sz field, bit[8], as sz = 1 for double-precision operations, or sz = 0 for single-precision operations.

The T32 instruction encoding of Advanced SIMD or floating-point registers is:

15 14 13 12 11 10 9 8 7	6	5 4 3 2 1 0	15 14 13 12 11 10 9	8	7	6	5	4	3 2 1 0
	D	Vn	Vd	sz	N	Q	M		Vm

The A32 instruction encoding of Advanced SIMD or floating-point registers is:

31 30 29 28 27 26 25 24 23	22	21 20 19 18	17 16 15 14 13 12	11 10 9	8	7	6	5	4	3 2 1 0
	D	Vn	Vd		sz	N	Q	M		Vm

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Some instructions use only one or two registers, and use the unused register fields as additional opcode bits.

Table F2-12 shows the encodings for the registers.

Table F2-12 Encoding of register numbers

Register mnemonic	Usual usage	Register number encoded in[a]	Notes[a]	Used in
`<Qd>`	Destination (quadword)	D, Vd (bits[22, 15:13])	bit[12] == 0[b]	Advanced SIMD
`<Qn>`	First operand (quadword)	N, Vn (bits[7, 19:17])	bit[16] == 0[b]	Advanced SIMD
`<Qm>`	Second operand (quadword)	M, Vm (bits[5, 3:1])	bit[0] == 0[b]	Advanced SIMD
`<Dd>`	Destination (doubleword)	D, Vd (bits[22, 15:12])	-	Both
`<Dn>`	First operand (doubleword)	N, Vn (bits[7, 19:16])	-	Both
`<Dm>`	Second operand (doubleword)	M, Vm (bits[5, 3:0])	-	Both
`<Sd>`	Destination (single-precision)	Vd, D (bits[15:12, 22])	-	Floating-point
`<Sn>`	First operand (single-precision)	Vn, N (bits[19:16, 7])	-	Floating-point
`<Sm>`	Second operand (single-precision)	Vm, M (bits[3:0, 5])	-	Floating-point

a. Bit numbers given for the A32 instruction encoding. See the figures in this section for the equivalent bits in the T32 encoding.
b. If this bit is 1, the instruction is UNDEFINED.

F2.9.9 Advanced SIMD scalars

Advanced SIMD scalars can be 8-bit, 16-bit, 32-bit, or 64-bit. Instructions other than multiply instructions can access any element in the register set. The instruction syntax refers to the scalars using an index into a doubleword vector. The descriptions of the individual instructions contain details of the encodings.

Table F2-13 shows the form of encoding for scalars used in multiply instructions. These instructions cannot access scalars in some registers. The descriptions of the individual instructions contain cross references to this section where appropriate.

32-bit Advanced SIMD scalars, when used as single-precision floating-point numbers, are equivalent to Floating-point single-precision registers. That is, `Dm[x]` in a 32-bit context (0 <= `m` <= 15, 0 <= `x` <=1) is equivalent to `S[2m + x]`.

Table F2-13 Encoding of scalars in multiply instructions

Scalar mnemonic	Usual usage	Scalar size	Register specifier	Index specifier	Accessible registers
`<Dm[x]>`	Second operand	16-bit	Vm[2:0]	M, Vm[3]	D0-D7
		32-bit	Vm[3:0]	M	D0-D15

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter F3
T32 Instruction Set Encoding

This chapter describes the encoding of the T32 instruction set. It contains the following sections:

- *T32 instruction set encoding* on page F3-4108.
- *About the T32 Advanced SIMD and floating-point instructions and their encoding* on page F3-4178.

In this chapter:

- In the decode tables, an entry of - for a field value means the value of the field does not affect the decoding.
- In the decode diagrams, a shaded field indicates that the bits in that field are not used in that level of decode.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F3.1 T32 instruction set encoding

The T32 instruction stream is a sequence of halfword-aligned halfwords. Each T32 instruction is either a single 16-bit halfword in that stream, or a 32-bit instruction consisting of two consecutive halfwords in that stream.

If the value of bits[15:11] of the halfword being decoded is one of the following, the halfword is the first halfword of a 32-bit instruction:

- `0b11101`.
- `0b11110`.
- `0b11111`.

Otherwise, the halfword is a 16-bit instruction.

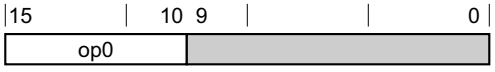
The T32 instruction encoding is:

Table F3-1 Main encoding table for the T32 instruction set

Decode fields		Decode group or instruction page
op0	op1	
`!= 111`	`-`	*16-bit*
`111`	`00`	B - T2 variant
`111`	`!= 00`	*32-bit* on page F3-4119

F3.1.1 16-bit

This section describes the encoding of the 16-bit group. The encodings in this section are decoded from *T32 instruction set encoding*.



This decode also imposes the constraint:

- `op0<5:3> != 111`.

Table F3-2 Encoding table for the 16-bit group

Decode fields	Decode group or instruction page
op0	
`00xxxx`	*Shift (immediate), add, subtract, move, and compare* on page F3-4112
`010000`	*Data-processing (two low registers)* on page F3-4109
`010001`	*Special data instructions and branch and exchange* on page F3-4114
`01001x`	LDR (literal) - *T1 variant* on page F5-4417
`0101xx`	*Load/store (register offset)* on page F3-4110

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table F3-2 Encoding table for the 16-bit group (continued)

Decode fields op0	Decode group or instruction page
`011xxx`	*Load/store word/byte (immediate offset)* on page F3-4110
`1000xx`	*Load/store halfword (immediate offset)* on page F3-4111
`1001xx`	*Load/store (SP-relative)* on page F3-4111
`1010xx`	*Add PC/SP (immediate)* on page F3-4111
`1011xx`	*Miscellaneous 16-bit instructions* on page F3-4115
`1100xx`	*Load/store multiple* on page F3-4112
`1101xx`	*Conditional branch, and Supervisor Call* on page F3-4118

Data-processing (two low registers)

This section describes the encoding of the Data-processing (two low registers) instruction class. The encodings in this section are decoded from *16-bit* on page F3-4108.

15	14	13	12	11	10	9 – 6	5 – 3	2 – 0
0	1	0	0	0	0	op	Rs	Rd

Decode fields op	Instruction page
`0000`	AND, ANDS (register)
`0001`	EOR, EORS (register)
`0010`	MOV, MOVS (register-shifted register) - Logical shift left variant
`0011`	MOV, MOVS (register-shifted register) - Logical shift right variant
`0100`	MOV, MOVS (register-shifted register) - Arithmetic shift right variant
`0101`	ADC, ADCS (register)
`0110`	SBC, SBCS (register)
`0111`	MOV, MOVS (register-shifted register) - Rotate right variant
`1000`	TST (register)
`1001`	RSB, RSBS (immediate)
`1010`	CMP (register)
`1011`	CMN (register)
`1100`	ORR, ORRS (register)

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields op	Instruction page
1101	MUL, MULS
1110	BIC, BICS (register)
1111	MVN, MVNS (register)

Load/store (register offset)

This section describes the encoding of the Load/store (register offset) instruction class. The encodings in this section are decoded from *16-bit* on page F3-4108.

15 14 13 12	11	10	9	8	6 5	3 2 0
0 1 0 1	L	B	H	Rm	Rn	Rt

Decode fields L	B	H	Instruction page
0	0	0	STR (register)
0	0	1	STRH (register)
0	1	0	STRB (register)
0	1	1	LDRSB (register)
1	0	0	LDR (register)
1	0	1	LDRH (register)
1	1	0	LDRB (register)
1	1	1	LDRSH (register)

Load/store word/byte (immediate offset)

This section describes the encoding of the Load/store word/byte (immediate offset) instruction class. The encodings in this section are decoded from *16-bit* on page F3-4108.

15 14 13 12	11	10	6 5	3 2 0
0 1 1	B	L	imm5	Rn	Rt

Decode fields B	L	Instruction page
0	0	STR (immediate)

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields		Instruction page
B	L	
0	1	LDR (immediate)
1	0	STRB (immediate)
1	1	LDRB (immediate)

Load/store halfword (immediate offset)

This section describes the encoding of the Load/store halfword (immediate offset) instruction class. The encodings in this section are decoded from *16-bit* on page F3-4108.

15 14 13 12	11	10 … 6	5 … 3	2 … 0
1 0 0 0	L	imm5	Rn	Rt

Decode fields	Instruction page
L	
0	STRH (immediate)
1	LDRH (immediate)

Load/store (SP-relative)

This section describes the encoding of the Load/store (SP-relative) instruction class. The encodings in this section are decoded from *16-bit* on page F3-4108.

15 14 13 12	11	10 … 8	7 … 0
1 0 0 1	L	Rt	imm8

Decode fields	Instruction page
L	
0	STR (immediate)
1	LDR (immediate)

Add PC/SP (immediate)

This section describes the encoding of the Add PC/SP (immediate) instruction class. The encodings in this section are decoded from *16-bit* on page F3-4108.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

15 14 13 12	11	10 8	7 0
1 0 1 0	SP	Rd	imm8

Decode fields	Instruction page
SP	
0	ADR
1	ADD, ADDS (SP plus immediate)

Load/store multiple

This section describes the encoding of the Load/store multiple instruction class. The encodings in this section are decoded from *16-bit* on page F3-4108.

15 14 13 12	11	10 8	7 0
1 1 0 0	L	Rn	register_list

Decode fields	Instruction page
L	
0	STM, STMIA, STMEA
1	LDM, LDMIA, LDMFD

F3.1.2 Shift (immediate), add, subtract, move, and compare

This section describes the encoding of the Shift (immediate), add, subtract, move, and compare group. The encodings in this section are decoded from *16-bit* on page F3-4108.

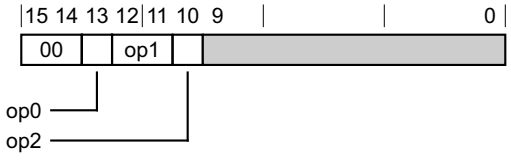


Table F3-3 Encoding table for the Shift (immediate), add, subtract, move, and compare group

Decode fields			Decode group or instruction page
op0	op1	op2	
0	11	0	*Add, subtract (three low registers)* on page F3-4113
0	11	1	*Add, subtract (two low registers and immediate)* on page F3-4113
0	!= 11	-	MOV, MOVS (register) - *T2 variant* on page F5-4518
1	-	-	*Add, subtract, compare, move (one low register and immediate)* on page F3-4113

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Add, subtract (three low registers)

This section describes the encoding of the Add, subtract (three low registers) instruction class. The encodings in this section are decoded from *Shift (immediate), add, subtract, move, and compare* on page F3-4112.

15	14	13	12	11	10	9	8	6	5	3	2	0
0	0	0	1	1	0	S	Rm		Rn		Rd	

Decode fields S	Instruction page
0	ADD, ADDS (register)
1	SUB, SUBS (register)

Add, subtract (two low registers and immediate)

This section describes the encoding of the Add, subtract (two low registers and immediate) instruction class. The encodings in this section are decoded from *Shift (immediate), add, subtract, move, and compare* on page F3-4112.

15	14	13	12	11	10	9	8	6	5	3	2	0
0	0	0	1	1	1	S	imm3		Rn		Rd	

Decode fields S	Instruction page
0	ADD, ADDS (immediate)
1	SUB, SUBS (immediate)

Add, subtract, compare, move (one low register and immediate)

This section describes the encoding of the Add, subtract, compare, move (one low register and immediate) instruction class. The encodings in this section are decoded from *Shift (immediate), add, subtract, move, and compare* on page F3-4112.

15	14	13	12	11	10	8	7	0
0	0	1	op		Rd		imm8	

Decode fields op	Instruction page
00	MOV, MOVS (immediate)

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields op	Instruction page
01	CMP (immediate)
10	ADD, ADDS (immediate)
11	SUB, SUBS (immediate)

F3.1.3 Special data instructions and branch and exchange

This section describes the encoding of the Special data instructions and branch and exchange group. The encodings in this section are decoded from *16-bit* on page F3-4108.

15	10 9 8	7 0
010001	op0	

Table F3-4 Encoding table for the Special data instructions and branch and exchange group

Decode fields op0	Decode group or instruction page
11	*Branch and exchange*
!= 11	*Add, subtract, compare, move (two high registers)*

Branch and exchange

This section describes the encoding of the Branch and exchange instruction class. The encodings in this section are decoded from *Special data instructions and branch and exchange*.

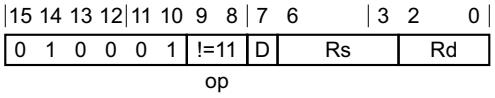
15 14 13 12	11 10 9 8	7	6 3	2	1	0
0 1 0 0	0 1 1 1	L	Rm	(0)	(0)	(0)

Decode fields L	Instruction page
0	BX
1	BLX (register)

Add, subtract, compare, move (two high registers)

This section describes the encoding of the Add, subtract, compare, move (two high registers) instruction class. The encodings in this section are decoded from *Special data instructions and branch and exchange*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta



Decode fields			Instruction page
op	D:Rd	Rs	
00	!= 1101	!= 1101	ADD, ADDS (register)
00	-	1101	ADD, ADDS (SP plus register) - *T1* on page F5-4266
00	1101	!= 1101	ADD, ADDS (SP plus register) - *T2* on page F5-4266
01	-	-	CMP (register)
10	-	-	MOV, MOVS (register)

F3.1.4 Miscellaneous 16-bit instructions

This section describes the encoding of the Miscellaneous 16-bit instructions group. The encodings in this section are decoded from *16-bit* on page F3-4108.

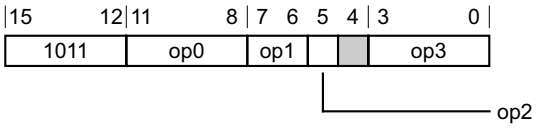


Table F3-5 Encoding table for the Miscellaneous 16-bit instructions group

Decode fields				Decode group or instruction page	Architecture version
op0	op1	op2	op3		
0000	-	-	-	*Adjust SP (immediate)* on page F3-4116	-
0010	-	-	-	*Extend* on page F3-4116	-
0110	00	0	-	SETPAN	Armv8.1
0110	00	1	-	Unallocated.	-
0110	01	-	-	*Change Processor State* on page F3-4116	-
0110	1x	-	-	Unallocated.	-
0111	-	-	-	Unallocated.	-
1000	-	-	-	Unallocated.	-
1010	10	-	-	HLT	-
1010	!= 10	-	-	*Reverse bytes* on page F3-4117	-
1110	-	-	-	BKPT	-
1111	-	-	0000	*Hints* on page F3-4117	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table F3-5 Encoding table for the Miscellaneous 16-bit instructions group (continued)

Decode fields				Decode group or instruction page	Architecture version
op0	op1	op2	op3		
1111	-	-	!= 0000	IT	-
x0x1	-	-	-	CBNZ, CBZ	-
x10x	-	-	-	*Push and Pop* on page F3-4118	-

Adjust SP (immediate)

This section describes the encoding of the Adjust SP (immediate) instruction class. The encodings in this section are decoded from *Miscellaneous 16-bit instructions* on page F3-4115.

15	14	13	12	11	10	9	8	7	6		0
1	0	1	1	0	0	0	0	S	imm7		

Decode fields	Instruction page
S	
0	ADD, ADDS (SP plus immediate)
1	SUB, SUBS (SP minus immediate)

Extend

This section describes the encoding of the Extend instruction class. The encodings in this section are decoded from *Miscellaneous 16-bit instructions* on page F3-4115.

15	14	13	12	11	10	9	8	7	6	5	3	2	0
1	0	1	1	0	0	1	0	U	B	Rm		Rd	

Decode fields		Instruction page
U	B	
0	0	SXTH
0	1	SXTB
1	0	UXTH
1	1	UXTB

Change Processor State

This section describes the encoding of the Change Processor State instruction class. The encodings in this section are decoded from *Miscellaneous 16-bit instructions* on page F3-4115.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

15	14	13	12	11	10	9	8	7	6	5	4	0
1	0	1	1	0	1	1	0	0	1	op	flags	

Decode fields		Instruction page
op	flags	
0	-	SETEND
1	-	CPS, CPSID, CPSIE

Reverse bytes

This section describes the encoding of the Reverse bytes instruction class. The encodings in this section are decoded from *Miscellaneous 16-bit instructions* on page F3-4115.

15	14	13	12	11	10	9	8	7	6	5	3	2	0
1	0	1	1	1	0	1	0	!=10		Rm		Rd	
								op					

Decode fields	Instruction page
op	
00	REV
01	REV16
11	REVSH

Hints

This section describes the encoding of the Hints instruction class. The encodings in this section are decoded from *Miscellaneous 16-bit instructions* on page F3-4115.

15	14	13	12	11	10	9	8	7	4	3	2	1	0
1	0	1	1	1	1	1	1	hint		0	0	0	0

Decode fields	Instruction page
hint	
0000	NOP
0001	YIELD
0010	WFE
0011	WFI
0100	SEV

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields hint	Instruction page
0101	SEVL
011x	Reserved hint, behaves as NOP.
1xxx	Reserved hint, behaves as NOP.

Push and Pop

This section describes the encoding of the Push and Pop instruction class. The encodings in this section are decoded from *Miscellaneous 16-bit instructions* on page F3-4115.

15 14 13 12	11	10	9	8	7 … 0
1 0 1 1	L	1	0	P	register_list

Decode fields L	Instruction page
0	PUSH
1	POP

F3.1.5 Conditional branch, and Supervisor Call

This section describes the encoding of the Conditional branch, and Supervisor Call group. The encodings in this section are decoded from *16-bit* on page F3-4108.

15 … 12	11 … 8	7 … 0
1101	op0	

Table F3-6 Encoding table for the Conditional branch, and Supervisor Call group

Decode fields op0	Decode group or instruction page
111x	*Exception generation*
!= 111x	B - *T1 variant* on page F5-4291

Exception generation

This section describes the encoding of the Exception generation instruction class. The encodings in this section are decoded from *Conditional branch, and Supervisor Call*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

15	14	13	12	11	10	9	8	7 … 0
1	1	0	1	1	1	1	S	imm8

Decode fields	Instruction page
S	
0	UDF
1	SVC

F3.1.6 32-bit

This section describes the encoding of the 32-bit group. The encodings in this section are decoded from *T32 instruction set encoding* on page F3-4108.

15 … 13	12 … 9	8 … 4	3 … 0	15	14 … 0
111	op0	op1			

op3

This decode also imposes the constraint:

- `op0<3:2> != 00`.

Table F3-7 Encoding table for the 32-bit group

Decode fields			Decode group or instruction page
op0	**op1**	**op3**	
x11x	-	-	*System register access, Advanced SIMD, and floating-point* on page F3-4124
0100	xx0xx	-	*Load/store multiple* on page F3-4120
0100	xx1xx	-	*Load/store dual, load/store exclusive, load-acquire/store-release, and table branch* on page F3-4150
0101	-	-	*Data-processing (shifted register)* on page F3-4120
10xx	-	1	*Branches and miscellaneous control* on page F3-4154
10x0	-	0	*Data-processing (modified immediate)* on page F3-4122
10x1	-	0	*Data-processing (plain binary immediate)* on page F3-4158
1100	1xxx0	-	*Advanced SIMD element or structure load/store* on page F3-4160
1100	!= 1xxx0	-	*Load/store single* on page F3-4163
1101	0xxxx	-	*Data-processing (register)* on page F3-4171
1101	10xxx	-	*Multiply, multiply accumulate, and absolute difference* on page F3-4175
1101	11xxx	-	*Long multiply and divide* on page F3-4123

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Load/store multiple

This section describes the encoding of the Load/store multiple instruction class. The encodings in this section are decoded from *32-bit* on page F3-4119.

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15	14	13 … 0
1	1	1	0	1	0	0	opc	0	W	L		Rn	P	M	register_list

Decode fields		Instruction page
opc	L	
00	0	SRS, SRSDA, SRSDB, SRSIA, SRSIB - *T1* on page F5-4729
00	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB - *T1* on page F5-4624
01	0	STM, STMIA, STMEA
01	1	LDM, LDMIA, LDMFD
10	0	STMDB, STMFD
10	1	LDMDB, LDMEA
11	0	SRS, SRSDA, SRSDB, SRSIA, SRSIB - *T2* on page F5-4730
11	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB - *T2* on page F5-4625

Data-processing (shifted register)

This section describes the encoding of the Data-processing (shifted register) instruction class. The encodings in this section are decoded from *32-bit* on page F3-4119.

15	14	13	12	11	10	9	8 … 5	4	3 … 0	15	14 … 12	11 … 8	7	6	5	4	3 … 0
1	1	1	0	1	0	1	op1	S	Rn	(0)	imm3	Rd	imm2		stype		Rm

Decode fields					Instruction page
op1	S	Rn	imm3:imm2:stype	Rd	
0000	0	-	-	-	AND, ANDS (register) - AND, rotate right with extend variant
0000	1	-	!= 0000011	!= 1111	AND, ANDS (register) - ANDS, shift or rotate by value variant
0000	1	-	!= 0000011	1111	TST (register) - Shift or rotate by value variant
0000	1	-	0000011	!= 1111	AND, ANDS (register) - ANDS, rotate right with extend variant
0000	1	-	0000011	1111	TST (register) - Rotate right with extend variant
0001	-	-	-	-	BIC, BICS (register)
0010	0	!= 1111	-	-	ORR, ORRS (register) - ORR, rotate right with extend variant
0010	0	1111	-	-	MOV, MOVS (register) - MOV, rotate right with extend variant
0010	1	!= 1111	-	-	ORR, ORRS (register) - ORRS, rotate right with extend variant

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields					Instruction page
op1	S	Rn	imm3:imm2:stype	Rd	
0010	1	1111	-	-	MOV, MOVS (register) - MOVS, rotate right with extend variant
0011	0	!= 1111	-	-	ORN, ORNS (register) - ORN, rotate right with extend variant
0011	0	1111	-	-	MVN, MVNS (register) - MVN, rotate right with extend variant
0011	1	!= 1111	-	-	ORN, ORNS (register) - ORNS, rotate right with extend variant
0011	1	1111	-	-	MVN, MVNS (register) - MVNS, rotate right with extend variant
0100	0	-	-	-	EOR, EORS (register) - EOR, rotate right with extend variant
0100	1	-	!= 0000011	!= 1111	EOR, EORS (register) - EORS, shift or rotate by value variant
0100	1	-	!= 0000011	1111	TEQ (register) - Shift or rotate by value variant
0100	1	-	0000011	!= 1111	EOR, EORS (register) - EORS, rotate right with extend variant
0100	1	-	0000011	1111	TEQ (register) - Rotate right with extend variant
0101	-	-	-	-	Unallocated.
0110	0	-	xxxxx00	-	PKHBT, PKHTB - PKHBT variant
0110	0	-	xxxxx01	-	Unallocated.
0110	0	-	xxxxx10	-	PKHBT, PKHTB - PKHTB variant
0110	0	-	xxxxx11	-	Unallocated.
0111	-	-	-	-	Unallocated.
1000	0	!= 1101	-	-	ADD, ADDS (register) - ADD, rotate right with extend variant
1000	0	1101	-	-	ADD, ADDS (SP plus register) - ADD, rotate right with extend variant
1000	1	!= 1101	-	!= 1111	ADD, ADDS (register) - ADDS, rotate right with extend variant
1000	1	1101	-	!= 1111	ADD, ADDS (SP plus register) - ADDS, rotate right with extend variant
1000	1	-	-	1111	CMN (register)
1001	-	-	-	-	Unallocated.
1010	-	-	-	-	ADC, ADCS (register)
1011	-	-	-	-	SBC, SBCS (register)
1100	-	-	-	-	Unallocated.
1101	0	!= 1101	-	-	SUB, SUBS (register) - SUB, rotate right with extend variant
1101	0	1101	-	-	SUB, SUBS (SP minus register) - SUB, rotate right with extend variant
1101	1	!= 1101	-	!= 1111	SUB, SUBS (register) - SUBS, rotate right with extend variant
1101	1	1101	-	!= 1111	SUB, SUBS (SP minus register) - SUBS, rotate right with extend variant
1101	1	-	-	1111	CMP (register)
1110	-	-	-	-	RSB, RSBS (register)
1111	-	-	-	-	Unallocated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Data-processing (modified immediate)

This section describes the encoding of the Data-processing (modified immediate) instruction class. The encodings in this section are decoded from *32-bit* on page F3-4119.

15 14 13 12	11	10	9	8 | 5	4	3 0	15	14 12	11 8	7 0
1 1 1 1 0	i	0		op1	S	Rn	0	imm3	Rd	imm8

Decode fields				Instruction page
op1	S	Rn	Rd	
0000	0	-	-	AND, ANDS (immediate) - AND variant
0000	1	-	!= 1111	AND, ANDS (immediate) - ANDS variant
0000	1	-	1111	TST (immediate)
0001	-	-	-	BIC, BICS (immediate)
0010	0	!= 1111	-	ORR, ORRS (immediate) - ORR variant
0010	0	1111	-	MOV, MOVS (immediate) - MOV variant
0010	1	!= 1111	-	ORR, ORRS (immediate) - ORRS variant
0010	1	1111	-	MOV, MOVS (immediate) - MOVS variant
0011	0	!= 1111	-	ORN, ORNS (immediate) - Not flag setting variant
0011	0	1111	-	MVN, MVNS (immediate) - MVN variant
0011	1	!= 1111	-	ORN, ORNS (immediate) - Flag setting variant
0011	1	1111	-	MVN, MVNS (immediate) - MVNS variant
0100	0	-	-	EOR, EORS (immediate) - EOR variant
0100	1	-	!= 1111	EOR, EORS (immediate) - EORS variant
0100	1	-	1111	TEQ (immediate)
0101	-	-	-	Unallocated.
011x	-	-	-	Unallocated.
1000	0	!= 1101	-	ADD, ADDS (immediate) - ADD variant
1000	0	1101	-	ADD, ADDS (SP plus immediate) - ADD variant
1000	1	!= 1101	!= 1111	ADD, ADDS (immediate) - ADDS variant
1000	1	1101	!= 1111	ADD, ADDS (SP plus immediate) - ADDS variant
1000	1	-	1111	CMN (immediate)
1001	-	-	-	Unallocated.
1010	-	-	-	ADC, ADCS (immediate)
1011	-	-	-	SBC, SBCS (immediate)
1100	-	-	-	Unallocated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields				Instruction page
op1	S	Rn	Rd	
1101	0	!= 1101	-	SUB, SUBS (immediate) - SUB variant
1101	0	1101	-	SUB, SUBS (SP minus immediate) - SUB variant
1101	1	!= 1101	!= 1111	SUB, SUBS (immediate) - SUBS variant
1101	1	1101	!= 1111	SUB, SUBS (SP minus immediate) - SUBS variant
1101	1	-	1111	CMP (immediate)
1110	-	-	-	RSB, RSBS (immediate)
1111	-	-	-	Unallocated.

Long multiply and divide

This section describes the encoding of the Long multiply and divide instruction class. The encodings in this section are decoded from *32-bit* on page F3-4119.

15 14 13 12	11 10 9 8	7 6 4	3 0	15 12	11 8	7 4	3 0
1 1 1 1	1 0 1 1	1 op1	Rn	RdLo	RdHi	op2	Rm

Decode fields		Instruction page
op1	op2	
000	!= 0000	Unallocated.
000	0000	SMULL, SMULLS
001	!= 1111	Unallocated.
001	1111	SDIV
010	!= 0000	Unallocated.
010	0000	UMULL, UMULLS
011	!= 1111	Unallocated.
011	1111	UDIV
100	0000	SMLAL, SMLALS
100	0001	Unallocated.
100	001x	Unallocated.
100	01xx	Unallocated.
100	1000	SMLALBB, SMLALBT, SMLALTB, SMLALTT - SMLALBB variant
100	1001	SMLALBB, SMLALBT, SMLALTB, SMLALTT - SMLALBT variant
100	1010	SMLALBB, SMLALBT, SMLALTB, SMLALTT - SMLALTB variant
100	1011	SMLALBB, SMLALBT, SMLALTB, SMLALTT - SMLALTT variant

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields		Instruction page
op1	op2	
100	1100	SMLALD, SMLALDX - SMLALD variant
100	1101	SMLALD, SMLALDX - SMLALDX variant
100	111x	Unallocated.
101	0xxx	Unallocated.
101	10xx	Unallocated.
101	1100	SMLSLD, SMLSLDX - SMLSLD variant
101	1101	SMLSLD, SMLSLDX - SMLSLDX variant
101	111x	Unallocated.
110	0000	UMLAL, UMLALS
110	0001	Unallocated.
110	001x	Unallocated.
110	010x	Unallocated.
110	0110	UMAAL
110	0111	Unallocated.
110	1xxx	Unallocated.
111	-	Unallocated.

F3.1.7 System register access, Advanced SIMD, and floating-point

This section describes the encoding of the System register access, Advanced SIMD, and floating-point group. The encodings in this section are decoded from *32-bit* on page F3-4119.

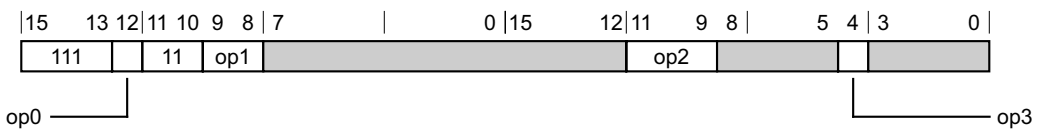


Table F3-8 Encoding table for the System register access, Advanced SIMD, and floating-point group

Decode fields				Decode group or instruction page
op0	op1	op2	op3	
-	0x	111	-	*System register load/store and 64-bit move* on page F3-4130
-	10	10x	0	*Floating-point data-processing* on page F3-4131
-	10	111	1	*System register 32-bit move* on page F3-4125
-	11	-	-	*Advanced SIMD data-processing* on page F3-4137
0	0x	10x	-	*Advanced SIMD load/store and 64-bit move* on page F3-4147

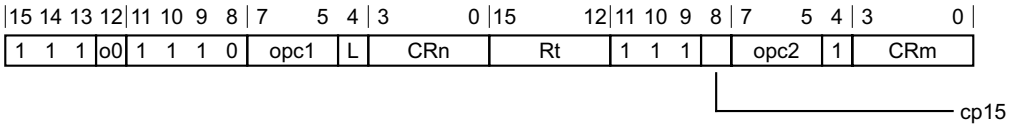
Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table F3-8 Encoding table for the System register access, Advanced SIMD, and floating-point group (continued)

Decode fields				Decode group or instruction page
op0	op1	op2	op3	
0	10	10x	1	*Advanced SIMD and floating-point 32-bit move* on page F3-4149
1	0x	1x0	-	*Advanced SIMD three registers of the same length extension*
1	10	1x0	-	*Advanced SIMD two registers and a scalar extension* on page F3-4128

System register 32-bit move

This section describes the encoding of the System register 32-bit move instruction class. The encodings in this section are decoded from *System register access, Advanced SIMD, and floating-point* on page F3-4124.

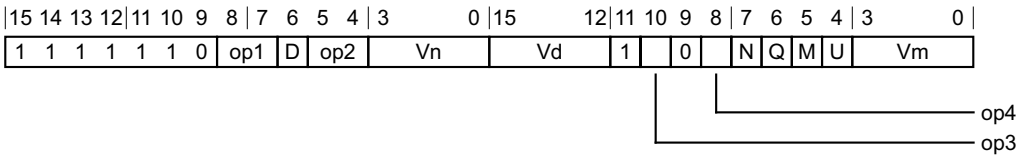


Decode fields		Instruction page
o0	L	
0	0	MCR
0	1	MRC
1	-	Unallocated.

Advanced SIMD three registers of the same length extension

This section describes the encoding of the Advanced SIMD three registers of the same length extension instruction class. The encodings in this section are decoded from *System register access, Advanced SIMD, and floating-point* on page F3-4124.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta



Decode fields						Instruction page	Architecture version
op1	op2	op3	op4	Q	U		
x1	0x	0	0	0	0	VCADD - 64-bit SIMD vector variant	Armv8.3
x1	0x	0	0	0	1	Unallocated.	-
x1	0x	0	0	1	0	VCADD - 128-bit SIMD vector variant	Armv8.3
x1	0x	0	0	1	1	Unallocated.	-
00	0x	0	0	-	-	Unallocated.	-
00	0x	0	1	-	-	Unallocated.	-
00	00	1	0	0	0	Unallocated.	-
00	00	1	0	0	1	Unallocated.	-
00	00	1	0	1	0	VMMLA	Armv8.6
00	00	1	0	1	1	Unallocated.	-
00	00	1	1	0	0	VDOT (vector) - 64-bit SIMD vector variant	Armv8.6
00	00	1	1	0	1	Unallocated.	-
00	00	1	1	1	0	VDOT (vector) - 128-bit SIMD vector variant	Armv8.6
00	00	1	1	1	1	Unallocated.	-
00	01	1	0	-	-	Unallocated.	-
00	01	1	1	-	-	Unallocated.	-
00	10	0	0	0	1	VFMAL (vector) - 64-bit SIMD vector variant	Armv8.2
00	10	0	1	-	-	Unallocated.	-
00	10	1	0	0	-	Unallocated.	-
00	10	1	0	1	0	VSMMLA	Armv8.6
00	10	1	0	1	1	VUMMLA	Armv8.6

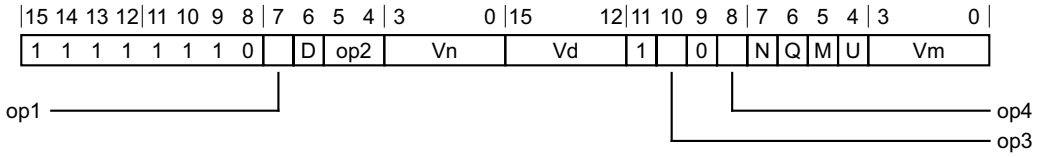
Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields						Instruction page	Architecture version
op1	op2	op3	op4	Q	U		
00	10	1	1	0	0	VSDOT (vector) - 64-bit SIMD vector variant	Armv8.2
00	10	1	1	0	1	VUDOT (vector) - 64-bit SIMD vector variant	Armv8.2
00	10	1	1	1	0	VSDOT (vector) - 128-bit SIMD vector variant	Armv8.2
00	10	1	1	1	1	VUDOT (vector) - 128-bit SIMD vector variant	Armv8.2
00	11	0	0	-	1	VFMAB, VFMAT (BFloat16, vector)	Armv8.6
00	11	0	0	1	1	VFMAL (vector) - 128-bit SIMD vector variant	Armv8.2
00	11	0	1	-	-	Unallocated.	-
00	11	1	0	-	-	Unallocated.	-
00	11	1	1	-	-	Unallocated.	-
01	10	0	0	-	1	VFMSL (vector)	Armv8.2
01	10	0	1	-	-	Unallocated.	-
01	10	1	0	0	-	Unallocated.	-
01	10	1	0	1	0	VUSMMLA	Armv8.6
01	10	1	0	1	1	Unallocated.	-
01	10	1	1	0	0	VUSDOT (vector) - 64-bit SIMD vector variant	Armv8.6
01	10	1	1	-	1	Unallocated.	-
01	10	1	1	1	0	VUSDOT (vector) - 128-bit SIMD vector variant	Armv8.6
01	11	-	-	-	-	Unallocated.	-
-	1x	0	0	-	0	VCMLA	Armv8.3
10	11	-	-	-	-	Unallocated.	-
11	11	-	-	-	-	Unallocated.	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Advanced SIMD two registers and a scalar extension

This section describes the encoding of the Advanced SIMD two registers and a scalar extension instruction class. The encodings in this section are decoded from *System register access, Advanced SIMD, and floating-point* on page F3-4124.



Decode fields						Instruction page	Architecture version
op1	op2	op3	op4	Q	U		
0	-	0	0	-	0	VCMLA (by element) - 128-bit SIMD vector of half-precision floating-point variant	Armv8.3
0	00	0	0	-	1	VFMAL (by scalar)	Armv8.2
0	00	0	1	-	-	Unallocated.	-
0	00	1	0	-	-	Unallocated.	-
0	00	1	1	0	0	VDOT (by element) - 64-bit SIMD vector variant	Armv8.6
0	00	1	1	-	1	Unallocated.	-
0	00	1	1	1	0	VDOT (by element) - 128-bit SIMD vector variant	Armv8.6
0	01	0	0	-	0	Unallocated.	-
0	01	0	0	0	1	VFMSL (by scalar) - 64-bit SIMD vector variant	Armv8.2
0	01	0	0	1	1	VFMSL (by scalar) - 128-bit SIMD vector variant	Armv8.2
0	01	0	1	-	-	Unallocated.	-
0	01	1	0	-	-	Unallocated.	-
0	10	0	-	-	-	Unallocated.	-
0	10	1	0	-	-	Unallocated.	-
0	10	1	1	0	0	VSDOT (by element) - 64-bit SIMD vector variant	Armv8.2

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields						Instruction page	Architecture version
op1	op2	op3	op4	Q	U		
0	10	1	1	0	1	VUDOT (by element) - 64-bit SIMD vector variant	Armv8.2
0	10	1	1	1	0	VSDOT (by element) - 128-bit SIMD vector variant	Armv8.2
0	10	1	1	1	1	VUDOT (by element) - 128-bit SIMD vector variant	Armv8.2
0	11	0	0	-	0	Unallocated.	-
0	11	0	0	-	1	VFMAB, VFMAT (BFloat16, by scalar)	Armv8.6
0	11	0	1	-	-	Unallocated.	-
0	11	1	-	-	-	Unallocated.	-
1	-	0	0	-	0	VCMLA (by element) - 128-bit SIMD vector of single-precision floating-point variant	Armv8.3
1	00	1	1	0	0	VUSDOT (by element) - 64-bit SIMD vector variant	Armv8.6
1	00	1	1	0	1	VSUDOT (by element) - 64-bit SIMD vector variant	Armv8.6
1	00	1	1	1	0	VUSDOT (by element) - 128-bit SIMD vector variant	Armv8.6
1	00	1	1	1	1	VSUDOT (by element) - 128-bit SIMD vector variant	Armv8.6
1	-	0	1	-	-	Unallocated.	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields						Instruction page	Architecture version
op1	op2	op3	op4	Q	U		
1	01	1	1	-	-	Unallocated.	-
1	1x	1	1	-	-	Unallocated.	-
1	-	1	0	-	-	Unallocated.	-

F3.1.8 System register load/store and 64-bit move

This section describes the encoding of the System register load/store and 64-bit move group. The encodings in this section are decoded from *System register access, Advanced SIMD, and floating-point* on page F3-4124.

This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see *About the T32 Advanced SIMD and floating-point instructions and their encoding* on page F3-4178

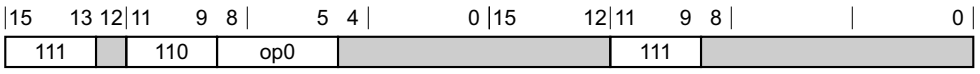
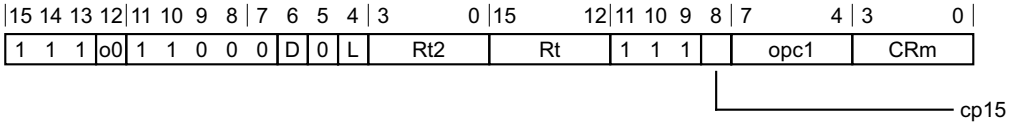


Table F3-9 Encoding table for the System register load/store and 64-bit move group

Decode fields	Decode group or instruction page
op0	
00x0	*System register 64-bit move*
!= 00x0	*System register Load/Store* on page F3-4131

System register 64-bit move

This section describes the encoding of the System register 64-bit move instruction class. The encodings in this section are decoded from *System register load/store and 64-bit move*.

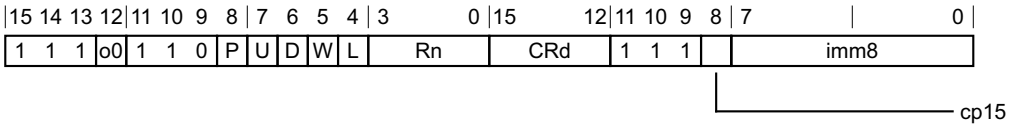


Decode fields			Instruction page
o0	D	L	
0	0	-	Unallocated.
0	1	0	MCRR
0	1	1	MRRC
1	0	-	Unallocated.
1	1	-	Unallocated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

System register Load/Store

This section describes the encoding of the System register Load/Store instruction class. The encodings in this section are decoded from *System register load/store and 64-bit move* on page F3-4130.



Decode fields							Instruction page
o0	**P:U:W**	**D**	**L**	**Rn**	**CRd**	**cp15**	
-	!= 000	-	-	-	!= 0101	0	Unallocated.
-	!= 000	-	-	-	-	1	Unallocated.
-	!= 000	1	-	-	0101	0	Unallocated.
0	!= 000	0	1	1111	0101	0	LDC (literal)
0	0x1	0	0	-	0101	0	STC - Post-indexed variant
0	0x1	0	1	!= 1111	0101	0	LDC (immediate) - Post-indexed variant
0	010	0	0	-	0101	0	STC - Unindexed variant
0	010	0	1	!= 1111	0101	0	LDC (immediate) - Unindexed variant
0	1x0	0	0	-	0101	0	STC - Offset variant
0	1x0	0	1	!= 1111	0101	0	LDC (immediate) - Offset variant
0	1x1	0	0	-	0101	0	STC - Pre-indexed variant
0	1x1	0	1	!= 1111	0101	0	LDC (immediate) - Pre-indexed variant
1	!= 000	0	-	-	0101	0	Unallocated.

F3.1.9 Floating-point data-processing

This section describes the encoding of the Floating-point data-processing group. The encodings in this section are decoded from *System register access, Advanced SIMD, and floating-point* on page F3-4124.

This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see *About the T32 Advanced SIMD and floating-point instructions and their encoding* on page F3-4178

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

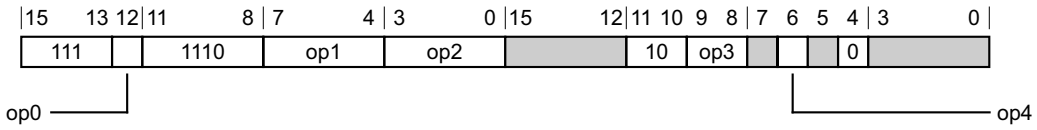
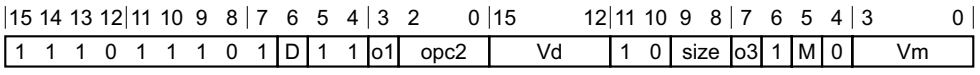


Table F3-10 Encoding table for the Floating-point data-processing group

Decode fields					Decode group or instruction page
op0	op1	op2	op3	op4	
0	1x11	-	-	1	*Floating-point data-processing (two registers)*
0	1x11	-	-	0	*Floating-point move immediate* on page F3-4134
0	!= 1x11	-	-	-	*Floating-point data-processing (three registers)* on page F3-4135
1	0xxx	-	!= 00	0	*Floating-point conditional select* on page F3-4136
1	1x00	-	!= 00	-	*Floating-point minNum/maxNum* on page F3-4136
1	1x11	0000	!= 00	1	*Floating-point extraction and insertion* on page F3-4136
1	1x11	1xxx	!= 00	1	*Floating-point directed convert to integer* on page F3-4137

Floating-point data-processing (two registers)

This section describes the encoding of the Floating-point data-processing (two registers) instruction class. The encodings in this section are decoded from *Floating-point data-processing* on page F3-4131.



Decode fields				Instruction page	Architecture version
o1	opc2	size	o3		
-	-	00	-	Unallocated.	-
0	000	01	0	Unallocated.	-
0	000	-	1	VABS	-
0	000	10	0	VMOV (register) - Single-precision scalar variant	-
0	000	11	0	VMOV (register) - Double-precision scalar variant	-
0	001	-	0	VNEG	-
0	001	-	1	VSQRT	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields				Instruction page	Architecture version
o1	opc2	size	o3		
0	010	-	0	VCVTB - Half-precision to double-precision variant	-
0	010	01	-	Unallocated.	-
0	010	-	1	VCVTT - Half-precision to double-precision variant	-
0	011	01	0	VCVTB (BFloat16)	Armv8.6
0	011	01	1	VCVTT (BFloat16)	Armv8.6
0	011	10	0	VCVTB - Single-precision to half-precision variant	-
0	011	10	1	VCVTT - Single-precision to half-precision variant	-
0	011	11	0	VCVTB - Double-precision to half-precision variant	-
0	011	11	1	VCVTT - Double-precision to half-precision variant	-
0	100	-	0	VCMP - *T1* on page F6-5087	-
0	100	-	1	VCMPE - *T1* on page F6-5091	-
0	101	-	0	VCMP - *T2* on page F6-5088	-
0	101	-	1	VCMPE - *T2* on page F6-5092	-
0	110	-	0	VRINTR	-
0	110	-	1	VRINTZ (floating-point)	-
0	111	-	0	VRINTX (floating-point)	-
0	111	01	1	Unallocated.	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields				Instruction page	Architecture version
o1	opc2	size	o3		
0	111	10	1	VCVT (between double-precision and single-precision) - Single-precision to double-precision variant	-
0	111	11	1	VCVT (between double-precision and single-precision) - Double-precision to single-precision variant	-
1	000	-	-	VCVT (integer to floating-point, floating-point)	-
1	001	01	-	Unallocated.	-
1	001	10	-	Unallocated.	-
1	001	11	0	Unallocated.	-
1	001	11	1	VJCVT	Armv8.3
1	01x	-	-	VCVT (between floating-point and fixed-point, floating-point)	-
1	100	-	0	VCVTR	-
1	100	-	1	VCVT (floating-point to integer, floating-point)	-
1	101	-	0	VCVTR	-
1	101	-	1	VCVT (floating-point to integer, floating-point)	-
1	11x	-	-	VCVT (between floating-point and fixed-point, floating-point)	-

Floating-point move immediate

This section describes the encoding of the Floating-point move immediate instruction class. The encodings in this section are decoded from *Floating-point data-processing* on page F3-4131.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	0	1	1	1	0	1	D	1	1	imm4H	Vd	1	0	size	(0)	0	(0)	0	imm4L

Decode fields size	Instruction page	Architecture version
00	Unallocated.	-
01	VMOV (immediate) - Half-precision scalar variant	Armv8.2
10	VMOV (immediate) - Single-precision scalar variant	-
11	VMOV (immediate) - Double-precision scalar variant	-

Floating-point data-processing (three registers)

This section describes the encoding of the Floating-point data-processing (three registers) instruction class. The encodings in this section are decoded from *Floating-point data-processing* on page F3-4131.

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	0	1	1	1	0	o0	D	o1		Vn	Vd	1	0	size		N	o2	M	0	Vm

Decode fields o0:o1	size	o2	Instruction page
!= 111	00	-	Unallocated.
000	-	0	VMLA (floating-point)
000	-	1	VMLS (floating-point)
001	-	0	VNMLS
001	-	1	VNMLA
010	-	0	VMUL (floating-point)
010	-	1	VNMUL
011	-	0	VADD (floating-point)
011	-	1	VSUB (floating-point)
100	-	0	VDIV
101	-	0	VFNMS
101	-	1	VFNMA
110	-	0	VFMA
110	-	1	VFMS

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Floating-point conditional select

This section describes the encoding of the Floating-point conditional select instruction class. The encodings in this section are decoded from *Floating-point data-processing* on page F3-4131.

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	1	1	1	1	0	0	D	cc		Vn	Vd	1	0	!=00		N	0	M	0	Vm
																size						

Decode fields		Instruction page
cc	size	
00	-	VSELEQ, VSELGE, VSELGT, VSELVS - VSELEQ,doubleprec variant
01	-	VSELEQ, VSELGE, VSELGT, VSELVS - VSELVS,doubleprec variant
-	01	Unallocated.
10	-	VSELEQ, VSELGE, VSELGT, VSELVS - VSELGE,doubleprec variant
11	-	VSELEQ, VSELGE, VSELGT, VSELVS - VSELGT,doubleprec variant

Floating-point minNum/maxNum

This section describes the encoding of the Floating-point minNum/maxNum instruction class. The encodings in this section are decoded from *Floating-point data-processing* on page F3-4131.

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	1	1	1	1	0	1	D	0	0	Vn	Vd	1	0	!=00		N	op	M	0	Vm
																size						

Decode fields		Instruction page
size	op	
-	0	VMAXNM
01	-	Unallocated.
-	1	VMINNM

Floating-point extraction and insertion

This section describes the encoding of the Floating-point extraction and insertion instruction class. The encodings in this section are decoded from *Floating-point data-processing* on page F3-4131.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	1	0	1	D	1	1	0	0	0	0	Vd	1	0	!=00	op	1	M	0	Vm	
																			size						

Decode fields		Instruction page	Architecture version
size	op		
01	-	Unallocated.	-
10	0	VMOVX	Armv8.2
10	1	VINS	Armv8.2
11	-	Unallocated.	-

Floating-point directed convert to integer

This section describes the encoding of the Floating-point directed convert to integer instruction class. The encodings in this section are decoded from *Floating-point data-processing* on page F3-4131.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	1	0	1	D	1	1	1	o1	RM		Vd	1	0	!=00	op	1	M	0	Vm	
																			size						

Decode fields			Instruction page
o1	RM	size	
0	00	-	VRINTA (floating-point)
0	01	-	VRINTN (floating-point)
-	-	01	Unallocated.
0	10	-	VRINTP (floating-point)
0	11	-	VRINTM (floating-point)
1	00	-	VCVTA (floating-point)
1	01	-	VCVTN (floating-point)
1	10	-	VCVTP (floating-point)
1	11	-	VCVTM (floating-point)

F3.1.10 Advanced SIMD data-processing

This section describes the encoding of the Advanced SIMD data-processing group. The encodings in this section are decoded from *System register access, Advanced SIMD, and floating-point* on page F3-4124.

This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see *About the T32 Advanced SIMD and floating-point instructions and their encoding* on page F3-4178

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

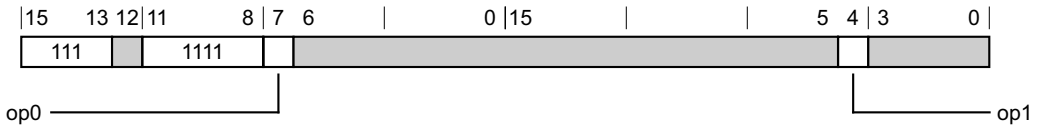


Table F3-11 Encoding table for the Advanced SIMD data-processing group

Decode fields		Decode group or instruction page
op0	**op1**	
0	-	*Advanced SIMD three registers of the same length*
1	0	*Advanced SIMD two registers, or three registers of different lengths* on page F3-4140
1	1	*Advanced SIMD shifts and immediate generation* on page F3-4145

Advanced SIMD three registers of the same length

This section describes the encoding of the Advanced SIMD three registers of the same length instruction class. The encodings in this section are decoded from *Advanced SIMD data-processing* on page F3-4137.

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11 8	7	6	5	4	3 0
1	1	1	U	1	1	1	1	0	D	size		Vn	Vd	opc	N	Q	M	o1	Vm

Decode fields					Instruction page	Architecture version
U	**size**	**opc**	**Q**	**o1**		
0	0x	1100	-	1	VFMA	-
0	0x	1101	-	0	VADD (floating-point)	-
0	0x	1101	-	1	VMLA (floating-point)	-
0	0x	1110	-	0	VCEQ (register) - *T2* on page F6-5048	-
0	0x	1111	-	0	VMAX (floating-point)	-
0	0x	1111	-	1	VRECPS	-
-	-	0000	-	0	VHADD	-
0	00	0001	-	1	VAND (register)	-
-	-	0000	-	1	VQADD	-
-	-	0001	-	0	VRHADD	-
0	00	1100	-	0	SHA1C	-
-	-	0010	-	0	VHSUB	-
0	01	0001	-	1	VBIC (register)	-
-	-	0010	-	1	VQSUB	-
-	-	0011	-	0	VCGT (register) - *T1* on page F6-5061	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields					Instruction page	Architecture version
U	size	opc	Q	o1		
-	-	0011	-	1	VCGE (register) - *T1* on page F6-5054	-
0	01	1100	-	0	SHA1P	-
0	1x	1100	-	1	VFMS	-
0	1x	1101	-	0	VSUB (floating-point)	-
0	1x	1101	-	1	VMLS (floating-point)	-
0	1x	1110	-	0	Unallocated.	-
0	1x	1111	-	0	VMIN (floating-point)	-
0	1x	1111	-	1	VRSQRTS	-
-	-	0100	-	0	VSHL (register)	-
0	-	1000	-	0	VADD (integer)	-
0	10	0001	-	1	VORR (register)	-
0	-	1000	-	1	VTST	-
-	-	0100	-	1	VQSHL (register)	-
0	-	1001	-	0	VMLA (integer)	-
-	-	0101	-	0	VRSHL	-
-	-	0101	-	1	VQRSHL	-
0	-	1011	-	0	VQDMULH	-
0	10	1100	-	0	SHA1M	-
0	-	1011	-	1	VPADD (integer)	-
-	-	0110	-	0	VMAX (integer)	-
0	11	0001	-	1	VORN (register)	-
-	-	0110	-	1	VMIN (integer)	-
-	-	0111	-	0	VABD (integer)	-
-	-	0111	-	1	VABA	-
0	11	1100	-	0	SHA1SU0	-
1	0x	1101	-	0	VPADD (floating-point)	-
1	0x	1101	-	1	VMUL (floating-point)	-
1	0x	1110	-	0	VCGE (register) - *T2* on page F6-5054	-
1	0x	1110	-	1	VACGE	-
1	0x	1111	0	0	VPMAX (floating-point)	-
1	0x	1111	-	1	VMAXNM	-
1	00	0001	-	1	VEOR	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields					Instruction page	Architecture version
U	size	opc	Q	o1		
-	-	1001	-	1	VMUL (integer and polynomial)	-
1	00	1100	-	0	SHA256H	-
-	-	1010	0	0	VPMAX (integer)	-
1	01	0001	-	1	VBSL	-
-	-	1010	0	1	VPMIN (integer)	-
-	-	1010	1	-	Unallocated.	-
1	01	1100	-	0	SHA256H2	-
1	1x	1101	-	0	VABD (floating-point)	-
1	1x	1110	-	0	VCGT (register) - *T2* on page F6-5061	-
1	1x	1110	-	1	VACGT	-
1	1x	1111	0	0	VPMIN (floating-point)	-
1	1x	1111	-	1	VMINNM	-
1	-	1000	-	0	VSUB (integer)	-
1	10	0001	-	1	VBIT	-
1	-	1000	-	1	VCEQ (register) - *T1* on page F6-5048	-
1	-	1001	-	0	VMLS (integer)	-
1	-	1011	-	0	VQRDMULH	-
1	10	1100	-	0	SHA256SU1	-
1	-	1011	-	1	VQRDMLAH	Armv8.1
1	11	0001	-	1	VBIF	-
1	-	1100	-	1	VQRDMLSH	Armv8.1
1	-	1111	1	0	Unallocated.	-

F3.1.11 Advanced SIMD two registers, or three registers of different lengths

This section describes the encoding of the Advanced SIMD two registers, or three registers of different lengths group. The encodings in this section are decoded from *Advanced SIMD data-processing* on page F3-4137.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

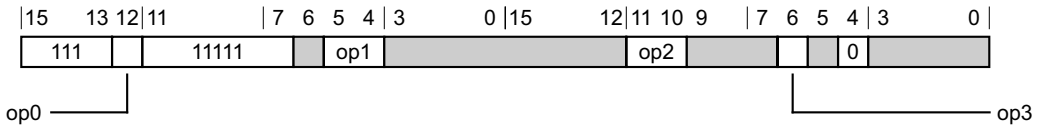
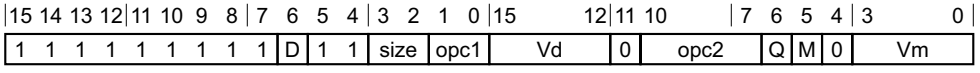


Table F3-12 Encoding table for the Advanced SIMD two registers, or three registers of different lengths group

Decode fields				Decode group or instruction page
op0	op1	op2	op3	
0	11	-	-	VEXT (byte elements)
1	11	0x	-	*Advanced SIMD two registers misc*
1	11	10	-	VTBL, VTBX
1	11	11	-	*Advanced SIMD duplicate (scalar)* on page F3-4143
-	!= 11	-	0	*Advanced SIMD three registers of different lengths* on page F3-4143
-	!= 11	-	1	*Advanced SIMD two registers and a scalar* on page F3-4144

Advanced SIMD two registers misc

This section describes the encoding of the Advanced SIMD two registers misc instruction class. The encodings in this section are decoded from *Advanced SIMD two registers, or three registers of different lengths* on page F3-4140.



Decode fields				Instruction page	Architecture version
size	opc1	opc2	Q		
-	00	0000	-	VREV64	-
-	00	0001	-	VREV32	-
-	00	0010	-	VREV16	-
-	00	0011	-	Unallocated.	-
-	00	010x	-	VPADDL	-
-	00	0110	0	AESE	-
-	00	0110	1	AESD	-
-	00	0111	0	AESMC	-
-	00	0111	1	AESIMC	-
-	00	1000	-	VCLS	-
00	10	0000	-	VSWP	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields				Instruction page	Architecture version
size	opc1	opc2	Q		
-	00	1001	-	VCLZ	-
-	00	1010	-	VCNT	-
-	00	1011	-	VMVN (register)	-
00	10	1100	1	Unallocated.	-
-	00	110x	-	VPADAL	-
-	00	1110	-	VQABS	-
-	00	1111	-	VQNEG	-
-	01	x000	-	VCGT (immediate #0)	-
-	01	x001	-	VCGE (immediate #0)	-
-	01	x010	-	VCEQ (immediate #0)	-
-	01	x011	-	VCLE (immediate #0)	-
-	01	x100	-	VCLT (immediate #0)	-
-	01	x110	-	VABS	-
-	01	x111	-	VNEG	-
-	01	0101	1	SHA1H	-
01	10	1100	1	VCVT (from single-precision to BFloat16, Advanced SIMD)	Armv8.6
-	10	0001	-	VTRN	-
-	10	0010	-	VUZP	-
-	10	0011	-	VZIP	-
-	10	0100	0	VMOVN	-
-	10	0100	1	VQMOVN, VQMOVUN - Unsigned result variant	-
-	10	0101	-	VQMOVN, VQMOVUN - Signed result variant	-
-	10	0110	0	VSHLL	-
-	10	0111	0	SHA1SU1	-
-	10	0111	1	SHA256SU0	-
-	10	1000	-	VRINTN (Advanced SIMD)	-
-	10	1001	-	VRINTX (Advanced SIMD)	-
-	10	1010	-	VRINTA (Advanced SIMD)	-
-	10	1011	-	VRINTZ (Advanced SIMD)	-
10	10	1100	1	Unallocated.	-
-	10	1100	0	VCVT (between half-precision and single-precision, Advanced SIMD) - Single-precision to half-precision variant	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields				Instruction page	Architecture version
size	opc1	opc2	Q		
-	10	1101	-	VRINTM (Advanced SIMD)	-
-	10	1110	0	VCVT (between half-precision and single-precision, Advanced SIMD) - Half-precision to single-precision variant	-
-	10	1110	1	Unallocated.	-
-	10	1111	-	VRINTP (Advanced SIMD)	-
-	11	000x	-	VCVTA (Advanced SIMD)	-
-	11	001x	-	VCVTN (Advanced SIMD)	-
-	11	010x	-	VCVTP (Advanced SIMD)	-
-	11	011x	-	VCVTM (Advanced SIMD)	-
-	11	10x0	-	VRECPE	-
-	11	10x1	-	VRSQRTE	-
11	10	1100	1	Unallocated.	-
-	11	11xx	-	VCVT (between floating-point and integer, Advanced SIMD)	-

Advanced SIMD duplicate (scalar)

This section describes the encoding of the Advanced SIMD duplicate (scalar) instruction class. The encodings in this section are decoded from *Advanced SIMD two registers, or three registers of different lengths* on page F3-4140.

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10	9 7	6	5	4	3 0
1 1 1 1	1 1 1 1	1	D	1	1	imm4	Vd	1 1	opc	Q	M	0	Vm

Decode fields	Instruction page
opc	
000	VDUP (scalar)
001	Unallocated.
01x	Unallocated.
1xx	Unallocated.

Advanced SIMD three registers of different lengths

This section describes the encoding of the Advanced SIMD three registers of different lengths instruction class. The encodings in this section are decoded from *Advanced SIMD two registers, or three registers of different lengths* on page F3-4140.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

15 14 13	12	11 10 9 8	7	6	5 4	3 0	15 12	11 8	7	6	5	4	3 0
1 1 1	U	1 1 1 1	1	D	!=11	Vn	Vd	opc	N	0	M	0	Vm
					size								

Decode fields		Instruction page
U	opc	
-	0000	VADDL
-	0001	VADDW
-	0010	VSUBL
0	0100	VADDHN
-	0011	VSUBW
0	0110	VSUBHN
0	1001	VQDMLAL
-	0101	VABAL
0	1011	VQDMLSL
0	1101	VQDMULL
-	0111	VABDL (integer)
-	1000	VMLAL (integer)
-	1010	VMLSL (integer)
1	0100	VRADDHN
1	0110	VRSUBHN
-	11x0	VMULL (integer and polynomial)
1	1001	Unallocated.
1	1011	Unallocated.
1	1101	Unallocated.
-	1111	Unallocated.

Advanced SIMD two registers and a scalar

This section describes the encoding of the Advanced SIMD two registers and a scalar instruction class. The encodings in this section are decoded from *Advanced SIMD two registers, or three registers of different lengths* on page F3-4140.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

15 14 13	12	11 10 9 8	7	6	5 4	3 0	15 12	11 8	7	6	5	4	3 0
1 1 1	Q	1 1 1 1	1	D	!=11 size	Vn	Vd	opc	N	1	M	0	Vm

Decode fields Q	opc	Instruction page	Architecture version
-	000x	VMLA (by scalar)	-
0	0011	VQDMLAL	-
-	0010	VMLAL (by scalar)	-
0	0111	VQDMLSL	-
-	010x	VMLS (by scalar)	-
0	1011	VQDMULL	-
-	0110	VMLSL (by scalar)	-
-	100x	VMUL (by scalar)	-
1	0011	Unallocated.	-
-	1010	VMULL (by scalar)	-
1	0111	Unallocated.	-
-	1100	VQDMULH	-
-	1101	VQRDMULH	-
1	1011	Unallocated.	-
-	1110	VQRDMLAH	Armv8.1
-	1111	VQRDMLSH	Armv8.1

F3.1.12 Advanced SIMD shifts and immediate generation

This section describes the encoding of the Advanced SIMD shifts and immediate generation group. The encodings in this section are decoded from *Advanced SIMD data-processing* on page F3-4137.

15 13	12	11 7	6	5 0	15 7	6	5	4	3 0
111		11111		op0				1	

Table F3-13 Encoding table for the Advanced SIMD shifts and immediate generation group

Decode fields op0	Decode group or instruction page
000xxxxxxxxxxx0	*Advanced SIMD one register and modified immediate* on page F3-4146
!= 000xxxxxxxxxxx0	*Advanced SIMD two registers and shift amount* on page F3-4146

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Advanced SIMD one register and modified immediate

This section describes the encoding of the Advanced SIMD one register and modified immediate instruction class. The encodings in this section are decoded from *Advanced SIMD shifts and immediate generation* on page F3-4145.

15 14 13	12	11 10 9 8	7	6	5 4	3 2 0	15 12	11 8	7	6	5	4	3 0
1 1 1	i	1 1 1 1	1	D	0 0	0 imm3	Vd	cmode	0	Q	op	1	imm4

Decode fields		Instruction page
cmode	op	
0xx0	0	VMOV (immediate) - *T1* on page F6-5322
0xx0	1	VMVN (immediate) - *T1* on page F6-5367
0xx1	0	VORR (immediate) - *T1* on page F6-5391
0xx1	1	VBIC (immediate) - *T1* on page F6-5033
10x0	0	VMOV (immediate) - *T3* on page F6-5323
10x0	1	VMVN (immediate) - *T2* on page F6-5367
10x1	0	VORR (immediate) - *T2* on page F6-5391
10x1	1	VBIC (immediate) - *T2* on page F6-5033
11xx	0	VMOV (immediate) - *T4* on page F6-5324
110x	1	VMVN (immediate) - *T3* on page F6-5368
1110	1	VMOV (immediate) - *T5* on page F6-5324
1111	1	Unallocated.

Advanced SIMD two registers and shift amount

This section describes the encoding of the Advanced SIMD two registers and shift amount instruction class. The encodings in this section are decoded from *Advanced SIMD shifts and immediate generation* on page F3-4145.

15 14 13	12	11 10 9 8	7	6	5	3 2 0	15 12	11 8	7	6	5	4	3 0
1 1 1	U	1 1 1 1	1	D	imm3H	imm3L	Vd	opc	L	Q	M	1	Vm

Decode fields					Instruction page
U	imm3H:L	imm3L	opc	Q	
-	!= 0000	-	0000	-	VSHR
-	!= 0000	-	0001	-	VSRA
-	!= 0000	000	1010	0	VMOVL
-	!= 0000	-	0010	-	VRSHR
-	!= 0000	-	0011	-	VRSRA

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields					Instruction page
U	imm3H:L	imm3L	opc	Q	
-	!= 0000	-	0111	-	VQSHL, VQSHLU (immediate) - VQSHL,quad,signed-result variant
-	!= 0000	-	1001	0	VQSHRN, VQSHRUN - Signed result variant
-	!= 0000	-	1001	1	VQRSHRN, VQRSHRUN - Signed result variant
-	!= 0000	-	1010	0	VSHLL
-	!= 0000	-	11xx	-	VCVT (between floating-point and fixed-point, Advanced SIMD)
0	!= 0000	-	0101	-	VSHL (immediate)
0	!= 0000	-	1000	0	VSHRN
0	!= 0000	-	1000	1	VRSHRN
1	!= 0000	-	0100	-	VSRI
1	!= 0000	-	0101	-	VSLI
1	!= 0000	-	0110	-	VQSHL, VQSHLU (immediate) - VQSHLU,quad,unsigned-result variant
1	!= 0000	-	1000	0	VQSHRN, VQSHRUN - Unsigned result variant
1	!= 0000	-	1000	1	VQRSHRN, VQRSHRUN - Unsigned result variant

F3.1.13 Advanced SIMD load/store and 64-bit move

This section describes the encoding of the Advanced SIMD load/store and 64-bit move group. The encodings in this section are decoded from *System register access, Advanced SIMD, and floating-point* on page F3-4124.

This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see *About the T32 Advanced SIMD and floating-point instructions and their encoding* on page F3-4178

Table F3-14 Encoding table for the Advanced SIMD load/store and 64-bit move group

Decode fields	Decode group or instruction page
op0	
00x0	*Advanced SIMD and floating-point 64-bit move*
!= 00x0	*Advanced SIMD and floating-point load/store* on page F3-4148

Advanced SIMD and floating-point 64-bit move

This section describes the encoding of the Advanced SIMD and floating-point 64-bit move instruction class. The encodings in this section are decoded from *Advanced SIMD load/store and 64-bit move*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 10 9 8	7 6 5 4	3 0
1 1 1 0	1 1 0 0	0 D 0 op	Rt2	Rt	1 0 size	opc2 M o3	Vm

Decode fields					Instruction page
D	op	size	opc2	o3	
0	-	-	-	-	Unallocated.
1	-	-	-	0	Unallocated.
1	-	0x	00	1	Unallocated.
1	-	-	01	-	Unallocated.
1	0	10	00	1	VMOV (between two general-purpose registers and two single-precision registers) - From general-purpose registers variant
1	0	11	00	1	VMOV (between two general-purpose registers and a doubleword floating-point register) - From general-purpose registers variant
1	-	-	1x	-	Unallocated.
1	1	10	00	1	VMOV (between two general-purpose registers and two single-precision registers) - To general-purpose registers variant
1	1	11	00	1	VMOV (between two general-purpose registers and a doubleword floating-point register) - To general-purpose registers variant

Advanced SIMD and floating-point load/store

This section describes the encoding of the Advanced SIMD and floating-point load/store instruction class. The encodings in this section are decoded from *Advanced SIMD load/store and 64-bit move* on page F3-4147.

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 10 9 8	7 0
1 1 1 0	1 1 0 P	U D W L	Rn	Vd	1 0 size	imm8

Decode fields							Instruction page
P	U	W	L	Rn	size	imm8	
0	0	1	-	-	-	-	Unallocated.
0	1	-	-	-	0x	-	Unallocated.
0	1	-	0	-	10	-	VSTM, VSTMDB, VSTMIA - *Increment After variant* on page F6-5619
0	1	-	0	-	11	xxxxxxx0	VSTM, VSTMDB, VSTMIA - *Increment After variant* on page F6-5619
0	1	-	0	-	11	xxxxxxx1	FSTMDBX, FSTMIAX - *Increment After variant* on page F6-4969
0	1	-	1	-	10	-	VLDM, VLDMDB, VLDMIA - *Increment After variant* on page F6-5261
0	1	-	1	-	11	xxxxxxx0	VLDM, VLDMDB, VLDMIA - *Increment After variant* on page F6-5261
0	1	-	1	-	11	xxxxxxx1	FLDM*X (FLDMDBX, FLDMIAX) - *Increment After variant* on page F6-4966

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields							Instruction page
P	U	W	L	Rn	size	imm8	
1	-	0	0	-	-	-	VSTR
1	-	0	-	-	00	-	Unallocated.
1	-	0	1	!= 1111	-	-	VLDR (immediate)
1	0	1	-	-	0x	-	Unallocated.
1	0	1	0	-	10	-	VSTM, VSTMDB, VSTMIA - *Decrement Before variant* on page F6-5619
1	0	1	0	-	11	xxxxxxx0	VSTM, VSTMDB, VSTMIA - *Decrement Before variant* on page F6-5618
1	0	1	0	-	11	xxxxxxx1	FSTMDBX, FSTMIAX - *Decrement Before variant* on page F6-4969
1	0	1	1	-	10	-	VLDM, VLDMDB, VLDMIA - *Decrement Before variant* on page F6-5261
1	0	1	1	-	11	xxxxxxx0	VLDM, VLDMDB, VLDMIA - *Decrement Before variant* on page F6-5260
1	0	1	1	-	11	xxxxxxx1	FLDM*X (FLDMDBX, FLDMIAX) - *Decrement Before variant* on page F6-4966
1	-	0	1	1111	-	-	VLDR (literal)
1	1	1	-	-	-	-	Unallocated.

F3.1.14 Advanced SIMD and floating-point 32-bit move

This section describes the encoding of the Advanced SIMD and floating-point 32-bit move group. The encodings in this section are decoded from *System register access, Advanced SIMD, and floating-point* on page F3-4124.

This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see *About the T32 Advanced SIMD and floating-point instructions and their encoding* on page F3-4178

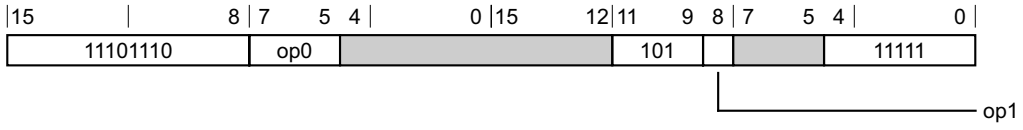


Table F3-15 Encoding table for the Advanced SIMD and floating-point 32-bit move group

Decode fields		Decode group or instruction page
op0	op1	
000	0	VMOV (between general-purpose register and single-precision)
111	0	*Floating-point move special register*
-	1	*Advanced SIMD 8/16/32-bit element move/duplicate* on page F3-4150

Floating-point move special register

This section describes the encoding of the Floating-point move special register instruction class. The encodings in this section are decoded from *Advanced SIMD and floating-point 32-bit move*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	1	1	L	reg	Rt	1	0	1	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)

Decode fields	Instruction page
L	
0	VMSR
1	VMRS

Advanced SIMD 8/16/32-bit element move/duplicate

This section describes the encoding of the Advanced SIMD 8/16/32-bit element move/duplicate instruction class. The encodings in this section are decoded from *Advanced SIMD and floating-point 32-bit move* on page F3-4149.

15	14	13	12	11	10	9	8	7 … 5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	opc1	L	Vn	Rt	1	0	1	1	N	opc2	1	(0)	(0)	(0)	(0)

Decode fields			Instruction page
opc1	L	opc2	
0xx	0	-	VMOV (general-purpose register to scalar)
-	1	-	VMOV (scalar to general-purpose register)
1xx	0	0x	VDUP (general-purpose register)
1xx	0	1x	Unallocated.

F3.1.15 Load/store dual, load/store exclusive, load-acquire/store-release, and table branch

This section describes the encoding of the Load/store dual, load/store exclusive, load-acquire/store-release, and table branch group. The encodings in this section are decoded from *32-bit* on page F3-4119.

15 … 9	8 … 5	4	3 … 0	15 … 8	7 … 5	4 … 0
1110100	op0		op2		op3	

op1

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This decode also imposes the constraint:

- `op0<1> == 1`.

Table F3-16 Encoding table for the Load/store dual, load/store exclusive, load-acquire/store-release, and table branch group

Decode fields				Decode group or instruction page
op0	op1	op2	op3	
0010	-	-	-	*Load/store exclusive*
0110	0	-	000	Unallocated.
0110	1	-	000	TBB, TBH
0110	-	-	01x	*Load/store exclusive byte/half/dual*
0110	-	-	1xx	*Load-acquire / Store-release* on page F3-4152
0x11	-	!= 1111	-	*Load/store dual (immediate, post-indexed)* on page F3-4153
1x10	-	!= 1111	-	*Load/store dual (immediate)* on page F3-4153
1x11	-	!= 1111	-	*Load/store dual (immediate, pre-indexed)* on page F3-4153
!= 0xx0	-	1111	-	LDRD (literal)

Load/store exclusive

This section describes the encoding of the Load/store exclusive instruction class. The encodings in this section are decoded from *Load/store dual, load/store exclusive, load-acquire/store-release, and table branch* on page F3-4150.

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 8	7 0
1 1 1 0	1 0 0 0	0 1 0 L	Rn	Rt	Rd	imm8

Decode fields	Instruction page
L	
0	STREX
1	LDREX

Load/store exclusive byte/half/dual

This section describes the encoding of the Load/store exclusive byte/half/dual instruction class. The encodings in this section are decoded from *Load/store dual, load/store exclusive, load-acquire/store-release, and table branch* on page F3-4150.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 8	7 6	5 4	3 0
1 1 1 0	1 0 0 0	1 1 0 L	Rn	Rt	Rt2	0 1	sz	Rd

Decode fields		Instruction page
L	sz	
0	00	STREXB
0	01	STREXH
0	10	Unallocated.
0	11	STREXD
1	00	LDREXB
1	01	LDREXH
1	10	Unallocated.
1	11	LDREXD

Load-acquire / Store-release

This section describes the encoding of the Load-acquire / Store-release instruction class. The encodings in this section are decoded from *Load/store dual, load/store exclusive, load-acquire/store-release, and table branch* on page F3-4150.

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 8	7	6	5 4	3 0
1 1 1 0	1 0 0 0	1 1 0 L	Rn	Rt	Rt2	1	op	sz	Rd

Decode fields			Instruction page
L	op	sz	
0	0	00	STLB
0	0	01	STLH
0	0	10	STL
0	0	11	Unallocated.
0	1	00	STLEXB
0	1	01	STLEXH
0	1	10	STLEX
0	1	11	STLEXD
1	0	00	LDAB
1	0	01	LDAH
1	0	10	LDA

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields			Instruction page
L	op	sz	
1	0	11	Unallocated.
1	1	00	LDAEXB
1	1	01	LDAEXH
1	1	10	LDAEX
1	1	11	LDAEXD

Load/store dual (immediate, post-indexed)

This section describes the encoding of the Load/store dual (immediate, post-indexed) instruction class. The encodings in this section are decoded from *Load/store dual, load/store exclusive, load-acquire/store-release, and table branch* on page F3-4150.

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11 … 8	7 … 0
1	1	1	0	1	0	0	0	U	1	1	L	!=1111 (Rn)	Rt	Rt2	imm8

Decode fields	Instruction page
L	
0	STRD (immediate)
1	LDRD (immediate)

Load/store dual (immediate)

This section describes the encoding of the Load/store dual (immediate) instruction class. The encodings in this section are decoded from *Load/store dual, load/store exclusive, load-acquire/store-release, and table branch* on page F3-4150.

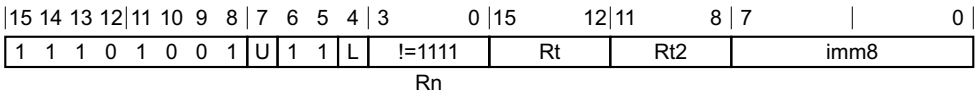
15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11 … 8	7 … 0
1	1	1	0	1	0	0	1	U	1	0	L	!=1111 (Rn)	Rt	Rt2	imm8

Decode fields	Instruction page
L	
0	STRD (immediate)
1	LDRD (immediate)

Load/store dual (immediate, pre-indexed)

This section describes the encoding of the Load/store dual (immediate, pre-indexed) instruction class. The encodings in this section are decoded from *Load/store dual, load/store exclusive, load-acquire/store-release, and table branch* on page F3-4150.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta



Decode fields	Instruction page
L	
0	STRD (immediate)
1	LDRD (immediate)

F3.1.16 Branches and miscellaneous control

This section describes the encoding of the Branches and miscellaneous control group. The encodings in this section are decoded from *32-bit* on page F3-4119.

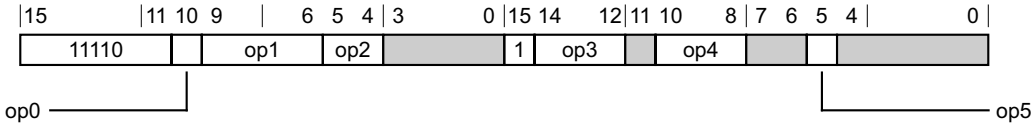


Table F3-17 Encoding table for the Branches and miscellaneous control group

Decode fields						Decode group or instruction page
op0	op1	op2	op3	op4	op5	
0	1110	0x	0x0	-	0	MSR (register)
0	1110	0x	0x0	-	1	MSR (Banked register)
0	1110	10	0x0	000	-	*Hints* on page F3-4155
0	1110	10	0x0	!= 000	-	*Change processor state* on page F3-4156
0	1110	11	0x0	-	-	*Miscellaneous system* on page F3-4156
0	1111	00	0x0	-	-	BXJ
0	1111	01	0x0	-	-	*Exception return* on page F3-4157
0	1111	1x	0x0	-	0	MRS
0	1111	1x	0x0	-	1	MRS (Banked register)
1	1110	00	000	-	-	*DCPS* on page F3-4157
1	1110	00	010	-	-	Unallocated.
1	1110	01	0x0	-	-	Unallocated.
1	1110	1x	0x0	-	-	Unallocated.
1	1111	0x	0x0	-	-	Unallocated.
1	1111	1x	0x0	-	-	*Exception generation* on page F3-4157
-	!= 111x	-	0x0	-	-	B - *T3 variant* on page F5-4292

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table F3-17 Encoding table for the Branches and miscellaneous control group (continued)

Decode fields						Decode group or instruction page
op0	op1	op2	op3	op4	op5	
-	-	-	0x1	-	-	B - *T4 variant* on page F5-4292
-	-	-	1x0	-	-	BL, BLX (immediate) - *T2 variant* on page F5-4310
-	-	-	1x1	-	-	BL, BLX (immediate) - *T1 variant* on page F5-4309

Hints

This section describes the encoding of the Hints instruction class. The encodings in this section are decoded from *Branches and miscellaneous control* on page F3-4154.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7 – 4	3 – 0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	hint	option

Decode fields		Instruction page	Architecture version
hint	option		
0000	0000	NOP	-
0000	0001	YIELD	-
0000	0010	WFE	-
0000	0011	WFI	-
0000	0100	SEV	-
0000	0101	SEVL	-
0000	011x	Reserved hint, behaves as NOP.	-
0000	1xxx	Reserved hint, behaves as NOP.	-
0001	0000	ESB	Armv8.2
0001	0001	Reserved hint, behaves as NOP.	-
0001	0010	TSB CSYNC	Armv8.4
0001	0011	Reserved hint, behaves as NOP.	-
0001	0100	CSDB	-
0001	0101	Reserved hint, behaves as NOP.	-
0001	011x	Reserved hint, behaves as NOP.	-
0001	1xxx	Reserved hint, behaves as NOP.	-
001x	-	Reserved hint, behaves as NOP.	-
01xx	-	Reserved hint, behaves as NOP.	-
10xx	-	Reserved hint, behaves as NOP.	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields		Instruction page	Architecture version
hint	option		
110x	-	Reserved hint, behaves as NOP.	-
1110	-	Reserved hint, behaves as NOP.	-
1111	-	DBG	-

Change processor state

This section describes the encoding of the Change processor state instruction class. The encodings in this section are decoded from *Branches and miscellaneous control* on page F3-4154.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	imod		M	A	I	F	mode	

Decode fields		Instruction page
imod	M	
00	1	CPS, CPSID, CPSIE - CPS variant
01	-	Unallocated.
10	-	CPS, CPSID, CPSIE - CPSIE variant
11	-	CPS, CPSID, CPSIE - CPSID variant

Miscellaneous system

This section describes the encoding of the Miscellaneous System instruction class. The encodings in this section are decoded from *Branches and miscellaneous control* on page F3-4154.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	4	3	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	opc		option	

Decode fields		Instruction page
opc	option	
000x	-	Unallocated.
0010	-	CLREX
0011	-	Unallocated.
0100	!= 0x00	DSB
0100	0000	SSBB
0100	0100	PSSBB
0101	-	DMB

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields		Instruction page
opc	option	
0110	-	ISB
0111	-	SB
1xxx	-	Unallocated.

Exception return

This section describes the encoding of the Exception return instruction class. The encodings in this section are decoded from *Branches and miscellaneous control* on page F3-4154.

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15	14	13	12	11	10	9	8	7 … 0
1	1	1	1	0	0	1	1	1	1	0	1	Rn	1	0	(0)	0	(1)	(1)	(1)	(1)	imm8

Decode fields		Instruction page
Rn	imm8	
-	!= 00000000	SUB, SUBS (immediate)
1110	00000000	ERET

DCPS

This section describes the encoding of the DCPS instruction class. The encodings in this section are decoded from *Branches and miscellaneous control* on page F3-4154.

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15	14	13	12	11 … 2	1	0
1	1	1	1	0	1	1	1	1	0	0	0	imm4	1	0	0	0	imm10	opt	

Decode fields			Instruction page
imm4	imm10	opt	
!= 1111	-	-	Unallocated.
1111	!= 0000000000	-	Unallocated.
1111	0000000000	00	Unallocated.
1111	0000000000	01	DCPS1
1111	0000000000	10	DCPS2
1111	0000000000	11	DCPS3

Exception generation

This section describes the encoding of the Exception generation instruction class. The encodings in this section are decoded from *Branches and miscellaneous control* on page F3-4154.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15	14	13	12	11 … 0
1	1	1	1	0	1	1	1	1	1	1	o1	imm4	1	0	o2	0	imm12

Decode fields		Instruction page
o1	o2	
0	0	HVC
0	1	Unallocated.
1	0	SMC
1	1	UDF

F3.1.17 Data-processing (plain binary immediate)

This section describes the encoding of the Data-processing (plain binary immediate) group. The encodings in this section are decoded from *32-bit* on page F3-4119.

15 … 11	10	9	8	7	6	5	4	3 … 0	15	14 … 0
11110		1			op1		0		0	

op0

Table F3-18 Encoding table for the Data-processing (plain binary immediate) group

Decode fields		Decode group or instruction page
op0	op1	
0	0x	*Data-processing (simple immediate)*
0	10	*Move Wide (16-bit immediate)* on page F3-4159
0	11	Unallocated.
1	-	*Saturate, Bitfield* on page F3-4159

Data-processing (simple immediate)

This section describes the encoding of the Data-processing (simple immediate) instruction class. The encodings in this section are decoded from *Data-processing (plain binary immediate)*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15	14 … 12	11 … 8	7 … 0
1	1	1	1	0	i	1	0	o1	0	o2	0	Rn	0	imm3	Rd	imm8

Decode fields			Instruction page
o1	o2	Rn	
0	0	!= 11x1	ADD, ADDS (immediate)
0	0	1101	ADD, ADDS (SP plus immediate)
0	0	1111	ADR - *T3* on page F5-4272
0	1	-	Unallocated.
1	0	-	Unallocated.
1	1	!= 11x1	SUB, SUBS (immediate)
1	1	1101	SUB, SUBS (SP minus immediate)
1	1	1111	ADR - *T2* on page F5-4271

Move Wide (16-bit immediate)

This section describes the encoding of the Move Wide (16-bit immediate) instruction class. The encodings in this section are decoded from *Data-processing (plain binary immediate)* on page F3-4158.

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15	14 … 12	11 … 8	7 … 0
1	1	1	1	0	i	1	0	o1	1	0	0	imm4	0	imm3	Rd	imm8

Decode fields	Instruction page
o1	
0	MOV, MOVS (immediate)
1	MOVT

Saturate, Bitfield

This section describes the encoding of the Saturate, Bitfield instruction class. The encodings in this section are decoded from *Data-processing (plain binary immediate)* on page F3-4158.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

15 14 13 12	11 10	9 8	7 5	4	3 0	15 14 12	11 8	7 6 5 4	0
1 1 1 1 0	(0)	1 1	op1	0	Rn	0 imm3	Rd	imm2 (0)	widthm1

Decode fields			Instruction page
op1	Rn	imm3:imm2	
000	-	-	SSAT - Logical shift left variant
001	-	!= 00000	SSAT - Arithmetic shift right variant
001	-	00000	SSAT16
010	-	-	SBFX
011	!= 1111	-	BFI
011	1111	-	BFC
100	-	-	USAT - Logical shift left variant
101	-	!= 00000	USAT - Arithmetic shift right variant
101	-	00000	USAT16
110	-	-	UBFX
111	-	-	Unallocated.

F3.1.18 Advanced SIMD element or structure load/store

This section describes the encoding of the Advanced SIMD element or structure load/store group. The encodings in this section are decoded from *32-bit* on page F3-4119.

This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see *About the T32 Advanced SIMD and floating-point instructions and their encoding* on page F3-4178

15 8	7	6 5	4	3 0	15 12	11 10 9	0
11111001			0			op1	

op0

Table F3-19 Encoding table for the Advanced SIMD element or structure load/store group

Decode fields		Decode group or instruction page
op0	op1	
0	-	*Advanced SIMD load/store multiple structures* on page F3-4161
1	11	*Advanced SIMD load single structure to all lanes* on page F3-4161
1	!= 11	*Advanced SIMD load/store single structure to one lane* on page F3-4162

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Advanced SIMD load/store multiple structures

This section describes the encoding of the Advanced SIMD load/store multiple structures instruction class. The encodings in this section are decoded from *Advanced SIMD element or structure load/store* on page F3-4160.

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 8	7 6 5 4	3 0
1 1 1 1	1 0 0 1 0	D L 0	Rn	Vd	itype	size	align	Rm

Decode fields		Instruction page
L	itype	
0	000x	VST4 (multiple 4-element structures)
0	0010	VST1 (multiple single elements) - *T4* on page F6-5585
0	0011	VST2 (multiple 2-element structures) - *T2* on page F6-5596
0	010x	VST3 (multiple 3-element structures)
0	0110	VST1 (multiple single elements) - *T3* on page F6-5584
0	0111	VST1 (multiple single elements) - *T1* on page F6-5583
0	100x	VST2 (multiple 2-element structures) - *T1* on page F6-5595
0	1010	VST1 (multiple single elements) - *T2* on page F6-5583
1	000x	VLD4 (multiple 4-element structures)
1	0010	VLD1 (multiple single elements) - *T4* on page F6-5218
1	0011	VLD2 (multiple 2-element structures) - *T2* on page F6-5232
1	010x	VLD3 (multiple 3-element structures)
-	1011	Unallocated.
1	0110	VLD1 (multiple single elements) - *T3* on page F6-5217
1	0111	VLD1 (multiple single elements) - *T1* on page F6-5216
-	11xx	Unallocated.
1	100x	VLD2 (multiple 2-element structures) - *T1* on page F6-5231
1	1010	VLD1 (multiple single elements) - *T2* on page F6-5216

Advanced SIMD load single structure to all lanes

This section describes the encoding of the Advanced SIMD load single structure to all lanes instruction class. The encodings in this section are decoded from *Advanced SIMD element or structure load/store* on page F3-4160.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 10 9 8	7 6 5 4	3 0
1 1 1 1	1 0 0 1 1	D L 0	Rn	Vd	1 1 N	size T a	Rm

Decode fields			Instruction page
L	N	a	
0	-	-	Unallocated.
1	00	-	VLD1 (single element to all lanes)
1	01	-	VLD2 (single 2-element structure to all lanes)
1	10	0	VLD3 (single 3-element structure to all lanes)
1	10	1	Unallocated.
1	11	-	VLD4 (single 4-element structure to all lanes)

Advanced SIMD load/store single structure to one lane

This section describes the encoding of the Advanced SIMD load/store single structure to one lane instruction class. The encodings in this section are decoded from *Advanced SIMD element or structure load/store* on page F3-4160.

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 10 9 8	7 4	3 0
1 1 1 1	1 0 0 1 1	D L 0	Rn	Vd	!=11 size N	index_align	Rm

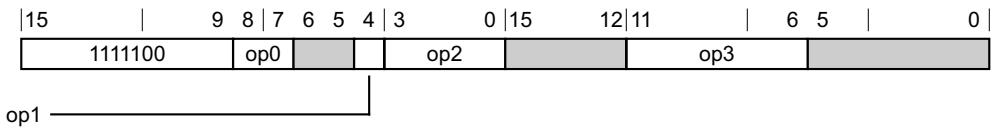
Decode fields			Instruction page
L	size	N	
0	00	00	VST1 (single element from one lane) - *T1* on page F6-5576
0	00	01	VST2 (single 2-element structure from one lane) - *T1* on page F6-5590
0	00	10	VST3 (single 3-element structure from one lane) - *T1* on page F6-5601
0	00	11	VST4 (single 4-element structure from one lane) - *T1* on page F6-5610
0	01	00	VST1 (single element from one lane) - *T2* on page F6-5577
0	01	01	VST2 (single 2-element structure from one lane) - *T2* on page F6-5591
0	01	10	VST3 (single 3-element structure from one lane) - *T2* on page F6-5602
0	01	11	VST4 (single 4-element structure from one lane) - *T2* on page F6-5611
0	10	00	VST1 (single element from one lane) - *T3* on page F6-5577
0	10	01	VST2 (single 2-element structure from one lane) - *T3* on page F6-5591
0	10	10	VST3 (single 3-element structure from one lane) - *T3* on page F6-5602
0	10	11	VST4 (single 4-element structure from one lane) - *T3* on page F6-5611
1	00	00	VLD1 (single element to one lane) - *T1* on page F6-5206

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields			Instruction page
L	size	N	
1	00	01	VLD2 (single 2-element structure to one lane) - *T1* on page F6-5223
1	00	10	VLD3 (single 3-element structure to one lane) - *T1* on page F6-5237
1	00	11	VLD4 (single 4-element structure to one lane) - *T1* on page F6-5249
1	01	00	VLD1 (single element to one lane) - *T2* on page F6-5207
1	01	01	VLD2 (single 2-element structure to one lane) - *T2* on page F6-5224
1	01	10	VLD3 (single 3-element structure to one lane) - *T2* on page F6-5238
1	01	11	VLD4 (single 4-element structure to one lane) - *T2* on page F6-5250
1	10	00	VLD1 (single element to one lane) - *T3* on page F6-5207
1	10	01	VLD2 (single 2-element structure to one lane) - *T3* on page F6-5224
1	10	10	VLD3 (single 3-element structure to one lane) - *T3* on page F6-5238
1	10	11	VLD4 (single 4-element structure to one lane) - *T3* on page F6-5250

F3.1.19 Load/store single

This section describes the encoding of the Load/store single group. The encodings in this section are decoded from *32-bit* on page F3-4119.



This decode also imposes the constraint:

- `op0<1>:op1 != 10`.

Table F3-20 Encoding table for the Load/store single group

Decode fields				Decode group or instruction page
op0	op1	op2	op3	
00	-	!= 1111	000000	*Load/store, unsigned (register offset)* on page F3-4164
00	-	!= 1111	000001	Unallocated.
00	-	!= 1111	00001x	Unallocated.
00	-	!= 1111	0001xx	Unallocated.
00	-	!= 1111	001xxx	Unallocated.
00	-	!= 1111	01xxxx	Unallocated.
00	-	!= 1111	10x0xx	Unallocated.
00	-	!= 1111	10x1xx	*Load/store, unsigned (immediate, post-indexed)* on page F3-4165

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table F3-20 Encoding table for the Load/store single group (continued)

Decode fields				Decode group or instruction page
op0	op1	op2	op3	
00	-	!= 1111	1100xx	*Load/store, unsigned (negative immediate)* on page F3-4165
00	-	!= 1111	1110xx	*Load/store, unsigned (unprivileged)* on page F3-4166
00	-	!= 1111	11x1xx	*Load/store, unsigned (immediate, pre-indexed)* on page F3-4166
01	-	!= 1111	-	*Load/store, unsigned (positive immediate)* on page F3-4167
0x	-	1111	-	*Load, unsigned (literal)* on page F3-4167
10	1	!= 1111	000000	*Load/store, signed (register offset)* on page F3-4168
10	1	!= 1111	000001	Unallocated.
10	1	!= 1111	00001x	Unallocated.
10	1	!= 1111	0001xx	Unallocated.
10	1	!= 1111	001xxx	Unallocated.
10	1	!= 1111	01xxxx	Unallocated.
10	1	!= 1111	10x0xx	Unallocated.
10	1	!= 1111	10x1xx	*Load/store, signed (immediate, post-indexed)* on page F3-4168
10	1	!= 1111	1100xx	*Load/store, signed (negative immediate)* on page F3-4169
10	1	!= 1111	1110xx	*Load/store, signed (unprivileged)* on page F3-4169
10	1	!= 1111	11x1xx	*Load/store, signed (immediate, pre-indexed)* on page F3-4170
11	1	!= 1111	-	*Load/store, signed (positive immediate)* on page F3-4170
1x	1	1111	-	*Load, signed (literal)* on page F3-4171

Load/store, unsigned (register offset)

This section describes the encoding of the Load/store, unsigned (register offset) instruction class. The encodings in this section are decoded from *Load/store single* on page F3-4163.

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 10 9 8	7 6 5 4	3 0
1 1 1 1	1 0 0 0 0	size L	!=1111 (Rn)	Rt	0 0 0 0 0 0	imm2	Rm

Decode fields			Instruction page
size	L	Rt	
00	0	-	STRB (register)
00	1	!= 1111	LDRB (register)
00	1	1111	PLD, PLDW (register) - Preload read variant
01	0	-	STRH (register)

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields			Instruction page
size	L	Rt	
01	1	!= 1111	LDRH (register)
01	1	1111	PLD, PLDW (register) - Preload write variant
10	0	-	STR (register)
10	1	-	LDR (register)
11	-	-	Unallocated.

Load/store, unsigned (immediate, post-indexed)

This section describes the encoding of the Load/store, unsigned (immediate, post-indexed) instruction class. The encodings in this section are decoded from *Load/store single* on page F3-4163.

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7 … 0
1	1	1	1	1	0	0	0	0	size		L	!=1111 (Rn)	Rt	1	0	U	1	imm8

Decode fields		Instruction page
size	L	
00	0	STRB (immediate)
00	1	LDRB (immediate)
01	0	STRH (immediate)
01	1	LDRH (immediate)
10	0	STR (immediate)
10	1	LDR (immediate)
11	-	Unallocated.

Load/store, unsigned (negative immediate)

This section describes the encoding of the Load/store, unsigned (negative immediate) instruction class. The encodings in this section are decoded from *Load/store single* on page F3-4163.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

15 14 13 12	11 10 9 8	7 6 5 4	3 … 0	15 … 12	11 10 9 8	7 … 0
1 1 1 1	1 0 0 0	0 size	L	!=1111 (Rn)	Rt	1 1 0 0	imm8

Decode fields			Instruction page
size	L	Rt	
00	0	-	STRB (immediate)
00	1	!= 1111	LDRB (immediate)
00	1	1111	PLD, PLDW (immediate) - Preload read variant
01	0	-	STRH (immediate)
01	1	!= 1111	LDRH (immediate)
01	1	1111	PLD, PLDW (immediate) - Preload write variant
10	0	-	STR (immediate)
10	1	-	LDR (immediate)
11	-	-	Unallocated.

Load/store, unsigned (unprivileged)

This section describes the encoding of the Load/store, unsigned (unprivileged) instruction class. The encodings in this section are decoded from *Load/store single* on page F3-4163.

15 14 13 12	11 10 9 8	7 6 5 4	3 … 0	15 … 12	11 10 9 8	7 … 0
1 1 1 1	1 0 0 0	0 size	L	!=1111 (Rn)	Rt	1 1 1 0	imm8

Decode fields		Instruction page
size	L	
00	0	STRBT
00	1	LDRBT
01	0	STRHT
01	1	LDRHT
10	0	STRT
10	1	LDRT
11	-	Unallocated.

Load/store, unsigned (immediate, pre-indexed)

This section describes the encoding of the Load/store, unsigned (immediate, pre-indexed) instruction class. The encodings in this section are decoded from *Load/store single* on page F3-4163.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

15 14 13 12	11 10 9 8	7 6 5 4	3 … 0	15 … 12	11 10 9 8	7 … 0
1 1 1 1	1 0 0 0	0 size	L !=1111 (Rn)	Rt	1 1 U 1	imm8

Decode fields		Instruction page
size	L	
00	0	STRB (immediate)
00	1	LDRB (immediate)
01	0	STRH (immediate)
01	1	LDRH (immediate)
10	0	STR (immediate)
10	1	LDR (immediate)
11	-	Unallocated.

Load/store, unsigned (positive immediate)

This section describes the encoding of the Load/store, unsigned (positive immediate) instruction class. The encodings in this section are decoded from *Load/store single* on page F3-4163.

15 14 13 12	11 10 9 8	7 6 5 4	3 … 0	15 … 12	11 … 0
1 1 1 1	1 0 0 0	1 size	L !=1111 (Rn)	Rt	imm12

Decode fields			Instruction page
size	L	Rt	
00	0	-	STRB (immediate)
00	1	!= 1111	LDRB (immediate)
00	1	1111	PLD, PLDW (immediate) - Preload read variant
01	0	-	STRH (immediate)
01	1	!= 1111	LDRH (immediate)
01	1	1111	PLD, PLDW (immediate) - Preload write variant
10	0	-	STR (immediate)
10	1	-	LDR (immediate)

Load, unsigned (literal)

This section describes the encoding of the Load, unsigned (literal) instruction class. The encodings in this section are decoded from *Load/store single* on page F3-4163.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15 – 12	11 – 0
1	1	1	1	1	0	0	0	U	size		L	1	1	1	1	Rt	imm12

Decode fields			Instruction page
size	L	Rt	
0x	1	1111	PLD (literal)
00	1	!= 1111	LDRB (literal)
01	1	!= 1111	LDRH (literal)
10	1	-	LDR (literal)
11	-	-	Unallocated.

Load/store, signed (register offset)

This section describes the encoding of the Load/store, signed (register offset) instruction class. The encodings in this section are decoded from *Load/store single* on page F3-4163.

15	14	13	12	11	10	9	8	7	6	5	4	3 – 0	15 – 12	11	10	9	8	7	6	5	4	3 – 0
1	1	1	1	1	0	0	1	0	size		1	!=1111 (Rn)	Rt	0	0	0	0	0	0	imm2		Rm

Decode fields		Instruction page
size	Rt	
00	!= 1111	LDRSB (register)
00	1111	PLI (register)
01	!= 1111	LDRSH (register)
01	1111	Reserved hint, behaves as NOP.
1x	-	Unallocated.

Load/store, signed (immediate, post-indexed)

This section describes the encoding of the Load/store, signed (immediate, post-indexed) instruction class. The encodings in this section are decoded from *Load/store single* on page F3-4163.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

15 14 13 12	11 10 9 8	7 6 5 4	3 … 0	15 … 12	11 10 9 8	7 … 0
1 1 1 1	1 0 0 1 0	size	1	!=1111 (Rn)	Rt	1 0 U 1	imm8

Decode fields	Instruction page
size	
00	LDRSB (immediate)
01	LDRSH (immediate)
1x	Unallocated.

Load/store, signed (negative immediate)

This section describes the encoding of the Load/store, signed (negative immediate) instruction class. The encodings in this section are decoded from *Load/store single* on page F3-4163.

15 14 13 12	11 10 9 8	7 6 5 4	3 … 0	15 … 12	11 10 9 8	7 … 0
1 1 1 1	1 0 0 1 0	size	1	!=1111 (Rn)	Rt	1 1 0 0	imm8

Decode fields		Instruction page
size	Rt	
00	!= 1111	LDRSB (immediate)
00	1111	PLI (immediate, literal)
01	!= 1111	LDRSH (immediate)
01	1111	Reserved hint, behaves as NOP.
1x	-	Unallocated.

Load/store, signed (unprivileged)

This section describes the encoding of the Load/store, signed (unprivileged) instruction class. The encodings in this section are decoded from *Load/store single* on page F3-4163.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

15 14 13 12	11 10 9 8	7 6 5 4	3 … 0	15 … 12	11 10 9 8	7 … 0
1 1 1 1	1 0 0 1 0	size	1	!=1111	Rt	1 1 1 0	imm8
			Rn			

Decode fields	Instruction page
size	
00	LDRSBT
01	LDRSHT
1x	Unallocated.

Load/store, signed (immediate, pre-indexed)

This section describes the encoding of the Load/store, signed (immediate, pre-indexed) instruction class. The encodings in this section are decoded from *Load/store single* on page F3-4163.

15 14 13 12	11 10 9 8	7 6 5 4	3 … 0	15 … 12	11 10 9 8	7 … 0
1 1 1 1	1 0 0 1 0	size	1	!=1111	Rt	1 1 U 1	imm8
			Rn			

Decode fields	Instruction page
size	
00	LDRSB (immediate)
01	LDRSH (immediate)
1x	Unallocated.

Load/store, signed (positive immediate)

This section describes the encoding of the Load/store, signed (positive immediate) instruction class. The encodings in this section are decoded from *Load/store single* on page F3-4163.

15 14 13 12	11 10 9 8	7 6 5 4	3 … 0	15 … 12	11 … 0
1 1 1 1	1 0 0 1 1	size	1	!=1111	Rt	imm12
			Rn		

Decode fields		Instruction page
size	Rt	
00	!= 1111	LDRSB (immediate)
00	1111	PLI (immediate, literal)
01	!= 1111	LDRSH (immediate)
01	1111	Reserved hint, behaves as NOP.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Load, signed (literal)

This section describes the encoding of the Load, signed (literal) instruction class. The encodings in this section are decoded from *Load/store single* on page F3-4163.

15 14 13 12	11 10 9 8	7	6 5	4 3 2 1	0	15 12	11 0
1 1 1 1	1 0 0 1	U	size	1 1 1 1	1	Rt	imm12

Decode fields		Instruction page
size	Rt	
00	!= 1111	LDRSB (literal)
00	1111	PLI (immediate, literal)
01	!= 1111	LDRSH (literal)
01	1111	Reserved hint, behaves as NOP.
1x	-	Unallocated.

F3.1.20 Data-processing (register)

This section describes the encoding of the Data-processing (register) group. The encodings in this section are decoded from *32-bit* on page F3-4119.

15 8	7	6 0	15 12	11 8	7 4	3 0
11111010			1111		op1	

op0

Table F3-21 Encoding table for the Data-processing (register) group

Decode fields		Decode group or instruction page
op0	op1	
0	0000	MOV, MOVS (register-shifted register) - *Flag setting variant* on page F5-4523
0	0001	Unallocated.
0	001x	Unallocated.
0	01xx	Unallocated.
0	1xxx	*Register extends*
1	0xxx	*Parallel add-subtract* on page F3-4172
1	10xx	*Data-processing (two source registers)* on page F3-4174
1	11xx	Unallocated.

Register extends

This section describes the encoding of the Register extends instruction class. The encodings in this section are decoded from *Data-processing (register)*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 14 13 12	11 8	7	6 5 4	3 0
1 1 1 1	1 0 1 0	0 op1	U Rn	1 1 1 1	Rd	1	(0) rotate	Rm

Decode fields			Instruction page
op1	U	Rn	
00	0	!= 1111	SXTAH
00	0	1111	SXTH
00	1	!= 1111	UXTAH
00	1	1111	UXTH
01	0	!= 1111	SXTAB16
01	0	1111	SXTB16
01	1	!= 1111	UXTAB16
01	1	1111	UXTB16
10	0	!= 1111	SXTAB
10	0	1111	SXTB
10	1	!= 1111	UXTAB
10	1	1111	UXTB
11	-	-	Unallocated.

Parallel add-subtract

This section describes the encoding of the Parallel add-subtract instruction class. The encodings in this section are decoded from *Data-processing (register)* on page F3-4171.

15 14 13 12	11 10 9 8	7 6 4	3 0	15 14 13 12	11 8	7	6	5	4	3 0
1 1 1 1	1 0 1 0	1 op1	Rn	1 1 1 1	Rd	0	U	H	S	Rm

Decode fields				Instruction page
op1	U	H	S	
000	0	0	0	SADD8
000	0	0	1	QADD8
000	0	1	0	SHADD8
000	0	1	1	Unallocated.
000	1	0	0	UADD8
000	1	0	1	UQADD8

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields				Instruction page
op1	U	H	S	
000	1	1	0	UHADD8
000	1	1	1	Unallocated.
001	0	0	0	SADD16
001	0	0	1	QADD16
001	0	1	0	SHADD16
001	0	1	1	Unallocated.
001	1	0	0	UADD16
001	1	0	1	UQADD16
001	1	1	0	UHADD16
001	1	1	1	Unallocated.
010	0	0	0	SASX
010	0	0	1	QASX
010	0	1	0	SHASX
010	0	1	1	Unallocated.
010	1	0	0	UASX
010	1	0	1	UQASX
010	1	1	0	UHASX
010	1	1	1	Unallocated.
100	0	0	0	SSUB8
100	0	0	1	QSUB8
100	0	1	0	SHSUB8
100	0	1	1	Unallocated.
100	1	0	0	USUB8
100	1	0	1	UQSUB8
100	1	1	0	UHSUB8
100	1	1	1	Unallocated.
101	0	0	0	SSUB16
101	0	0	1	QSUB16
101	0	1	0	SHSUB16
101	0	1	1	Unallocated.
101	1	0	0	USUB16
101	1	0	1	UQSUB16

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields				Instruction page
op1	U	H	S	
101	1	1	0	UHSUB16
101	1	1	1	Unallocated.
110	0	0	0	SSAX
110	0	0	1	QSAX
110	0	1	0	SHSAX
110	0	1	1	Unallocated.
110	1	0	0	USAX
110	1	0	1	UQSAX
110	1	1	0	UHSAX
110	1	1	1	Unallocated.
111	-	-	-	Unallocated.

Data-processing (two source registers)

This section describes the encoding of the Data-processing (two source registers) instruction class. The encodings in this section are decoded from *Data-processing (register)* on page F3-4171.

15 14 13 12	11 10 9 8	7 6 4	3 0	15 14 13 12	11 8	7 6	5 4	3 0
1 1 1 1	1 0 1 0	1 op1	Rn	1 1 1 1	Rd	1 0	op2	Rm

Decode fields		Instruction page
op1	op2	
000	00	QADD
000	01	QDADD
000	10	QSUB
000	11	QDSUB
001	00	REV
001	01	REV16
001	10	RBIT
001	11	REVSH
010	00	SEL
010	01	Unallocated.
010	1x	Unallocated.
011	00	CLZ

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields		Instruction page
op1	op2	
011	01	Unallocated.
011	1x	Unallocated.
100	00	CRC32 - CRC32B variant
100	01	CRC32 - CRC32H variant
100	10	CRC32 - CRC32W variant
100	11	CONSTRAINED UNPREDICTABLE
101	00	CRC32C - CRC32CB variant
101	01	CRC32C - CRC32CH variant
101	10	CRC32C - CRC32CW variant
101	11	CONSTRAINED UNPREDICTABLE
11x	-	Unallocated.

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in *CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings* on page K1-7842.

F3.1.21 Multiply, multiply accumulate, and absolute difference

This section describes the encoding of the Multiply, multiply accumulate, and absolute difference group. The encodings in this section are decoded from *32-bit* on page F3-4119.

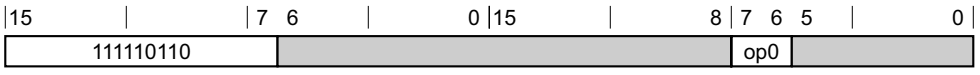


Table F3-22 Encoding table for the Multiply, multiply accumulate, and absolute difference group

Decode fields	Decode group or instruction page
op0	
00	*Multiply and absolute difference*
01	Unallocated.
1x	Unallocated.

Multiply and absolute difference

This section describes the encoding of the Multiply and absolute difference instruction class. The encodings in this section are decoded from *Multiply, multiply accumulate, and absolute difference*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

15 14 13 12	11 10 9 8	7 6 4	3 0	15 12	11 8	7 6	5 4	3 0
1 1 1 1	1 0 1 1	0	op1	Rn	Ra	Rd	0 0	op2	Rm

Decode fields			Instruction page
op1	Ra	op2	
000	!= 1111	00	MLA, MLAS
000	-	01	MLS
000	-	1x	Unallocated.
000	1111	00	MUL, MULS
001	!= 1111	00	SMLABB, SMLABT, SMLATB, SMLATT - SMLABB variant
001	!= 1111	01	SMLABB, SMLABT, SMLATB, SMLATT - SMLABT variant
001	!= 1111	10	SMLABB, SMLABT, SMLATB, SMLATT - SMLATB variant
001	!= 1111	11	SMLABB, SMLABT, SMLATB, SMLATT - SMLATT variant
001	1111	00	SMULBB, SMULBT, SMULTB, SMULTT - SMULBB variant
001	1111	01	SMULBB, SMULBT, SMULTB, SMULTT - SMULBT variant
001	1111	10	SMULBB, SMULBT, SMULTB, SMULTT - SMULTB variant
001	1111	11	SMULBB, SMULBT, SMULTB, SMULTT - SMULTT variant
010	!= 1111	00	SMLAD, SMLADX - SMLAD variant
010	!= 1111	01	SMLAD, SMLADX - SMLADX variant
010	-	1x	Unallocated.
010	1111	00	SMUAD, SMUADX - SMUAD variant
010	1111	01	SMUAD, SMUADX - SMUADX variant
011	!= 1111	00	SMLAWB, SMLAWT - SMLAWB variant
011	!= 1111	01	SMLAWB, SMLAWT - SMLAWT variant
011	-	1x	Unallocated.
011	1111	00	SMULWB, SMULWT - SMULWB variant
011	1111	01	SMULWB, SMULWT - SMULWT variant
100	!= 1111	00	SMLSD, SMLSDX - SMLSD variant
100	!= 1111	01	SMLSD, SMLSDX - SMLSDX variant
100	-	1x	Unallocated.
100	1111	00	SMUSD, SMUSDX - SMUSD variant
100	1111	01	SMUSD, SMUSDX - SMUSDX variant
101	!= 1111	00	SMMLA, SMMLAR - SMMLA variant

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields			Instruction page
op1	Ra	op2	
101	!= 1111	01	SMMLA, SMMLAR - SMMLAR variant
101	-	1x	Unallocated.
101	1111	00	SMMUL, SMMULR - SMMUL variant
101	1111	01	SMMUL, SMMULR - SMMULR variant
110	-	00	SMMLS, SMMLSR - SMMLS variant
110	-	01	SMMLS, SMMLSR - SMMLSR variant
110	-	1x	Unallocated.
111	!= 1111	00	USADA8
111	-	01	Unallocated.
111	-	1x	Unallocated.
111	1111	00	USAD8

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F3.2 About the T32 Advanced SIMD and floating-point instructions and their encoding

The Advanced SIMD and floating-point instructions are common to the T32 and A32 instruction sets. These instructions perform Advanced SIMD and floating-point operations on a common register file, the SIMD&FP register file. This means:

- In general, the instructions that load or store registers in this file, or move data between general-purpose registers and this register file, are common to the Advanced SIMD and floating-point instructions.
- There are distinct Advanced SIMD data-processing instructions and floating-point data-processing instructions.

All T32 Advanced SIMD and floating-point instructions have 32-bit encodings. Different groups of these instructions are decoded from different points in the 32-bit T32 instruction decode structure. Table F3-23 shows these instruction groups, and where each group is decoded from the overall T32 decode structure:

Table F3-23 Advanced SIMD and floating-point instructions in the T32 decode structure

Advanced SIMD and floating-point instruction group	T32 decode is from
Advanced SIMD load/store and 64-bit move on page F3-4147	*System register access, Advanced SIMD, and floating-point* on page F3-4124
Floating-point data-processing on page F3-4131	*System register access, Advanced SIMD, and floating-point* on page F3-4124
Advanced SIMD and floating-point 32-bit move on page F3-4149	*System register access, Advanced SIMD, and floating-point* on page F3-4124
Advanced SIMD data-processing on page F3-4137	*System register access, Advanced SIMD, and floating-point* on page F3-4124
Advanced SIMD element or structure load/store on page F3-4160	*32-bit* on page F3-4119

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter F4
A32 Instruction Set Encoding

This chapter describes the encoding of the A32 instruction set. It contains the following sections:

- *A32 instruction set encoding* on page F4-4180.
- *About the A32 Advanced SIMD and floating-point instructions and their encoding* on page F4-4240.

In this chapter:

- In the decode tables, an entry of - for a field value means the value of the field does not affect the decoding.
- In the decode diagrams, a shaded field indicates that the bits in that field are not used in that level of decode.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F4.1 A32 instruction set encoding

The A32 instruction stream is a sequence of word-aligned words. Each A32 instruction is either a single 32-bit word in that stream.

Most A32 instructions can be conditional, with a condition determined by bits[31:28] of the instruction, the `cond` field. For more information see *The Condition code field in A32 instruction encodings* on page F2-4082. This applies to all instructions except those with the `cond` field equal to `0b111`.

The behavior of an attempt to execute an unallocated instruction is described in *UNDEFINED, UNPREDICTABLE, and CONSTRAINED UNPREDICTABLE instruction set space* on page F2-4088.

For more information on A32 instruction encodings see Chapter F2 *About the T32 and A32 Instruction Descriptions*.

The A32 instruction encoding is:

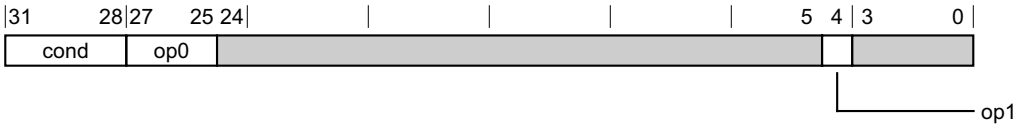


Table F4-1 Main encoding table for the A32 instruction set

Decode fields			Decode group or instruction page
cond	**op0**	**op1**	
`!= 1111`	`00x`	`-`	*Data-processing and miscellaneous instructions*
`!= 1111`	`010`	`-`	*Load/Store Word, Unsigned Byte (immediate, literal)* on page F4-4196
`!= 1111`	`011`	`0`	*Load/Store Word, Unsigned Byte (register)* on page F4-4197
`!= 1111`	`011`	`1`	*Media instructions* on page F4-4198
`-`	`10x`	`-`	*Branch, branch with link, and block data transfer* on page F4-4206
`-`	`11x`	`-`	*System register access, Advanced SIMD, floating-point, and Supervisor call* on page F4-4208
`1111`	`0xx`	`-`	*Unconditional instructions* on page F4-4223

F4.1.1 Data-processing and miscellaneous instructions

This section describes the encoding of the Data-processing and miscellaneous instructions group. The encodings in this section are decoded from *A32 instruction set encoding*.

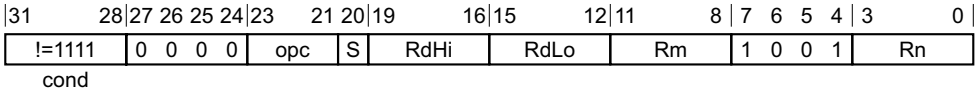
Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table F4-2 Encoding table for the Data-processing and miscellaneous instructions group

Decode fields					Decode group or instruction page
op0	op1	op2	op3	op4	
0	-	1	!= 00	1	*Extra load/store* on page F4-4182
0	0xxxx	1	00	1	*Multiply and Accumulate*
0	1xxxx	1	00	1	*Synchronization primitives and Load-Acquire/Store-Release* on page F4-4185
0	10xx0	0	-	-	*Miscellaneous* on page F4-4186
0	10xx0	1	-	0	*Halfword Multiply and Accumulate* on page F4-4182
0	!= 10xx0	-	-	0	*Data-processing register (immediate shift)* on page F4-4189
0	!= 10xx0	0	-	1	*Data-processing register (register shift)* on page F4-4191
1	-	-	-	-	*Data-processing immediate* on page F4-4193

Multiply and Accumulate

This section describes the encoding of the Multiply and Accumulate instruction class. The encodings in this section are decoded from *Data-processing and miscellaneous instructions* on page F4-4180.



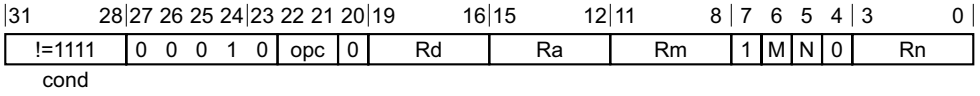
Decode fields		Instruction page
opc	S	
000	-	MUL, MULS
001	-	MLA, MLAS
010	0	UMAAL
010	1	Unallocated.
011	0	MLS
011	1	Unallocated.
100	-	UMULL, UMULLS

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields		Instruction page
opc	S	
101	-	UMLAL, UMLALS
110	-	SMULL, SMULLS
111	-	SMLAL, SMLALS

Halfword Multiply and Accumulate

This section describes the encoding of the Halfword Multiply and Accumulate instruction class. The encodings in this section are decoded from *Data-processing and miscellaneous instructions* on page F4-4180.



Decode fields			Instruction page
opc	M	N	
00	-	-	SMLABB, SMLABT, SMLATB, SMLATT
01	0	0	SMLAWB, SMLAWT - SMLAWB variant
01	0	1	SMULWB, SMULWT - SMULWB variant
01	1	0	SMLAWB, SMLAWT - SMLAWT variant
01	1	1	SMULWB, SMULWT - SMULWT variant
10	-	-	SMLALBB, SMLALBT, SMLALTB, SMLALTT
11	-	-	SMULBB, SMULBT, SMULTB, SMULTT

F4.1.2 Extra load/store

This section describes the encoding of the Extra load/store group. The encodings in this section are decoded from *Data-processing and miscellaneous instructions* on page F4-4180.

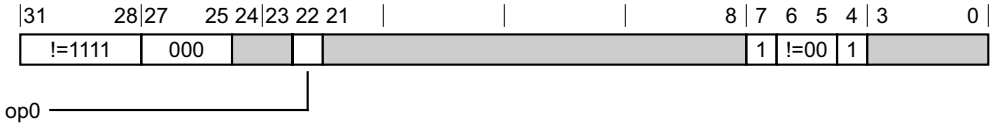


Table F4-3 Encoding table for the Extra load/store group

Decode fields	Decode group or instruction page
op0	
0	*Load/Store Dual, Half, Signed Byte (register)* on page F4-4183
1	*Load/Store Dual, Half, Signed Byte (immediate, literal)* on page F4-4183

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Load/Store Dual, Half, Signed Byte (register)

This section describes the encoding of the Load/Store Dual, Half, Signed Byte (register) instruction class. The encodings in this section are decoded from *Extra load/store* on page F4-4182.

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	0	0	0	P	U	0	W	o1	Rn	Rt	(0)	(0)	(0)	(0)	1	!=00		1	Rm
cond																op2			

Decode fields				Instruction page
P	W	o1	op2	
0	0	0	01	STRH (register) - Post-indexed variant
0	0	0	10	LDRD (register) - Post-indexed variant
0	0	0	11	STRD (register) - Post-indexed variant
0	0	1	01	LDRH (register) - Post-indexed variant
0	0	1	10	LDRSB (register) - Post-indexed variant
0	0	1	11	LDRSH (register) - Post-indexed variant
0	1	0	01	STRHT
0	1	0	10	Unallocated.
0	1	0	11	Unallocated.
0	1	1	01	LDRHT
0	1	1	10	LDRSBT
0	1	1	11	LDRSHT
1	-	0	01	STRH (register) - Pre-indexed variant
1	-	0	10	LDRD (register) - Pre-indexed variant
1	-	0	11	STRD (register) - Pre-indexed variant
1	-	1	01	LDRH (register) - Pre-indexed variant
1	-	1	10	LDRSB (register) - Pre-indexed variant
1	-	1	11	LDRSH (register) - Pre-indexed variant

Load/Store Dual, Half, Signed Byte (immediate, literal)

This section describes the encoding of the Load/Store Dual, Half, Signed Byte (immediate, literal) instruction class. The encodings in this section are decoded from *Extra load/store* on page F4-4182.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6–5	4	3–0
!=1111	0	0	0	P	U	1	W	o1	Rn	Rt	imm4H	1	!=00	1	imm4L
cond													op2		

Decode fields				Instruction page
P:W	**o1**	**Rn**	**op2**	
-	0	1111	10	LDRD (literal)
!= 01	1	1111	01	LDRH (literal)
!= 01	1	1111	10	LDRSB (literal)
!= 01	1	1111	11	LDRSH (literal)
00	0	!= 1111	10	LDRD (immediate) - Post-indexed variant
00	0	-	01	STRH (immediate) - Post-indexed variant
00	0	-	11	STRD (immediate) - Post-indexed variant
00	1	!= 1111	01	LDRH (immediate) - Post-indexed variant
00	1	!= 1111	10	LDRSB (immediate) - Post-indexed variant
00	1	!= 1111	11	LDRSH (immediate) - Post-indexed variant
01	0	!= 1111	10	Unallocated.
01	0	-	01	STRHT
01	0	-	11	Unallocated.
01	1	-	01	LDRHT
01	1	-	10	LDRSBT
01	1	-	11	LDRSHT
10	0	!= 1111	10	LDRD (immediate) - Offset variant
10	0	-	01	STRH (immediate) - Offset variant
10	0	-	11	STRD (immediate) - Offset variant
10	1	!= 1111	01	LDRH (immediate) - Offset variant
10	1	!= 1111	10	LDRSB (immediate) - Offset variant
10	1	!= 1111	11	LDRSH (immediate) - Offset variant
11	0	!= 1111	10	LDRD (immediate) - Pre-indexed variant
11	0	-	01	STRH (immediate) - Pre-indexed variant
11	0	-	11	STRD (immediate) - Pre-indexed variant
11	1	!= 1111	01	LDRH (immediate) - Pre-indexed variant
11	1	!= 1111	10	LDRSB (immediate) - Pre-indexed variant
11	1	!= 1111	11	LDRSH (immediate) - Pre-indexed variant

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F4.1.3 Synchronization primitives and Load-Acquire/Store-Release

This section describes the encoding of the Synchronization primitives and Load-Acquire/Store-Release group. The encodings in this section are decoded from *Data-processing and miscellaneous instructions* on page F4-4180.

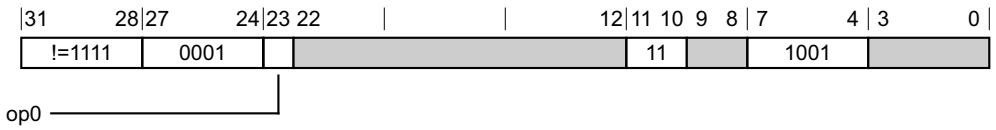
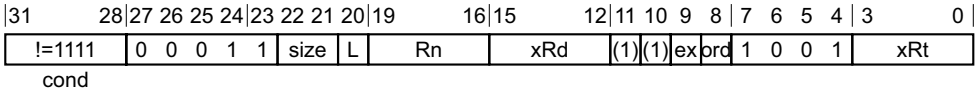


Table F4-4 Encoding table for the Synchronization primitives and Load-Acquire/Store-Release group

Decode fields op0	Decode group or instruction page
0	Unallocated.
1	*Load/Store Exclusive and Load-Acquire/Store-Release*

Load/Store Exclusive and Load-Acquire/Store-Release

This section describes the encoding of the Load/Store Exclusive and Load-Acquire/Store-Release instruction class. The encodings in this section are decoded from *Synchronization primitives and Load-Acquire/Store-Release*.



Decode fields size	L	ex	ord	Instruction page
00	0	0	0	STL
00	0	0	1	Unallocated.
00	0	1	0	STLEX
00	0	1	1	STREX
00	1	0	0	LDA
00	1	0	1	Unallocated.
00	1	1	0	LDAEX
00	1	1	1	LDREX
01	0	0	-	Unallocated.
01	0	1	0	STLEXD
01	0	1	1	STREXD
01	1	0	-	Unallocated.
01	1	1	0	LDAEXD

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields				Instruction page
size	L	ex	ord	
01	1	1	1	LDREXD
10	0	0	0	STLB
10	0	0	1	Unallocated.
10	0	1	0	STLEXB
10	0	1	1	STREXB
10	1	0	0	LDAB
10	1	0	1	Unallocated.
10	1	1	0	LDAEXB
10	1	1	1	LDREXB
11	0	0	0	STLH
11	0	0	1	Unallocated.
11	0	1	0	STLEXH
11	0	1	1	STREXH
11	1	0	0	LDAH
11	1	0	1	Unallocated.
11	1	1	0	LDAEXH
11	1	1	1	LDREXH

F4.1.4 Miscellaneous

This section describes the encoding of the Miscellaneous group. The encodings in this section are decoded from *Data-processing and miscellaneous instructions* on page F4-4180.

31–28	27–23	22 21 20	19–8	7	6–4	3–0
!=1111	00010	op0	0	0	op1	

Table F4-5 Encoding table for the Miscellaneous group

Decode fields		Decode group or instruction page
op0	op1	
00	001	Unallocated.
00	010	Unallocated.
00	011	Unallocated.
00	110	Unallocated.
01	001	BX

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table F4-5 Encoding table for the Miscellaneous group (continued)

Decode fields		Decode group or instruction page
op0	op1	
01	010	BXJ
01	011	BLX (register)
01	110	Unallocated.
10	001	Unallocated.
10	010	Unallocated.
10	011	Unallocated.
10	110	Unallocated.
11	001	CLZ
11	010	Unallocated.
11	011	Unallocated.
11	110	ERET
-	111	*Exception Generation*
-	000	*Move special register (register)*
-	100	*Cyclic Redundancy Check* on page F4-4188
-	101	*Integer Saturating Arithmetic* on page F4-4188

Exception Generation

This section describes the encoding of the Exception Generation instruction class. The encodings in this section are decoded from *Miscellaneous* on page F4-4186.

31–28	27 26 25 24	23 22 21 20	19–8	7 6 5 4	3–0
!=1111	0 0 0 1 0	opc	0	imm12	0 1 1 1	imm4
cond					

Decode fields	Instruction page
opc	
00	HLT
01	BKPT
10	HVC
11	SMC

Move special register (register)

This section describes the encoding of the Move special register (register) instruction class. The encodings in this section are decoded from *Miscellaneous* on page F4-4186.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	0	0	0	1	0	opc		0	mask	Rd	(0)	(0)	B	m	0	0	0	0	Rn
cond																			

Decode fields		Instruction page
opc	B	
x0	0	MRS
x0	1	MRS (Banked register)
x1	0	MSR (register)
x1	1	MSR (Banked register)

Cyclic Redundancy Check

This section describes the encoding of the Cyclic Redundancy Check instruction class. The encodings in this section are decoded from *Miscellaneous* on page F4-4186.

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	0	0	0	1	0	sz		0	Rn	Rd	(0)	(0)	C	(0)	0	1	0	0	Rm
cond																			

Decode fields		Instruction page
sz	C	
00	0	CRC32 - CRC32B variant
00	1	CRC32C - CRC32CB variant
01	0	CRC32 - CRC32H variant
01	1	CRC32C - CRC32CH variant
10	0	CRC32 - CRC32W variant
10	1	CRC32C - CRC32CW variant
11	-	CONSTRAINED UNPREDICTABLE

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in *CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings* on page K1-7842.

Integer Saturating Arithmetic

This section describes the encoding of the Integer Saturating Arithmetic instruction class. The encodings in this section are decoded from *Miscellaneous* on page F4-4186.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111 (cond)	0	0	0	1	0	opc		0	Rn	Rd	(0)	(0)	(0)	(0)	0	1	0	1	Rm

Decode fields opc	Instruction page
00	QADD
01	QSUB
10	QDADD
11	QDSUB

F4.1.5 Data-processing register (immediate shift)

This section describes the encoding of the Data-processing register (immediate shift) group. The encodings in this section are decoded from *Data-processing and miscellaneous instructions* on page F4-4180.

31–28	27–25	24	23–22	21	20	19–5	4	3–0
!=1111	000	op0			op1		0	

This decode also imposes the constraint:

- `op0:op1 != 100`.

Table F4-6 Encoding table for the Data-processing register (immediate shift) group

Decode fields op0	op1	Decode group or instruction page
0x	-	*Integer Data Processing (three register, immediate shift)*
10	1	*Integer Test and Compare (two register, immediate shift)* on page F4-4190
11	-	*Logical Arithmetic (three register, immediate shift)* on page F4-4191

Integer Data Processing (three register, immediate shift)

This section describes the encoding of the Integer Data Processing (three register, immediate shift) instruction class. The encodings in this section are decoded from *Data-processing register (immediate shift)*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31–28	27 26 25 24	23–21	20	19–16	15–12	11–7	6 5	4	3–0
!=1111	0 0 0 0	opc	S	Rn	Rd	imm5	stype	0	Rm
cond									

Decode fields			Instruction page
opc	S	Rn	
000	-	-	AND, ANDS (register)
001	-	-	EOR, EORS (register)
010	0	!= 1101	SUB, SUBS (register) - SUB, rotate right with extend variant
010	0	1101	SUB, SUBS (SP minus register) - SUB, rotate right with extend variant
010	1	!= 1101	SUB, SUBS (register) - SUBS, rotate right with extend variant
010	1	1101	SUB, SUBS (SP minus register) - SUBS, rotate right with extend variant
011	-	-	RSB, RSBS (register)
100	0	!= 1101	ADD, ADDS (register) - ADD, rotate right with extend variant
100	0	1101	ADD, ADDS (SP plus register) - ADD, rotate right with extend variant
100	1	!= 1101	ADD, ADDS (register) - ADDS, rotate right with extend variant
100	1	1101	ADD, ADDS (SP plus register) - ADDS, rotate right with extend variant
101	-	-	ADC, ADCS (register)
110	-	-	SBC, SBCS (register)
111	-	-	RSC, RSCS (register)

Integer Test and Compare (two register, immediate shift)

This section describes the encoding of the Integer Test and Compare (two register, immediate shift) instruction class. The encodings in this section are decoded from *Data-processing register (immediate shift)* on page F4-4189.

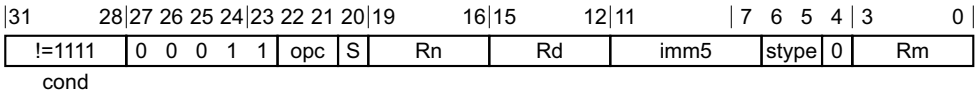
31–28	27 26 25 24	23 22 21	20	19–16	15 14 13 12	11–7	6 5	4	3–0
!=1111	0 0 0 1 0	opc	1	Rn	(0)(0)(0)(0)	imm5	stype	0	Rm
cond									

Decode fields	Instruction page
opc	
00	TST (register)
01	TEQ (register)
10	CMP (register)
11	CMN (register)

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Logical Arithmetic (three register, immediate shift)

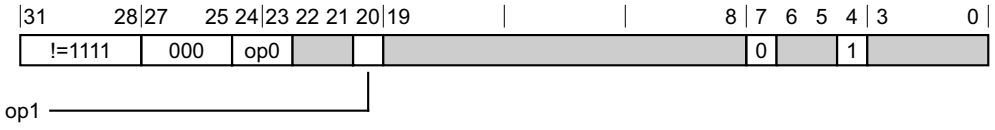
This section describes the encoding of the Logical Arithmetic (three register, immediate shift) instruction class. The encodings in this section are decoded from *Data-processing register (immediate shift)* on page F4-4189.



Decode fields opc	Instruction page
00	ORR, ORRS (register)
01	MOV, MOVS (register)
10	BIC, BICS (register)
11	MVN, MVNS (register)

F4.1.6 Data-processing register (register shift)

This section describes the encoding of the Data-processing register (register shift) group. The encodings in this section are decoded from *Data-processing and miscellaneous instructions* on page F4-4180.



This decode also imposes the constraint:

- `op0:op1 != 100`.

Table F4-7 Encoding table for the Data-processing register (register shift) group

Decode fields op0	op1	Decode group or instruction page
0x	-	*Integer Data Processing (three register, register shift)*
10	1	*Integer Test and Compare (two register, register shift)* on page F4-4192
11	-	*Logical Arithmetic (three register, register shift)* on page F4-4192

Integer Data Processing (three register, register shift)

This section describes the encoding of the Integer Data Processing (three register, register shift) instruction class. The encodings in this section are decoded from *Data-processing register (register shift)*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31–28	27 26 25 24	23–21	20	19–16	15–12	11–8	7	6 5	4	3–0
!=1111	0 0 0 0	opc	S	Rn	Rd	Rs	0	stype	1	Rm
cond										

Decode fields opc	Instruction page
000	AND, ANDS (register-shifted register)
001	EOR, EORS (register-shifted register)
010	SUB, SUBS (register-shifted register)
011	RSB, RSBS (register-shifted register)
100	ADD, ADDS (register-shifted register)
101	ADC, ADCS (register-shifted register)
110	SBC, SBCS (register-shifted register)
111	RSC, RSCS (register-shifted register)

Integer Test and Compare (two register, register shift)

This section describes the encoding of the Integer Test and Compare (two register, register shift) instruction class. The encodings in this section are decoded from *Data-processing register (register shift)* on page F4-4191.

31–28	27 26 25 24	23 22 21	20	19–16	15 14 13 12	11–8	7	6 5	4	3–0
!=1111	0 0 0 1 0	opc	1	Rn	(0)(0)(0)(0)	Rs	0	stype	1	Rm
cond										

Decode fields opc	Instruction page
00	TST (register-shifted register)
01	TEQ (register-shifted register)
10	CMP (register-shifted register)
11	CMN (register-shifted register)

Logical Arithmetic (three register, register shift)

This section describes the encoding of the Logical Arithmetic (three register, register shift) instruction class. The encodings in this section are decoded from *Data-processing register (register shift)* on page F4-4191.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6–5	4	3–0
!=1111	0	0	0	1	1	opc		S	Rn	Rd	Rs	0	stype	1	Rm
cond															

Decode fields opc	Instruction page
00	ORR, ORRS (register-shifted register)
01	MOV, MOVS (register-shifted register)
10	BIC, BICS (register-shifted register)
11	MVN, MVNS (register-shifted register)

F4.1.7 Data-processing immediate

This section describes the encoding of the Data-processing immediate group. The encodings in this section are decoded from *Data-processing and miscellaneous instructions* on page F4-4180.

31–28	27–25	24	23	22	21	20	19–0
!=1111	001	op0			op1		

Table F4-8 Encoding table for the Data-processing immediate group

Decode fields op0	op1	Decode group or instruction page
0x	-	*Integer Data Processing (two register and immediate)*
10	00	*Move Halfword (immediate)* on page F4-4194
10	10	*Move Special Register and Hints (immediate)* on page F4-4195
10	x1	*Integer Test and Compare (one register and immediate)* on page F4-4195
11	-	*Logical Arithmetic (two register and immediate)* on page F4-4196

Integer Data Processing (two register and immediate)

This section describes the encoding of the Integer Data Processing (two register and immediate) instruction class. The encodings in this section are decoded from *Data-processing immediate*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31–28	27	26	25	24	23–21	20	19–16	15–12	11–0
!=1111	0	0	1	0	opc	S	Rn	Rd	imm12
cond									

Decode fields			Instruction page
opc	S	Rn	
000	-	-	AND, ANDS (immediate)
001	-	-	EOR, EORS (immediate)
010	0	!= 11x1	SUB, SUBS (immediate) - SUB variant
010	0	1101	SUB, SUBS (SP minus immediate) - SUB variant
010	0	1111	ADR - *A2* on page F5-4271
010	1	!= 1101	SUB, SUBS (immediate) - SUBS variant
010	1	1101	SUB, SUBS (SP minus immediate) - SUBS variant
011	-	-	RSB, RSBS (immediate)
100	0	!= 11x1	ADD, ADDS (immediate) - ADD variant
100	0	1101	ADD, ADDS (SP plus immediate) - ADD variant
100	0	1111	ADR - *A1* on page F5-4271
100	1	!= 1101	ADD, ADDS (immediate) - ADDS variant
100	1	1101	ADD, ADDS (SP plus immediate) - ADDS variant
101	-	-	ADC, ADCS (immediate)
110	-	-	SBC, SBCS (immediate)
111	-	-	RSC, RSCS (immediate)

Move Halfword (immediate)

This section describes the encoding of the Move Halfword (immediate) instruction class. The encodings in this section are decoded from *Data-processing immediate* on page F4-4193.

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
!=1111	0	0	1	1	0	H	0	0	imm4	Rd	imm12
cond											

Decode fields	Instruction page
H	
0	MOV, MOVS (immediate)
1	MOVT

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Move Special Register and Hints (immediate)

This section describes the encoding of the Move Special Register and Hints (immediate) instruction class. The encodings in this section are decoded from *Data-processing immediate* on page F4-4193.

31–28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11–0
!=1111 (cond)	0	0	1	1	0	R	1	0	imm4	(1)	(1)	(1)	(1)	imm12

Decode fields		Instruction page	Architecture version
R:imm4	imm12		
`!= 00000`	-	MSR (immediate)	-
`00000`	`xxxx00000000`	NOP	-
`00000`	`xxxx00000001`	YIELD	-
`00000`	`xxxx00000010`	WFE	-
`00000`	`xxxx00000011`	WFI	-
`00000`	`xxxx00000100`	SEV	-
`00000`	`xxxx00000101`	SEVL	-
`00000`	`xxxx0000011x`	Reserved hint, behaves as NOP.	-
`00000`	`xxxx00001xxx`	Reserved hint, behaves as NOP.	-
`00000`	`xxxx00010000`	ESB	Armv8.2
`00000`	`xxxx00010001`	Reserved hint, behaves as NOP.	-
`00000`	`xxxx00010010`	TSB CSYNC	Armv8.4
`00000`	`xxxx00010011`	Reserved hint, behaves as NOP.	-
`00000`	`xxxx00010100`	CSDB	-
`00000`	`xxxx00010101`	Reserved hint, behaves as NOP.	-
`00000`	`xxxx00011xxx`	Reserved hint, behaves as NOP.	-
`00000`	`xxxx0001111x`	Reserved hint, behaves as NOP.	-
`00000`	`xxxx001xxxxx`	Reserved hint, behaves as NOP.	-
`00000`	`xxxx01xxxxxx`	Reserved hint, behaves as NOP.	-
`00000`	`xxxx10xxxxxx`	Reserved hint, behaves as NOP.	-
`00000`	`xxxx110xxxxx`	Reserved hint, behaves as NOP.	-
`00000`	`xxxx1110xxxx`	Reserved hint, behaves as NOP.	-
`00000`	`xxxx1111xxxx`	DBG	-

Integer Test and Compare (one register and immediate)

This section describes the encoding of the Integer Test and Compare (one register and immediate) instruction class. The encodings in this section are decoded from *Data-processing immediate* on page F4-4193.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31–28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11–0
!=1111 (cond)	0	0	1	1	0	opc		1	Rn	(0)	(0)	(0)	(0)	imm12

Decode fields opc	Instruction page
00	TST (immediate)
01	TEQ (immediate)
10	CMP (immediate)
11	CMN (immediate)

Logical Arithmetic (two register and immediate)

This section describes the encoding of the Logical Arithmetic (two register and immediate) instruction class. The encodings in this section are decoded from *Data-processing immediate* on page F4-4193.

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
!=1111 (cond)	0	0	1	1	1	opc		S	Rn	Rd	imm12

Decode fields opc	Instruction page
00	ORR, ORRS (immediate)
01	MOV, MOVS (immediate)
10	BIC, BICS (immediate)
11	MVN, MVNS (immediate)

F4.1.8 Load/Store Word, Unsigned Byte (immediate, literal)

This section describes the encoding of the Load/Store Word, Unsigned Byte (immediate, literal) instruction class. The encodings in this section are decoded from *A32 instruction set encoding* on page F4-4180.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
!=1111	0	1	0	P	U	o2	W	o1	Rn	Rt	imm12
cond											

Decode fields				Instruction page
P:W	o2	o1	Rn	
!= 01	0	1	1111	LDR (literal)
!= 01	1	1	1111	LDRB (literal)
00	0	0	-	STR (immediate) - Post-indexed variant
00	0	1	!= 1111	LDR (immediate) - Post-indexed variant
00	1	0	-	STRB (immediate) - Post-indexed variant
00	1	1	!= 1111	LDRB (immediate) - Post-indexed variant
01	0	0	-	STRT
01	0	1	-	LDRT
01	1	0	-	STRBT
01	1	1	-	LDRBT
10	0	0	-	STR (immediate) - Offset variant
10	0	1	!= 1111	LDR (immediate) - Offset variant
10	1	0	-	STRB (immediate) - Offset variant
10	1	1	!= 1111	LDRB (immediate) - Offset variant
11	0	0	-	STR (immediate) - Pre-indexed variant
11	0	1	!= 1111	LDR (immediate) - Pre-indexed variant
11	1	0	-	STRB (immediate) - Pre-indexed variant
11	1	1	!= 1111	LDRB (immediate) - Pre-indexed variant

F4.1.9 Load/Store Word, Unsigned Byte (register)

This section describes the encoding of the Load/Store Word, Unsigned Byte (register) instruction class. The encodings in this section are decoded from *A32 instruction set encoding* on page F4-4180.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–7	6–5	4	3–0
!=1111 (cond)	0	1	1	P	U	o2	W	o1	Rn	Rt	imm5	stype	0	Rm

Decode fields				Instruction page
P	o2	W	o1	
0	0	0	0	STR (register) - Post-indexed variant
0	0	0	1	LDR (register) - Post-indexed variant
0	0	1	0	STRT
0	0	1	1	LDRT
0	1	0	0	STRB (register) - Post-indexed variant
0	1	0	1	LDRB (register) - Post-indexed variant
0	1	1	0	STRBT
0	1	1	1	LDRBT
1	0	-	0	STR (register) - Pre-indexed variant
1	0	-	1	LDR (register) - Pre-indexed variant
1	1	-	0	STRB (register) - Pre-indexed variant
1	1	-	1	LDRB (register) - Pre-indexed variant

F4.1.10 Media instructions

This section describes the encoding of the Media instructions group. The encodings in this section are decoded from *A32 instruction set encoding* on page F4-4180.

31–28	27–25	24–20	19–8	7–5	4	3–0
!=1111	011	op0		op1	1	

Table F4-9 Encoding table for the Media instructions group

Decode fields		Decode group or instruction page
op0	op1	
00xxx	-	*Parallel Arithmetic* on page F4-4199
01000	101	SEL
01000	001	Unallocated.
01000	xx0	PKHBT, PKHTB
01001	x01	Unallocated.
01001	xx0	Unallocated.
0110x	x01	Unallocated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table F4-9 Encoding table for the Media instructions group (continued)

Decode fields		Decode group or instruction page
op0	op1	
0110x	xx0	Unallocated.
01x10	001	*Saturate 16-bit* on page F4-4201
01x10	101	Unallocated.
01x11	x01	*Reverse Bit/Byte* on page F4-4202
01x1x	xx0	*Saturate 32-bit* on page F4-4202
01xxx	111	Unallocated.
01xxx	011	*Extend and Add* on page F4-4203
10xxx	-	*Signed multiply, Divide* on page F4-4203
11000	000	*Unsigned Sum of Absolute Differences* on page F4-4204
11000	100	Unallocated.
11001	x00	Unallocated.
1101x	x00	Unallocated.
110xx	111	Unallocated.
1110x	111	Unallocated.
1110x	x00	*Bitfield Insert* on page F4-4205
11110	111	Unallocated.
11111	111	*Permanently UNDEFINED* on page F4-4205
1111x	x00	Unallocated.
11x0x	x10	Unallocated.
11x1x	x10	*Bitfield Extract* on page F4-4206
11xxx	011	Unallocated.
11xxx	x01	Unallocated.

Parallel Arithmetic

This section describes the encoding of the Parallel Arithmetic instruction class. The encodings in this section are decoded from *Media instructions* on page F4-4198.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31–28	27	26	25	24	23	22–20	19–16	15–12	11	10	9	8	7	6–5	4	3–0
!=1111	0	1	1	0	0	op1	Rn	Rd	(1)	(1)	(1)	(1)	B	op2	1	Rm
cond																

Decode fields			Instruction page
op1	B	op2	
000	-	-	Unallocated.
001	0	00	SADD16
001	0	01	SASX
001	0	10	SSAX
001	0	11	SSUB16
001	1	00	SADD8
001	1	01	Unallocated.
001	1	10	Unallocated.
001	1	11	SSUB8
010	0	00	QADD16
010	0	01	QASX
010	0	10	QSAX
010	0	11	QSUB16
010	1	00	QADD8
010	1	01	Unallocated.
010	1	10	Unallocated.
010	1	11	QSUB8
011	0	00	SHADD16
011	0	01	SHASX
011	0	10	SHSAX
011	0	11	SHSUB16
011	1	00	SHADD8
011	1	01	Unallocated.
011	1	10	Unallocated.
011	1	11	SHSUB8
100	-	-	Unallocated.
101	0	00	UADD16
101	0	01	UASX

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields			Instruction page
op1	B	op2	
101	0	10	USAX
101	0	11	USUB16
101	1	00	UADD8
101	1	01	Unallocated.
101	1	10	Unallocated.
101	1	11	USUB8
110	0	00	UQADD16
110	0	01	UQASX
110	0	10	UQSAX
110	0	11	UQSUB16
110	1	00	UQADD8
110	1	01	Unallocated.
110	1	10	Unallocated.
110	1	11	UQSUB8
111	0	00	UHADD16
111	0	01	UHASX
111	0	10	UHSAX
111	0	11	UHSUB16
111	1	00	UHADD8
111	1	01	Unallocated.
111	1	10	Unallocated.
111	1	11	UHSUB8

Saturate 16-bit

This section describes the encoding of the Saturate 16-bit instruction class. The encodings in this section are decoded from *Media instructions* on page F4-4198.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111 (cond)	0	1	1	0	1	U	1	0	sat_imm	Rd	(1)	(1)	(1)	(1)	0	0	1	1	Rn

Decode fields	Instruction page
U	
0	SSAT16
1	USAT16

Reverse Bit/Byte

This section describes the encoding of the Reverse Bit/Byte instruction class. The encodings in this section are decoded from *Media instructions* on page F4-4198.

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111 (cond)	0	1	1	0	1	o1	1	1	(1)	(1)	(1)	(1)	Rd	(1)	(1)	(1)	(1)	o2	0	1	1	Rm

Decode fields		Instruction page
o1	o2	
0	0	REV
0	1	REV16
1	0	RBIT
1	1	REVSH

Saturate 32-bit

This section describes the encoding of the Saturate 32-bit instruction class. The encodings in this section are decoded from *Media instructions* on page F4-4198.

31–28	27	26	25	24	23	22	21	20–16	15–12	11	7	6	5	4	3–0
!=1111 (cond)	0	1	1	0	1	U	1	sat_imm	Rd	imm5	sh	0	1		Rn

Decode fields	Instruction page
U	
0	SSAT
1	USAT

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Extend and Add

This section describes the encoding of the Extend and Add instruction class. The encodings in this section are decoded from *Media instructions* on page F4-4198.

31–28	27 26 25 24	23	22	21 20	19–16	15–12	11 10	9	8	7 6 5 4	3–0
!=1111 (cond)	0 1 1 0	1	U	op	Rn	Rd	rotate	(0)	(0)	0 1 1 1	Rm

Decode fields			Instruction page
U	op	Rn	
0	00	!= 1111	SXTAB16
0	00	1111	SXTB16
0	10	!= 1111	SXTAB
0	10	1111	SXTB
0	11	!= 1111	SXTAH
0	11	1111	SXTH
1	00	!= 1111	UXTAB16
1	00	1111	UXTB16
1	10	!= 1111	UXTAB
1	10	1111	UXTB
1	11	!= 1111	UXTAH
1	11	1111	UXTH

Signed multiply, Divide

This section describes the encoding of the Signed multiply, Divide instruction class. The encodings in this section are decoded from *Media instructions* on page F4-4198.

31–28	27 26 25 24	23	22–20	19–16	15–12	11–8	7–5	4	3–0
!=1111 (cond)	0 1 1 1	0	op1	Rd	Ra	Rm	op2	1	Rn

Decode fields			Instruction page
op1	Ra	op2	
000	!= 1111	000	SMLAD, SMLADX - SMLAD variant
000	!= 1111	001	SMLAD, SMLADX - SMLADX variant
000	!= 1111	010	SMLSD, SMLSDX - SMLSD variant
000	!= 1111	011	SMLSD, SMLSDX - SMLSDX variant
000	-	1xx	Unallocated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields			Instruction page
op1	Ra	op2	
000	1111	000	SMUAD, SMUADX - SMUAD variant
000	1111	001	SMUAD, SMUADX - SMUADX variant
000	1111	010	SMUSD, SMUSDX - SMUSD variant
000	1111	011	SMUSD, SMUSDX - SMUSDX variant
001	-	000	SDIV
001	-	!= 000	Unallocated.
010	-	-	Unallocated.
011	-	000	UDIV
011	-	!= 000	Unallocated.
100	-	000	SMLALD, SMLALDX - SMLALD variant
100	-	001	SMLALD, SMLALDX - SMLALDX variant
100	-	010	SMLSLD, SMLSLDX - SMLSLD variant
100	-	011	SMLSLD, SMLSLDX - SMLSLDX variant
100	-	1xx	Unallocated.
101	!= 1111	000	SMMLA, SMMLAR - SMMLA variant
101	!= 1111	001	SMMLA, SMMLAR - SMMLAR variant
101	-	01x	Unallocated.
101	-	10x	Unallocated.
101	-	110	SMMLS, SMMLSR - SMMLS variant
101	-	111	SMMLS, SMMLSR - SMMLSR variant
101	1111	000	SMMUL, SMMULR - SMMUL variant
101	1111	001	SMMUL, SMMULR - SMMULR variant
11x	-	-	Unallocated.

Unsigned Sum of Absolute Differences

This section describes the encoding of the Unsigned Sum of Absolute Differences instruction class. The encodings in this section are decoded from *Media instructions* on page F4-4198.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–8	7 6 5 4	3–0
!=1111 (cond)	0 1 1 1	1 0 0 0	Rd	Ra	Rm	0 0 0 1	Rn

Decode fields Ra	Instruction page
`!= 1111`	USADA8
`1111`	USAD8

Bitfield Insert

This section describes the encoding of the Bitfield Insert instruction class. The encodings in this section are decoded from *Media instructions* on page F4-4198.

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–7	6 5 4	3–0
!=1111 (cond)	0 1 1 1	1 1 0	msb	Rd	lsb	0 0 1	Rn

Decode fields Rn	Instruction page
`!= 1111`	BFI
`1111`	BFC

Permanently UNDEFINED

This section describes the encoding of the Permanently UNDEFINED instruction class. The encodings in this section are decoded from *Media instructions* on page F4-4198.

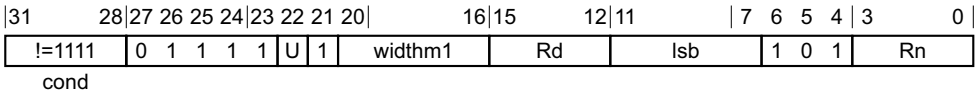
31–28	27 26 25 24	23 22 21 20	19–8	7 6 5 4	3–0
!=1111 (cond)	0 1 1 1	1 1 1 1	imm12	1 1 1 1	imm4

Decode fields cond	Instruction page
`0xxx`	Unallocated.
`10xx`	Unallocated.
`110x`	Unallocated.
`1110`	UDF

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bitfield Extract

This section describes the encoding of the Bitfield Extract instruction class. The encodings in this section are decoded from *Media instructions* on page F4-4198.



Decode fields	Instruction page
U	
0	SBFX
1	UBFX

F4.1.11 Branch, branch with link, and block data transfer

This section describes the encoding of the Branch, branch with link, and block data transfer group. The encodings in this section are decoded from *A32 instruction set encoding* on page F4-4180.

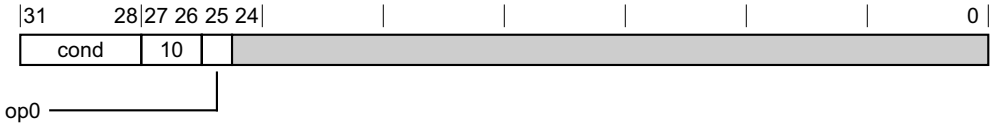


Table F4-10 Encoding table for the Branch, branch with link, and block data transfer group

Decode fields		Decode group or instruction page
cond	op0	
1111	0	*Exception Save/Restore*
!= 1111	0	*Load/Store Multiple* on page F4-4207
-	1	*Branch (immediate)* on page F4-4208

Exception Save/Restore

This section describes the encoding of the Exception Save/Restore instruction class. The encodings in this section are decoded from *Branch, branch with link, and block data transfer*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31 30 29 28	27 26 25	24	23	22	21	20	19 16	15 5	4 0
1 1 1 1	1 0 0	P	U	S	W	L	Rn	op	mode

Decode fields				Instruction page
P	U	S	L	
-	-	0	0	Unallocated.
0	0	0	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB - Decrement After variant
0	0	1	0	SRS, SRSDA, SRSDB, SRSIA, SRSIB - Decrement After variant
0	1	0	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB - Increment After variant
0	1	1	0	SRS, SRSDA, SRSDB, SRSIA, SRSIB - Increment After variant
1	0	0	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB - Decrement Before variant
1	0	1	0	SRS, SRSDA, SRSDB, SRSIA, SRSIB - Decrement Before variant
-	-	1	1	Unallocated.
1	1	0	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB - Increment Before variant
1	1	1	0	SRS, SRSDA, SRSDB, SRSIA, SRSIB - Increment Before variant

Load/Store Multiple

This section describes the encoding of the Load/Store Multiple instruction class. The encodings in this section are decoded from *Branch, branch with link, and block data transfer* on page F4-4206.

31 28	27 26 25	24	23	22	21	20	19 16	15 0
!=1111	1 0 0	P	U	op	W	L	Rn	register_list
cond								

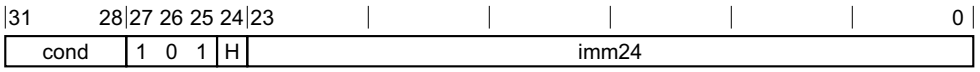
Decode fields					Instruction page
P	U	op	L	register_list	
0	0	0	0	-	STMDA, STMED
0	0	0	1	-	LDMDA, LDMFA
0	1	0	0	-	STM, STMIA, STMEA
0	1	0	1	-	LDM, LDMIA, LDMFD
-	-	1	0	-	STM (User registers)
1	0	0	0	-	STMDB, STMFD
1	0	0	1	-	LDMDB, LDMEA
-	-	1	1	0xxxxxxxxxxxxxxx	LDM (User registers)

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields					Instruction page
P	U	op	L	register_list	
1	1	0	0	-	STMIB, STMFA
1	1	0	1	-	LDMIB, LDMED
-	-	1	1	1xxxxxxxxxxxxxxx	LDM (exception return)

Branch (immediate)

This section describes the encoding of the Branch (immediate) instruction class. The encodings in this section are decoded from *Branch, branch with link, and block data transfer* on page F4-4206.



Decode fields		Instruction page
cond	H	
!= 1111	0	B
!= 1111	1	BL, BLX (immediate) - *A1* on page F5-4309
1111	-	BL, BLX (immediate) - *A2* on page F5-4309

F4.1.12 System register access, Advanced SIMD, floating-point, and Supervisor call

This section describes the encoding of the System register access, Advanced SIMD, floating-point, and Supervisor call group. The encodings in this section are decoded from *A32 instruction set encoding* on page F4-4180.

Table F4-11 Encoding table for the System register access, Advanced SIMD, floating-point, and Supervisor call group

Decode fields				Decode group or instruction page
cond	op0	op1	op2	
-	0x	111	-	*System register load/store and 64-bit move* on page F4-4212
-	10	10x	0	*Floating-point data-processing* on page F4-4214
-	10	111	1	*System register 32-bit move* on page F4-4209
-	11	-	-	*Supervisor call* on page F4-4220
1111	0x	1x0	-	*Advanced SIMD three registers of the same length extension* on page F4-4209

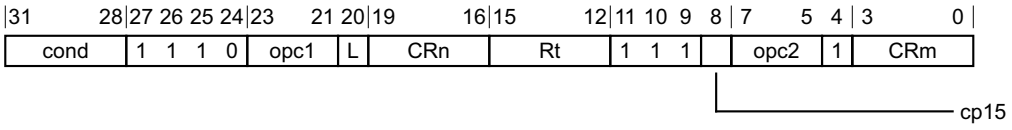
Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table F4-11 Encoding table for the System register access, Advanced SIMD, floating-point, and Supervisor call group (continued)

Decode fields				Decode group or instruction page
cond	**op0**	**op1**	**op2**	
1111	10	1x0	-	*Advanced SIMD two registers and a scalar extension* on page F4-4211
!= 1111	0x	10x	-	*Advanced SIMD load/store and 64-bit move* on page F4-4220
!= 1111	10	10x	1	*Advanced SIMD and floating-point 32-bit move* on page F4-4222

System register 32-bit move

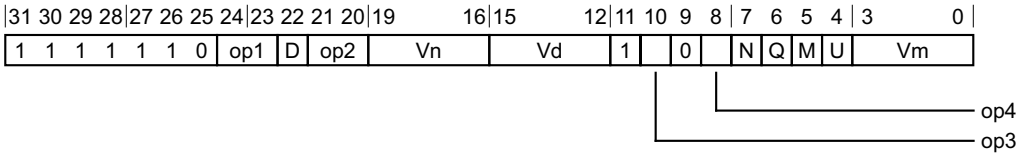
This section describes the encoding of the System register 32-bit move instruction class. The encodings in this section are decoded from *System register access, Advanced SIMD, floating-point, and Supervisor call* on page F4-4208.



Decode fields		Instruction page
cond	**L**	
!= 1111	0	MCR
!= 1111	1	MRC
1111	-	Unallocated.

Advanced SIMD three registers of the same length extension

This section describes the encoding of the Advanced SIMD three registers of the same length extension instruction class. The encodings in this section are decoded from *System register access, Advanced SIMD, floating-point, and Supervisor call* on page F4-4208.



Decode fields						Instruction page	Architecture version
op1	**op2**	**op3**	**op4**	**Q**	**U**		
x1	0x	0	0	0	0	VCADD - 64-bit SIMD vector variant	Armv8.3
x1	0x	0	0	0	1	Unallocated.	-
x1	0x	0	0	1	0	VCADD - 128-bit SIMD vector variant	Armv8.3

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

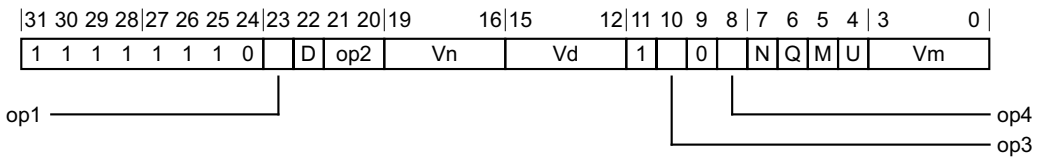
Decode fields						Instruction page	Architecture version
op1	op2	op3	op4	Q	U		
x1	0x	0	0	1	1	Unallocated.	-
00	0x	0	0	-	-	Unallocated.	-
00	0x	0	1	-	-	Unallocated.	-
00	00	1	0	0	0	Unallocated.	-
00	00	1	0	0	1	Unallocated.	-
00	00	1	0	1	0	VMMLA	Armv8.6
00	00	1	0	1	1	Unallocated.	-
00	00	1	1	0	0	VDOT (vector) - 64-bit SIMD vector variant	Armv8.6
00	00	1	1	0	1	Unallocated.	-
00	00	1	1	1	0	VDOT (vector) - 128-bit SIMD vector variant	Armv8.6
00	00	1	1	1	1	Unallocated.	-
00	01	1	0	-	-	Unallocated.	-
00	01	1	1	-	-	Unallocated.	-
00	10	0	0	-	1	VFMAL (vector)	Armv8.2
00	10	0	1	-	-	Unallocated.	-
00	10	1	0	0	-	Unallocated.	-
00	10	1	0	1	0	VSMMLA	Armv8.6
00	10	1	0	1	1	VUMMLA	Armv8.6
00	10	1	1	0	0	VSDOT (vector) - 64-bit SIMD vector variant	Armv8.2
00	10	1	1	0	1	VUDOT (vector) - 64-bit SIMD vector variant	Armv8.2
00	10	1	1	1	0	VSDOT (vector) - 128-bit SIMD vector variant	Armv8.2
00	10	1	1	1	1	VUDOT (vector) - 128-bit SIMD vector variant	Armv8.2
00	11	0	0	-	1	VFMAB, VFMAT (BFloat16, vector)	Armv8.6
00	11	0	1	-	-	Unallocated.	-
00	11	1	0	-	-	Unallocated.	-
00	11	1	1	-	-	Unallocated.	-
01	10	0	0	-	1	VFMSL (vector)	Armv8.2
01	10	0	1	-	-	Unallocated.	-
01	10	1	0	0	-	Unallocated.	-
01	10	1	0	1	0	VUSMMLA	Armv8.6
01	10	1	0	1	1	Unallocated.	-
01	10	1	1	0	0	VUSDOT (vector) - 64-bit SIMD vector variant	Armv8.6

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields						Instruction page	Architecture version
op1	op2	op3	op4	Q	U		
01	10	1	1	-	1	Unallocated.	-
01	10	1	1	1	0	VUSDOT (vector) - 128-bit SIMD vector variant	Armv8.6
01	11	-	-	-	-	Unallocated.	-
-	1x	0	0	-	0	VCMLA	Armv8.3
10	11	-	-	-	-	Unallocated.	-
11	11	-	-	-	-	Unallocated.	-

Advanced SIMD two registers and a scalar extension

This section describes the encoding of the Advanced SIMD two registers and a scalar extension instruction class. The encodings in this section are decoded from *System register access, Advanced SIMD, floating-point, and Supervisor call* on page F4-4208.



Decode fields						Instruction page	Architecture version
op1	op2	op3	op4	Q	U		
0	-	0	0	-	0	VCMLA (by element) - 128-bit SIMD vector of half-precision floating-point variant	Armv8.3
0	00	0	0	-	1	VFMAL (by scalar)	Armv8.2
0	00	0	1	-	-	Unallocated.	-
0	00	1	0	-	-	Unallocated.	-
0	00	1	1	0	0	VDOT (by element) - 64-bit SIMD vector variant	Armv8.6
0	00	1	1	-	1	Unallocated.	-
0	00	1	1	1	0	VDOT (by element) - 128-bit SIMD vector variant	Armv8.6
0	01	0	0	-	0	Unallocated.	-
0	01	0	0	0	1	VFMSL (by scalar) - 64-bit SIMD vector variant	Armv8.2
0	01	0	0	1	1	VFMSL (by scalar) - 128-bit SIMD vector variant	Armv8.2
0	01	0	1	-	-	Unallocated.	-
0	01	1	0	-	-	Unallocated.	-
0	10	0	-	-	-	Unallocated.	-
0	10	1	0	-	-	Unallocated.	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields						Instruction page	Architecture version
op1	op2	op3	op4	Q	U		
0	10	1	1	0	0	VSDOT (by element) - 64-bit SIMD vector variant	Armv8.2
0	10	1	1	0	1	VUDOT (by element) - 64-bit SIMD vector variant	Armv8.2
0	10	1	1	1	0	VSDOT (by element) - 128-bit SIMD vector variant	Armv8.2
0	10	1	1	1	1	VUDOT (by element) - 128-bit SIMD vector variant	Armv8.2
0	11	0	0	-	0	Unallocated.	-
0	11	0	0	-	1	VFMAB, VFMAT (BFloat16, by scalar)	Armv8.6
0	11	0	1	-	-	Unallocated.	-
0	11	1	-	-	-	Unallocated.	-
1	-	0	0	-	0	VCMLA (by element) - 128-bit SIMD vector of single-precision floating-point variant	Armv8.3
1	00	1	1	0	0	VUSDOT (by element) - 64-bit SIMD vector variant	Armv8.6
1	00	1	1	0	1	VSUDOT (by element) - 64-bit SIMD vector variant	Armv8.6
1	00	1	1	1	0	VUSDOT (by element) - 128-bit SIMD vector variant	Armv8.6
1	00	1	1	1	1	VSUDOT (by element) - 128-bit SIMD vector variant	Armv8.6
1	-	0	1	-	-	Unallocated.	-
1	01	1	1	-	-	Unallocated.	-
1	1x	1	1	-	-	Unallocated.	-
1	-	1	0	-	-	Unallocated.	-

F4.1.13 System register load/store and 64-bit move

This section describes the encoding of the System register load/store and 64-bit move group. The encodings in this section are decoded from *System register access, Advanced SIMD, floating-point, and Supervisor call* on page F4-4208.

This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see *About the A32 Advanced SIMD and floating-point instructions and their encoding* on page F4-4240

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31–28	27–25	24–21	20–12	11–9	8–0
	110	op0		111	

Table F4-12 Encoding table for the System register load/store and 64-bit move group

Decode fields	Decode group or instruction page
op0	
00x0	*System register 64-bit move* on page F4-4213
!= 00x0	*System register load/store* on page F4-4213

System register 64-bit move

This section describes the encoding of the System register 64-bit move instruction class. The encodings in this section are decoded from *System register load/store and 64-bit move* on page F4-4212.

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7–4	3–0
cond	1	1	0	0	0	D	0	L	Rt2	Rt	1	1	1		opc1	CRm

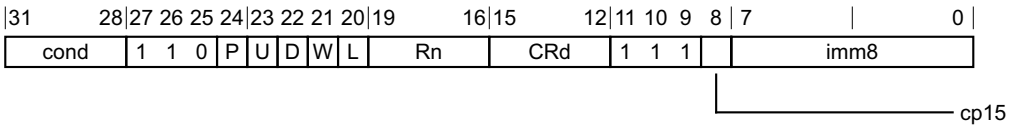
cp15

Decode fields			Instruction page
cond	D	L	
!= 1111	1	0	MCRR
!= 1111	1	1	MRRC
-	0	-	Unallocated.
1111	1	-	Unallocated.

System register load/store

This section describes the encoding of the System register load/store instruction class. The encodings in this section are decoded from *System register load/store and 64-bit move* on page F4-4212.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta



Decode fields							Instruction page
cond	**P:U:W**	**D**	**L**	**Rn**	**CRd**	**cp15**	
!= 1111	!= 000	0	-	-	!= 0101	0	Unallocated.
!= 1111	!= 000	0	1	1111	0101	0	LDC (literal)
!= 1111	!= 000	-	-	-	-	1	Unallocated.
!= 1111	!= 000	1	-	-	0101	0	Unallocated.
!= 1111	0x1	0	0	-	0101	0	STC - Post-indexed variant
!= 1111	0x1	0	1	!= 1111	0101	0	LDC (immediate) - Post-indexed variant
!= 1111	010	0	0	-	0101	0	STC - Unindexed variant
!= 1111	010	0	1	!= 1111	0101	0	LDC (immediate) - Unindexed variant
!= 1111	1x0	0	0	-	0101	0	STC - Offset variant
!= 1111	1x0	0	1	!= 1111	0101	0	LDC (immediate) - Offset variant
!= 1111	1x1	0	0	-	0101	0	STC - Pre-indexed variant
!= 1111	1x1	0	1	!= 1111	0101	0	LDC (immediate) - Pre-indexed variant
1111	!= 000	-	-	-	-	-	Unallocated.

F4.1.14 Floating-point data-processing

This section describes the encoding of the Floating-point data-processing group. The encodings in this section are decoded from *System register access, Advanced SIMD, floating-point, and Supervisor call* on page F4-4208.

This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see *About the A32 Advanced SIMD and floating-point instructions and their encoding* on page F4-4240

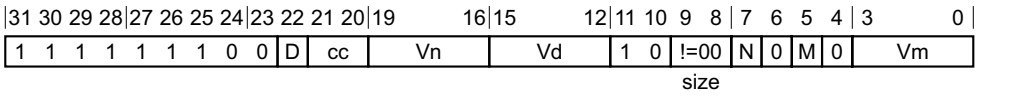
Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table F4-13 Encoding table for the Floating-point data-processing group

Decode fields					Decode group or instruction page
cond	**op0**	**op1**	**op2**	**op3**	
1111	0xxx	-	!= 00	0	*Floating-point conditional select*
1111	1x00	-	!= 00	-	*Floating-point minNum/maxNum* on page F4-4215
1111	1x11	0000	!= 00	1	*Floating-point extraction and insertion* on page F4-4216
1111	1x11	1xxx	!= 00	1	*Floating-point directed convert to integer* on page F4-4216
!= 1111	1x11	-	-	1	*Floating-point data-processing (two registers)* on page F4-4217
!= 1111	1x11	-	-	0	*Floating-point move immediate* on page F4-4218
!= 1111	!= 1x11	-	-	-	*Floating-point data-processing (three registers)* on page F4-4219

Floating-point conditional select

This section describes the encoding of the Floating-point conditional select instruction class. The encodings in this section are decoded from *Floating-point data-processing*.



Decode fields		Instruction page
cc	**size**	
00	-	VSELEQ, VSELGE, VSELGT, VSELVS - VSELEQ,doubleprec variant
01	-	VSELEQ, VSELGE, VSELGT, VSELVS - VSELVS,doubleprec variant
-	01	Unallocated.
10	-	VSELEQ, VSELGE, VSELGT, VSELVS - VSELGE,doubleprec variant
11	-	VSELEQ, VSELGE, VSELGT, VSELVS - VSELGT,doubleprec variant

Floating-point minNum/maxNum

This section describes the encoding of the Floating-point minNum/maxNum instruction class. The encodings in this section are decoded from *Floating-point data-processing* on page F4-4214.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11	10	9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	1	D	0	0	Vn	Vd	1	0	!=00 (size)	N	op	M	0	Vm

Decode fields		Instruction page
size	op	
-	0	VMAXNM
01	-	Unallocated.
-	1	VMINNM

Floating-point extraction and insertion

This section describes the encoding of the Floating-point extraction and insertion instruction class. The encodings in this section are decoded from *Floating-point data-processing* on page F4-4214.

31 30 29 28	27 26 25 24	23	22	21	20	19	18	17	16	15 12	11	10	9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	1	D	1	1	0	0	0	0	Vd	1	0	!=00 (size)	op	1	M	0	Vm

Decode fields		Instruction page	Architecture version
size	op		
01	-	Unallocated.	-
10	0	VMOVX	Armv8.2
10	1	VINS	Armv8.2
11	-	Unallocated.	-

Floating-point directed convert to integer

This section describes the encoding of the Floating-point directed convert to integer instruction class. The encodings in this section are decoded from *Floating-point data-processing* on page F4-4214.

31 30 29 28	27 26 25 24	23	22	21	20	19	18	17	16	15 12	11	10	9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	1	D	1	1	1	o1	RM		Vd	1	0	!=00 (size)	op	1	M	0	Vm

Decode fields			Instruction page
o1	RM	size	
0	00	-	VRINTA (floating-point)
0	01	-	VRINTN (floating-point)
-	-	01	Unallocated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields			Instruction page
o1	RM	size	
0	10	-	VRINTP (floating-point)
0	11	-	VRINTM (floating-point)
1	00	-	VCVTA (floating-point)
1	01	-	VCVTN (floating-point)
1	10	-	VCVTP (floating-point)
1	11	-	VCVTM (floating-point)

Floating-point data-processing (two registers)

This section describes the encoding of the Floating-point data-processing (two registers) instruction class. The encodings in this section are decoded from *Floating-point data-processing* on page F4-4214.

31–28	27	26	25	24	23	22	21	20	19–18	16–15	12–11	10	9–8	7	6	5	4	3–0
!=1111 (cond)	1	1	1	0	1	D	1	1	o1	opc2	Vd	1	0	size	o3	1	M	0	Vm

Decode fields				Instruction page	Architecture version
o1	opc2	size	o3		
-	-	00	-	Unallocated.	-
0	000	01	0	Unallocated.	-
0	000	-	1	VABS	-
0	000	10	0	VMOV (register) - Single-precision scalar variant	-
0	000	11	0	VMOV (register) - Double-precision scalar variant	-
0	001	-	0	VNEG	-
0	001	-	1	VSQRT	-
0	010	-	0	VCVTB - Half-precision to double-precision variant	-
0	010	01	-	Unallocated.	-
0	010	-	1	VCVTT - Half-precision to double-precision variant	-
0	011	01	0	VCVTB (BFloat16)	Armv8.6
0	011	01	1	VCVTT (BFloat16)	Armv8.6
0	011	10	0	VCVTB - Single-precision to half-precision variant	-
0	011	10	1	VCVTT - Single-precision to half-precision variant	-
0	011	11	0	VCVTB - Double-precision to half-precision variant	-
0	011	11	1	VCVTT - Double-precision to half-precision variant	-
0	100	-	0	VCMP - *A1* on page F6-5086	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields				Instruction page	Architecture version
o1	opc2	size	o3		
0	100	-	1	VCMPE - *A1* on page F6-5090	-
0	101	-	0	VCMP - *A2* on page F6-5086	-
0	101	-	1	VCMPE - *A2* on page F6-5090	-
0	110	-	0	VRINTR	-
0	110	-	1	VRINTZ (floating-point)	-
0	111	-	0	VRINTX (floating-point)	-
0	111	01	1	Unallocated.	-
0	111	10	1	VCVT (between double-precision and single-precision) - Single-precision to double-precision variant	-
0	111	11	1	VCVT (between double-precision and single-precision) - Double-precision to single-precision variant	-
1	000	-	-	VCVT (integer to floating-point, floating-point)	-
1	001	01	-	Unallocated.	-
1	001	10	-	Unallocated.	-
1	001	11	0	Unallocated.	-
1	001	11	1	VJCVT	Armv8.3
1	01x	-	-	VCVT (between floating-point and fixed-point, floating-point)	-
1	100	-	0	VCVTR	-
1	100	-	1	VCVT (floating-point to integer, floating-point)	-
1	101	-	0	VCVTR	-
1	101	-	1	VCVT (floating-point to integer, floating-point)	-
1	11x	-	-	VCVT (between floating-point and fixed-point, floating-point)	-

Floating-point move immediate

This section describes the encoding of the Floating-point move immediate instruction class. The encodings in this section are decoded from *Floating-point data-processing* on page F4-4214.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	1	1	1	0	1	D	1	1	imm4H	Vd	1	0	size	(0)	0	(0)	0	imm4L	
cond																			

Decode fields size	Instruction page	Architecture version
00	Unallocated.	-
01	VMOV (immediate) - Half-precision scalar variant	Armv8.2
10	VMOV (immediate) - Single-precision scalar variant	-
11	VMOV (immediate) - Double-precision scalar variant	-

Floating-point data-processing (three registers)

This section describes the encoding of the Floating-point data-processing (three registers) instruction class. The encodings in this section are decoded from *Floating-point data-processing* on page F4-4214.

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	1	1	1	0	o0	D	o1		Vn	Vd	1	0	size		N	o2	M	0	Vm
cond																			

Decode fields o0:o1	size	o2	Instruction page
!= 111	00	-	Unallocated.
000	-	0	VMLA (floating-point)
000	-	1	VMLS (floating-point)
001	-	0	VNMLS
001	-	1	VNMLA
010	-	0	VMUL (floating-point)
010	-	1	VNMUL
011	-	0	VADD (floating-point)
011	-	1	VSUB (floating-point)
100	-	0	VDIV
101	-	0	VFNMS
101	-	1	VFNMA
110	-	0	VFMA
110	-	1	VFMS

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F4.1.15 Supervisor call

This section describes the encoding of the Supervisor call group. The encodings in this section are decoded from *System register access, Advanced SIMD, floating-point, and Supervisor call* on page F4-4208.

31–28	27–24	23–0
cond	1111	

Table F4-14 Encoding table for the Supervisor call group

Decode fields cond	Decode group or instruction page
1111	Unallocated.
!= 1111	SVC

F4.1.16 Advanced SIMD load/store and 64-bit move

This section describes the encoding of the Advanced SIMD load/store and 64-bit move group. The encodings in this section are decoded from *System register access, Advanced SIMD, floating-point, and Supervisor call* on page F4-4208.

This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see *About the A32 Advanced SIMD and floating-point instructions and their encoding* on page F4-4240

31–28	27–25	24–21	20–12	11–10	9–0
!=1111	110	op0		10	

Table F4-15 Encoding table for the Advanced SIMD load/store and 64-bit move group

Decode fields op0	Decode group or instruction page
00x0	*Advanced SIMD and floating-point 64-bit move*
!= 00x0	*Advanced SIMD and floating-point load/store* on page F4-4221

Advanced SIMD and floating-point 64-bit move

This section describes the encoding of the Advanced SIMD and floating-point 64-bit move instruction class. The encodings in this section are decoded from *Advanced SIMD load/store and 64-bit move*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111 (cond)	1	1	0	0	0	D	0	op	Rt2	Rt	1	0	size		opc2		M	o3	Vm

Decode fields					Instruction page
D	op	size	opc2	o3	
0	-	-	-	-	Unallocated.
1	-	-	-	0	Unallocated.
1	-	0x	00	1	Unallocated.
1	-	-	01	-	Unallocated.
1	0	10	00	1	VMOV (between two general-purpose registers and two single-precision registers) - From general-purpose registers variant
1	0	11	00	1	VMOV (between two general-purpose registers and a doubleword floating-point register) - From general-purpose registers variant
1	-	-	1x	-	Unallocated.
1	1	10	00	1	VMOV (between two general-purpose registers and two single-precision registers) - To general-purpose registers variant
1	1	11	00	1	VMOV (between two general-purpose registers and a doubleword floating-point register) - To general-purpose registers variant

Advanced SIMD and floating-point load/store

This section describes the encoding of the Advanced SIMD and floating-point load/store instruction class. The encodings in this section are decoded from *Advanced SIMD load/store and 64-bit move* on page F4-4220.

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7–0
!=1111 (cond)	1	1	0	P	U	D	W	L	Rn	Vd	1	0	size		imm8

Decode fields							Instruction page
P	U	W	L	Rn	size	imm8	
0	0	1	-	-	-	-	Unallocated.
0	1	-	-	-	0x	-	Unallocated.
0	1	-	0	-	10	-	VSTM, VSTMDB, VSTMIA - *Increment After variant* on page F6-5618
0	1	-	0	-	11	xxxxxxx0	VSTM, VSTMDB, VSTMIA - *Increment After variant* on page F6-5617
0	1	-	0	-	11	xxxxxxx1	FSTMDBX, FSTMIAX - *Increment After variant* on page F6-4968
0	1	-	1	-	10	-	VLDM, VLDMDB, VLDMIA - *Increment After variant* on page F6-5260
0	1	-	1	-	11	xxxxxxx0	VLDM, VLDMDB, VLDMIA - *Increment After variant* on page F6-5259
0	1	-	1	-	11	xxxxxxx1	FLDM*X (FLDMDBX, FLDMIAX) - *Increment After variant* on page F6-4965

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields							Instruction page
P	U	W	L	Rn	size	imm8	
1	-	0	0	-	-	-	VSTR
1	-	0	-	-	00	-	Unallocated.
1	-	0	1	!= 1111	-	-	VLDR (immediate)
1	0	1	-	-	0x	-	Unallocated.
1	0	1	0	-	10	-	VSTM, VSTMDB, VSTMIA - *Decrement Before variant* on page F6-5618
1	0	1	0	-	11	xxxxxxx0	VSTM, VSTMDB, VSTMIA - *Decrement Before variant* on page F6-5617
1	0	1	0	-	11	xxxxxxx1	FSTMDBX, FSTMIAX - *Decrement Before variant* on page F6-4968
1	0	1	1	-	10	-	VLDM, VLDMDB, VLDMIA - *Decrement Before variant* on page F6-5260
1	0	1	1	-	11	xxxxxxx0	VLDM, VLDMDB, VLDMIA - *Decrement Before variant* on page F6-5259
1	0	1	1	-	11	xxxxxxx1	FLDM*X (FLDMDBX, FLDMIAX) - *Decrement Before variant* on page F6-4965
1	-	0	1	1111	-	-	VLDR (literal)
1	1	1	-	-	-	-	Unallocated.

F4.1.17 Advanced SIMD and floating-point 32-bit move

This section describes the encoding of the Advanced SIMD and floating-point 32-bit move group. The encodings in this section are decoded from *System register access, Advanced SIMD, floating-point, and Supervisor call* on page F4-4208.

This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see *About the A32 Advanced SIMD and floating-point instructions and their encoding* on page F4-4240

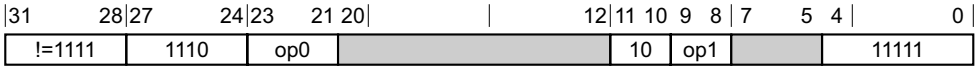


Table F4-16 Encoding table for the Advanced SIMD and floating-point 32-bit move group

Decode fields		Decode group or instruction page	Architecture version
op0	op1		
000	01	VMOV (between general-purpose register and half-precision)	Armv8.2
000	10	VMOV (between general-purpose register and single-precision)	-
111	10	*Floating-point move special register*	-
-	11	*Advanced SIMD 8/16/32-bit element move/duplicate*	-

Floating-point move special register

This section describes the encoding of the Floating-point move special register instruction class. The encodings in this section are decoded from *Advanced SIMD and floating-point 32-bit move* on page F4-4222.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3	2	1	0
!=1111 (cond)	1	1	1	0	1	1	1	L	reg	Rt	1	0	1	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)

Decode fields	Instruction page
L	
0	VMSR
1	VMRS

Advanced SIMD 8/16/32-bit element move/duplicate

This section describes the encoding of the Advanced SIMD 8/16/32-bit element move/duplicate instruction class. The encodings in this section are decoded from *Advanced SIMD and floating-point 32-bit move* on page F4-4222.

31–28	27	26	25	24	23–21	20	19–16	15–12	11	10	9	8	7	6–5	4	3	2	1	0
!=1111 (cond)	1	1	1	0	opc1	L	Vn	Rt	1	0	1	1	N	opc2	1	(0)	(0)	(0)	(0)

Decode fields			Instruction page
opc1	L	opc2	
0xx	0	-	VMOV (general-purpose register to scalar)
-	1	-	VMOV (scalar to general-purpose register)
1xx	0	0x	VDUP (general-purpose register)
1xx	0	1x	Unallocated.

F4.1.18 Unconditional instructions

This section describes the encoding of the Unconditional instructions group. The encodings in this section are decoded from *A32 instruction set encoding* on page F4-4180.

31–27	26	25	24–21	20	19–0
11110	op0			op1	

Table F4-17 Encoding table for the Unconditional instructions group

Decode fields		Decode group or instruction page
op0	op1	
00	-	*Miscellaneous*
01	-	*Advanced SIMD data-processing* on page F4-4225

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table F4-17 Encoding table for the Unconditional instructions group (continued)

Decode fields		Decode group or instruction page
op0	op1	
1x	1	*Memory hints and barriers* on page F4-4234
10	0	*Advanced SIMD element or structure load/store* on page F4-4237
11	0	Unallocated.

F4.1.19 Miscellaneous

This section describes the encoding of the Miscellaneous group. The encodings in this section are decoded from *Unconditional instructions*.

31 – 25	24 – 20	19 – 8	7 – 4	3 – 0
1111000	op0		op1	

Table F4-18 Encoding table for the Miscellaneous group

Decode fields		Decode group or instruction page	Architecture version
op0	op1		
0xxxx	-	Unallocated.	-
10000	xx0x	*Change Process State* on page F4-4225	-
10001	1000	Unallocated.	-
10001	x100	Unallocated.	-
10001	xx01	Unallocated.	-
10001	0000	SETPAN	Armv8.1
1000x	0111	Unallocated.	-
10010	0111	CONSTRAINED UNPREDICTABLE	-
10011	0111	Unallocated.	-
1001x	xx0x	Unallocated.	-
100xx	0011	Unallocated.	-
100xx	0x10	Unallocated.	-
100xx	1x1x	Unallocated.	-
101xx	-	Unallocated.	-
11xxx	-	Unallocated.	-

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in *CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings* on page K1-7842.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Change Process State

This section describes the encoding of the Change Process State instruction class. The encodings in this section are decoded from *Miscellaneous* on page F4-4224.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	0
1	1	1	1	0	0	0	1	0	0	0	0	imod		M	op	(0)	(0)	(0)	(0)	(0)	(0)	E	A	I	F	0	mode	

Decode fields				Instruction page
imod	M	op	mode	
-	-	1	0xxxx	SETEND
-	-	0	-	CPS, CPSID, CPSIE
-	-	1	1xxxx	Unallocated.

F4.1.20 Advanced SIMD data-processing

This section describes the encoding of the Advanced SIMD data-processing group. The encodings in this section are decoded from *Unconditional instructions* on page F4-4223.

This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see *About the A32 Advanced SIMD and floating-point instructions and their encoding* on page F4-4240

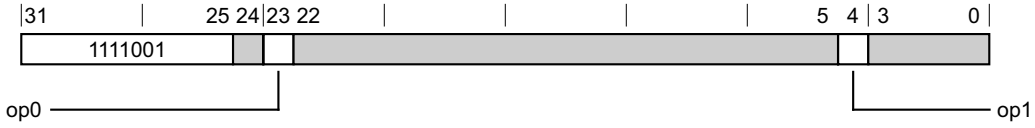


Table F4-19 Encoding table for the Advanced SIMD data-processing group

Decode fields		Decode group or instruction page
op0	op1	
0	-	*Advanced SIMD three registers of the same length*
1	0	*Advanced SIMD two registers, or three registers of different lengths* on page F4-4228
1	1	*Advanced SIMD shifts and immediate generation* on page F4-4233

Advanced SIMD three registers of the same length

This section describes the encoding of the Advanced SIMD three registers of the same length instruction class. The encodings in this section are decoded from *Advanced SIMD data-processing* on page F4-4225.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31 30 29 28	27 26 25 24	23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
1 1 1 1	0 0 1 U	0 D size	Vn	Vd	opc	N Q M o1	Vm

Decode fields					Instruction page	Architecture version
U	size	opc	Q	o1		
0	0x	1100	-	1	VFMA	-
0	0x	1101	-	0	VADD (floating-point)	-
0	0x	1101	-	1	VMLA (floating-point)	-
0	0x	1110	-	0	VCEQ (register) - *A2* on page F6-5047	-
0	0x	1111	-	0	VMAX (floating-point)	-
0	0x	1111	-	1	VRECPS	-
-	-	0000	-	0	VHADD	-
0	00	0001	-	1	VAND (register)	-
-	-	0000	-	1	VQADD	-
-	-	0001	-	0	VRHADD	-
0	00	1100	-	0	SHA1C	-
-	-	0010	-	0	VHSUB	-
0	01	0001	-	1	VBIC (register)	-
-	-	0010	-	1	VQSUB	-
-	-	0011	-	0	VCGT (register) - *A1* on page F6-5060	-
-	-	0011	-	1	VCGE (register) - *A1* on page F6-5053	-
0	01	1100	-	0	SHA1P	-
0	1x	1100	-	1	VFMS	-
0	1x	1101	-	0	VSUB (floating-point)	-
0	1x	1101	-	1	VMLS (floating-point)	-
0	1x	1110	-	0	Unallocated.	-
0	1x	1111	-	0	VMIN (floating-point)	-
0	1x	1111	-	1	VRSQRTS	-
-	-	0100	-	0	VSHL (register)	-
0	-	1000	-	0	VADD (integer)	-
0	10	0001	-	1	VORR (register)	-
0	-	1000	-	1	VTST	-
-	-	0100	-	1	VQSHL (register)	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields					Instruction page	Architecture version
U	size	opc	Q	o1		
0	-	1001	-	0	VMLA (integer)	-
-	-	0101	-	0	VRSHL	-
-	-	0101	-	1	VQRSHL	-
0	-	1011	-	0	VQDMULH	-
0	10	1100	-	0	SHA1M	-
0	-	1011	-	1	VPADD (integer)	-
-	-	0110	-	0	VMAX (integer)	-
0	11	0001	-	1	VORN (register)	-
-	-	0110	-	1	VMIN (integer)	-
-	-	0111	-	0	VABD (integer)	-
-	-	0111	-	1	VABA	-
0	11	1100	-	0	SHA1SU0	-
1	0x	1101	-	0	VPADD (floating-point)	-
1	0x	1101	-	1	VMUL (floating-point)	-
1	0x	1110	-	0	VCGE (register) - *A2* on page F6-5053	-
1	0x	1110	-	1	VACGE	-
1	0x	1111	0	0	VPMAX (floating-point)	-
1	0x	1111	-	1	VMAXNM	-
1	00	0001	-	1	VEOR	-
-	-	1001	-	1	VMUL (integer and polynomial)	-
1	00	1100	-	0	SHA256H	-
-	-	1010	0	0	VPMAX (integer)	-
1	01	0001	-	1	VBSL	-
-	-	1010	0	1	VPMIN (integer)	-
-	-	1010	1	-	Unallocated.	-
1	01	1100	-	0	SHA256H2	-
1	1x	1101	-	0	VABD (floating-point)	-
1	1x	1110	-	0	VCGT (register) - *A2* on page F6-5060	-
1	1x	1110	-	1	VACGT	-
1	1x	1111	0	0	VPMIN (floating-point)	-
1	1x	1111	-	1	VMINNM	-
1	-	1000	-	0	VSUB (integer)	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields					Instruction page	Architecture version
U	size	opc	Q	o1		
1	10	0001	-	1	VBIT	-
1	-	1000	-	1	VCEQ (register) - *A1* on page F6-5047	-
1	-	1001	-	0	VMLS (integer)	-
1	-	1011	-	0	VQRDMULH	-
1	10	1100	-	0	SHA256SU1	-
1	-	1011	-	1	VQRDMLAH	Armv8.1
1	11	0001	-	1	VBIF	-
1	-	1100	-	1	VQRDMLSH	Armv8.1
1	-	1111	1	0	Unallocated.	-

F4.1.21 Advanced SIMD two registers, or three registers of different lengths

This section describes the encoding of the Advanced SIMD two registers, or three registers of different lengths group. The encodings in this section are decoded from *Advanced SIMD data-processing* on page F4-4225.

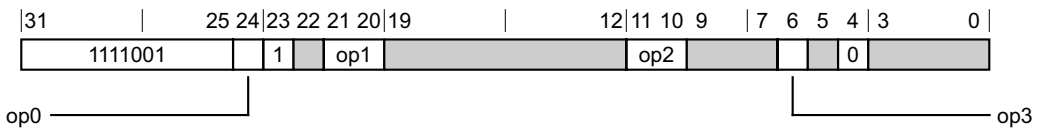


Table F4-20 Encoding table for the Advanced SIMD two registers, or three registers of different lengths group

Decode fields				Decode group or instruction page
op0	op1	op2	op3	
0	11	-	-	VEXT (byte elements)
1	11	0x	-	*Advanced SIMD two registers misc*
1	11	10	-	VTBL, VTBX
1	11	11	-	*Advanced SIMD duplicate (scalar)* on page F4-4230
-	!= 11	-	0	*Advanced SIMD three registers of different lengths* on page F4-4231
-	!= 11	-	1	*Advanced SIMD two registers and a scalar* on page F4-4232

Advanced SIMD two registers misc

This section describes the encoding of the Advanced SIMD two registers misc instruction class. The encodings in this section are decoded from *Advanced SIMD two registers, or three registers of different lengths* on page F4-4228.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31 30 29 28	27 26 25 24	23 22	21 20	19 18 17 16	15 12	11 10	7	6	5	4	3 0
1 1 1 1	0 0 1 1 1	D	1 1	size	opc1	Vd	0	opc2	Q	M	0	Vm

Decode fields				Instruction page	Architecture version
size	opc1	opc2	Q		
-	00	0000	-	VREV64	-
-	00	0001	-	VREV32	-
-	00	0010	-	VREV16	-
-	00	0011	-	Unallocated.	-
-	00	010x	-	VPADDL	-
-	00	0110	0	AESE	-
-	00	0110	1	AESD	-
-	00	0111	0	AESMC	-
-	00	0111	1	AESIMC	-
-	00	1000	-	VCLS	-
00	10	0000	-	VSWP	-
-	00	1001	-	VCLZ	-
-	00	1010	-	VCNT	-
-	00	1011	-	VMVN (register)	-
00	10	1100	1	Unallocated.	-
-	00	110x	-	VPADAL	-
-	00	1110	-	VQABS	-
-	00	1111	-	VQNEG	-
-	01	x000	-	VCGT (immediate #0)	-
-	01	x001	-	VCGE (immediate #0)	-
-	01	x010	-	VCEQ (immediate #0)	-
-	01	x011	-	VCLE (immediate #0)	-
-	01	x100	-	VCLT (immediate #0)	-
-	01	x110	-	VABS	-
-	01	x111	-	VNEG	-
-	01	0101	1	SHA1H	-
01	10	1100	1	VCVT (from single-precision to BFloat16, Advanced SIMD)	Armv8.6
-	10	0001	-	VTRN	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields				Instruction page	Architecture version
size	opc1	opc2	Q		
-	10	0010	-	VUZP	-
-	10	0011	-	VZIP	-
-	10	0100	0	VMOVN	-
-	10	0100	1	VQMOVN, VQMOVUN - Unsigned result variant	-
-	10	0101	-	VQMOVN, VQMOVUN - Signed result variant	-
-	10	0110	0	VSHLL	-
-	10	0111	0	SHA1SU1	-
-	10	0111	1	SHA256SU0	-
-	10	1000	-	VRINTN (Advanced SIMD)	-
-	10	1001	-	VRINTX (Advanced SIMD)	-
-	10	1010	-	VRINTA (Advanced SIMD)	-
-	10	1011	-	VRINTZ (Advanced SIMD)	-
10	10	1100	1	Unallocated.	-
-	10	1100	0	VCVT (between half-precision and single-precision, Advanced SIMD) - Single-precision to half-precision variant	-
-	10	1101	-	VRINTM (Advanced SIMD)	-
-	10	1110	0	VCVT (between half-precision and single-precision, Advanced SIMD) - Half-precision to single-precision variant	-
-	10	1110	1	Unallocated.	-
-	10	1111	-	VRINTP (Advanced SIMD)	-
-	11	000x	-	VCVTA (Advanced SIMD)	-
-	11	001x	-	VCVTN (Advanced SIMD)	-
-	11	010x	-	VCVTP (Advanced SIMD)	-
-	11	011x	-	VCVTM (Advanced SIMD)	-
-	11	10x0	-	VRECPE	-
-	11	10x1	-	VRSQRTE	-
11	10	1100	1	Unallocated.	-
-	11	11xx	-	VCVT (between floating-point and integer, Advanced SIMD)	-

Advanced SIMD duplicate (scalar)

This section describes the encoding of the Advanced SIMD duplicate (scalar) instruction class. The encodings in this section are decoded from *Advanced SIMD two registers, or three registers of different lengths* on page F4-4228.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9–7	6	5	4	3–0
1	1	1	1	0	0	1	1	1	D	1	1	imm4	Vd	1	1	opc	Q	M	0	Vm

Decode fields	Instruction page
opc	
000	VDUP (scalar)
001	Unallocated.
01x	Unallocated.
1xx	Unallocated.

Advanced SIMD three registers of different lengths

This section describes the encoding of the Advanced SIMD three registers of different lengths instruction class. The encodings in this section are decoded from *Advanced SIMD two registers, or three registers of different lengths* on page F4-4228.

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
1	1	1	1	0	0	1	U	1	D	!=11		Vn	Vd	opc	N	0	M	0	Vm
										size									

Decode fields		Instruction page
U	opc	
-	0000	VADDL
-	0001	VADDW
-	0010	VSUBL
0	0100	VADDHN
-	0011	VSUBW
0	0110	VSUBHN
0	1001	VQDMLAL
-	0101	VABAL
0	1011	VQDMLSL
0	1101	VQDMULL
-	0111	VABDL (integer)
-	1000	VMLAL (integer)
-	1010	VMLSL (integer)
1	0100	VRADDHN
1	0110	VRSUBHN

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields		Instruction page
U	opc	
-	11x0	VMULL (integer and polynomial)
1	1001	Unallocated.
1	1011	Unallocated.
1	1101	Unallocated.
-	1111	Unallocated.

Advanced SIMD two registers and a scalar

This section describes the encoding of the Advanced SIMD two registers and a scalar instruction class. The encodings in this section are decoded from *Advanced SIMD two registers, or three registers of different lengths* on page F4-4228.

31 30 29 28	27 26 25 24	23	22	21 20	19 16	15 12	11 8	7	6	5	4	3 0
1 1 1 1	0 0 0 1	Q	1	D	!=11	Vn	Vd	opc	N	1	M	0	Vm
				size								

Decode fields		Instruction page	Architecture version
Q	opc		
-	000x	VMLA (by scalar)	-
0	0011	VQDMLAL	-
-	0010	VMLAL (by scalar)	-
0	0111	VQDMLSL	-
-	010x	VMLS (by scalar)	-
0	1011	VQDMULL	-
-	0110	VMLSL (by scalar)	-
-	100x	VMUL (by scalar)	-
1	0011	Unallocated.	-
-	1010	VMULL (by scalar)	-
1	0111	Unallocated.	-
-	1100	VQDMULH	-
-	1101	VQRDMULH	-
1	1011	Unallocated.	-
-	1110	VQRDMLAH	Armv8.1
-	1111	VQRDMLSH	Armv8.1

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F4.1.22 Advanced SIMD shifts and immediate generation

This section describes the encoding of the Advanced SIMD shifts and immediate generation group. The encodings in this section are decoded from *Advanced SIMD data-processing* on page F4-4225.

31 – 25	24	23	22	21 – 7	6	5	4	3 – 0
1111001		1		op0			1	

Table F4-21 Encoding table for the Advanced SIMD shifts and immediate generation group

Decode fields op0	Decode group or instruction page
`000xxxxxxxxxxx0`	*Advanced SIMD one register and modified immediate* on page F4-4233
`!= 000xxxxxxxxxxx0`	*Advanced SIMD two registers and shift amount* on page F4-4234

Advanced SIMD one register and modified immediate

This section describes the encoding of the Advanced SIMD one register and modified immediate instruction class. The encodings in this section are decoded from *Advanced SIMD shifts and immediate generation* on page F4-4233.

31	30	29	28	27	26	25	24	23	22	21	20	19 – 18 – 16	15 – 12	11 – 8	7	6	5	4	3 – 0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3	Vd	cmode	0	Q	op	1	imm4

Decode fields cmode	op	Instruction page
`0xx0`	`0`	VMOV (immediate) - *A1* on page F6-5320
`0xx0`	`1`	VMVN (immediate) - *A1* on page F6-5366
`0xx1`	`0`	VORR (immediate) - *A1* on page F6-5390
`0xx1`	`1`	VBIC (immediate) - *A1* on page F6-5032
`10x0`	`0`	VMOV (immediate) - *A3* on page F6-5321
`10x0`	`1`	VMVN (immediate) - *A2* on page F6-5366
`10x1`	`0`	VORR (immediate) - *A2* on page F6-5390
`10x1`	`1`	VBIC (immediate) - *A2* on page F6-5032
`11xx`	`0`	VMOV (immediate) - *A4* on page F6-5321
`110x`	`1`	VMVN (immediate) - *A3* on page F6-5367
`1110`	`1`	VMOV (immediate) - *A5* on page F6-5322
`1111`	`1`	Unallocated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Advanced SIMD two registers and shift amount

This section describes the encoding of the Advanced SIMD two registers and shift amount instruction class. The encodings in this section are decoded from *Advanced SIMD shifts and immediate generation* on page F4-4233.

31 30 29 28	27 26 25 24	23	22	21	19 18 16	15 12	11 8	7	6	5	4	3 0
1 1 1 1	0 0 1 U	1	D	imm3H	imm3L	Vd	opc	L	Q	M	1	Vm

Decode fields					Instruction page
U	imm3H:L	imm3L	opc	Q	
-	!= 0000	-	0000	-	VSHR
-	!= 0000	-	0001	-	VSRA
-	!= 0000	000	1010	0	VMOVL
-	!= 0000	-	0010	-	VRSHR
-	!= 0000	-	0011	-	VRSRA
-	!= 0000	-	0111	-	VQSHL, VQSHLU (immediate) - VQSHL,quad,signed-result variant
-	!= 0000	-	1001	0	VQSHRN, VQSHRUN - Signed result variant
-	!= 0000	-	1001	1	VQRSHRN, VQRSHRUN - Signed result variant
-	!= 0000	-	1010	0	VSHLL
-	!= 0000	-	11xx	-	VCVT (between floating-point and fixed-point, Advanced SIMD)
0	!= 0000	-	0101	-	VSHL (immediate)
0	!= 0000	-	1000	0	VSHRN
0	!= 0000	-	1000	1	VRSHRN
1	!= 0000	-	0100	-	VSRI
1	!= 0000	-	0101	-	VSLI
1	!= 0000	-	0110	-	VQSHL, VQSHLU (immediate) - VQSHLU,quad,unsigned-result variant
1	!= 0000	-	1000	0	VQSHRN, VQSHRUN - Unsigned result variant
1	!= 0000	-	1000	1	VQRSHRN, VQRSHRUN - Unsigned result variant

F4.1.23 Memory hints and barriers

This section describes the encoding of the Memory hints and barriers group. The encodings in this section are decoded from *Unconditional instructions* on page F4-4223.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

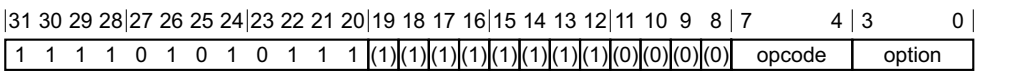
Table F4-22 Encoding table for the Memory hints and barriers group

Decode fields		Decode group or instruction page
op0	**op1**	
00xx1	-	CONSTRAINED UNPREDICTABLE
01001	-	CONSTRAINED UNPREDICTABLE
01011	-	*Barriers* on page F4-4235
011x1	-	CONSTRAINED UNPREDICTABLE
0xxx0	-	*Preload (immediate)* on page F4-4236
1xxx0	0	*Preload (register)* on page F4-4236
1xxx1	0	CONSTRAINED UNPREDICTABLE
1xxxx	1	Unallocated.

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in *CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings* on page K1-7842.

Barriers

This section describes the encoding of the Barriers instruction class. The encodings in this section are decoded from *Memory hints and barriers* on page F4-4234.



Decode fields		Instruction page
opcode	**option**	
0000	-	CONSTRAINED UNPREDICTABLE
0001	-	CLREX
001x	-	CONSTRAINED UNPREDICTABLE
0100	!= 0x00	DSB
0100	0000	SSBB
0100	0100	PSSBB
0101	-	DMB

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields		Instruction page
opcode	option	
0110	-	ISB
0111	-	SB
1xxx	-	CONSTRAINED UNPREDICTABLE

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in *CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings* on page K1-7842.

Preload (immediate)

This section describes the encoding of the Preload (immediate) instruction class. The encodings in this section are decoded from *Memory hints and barriers* on page F4-4234.

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15	14	13	12	11 0
1 1 1 1	0 1 0	D	U	R	0	1	Rn	(1)	(1)	(1)	(1)	imm12

Decode fields			Instruction page
D	R	Rn	
0	0	-	Reserved hint, behaves as NOP.
0	1	-	PLI (immediate, literal)
1	-	1111	PLD (literal)
1	0	!= 1111	PLD, PLDW (immediate) - Preload write variant
1	1	!= 1111	PLD, PLDW (immediate) - Preload read variant

Preload (register)

This section describes the encoding of the Preload (register) instruction class. The encodings in this section are decoded from *Memory hints and barriers* on page F4-4234.

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15	14	13	12	11 7	6 5	4	3 0
1 1 1 1	0 1 1	D	U	o2	0	1	Rn	(1)	(1)	(1)	(1)	imm5	stype	0	Rm

Decode fields		Instruction page
D	o2	
0	0	Reserved hint, behaves as NOP.
0	1	PLI (register)
1	0	PLD, PLDW (register) - Preload write, rotate right with extend variant
1	1	PLD, PLDW (register) - Preload read, rotate right with extend variant

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F4.1.24 Advanced SIMD element or structure load/store

This section describes the encoding of the Advanced SIMD element or structure load/store group. The encodings in this section are decoded from *Unconditional instructions* on page F4-4223.

This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see *About the A32 Advanced SIMD and floating-point instructions and their encoding* on page F4-4240

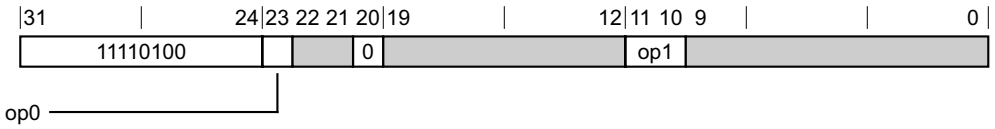
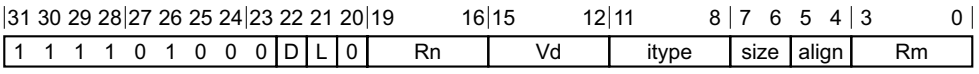


Table F4-23 Encoding table for the Advanced SIMD element or structure load/store group

Decode fields		Decode group or instruction page
op0	**op1**	
0	-	*Advanced SIMD load/store multiple structures* on page F4-4237
1	11	*Advanced SIMD load single structure to all lanes* on page F4-4238
1	!= 11	*Advanced SIMD load/store single structure to one lane* on page F4-4238

Advanced SIMD load/store multiple structures

This section describes the encoding of the Advanced SIMD load/store multiple structures instruction class. The encodings in this section are decoded from *Advanced SIMD element or structure load/store* on page F4-4237.



Decode fields		Instruction page
L	**itype**	
0	000x	VST4 (multiple 4-element structures)
0	0010	VST1 (multiple single elements) - *A4* on page F6-5582
0	0011	VST2 (multiple 2-element structures) - *A2* on page F6-5595
0	010x	VST3 (multiple 3-element structures)
0	0110	VST1 (multiple single elements) - *A3* on page F6-5581
0	0111	VST1 (multiple single elements) - *A1* on page F6-5580
0	100x	VST2 (multiple 2-element structures) - *A1* on page F6-5594
0	1010	VST1 (multiple single elements) - *A2* on page F6-5580
1	000x	VLD4 (multiple 4-element structures)
1	0010	VLD1 (multiple single elements) - *A4* on page F6-5215

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode fields		Instruction page
L	itype	
1	0011	VLD2 (multiple 2-element structures) - *A2* on page F6-5231
1	010x	VLD3 (multiple 3-element structures)
-	1011	Unallocated.
1	0110	VLD1 (multiple single elements) - *A3* on page F6-5214
1	0111	VLD1 (multiple single elements) - *A1* on page F6-5213
-	11xx	Unallocated.
1	100x	VLD2 (multiple 2-element structures) - *A1* on page F6-5230
1	1010	VLD1 (multiple single elements) - *A2* on page F6-5213

Advanced SIMD load single structure to all lanes

This section describes the encoding of the Advanced SIMD load single structure to all lanes instruction class. The encodings in this section are decoded from *Advanced SIMD element or structure load/store* on page F4-4237.

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10	9 8	7 6	5	4	3 0
1 1 1 1	0 1 0 0 1	D	L		0	Rn	Vd	1 1	N	size	T	a	Rm

Decode fields			Instruction page
L	N	a	
0	-	-	Unallocated.
1	00	-	VLD1 (single element to all lanes)
1	01	-	VLD2 (single 2-element structure to all lanes)
1	10	0	VLD3 (single 3-element structure to all lanes)
1	10	1	Unallocated.
1	11	-	VLD4 (single 4-element structure to all lanes)

Advanced SIMD load/store single structure to one lane

This section describes the encoding of the Advanced SIMD load/store single structure to one lane instruction class. The encodings in this section are decoded from *Advanced SIMD element or structure load/store* on page F4-4237.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

31 30 29 28	27 26 25 24	23 22 21 20	19 16	15 12	11 10	9 8	7 4	3 0
1 1 1 1	0 1 0 0 1	D L 0	Rn	Vd	!=11	N	index_align	Rm
					size			

Decode fields

L	size	N	Rm	Instruction page
0	00	00	-	VST1 (single element from one lane) - *A1* on page F6-5575
0	00	01	-	VST2 (single 2-element structure from one lane) - *A1* on page F6-5588
0	00	10	-	VST3 (single 3-element structure from one lane) - *A1* on page F6-5599
0	00	11	-	VST4 (single 4-element structure from one lane)
0	01	00	-	VST1 (single element from one lane) - *A2* on page F6-5575
0	01	01	-	VST2 (single 2-element structure from one lane) - *A2* on page F6-5588
0	01	10	-	VST3 (single 3-element structure from one lane) - *A2* on page F6-5599
0	01	11	-	VST4 (single 4-element structure from one lane)
0	10	00	-	VST1 (single element from one lane) - *A3* on page F6-5576
0	10	01	-	VST2 (single 2-element structure from one lane) - *A3* on page F6-5589
0	10	10	-	VST3 (single 3-element structure from one lane) - *A3* on page F6-5600
0	10	11	!= 11x1	VST4 (single 4-element structure from one lane) - *A3* on page F6-5609
0	10	11	1101	VST4 (single 4-element structure from one lane) - *A3* on page F6-5609
0	10	11	1111	VST4 (single 4-element structure from one lane) - *A3* on page F6-5609
1	00	00	-	VLD1 (single element to one lane) - *A1* on page F6-5205
1	00	01	-	VLD2 (single 2-element structure to one lane) - *A1* on page F6-5221
1	00	10	-	VLD3 (single 3-element structure to one lane) - *A1* on page F6-5235
1	00	11	-	VLD4 (single 4-element structure to one lane) - *A1* on page F6-5247
1	01	00	-	VLD1 (single element to one lane) - *A2* on page F6-5205
1	01	01	-	VLD2 (single 2-element structure to one lane) - *A2* on page F6-5221
1	01	10	-	VLD3 (single 3-element structure to one lane) - *A2* on page F6-5235
1	01	11	-	VLD4 (single 4-element structure to one lane) - *A2* on page F6-5247
1	10	00	-	VLD1 (single element to one lane) - *A3* on page F6-5206
1	10	01	-	VLD2 (single 2-element structure to one lane) - *A3* on page F6-5222
1	10	10	-	VLD3 (single 3-element structure to one lane) - *A3* on page F6-5236
1	10	11	-	VLD4 (single 4-element structure to one lane) - *A3* on page F6-5248

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F4.2 About the A32 Advanced SIMD and floating-point instructions and their encoding

The Advanced SIMD and floating-point instructions are common to the T32 and A32 instruction sets. These instructions perform Advanced SIMD and floating-point operations on a common register file, the SIMD&FP register file. This means:

- In general, the instructions that load or store registers in this file, or move data between general-purpose registers and this register file, are common to the Advanced SIMD and floating-point instructions.
- There are distinct Advanced SIMD data-processing instructions and floating-point data-processing instructions.

All A32 Advanced SIMD and floating-point instructions have 32-bit encodings. Different groups of these instructions are decoded from different points in the 32-bit A32 instruction decode structure. Table F4-24 shows these instruction groups, and where each group is decoded from the overall A32 decode structure:

Table F4-24 Advanced SIMD and floating-point instructions in the A32 decode structure

Advanced SIMD and floating-point instruction group	A32 decode is from
Advanced SIMD load/store and 64-bit move on page F4-4220	*System register access, Advanced SIMD, floating-point, and Supervisor call* on page F4-4208
Floating-point data-processing on page F4-4214	*System register access, Advanced SIMD, floating-point, and Supervisor call* on page F4-4208
Advanced SIMD and floating-point 32-bit move on page F4-4222	*System register access, Advanced SIMD, floating-point, and Supervisor call* on page F4-4208
Advanced SIMD data-processing on page F4-4225	*Unconditional instructions* on page F4-4223
Advanced SIMD element or structure load/store on page F4-4237	*Unconditional instructions* on page F4-4223

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter F5
T32 and A32 Base Instruction Set Instruction Descriptions

This chapter describes each instruction. It contains the following sections:

- *Alphabetical list of T32 and A32 base instruction set instructions* on page F5-4242.
- *Encoding and use of banked register transfer instructions* on page F5-4951.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1 Alphabetical list of T32 and A32 base instruction set instructions

This section lists every instruction in the T32 and A32 base instruction sets. For details of the format used see *Format of instruction descriptions* on page F2-4076.

This section is formatted so that each instruction description starts on a new page.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.1 ADC, ADCS (immediate)

Add with Carry (immediate) adds an immediate value and the Carry flag value to a register value, and writes the result to the destination register.

If the destination register is not the PC, the ADCS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The ADC variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- The ADCS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–0
!=1111 (cond)	0 0 1 0	1 0 1 S	Rn	Rd	imm12

ADC variant

Applies when `S == 0`.

```
ADC{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

ADCS variant

Applies when `S == 1`.

```
ADCS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = A32ExpandImm(imm12);
```

T1

15 14 13 12	11	10	9	8	7 6 5 4	3–0	15	14–12	11–8	7–0
1 1 1 1 0	i	0	1	0	1 0 S	Rn	0	imm3	Rd	imm8

ADC variant

Applies when `S == 0`.

```
ADC{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

ADCS variant

Applies when `S == 1`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
ADCS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = T32ExpandImm(i:imm3:imm8);
if d == 15 || n == 15 then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the ADC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- For the ADCS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`.

`<Rn>` For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

`<const>` For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* on page F2-4096 for the range of values.

For encoding T1: an immediate value. See *Modified immediate constants in T32 instructions* on page F2-4095 for the range of values.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, nzcv) = AddWithCarry(R[n], imm32, PSTATE.C);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.2 ADC, ADCS (register)

Add with Carry (register) adds a register value, the Carry flag value, and an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the ADCS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The ADC variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- The ADCS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–7	6 5	4	3–0
!=1111	0 0 0 0	1 0 1 S	Rn	Rd	imm5	stype	0	Rm
cond								

ADC, rotate right with extend variant

Applies when `S == 0 && imm5 == 00000 && stype == 11`.

```
ADC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

ADC, shift or rotate by value variant

Applies when `S == 0 && !(imm5 == 00000 && stype == 11)`.

```
ADC{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

ADCS, rotate right with extend variant

Applies when `S == 1 && imm5 == 00000 && stype == 11`.

```
ADCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

ADCS, shift or rotate by value variant

Applies when `S == 1 && !(imm5 == 00000 && stype == 11)`.

```
ADCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15 14 13 12	11 10 9 8	7 6 5	3 2 0
0 1 0 0 0 0	0 1 0 1	Rm	Rdn

T1 variant

```
ADC<c>{<q>} {<Rdn>,} <Rdn>, <Rm> // Inside IT block
ADCS{<q>} {<Rdn>,} <Rdn>, <Rm> // Outside IT block
```

Decode for this encoding

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 14 12	11 8	7 6 5 4	3 0
1 1 1 0 1 0 1	1 0 1 0	S	Rn	(0) imm3	Rd	imm2 stype	Rm

ADC, rotate right with extend variant

Applies when `S == 0 && imm3 == 000 && imm2 == 00 && stype == 11`.

```
ADC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

ADC, shift or rotate by value variant

Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11)`.

```
ADC<c>.W {<Rd>,} <Rn>, <Rm> // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
ADC{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

ADCS, rotate right with extend variant

Applies when `S == 1 && imm3 == 000 && imm2 == 00 && stype == 11`.

```
ADCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

ADCS, shift or rotate by value variant

Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11)`.

```
ADCS.W {<Rd>,} <Rn>, <Rm> // Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
ADCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rdn>` Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the ADC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- For the ADCS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`.

`<Rn>` For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T2: is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

`LSL`	when `stype = 00`
`LSR`	when `stype = 01`
`ASR`	when `stype = 10`
`ROR`	when `stype = 11`

`<amount>` For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

In T32 assembly:

- Outside an IT block, if `ADCS <Rd>, <Rn>, <Rd>` has `<Rd>` and `<Rn>` both in the range R0-R7, it is assembled using encoding T1 as though `ADCS <Rd>, <Rn>` had been written.
- Inside an IT block, if `ADC<c> <Rd>, <Rn>, <Rd>` has `<Rd>` and `<Rn>` both in the range R0-R7, it is assembled using encoding T1 as though `ADC<c> <Rd>, <Rn>` had been written.

To prevent either of these happening, use the .W qualifier.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], shifted, PSTATE.C);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
else
    R[d] = result;
    if setflags then
        PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.3 ADC, ADCS (register-shifted register)

Add with Carry (register-shifted register) adds a register value, the Carry flag value, and a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–8	7	6 5	4	3–0
!=1111 (cond)	0 0 0 0	1 0 1 S	Rn	Rd	Rs	0	stype	1	Rm

Flag setting variant

Applies when `S == 1`.

```
ADCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

Not flag setting variant

Applies when `S == 0`.

```
ADC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

- `LSL` when `stype = 00`
- `LSR` when `stype = 01`
- `ASR` when `stype = 10`
- `ROR` when `stype = 11`

`<Rs>` Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], shifted, PSTATE.C);
    R[d] = result;
    if setflags then
        PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.4 ADD, ADDS (immediate)

Add (immediate) adds an immediate value to a register value, and writes the result to the destination register.

If the destination register is not the PC, the ADDS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The ADD variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- The ADDS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–0
!=1111 (cond)	0 0 1 0	1 0 0 S	Rn	Rd	imm12

ADD variant

Applies when `S == 0 && Rn != 11x1`.

```
ADD{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

ADDS variant

Applies when `S == 1 && Rn != 1101`.

```
ADDS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

Decode for all variants of this encoding

```
if Rn == '1111' && S == '0' then SEE "ADR";
if Rn == '1101' then SEE "ADD (SP plus immediate)";
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = A32ExpandImm(imm12);
```

T1

15 14 13 12	11 10	9	8–6	5–3	2–0
0 0 0 1	1 1	0	imm3	Rn	Rd

T1 variant

```
ADD<c>{<q>} <Rd>, <Rn>, #<imm3> // Inside IT block
ADDS{<q>} <Rd>, <Rn>, #<imm3> // Outside IT block
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm3, 32);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T2

15 14 13	12 11	10 8	7 0
0 0 1	1 0	Rdn	imm8

T2 variant

```
ADD<c>{<q>} <Rdn>, #<imm8> // Inside IT block, and <Rdn>, <imm8> can be represented in T1
ADD<c>{<q>} {<Rdn>,} <Rdn>, #<imm8> // Inside IT block, and <Rdn>, <imm8> cannot be represented in T1
ADDS{<q>} <Rdn>, #<imm8> // Outside IT block, and <Rdn>, <imm8> can be represented in T1
ADDS{<q>} {<Rdn>,} <Rdn>, #<imm8> // Outside IT block, and <Rdn>, <imm8> cannot be represented in T1
```

Decode for this encoding

```
d = UInt(Rdn);  n = UInt(Rdn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);
```

T3

15 14 13 12	11	10	9	8	7 6 5 4	3 0	15	14 12	11 8	7 0
1 1 1 1 0	i	0	1	0	0 0 S	!=1101	0	imm3	Rd	imm8
						Rn				

ADD variant

Applies when `S == 0`.

```
ADD<c>.W {<Rd>,} <Rn>, #<const> // Inside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or
T2
ADD{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

ADDS variant

Applies when `S == 1 && Rd != 1111`.

```
ADDS.W {<Rd>,} <Rn>, #<const> // Outside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or T2
ADDS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then SEE "CMN (immediate)";
if Rn == '1101' then SEE "ADD (SP plus immediate)";
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = T32ExpandImm(i:imm3:imm8);
if (d == 15 && !setflags) || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

T4

15 14 13 12	11	10 9	8	7 6 5 4	3 0	15	14 12	11 8	7 0
1 1 1 1 0	i	1 0	0	0 0 0	!=11x1	0	imm3	Rd	imm8
					Rn				

T4 variant

```
ADD{<c>}{<q>} {<Rd>,} <Rn>, #<imm12> // <imm12> cannot be represented in T1, T2, or T3
ADDW{<c>}{<q>} {<Rd>,} <Rn>, #<imm12> // <imm12> can be represented in T1, T2, or T3
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
if Rn == '1111' then SEE "ADR";
if Rn == '1101' then SEE "ADD (SP plus immediate)";
d = UInt(Rd);  n = UInt(Rn);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE;   // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rdn>` Is the general-purpose source and destination register, encoded in the "Rdn" field.

`<imm8>` Is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. If the PC is used:

- For the ADD variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- For the ADDS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>. Arm deprecates use of this instruction.

For encoding T1, T3 and T4: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`.

`<Rn>` For encoding A1 and T4: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see ADD, ADDS (SP plus immediate). If the PC is used, see ADR.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

For encoding T3: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see ADD, ADDS (SP plus immediate).

`<imm3>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "imm3" field.

`<imm12>` Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.

`<const>` For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* on page F2-4096 for the range of values.

For encoding T3: an immediate value. See *Modified immediate constants in T32 instructions* on page F2-4095 for the range of values.

When multiple encodings of the same length are available for an instruction, encoding T3 is preferred to encoding T4 (if encoding T4 is required, use the `ADDW` syntax). Encoding T1 is preferred to encoding T2 if `<Rd>` is specified and encoding T2 is preferred to encoding T1 if `<Rd>` is omitted.

Operation for all encodings

```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        (result, nzcv) = AddWithCarry(R[n], imm32, '0');
        if d == 15 then          // Can only occur for A32 encoding
            if setflags then
                ALUExceptionReturn(result);
            else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
                ALUWritePC(result);
        else
            R[d] = result;
            if setflags then
                PSTATE.<N,Z,C,V> = nzcv;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        (result, nzcv) = AddWithCarry(R[n], imm32, '0');
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.5 ADD, ADDS (register)

Add (register) adds a register value and an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the ADDS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The ADD variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- The ADDS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–7	6 5	4	3–0
!=1111	0 0 0 0	1 0 0 S	!=1101	Rd	imm5	stype	0	Rm
cond			Rn					

ADD, rotate right with extend variant

Applies when `S == 0 && imm5 == 00000 && stype == 11`.

```
ADD{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

ADD, shift or rotate by value variant

Applies when `S == 0 && !(imm5 == 00000 && stype == 11)`.

```
ADD{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

ADDS, rotate right with extend variant

Applies when `S == 1 && imm5 == 00000 && stype == 11`.

```
ADDS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

ADDS, shift or rotate by value variant

Applies when `S == 1 && !(imm5 == 00000 && stype == 11)`.

```
ADDS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if Rn == '1101' then SEE "ADD (SP plus register)";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15 14 13 12	11 10	9	8	6 5	3 2 0
0 0 0 1	1 0	0	Rm	Rn	Rd

T1 variant

```
ADD<c>{<q>} <Rd>, <Rn>, <Rm> // Inside IT block
ADDS{<q>} {<Rd>,} <Rn>, <Rm> // Outside IT block
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2

15 14 13 12	11 10	9	8	7	6	3 2 0
0 1 0 0	0 1	0	0		!=1101	Rdn
				DN	Rm	

T2 variant

Applies when `!(DN == 1 && Rdn == 101)`.

```
ADD<c>{<q>} <Rdn>, <Rm> // Preferred syntax, Inside IT block
ADD{<c>}{<q>} {<Rdn>,} <Rdn>, <Rm>
```

Decode for this encoding

```
if (DN:Rdn) == '1101' || Rm == '1101' then SEE "ADD (SP plus register)";
d = UInt(DN:Rdn);  n = d;  m = UInt(Rm);  setflags = FALSE;  (shift_t, shift_n) = (SRType_LSL, 0);
if n == 15 && m == 15 then UNPREDICTABLE;
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

T3

15 14 13 12	11 10	9	8	7 6 5 4	3 0	15 14 12	11 8	7 6 5 4	3 0
1 1 1 0	1 0	1	1	0 0 0 S	!=1101	(0) imm3	Rd	imm2 stype	Rm
					Rn				

ADD, rotate right with extend variant

Applies when `S == 0 && imm3 == 000 && imm2 == 00 && stype == 11`.

```
ADD{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

ADD, shift or rotate by value variant

Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11)`.

```
ADD<c>.W {<Rd>,} <Rn>, <Rm> // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
ADD{<c>}.W {<Rd>,} <Rn>, <Rm> // <Rd> == <Rn>, and <Rd>, <Rn>, <Rm> can be represented in T2
ADD{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ADDS, rotate right with extend variant

Applies when `S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11`.

```
ADDS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

ADDS, shift or rotate by value variant

Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11) && Rd != 1111`.

```
ADDS.W {<Rd>,} <Rn>, <Rm> // Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1 or T2
ADDS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then SEE "CMN (register)";
if Rn == '1101' then SEE "ADD (SP plus register)";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if (d == 15 && !setflags) || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rdn>` Is the general-purpose source and destination register, encoded in the "DN:Rdn" field. If the PC is used, the instruction is a branch to the address calculated by the operation. This is a simple branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.

The assembler language allows `<Rdn>` to be specified once or twice in the assembler syntax. When used inside an IT block, and `<Rdn>` and `<Rm>` are in the range R0 to R7, `<Rdn>` must be specified once so that encoding T2 is preferred to encoding T1. In all other cases there is no difference in behavior when `<Rdn>` is specified once or twice.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. If the PC is used:

- For the ADD variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- For the ADDS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>. Arm deprecates use of this instruction.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field.

When used inside an IT block, `<Rd>` must be specified. When used outside an IT block, `<Rd>` is optional, and:

- If omitted, this register is the same as `<Rn>`.
- If present, encoding T1 is preferred to encoding T2.

For encoding T3: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`.

`<Rn>` For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used. If the SP is used, see ADD, ADDS (SP plus register).

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For encoding T1: is the first general-purpose source register, encoded in the "Rn" field.

For encoding T3: is the first general-purpose source register, encoded in the "Rn" field. If the SP is used, see ADD, ADDS (SP plus register).

`<Rm>` For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1 and T3: is the second general-purpose source register, encoded in the "Rm" field.

For encoding T2: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

`LSL` when `stype = 00`

`LSR` when `stype = 01`

`ASR` when `stype = 10`

`ROR` when `stype = 11`

`<amount>` For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T3: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Inside an IT block, if `ADD<c> <Rd>, <Rn>, <Rd>` cannot be assembled using encoding T1, it is assembled using encoding T2 as though `ADD<c> <Rd>, <Rn>` had been written. To prevent this happening, use the `.W` qualifier.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], shifted, '0');
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.6 ADD, ADDS (register-shifted register)

Add (register-shifted register) adds a register value and a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–8	7	6 5	4	3–0
!=1111 (cond)	0 0 0 0	1 0 0 S	Rn	Rd	Rs	0	stype	1	Rm

Flag setting variant

Applies when `S == 1`.

```
ADDS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

Not flag setting variant

Applies when `S == 0`.

```
ADD{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

- `LSL` when `stype = 00`
- `LSR` when `stype = 01`
- `ASR` when `stype = 10`
- `ROR` when `stype = 11`

`<Rs>` Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], shifted, '0');
    R[d] = result;
    if setflags then
        PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.7 ADD, ADDS (SP plus immediate)

Add to SP (immediate) adds an immediate value to the SP value, and writes the result to the destination register.

If the destination register is not the PC, the ADDS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. However, when the destination register is the PC:

- The ADD variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- The ADDS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

31–28	27 26 25 24	23 22 21 20	19 18 17 16	15–12	11–0
!=1111 (cond)	0 0 1 0	1 0 0 S	1 1 0 1	Rd	imm12

ADD variant

Applies when `S == 0`.

```
ADD{<c>}{<q>} {<Rd>,} SP, #<const>
```

ADDS variant

Applies when `S == 1`.

```
ADDS{<c>}{<q>} {<Rd>,} SP, #<const>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  setflags = (S == '1');  imm32 = A32ExpandImm(imm12);
```

T1

15 14 13 12	11	10–8	7–0
1 0 1 0	1	Rd	imm8

T1 variant

```
ADD{<c>}{<q>} <Rd>, SP, #<imm8>
```

Decode for this encoding

```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(imm8:'00', 32);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T2

15	14	13	12	11	10	9	8	7	6		0
1	0	1	1	0	0	0	0	0	imm7		

T2 variant

```
ADD{<c>}{<q>} {SP,} SP, #<imm7>
```

Decode for this encoding

```
d = 13;  setflags = FALSE;  imm32 = ZeroExtend(imm7:'00', 32);
```

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	12	11	8	7	0
1	1	1	1	0	i	0	1	0	0	0	S	1	1	0	1	0	imm3		Rd		imm8	

ADD variant

Applies when `S == 0`.

```
ADD{<c>}.W {<Rd>,} SP, #<const> // <Rd>, <const> can be represented in T1 or T2
ADD{<c>}{<q>} {<Rd>,} SP, #<const>
```

ADDS variant

Applies when `S == 1 && Rd != 1111`.

```
ADDS{<c>}{<q>} {<Rd>,} SP, #<const>
```

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then SEE "CMN (immediate)";
d = UInt(Rd);  setflags = (S == '1');  imm32 = T32ExpandImm(i:imm3:imm8);
if d == 15 && !setflags then UNPREDICTABLE;
```

T4

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	12	11	8	7	0
1	1	1	1	0	i	1	0	0	0	0	0	1	1	0	1	0	imm3		Rd		imm8	

T4 variant

```
ADD{<c>}{<q>} {<Rd>,} SP, #<imm12> // <imm12> cannot be represented in T1, T2, or T3
ADDW{<c>}{<q>} {<Rd>,} SP, #<imm12> // <imm12> can be represented in T1, T2, or T3
```

Decode for this encoding

```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`SP,` Is the stack pointer.

`<imm7>` Is the unsigned immediate, a multiple of 4, in the range 0 to 508, encoded in the "imm7" field as <imm7>/4.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the ADD variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- For the ADDS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field.

For encoding T3 and T4: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP.

`<imm8>` Is an unsigned immediate, a multiple of 4, in the range 0 to 1020, encoded in the "imm8" field as <imm8>/4.

`<imm12>` Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.

`<const>` For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* on page F2-4096 for the range of values.

For encoding T3: an immediate value. See *Modified immediate constants in T32 instructions* on page F2-4095 for the range of values.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, nzcv) = AddWithCarry(SP, imm32, '0');
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.8 ADD, ADDS (SP plus register)

Add to SP (register) adds an optionally-shifted register value to the SP value, and writes the result to the destination register.

If the destination register is not the PC, the ADDS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The ADD variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- The ADDS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

31–28	27 26 25 24	23 22 21 20	19 18 17 16	15–12	11–7	6 5	4	3–0
!=1111 (cond)	0 0 0 0	1 0 0 S	1 1 0 1	Rd	imm5	stype	0	Rm

ADD, rotate right with extend variant

Applies when `S == 0 && imm5 == 00000 && stype == 11`.

```
ADD{<c>}{<q>} {<Rd>,} SP, <Rm> , RRX
```

ADD, shift or rotate by value variant

Applies when `S == 0 && !(imm5 == 00000 && stype == 11)`.

```
ADD{<c>}{<q>} {<Rd>,} SP, <Rm> {, <shift> #<amount>}
```

ADDS, rotate right with extend variant

Applies when `S == 1 && imm5 == 00000 && stype == 11`.

```
ADDS{<c>}{<q>} {<Rd>,} SP, <Rm> , RRX
```

ADDS, shift or rotate by value variant

Applies when `S == 1 && !(imm5 == 00000 && stype == 11)`.

```
ADDS{<c>}{<q>} {<Rd>,} SP, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	0
0	1	0	0	0	1	0	0		1	1	0	1	Rdm	

DM

T1 variant

```
ADD{<c>}{<q>} {<Rdm>,} SP, <Rdm>
```

Decode for this encoding

```
d = UInt(DM:Rdm);  m = UInt(DM:Rdm);  setflags = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

T2

15	14	13	12	11	10	9	8	7	6	3	2	1	0
0	1	0	0	0	1	0	0	1	!=1101		1	0	1

Rm

T2 variant

```
ADD{<c>}{<q>} {SP,} SP, <Rm>
```

Decode for this encoding

```
if Rm == '1101' then SEE "encoding T1";
d = 13;  m = UInt(Rm);  setflags = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	12	11	8	7	6	5	4	3	0
1	1	1	0	1	0	1	1	0	0	0	S	1	1	0	1	(0)	imm3		Rd		imm2		stype		Rm	

ADD, rotate right with extend variant

Applies when `S == 0 && imm3 == 000 && imm2 == 00 && stype == 11`.

```
ADD{<c>}{<q>} {<Rd>,} SP, <Rm>, RRX
```

ADD, shift or rotate by value variant

Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11)`.

```
ADD{<c>}.W {<Rd>,} SP, <Rm> // <Rd>, <Rm> can be represented in T1 or T2
ADD{<c>}{<q>} {<Rd>,} SP, <Rm> {, <shift> #<amount>}
```

ADDS, rotate right with extend variant

Applies when `S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11`.

```
ADDS{<c>}{<q>} {<Rd>,} SP, <Rm>, RRX
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ADDS, shift or rotate by value variant

Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11) && Rd != 1111`.

```
ADDS{<c>}{<q>} {<Rd>,} SP, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then SEE "CMN (register)";
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if (d == 15 && !setflags) || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`SP,` Is the stack pointer.

`<Rdm>` Is the general-purpose destination and second source register, encoded in the "Rdm" field. If omitted, this register is the SP. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is a simple branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the ADD variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- For the ADDS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T3: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP.

`<Rm>` For encoding A1 and T2: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T3: is the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

`LSL`	when `stype = 00`
`LSR`	when `stype = 01`
`ASR`	when `stype = 10`
`ROR`	when `stype = 11`

`<amount>` For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T3: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(SP, shifted, '0');
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.9 ADD (immediate, to PC)

Add to PC adds an immediate value to the Align(PC, 4) value to form a PC-relative address, and writes the result to the destination register. Arm recommends that, where possible, software avoids using this alias

This instruction is a pseudo-instruction of the ADR instruction. This means that:

- The encodings in this description are named to match the encodings of ADR.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of ADR gives the operational pseudocode for this instruction.

A1

31–28	27 26 25 24	23 22 21 20	19 18 17 16	15–12	11–0
!=1111 (cond)	0 0 1 0	1 0 0 0	1 1 1 1	Rd	imm12

A1 variant

```
ADD{<c>}{<q>} <Rd>, PC, #<const>
```

is equivalent to

```
ADR{<c>}{<q>} <Rd>, <label>
```

and is never the preferred disassembly.

T1

15 14 13 12	11	10–8	7–0
1 0 1 0	0	Rd	imm8

T1 variant

```
ADD{<c>}{<q>} <Rd>, PC, #<imm8>
```

is equivalent to

```
ADR{<c>}{<q>} <Rd>, <label>
```

and is never the preferred disassembly.

T3

15 14 13 12	11	10	9	8	7	6	5	4	3	2	1	0	15	14–12	11–8	7–0
1 1 1 1	0	i	1	0	0	0	0	0	1	1	1	1	0	imm3	Rd	imm8

T3 variant

```
ADDW{<c>}{<q>} <Rd>, PC, #<imm12> // <Rd>, <imm12> can be represented in T1
```

is equivalent to

```
ADR{<c>}{<q>} <Rd>, <label>
```

and is never the preferred disassembly.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
ADD{<c>}{<q>} <Rd>, PC, #<imm12>
```

is equivalent to

```
ADR{<c>}{<q>} <Rd>, <label>
```

and is never the preferred disassembly.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.

For encoding T1 and T3: is the general-purpose destination register, encoded in the "Rd" field.

`<label>` For encoding A1: the label of an instruction or literal data item whose address is to be loaded into `<Rd>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the `ADR` instruction to this label.

If the offset is zero or positive, encoding A1 is used, with `imm32` equal to the offset.

If the offset is negative, encoding A2 is used, with `imm32` equal to the size of the offset. That is, the use of encoding A2 indicates that the required offset is minus the value of `imm32`.

Permitted values of the size of the offset are any of the constants described in *Modified immediate constants in A32 instructions* on page F2-4096.

For encoding T1: the label of an instruction or literal data item whose address is to be loaded into `<Rd>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the `ADR` instruction to this label. Permitted values of the size of the offset are multiples of 4 in the range 0 to 1020.

For encoding T3: the label of an instruction or literal data item whose address is to be loaded into `<Rd>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the `ADR` instruction to this label.

If the offset is zero or positive, encoding T3 is used, with `imm32` equal to the offset.

If the offset is negative, encoding T2 is used, with `imm32` equal to the size of the offset. That is, the use of encoding T2 indicates that the required offset is minus the value of `imm32`.

Permitted values of the size of the offset are 0-4095.

`<imm8>` Is an unsigned immediate, a multiple of 4, in the range 0 to 1020, encoded in the "imm8" field as <imm8>/4.

`<imm12>` Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.

`<const>` An immediate value. See *Modified immediate constants in A32 instructions* on page F2-4096 for the range of values.

Operation for all encodings

The description of ADR gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.10 ADR

Form PC-relative address adds an immediate value to the PC value to form a PC-relative address, and writes the result to the destination register.

This instruction is used by the pseudo-instructions ADD (immediate, to PC) and SUB (immediate, from PC). The pseudo-instruction is never the preferred disassembly.

A1

31–28	27 26 25 24	23 22 21 20	19 18 17 16	15–12	11–0
!=1111 (cond)	0 0 1 0	1 0 0 0	1 1 1 1	Rd	imm12

A1 variant

```
ADR{<c>}{<q>} <Rd>, <label>
```

Decode for this encoding

```
d = UInt(Rd);  imm32 = A32ExpandImm(imm12);  add = TRUE;
```

A2

31–28	27 26 25 24	23 22 21 20	19 18 17 16	15–12	11–0
!=1111 (cond)	0 0 1 0	0 1 0 0	1 1 1 1	Rd	imm12

A2 variant

```
ADR{<c>}{<q>} <Rd>, <label>
```

Decode for this encoding

```
d = UInt(Rd);  imm32 = A32ExpandImm(imm12);  add = FALSE;
```

T1

15 14 13 12	11	10–8	7–0
1 0 1 0	0	Rd	imm8

T1 variant

```
ADR{<c>}{<q>} <Rd>, <label>
```

Decode for this encoding

```
d = UInt(Rd);  imm32 = ZeroExtend(imm8:'00', 32);  add = TRUE;
```

T2

15 14 13 12	11	10	9	8	7	6	5	4	3	2	1	0	15	14–12	11–8	7–0
1 1 1 1 0	i	1	0	1	0	1	0	1	1	1	1	0	imm3	Rd	imm8	

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T2 variant

```
ADR{<c>}{<q>} <Rd>, <label>
```

Decode for this encoding

```
d = UInt(Rd);  imm32 = ZeroExtend(i:imm3:imm8, 32);  add = FALSE;
if d == 15 then UNPREDICTABLE;     // Armv8-A removes UNPREDICTABLE for R13
```

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14 12	11 8	7 0
1	1	1	1	0	i	1	0	0	0	0	0	1	1	1	1	0	imm3	Rd	imm8

T3 variant

```
ADR{<c>}.W <Rd>, <label> // <Rd>, <label> can be presented in T1
ADR{<c>}{<q>} <Rd>, <label>
```

Decode for this encoding

```
d = UInt(Rd);  imm32 = ZeroExtend(i:imm3:imm8, 32);  add = TRUE;
if d == 15 then UNPREDICTABLE;   // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Alias conditions

Alias or pseudo-instruction	of variant	is preferred when
ADD (immediate, to PC)	-	Never
SUB (immediate, from PC)	T2	`i:imm3:imm8 == '000000000000'`
SUB (immediate, from PC)	A2	`imm12 == '000000000000'`

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` For encoding A1 and A2: is the general-purpose destination register, encoded in the "Rd" field. If the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.

For encoding T1, T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.

`<label>` For encoding A1 and A2: the label of an instruction or literal data item whose address is to be loaded into `<Rd>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the `ADR` instruction to this label.

If the offset is zero or positive, encoding A1 is used, with `imm32` equal to the offset.

If the offset is negative, encoding A2 is used, with `imm32` equal to the size of the offset. That is, the use of encoding A2 indicates that the required offset is minus the value of `imm32`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Permitted values of the size of the offset are any of the constants described in *Modified immediate constants in A32 instructions* on page F2-4096.

For encoding T1: the label of an instruction or literal data item whose address is to be loaded into `<Rd>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the `ADR` instruction to this label. Permitted values of the size of the offset are multiples of 4 in the range 0 to 1020.

For encoding T2 and T3: the label of an instruction or literal data item whose address is to be loaded into `<Rd>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the `ADR` instruction to this label.

If the offset is zero or positive, encoding T3 is used, with `imm32` equal to the offset.

If the offset is negative, encoding T2 is used, with `imm32` equal to the size of the offset. That is, the use of encoding T2 indicates that the required offset is minus the value of `imm32`.

Permitted values of the size of the offset are 0-4095.

The instruction aliases permit the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page F1-4037.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
    if d == 15 then          // Can only occur for A32 encodings
        ALUWritePC(result);
    else
        R[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.11 AND, ANDS (immediate)

Bitwise AND (immediate) performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register.

If the destination register is not the PC, the ANDS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The AND variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- The ANDS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–0
!=1111 (cond)	0 0 1 0	0 0 0 S	Rn	Rd	imm12

AND variant

Applies when `S == 0`.

```
AND{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

ANDS variant

Applies when `S == 1`.

```
ANDS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = A32ExpandImm_C(imm12, PSTATE.C);
```

T1

15 14 13 12	11	10	9	8	7 6 5 4	3–0	15	14–12	11–8	7–0
1 1 1 1 0	i	0	0	0	0 0 0 S	Rn	0	imm3	Rd	imm8

AND variant

Applies when `S == 0`.

```
AND{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

ANDS variant

Applies when `S == 1 && Rd != 1111`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
ANDS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then SEE "TST (immediate)";
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
if (d == 15 && !setflags) || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the AND variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- For the ANDS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`.

`<Rn>` For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

`<const>` For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* on page F2-4096 for the range of values.

For encoding T1: an immediate value. See *Modified immediate constants in T32 instructions* on page F2-4095 for the range of values.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] AND imm32;
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.12 AND, ANDS (register)

Bitwise AND (register) performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the ANDS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The AND variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- The ANDS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

31–28	27 26 25 24	23 22 21 20	19	16–19	15–12	11–7	6 5	4	3–0
!=1111 (cond)	0 0 0 0	0 0 0	S	Rn	Rd	imm5	stype	0	Rm

AND, rotate right with extend variant

Applies when `S == 0 && imm5 == 00000 && stype == 11`.

```
AND{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

AND, shift or rotate by value variant

Applies when `S == 0 && !(imm5 == 00000 && stype == 11)`.

```
AND{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

ANDS, rotate right with extend variant

Applies when `S == 1 && imm5 == 00000 && stype == 11`.

```
ANDS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

ANDS, shift or rotate by value variant

Applies when `S == 1 && !(imm5 == 00000 && stype == 11)`.

```
ANDS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15 14 13 12	11 10 9 8	7 6 5	3 2 0
0 1 0 0 0 0	0 0 0 0	Rm	Rdn

T1 variant

```
AND<c>{<q>} {<Rdn>,} <Rdn>, <Rm> // Inside IT block
ANDS{<q>} {<Rdn>,} <Rdn>, <Rm> // Outside IT block
```

Decode for this encoding

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 14 12	11 8	7 6 5 4	3 0
1 1 1 0 1 0 1	0 0 0 0	S	Rn	(0) imm3	Rd	imm2 stype	Rm

AND, rotate right with extend variant

Applies when `S == 0 && imm3 == 000 && imm2 == 00 && stype == 11`.

```
AND{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

AND, shift or rotate by value variant

Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11)`.

```
AND<c>.W {<Rd>,} <Rn>, <Rm> // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
AND{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

ANDS, rotate right with extend variant

Applies when `S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11`.

```
ANDS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

ANDS, shift or rotate by value variant

Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11) && Rd != 1111`.

```
ANDS.W {<Rd>,} <Rn>, <Rm> // Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
ANDS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then SEE "TST (register)";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if (d == 15 && !setflags) || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rdn>` Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the AND variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- For the ANDS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`.

`<Rn>` For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T2: is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

`LSL`	when `stype = 00`
`LSR`	when `stype = 01`
`ASR`	when `stype = 10`
`ROR`	when `stype = 11`

`<amount>` For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

In T32 assembly:

- Outside an IT block, if `ANDS <Rd>, <Rn>, <Rd>` has `<Rd>` and `<Rn>` both in the range R0-R7, it is assembled using encoding T1 as though `ANDS <Rd>, <Rn>` had been written.
- Inside an IT block, if `AND<c> <Rd>, <Rn>, <Rd>` has `<Rd>` and `<Rn>` both in the range R0-R7, it is assembled using encoding T1 as though `AND<c> <Rd>, <Rn>` had been written.

To prevent either of these happening, use the `.W` qualifier.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] AND shifted;
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
else
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.13 AND, ANDS (register-shifted register)

Bitwise AND (register-shifted register) performs a bitwise AND of a register value and a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.

A1

31–28	27–24	23–20		19–16	15–12	11–8	7	6–5	4	3–0
!=1111 (cond)	0 0 0 0	0 0 0	S	Rn	Rd	Rs	0	stype	1	Rm

Flag setting variant

Applies when `S == 1`.

```
ANDS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

Not flag setting variant

Applies when `S == 0`.

```
AND{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

- `LSL` when `stype = 00`
- `LSR` when `stype = 01`
- `ASR` when `stype = 10`
- `ROR` when `stype = 11`

`<Rs>` Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] AND shifted;
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.14 ASR (immediate)

Arithmetic Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in copies of its sign bit, and writes the result to the destination register.

This instruction is an alias of the MOV, MOVS (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

A1

31–28	27 26 25 24	23 22 21 20	19 18 17 16	15–12	11–7	6 5	4	3–0
!=1111	0 0 0 1 1	0 1 0	(0) (0) (0) (0)	Rd	imm5	1 0	0	Rm
cond		S				stype		

MOV, shift or rotate by value variant

```
ASR{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ASR #<imm>
```

and is always the preferred disassembly.

T2

15 14 13	12 11	10–6	5–3	2–0
0 0 0	1 0	imm5	Rm	Rd
	op			

T2 variant

```
ASR<c>{<q>} {<Rd>,} <Rm>, #<imm> // Inside IT block
```

is equivalent to

```
MOV<c>{<q>} <Rd>, <Rm>, ASR #<imm>
```

and is the preferred disassembly when `InITBlock()`.

T3

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 14	12–11	8	7 6	5 4	3–0
1 1 1 0	1 0 1	0 0 1 0	0	1 1 1 1	(0)	imm3	Rd	imm2	1 0	Rm
			S					stype	

MOV, shift or rotate by value variant

```
ASR<c>.W {<Rd>,} <Rm>, #<imm> // Inside IT block, and <Rd>, <Rm>, <imm> can be represented in T2
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ASR #<imm>
```

and is always the preferred disassembly.

```
ASR{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ASR #<imm>
```

and is always the preferred disassembly.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.

For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.

`<imm>` For encoding A1 and T2: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as <imm> modulo 32.

For encoding T3: is the shift amount, in the range 1 to 32, encoded in the "imm3:imm2" field as <imm> modulo 32.

Operation for all encodings

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.15 ASR (register)

Arithmetic Shift Right (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

A1

31–28	27–25	24–23	22–21	20	19–16	15–12	11–8	7	6–5	4	3–0
!=1111	0 0 0	1 1	0 1	0	(0) (0) (0) (0)	Rd	Rs	0	1 0	1	Rm
cond				S					stype		

Not flag setting variant

```
ASR{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ASR <Rs>
```

and is always the preferred disassembly.

T1

15–14	13–12	11–10	9–6	5–3	2–0
0 1	0 0	0 0	0 1 0 0	Rs	Rdm
			op		

Arithmetic shift right variant

```
ASR<c>{<q>} {<Rdm>,} <Rdm>, <Rs> // Inside IT block
```

is equivalent to

```
MOV<c>{<q>} <Rdm>, <Rdm>, ASR <Rs>
```

and is the preferred disassembly when `InITBlock()`.

T2

15–14	13–12	11–10	9–8	7–6	5	4	3–0	15–14	13–12	11–8	7–6	5–4	3–0
1 1	1 1	1 0	1 0	1 0		0	Rm	1 1	1 1	Rd	0 0	0 0	Rs
				stype		S							

Not flag setting variant

```
ASR<c>.W {<Rd>,} <Rm>, <Rs> // Inside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ASR <Rs>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

and is always the preferred disassembly.

```
ASR{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ASR <Rs>
```

and is always the preferred disassembly.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rdm>`	Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<Rm>`	Is the first general-purpose source register, encoded in the "Rm" field.
`<Rs>`	Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation for all encodings

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.16 ASRS (immediate)

Arithmetic Shift Right, setting flags (immediate) shifts a register value right by an immediate number of bits, shifting in copies of its sign bit, and writes the result to the destination register.

If the destination register is not the PC, this instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This instruction is an alias of the MOV, MOVS (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

A1

31–28	27 26 25 24	23 22 21 20	19 18 17 16	15–12	11–7	6 5	4	3–0
!=1111	0 0 0 1 1	0 1 1	(0) (0) (0) (0)	Rd	imm5	1 0	0	Rm
cond		S				stype		

MOVS, shift or rotate by value variant

```
ASRS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ASR #<imm>
```

and is always the preferred disassembly.

T2

15 14 13 12	11 10	6 5	3 2 0
0 0 0 1 0	imm5	Rm	Rd
op			

T2 variant

```
ASRS{<q>} {<Rd>,} <Rm>, #<imm> // Outside IT block
```

is equivalent to

```
MOVS{<q>} <Rd>, <Rm>, ASR #<imm>
```

and is the preferred disassembly when `!InITBlock()`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T3

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 14 12	11 8	7 6 5 4	3 0
1 1 1 0	1 0 1 0	0 0 1 0	1 1 1 1 1	(0) imm3	Rd	imm2 1 0	Rm
			S			stype	

MOVS, shift or rotate by value variant

```
ASRS.W {<Rd>,} <Rm>, #<imm> // Outside IT block, and <Rd>, <Rm>, <imm> can be represented in T2
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ASR #<imm>
```

and is always the preferred disassembly.

```
ASRS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ASR #<imm>
```

and is always the preferred disassembly.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.

`<imm>` For encoding A1 and T2: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as <imm> modulo 32.

For encoding T3: is the shift amount, in the range 1 to 32, encoded in the "imm3:imm2" field as <imm> modulo 32.

Operation for all encodings

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.17 ASRS (register)

Arithmetic Shift Right, setting flags (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

A1

31–28	27–20								19–16	15–12	11–8	7	6–5	4	3–0
!=1111	0	0	0	1	1	0	1	1	(0) (0) (0) (0)	Rd	Rs	0	1 0	1	Rm
cond								S					stype		

Flag setting variant

```
ASRS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ASR <Rs>
```

and is always the preferred disassembly.

T1

15–10	9–6	5–3	2–0
0 1 0 0 0 0	0 1 0 0	Rs	Rdm
	op		

Arithmetic shift right variant

```
ASRS{<q>} {<Rdm>,} <Rdm>, <Rs> // Outside IT block
```

is equivalent to

```
MOVS{<q>} <Rdm>, <Rdm>, ASR <Rs>
```

and is the preferred disassembly when `!InITBlock()`.

T2

15–11	10–9	8–7	6–5	4	3–0	15–12	11–8	7–4	3–0
1 1 1 1 1	0 1	0 0	1 0	1	Rm	1 1 1 1	Rd	0 0 0 0	Rs
			stype	S					

Flag setting variant

```
ASRS.W {<Rd>,} <Rm>, <Rs> // Outside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ASR <Rs>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

and is always the preferred disassembly.

```
ASRS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ASR <Rs>
```

and is always the preferred disassembly.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rdm>`	Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<Rm>`	Is the first general-purpose source register, encoded in the "Rm" field.
`<Rs>`	Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation for all encodings

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.18 B

Branch causes a branch to a target address.

A1

31–28	27 26 25	24	23–0
!=1111 (cond)	1 0 1	0	imm24

A1 variant

```
B{<c>}{<q>} <label>
```

Decode for this encoding

```
imm32 = SignExtend(imm24:'00', 32);
```

T1

15 14 13 12	11–8	7–0
1 1 0 1	!=111x (cond)	imm8

T1 variant

```
B<c>{<q>} <label> // Not permitted in IT block
```

Decode for this encoding

```
if cond == '1110' then SEE "UDF";
if cond == '1111' then SEE "SVC";
imm32 = SignExtend(imm8:'0', 32);
if InITBlock() then UNPREDICTABLE;
```

T2

15 14 13 12	11 10	0
1 1 1 0 0	imm11	

T2 variant

```
B{<c>}{<q>} <label> // Outside or last in IT block
```

Decode for this encoding

```
imm32 = SignExtend(imm11:'0', 32);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

T3

15 14 13 12	11	10–9	6 5	0	15 14 13 12	11	10	9	11 10	0
1 1 1 1 0	S	!=111x (cond)	imm6		1 0	J1	0	J2	imm11	

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T3 variant

```
B<c>.W <label> // Not permitted in IT block, and <label> can be represented in T1
B<c>{<q>} <label> // Not permitted in IT block
```

Decode for this encoding

```
if cond<3:1> == '111' then SEE "Related encodings";
imm32 = SignExtend(S:J2:J1:imm6:imm11:'0', 32);
if InITBlock() then UNPREDICTABLE;
```

T4

15	14	13	12	11	10	9	…	0	15	14	13	12	11	10	…	0
1	1	1	1	0	S	imm10			1	0	J1	1	J2	imm11		

T4 variant

```
B{<c>}.W <label> // <label> can be represented in T2
B{<c>}{<q>} <label>
```

Decode for this encoding

```
I1 = NOT(J1 EOR S);  I2 = NOT(J2 EOR S);  imm32 = SignExtend(S:I1:I2:imm10:imm11:'0', 32);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Related encodings: *Branches and miscellaneous control* on page F3-4154.

Assembler symbols

`<c>` For encoding A1, T2 and T4: see *Standard assembler syntax fields* on page F2-4080.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080. Must not be `AL` or omitted.

For encoding T3: see *Standard assembler syntax fields* on page F2-4080. `<c>` must not be `AL` or omitted.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<label>` For encoding A1: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the `B` instruction to this label, then selects an encoding that sets `imm32` to that offset.

Permitted offsets are multiples of 4 in the range –33554432 to 33554428.

For encoding T1: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the `B` instruction to this label, then selects an encoding that sets `imm32` to that offset. Permitted offsets are even numbers in the range –256 to 254.

For encoding T2: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the `B` instruction to this label, then selects an encoding that sets `imm32` to that offset. Permitted offsets are even numbers in the range –2048 to 2046.

For encoding T3: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the `B` instruction to this label, then selects an encoding that sets `imm32` to that offset.

Permitted offsets are even numbers in the range –1048576 to 1048574.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For encoding T4: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the `B` instruction to this label, then selects an encoding that sets `imm32` to that offset.

Permitted offsets are even numbers in the range –16777216 to 16777214.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    BranchWritePC(PC + imm32, BranchType_DIR);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.19 BFC

Bit Field Clear clears any number of adjacent bits at any position in a register, without affecting the other bits in the register.

A1

31–28	27 26 25 24	23 22 21 20	16	15–12	11–7	6 5	4	3 2 1 0
!=1111 (cond)	0 1 1 1	1 1 1 0	msb	Rd	lsb	0 0	1	1 1 1 1

A1 variant

```
BFC{<c>}{<q>} <Rd>, #<lsb>, #<width>
```

Decode for this encoding

```
d = UInt(Rd);  msbit = UInt(msb);  lsbit = UInt(lsb);
if d == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10	9 8	7 6	5	4	3 2 1 0	15	14–12	11–8	7 6	5	4–0
1 1 1 1 0	(0)	1 1	0 1	1	0	1 1 1 1	0	imm3	Rd	imm2	(0)	msb

T1 variant

```
BFC{<c>}{<q>} <Rd>, #<lsb>, #<width>
```

Decode for this encoding

```
d = UInt(Rd);  msbit = UInt(msb);  lsbit = UInt(imm3:imm2);
if d == 15 then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<lsb>`	For encoding A1: is the least significant bit to be cleared, in the range 0 to 31, encoded in the "lsb" field. For encoding T1: is the least significant bit that is to be cleared, in the range 0 to 31, encoded in the "imm3:imm2" field.
`<width>`	Is the number of bits to be cleared, in the range 1 to 32-<lsb>, encoded in the "msb" field as <lsb>+<width>-1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if msbit >= lsbit then
        R[d]<msbit:lsbit> = Replicate('0', msbit-lsbit+1);
        // Other bits of R[d] are unchanged
    else
        UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `msbit < lsbit`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.20 BFI

Bit Field Insert copies any number of low order bits from a register into the same number of adjacent bits at any position in the destination register.

A1

31–28	27 26 25 24	23 22 21 20	16	15–12	11–7	6 5 4	3–0
!=1111	0 1 1 1	1 1 1 0	msb	Rd	lsb	0 0 1	!=1111
cond							Rn

A1 variant

```
BFI{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>
```

Decode for this encoding

```
if Rn == '1111' then SEE "BFC";
d = UInt(Rd);  n = UInt(Rn);  msbit = UInt(msb);  lsbit = UInt(lsb);
if d == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10	9 8	7 6	5 4	3–0	15	14–12	11–8	7 6	5 4	0
1 1 1 1	0 (0)	1 1	0 1	1 0	!=1111	0	imm3	Rd	imm2	(0)	msb
					Rn						

T1 variant

```
BFI{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>
```

Decode for this encoding

```
if Rn == '1111' then SEE "BFC";
d = UInt(Rd);  n = UInt(Rn);  msbit = UInt(msb);  lsbit = UInt(imm3:imm2);
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<Rn>`	Is the general-purpose source register, encoded in the "Rn" field.
`<lsb>`	For encoding A1: is the least significant destination bit, in the range 0 to 31, encoded in the "lsb" field. For encoding T1: is the least significant destination bit, in the range 0 to 31, encoded in the "imm3:imm2" field.
`<width>`	Is the number of bits to be copied, in the range 1 to 32-<lsb>, encoded in the "msb" field as <lsb>+<width>-1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if msbit >= lsbit then
        R[d]<msbit:lsbit> = R[n]<(msbit-lsbit):0>;
        // Other bits of R[d] are unchanged
    else
        UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `msbit < lsbit`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.21 BIC, BICS (immediate)

Bitwise Bit Clear (immediate) performs a bitwise AND of a register value and the complement of an immediate value, and writes the result to the destination register.

If the destination register is not the PC, the BICS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The BIC variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- The BICS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–0
!=1111	0 0 1 1 1	1 0 S	Rn	Rd	imm12
cond					

BIC variant

Applies when `S == 0`.

```
BIC{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

BICS variant

Applies when `S == 1`.

```
BICS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = A32ExpandImm_C(imm12, PSTATE.C);
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15 14–12	11–8	7–0
1 1 1 1 0	i 0	0 0 0 1 S	Rn	0 imm3	Rd	imm8

BIC variant

Applies when `S == 0`.

```
BIC{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

BICS variant

Applies when `S == 1`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
BICS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
if d == 15 || n == 15 then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the BIC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- For the BICS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`.

`<Rn>` For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

`<const>` For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* on page F2-4096 for the range of values.

For encoding T1: an immediate value. See *Modified immediate constants in T32 instructions* on page F2-4095 for the range of values.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] AND NOT(imm32);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.22 BIC, BICS (register)

Bitwise Bit Clear (register) performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the BICS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The BIC variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- The BICS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–7	6 5	4	3–0
!=1111 (cond)	0 0 0 1 1	1 0 S	Rn	Rd	imm5	stype	0	Rm

BIC, rotate right with extend variant

Applies when `S == 0 && imm5 == 00000 && stype == 11`.

```
BIC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

BIC, shift or rotate by value variant

Applies when `S == 0 && !(imm5 == 00000 && stype == 11)`.

```
BIC{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

BICS, rotate right with extend variant

Applies when `S == 1 && imm5 == 00000 && stype == 11`.

```
BICS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

BICS, shift or rotate by value variant

Applies when `S == 1 && !(imm5 == 00000 && stype == 11)`.

```
BICS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15 14 13 12	11 10 9 8	7 6 5	3 2 0
0 1 0 0 0 0	1 1 1 0	Rm	Rdn

T1 variant

```
BIC<c>{<q>} {<Rdn>,} <Rdn>, <Rm> // Inside IT block
BICS{<q>} {<Rdn>,} <Rdn>, <Rm> // Outside IT block
```

Decode for this encoding

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 14 12	11 8	7 6 5 4	3 0
1 1 1 0 1 0 1	0 0 0 1	S	Rn	(0) imm3	Rd	imm2 stype	Rm

BIC, rotate right with extend variant

Applies when `S == 0 && imm3 == 000 && imm2 == 00 && stype == 11`.

```
BIC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

BIC, shift or rotate by value variant

Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11)`.

```
BIC<c>.W {<Rd>,} <Rn>, <Rm> // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
BIC{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

BICS, rotate right with extend variant

Applies when `S == 1 && imm3 == 000 && imm2 == 00 && stype == 11`.

```
BICS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

BICS, shift or rotate by value variant

Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11)`.

```
BICS.W {<Rd>,} <Rn>, <Rm> // Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
BICS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rdn>` Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the BIC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- For the BICS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`.

`<Rn>` For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T2: is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

`LSL`	when `stype = 00`
`LSR`	when `stype = 01`
`ASR`	when `stype = 10`
`ROR`	when `stype = 11`

`<amount>` For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] AND NOT(shifted);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.23 BIC, BICS (register-shifted register)

Bitwise Bit Clear (register-shifted register) performs a bitwise AND of a register value and the complement of a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.

A1

31–28	27 26 25 24	23 22 21 20		19–16	15–12	11–8	7	6 5	4	3–0
!=1111	0 0 0 1	1 1 0	S	Rn	Rd	Rs	0	stype	1	Rm
cond										

Flag setting variant

Applies when `S == 1`.

```
BICS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

Not flag setting variant

Applies when `S == 0`.

```
BIC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

- `LSL` when `stype = 00`
- `LSR` when `stype = 01`
- `ASR` when `stype = 10`
- `ROR` when `stype = 11`

`<Rs>` Is the general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] AND NOT(shifted);
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.24 BKPT

Breakpoint causes a Breakpoint Instruction exception.

Breakpoint is always unconditional, even when inside an IT block.

A1

31–28	27 26 25 24	23 22 21 20	19–8	7 6 5 4	3–0
!=1111 (cond)	0 0 0 1 0	0 1 0	imm12	0 1 1 1	imm4

A1 variant

```
BKPT{<q>} {#}<imm>
```

Decode for this encoding

```
imm16 = imm12:imm4;
if cond != '1110' then UNPREDICTABLE;  // BKPT must be encoded with AL condition
```

CONSTRAINED UNPREDICTABLE behavior

If `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1

15 14 13 12	11 10 9 8	7–0
1 0 1 1	1 1 1 0	imm8

T1 variant

```
BKPT{<q>} {#}<imm>
```

Decode for this encoding

```
imm16 = ZeroExtend(imm8, 16);
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<q>` See *Standard assembler syntax fields* on page F2-4080. An `BKPT` instruction must be unconditional.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<imm>` For encoding A1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm12:imm4" field. This value:

- Is recorded in the Comment field of ESR_ELx.ISS if the Software Breakpoint Instruction exception is taken to an Exception level that is using AArch64.
- Is ignored otherwise.

For encoding T1: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field. This value:

- Is recorded in the Comment field of ESR_ELx.ISS if the Software Breakpoint Instruction exception is taken to an Exception level that is using AArch64.
- Is ignored otherwise.

Operation for all encodings

```
EncodingSpecificOperations();
AArch32.SoftwareBreakpoint(imm16);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.25 BL, BLX (immediate)

Branch with Link calls a subroutine at a PC-relative address, and setting LR to the return address.

Branch with Link and Exchange Instruction Sets (immediate) calls a subroutine at a PC-relative address, setting LR to the return address, and changes the instruction set from A32 to T32, or from T32 to A32.

A1

31–28	27 26 25	24	23–0
!=1111 (cond)	1 0 1	1	imm24

A1 variant

```
BL{<c>}{<q>} <label>
```

Decode for this encoding

```
imm32 = SignExtend(imm24:'00', 32);  targetInstrSet = InstrSet_A32;
```

A2

31–28	27 26 25	24	23–0
1 1 1 1 (cond)	1 0 1	H	imm24

A2 variant

```
BLX{<c>}{<q>} <label>
```

Decode for this encoding

```
imm32 = SignExtend(imm24:H:'0', 32);  targetInstrSet = InstrSet_T32;
```

T1

15 14 13 12	11 10 9	0	15 14 13 12	11 10	0
1 1 1 1 0	S	imm10	1 1 J1 1	J2	imm11

T1 variant

```
BL{<c>}{<q>} <label>
```

Decode for this encoding

```
I1 = NOT(J1 EOR S);  I2 = NOT(J2 EOR S);  imm32 = SignExtend(S:I1:I2:imm10:imm11:'0', 32);
targetInstrSet = InstrSet_T32;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

T2

15 14 13 12	11 10 9	0	15 14 13 12	11 10	1	0
1 1 1 1 0	S	imm10H	1 1 J1 0	J2	imm10L	H

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T2 variant

```
BLX{<c>}{<q>} <label>
```

Decode for this encoding

```
if H == '1' then UNDEFINED;
I1 = NOT(J1 EOR S);  I2 = NOT(J2 EOR S);  imm32 = SignExtend(S:I1:I2:imm10H:imm10L:'00', 32);
targetInstrSet = InstrSet_A32;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` For encoding A1, T1 and T2: see *Standard assembler syntax fields* on page F2-4080.

For encoding A2: see *Standard assembler syntax fields* on page F2-4080. `<c>` must be `AL` or omitted.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<label>` For encoding A1: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the `BL` instruction to this label, then selects an encoding that sets `imm32` to that offset.

Permitted offsets are multiples of 4 in the range –33554432 to 33554428.

For encoding A2: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the `BLX` instruction to this label, then selects an encoding with `imm32` set to that offset.

Permitted offsets are even numbers in the range –33554432 to 33554430.

For encoding T1: the label of the instruction that is to be branched to.

The assembler calculates the required value of the offset from the PC value of the `BL` instruction to this label, then selects an encoding with `imm32` set to that offset.

Permitted offsets are even numbers in the range –16777216 to 16777214.

For encoding T2: the label of the instruction that is to be branched to.

The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the `BLX` instruction to this label, then selects an encoding with `imm32` set to that offset.

Permitted offsets are multiples of 4 in the range –16777216 to 16777212.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if CurrentInstrSet() == InstrSet_A32 then
        LR = PC - 4;
    else
        LR = PC<31:1> : '1';
    if targetInstrSet == InstrSet_A32 then
        targetAddress = Align(PC,4) + imm32;
    else
        targetAddress = PC + imm32;
    SelectInstrSet(targetInstrSet);
    BranchWritePC(targetAddress, BranchType_DIRCALL);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.26 BLX (register)

Branch with Link and Exchange (register) calls a subroutine at an address specified in the register, and if necessary changes to the instruction set indicated by bit[0] of the register value. If the value in bit[0] is 0, the instruction set after the branch will be A32. If the value in bit[0] is 1, the instruction set after the branch will be T32.

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3–0
!=1111	0	0	0	1	0	0	1	0	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	0	0	1	1	Rm
cond																									

A1 variant

```
BLX{<c>}{<q>} <Rm>
```

Decode for this encoding

```
m = UInt(Rm);
if m == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6–3	2	1	0
0	1	0	0	0	1	1	1	1	Rm	(0)	(0)	(0)

T1 variant

```
BLX{<c>}{<q>} <Rm>
```

Decode for this encoding

```
m = UInt(Rm);
if m == 15 then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

<Rm> Is the general-purpose register holding the address to be branched to, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    target = R[m];
    if CurrentInstrSet() == InstrSet_A32 then
        next_instr_addr = PC - 4;
        LR = next_instr_addr;
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    next_instr_addr = PC - 2;
    LR = next_instr_addr<31:1> : '1';
BXWritePC(target, BranchType_INDCALL);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.27 BX

Branch and Exchange causes a branch to an address and instruction set specified by a register.

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3–0
!=1111	0	0	0	1	0	0	1	0	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	0	0	0	1	Rm
cond																									

A1 variant

```
BX{<c>}{<q>} <Rm>
```

Decode for this encoding

```
m = UInt(Rm);
```

T1

15	14	13	12	11	10	9	8	7	6–3	2	1	0
0	1	0	0	0	1	1	1	0	Rm	(0)	(0)	(0)

T1 variant

```
BX{<c>}{<q>} <Rm>
```

Decode for this encoding

```
m = UInt(Rm);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rm>` For encoding A1: is the general-purpose register holding the address to be branched to, encoded in the "Rm" field. The PC can be used.

For encoding T1: is the general-purpose register holding the address to be branched to, encoded in the "Rm" field. The PC can be used.

Note

If `<Rm>` is the PC at a non word-aligned address, it results in UNPREDICTABLE behavior because the address passed to the `BXWritePC()` pseudocode function has bits<1:0> = '10'.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    BXWritePC(R[m], BranchType_INDIR);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.28 BXJ

Branch and Exchange, previously Branch and Exchange Jazelle.

In Armv8, `BXJ` behaves as a `BX` instruction, see BX. This means it causes a branch to an address and instruction set specified by a register.

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3–0
!=1111	0	0	0	1	0	0	1	0	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	0	0	1	0	Rm
cond																									

A1 variant

```
BXJ{<c>}{<q>} <Rm>
```

Decode for this encoding

```
m = UInt(Rm);
if m == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	1	0	0	Rm	1	0	(0)	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

T1 variant

```
BXJ{<c>}{<q>} <Rm>
```

Decode for this encoding

```
m = UInt(Rm);
if m == 15 then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rm>` Is the general-purpose register holding the address to be branched to, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    BXWritePC(R[m], BranchType_INDIR);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.29 CBNZ, CBZ

Compare and Branch on Nonzero and Compare and Branch on Zero compare the value in a register with zero, and conditionally branch forward a constant value. They do not affect the condition flags.

T1

15 14 13 12	11	10	9	8	7 … 3	2 … 0
1 0 1 1	op	0	i	1	imm5	Rn

CBNZ variant

Applies when `op == 1`.

```
CBNZ{<q>} <Rn>, <label>
```

CBZ variant

Applies when `op == 0`.

```
CBZ{<q>} <Rn>, <label>
```

Decode for all variants of this encoding

```
n = UInt(Rn);  imm32 = ZeroExtend(i:imm5:'0', 32);  nonzero = (op == '1');
if InITBlock() then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rn>` Is the general-purpose register to be tested, encoded in the "Rn" field.

`<label>` Is the program label to be conditionally branched to. Its offset from the PC, a multiple of 2 and in the range 0 to 126, is encoded as "i:imm5" times 2.

Operation

```
EncodingSpecificOperations();
if nonzero != IsZero(R[n]) then
    BranchWritePC(PC + imm32, BranchType_DIR);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.30 CLREX

Clear-Exclusive clears the local monitor of the executing PE.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	1	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	1	(1)	(1)	(1)	(1)

A1 variant

```
CLREX{<c>}{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	0	1	0	(1)	(1)	(1)	(1)

T1 variant

```
CLREX{<c>}{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-4080. Must be AL or omitted.
For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ClearExclusiveLocal(ProcessorID());
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.31 CLZ

Count Leading Zeros returns the number of binary zero bits before the first binary one bit in a value.

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111 (cond)	0	0	0	1	0	1	1	0	(1)	(1)	(1)	(1)	Rd	(1)	(1)	(1)	(1)	0	0	0	1	Rm

A1 variant

```
CLZ{<c>}{<q>} <Rd>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15	14	13	12	11–8	7	6	5	4	3–0
1	1	1	1	1	0	1	0	1	0	1	1	Rn	1	1	1	1	Rd	1	0	0	0	Rm

T1 variant

```
CLZ{<c>}{<q>} <Rd>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);  n = UInt(Rn);
if m != n || d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `m != n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction executes with the additional decode: m = UInt(Rn);.
- The value in the destination register is UNKNOWN.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Rm>` For encoding A1: is the general-purpose source register, encoded in the "Rm" field.

For encoding T1: is the general-purpose source register, encoded in the "Rm" field. It must be encoded with an identical value in the "Rn" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = CountLeadingZeroBits(R[m]);
    R[d] = result<31:0>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.32 CMN (immediate)

Compare Negative (immediate) adds a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11–0
!=1111 (cond)	0	0	1	1	0	1	1	1	Rn	(0)	(0)	(0)	(0)	imm12

A1 variant

```
CMN{<c>}{<q>} <Rn>, #<const>
```

Decode for this encoding

```
n = UInt(Rn);  imm32 = A32ExpandImm(imm12);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15	14–12	11	10	9	8	7–0
1	1	1	1	0	i	0	1	0	0	0	1	Rn	0	imm3	1	1	1	1	imm8

T1 variant

```
CMN{<c>}{<q>} <Rn>, #<const>
```

Decode for this encoding

```
n = UInt(Rn);  imm32 = T32ExpandImm(i:imm3:imm8);
if n == 15 then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rn>` For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

`<const>` For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* on page F2-4096 for the range of values.

For encoding T1: an immediate value. See *Modified immediate constants in T32 instructions* on page F2-4095 for the range of values.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, nzcv) = AddWithCarry(R[n], imm32, '0');
    PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.33 CMN (register)

Compare Negative (register) adds a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11–7	6–5	4	3–0
!=1111	0	0	0	1	0	1	1	1	Rn	(0)	(0)	(0)	(0)	imm5	stype	0	Rm
cond																	

Rotate right with extend variant

Applies when `imm5 == 00000 && stype == 11`.

```
CMN{<c>}{<q>} <Rn>, <Rm>, RRX
```

Shift or rotate by value variant

Applies when `!(imm5 == 00000 && stype == 11)`.

```
CMN{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```

T1

15	14	13	12	11	10	9	8	7	6	5–3	2–0
0	1	0	0	0	0	1	0	1	1	Rm	Rn

T1 variant

```
CMN{<c>}{<q>} <Rn>, <Rm>
```

Decode for this encoding

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15	14–12	11	10	9	8	7	6	5	4	3–0
1	1	1	0	1	0	1	1	0	0	0	1	Rn	(0)	imm3	1	1	1	1	imm2		stype		Rm

Rotate right with extend variant

Applies when `imm3 == 000 && imm2 == 00 && stype == 11`.

```
CMN{<c>}{<q>} <Rn>, <Rm>, RRX
```

Shift or rotate by value variant

Applies when `!(imm3 == 000 && imm2 == 00 && stype == 11)`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
CMN{<c>}.W <Rn>, <Rm> // <Rn>, <Rm> can be represented in T1
CMN{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rn>` For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

`LSL`	when `stype = 00`
`LSR`	when `stype = 01`
`ASR`	when `stype = 10`
`ROR`	when `stype = 11`

`<amount>` For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], shifted, '0');
    PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.34 CMN (register-shifted register)

Compare Negative (register-shifted register) adds a register value and a register-shifted register value. It updates the condition flags based on the result, and discards the result.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11–8	7	6–5	4	3–0
!=1111 (cond)	0	0	0	1	0	1	1	1	Rn	(0)	(0)	(0)	(0)	Rs	0	stype	1	Rm

A1 variant

```
CMN{<c>}{<q>} <Rn>, <Rm>, <type> <Rs>
```

Decode for this encoding

```
n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
shift_t = DecodeRegShift(stype);
if n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<type>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

- `LSL` when `stype = 00`
- `LSR` when `stype = 01`
- `ASR` when `stype = 10`
- `ROR` when `stype = 11`

`<Rs>` Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], shifted, '0');
    PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.35 CMP (immediate)

Compare (immediate) subtracts an immediate value from a register value. It updates the condition flags based on the result, and discards the result.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15 14 13 12	11–0
!=1111 (cond)	0 0 1 1 0	1 0 1	Rn	(0)(0)(0)(0)	imm12

A1 variant

```
CMP{<c>}{<q>} <Rn>, #<const>
```

Decode for this encoding

```
n = UInt(Rn);  imm32 = A32ExpandImm(imm12);
```

T1

15 14 13	12 11	10–8	7–0
0 0 1	0 1	Rn	imm8

T1 variant

```
CMP{<c>}{<q>} <Rn>, #<imm8>
```

Decode for this encoding

```
n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
```

T2

15 14 13 12	11	10	9	8	7 6 5 4	3–0	15	14–12	11 10 9 8	7–0
1 1 1 1 0	i	0	1	1	0 1 1	Rn	0	imm3	1 1 1 1	imm8

T2 variant

```
CMP{<c>}.W <Rn>, #<const> // <Rd>, <const> can be represented in T1
CMP{<c>}{<q>} <Rn>, #<const>
```

Decode for this encoding

```
n = UInt(Rn);  imm32 = T32ExpandImm(i:imm3:imm8);
if n == 15 then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rn>` For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1: is a general-purpose source register, encoded in the "Rn" field.

For encoding T2: is the general-purpose source register, encoded in the "Rn" field.

`<imm8>` Is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.

`<const>` For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* on page F2-4096 for the range of values.

For encoding T2: an immediate value. See *Modified immediate constants in T32 instructions* on page F2-4095 for the range of values.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, nzcv) = AddWithCarry(R[n], NOT(imm32), '1');
    PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.36 CMP (register)

Compare (register) subtracts an optionally-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15 14 13 12	11–7	6 5	4	3–0
!=1111 (cond)	0 0 0 1 0	1 0 1	Rn	(0)(0)(0)(0)	imm5	stype	0	Rm

Rotate right with extend variant

Applies when `imm5 == 00000 && stype == 11`.

```
CMP{<c>}{<q>} <Rn>, <Rm>, RRX
```

Shift or rotate by value variant

Applies when `!(imm5 == 00000 && stype == 11)`.

```
CMP{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```

T1

15 14 13 12	11 10 9 8	7 6 5	3 2 0
0 1 0 0 0 0	1 0 1 0	Rm	Rn

T1 variant

```
CMP{<c>}{<q>} <Rn>, <Rm> // <Rn> and <Rm> both from R0-R7
```

Decode for this encoding

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2

15 14 13 12	11 10 9 8	7	6	3 2 0
0 1 0 0 0 1	0 1	N	Rm	Rn

T2 variant

```
CMP{<c>}{<q>} <Rn>, <Rm> // <Rn> and <Rm> not both from R0-R7
```

Decode for this encoding

```
n = UInt(N:Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
if n < 8 && m < 8 then UNPREDICTABLE;
if n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CONSTRAINED UNPREDICTABLE behavior

If `n < 8 && m < 8`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The condition flags become UNKNOWN.

T3

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 14 12	11 10 9 8	7 6 5 4	3 0
1 1 1 0	1 0 1	1 1 0 1	1	Rn	(0) imm3	1 1 1 1	imm2 stype	Rm

Rotate right with extend variant

Applies when `imm3 == 000 && imm2 == 00 && stype == 11`.

```
CMP{<c>}{<q>} <Rn>, <Rm>, RRX
```

Shift or rotate by value variant

Applies when `!(imm3 == 000 && imm2 == 00 && stype == 11)`.

```
CMP{<c>}.W <Rn>, <Rm> // <Rn>, <Rm> can be represented in T1 or T2
CMP{<c>}{<q>} <Rn>, <Rm>, <shift> #<amount>
```

Decode for all variants of this encoding

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rn>` For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1 and T3: is the first general-purpose source register, encoded in the "Rn" field.

For encoding T2: is the first general-purpose source register, encoded in the "N:Rn" field.

`<Rm>` For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1, T2 and T3: is the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

`LSL` when `stype = 00`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

LSR	when stype = 01
ASR	when stype = 10
ROR	when stype = 11

<amount> For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T3: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], NOT(shifted), '1');
    PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.37 CMP (register-shifted register)

Compare (register-shifted register) subtracts a register-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15 14 13 12	11–8	7	6 5	4	3–0
!=1111 (cond)	0 0 0 1 0	1 0 1	Rn	(0)(0)(0)(0)	Rs	0	stype	1	Rm

A1 variant

```
CMP{<c>}{<q>} <Rn>, <Rm>, <type> <Rs>
```

Decode for this encoding

```
n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
shift_t = DecodeRegShift(stype);
if n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<type>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

- `LSL` when `stype = 00`
- `LSR` when `stype = 01`
- `ASR` when `stype = 10`
- `ROR` when `stype = 11`

`<Rs>` Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], NOT(shifted), '1');
    PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.38 CPS, CPSID, CPSIE

Change PE State changes one or more of the PSTATE.{A, I, F} interrupt mask bits and, optionally, the PSTATE.M mode field, without changing any other PSTATE bits.

`CPS` is treated as `NOP` if executed in User mode unless it is defined as being CONSTRAINED UNPREDICTABLE elsewhere in this section.

The PE checks whether the value being written to PSTATE.M is legal. See *Illegal changes to PSTATE.M* on page G1-5699.

A1

31 30 29 28	27 26 25 24	23 22 21 20	19 18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	0
1 1 1 1	0 0 0 1	0 0 0 0	imod	M	0	(0)	(0)	(0)	(0)	(0)	(0)	(0)	A	I	F	0	mode	

CPS variant

Applies when `imod == 00 && M == 1`.

```
CPS{<q>} #<mode> // Cannot be conditional
```

CPSID variant

Applies when `imod == 11 && M == 0`.

```
CPSID{<q>} <iflags> // Cannot be conditional
```

CPSID variant

Applies when `imod == 11 && M == 1`.

```
CPSID{<q>} <iflags> , #<mode> // Cannot be conditional
```

CPSIE variant

Applies when `imod == 10 && M == 0`.

```
CPSIE{<q>} <iflags> // Cannot be conditional
```

CPSIE variant

Applies when `imod == 10 && M == 1`.

```
CPSIE{<q>} <iflags> , #<mode> // Cannot be conditional
```

Decode for all variants of this encoding

```
if mode != '00000' && M == '0' then UNPREDICTABLE;
if (imod<1> == '1' && A:I:F == '000') || (imod<1> == '0' && A:I:F != '000') then UNPREDICTABLE;
enable = (imod == '10');  disable = (imod == '11');  changemode = (M == '1');
affectA = (A == '1');  affectI = (I == '1');  affectF = (F == '1');
if (imod == '00' && M == '0') || imod == '01' then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `imod == '01'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If `imod == '00' && M == '0'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

If `mode != '00000' && M == '0'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: changemode = TRUE.
- The instruction executes as described, and the value specified by mode is ignored. There are no additional side-effects.

If `imod<1> == '1' && A:I:F == '000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction behaves as if imod<1> == '0'.
- The instruction behaves as if A:I:F has an UNKNOWN nonzero value.

If `imod<1> == '0' && A:I:F != '000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction behaves as if imod<1> == '1'.
- The instruction behaves as if A:I:F == '000'.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	1	0	0	1	1	im	(0)	A	I	F

CPSID variant

Applies when `im == 1`.

```
CPSID{<q>} <iflags> // Not permitted in IT block
```

CPSIE variant

Applies when `im == 0`.

```
CPSIE{<q>} <iflags> // Not permitted in IT block
```

Decode for all variants of this encoding

```
if A:I:F == '000' then UNPREDICTABLE;
enable = (im == '0');  disable = (im == '1');  changemode = FALSE;
affectA = (A == '1');  affectI = (I == '1');  affectF = (F == '1');
if InITBlock() then UNPREDICTABLE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CONSTRAINED UNPREDICTABLE behavior

If `A:I:F == '000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	imod		M	A	I	F	mode	

CPS variant

Applies when `imod == 00 && M == 1`.

```
CPS{<q>} #<mode> // Not permitted in IT block
```

CPSID variant

Applies when `imod == 11 && M == 0`.

```
CPSID.W <iflags> // Not permitted in IT block
```

CPSID variant

Applies when `imod == 11 && M == 1`.

```
CPSID{<q>} <iflags>, #<mode> // Not permitted in IT block
```

CPSIE variant

Applies when `imod == 10 && M == 0`.

```
CPSIE.W <iflags> // Not permitted in IT block
```

CPSIE variant

Applies when `imod == 10 && M == 1`.

```
CPSIE{<q>} <iflags>, #<mode> // Not permitted in IT block
```

Decode for all variants of this encoding

```
if imod == '00' && M == '0' then SEE "Hint instructions";
if mode != '00000' && M == '0' then UNPREDICTABLE;
if (imod<1> == '1' && A:I:F == '000') || (imod<1> == '0' && A:I:F != '000') then UNPREDICTABLE;
enable = (imod == '10');  disable = (imod == '11');  changemode = (M == '1');
affectA = (A == '1');  affectI = (I == '1');  affectF = (F == '1');
if imod == '01' || InITBlock() then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `imod == '01'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

If `mode != '00000' && M == '0'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The instruction executes as NOP.
- The instruction executes with the additional decode: changemode = TRUE.
- The instruction executes as described, and the value specified by mode is ignored. There are no additional side-effects.

If `imod<1> == '1' && A:I:F == '000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction behaves as if imod<1> == '0'.
- The instruction behaves as if A:I:F has an UNKNOWN nonzero value.

If `imod<1> == '0' && A:I:F != '000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction behaves as if imod<1> == '1'.
- The instruction behaves as if A:I:F == '000'.

Notes for all encodings

Hint instructions: In encoding T2, if the `imod` field is `00` and the `M` bit is `0`, a hint instruction is encoded. To determine which hint instruction, see *Branches and miscellaneous control* on page F3-4154.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<iflags>` Is a sequence of one or more of the following, specifying which interrupt mask bits are affected:

- `a` Sets the A bit in the instruction, causing the specified effect on PSTATE.A, the SError interrupt mask bit.
- `i` Sets the I bit in the instruction, causing the specified effect on PSTATE.I, the IRQ interrupt mask bit.
- `f` Sets the F bit in the instruction, causing the specified effect on PSTATE.F, the FIQ interrupt mask bit.

`<mode>` Is the number of the mode to change to, in the range 0 to 31, encoded in the "mode" field.

Operation for all encodings

```
if CurrentInstrSet() == InstrSet_A32 then
    EncodingSpecificOperations();
    if PSTATE.EL != EL0 then
        if enable then
            if affectA then PSTATE.A = '0';
            if affectI then PSTATE.I = '0';
            if affectF then PSTATE.F = '0';
        if disable then
            if affectA then PSTATE.A = '1';
            if affectI then PSTATE.I = '1';
            if affectF then PSTATE.F = '1';
        if changemode then
            // AArch32.WriteModeByInstr() sets PSTATE.IL to 1 if this is an illegal mode change.
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            AArch32.WriteModeByInstr(mode);
else
    EncodingSpecificOperations();
    if PSTATE.EL != EL0 then
        if enable then
            if affectA then PSTATE.A = '0';
            if affectI then PSTATE.I = '0';
            if affectF then PSTATE.F = '0';
        if disable then
            if affectA then PSTATE.A = '1';
            if affectI then PSTATE.I = '1';
            if affectF then PSTATE.F = '1';
        if changemode then
            // AArch32.WriteModeByInstr() sets PSTATE.IL to 1 if this is an illegal mode change.
            AArch32.WriteModeByInstr(mode);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.39 CRC32

`CRC32` performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, or 32 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial `0x04C11DB7` is used for the CRC calculation.

In Armv8-A, this is an OPTIONAL instruction, and in Armv8.1 it is mandatory for all implementations to implement it.

Note

ID_ISAR5.CRC32 indicates whether this instruction is supported in the T32 and A32 instruction sets.

A1

31–28	27 26 25 24	23 22 21	20	19–16	15–12	11 10	9	8	7 6 5 4	3–0
!=1111	0 0 0 1 0	sz	0	Rn	Rd	(0)(0)	0	(0)	0 1 0 0	Rm
cond							C			

CRC32B variant

Applies when `sz == 00`.

```
CRC32B{<q>} <Rd>, <Rn>, <Rm>
```

CRC32H variant

Applies when `sz == 01`.

```
CRC32H{<q>} <Rd>, <Rn>, <Rm>
```

CRC32W variant

Applies when `sz == 10`.

```
CRC32W{<q>} <Rd>, <Rn>, <Rm>
```

Decode for all variants of this encoding

```
if ! HaveCRCExt() then UNDEFINED;
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
size = 8 << UInt(sz);
crc32c = (C == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if size == 64 then UNPREDICTABLE;
if cond != '1110' then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == 64`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: size = 32;.

If `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The instruction executes unconditionally.
- The instruction executes conditionally.

T1

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 14 13 12	11 8	7 6	5 4	3 0
1 1 1 1	1 0 1 0 1	1 0 0	Rn	1 1 1 1	Rd	1 0	sz	Rm
		C						

CRC32B variant

Applies when `sz == 00`.

```
CRC32B{<q>} <Rd>, <Rn>, <Rm>
```

CRC32H variant

Applies when `sz == 01`.

```
CRC32H{<q>} <Rd>, <Rn>, <Rm>
```

CRC32W variant

Applies when `sz == 10`.

```
CRC32W{<q>} <Rd>, <Rn>, <Rm>
```

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if ! HaveCRCExt() then UNDEFINED;
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
size = 8 << UInt(sz);
crc32c = (C == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if size == 64 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == 64`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: size = 32;.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<q>`	See *Standard assembler syntax fields* on page F2-4080. An `CRC32` instruction must be unconditional.
`<Rd>`	Is the general-purpose accumulator output register, encoded in the "Rd" field.
`<Rn>`	Is the general-purpose accumulator input register, encoded in the "Rn" field.
`<Rm>`	Is the general-purpose data source register, encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();

    acc = R[n];             // accumulator
    val = R[m]<size-1:0>;   // input value
    poly = (if crc32c then 0x1EDC6F41 else 0x04C11DB7)<31:0>;
    tempacc = BitReverse(acc):Zeros(size);
    tempval = BitReverse(val):Zeros(32);
    // Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
    R[d] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.40 CRC32C

`CRC32C` performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, or 32 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial `0x1EDC6F41` is used for the CRC calculation.

In Armv8-A, this is an OPTIONAL instruction, and in Armv8.1 it is mandatory for all implementations to implement it.

Note

ID_ISAR5.CRC32 indicates whether this instruction is supported in the T32 and A32 instruction sets.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	0	0	0	1	0	sz		0	Rn	Rd	(0)	(0)	1	(0)	0	1	0	0	Rm
cond													C						

CRC32CB variant

Applies when `sz == 00`.

```
CRC32CB{<q>} <Rd>, <Rn>, <Rm>
```

CRC32CH variant

Applies when `sz == 01`.

```
CRC32CH{<q>} <Rd>, <Rn>, <Rm>
```

CRC32CW variant

Applies when `sz == 10`.

```
CRC32CW{<q>} <Rd>, <Rn>, <Rm>
```

Decode for all variants of this encoding

```
if ! HaveCRCExt() then UNDEFINED;
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
size = 8 << UInt(sz);
crc32c = (C == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if size == 64 then UNPREDICTABLE;
if cond != '1110' then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == 64`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: size = 32;.

If `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The instruction executes unconditionally.
- The instruction executes conditionally.

T1

15 14 13 12	11 10 9 8	7	6 5	4	3 0	15 14 13 12	11 8	7 6	5 4	3 0
1 1 1 1	1 0 1 0	1	1 0	1	Rn	1 1 1 1	Rd	1 0	sz	Rm
				C						

CRC32CB variant

Applies when `sz == 00`.

```
CRC32CB{<q>} <Rd>, <Rn>, <Rm>
```

CRC32CH variant

Applies when `sz == 01`.

```
CRC32CH{<q>} <Rd>, <Rn>, <Rm>
```

CRC32CW variant

Applies when `sz == 10`.

```
CRC32CW{<q>} <Rd>, <Rn>, <Rm>
```

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if ! HaveCRCExt() then UNDEFINED;
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
size = 8 << UInt(sz);
crc32c = (C == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if size == 64 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == 64`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: size = 32;.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<q>`	See *Standard assembler syntax fields* on page F2-4080. An `CRC32C` instruction must be unconditional.
`<Rd>`	Is the general-purpose accumulator output register, encoded in the "Rd" field.
`<Rn>`	Is the general-purpose accumulator input register, encoded in the "Rn" field.
`<Rm>`	Is the general-purpose data source register, encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();

    acc = R[n];             // accumulator
    val = R[m]<size-1:0>;   // input value
    poly = (if crc32c then 0x1EDC6F41 else 0x04C11DB7)<31:0>;
    tempacc = BitReverse(acc):Zeros(size);
    tempval = BitReverse(val):Zeros(32);
    // Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
    R[d] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.41 CSDB

Consumption of Speculative Data Barrier is a memory barrier that controls speculative execution and data value prediction.

No instruction other than branch instructions and instructions that write to the PC appearing in program order after the CSDB can be speculatively executed using the results of any:

- Data value predictions of any instructions.
- PSTATE.{N,Z,C,V} predictions of any instructions other than conditional branch instructions and conditional instructions that write to the PC appearing in program order before the CSDB that have not been architecturally resolved.

Note

For purposes of the definition of CSDB, PSTATE.{N,Z,C,V} is not considered a data value. This definition permits:

- Control flow speculation before and after the CSDB.
- Speculative execution of conditional data processing instructions after the CSDB, unless they use the results of data value or PSTATE.{N,Z,C,V} predictions of instructions appearing in program order before the CSDB that have not been architecturally resolved.

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!=1111 (cond)	0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	1	0	1	0	0

A1 variant

```
CSDB{<c>}{<q>}
```

Decode for this encoding

```
if cond != '1110' then UNPREDICTABLE;      // CSDB must be encoded with AL condition
```

CONSTRAINED UNPREDICTABLE behavior

If `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	1	0	1	0	0

T1 variant

```
CSDB{<c>}.W
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes unconditionally.
- The instruction executes conditionally.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();

    ConsumptionOfSpeculativeDataBarrier();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.42 DBG

In Armv8, DBG executes as a NOP. Arm deprecates any use of the DBG instruction.

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3–0
!=1111	0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	1	1	1	1	option
cond																									

A1 variant

```
DBG{<c>}{<q>} #<option>
```

Decode for this encoding

```
// DBG executes as a NOP. The 'option' field is ignored
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	1	1	1	1	option

T1 variant

```
DBG{<c>}{<q>} #<option>
```

Decode for this encoding

```
// DBG executes as a NOP. The 'option' field is ignored
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

<option> Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "option" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.43 DCPS1

Debug Change PE State to EL1 allows the debugger to move the PE into EL1 from EL0 or to a specific mode at the current Exception level.

`DCPS1` is UNDEFINED if any of:

- The PE is in Non-debug state.
- EL2 is implemented, EL2 is implemented and enabled in the current Security state, and any of:
  - EL2 is using AArch32 and HCR.TGE is set to 1.
  - EL2 is using AArch64 and HCR_EL2.TGE is set to 1.

When the PE executes `DCPS1` at EL0, EL1 or EL3:

- If EL3 or EL1 is using AArch32, the PE enters SVC mode and LR_svc, SPSR_svc, DLR, and DSPSR become UNKNOWN. If `DCPS1` is executed in Monitor mode, SCR.NS is cleared to 0.
- If EL1 is using AArch64, the PE enters EL1 using AArch64, selects SP_EL1, and ELR_EL1, ESR_EL1, SPSR_EL1, DLR_EL0 and DSPSR_EL0 become UNKNOWN.

When the PE executes `DCPS1` at EL2 the PE does not change mode, and ELR_hyp, HSR, SPSR_hyp, DLR and DSPSR become UNKNOWN.

For more information on the operation of this instruction, see *DCPS<n>* on page H2-6954.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	1	1	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

T1 variant

`DCPS1`

Decode for this encoding

```
// No additional decoding required.
```

Operation

```
if !Halted() then UNDEFINED;

if EL2Enabled() && PSTATE.EL == EL0 then
    tge = if ELUsingAArch32(EL2) then HCR.TGE else HCR_EL2.TGE;
    if tge == '1' then UNDEFINED;

if PSTATE.EL != EL0 || ELUsingAArch32(EL1) then
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    if PSTATE.EL != EL2 then
        AArch32.WriteMode(M32_Svc);
        PSTATE.E = SCTLR.EE;
        if HavePANExt() && SCTLR.SPAN == '0' then PSTATE.PAN = '1';
        LR_svc = bits(32) UNKNOWN;
        SPSR_svc = bits(32) UNKNOWN;
    else
        PSTATE.E = HSCTLR.EE;
        ELR_hyp = bits(32) UNKNOWN;
        HSR = bits(32) UNKNOWN;
        SPSR_hyp = bits(32) UNKNOWN;

    DLR = bits(32) UNKNOWN;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    DSPSR = bits(32) UNKNOWN;
else                                    // Targeting EL1 using AArch64
    AArch64.MaybeZeroRegisterUppers();
    MaybeZeroSVEUppers(EL1);
    PSTATE.nRW = '0';
    PSTATE.SP = '1';
    PSTATE.EL = EL1;
    if HavePANExt() && SCTLR_EL1.SPAN == '0' then PSTATE.PAN = '1';
    if HaveUAOExt() then PSTATE.UAO = '0';

    ELR_EL1 = bits(64) UNKNOWN;
    ESR_EL1 = bits(32) UNKNOWN;
    SPSR_EL1 = bits(32) UNKNOWN;

    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(32) UNKNOWN;

    // SCTLR_EL1.IESB might be ignored in Debug state.
    if HaveIESB() && SCTLR_EL1.IESB == '1' &&  !ConstrainUnpredictableBool() then
        SynchronizeErrors();

UpdateEDSCRFields();                    // Update EDSCR PE state flags
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.44 DCPS2

Debug Change PE State to EL2 allows the debugger to move the PE into EL2 from a lower Exception level.

DCPS2 is UNDEFINED if any of:

- The PE is in Non-debug state.
- EL2 is not implemented.
- The PE is in Secure state and any of:
  - Secure EL2 is not implemented.
  - Secure EL2 is implemented and Secure EL2 is disabled.

When the PE executes DCPS2:

- If EL2 is using AArch32, the PE enters Hyp mode and ELR_hyp, HSR, SPSR_hyp, DLR and DSPSR become UNKNOWN.
- If EL2 is using AArch64, the PE enters EL2 using AArch64, selects SP_EL2, and ELR_EL2, ESR_EL2, SPSR_EL2, DLR_EL0 and DSPSR_EL0 become UNKNOWN.

For more information on the operation of this instruction, see *DCPS<n>* on page H2-6954.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	1	1	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

T1 variant

DCPS2

Decode for this encoding

```
if !HaveEL(EL2) then UNDEFINED;
```

Operation

```
if !Halted() || IsSecure() then UNDEFINED;

if ELUsingAArch32(EL2) then
    AArch32.WriteMode(M32_Hyp);
    PSTATE.E = HSCTLR.EE;

    ELR_hyp = bits(32) UNKNOWN;
    HSR = bits(32) UNKNOWN;
    SPSR_hyp = bits(32) UNKNOWN;

    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
else                                        // Targeting EL2 using AArch64
    AArch64.MaybeZeroRegisterUppers();
    MaybeZeroSVEUppers(EL2);
    PSTATE.nRW = '0';
    PSTATE.SP = '1';
    PSTATE.EL = EL2;
    if HavePANExt() && SCTLR_EL2.SPAN == '0' && HCR_EL2.E2H == '1' && HCR_EL2.TGE == '1' then
        PSTATE.PAN = '1';
    if HaveUAOExt() then PSTATE.UAO = '0';

    ELR_EL2 = bits(64) UNKNOWN;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    ESR_EL2 = bits(32) UNKNOWN;
    SPSR_EL2 = bits(32) UNKNOWN;

    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(32) UNKNOWN;

    // SCTLR_EL2.IESB might be ignored in Debug state.
    if HaveIESB() && SCTLR_EL2.IESB == '1' && !ConstrainUnpredictableBool();

UpdateEDSCRFields();                        // Update EDSCR PE state flags
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.45 DCPS3

Debug Change PE State to EL3 allows the debugger to move the PE into EL3 from a lower Exception level or to a specific mode at the current Exception level.

DCPS3 is UNDEFINED if any of:

- The PE is in Non-debug state.
- EL3 is not implemented.
- EDSCR.SDD is set to 1.

When the PE executes DCPS3:

- If EL3 is using AArch32, the PE enters Monitor mode and LR_mon, SPSR_mon, DLR and DSPSR become UNKNOWN. If DCPS3 is executed in Monitor mode, SCR.NS is cleared to 0.
- If EL3 is using AArch64, the PE enters EL3 using AArch64, selects SP_EL3, and ELR_EL3, ESR_EL3, SPSR_EL3, DLR_EL0 and DSPSR_EL0 become UNKNOWN.

For more information on the operation of this instruction, see *DCPS<n>* on page H2-6954.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	1	1	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

T1 variant

DCPS3

Decode for this encoding

```
if !HaveEL(EL3) then UNDEFINED;
```

Operation

```
if !Halted() || EDSCR.SDD == '1' then UNDEFINED;

if ELUsingAArch32(EL3) then
    from_secure = IsSecure();
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    if HavePANExt() then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTLR.SPAN == '0' then
            PSTATE.PAN = '1';
    PSTATE.E = SCTLR.EE;

    LR_mon = bits(32) UNKNOWN;
    SPSR_mon = bits(32) UNKNOWN;

    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
else                                        // Targeting EL3 using AArch64
    AArch64.MaybeZeroRegisterUppers();
    MaybeZeroSVEUppers(EL3);
    PSTATE.nRW = '0';
    PSTATE.SP = '1';
    PSTATE.EL = EL3;
    if HaveUAOExt() then PSTATE.UAO = '0';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    ELR_EL3 = bits(64) UNKNOWN;
    ESR_EL3 = bits(32) UNKNOWN;
    SPSR_EL3 = bits(32) UNKNOWN;

    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(32) UNKNOWN;

    sync_errors = HaveIESB() && SCTLR_EL3.IESB == '1';
    if HaveDoubleFaultExt() && SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' then
        sync_errors = TRUE;
    // SCTLR_EL3.IESB might be ignored in Debug state.
    if !ConstrainUnpredictableBool() then
        sync_errors = FALSE;
    if sync_errors then SynchronizeErrors();

UpdateEDSCRFields();                        // Update EDSCR PE state flags
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.46 DMB

Data Memory Barrier is a memory barrier that ensures the ordering of observations of memory accesses, see *Data Memory Barrier (DMB)* on page E2-3992.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	0
1	1	1	1	0	1	0	1	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	1	0	1	option	

A1 variant

```
DMB{<c>}{<q>} {<option>}
```

Decode for this encoding

```
// No additional decoding required
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	1	0	1	option	

T1 variant

```
DMB{<c>}{<q>} {<option>}
```

Decode for this encoding

```
// No additional decoding required
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. Must be `AL` or omitted.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<option>` Specifies an optional limitation on the barrier operation. Values are:

- `SY` Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Can be omitted. This option is referred to as the full system barrier. Encoded as option = `0b1111`.
- `ST` Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. `SYST` is a synonym for `ST`. Encoded as option = `0b1110`.
- `LD` Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = `0b1101`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`ISH` Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as option = `0b1011`.

`ISHST` Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as option = `0b1010`.

`ISHLD` Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = `0b1001`.

`NSH` Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as option = `0b0111`.

`NSHST` Non-shareable is the required shareability domain, writes are the required access type both before and after the barrier instruction. Encoded as option = `0b0110`.

`NSHLD` Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = `0b0101`.

`OSH` Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as option = `0b0011`.

`OSHST` Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as option = `0b0010`.

`OSHLD` Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = `0b0001`.

For more information on whether an access is before or after a barrier instruction, see *Data Memory Barrier (DMB)* on page E2-3992. All other encodings of option are reserved. All unsupported and reserved options must execute as a full system DMB operation, but software must not rely on this behavior.

Note

The instruction supports the following alternative `<option>` values, but Arm recommends that software does not use these alternative values:

- `SH` as an alias for `ISH`.
- `SHST` as an alias for `ISHST`.
- `UN` as an alias for `NSH`.
- `UNST` as an alias for `NSHST`.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    case option of
        when '0001'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_Reads;
        when '0010'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_Writes;
        when '0011'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_All;
        when '0101'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_Reads;
        when '0110'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_Writes;
        when '0111'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_All;
        when '1001'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_Reads;
        when '1010'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_Writes;
        when '1011'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_All;
        when '1101'  domain = MBReqDomain_FullSystem;      types = MBReqTypes_Reads;
        when '1110'  domain = MBReqDomain_FullSystem;      types = MBReqTypes_Writes;
        otherwise    domain = MBReqDomain_FullSystem;      types = MBReqTypes_All;

    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        if HCR.BSU == '11' then
            domain = MBReqDomain_FullSystem;
        if HCR.BSU == '10' && domain != MBReqDomain_FullSystem then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        domain = MBReqDomain_OuterShareable;
    if HCR.BSU == '01' && domain == MBReqDomain_Nonshareable then
        domain = MBReqDomain_InnerShareable;

DataMemoryBarrier(domain, types);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.47 DSB

Data Synchronization Barrier is a memory barrier that ensures the completion of memory accesses, see *Data Synchronization Barrier (DSB)* on page E2-3993.

A1

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 … 0
1 1 1 1	0 1 0 1	0 1 1 1	(1)(1)(1)(1)	(1)(1)(1)(1)	(0)(0)(0)(0)	0 1 0 0	!=0x00 option

A1 variant

```
DSB{<c>}{<q>} {<option>}
```

Decode for this encoding

```
// No additional decoding required
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 … 0
1 1 1 1	0 0 1 1	1 0 1 1	(1)(1)(1)(1)	1 0 (0) 0	(1)(1)(1)(1)	0 1 0 0	!=0x00 option

T1 variant

```
DSB{<c>}{<q>} {<option>}
```

Decode for this encoding

```
// No additional decoding required
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. Must be `AL` or omitted.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<option>` Specifies an optional limitation on the barrier operation. Values are:

- `SY` Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Can be omitted. This option is referred to as the full system barrier. Encoded as option = `0b1111`.
- `ST` Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. `SYST` is a synonym for `ST`. Encoded as option = `0b1110`.
- `LD` Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = `0b1101`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`ISH` Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as option = `0b1011`.

`ISHST` Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as option = `0b1010`.

`ISHLD` Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = `0b1001`.

`NSH` Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as option = `0b0111`.

`NSHST` Non-shareable is the required shareability domain, writes are the required access type both before and after the barrier instruction. Encoded as option = `0b0110`.

`NSHLD` Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = `0b0101`.

`OSH` Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as option = `0b0011`.

`OSHST` Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as option = `0b0010`.

`OSHLD` Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = `0b0001`.

For more information on whether an access is before or after a barrier instruction, see *Data Synchronization Barrier (DSB)* on page E2-3993. All other encodings of option are reserved, other than the values `0b0000` and `0b0100`. All unsupported and reserved options must execute as a full system DSB operation, but software must not rely on this behavior.

Note

The value `0b0000` is used to encode SSBB and the value `0b0100` is used to encode PSSBB.

The instruction supports the following alternative `<option>` values, but Arm recommends that software does not use these alternative values:

- `SH` as an alias for `ISH`.
- `SHST` as an alias for `ISHST`.
- `UN` as an alias for `NSH`.
- `UNST` as an alias for `NSHST`.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    case option of
        when '0001'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_Reads;
        when '0010'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_Writes;
        when '0011'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_All;
        when '0101'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_Reads;
        when '0110'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_Writes;
        when '0111'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_All;
        when '1001'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_Reads;
        when '1010'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_Writes;
        when '1011'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_All;
        when '1101'  domain = MBReqDomain_FullSystem;      types = MBReqTypes_Reads;
        when '1110'  domain = MBReqDomain_FullSystem;      types = MBReqTypes_Writes;
        otherwise
            if       option == '0000' then SEE "SSBB";
            elsif    option == '0100' then SEE "PSSBB";
            else     domain = MBReqDomain_FullSystem;      types = MBReqTypes_All;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
    if HCR.BSU == '11' then
        domain = MBReqDomain_FullSystem;
    if HCR.BSU == '10' && domain != MBReqDomain_FullSystem then
        domain = MBReqDomain_OuterShareable;
    if HCR.BSU == '01' && domain == MBReqDomain_Nonshareable then
        domain = MBReqDomain_InnerShareable;

DataSynchronizationBarrier(domain, types);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.48 EOR, EORS (immediate)

Bitwise Exclusive OR (immediate) performs a bitwise Exclusive OR of a register value and an immediate value, and writes the result to the destination register.

If the destination register is not the PC, the EORS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The EOR variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- The EORS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–0
!=1111 (cond)	0 0 1 0	0 0 1 S	Rn	Rd	imm12

EOR variant

Applies when `S == 0`.

```
EOR{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

EORS variant

Applies when `S == 1`.

```
EORS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = A32ExpandImm_C(imm12, PSTATE.C);
```

T1

15 14 13 12	11	10	9	8	7 6 5 4	3–0	15	14–12	11–8	7–0
1 1 1 1 0	i	0	0	1	0 0 S	Rn	0	imm3	Rd	imm8

EOR variant

Applies when `S == 0`.

```
EOR{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

EORS variant

Applies when `S == 1 && Rd != 1111`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
EORS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then SEE "TEQ (immediate)";
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
if (d == 15 && !setflags) || n == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the EOR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- For the EORS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`.

`<Rn>` For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

`<const>` For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* on page F2-4096 for the range of values.

For encoding T1: an immediate value. See *Modified immediate constants in T32 instructions* on page F2-4095 for the range of values.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] EOR imm32;
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.49 EOR, EORS (register)

Bitwise Exclusive OR (register) performs a bitwise Exclusive OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the EORS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The EOR variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- The EORS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–7	6 5	4	3–0
!=1111	0 0 0 0	0 0 0 1	S	Rn	Rd	imm5	stype	0	Rm
cond								

EOR, rotate right with extend variant

Applies when `S == 0 && imm5 == 00000 && stype == 11`.

```
EOR{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

EOR, shift or rotate by value variant

Applies when `S == 0 && !(imm5 == 00000 && stype == 11)`.

```
EOR{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

EORS, rotate right with extend variant

Applies when `S == 1 && imm5 == 00000 && stype == 11`.

```
EORS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

EORS, shift or rotate by value variant

Applies when `S == 1 && !(imm5 == 00000 && stype == 11)`.

```
EORS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15 14 13 12	11 10 9 8	7 6 5	3 2 0
0 1 0 0 0 0	0 0 0 1	Rm	Rdn

T1 variant

```
EOR<c>{<q>} {<Rdn>,} <Rdn>, <Rm> // Inside IT block
EORS{<q>} {<Rdn>,} <Rdn>, <Rm> // Outside IT block
```

Decode for this encoding

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 14 12	11 8	7 6 5 4	3 0
1 1 1 0 1 0 1	0 1 0 0	S	Rn	(0) imm3	Rd	imm2 stype	Rm

EOR, rotate right with extend variant

Applies when `S == 0 && imm3 == 000 && imm2 == 00 && stype == 11`.

```
EOR{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

EOR, shift or rotate by value variant

Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11)`.

```
EOR<c>.W {<Rd>,} <Rn>, <Rm> // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
EOR{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

EORS, rotate right with extend variant

Applies when `S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11`.

```
EORS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

EORS, shift or rotate by value variant

Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11) && Rd != 1111`.

```
EORS.W {<Rd>,} <Rn>, <Rm> // Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
EORS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then SEE "TEQ (register)";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if (d == 15 && !setflags) || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rdn>` Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the EOR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- For the EORS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`.

`<Rn>` For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T2: is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

`LSL`	when `stype = 00`
`LSR`	when `stype = 01`
`ASR`	when `stype = 10`
`ROR`	when `stype = 11`

`<amount>` For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

In T32 assembly:

- Outside an IT block, if `EORS <Rd>, <Rn>, <Rd>` has `<Rd>` and `<Rn>` both in the range R0-R7, it is assembled using encoding T1 as though `EORS <Rd>, <Rn>` had been written
- Inside an IT block, if `EOR<c> <Rd>, <Rn>, <Rd>` has `<Rd>` and `<Rn>` both in the range R0-R7, it is assembled using encoding T1 as though `EOR<c> <Rd>, <Rn>` had been written.

To prevent either of these happening, use the `.W` qualifier.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] EOR shifted;
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
else
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.50 EOR, EORS (register-shifted register)

Bitwise Exclusive OR (register-shifted register) performs a bitwise Exclusive OR of a register value and a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.

A1

31–28	27–24	23–20		19–16	15–12	11–8	7	6–5	4	3–0
!=1111 (cond)	0 0 0 0	0 0 1	S	Rn	Rd	Rs	0	stype	1	Rm

Flag setting variant

Applies when `S == 1`.

```
EORS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

Not flag setting variant

Applies when `S == 0`.

```
EOR{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

- `LSL` when `stype = 00`
- `LSR` when `stype = 01`
- `ASR` when `stype = 10`
- `ROR` when `stype = 11`

`<Rs>` Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] EOR shifted;
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.51 ERET

Exception Return.

The PE branches to the address held in the register holding the preferred return address, and restores PSTATE from SPSR_<current_mode>.

The register holding the preferred return address is:

- ELR_hyp, when executing in Hyp mode.
- LR, when executing in a mode other than Hyp mode, User mode, or System mode.

The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.

Exception Return is CONSTRAINED UNPREDICTABLE in User mode and System mode.

In Debug state, the T1 encoding of ERET executes the DRPS operation.

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!=1111	0	0	0	1	0	1	1	0	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	0	1	1	0	(1)	(1)	(1)	(0)
cond																												

A1 variant

```
ERET{<c>}{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	1	0	1	1	1	1	0	1	0	(0)	0	(1)	(1)	(1)	(1)	0	0	0	0	0	0	0	0

T1 variant

```
ERET{<c>}{<q>}
```

Decode for this encoding

```
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Halted() then
        if PSTATE.M IN {M32_User,M32_System} then
            UNPREDICTABLE;                        // UNDEFINED or NOP
        else
            new_pc_value = if PSTATE.EL == EL2 then ELR_hyp else R[14];
            AArch32.ExceptionReturn(new_pc_value, SPSR[]);
    else                                          // Perform DRPS operation in Debug state
        if PSTATE.M == M32_User then
            UNDEFINED;
        elsif PSTATE.M == M32_System then
            UNPREDICTABLE;                        // UNDEFINED or NOP
        else
            SynchronizeContext();
            SetPSTATEFromPSR(SPSR[]);
            // PSTATE.{N,Z,C,V,Q,GE,SS,A,I,F} are not observable and ignored in Debug state, so
            // behave as if UNKNOWN.
            PSTATE.<N,Z,C,V,Q,GE,SS,A,I,F> = bits(13) UNKNOWN;
            // In AArch32 Debug state, all instructions are T32 and unconditional.
            PSTATE.IT = '00000000';  PSTATE.T = '1';        // PSTATE.J is RES0
            DLR = bits(32) UNKNOWN;  DSPSR = bits(32) UNKNOWN;
            UpdateEDSCRFields();                  // Update EDSCR PE state flags
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.M IN {M32_User,M32_System}`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.52 ESB

Error Synchronization Barrier is an error synchronization event that might also update DISR and VDISR. This instruction can be used at all Exception levels and in Debug state.

In Debug state, this instruction behaves as if SError interrupts are masked at all Exception levels. See Error Synchronization Barrier in the ARM® Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for ARMv8-A architecture profile.

If the RAS Extension is not implemented, this instruction executes as a `NOP`.

A1

ARMv8.2

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!=1111	0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	1	0	0	0	0
cond																												

A1 variant

```
ESB{<c>}{<q>}
```

Decode for this encoding

```
if !HaveRASExt() then EndOfInstruction();  // Instruction executes as NOP
if cond != '1110' then UNPREDICTABLE;      // ESB must be encoded with AL condition
```

CONSTRAINED UNPREDICTABLE behavior

If `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1

ARMv8.2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	1	0	0	0	0

T1 variant

```
ESB{<c>}.W
```

Decode for this encoding

```
if !HaveRASExt() then EndOfInstruction();  // Instruction executes as NOP
if InITBlock() then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();

    SynchronizeErrors();
    AArch32.ESBOperation();
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch32.vESBOperation();
    TakeUnmaskedSErrorInterrupts();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.53 HLT

Halting breakpoint causes a software breakpoint to occur.

Halting breakpoint is always unconditional, even inside an IT block.

A1

31–28	27 26 25 24	23 22 21 20	19–8	7 6 5 4	3–0
!=1111 (cond)	0 0 0 1 0	0 0 0	imm12	0 1 1 1	imm4

A1 variant

```
HLT{<q>} {#}<imm>
```

Decode for this encoding

```
if EDSCR.HDE == '0' || !HaltingAllowed() then UNDEFINED;
if cond != '1110' then UNPREDICTABLE; // HLT must be encoded with AL condition
```

CONSTRAINED UNPREDICTABLE behavior

If `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1

15 14 13 12	11 10 9 8	7 6 5	0
1 0 1 1	1 0 1 0	1 0	imm6

T1 variant

```
HLT{<q>} {#}<imm>
```

Decode for this encoding

```
if EDSCR.HDE == '0' || !HaltingAllowed() then UNDEFINED;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<q>` See *Standard assembler syntax fields* on page F2-4080. An `HLT` instruction must be unconditional.

`<imm>` For encoding A1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm12:imm4" field. This value is for assembly and disassembly only. It is ignored by the PE, but can be used by a debugger to store more information about the halting breakpoint.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For encoding T1: is a 6-bit unsigned immediate, in the range 0 to 63, encoded in the "imm6" field. This value is for assembly and disassembly only. It is ignored by the PE, but can be used by a debugger to store more information about the halting breakpoint.

Operation for all encodings

```
EncodingSpecificOperations();
Halt(DebugHalt_HaltInstruction);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.54 HVC

Hypervisor Call causes a Hypervisor Call exception. For more information see *Hypervisor Call (HVC) exception* on page G1-5744. Non-secure software executing at EL1 can use this instruction to call the hypervisor to request a service.

The `HVC` instruction is:

- UNDEFINED in Secure state, and in User mode in Non-secure state.
- When SCR.HCE is set to 0, UNDEFINED in Non-secure EL1 modes and CONSTRAINED UNPREDICTABLE in Hyp mode.

On executing an `HVC` instruction, the HSR reports the exception as a Hypervisor Call exception, using the EC value `0x12`, and captures the value of the immediate argument, see *Use of the HSR* on page G5-6038.

A1

31–28	27	26	25	24	23	22	21	20	19–8	7	6	5	4	3–0
!=1111	0	0	0	1	0	1	0	0	imm12	0	1	1	1	imm4
cond														

A1 variant

```
HVC{<q>} {#}<imm16>
```

Decode for this encoding

```
if cond != '1110' then UNPREDICTABLE;
imm16 = imm12:imm4;
```

CONSTRAINED UNPREDICTABLE behavior

If `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15	14	13	12	11–0
1	1	1	1	0	1	1	1	1	1	1	0	imm4	1	0	0	0	imm12

T1 variant

```
HVC{<q>} {#}<imm16>
```

Decode for this encoding

```
imm16 = imm4:imm12;
if InITBlock() then UNPREDICTABLE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<q>` See *Standard assembler syntax fields* on page F2-4080. An `HVC` instruction must be unconditional.

`<imm16>` For encoding A1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm12:imm4" field. This value is for assembly and disassembly only. It is reported in the HSR but otherwise is ignored by hardware. An HVC handler might interpret imm16, for example to determine the required service.

For encoding T1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:imm12" field. This value is for assembly and disassembly only. It is reported in the HSR but otherwise is ignored by hardware. An HVC handler might interpret imm16, for example to determine the required service.

Operation for all encodings

```
EncodingSpecificOperations();
if !HaveEL(EL2) || PSTATE.EL == EL0 || (IsSecure() && !IsSecureEL2Enabled()) then
    UNDEFINED;

if HaveEL(EL3) then
    if ELUsingAArch32(EL3) && SCR.HCE == '0' && PSTATE.EL == EL2 then
        UNPREDICTABLE;
    else
        hvc_enable = SCR_GEN[].HCE;
else
    hvc_enable = if ELUsingAArch32(EL2) then NOT(HCR.HCD) else NOT(HCR_EL2.HCD);

if hvc_enable == '0' then
    UNDEFINED;
else
    AArch32.CallHypervisor(imm16);
```

CONSTRAINED UNPREDICTABLE behavior

If `ELUsingAArch32(EL3) && SCR.HCE == '0' && PSTATE.EL == EL2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.55 ISB

Instruction Synchronization Barrier flushes the pipeline in the PE and is a context synchronization event. For more information, see *Instruction Synchronization Barrier (ISB)* on page E2-3992.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3 – 0
1	1	1	1	0	1	0	1	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	1	1	0	option

A1 variant

```
ISB{<c>}{<q>} {<option>}
```

Decode for this encoding

```
// No additional decoding required
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3 – 0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	1	1	0	option

T1 variant

```
ISB{<c>}{<q>} {<option>}
```

Decode for this encoding

```
// No additional decoding required
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. Must be `AL` or omitted.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<option>` Specifies an optional limitation on the barrier operation. Values are:

`SY` Full system barrier operation, encoded as option = `0b1111`. Can be omitted.

All other encodings of option are reserved. The corresponding instructions execute as full system barrier operations, but must not be relied upon by software.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    InstructionSynchronizationBarrier();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.56 IT

If-Then makes up to four following instructions (the IT block) conditional. The conditions for the instructions in the IT block are the same as, or the inverse of, the condition the `IT` instruction specifies for the first instruction in the block.

The `IT` instruction itself does not affect the condition flags, but the execution of the instructions in the IT block can change the condition flags.

16-bit instructions in the IT block, other than `CMP`, `CMN` and `TST`, do not set the condition flags. An `IT` instruction with the `AL` condition can change the behavior without conditional execution.

The architecture permits exception return to an instruction in the IT block only if the restoration of the CPSR restores ITSTATE to a state consistent with the conditions specified by the `IT` instruction. Any other exception return to an instruction in an IT block is UNPREDICTABLE. Any branch to a target instruction in an IT block is not permitted, and if such a branch is made it is UNPREDICTABLE what condition is used when executing that target instruction and any subsequent instruction in the IT block.

Many uses of the IT instruction are deprecated for performance reasons, and an implementation might include ITD controls that can disable those uses of IT, making them UNDEFINED.

For more information see *Conditional execution* on page F2-4081 and *Conditional instructions* on page F1-4037. The first of these sections includes more information about the ITD controls.

T1

15	14	13	12	11	10	9	8	7 – 4	3 – 0
1	0	1	1	1	1	1	1	firstcond	!=0000
									mask

T1 variant

```
IT{<x>{<y>{<z>}}}{<q>} <cond>
```

Decode for this encoding

```
if mask == '0000' then SEE "Related encodings";
if firstcond == '1111' || (firstcond == '1110' && BitCount(mask) != 1) then UNPREDICTABLE;
if InITBlock() then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `firstcond == '1111' || (firstcond == '1110' && BitCount(mask) != 1)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The '1111' condition is treated as being the same as the '1110' condition, meaning always, and the ITSTATE state machine is progressed in the same way as for any other cond_base value.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Related encodings: *Miscellaneous 16-bit instructions* on page F3-4115.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

<x> The condition for the second instruction in the IT block. If omitted, the "mask" field is set to `0b1000`. If present it is encoded in the "mask[3]" field:

- `T` firstcond[0]
- `E` NOT firstcond[0]

<y> The condition for the third instruction in the IT block. If omitted and <x> is present, the "mask[2:0]" field is set to `0b100`. If <y> is present it is encoded in the "mask[2]" field:

- `T` firstcond[0]
- `E` NOT firstcond[0]

<z> The condition for the fourth instruction in the IT block. If omitted and <y> is present, the "mask[1:0]" field is set to `0b10`. If <z> is present, the "mask[0]" field is set to 1, and it is encoded in the "mask[1]" field:

- `T` firstcond[0]
- `E` NOT firstcond[0]

<q> See *Standard assembler syntax fields* on page F2-4080.

<cond> The condition for the first instruction in the IT block, encoded in the "firstcond" field. See Table F2-1 on page F2-4081 for the range of conditions available, and the encodings.

The conditions specified in an `IT` instruction must match those specified in the syntax of the instructions in its IT block. When assembling to A32 code, assemblers check `IT` instruction syntax for validity but do not generate assembled instructions for them. See *Conditional instructions* on page F1-4037.

Operation

```
EncodingSpecificOperations();
AArch32.CheckITEnabled(mask);
PSTATE.IT<7:0> = firstcond:mask;
ShouldAdvanceIT = FALSE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.57 LDA

Load-Acquire Word loads a word from memory and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page E2-3996

For more information about support for shared memory see *Synchronization and semaphores* on page E2-4023. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3	2	1	0
!=1111	0	0	0	1	1	0	0	1	Rn	Rt	(1)	(1)	0	0	1	0	0	1	(1)	(1)	(1)	(1)
cond																						

A1 variant

```
LDA{<c>}{<q>} <Rt>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	1	Rn	Rt	(1)	(1)	(1)	(1)	1	0	1	0	(1)	(1)	(1)	(1)

T1 variant

```
LDA{<c>}{<q>} <Rt>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    R[t] = MemO[address, 4];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.58 LDAB

Load-Acquire Byte loads a byte from memory, zero-extends it to form a 32-bit word and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page E2-3996.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-4023. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3	2	1	0
!=1111	0	0	0	1	1	1	0	1	Rn	Rt	(1)	(1)	0	0	1	0	0	1	(1)	(1)	(1)	(1)
cond																						

A1 variant

```
LDAB{<c>}{<q>} <Rt>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	1	Rn	Rt	(1)	(1)	(1)	(1)	1	0	0	0	(1)	(1)	(1)	(1)

T1 variant

```
LDAB{<c>}{<q>} <Rt>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    R[t] = ZeroExtend(MemO[address, 1], 32);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.59 LDAEX

Load-Acquire Exclusive Word loads a word from memory, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page E2-3996.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-4023. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11 10	9	8	7 6 5 4	3 2 1 0
!=1111 (cond)	0 0 0 1	1 0 0 1	Rn	Rt	(1) (1)	1	0	1 0 0 1	(1) (1) (1) (1)

A1 variant

```
LDAEX{<c>}{<q>} <Rt>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15–12	11 10 9 8	7	6	5 4	3 2 1 0
1 1 1 0	1 0 0 0	1 1 0 1	Rn	Rt	(1) (1) (1) (1)	1	1	1 0	(1) (1) (1) (1)

T1 variant

```
LDAEX{<c>}{<q>} <Rt>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    AArch32.SetExclusiveMonitors(address, 4);
    R[t] = MemO[address, 4];
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.60 LDAEXB

Load-Acquire Exclusive Byte loads a byte from memory, zero-extends it to form a 32-bit word, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page E2-3996.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-4023. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3	2	1	0
!=1111	0	0	0	1	1	1	0	1	Rn	Rt	(1)	(1)	1	0	1	0	0	1	(1)	(1)	(1)	(1)
cond																						

A1 variant

```
LDAEXB{<c>}{<q>} <Rt>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	1	Rn	Rt	(1)	(1)	(1)	(1)	1	1	0	0	(1)	(1)	(1)	(1)

T1 variant

```
LDAEXB{<c>}{<q>} <Rt>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    AArch32.SetExclusiveMonitors(address, 1);
    R[t] = ZeroExtend(MemO[address, 1], 32);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.61 LDAEXD

Load-Acquire Exclusive Doubleword loads a doubleword from memory, writes it to two registers and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also acts as a barrier instruction with the ordering requirements described in *Load-Acquire, Store-Release* on page E2-3996.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-4023. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3	2	1	0
!=1111	0	0	0	1	1	0	1	1	Rn	Rt	(1)	(1)	1	0	1	0	0	1	(1)	(1)	(1)	(1)
cond																						

A1 variant

```
LDAEXD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt);  t2 = t + 1;  n = UInt(Rn);
if Rt<0> == '1' || t2 == 15 || n == 15 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `Rt<0> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: t<0> = '0'.
- The instruction executes with the additional decode: t2 = t.
- The instruction executes as described, with no change to its behavior and no additional side effects.

If `Rt == '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction is handled as described in *Using R15* on page K1-7831.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11–8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	1	Rn	Rt	Rt2	1	1	1	1	(1)	(1)	(1)	(1)

T1 variant

```
LDAEXD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);
if t == 15 || t2 == 15 || t == t2 || n == 15 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The load instruction executes but the destination register takes an UNKNOWN value.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rt>`	For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. `<Rt>` must be even-numbered and not R14. For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.
`<Rt2>`	For encoding A1: is the second general-purpose register to be transferred. `<Rt2>` must be `<R(t+1)>`. For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.
`<Rn>`	Is the general-purpose base register, encoded in the "Rn" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    AArch32.SetExclusiveMonitors(address, 8);
    value = MemO[address, 8];
    // Extract words from 64-bit loaded value such that R[t] is
    // loaded from address and R[t2] from address+4.
    R[t]  = if BigEndian() then value<63:32> else value<31:0>;
    R[t2] = if BigEndian() then value<31:0>  else value<63:32>;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.62 LDAEXH

Load-Acquire Exclusive Halfword loads a halfword from memory, zero-extends it to form a 32-bit word, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page E2-3996.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-4023. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3	2	1	0
!=1111	0	0	0	1	1	1	1	1	Rn	Rt	(1)	(1)	1	0	1	0	0	1	(1)	(1)	(1)	(1)
cond																						

A1 variant

```
LDAEXH{<c>}{<q>} <Rt>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	1	Rn	Rt	(1)	(1)	(1)	(1)	1	1	0	1	(1)	(1)	(1)	(1)

T1 variant

```
LDAEXH{<c>}{<q>} <Rt>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rt>`	Is the general-purpose register to be transferred, encoded in the "Rt" field.
`<Rn>`	Is the general-purpose base register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    AArch32.SetExclusiveMonitors(address, 2);
    R[t] = ZeroExtend(MemO[address, 2], 32);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.63 LDAH

Load-Acquire Halfword loads a halfword from memory, zero-extends it to form a 32-bit word and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page E2-3996.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-4023. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3	2	1	0
!=1111	0	0	0	1	1	1	1	1	Rn	Rt	(1)	(1)	0	0	1	0	0	1	(1)	(1)	(1)	(1)
cond																						

A1 variant

```
LDAH{<c>}{<q>} <Rt>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	1	Rn	Rt	(1)	(1)	(1)	(1)	1	0	0	1	(1)	(1)	(1)	(1)

T1 variant

```
LDAH{<c>}{<q>} <Rt>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rt>`	Is the general-purpose register to be transferred, encoded in the "Rt" field.
`<Rn>`	Is the general-purpose base register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    R[t] = ZeroExtend(MemO[address, 2], 32);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.64 LDC (immediate)

Load data to System register (immediate) calculates an address from a base register value and an immediate offset, loads a word from memory, and writes it to the DBGDTRTXint System register. It can use offset, post-indexed, pre-indexed, or unindexed addressing. For information about memory accesses see *Memory accesses* on page F2-4085.

In an implementation that includes EL2, the permitted `LDC` access to DBGDTRTXint can be trapped to Hyp mode, meaning that an attempt to execute an `LDC` instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see *Trapping general Non-secure System register accesses to debug registers* on page G1-5803.

For simplicity, the `LDC` pseudocode does not show this possible trap to Hyp mode.

A1

31–28	27 26 25	24	23	22	21	20	19–16	15 14 13 12	11 10 9	8	7–0
!=1111	1 1 0	P	U	0	W	1	!=1111	0 1 0 1	1 1 1	0	imm8
cond							Rn				

Offset variant

Applies when `P == 1 && W == 0`.

```
LDC{<c>}{<q>} p14, c5, [<Rn>{, #{+/-}<imm>}]
```

Post-indexed variant

Applies when `P == 0 && W == 1`.

```
LDC{<c>}{<q>} p14, c5, [<Rn>], #{+/-}<imm>
```

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
LDC{<c>}{<q>} p14, c5, [<Rn>, #{+/-}<imm>]!
```

Unindexed variant

Applies when `P == 0 && U == 1 && W == 0`.

```
LDC{<c>}{<q>} p14, c5, [<Rn>], <option>
```

Decode for all variants of this encoding

```
if Rn == '1111' then SEE "LDC (literal)";
if P == '0' && U == '0' && W == '0' then UNDEFINED;
n = UInt(Rn);  cp = 14;
imm32 = ZeroExtend(imm8:'00', 32);  index = (P == '1');  add = (U == '1');  wback = (W == '1');
```

T1

15 14 13 12	11 10 9	8	7	6	5	4	3–0	15 14 13 12	11 10 9	8	7–0
1 1 1 0	1 1 0	P	U	0	W	1	!=1111	0 1 0 1	1 1 1	0	imm8
							Rn				

Offset variant

Applies when `P == 1 && W == 0`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
LDC{<c>}{<q>} p14, c5, [<Rn>{, #{+/-}<imm>}]
```

Post-indexed variant

Applies when `P == 0 && W == 1`.

```
LDC{<c>}{<q>} p14, c5, [<Rn>], #{+/-}<imm>
```

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
LDC{<c>}{<q>} p14, c5, [<Rn>, #{+/-}<imm>]!
```

Unindexed variant

Applies when `P == 0 && U == 1 && W == 0`.

```
LDC{<c>}{<q>} p14, c5, [<Rn>], <option>
```

Decode for all variants of this encoding

```
if Rn == '1111' then SEE "LDC (literal)";
if P == '0' && U == '0' && W == '0' then UNDEFINED;
n = UInt(Rn);  cp = 14;
imm32 = ZeroExtend(imm8:'00', 32);  index = (P == '1');  add = (U == '1');  wback = (W == '1');
```

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field. If the PC is used, see LDC (literal).

`<option>` Is an 8-bit immediate, in the range 0 to 255 enclosed in { }, encoded in the "imm8" field. The value of this field is ignored when executing this instruction.

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`<imm>` Is the immediate offset used for forming the address, a multiple of 4 in the range 0-1020, defaulting to 0 and encoded in the "imm8" field, as <imm>/4.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];

    // System register write to DBGDTRTXint.
    DBGDTR_EL0[] = MemA[address,4];

    if wback then R[n] = offset_addr;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.65 LDC (literal)

Load data to System register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to the DBGDTRTXint System register. For information about memory accesses see *Memory accesses* on page F2-4085.

In an implementation that includes EL2, the permitted LDC access to DBGDTRTXint can be trapped to Hyp mode, meaning that an attempt to execute an LDC instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see *Trapping general Non-secure System register accesses to debug registers* on page G1-5803.

For simplicity, the LDC pseudocode does not show this possible trap to Hyp mode.

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7–0
!=1111	1	1	0	P	U	0	W	1	1	1	1	1	0	1	0	1	1	1	1	0	imm8
cond																					

A1 variant

Applies when `!(P == 0 && U == 0 && W == 0)`.

```
LDC{<c>}{<q>} p14, c5, <label>
LDC{<c>}{<q>} p14, c5, [PC, #{+/-}<imm>]
LDC{<c>}{<q>} p14, c5, [PC], <option>
```

Decode for this encoding

```
if P == '0' && U == '0' && W == '0' then UNDEFINED;
index = (P == '1');  add = (U == '1');  cp = 14;  imm32 = ZeroExtend(imm8:'00', 32);
if W == '1' || (P == '0' && CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `W == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7–0
1	1	1	0	1	1	0	P	U	0	W	1	1	1	1	1	0	1	0	1	1	1	1	0	imm8

T1 variant

Applies when `!(P == 0 && U == 0 && W == 0)`.

```
LDC{<c>}{<q>} p14, c5, <label>
LDC{<c>}{<q>} p14, c5, [PC, #{+/-}<imm>]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
if P == '0' && U == '0' && W == '0' then UNDEFINED;
index = (P == '1');  add = (U == '1');  cp = 14;  imm32 = ZeroExtend(imm8:'00', 32);
if W == '1' || (P == '0' && CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `W == '1' || P == '0'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction executes as `LDC` (immediate) with writeback to the PC. The instruction is handled as described in *Using R15* on page K1-7831.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<option>` Is an 8-bit immediate, in the range 0 to 255 enclosed in { }, encoded in the "imm8" field. The value of this field is ignored when executing this instruction.

`<label>` The label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are multiples of 4 in the range -1020 to 1020.

If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE` (encoded as U == 1).

If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE (encoded as U == 0)`.

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`<imm>` Is the immediate offset used for forming the address, a multiple of 4 in the range 0-1020, defaulting to 0 and encoded in the "imm8" field, as <imm>/4.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page F1-4037.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
    address = if index then offset_addr else Align(PC,4);

    // System register write to DBGDTRTXint.
    DBGDTR_EL0[] = MemA[address,4];
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.66 LDM, LDMIA, LDMFD

Load Multiple (Increment After, Full Descending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations start at this address, and the address just above the highest of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC* on page F2-4086.

Armv8.2 permits the deprecation of some Load Multiple ordering behaviors in AArch32 state, for more information see ARMv8.2-LSMAOC. The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953. Related system instructions are LDM (User registers) and LDM (exception return).

This instruction is used by the alias POP (multiple registers). See *Alias conditions* on page F5-4400 for details of when each alias is preferred.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–0
!=1111 (cond)	1	0	0	0	1	0	W	1	Rn	register_list

A1 variant

```
LDM{IA}{<c>}{<q>} <Rn>{!}, <registers> // Preferred syntax
LDMFD{<c>}{<q>} <Rn>{!}, <registers> // Alternate syntax, Full Descending stack
```

Decode for this encoding

```
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15 14 13 12	11	10 8	7 0
1 1 0 0	1	Rn	register_list

T1 variant

```
LDM{IA}{<c>}{<q>} <Rn>{!}, <registers> // Preferred syntax
LDMFD{<c>}{<q>} <Rn>{!}, <registers> // Alternate syntax, Full Descending stack
```

Decode for this encoding

```
n = UInt(Rn);  registers = '00000000':register_list;  wback = (registers<n> == '0');
if BitCount(registers) < 1 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

T2

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15	14	13 0
1 1 1 0	1 0 0	0	1	0	W	1	Rn	P	M	register_list

T2 variant

```
LDM{IA}{<c>}.W <Rn>{!}, <registers> // Preferred syntax, if <Rn>, '!' and <registers> can be represented
in T1
LDMFD{<c>}.W <Rn>{!}, <registers> // Alternate syntax, Full Descending stack, if <Rn>, '!' and
<registers> can be represented in T1
LDM{IA}{<c>}{<q>} <Rn>{!}, <registers> // Preferred syntax
LDMFD{<c>}{<q>} <Rn>{!}, <registers> // Alternate syntax, Full Descending stack
```

Decode for this encoding

```
n = UInt(Rn);  registers = P:M:register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 2 || (P == '1' && M == '1') then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
if registers<13> == '1' then UNPREDICTABLE;
if registers<15> == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If `BitCount(registers) == 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction loads a single register using the specified addressing modes.
- The instruction executes as `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

If `registers<13> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode, but R13 is UNKNOWN.

If `P == '1' && M == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction loads the register list and either R14 or R15, both R14 and R15, or neither of these registers.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Alias conditions

Alias	of variant	is preferred when
POP (multiple registers)	T2	`W == '1' && Rn == '1101' && BitCount(P:M:register_list) > 1`
POP (multiple registers)	A1	`W == '1' && Rn == '1101' && BitCount(register_list) > 1`

Assembler symbols

`IA`	Is an optional suffix for the Increment After form.
`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rn>`	Is the general-purpose base register, encoded in the "Rn" field.
`!`	For encoding A1 and T2: the address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For encoding T1: the address adjusted by the size of the data loaded is written back to the base register. It is omitted if `<Rn>` is included in `<registers>`, otherwise it must be present.

`<registers>` For encoding A1: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }.

The PC can be in the list.

Arm deprecates using these instructions with both the LR and the PC in the list.

For encoding T1: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R7, encoded in the "register_list" field.

For encoding T2: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain one of the LR or the PC. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0. If the PC is in the list, the "P" field is set to 1, otherwise it defaults to 0.

If the PC is in the list:

- The LR must not be in the list.
- The instruction must be either outside any IT block, or the last instruction in an IT block.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemA[address,4];  address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemA[address,4]);
    if wback && registers<n> == '0' then R[n] = R[n] + 4*BitCount(registers);
    if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.67 LDM (exception return)

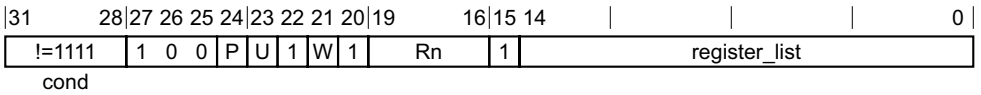
Load Multiple (exception return) loads multiple registers from consecutive memory locations using an address from a base register. The SPSR of the current mode is copied to the CPSR. An address adjusted by the size of the data loaded can optionally be written back to the base register.

The registers loaded include the PC. The word loaded for the PC is treated as an address and a branch occurs to that address.

Load Multiple (exception return) is:

- UNDEFINED in Hyp mode.
- UNPREDICTABLE in debug state, and in User mode and System mode.

A1



A1 variant

```
LDM{<amode>}{<c>}{<q>} <Rn>{!}, <registers_with_pc>^
```

Decode for this encoding

```
n = UInt(Rn);  registers = register_list;
wback = (W == '1');  increment = (U == '1');  wordhigher = (P == U);
if n == 15 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all the loads using the specified addressing mode and the content of the register being written back is UNKNOWN. In addition, if an exception occurs during the execution of this instruction, the base address might be corrupted so that the instruction cannot be repeated.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<amode>`	is one of:	
	`DA`	Decrement After. The consecutive memory addresses end at the address in the base register. Encoded as P = 0, U = 0.
	`FA`	Full Ascending. For this instruction, a synonym for `DA`.
	`DB`	Decrement Before. The consecutive memory addresses end one word below the address in the base register. Encoded as P = 1, U = 0.
	`EA`	Empty Ascending. For this instruction, a synonym for `DB`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

IA Increment After. The consecutive memory addresses start at the address in the base register. This is the default. Encoded as P = 0, U = 1.

FD Full Descending. For this instruction, a synonym for IA.

IB Increment Before. The consecutive memory addresses start one word above the address in the base register. Encoded as P = 1, U = 1.

ED Empty Descending. For this instruction, a synonym for IB.

<c> See *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

! The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

<registers_with_pc> Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be loaded. The registers are loaded with the lowest-numbered register from the lowest memory address, through to the highest-numbered register from the highest memory address. The PC must be specified in the register list, and the instruction causes a branch to the address (data) loaded into the PC. See also *Encoding of lists of general-purpose registers and the PC* on page F2-4086.

Instructions with similar syntax but without the PC included in the registers list are described in LDM (User registers).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then
        UNDEFINED;
    elsif PSTATE.M IN {M32_User,M32_System} then
        UNPREDICTABLE;                        // UNDEFINED or NOP
    else
        length = 4*BitCount(registers) + 4;
        address = if increment then R[n] else R[n]-length;
        if wordhigher then address = address+4;

        for i = 0 to 14
            if registers<i> == '1' then
                R[i] = MemA[address,4];  address = address + 4;
        new_pc_value = MemA[address,4];

        if wback && registers<n> == '0' then R[n] = if increment then R[n]+length else R[n]-length;
        if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;

        AArch32.ExceptionReturn(new_pc_value, SPSR[]);
```

CONSTRAINED UNPREDICTABLE behavior

If PSTATE.M IN {M32_User,M32_System}, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.68 LDM (User registers)

In an EL1 mode other than System mode, Load Multiple (User registers) loads multiple User mode registers from consecutive memory locations using an address from a base register. The registers loaded cannot include the PC. The PE reads the base register value normally, using the current mode to determine the correct Banked version of the register. This instruction cannot writeback to the base register.

Load Multiple (User registers) is UNDEFINED in Hyp mode, and UNPREDICTABLE in User and System modes.

Armv8.2 permits the deprecation of some Load Multiple ordering behaviors in AArch32 state, for more information see ARMv8.2-LSMAOC.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15	14–0
!=1111	1	0	0	P	U	1	(0)	1	Rn	0	register_list
cond											

A1 variant

```
LDM{<amode>}{<c>}{<q>} <Rn>, <registers_without_pc>^
```

Decode for this encoding

```
n = UInt(Rn);  registers = register_list;  increment = (U == '1');  wordhigher = (P == U);
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as an `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<amode>` is one of:

`DA`	Decrement After. The consecutive memory addresses end at the address in the base register. Encoded as P = 0, U = 0.
`FA`	Full Ascending. For this instruction, a synonym for `DA`.
`DB`	Decrement Before. The consecutive memory addresses end one word below the address in the base register. Encoded as P = 1, U = 0.
`EA`	Empty Ascending. For this instruction, a synonym for `DB`.
`IA`	Increment After. The consecutive memory addresses start at the address in the base register. This is the default. Encoded as P = 0, U = 1.
`FD`	Full Descending. For this instruction, a synonym for `IA`.
`IB`	Increment Before. The consecutive memory addresses start one word above the address in the base register. Encoded as P = 1, U = 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ED Empty Descending. For this instruction, a synonym for IB.

<c> See *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<registers_without_pc>

Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be loaded by the LDM instruction. The registers are loaded with the lowest-numbered register from the lowest memory address, through to the highest-numbered register from the highest memory address. The PC must not be in the register list. See also *Encoding of lists of general-purpose registers and the PC* on page F2-4086.

Instructions with similar syntax but with the PC included in <registers_without_pc> are described in LDM (exception return).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UNDEFINED;
    elsif PSTATE.M IN {M32_User,M32_System} then UNPREDICTABLE;
    else
        length = 4*BitCount(registers);
        address = if increment then R[n] else R[n]-length;
        if wordhigher then address = address+4;
        for i = 0 to 14
            if registers<i> == '1' then  // Load User mode register
                Rmode[i, M32_User] = MemA[address,4];  address = address + 4;
```

CONSTRAINED UNPREDICTABLE behavior

If PSTATE.M IN {M32_User,M32_System}, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.69 LDMDA, LDMFA

Load Multiple Decrement After (Full Ascending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations end at this address, and the address just below the lowest of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC* on page F2-4086.

Armv8.2 permits the deprecation of some Load Multiple ordering behaviors in AArch32 state, for more information see ARMv8.2-LSMAOC. The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953. Related system instructions are LDM (User registers) and LDM (exception return).

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–0
!=1111 (cond)	1	0	0	0	0	0	W	1	Rn	register_list

A1 variant

```
LDMDA{<c>}{<q>} <Rn>{!}, <registers> // Preferred syntax
LDMFA{<c>}{<q>} <Rn>{!}, <registers> // Alternate syntax, Full Ascending stack
```

Decode for this encoding

```
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as an `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`!` The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

`<registers>` Is a list of one or more registers to be loaded, separated by commas and surrounded by { and }.

The PC can be in the list.

Arm deprecates using these instructions with both the LR and the PC in the list.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] - 4*BitCount(registers) + 4;
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemA[address,4];  address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemA[address,4]);
    if wback && registers<n> == '0' then R[n] = R[n] - 4*BitCount(registers);
    if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.70 LDMDB, LDMEA

Load Multiple Decrement Before (Empty Ascending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations end just below this address, and the address of the lowest of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC* on page F2-4086.

Armv8.2 permits the deprecation of some Load Multiple ordering behaviors in AArch32 state, for more information see ARMv8.2-LSMAOC. The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953. Related system instructions are LDM (User registers) and LDM (exception return).

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–0
!=1111	1	0	0	1	0	0	W	1	Rn	register_list
cond										

A1 variant

```
LDMDB{<c>}{<q>} <Rn>{!}, <registers> // Preferred syntax
LDMEA{<c>}{<q>} <Rn>{!}, <registers> // Alternate syntax, Empty Ascending stack
```

Decode for this encoding

```
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15	14	13–0
1	1	1	0	1	0	0	1	0	0	W	1	Rn	P	M	register_list

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1 variant

```
LDMDB{<c>}{<q>} <Rn>{!}, <registers> // Preferred syntax
LDMEA{<c>}{<q>} <Rn>{!}, <registers> // Alternate syntax, Empty Ascending stack
```

Decode for this encoding

```
n = UInt(Rn);  registers = P:M:register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 2 || (P == '1' && M == '1') then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
if registers<13> == '1' then UNPREDICTABLE;
if registers<15> == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

If `BitCount(registers) == 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction loads a single register using the specified addressing modes.
- The instruction executes as `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

If `registers<13> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode, but R13 is UNKNOWN.

If `P == '1' && M == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction loads the register list and either R14 or R15, both R14 and R15, or neither of these registers.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`!` The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

`<registers>` For encoding A1: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }.

The PC can be in the list.

Arm deprecates using these instructions with both the LR and the PC in the list.

For encoding T1: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain one of the LR or the PC. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0. If the PC is in the list, the "P" field is set to 1, otherwise it defaults to 0.

If the PC is in the list:

- The LR must not be in the list.
- The instruction must be either outside any IT block, or the last instruction in an IT block.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] - 4*BitCount(registers);
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemA[address,4];  address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemA[address,4]);
    if wback && registers<n> == '0' then R[n] = R[n] - 4*BitCount(registers);
    if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.71 LDMIB, LDMED

Load Multiple Increment Before (Empty Descending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations start just above this address, and the address of the last of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC* on page F2-4086.

Armv8.2 permits the deprecation of some Load Multiple ordering behaviors in AArch32 state, for more information see ARMv8.2-LSMAOC. The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953. Related system instructions are LDM (User registers) and LDM (exception return).

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–0
!=1111	1	0	0	1	1	0	W	1	Rn	register_list
cond										

A1 variant

```
LDMIB{<c>}{<q>} <Rn>{!}, <registers> // Preferred syntax
LDMED{<c>}{<q>} <Rn>{!}, <registers> // Alternate syntax, Empty Descending stack
```

Decode for this encoding

```
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as an `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`!` The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

`<registers>` Is a list of one or more registers to be loaded, separated by commas and surrounded by { and }.

The PC can be in the list.

Arm deprecates using these instructions with both the LR and the PC in the list.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + 4;
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemA[address,4];  address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemA[address,4]);
    if wback && registers<n> == '0' then R[n] = R[n] + 4*BitCount(registers);
    if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.72 LDR (immediate)

Load Register (immediate) calculates an address from a base register value and an immediate offset, loads a word from memory, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page F2-4085.

This instruction is used by the alias POP (single register). See *Alias conditions* on page F5-4415 for details of when each alias is preferred.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
!=1111	0	1	0	P	U	0	W	1	!=1111	Rt	imm12
cond									Rn		

Offset variant

Applies when `P == 1 && W == 0`.

```
LDR{<c>}{<q>} <Rt>, [<Rn> {, #{+/-}<imm>}]
```

Post-indexed variant

Applies when `P == 0 && W == 0`.

```
LDR{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
LDR{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if Rn == '1111' then SEE "LDR (literal)";
if P == '0' && W == '1' then SEE "LDRT";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if wback && n == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

15	14	13	12	11	10–6	5–3	2–0
0	1	1	0	1	imm5	Rn	Rt

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1 variant

```
LDR{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

T2

15 14 13 12	11	10 8	7 0
1 0 0 1	1	Rt	imm8

T2 variant

```
LDR{<c>}{<q>} <Rt>, [SP{, #{+}<imm>}]
```

Decode for this encoding

```
t = UInt(Rt);  n = 13;  imm32 = ZeroExtend(imm8:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

T3

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 0
1 1 1 1	1 0 0 0 1	1 0 1	!=1111	Rt	imm12
			Rn		

T3 variant

```
LDR{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}] // <Rt>, <Rn>, <imm> can be represented in T1 or T2
LDR{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then SEE "LDR (literal)";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32); index = TRUE;  add = TRUE;
wback = FALSE; if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

T4

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11	10	9	8	7 0
1 1 1 1	1 0 0 0 0	1 0 1	!=1111	Rt	1	P	U	W	imm8
			Rn						

Offset variant

Applies when `P == 1 && U == 0 && W == 0`.

```
LDR{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

Post-indexed variant

Applies when `P == 0 && W == 1`.

```
LDR{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
LDR{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if Rn == '1111' then SEE "LDR (literal)";
if P == '1' && U == '1' && W == '0' then SEE "LDRT";
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);
imm32 = ZeroExtend(imm8, 32); index = (P == '1');  add = (U == '1');  wback = (W == '1');
if (wback && n == t) || (t == 15 && InITBlock() && !LastInITBlock()) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Alias conditions

Alias	of variant	is preferred when
POP (single register)	A1 (post-indexed)	`P == '0' && U == '1' && W == '0' && Rn == '1101' && imm12 == '000000000100'`
POP (single register)	T4 (post-indexed)	`Rn == '1101' && U == '1' && imm8 == '00000100'`

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rt>` For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.

For encoding T1 and T2: is the general-purpose register to be transferred, encoded in the "Rt" field.

For encoding T3 and T4: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.

`<Rn>` For encoding A1, T3 and T4: is the general-purpose base register, encoded in the "Rn" field. For PC use see LDR (literal).

For encoding T1: is the general-purpose base register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

+/-
: Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

  -
  : when `U = 0`

  +
  : when `U = 1`

+
: Specifies the offset is added to the base register.

<imm>
: For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

  For encoding T1: is the optional positive unsigned immediate byte offset, a multiple of 4, in the range 0 to 124, defaulting to 0 and encoded in the "imm5" field as <imm>/4.

  For encoding T2: is the optional positive unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 and encoded in the "imm8" field as <imm>/4.

  For encoding T3: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

  For encoding T4: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

Operation for all encodings

```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        data = MemU[address,4];
        if wback then R[n] = offset_addr;
        if t == 15 then
            if address<1:0> == '00' then
                LoadWritePC(data);
            else
                UNPREDICTABLE;
        else
            R[t] = data;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        data = MemU[address,4];
        if wback then R[n] = offset_addr;
        if t == 15 then
            if address<1:0> == '00' then
                LoadWritePC(data);
            else
                UNPREDICTABLE;
        else
            R[t] = data;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.73 LDR (literal)

Load Register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11–0
!=1111 (cond)	0	1	0	P	U	0	W	1	1	1	1	1	Rt	imm12

A1 variant

Applies when `!(P == 0 && W == 1)`.

```
LDR{<c>}{<q>} <Rt>, <label> // Normal form
LDR{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // Alternative form
```

Decode for this encoding

```
if P == '0' && W == '1' then SEE "LDRT";
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);
add = (U == '1');  wback = (P == '0') || (W == '1');
if wback then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: wback = FALSE;.
- The instruction treats bit[24] as the P bit, and bit[21] as the writeback (W) bit, and uses the same addressing mode as described in LDR (immediate). The instruction uses post-indexed addressing when P == '0' and uses pre-indexed addressing otherwise. The instruction is handled as described in *Using R15* on page K1-7831.

T1

15	14	13	12	11	10–8	7–0
0	1	0	0	1	Rt	imm8

T1 variant

```
LDR{<c>}{<q>} <Rt>, <label> // Normal form
```

Decode for this encoding

```
t = UInt(Rt);  imm32 = ZeroExtend(imm8:'00', 32);  add = TRUE;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15–12	11–0
1	1	1	1	1	0	0	0	U	1	0	1	1	1	1	1	Rt	imm12

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T2 variant

```
LDR{<c>}.W <Rt>, <label> // Preferred syntax, and <Rt>, <label> can be represented in T1
LDR{<c>}{<q>} <Rt>, <label> // Preferred syntax
LDR{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // Alternative syntax
```

Decode for this encoding

```
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rt>` For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.

For encoding T1: is the general-purpose register to be transferred, encoded in the "Rt" field.

For encoding T2: is the general-purpose register to be transferred, encoded in the "Rt" field. The SP can be used. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.

`<label>` For encoding A1 and T2: the label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are -4095 to 4095.

If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`, encoded as U == 1.

If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`, encoded as U == 0.

For encoding T1: the label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are Multiples of four in the range 0 to 1020.

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`<imm>` For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

For encoding T2: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page F1-4037.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    data = MemU[address,4];
    if t == 15 then
        if address<1:0> == '00' then
            LoadWritePC(data);
        else
            UNPREDICTABLE;
    else
        R[t] = data;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.74 LDR (register)

Load Register (register) calculates an address from a base register value and an offset register value, loads a word from memory, and writes it to a register. The offset register value can optionally be shifted. For information about memory accesses, see *Memory accesses* on page F2-4085.

The T32 form of LDR (register) does not support register writeback.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–7	6–5	4	3–0
!=1111	0	1	1	P	U	0	W	1	Rn	Rt	imm5	stype	0	Rm
cond														

Offset variant

Applies when `P == 1 && W == 0`.

```
LDR{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]
```

Post-indexed variant

Applies when `P == 0 && W == 0`.

```
LDR{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
LDR{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]!
```

Decode for all variants of this encoding

```
if P == '0' && W == '1' then SEE "LDRT";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
if m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

15	14	13	12	11	10	9	8–6	5–3	2–0
0	1	0	1	1	0	0	Rm	Rn	Rt

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1 variant

```
LDR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 10 9 8	7 6 5 4	3 0
1 1 1 1	1 0 0 0 0	1 0 1	!=1111	Rt	0 0 0 0	0 0 imm2	Rm
			Rn				

T2 variant

```
LDR{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // <Rt>, <Rn>, <Rm> can be represented in T1
LDR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then SEE "LDR (literal)";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rt>` For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used. If the PC is used, the instruction branches to the address (data) loaded to the PC. This branch is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.

For encoding T1: is the general-purpose register to be transferred, encoded in the "Rt" field.

For encoding T2: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.

`<Rn>` For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.

For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.

`+/-` Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`+` Specifies the index register is added to the base register.

`<Rm>` Is the general-purpose index register, encoded in the "Rm" field.

`<shift>` The shift to apply to the value read from `<Rm>`. If absent, no shift is applied. Otherwise, see *Shifts applied to a register* on page F2-4083.

`<imm>` If present, the size of the left shift to apply to the value from `<Rm>`, in the range 1-3. `<imm>` is encoded in imm2. If absent, no shift is specified and imm2 is encoded as `0b00`.

Operation for all encodings

```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
        offset_addr = if add then (R[n] + offset) else (R[n] - offset);
        address = if index then offset_addr else R[n];
        data = MemU[address,4];
        if wback then R[n] = offset_addr;
        if t == 15 then
            if address<1:0> == '00' then
                LoadWritePC(data);
            else
                UNPREDICTABLE;
        else
            R[t] = data;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
        offset_addr = (R[n] + offset);
        address = offset_addr;
        data = MemU[address,4];
        if t == 15 then
            if address<1:0> == '00' then
                LoadWritePC(data);
            else
                UNPREDICTABLE;
        else
            R[t] = data;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.75 LDRB (immediate)

Load Register Byte (immediate) calculates an address from a base register value and an immediate offset, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27 26 25	24	23	22	21	20	19–16	15–12	11–0
!=1111	0 1 0	P	U	1	W	1	!=1111	Rt	imm12
cond							Rn		

Offset variant

Applies when `P == 1 && W == 0`.

```
LDRB{<c>}{<q>} <Rt>, [<Rn> {, #{+/-}<imm>}]
```

Post-indexed variant

Applies when `P == 0 && W == 0`.

```
LDRB{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
LDRB{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if Rn == '1111' then SEE "LDRB (literal)";
if P == '0' && W == '1' then SEE "LDRBT";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

15 14 13 12	11	10–6	5–3	2–0
0 1 1 1	1	imm5	Rn	Rt

T1 variant

```
LDRB{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11 … 0
1	1	1	1	1	0	0	0	1	0	0	1	!=1111 (Rn)	!=1111 (Rt)	imm12

T2 variant

```
LDRB{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}] // <Rt>, <Rn>, <imm> can be represented in T1
LDRB{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
if Rt == '1111' then SEE "PLD";
if Rn == '1111' then SEE "LDRB (literal)";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
```

T3

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7 … 0
1	1	1	1	1	0	0	0	0	0	0	1	!=1111 (Rn)	Rt	1	P	U	W	imm8

Offset variant

Applies when `Rt != 1111 && P == 1 && U == 0 && W == 0`.

```
LDRB{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

Post-indexed variant

Applies when `P == 0 && W == 1`.

```
LDRB{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
LDRB{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "PLD, PLDW (immediate)";
if Rn == '1111' then SEE "LDRB (literal)";
if P == '1' && U == '1' && W == '0' then SEE "LDRBT";
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if  (t == 15 &&  W == '1') || (wback && n == t) then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.

`<Rn>` For encoding A1, T2 and T3: is the general-purpose base register, encoded in the "Rn" field. For PC use see LDRB (literal).

For encoding T1: is the general-purpose base register, encoded in the "Rn" field.

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

`-` when `U = 0`

`+` when `U = 1`

`+` Specifies the offset is added to the base register.

`<imm>` For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

For encoding T1: is an optional 5-bit unsigned immediate byte offset, in the range 0 to 31, defaulting to 0 and encoded in the "imm5" field.

For encoding T2: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T3: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

Operation for all encodings

```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        R[t] = ZeroExtend(MemU[address,1], 32);
        if wback then R[n] = offset_addr;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        R[t] = ZeroExtend(MemU[address,1], 32);
        if wback then R[n] = offset_addr;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.76 LDRB (literal)

Load Register Byte (literal) calculates an address from the PC value and an immediate offset, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11–0
!=1111 (cond)	0	1	0	P	U	1	W	1	1	1	1	1	Rt	imm12

A1 variant

Applies when `!(P == 0 && W == 1)`.

```
LDRB{<c>}{<q>} <Rt>, <label> // Normal form
LDRB{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // Alternative form
```

Decode for this encoding

```
if P == '0' && W == '1' then SEE "LDRBT";
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);
add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || wback then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: wback = FALSE;.
- The instruction treats bit[24] as the P bit, and bit[21] as the writeback (W) bit, and uses the same addressing mode as described in LDRB (immediate). The instruction uses post-indexed addressing when P == '0' and uses pre-indexed addressing otherwise. The instruction is handled as described in *Using R15* on page K1-7831.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15–12	11–0
1	1	1	1	1	0	0	0	U	0	0	1	1	1	1	1	!=1111 (Rt)	imm12

T1 variant

```
LDRB{<c>}{<q>} <Rt>, <label> // Preferred syntax
LDRB{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // Alternative syntax
```

Decode for this encoding

```
if Rt == '1111' then SEE "PLD";
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
// Armv8-A removes UNPREDICTABLE for R13
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.

`<label>` The label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are -4095 to 4095.

If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`, encoded as U == 1.

If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`, encoded as U == 0.

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

`-` when `U = 0`

`+` when `U = 1`

`<imm>` For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

For encoding T1: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page F1-4037.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    R[t] = ZeroExtend(MemU[address,1], 32);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.77 LDRB (register)

Load Register Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. The offset register value can optionally be shifted. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27 26 25	24	23	22	21	20	19–16	15–12	11–7	6 5	4	3–0
!=1111 (cond)	0 1 1	P	U	1	W	1	Rn	Rt	imm5	stype	0	Rm

Offset variant

Applies when `P == 1 && W == 0`.

```
LDRB{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]
```

Post-indexed variant

Applies when `P == 0 && W == 0`.

```
LDRB{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
LDRB{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]!
```

Decode for all variants of this encoding

```
if P == '0' && W == '1' then SEE "LDRBT";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

15 14 13 12	11	10	9	8–6	5–3	2–0
0 1 0 1	1	1	0	Rm	Rn	Rt

T1 variant

```
LDRB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	1	1	0	0	0	0	0	0	1	!=1111 (Rn)	!=1111 (Rt)	0	0	0	0	0	0	imm2		Rm

T2 variant

```
LDRB{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // <Rt>, <Rn>, <Rm> can be represented in T1
LDRB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

Decode for this encoding

```
if Rt == '1111' then SEE "PLD";
if Rn == '1111' then SEE "LDRB (literal)";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.

`<Rn>` For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.

For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.

`+/-` Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`+` Specifies the index register is added to the base register.

`<Rm>` Is the general-purpose index register, encoded in the "Rm" field.

`<shift>` The shift to apply to the value read from `<Rm>`. If absent, no shift is applied. Otherwise, see *Shifts applied to a register* on page F2-4083.

`<imm>` If present, the size of the left shift to apply to the value from `<Rm>`, in the range 1-3. `<imm>` is encoded in imm2. If absent, no shift is specified and imm2 is encoded as `0b00`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    R[t] = ZeroExtend(MemU[address,1],32);
    if wback then R[n] = offset_addr;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.78 LDRBT

Load Register Byte Unprivileged loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses* on page F2-4085.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

`LDRBT` is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
!=1111	0	1	0	0	U	1	1	1	Rn	Rt	imm12
cond											

A1 variant

```
LDRBT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/-}<imm>}
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm12, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in *Using R15* on page K1-7831.
- The instruction uses immediate offset addressing with the base register as PC, without writeback.

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

A2

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–7	6–5	4	3–0
!=1111	0	1	1	0	U	1	1	1	Rn	Rt	imm5	stype	0	Rm
cond														

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A2 variant

```
LDRBT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;  (shift_t, shift_n) = DecodeImmShift(stype, imm5);
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7 … 0
1	1	1	1	1	0	0	0	0	0	0	1	!=1111 (Rn)	Rt	1	1	1	0	imm8

T1 variant

```
LDRBT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then SEE "LDRB (literal)";
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rt>` For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.

For encoding A2 and T1: is the general-purpose register to be transferred, encoded in the "Rt" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`+/-` For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- when U = 0

+ when U = 1

<Rm> Is the general-purpose index register, encoded in the "Rm" field.

<shift> The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see *Shifts applied to a register* on page F2-4083.

+ Specifies the offset is added to the base register.

<imm> For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

Operation for all encodings

```
if ConditionPassed() then
    if PSTATE.EL == EL2 then UNPREDICTABLE;               // Hyp mode
    EncodingSpecificOperations();
    offset = if register_form then Shift(R[m], shift_t, shift_n, PSTATE.C) else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    R[t] = ZeroExtend(MemU_unpriv[address,1],32);
    if postindex then R[n] = offset_addr;
```

CONSTRAINED UNPREDICTABLE behavior

If PSTATE.EL == EL2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDRB (immediate).

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.79 LDRD (immediate)

Load Register Dual (immediate) calculates an address from a base register value and an immediate offset, loads two words from memory, and writes them to two registers. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
!=1111	0	0	0	P	U	1	W	0	!=1111	Rt	imm4H	1	1	0	1	imm4L
cond									Rn							

Offset variant

Applies when `P == 1 && W == 0`.

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn> {, #{+/-}<imm>}]
```

Post-indexed variant

Applies when `P == 0 && W == 0`.

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], #{+/-}<imm>
```

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if Rn == '1111' then SEE "LDRD (literal)";
if Rt<0> == '1' then UNPREDICTABLE;
t = UInt(Rt);  t2 = t+1;  n = UInt(Rn);  imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if P == '0' && W == '1' then UNPREDICTABLE;
if wback && (n == t || n == t2) then UNPREDICTABLE;
if t2 == 15 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && (n == t || n == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If `P == '0' && W == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as an `LDRD` using one of offset, post-indexed, or pre-indexed addressing.

If `Rt<0> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The instruction executes with the additional decode: t<0> = '0'.
- The instruction executes with the additional decode: t2 = t.
- The instruction executes as described, with no change to its behavior and no additional side-effects. This does not apply when Rt == '1111'.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11 … 8	7 … 0
1	1	1	0	1	0	0	P	U	1	W	1	!=1111 (Rn)	Rt	Rt2	imm8

Offset variant

Applies when `P == 1 && W == 0`.

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn> {, #{+/-}<imm>}]
```

Post-indexed variant

Applies when `P == 0 && W == 1`.

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], #{+/-}<imm>
```

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if P == '0' && W == '0' then SEE "Related encodings";
if Rn == '1111' then SEE "LDRD (literal)";
t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if wback && (n == t || n == t2) then UNPREDICTABLE;
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && (n == t || n == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The load instruction executes but the destination register takes an UNKNOWN value.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Related encodings: *Load/store dual, load/store exclusive, load-acquire/store-release, and table branch* on page F3-4150.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rt>` For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14.

For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.

`<Rt2>` For encoding A1: is the second general-purpose register to be transferred. This register must be `<R(t+1)>`.

For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field. For PC use see LDRD (literal).

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

`-` when `U = 0`

`+` when `U = 1`

`<imm>` For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is the unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 if omitted, and encoded in the "imm8" field as <imm>/4.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    if address == Align(address, 8) then
        data = MemA[address,8];
        if BigEndian()  then
            R[t] = data<63:32>;
            R[t2] = data<31:0>;
        else
            R[t] = data<31:0>;
            R[t2] = data<63:32>;
    else
        R[t] = MemA[address,4];
        R[t2] = MemA[address+4,4];
    if wback then R[n] = offset_addr;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.80 LDRD (literal)

Load Register Dual (literal) calculates an address from the PC value and an immediate offset, loads two words from memory, and writes them to two registers. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11–8	7	6	5	4	3–0
!=1111	0	0	0	(1)	U	1	(0)	0	1	1	1	1	Rt	imm4H	1	1	0	1	imm4L
cond																			

A1 variant

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, <label> // Normal form
LDRD{<c>}{<q>} <Rt>, <Rt2>, [PC, #{+/-}<imm>] // Alternative form
```

Decode for this encoding

```
if Rt<0> == '1' then UNPREDICTABLE;
t = UInt(Rt);  t2 = t+1;  imm32 = ZeroExtend(imm4H:imm4L, 32);  add = (U == '1');
if t2 == 15 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `Rt<0> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: t<0> = '0';.
- The instruction executes with the additional decode: t2 = t;.
- The instruction executes as described, with no change to its behavior and no additional side-effects. This does not apply when Rt == '1111'.

If `P == '0' || W == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as if P == 1 and W == 0.'

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15–12	11–8	7–0
1	1	1	0	1	0	0	P	U	1	W	1	1	1	1	1	Rt	Rt2	imm8

T1 variant

Applies when `!(P == 0 && W == 0)`.

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, <label> // Normal form
LDRD{<c>}{<q>} <Rt>, <Rt2>, [PC, #{+/-}<imm>] // Alternative form
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
if P == '0' && W == '0' then SEE "Related encodings";
t = UInt(Rt);  t2 = UInt(Rt2);
imm32 = ZeroExtend(imm8:'00', 32);  add = (U == '1');
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if W == '1' then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The load instruction executes but the destination register takes an UNKNOWN value.

If `W == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction uses post-indexed addressing when P == '0' and uses pre-indexed addressing otherwise. The instruction is handled as described in *Using R15* on page K1-7831.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Related encodings: *Load/store dual, load/store exclusive, load-acquire/store-release, and table branch* on page F3-4150.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rt>` For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14.

For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.

`<Rt2>` For encoding A1: is the second general-purpose register to be transferred. This register must be `<R(t+1)>`.

For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.

`<label>` For encoding A1: the label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Any value in the range -255 to 255 is permitted.

If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`, encoded as U == 1. If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`, encoded as U == 0.

For encoding T1: the label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are multiples of 4 in the range -1020 to 1020.

If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`, encoded as U == 1.

If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`, encoded as U == 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`<imm>` For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is the optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page F1-4037.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
    if address == Align(address, 8) then
        data = MemA[address,8];
        if BigEndian() then
            R[t] = data<63:32>;
            R[t2] = data<31:0>;
        else
            R[t] = data<31:0>;
            R[t2] = data<63:32>;
    else
        R[t] = MemA[address,4];
        R[t2] = MemA[address+4,4];
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.81 LDRD (register)

Load Register Dual (register) calculates an address from a base register value and a register offset, loads two words from memory, and writes them to two registers. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	0	0	0	P	U	0	W	0	Rn	Rt	(0)	(0)	(0)	(0)	1	1	0	1	Rm
cond																			

Offset variant

Applies when `P == 1 && W == 0`.

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, {+/-}<Rm>]
```

Post-indexed variant

Applies when `P == 0 && W == 0`.

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], {+/-}<Rm>
```

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, {+/-}<Rm>]!
```

Decode for all variants of this encoding

```
if Rt<0> == '1' then UNPREDICTABLE;
t = UInt(Rt);  t2 = t+1;  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if P == '0' && W == '1' then UNPREDICTABLE;
if t2 == 15 || m == 15 || m == t || m == t2 then UNPREDICTABLE;
if wback && (n == 15 || n == t || n == t2) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && (n == t || n == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

If `P == '0' && W == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as an `LDRD` using one of offset, post-indexed, or pre-indexed addressing.

If `m == t || m == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The instruction loads register Rm with an UNKNOWN value.

If `Rt<0> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: t<0> = '0'.
- The instruction executes with the additional decode: t2 = t.
- The instruction executes as described, with no change to its behavior and no additional side-effects. This does not apply when Rt == '1111'.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rt>` Is the first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14.

`<Rt2>` Is the second general-purpose register to be transferred. This register must be `<R(t+1)>`.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.

`+/-` Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`<Rm>` Is the general-purpose index register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + R[m]) else (R[n] - R[m]);
    address = if index then offset_addr else R[n];
    if address == Align(address, 8) then
        data = MemA[address,8];
        if BigEndian() then
            R[t] = data<63:32>;
            R[t2] = data<31:0>;
        else
            R[t] = data<31:0>;
            R[t2] = data<63:32>;
    else
        R[t] = MemA[address,4];
        R[t2] = MemA[address+4,4];

    if wback then R[n] = offset_addr;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.82 LDREX

Load Register Exclusive calculates an address from a base register value and an immediate offset, loads a word from memory, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-4023. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3	2	1	0
!=1111	0	0	0	1	1	0	0	1	Rn	Rt	(1)	(1)	1	1	1	0	0	1	(1)	(1)	(1)	(1)
cond																						

A1 variant

```
LDREX{<c>}{<q>} <Rt>, [<Rn> {, {#}<imm>}]
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = Zeros(32); // Zero offset
if t == 15 || n == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7–0
1	1	1	0	1	0	0	0	0	1	0	1	Rn	Rt	(1)	(1)	(1)	(1)	imm8

T1 variant

```
LDREX{<c>}{<q>} <Rt>, [<Rn> {, #<imm>}]
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
if t == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rt>`	Is the general-purpose register to be transferred, encoded in the "Rt" field.
`<Rn>`	Is the general-purpose base register, encoded in the "Rn" field.
`<imm>`	For encoding A1: the immediate offset added to the value of `<Rn>` to calculate the address. `<imm>` can only be 0 or omitted.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For encoding T1: the immediate offset added to the value of `<Rn>` to calculate the address. `<imm>` can be omitted, meaning an offset of 0. Values are multiples of 4 in the range 0-1020.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    AArch32.SetExclusiveMonitors(address,4);
    R[t] = MemA[address,4];
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.83 LDREXB

Load Register Exclusive Byte derives an address from a base register value, loads a byte from memory, zero-extends it to form a 32-bit word, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-4023. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3	2	1	0
!=1111	0	0	0	1	1	1	0	1	Rn	Rt	(1)	(1)	1	1	1	0	0	1	(1)	(1)	(1)	(1)
cond																						

A1 variant

```
LDREXB{<c>}{<q>} <Rt>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	1	Rn	Rt	(1)	(1)	(1)	(1)	0	1	0	0	(1)	(1)	(1)	(1)

T1 variant

```
LDREXB{<c>}{<q>} <Rt>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    AArch32.SetExclusiveMonitors(address,1);
    R[t] = ZeroExtend(MemA[address,1], 32);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.84 LDREXD

Load Register Exclusive Doubleword derives an address from a base register value, loads a 64-bit doubleword from memory, writes it to two registers and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-4023. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3	2	1	0
!=1111	0	0	0	1	1	0	1	1	Rn	Rt	(1)	(1)	1	1	1	0	0	1	(1)	(1)	(1)	(1)
cond																						

A1 variant

```
LDREXD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt);  t2 = t + 1;  n = UInt(Rn);
if Rt<0> == '1' || t2 == 15 || n == 15 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `Rt<0> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: t<0> = '0'.
- The instruction executes with the additional decode: t2 = t.
- The instruction executes as described, with no change to its behavior and no additional side effects.

If `Rt == '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction is handled as described in *Using R15* on page K1-7831.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11–8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	1	Rn	Rt	Rt2	0	1	1	1	(1)	(1)	(1)	(1)

T1 variant

```
LDREXD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);
if t == 15 || t2 == 15 || t == t2 || n == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The load instruction executes but the destination register takes an UNKNOWN value.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rt>`	For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. `<Rt>` must be even-numbered and not R14. For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.
`<Rt2>`	For encoding A1: is the second general-purpose register to be transferred. `<Rt2>` must be `<R(t+1)>`. For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.
`<Rn>`	Is the general-purpose base register, encoded in the "Rn" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    AArch32.SetExclusiveMonitors(address,8);
    value = MemA[address,8];
    // Extract words from 64-bit loaded value such that R[t] is
    // loaded from address and R[t2] from address+4.
    R[t]  = if BigEndian() then value<63:32> else value<31:0>;
    R[t2] = if BigEndian() then value<31:0> else value<63:32>;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.85 LDREXH

Load Register Exclusive Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it to form a 32-bit word, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-4023. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3	2	1	0
!=1111	0	0	0	1	1	1	1	1	Rn	Rt	(1)	(1)	1	1	1	0	0	1	(1)	(1)	(1)	(1)
cond																						

A1 variant

```
LDREXH{<c>}{<q>} <Rt>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	1	Rn	Rt	(1)	(1)	(1)	(1)	0	1	0	1	(1)	(1)	(1)	(1)

T1 variant

```
LDREXH{<c>}{<q>} <Rt>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    AArch32.SetExclusiveMonitors(address,2);
    R[t] = ZeroExtend(MemA[address,2], 32);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.86 LDRH (immediate)

Load Register Halfword (immediate) calculates an address from a base register value and an immediate offset, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27 26 25	24	23	22	21	20	19–16	15–12	11–8	7	6 5	4	3–0
!=1111 (cond)	0 0 0	P	U	1	W	1	!=1111 (Rn)	Rt	imm4H	1	0 1	1	imm4L

Offset variant

Applies when `P == 1 && W == 0`.

```
LDRH{<c>}{<q>} <Rt>, [<Rn> {, #{+/-}<imm>}]
```

Post-indexed variant

Applies when `P == 0 && W == 0`.

```
LDRH{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
LDRH{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if Rn == '1111' then SEE "LDRH (literal)";
if P == '0' && W == '1' then SEE "LDRHT";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

15 14 13 12	11	10–6	5–3	2–0
1 0 0 0	1	imm5	Rn	Rt

T1 variant

```
LDRH{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'0', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11 … 0
1	1	1	1	1	0	0	0	1	0	1	1	!=1111 (Rn)	!=1111 (Rt)	imm12

T2 variant

```
LDRH{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}] // <Rt>, <Rn>, <imm> can be represented in T1
LDRH{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
if Rt == '1111' then SEE "PLD (immediate)";
if Rn == '1111' then SEE "LDRH (literal)";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
```

T3

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7 … 0
1	1	1	1	1	0	0	0	0	0	1	1	!=1111 (Rn)	Rt	1	P	U	W	imm8

Offset variant

Applies when `Rt != 1111 && P == 1 && U == 0 && W == 0`.

```
LDRH{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

Post-indexed variant

Applies when `P == 0 && W == 1`.

```
LDRH{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
LDRH{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if Rn == '1111' then SEE "LDRH (literal)";
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "PLDW (immediate)";
if P == '1' && U == '1' && W == '0' then SEE "LDRHT";
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if (t == 15 && W == '1') || (wback && n == t) then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.

`<Rn>` For encoding A1, T2 and T3: is the general-purpose base register, encoded in the "Rn" field. For PC use see LDRH (literal).

For encoding T1: is the general-purpose base register, encoded in the "Rn" field.

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

`-` when `U = 0`

`+` when `U = 1`

`+` Specifies the offset is added to the base register.

`<imm>` For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is the optional positive unsigned immediate byte offset, a multiple of 2, in the range 0 to 62, defaulting to 0 and encoded in the "imm5" field as <imm>/2.

For encoding T2: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T3: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

Operation for all encodings

```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        data = MemU[address,2];
        if wback then R[n] = offset_addr;
        R[t] = ZeroExtend(data, 32);
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
data = MemU[address,2];
if wback then R[n] = offset_addr;
R[t] = ZeroExtend(data, 32);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.87 LDRH (literal)

Load Register Halfword (literal) calculates an address from the PC value and an immediate offset, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11–8	7	6	5	4	3–0
!=1111 (cond)	0	0	0	P	U	1	W	1	1	1	1	1	Rt	imm4H	1	0	1	1	imm4L

A1 variant

Applies when `!(P == 0 && W == 1)`.

```
LDRH{<c>}{<q>} <Rt>, <label> // Normal form
LDRH{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // Alternative form
```

Decode for this encoding

```
if P == '0' && W == '1' then SEE "LDRHT";
t = UInt(Rt);  imm32 = ZeroExtend(imm4H:imm4L, 32);
add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || wback then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: wback = FALSE;.
- The instruction treats bit[24] as the P bit, and bit[21] as the writeback (W) bit, and uses the same addressing mode as described in LDRH (immediate). The instruction uses post-indexed addressing when P == '0' and uses pre-indexed addressing otherwise. The instruction is handled as described in *Using R15* on page K1-7831.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15–12	11–0
1	1	1	1	1	0	0	0	U	0	1	1	1	1	1	1	!=1111 (Rt)	imm12

T1 variant

```
LDRH{<c>}{<q>} <Rt>, <label> // Preferred syntax
LDRH{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // Alternative syntax
```

Decode for this encoding

```
if Rt == '1111' then SEE "PLD (literal)";
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
// Armv8-A removes UNPREDICTABLE for R13
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.

`<label>` For encoding A1: the label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Any value in the range -255 to 255 is permitted.

If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`, encoded as U == 1. If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`, encoded as U == 0.

For encoding T1: the label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are -4095 to 4095.

If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`, encoded as U == 1.

If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`, encoded as U == 0.

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

`-` when `U = 0`

`+` when `U = 1`

`<imm>` For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page F1-4037.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    data = MemU[address,2];
    R[t] = ZeroExtend(data, 32);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.88 LDRH (register)

Load Register Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	0	0	0	P	U	0	W	1	Rn	Rt	(0)	(0)	(0)	(0)	1	0	1	1	Rm
cond																			

Offset variant

Applies when `P == 1 && W == 0`.

```
LDRH{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]
```

Post-indexed variant

Applies when `P == 0 && W == 0`.

```
LDRH{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>
```

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
LDRH{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]!
```

Decode for all variants of this encoding

```
if P == '0' && W == '1' then SEE "LDRHT";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = (SRType_LSL, 0);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

15	14	13	12	11	10	9	8–6	5–3	2–0
0	1	0	1	1	0	1	Rm	Rn	Rt

T1 variant

```
LDRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 10 9 8	7 6 5 4	3 0
1 1 1 1	1 0 0 0 0	0 1 1	!=1111 (Rn)	!=1111 (Rt)	0 0 0 0	0 0 imm2	Rm

T2 variant

```
LDRH{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // <Rt>, <Rn>, <Rm> can be represented in T1
LDRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then SEE "LDRH (literal)";
if Rt == '1111' then SEE "PLDW (register)";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.

`<Rn>` For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.

For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.

`+/-` Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`+` Specifies the index register is added to the base register.

`<Rm>` Is the general-purpose index register, encoded in the "Rm" field.

`<imm>` If present, the size of the left shift to apply to the value from `<Rm>`, in the range 1-3. `<imm>` is encoded in imm2. If absent, no shift is specified and imm2 is encoded as `0b00`.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
offset_addr = if add then (R[n] + offset) else (R[n] - offset);
address = if index then offset_addr else R[n];
data = MemU[address,2];
if wback then R[n] = offset_addr;
R[t] = ZeroExtend(data, 32);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.89 LDRHT

Load Register Halfword Unprivileged loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses* on page F2-4085.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

`LDRHT` is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
!=1111	0	0	0	0	U	1	1	1	Rn	Rt	imm4H	1	0	1	1	imm4L
cond																

A1 variant

```
LDRHT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/-}<imm>}
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm4H:imm4L, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in *Using R15* on page K1-7831.
- The instruction is treated as if bit[24] == '1' and bit[21] == '0'. The instruction uses immediate offset addressing with the base register as PC, without writeback.

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A2

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111 (cond)	0	0	0	0	U	0	1	1	Rn	Rt	(0)	(0)	(0)	(0)	1	0	1	1	Rm

A2 variant

```
LDRHT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7–0
1	1	1	1	1	0	0	0	0	0	1	1	!=1111 (Rn)	Rt	1	1	1	0	imm8

T1 variant

```
LDRHT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then SEE "LDRH (literal)";
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rt>`	Is the general-purpose register to be transferred, encoded in the "Rt" field.
`<Rn>`	Is the general-purpose base register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

+/-
: For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

  - when U = 0
  + when U = 1

  For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

  - when U = 0
  + when U = 1

<Rm>
: Is the general-purpose index register, encoded in the "Rm" field.

+
: Specifies the offset is added to the base register.

<imm>
: For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

  For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

Operation for all encodings

```
if ConditionPassed() then
    if PSTATE.EL == EL2 then UNPREDICTABLE;               // Hyp mode
    EncodingSpecificOperations();
    offset = if register_form then R[m] else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    data = MemU_unpriv[address,2];
    if postindex then R[n] = offset_addr;
    R[t] = ZeroExtend(data, 32);
```

CONSTRAINED UNPREDICTABLE behavior

If PSTATE.EL == EL2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDRH (immediate).

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.90 LDRSB (immediate)

Load Register Signed Byte (immediate) calculates an address from a base register value and an immediate offset, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
!=1111	0	0	0	P	U	1	W	1	!=1111	Rt	imm4H	1	1	0	1	imm4L
cond									Rn							

Offset variant

Applies when `P == 1 && W == 0`.

```
LDRSB{<c>}{<q>} <Rt>, [<Rn> {, #{+/-}<imm>}]
```

Post-indexed variant

Applies when `P == 0 && W == 0`.

```
LDRSB{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
LDRSB{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if Rn == '1111' then SEE "LDRSB (literal)";
if P == '0' && W == '1' then SEE "LDRSBT";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11–0
1	1	1	1	1	0	0	1	1	0	0	1	!=1111	!=1111	imm12
												Rn	Rt	

T1 variant

```
LDRSB{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
if Rt == '1111' then SEE "PLI";
if Rn == '1111' then SEE "LDRSB (literal)";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7 … 0
1	1	1	1	1	0	0	1	0	0	0	1	!=1111 (Rn)	Rt	1	P	U	W	imm8

Offset variant

Applies when `Rt != 1111 && P == 1 && U == 0 && W == 0`.

```
LDRSB{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

Post-indexed variant

Applies when `P == 0 && W == 1`.

```
LDRSB{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
LDRSB{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "PLI";
if Rn == '1111' then SEE "LDRSB (literal)";
if P == '1' && U == '1' && W == '0' then SEE "LDRSBT";
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if (t == 15 && W == '1') || (wback && n == t) then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field. For PC use see LDRSB (literal).

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`+` Specifies the offset is added to the base register.

`<imm>` For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T2: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    R[t] = SignExtend(MemU[address,1], 32);
    if wback then R[n] = offset_addr;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.91 LDRSB (literal)

Load Register Signed Byte (literal) calculates an address from the PC value and an immediate offset, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11–8	7	6	5	4	3–0
!=1111	0	0	0	P	U	1	W	1	1	1	1	1	Rt	imm4H	1	1	0	1	imm4L
cond																			

A1 variant

Applies when `!(P == 0 && W == 1)`.

```
LDRSB{<c>}{<q>} <Rt>, <label> // Normal form
LDRSB{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // Alternative form
```

Decode for this encoding

```
if P == '0' && W == '1' then SEE "LDRSBT";
t = UInt(Rt);  imm32 = ZeroExtend(imm4H:imm4L, 32);
add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || wback then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: wback = FALSE;.
- The instruction treats bit[24] as the P bit, and bit[21] as the writeback (W) bit, and uses the same addressing mode as described in LDRSB (immediate). The instruction uses post-indexed addressing when P == '0' and uses pre-indexed addressing otherwise. The instruction is handled as described in *Using R15* on page K1-7831.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15–12	11–0
1	1	1	1	1	0	0	1	U	0	0	1	1	1	1	1	!=1111	imm12
																Rt	

T1 variant

```
LDRSB{<c>}{<q>} <Rt>, <label> // Preferred syntax
LDRSB{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // Alternative syntax
```

Decode for this encoding

```
if Rt == '1111' then SEE "PLI";
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
// Armv8-A removes UNPREDICTABLE for R13
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.

`<label>` For encoding A1: the label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Any value in the range -255 to 255 is permitted.

If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`, encoded as U == 1. If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`, encoded as U == 0.

For encoding T1: the label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are -4095 to 4095.

If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`, encoded as U == 1.

If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`, encoded as U == 0.

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

`-` when `U = 0`

`+` when `U = 1`

`<imm>` For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page F1-4037.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    R[t] = SignExtend(MemU[address,1], 32);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.92 LDRSB (register)

Load Register Signed Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	0	0	0	P	U	0	W	1	Rn	Rt	(0)	(0)	(0)	(0)	1	1	0	1	Rm
cond																			

Offset variant

Applies when `P == 1 && W == 0`.

```
LDRSB{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]
```

Post-indexed variant

Applies when `P == 0 && W == 0`.

```
LDRSB{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>
```

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
LDRSB{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]!
```

Decode for all variants of this encoding

```
if P == '0' && W == '1' then SEE "LDRSBT";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = (SRType_LSL, 0);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

15	14	13	12	11	10	9	8–6	5–3	2–0
0	1	0	1	0	1	1	Rm	Rn	Rt

T1 variant

```
LDRSB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 10 9 8	7 6 5 4	3 0
1 1 1 1	1 0 0 1 0	0 0 1	!=1111 (Rn)	!=1111 (Rt)	0 0 0 0	0 0 imm2	Rm

T2 variant

```
LDRSB{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // <Rt>, <Rn>, <Rm> can be represented in T1
LDRSB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

Decode for this encoding

```
if Rt == '1111' then SEE "PLI";
if Rn == '1111' then SEE "LDRSB (literal)";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.

`<Rn>` For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.

For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.

`+/-` Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`+` Specifies the index register is added to the base register.

`<Rm>` Is the general-purpose index register, encoded in the "Rm" field.

`<imm>` If present, the size of the left shift to apply to the value from `<Rm>`, in the range 1-3. `<imm>` is encoded in imm2. If absent, no shift is specified and imm2 is encoded as `0b00`.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
offset_addr = if add then (R[n] + offset) else (R[n] - offset);
address = if index then offset_addr else R[n];
R[t] = SignExtend(MemU[address,1], 32);
if wback then R[n] = offset_addr;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.93 LDRSBT

Load Register Signed Byte Unprivileged loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses* on page F2-4085.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

`LDRSBT` is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
!=1111	0	0	0	0	U	1	1	1	Rn	Rt	imm4H	1	1	0	1	imm4L
cond																

A1 variant

```
LDRSBT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/-}<imm>}
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm4H:imm4L, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in *Using R15* on page K1-7831.
- The instruction is treated as if bit[24] == '1' and bit[21] == '0'. The instruction uses immediate offset addressing with the base register as PC, without writeback.

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A2

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	0	0	0	0	U	0	1	1	Rn	Rt	(0)	(0)	(0)	(0)	1	1	0	1	Rm
cond																			

A2 variant

```
LDRSBT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7–0
1	1	1	1	1	0	0	1	0	0	0	1	!=1111	Rt	1	1	1	0	imm8
												Rn						

T1 variant

```
LDRSBT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then SEE "LDRSB (literal)";
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rt>`	Is the general-purpose register to be transferred, encoded in the "Rt" field.
`<Rn>`	Is the general-purpose base register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

+/-
: For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

  - when U = 0
  + when U = 1

  For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

  - when U = 0
  + when U = 1

<Rm>
: Is the general-purpose index register, encoded in the "Rm" field.

+
: Specifies the offset is added to the base register.

<imm>
: For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

  For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

Operation for all encodings

```
if ConditionPassed() then
    if PSTATE.EL == EL2 then UNPREDICTABLE;               // Hyp mode
    EncodingSpecificOperations();
    offset = if register_form then R[m] else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    R[t] = SignExtend(MemU_unpriv[address,1], 32);
    if postindex then R[n] = offset_addr;
```

CONSTRAINED UNPREDICTABLE behavior

If PSTATE.EL == EL2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDRSB (immediate).

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.94 LDRSH (immediate)

Load Register Signed Halfword (immediate) calculates an address from a base register value and an immediate offset, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
!=1111	0	0	0	P	U	1	W	1	!=1111	Rt	imm4H	1	1	1	1	imm4L
cond									Rn							

Offset variant

Applies when `P == 1 && W == 0`.

```
LDRSH{<c>}{<q>} <Rt>, [<Rn> {, #{+/-}<imm>}]
```

Post-indexed variant

Applies when `P == 0 && W == 0`.

```
LDRSH{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
LDRSH{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if Rn == '1111' then SEE "LDRSH (literal)";
if P == '0' && W == '1' then SEE "LDRSHT";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11–0
1	1	1	1	1	0	0	1	1	0	1	1	!=1111	!=1111	imm12
												Rn	Rt	

T1 variant

```
LDRSH{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
if Rn == '1111' then SEE "LDRSH (literal)";
if Rt == '1111' then SEE "Related instructions";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7 … 0
1	1	1	1	1	0	0	1	0	0	1	1	!=1111 (Rn)	Rt	1	P	U	W	imm8

Offset variant

Applies when `Rt != 1111 && P == 1 && U == 0 && W == 0`.

```
LDRSH{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

Post-indexed variant

Applies when `P == 0 && W == 1`.

```
LDRSH{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
LDRSH{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if Rn == '1111' then SEE "LDRSH (literal)";
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "Related instructions";
if P == '1' && U == '1' && W == '0' then SEE "LDRSHT";
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if (t == 15 && W == '1') || (wback && n == t) then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Related instructions: *Load/store single* on page F3-4163.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field. For PC use see LDRSH (literal).

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`+` Specifies the offset is added to the base register.

`<imm>` For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T2: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    R[t] = SignExtend(data, 32);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.95 LDRSH (literal)

Load Register Signed Halfword (literal) calculates an address from the PC value and an immediate offset, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11–8	7	6	5	4	3–0
!=1111	0	0	0	P	U	1	W	1	1	1	1	1	Rt	imm4H	1	1	1	1	imm4L
cond																			

A1 variant

Applies when `!(P == 0 && W == 1)`.

```
LDRSH{<c>}{<q>} <Rt>, <label> // Normal form
LDRSH{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // Alternative form
```

Decode for this encoding

```
if P == '0' && W == '1' then SEE "LDRSHT";
t = UInt(Rt);  imm32 = ZeroExtend(imm4H:imm4L, 32);
add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || wback then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: wback = FALSE;.
- The instruction treats bit[24] as the P bit, and bit[21] as the writeback (W) bit, and uses the same addressing mode as described in LDRSH (immediate). The instruction uses post-indexed addressing when P == '0' and uses pre-indexed addressing otherwise. The instruction is handled as described in *Using R15* on page K1-7831.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15–12	11–0
1	1	1	1	1	0	0	1	U	0	1	1	1	1	1	1	!=1111	imm12
																Rt	

T1 variant

```
LDRSH{<c>}{<q>} <Rt>, <label> // Preferred syntax
LDRSH{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // Alternative syntax
```

Decode for this encoding

```
if Rt == '1111' then SEE "Related instructions";
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
// Armv8-A removes UNPREDICTABLE for R13
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Related instructions: *Load, signed (literal)* on page F3-4171.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.

`<label>` For encoding A1: the label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Any value in the range -255 to 255 is permitted.

If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`, encoded as U == 1. If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`, encoded as U == 0.

For encoding T1: the label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are -4095 to 4095.

If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`, encoded as U == 1.

If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`, encoded as U == 0.

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

`-` when `U = 0`

`+` when `U = 1`

`<imm>` For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page F1-4037.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    data = MemU[address,2];
    R[t] = SignExtend(data, 32);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.96 LDRSH (register)

Load Register Signed Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	0	0	0	P	U	0	W	1	Rn	Rt	(0)	(0)	(0)	(0)	1	1	1	1	Rm
cond																			

Offset variant

Applies when `P == 1 && W == 0`.

```
LDRSH{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]
```

Post-indexed variant

Applies when `P == 0 && W == 0`.

```
LDRSH{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>
```

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
LDRSH{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]!
```

Decode for all variants of this encoding

```
if P == '0' && W == '1' then SEE "LDRSHT";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = (SRType_LSL, 0);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is <arm-defined-word>unknown</arm-defined-word>. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

15	14	13	12	11	10	9	8–6	5–3	2–0
0	1	0	1	1	1	1	Rm	Rn	Rt

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1 variant

```
LDRSH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 10 9 8	7 6 5 4	3 0
1 1 1 1	1 0 0 1 0	0 1 1	!=1111	!=1111	0 0 0 0	0 0 0 imm2	Rm
			Rn	Rt			

T2 variant

```
LDRSH{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // <Rt>, <Rn>, <Rm> can be represented in T1
LDRSH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then SEE "LDRSH (literal)";
if Rt == '1111' then SEE "Related instructions";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Related instructions: *Load/store, signed (register offset)* on page F3-4168.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.

`<Rn>` For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.

For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.

`+/-` Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`+` Specifies the index register is added to the base register.

`<Rm>` Is the general-purpose index register, encoded in the "Rm" field.

`<imm>` If present, the size of the left shift to apply to the value from `<Rm>`, in the range 1-3. `<imm>` is encoded in imm2. If absent, no shift is specified and imm2 is encoded as `0b00`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    R[t] = SignExtend(data, 32);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.97 LDRSHT

Load Register Signed Halfword Unprivileged loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses* on page F2-4085.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

`LDRSHT` is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
!=1111	0	0	0	0	U	1	1	1	Rn	Rt	imm4H	1	1	1	1	imm4L
cond																

A1 variant

```
LDRSHT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/-}<imm>}
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm4H:imm4L, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in *Using R15* on page K1-7831.
- The instruction is treated as if bit[24] == '1' and bit[21] == '0'. The instruction uses immediate offset addressing with the base register as PC, without writeback.

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A2

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111 (cond)	0	0	0	0	U	0	1	1	Rn	Rt	(0)	(0)	(0)	(0)	1	1	1	1	Rm

A2 variant

```
LDRSHT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7–0
1	1	1	1	1	0	0	1	0	0	1	1	!=1111 (Rn)	Rt	1	1	1	0	imm8

T1 variant

```
LDRSHT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then SEE "LDRSH (literal)";
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rt>`	Is the general-purpose register to be transferred, encoded in the "Rt" field.
`<Rn>`	Is the general-purpose base register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

+/-
: For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

  - when U = 0
  + when U = 1

  For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

  - when U = 0
  + when U = 1

<Rm>
: Is the general-purpose index register, encoded in the "Rm" field.

+
: Specifies the offset is added to the base register.

<imm>
: For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

  For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

Operation for all encodings

```
if ConditionPassed() then
    if PSTATE.EL == EL2 then UNPREDICTABLE;               // Hyp mode
    EncodingSpecificOperations();
    offset = if register_form then R[m] else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    data = MemU_unpriv[address,2];
    if postindex then R[n] = offset_addr;
    R[t] = SignExtend(data, 32);
```

CONSTRAINED UNPREDICTABLE behavior

If PSTATE.EL == EL2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDRSH (immediate).

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.98 LDRT

Load Register Unprivileged loads a word from memory, and writes it to a register. For information about memory accesses see *Memory accesses* on page F2-4085.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

`LDRT` is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
!=1111	0	1	0	0	U	0	1	1	Rn	Rt	imm12
cond											

A1 variant

```
LDRT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/-}<imm>}
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm12, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in *Using R15* on page K1-7831.
- The instruction is treated as if bit[24] == '1' and bit[21] == '0'. The instruction uses immediate offset addressing with the base register as PC, without writeback.

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A2

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–7	6–5	4	3–0
!=1111	0	1	1	0	U	0	1	1	Rn	Rt	imm5	stype	0	Rm
cond														

A2 variant

```
LDRT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;  (shift_t, shift_n) = DecodeImmShift(stype, imm5);
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7–0
1	1	1	1	1	0	0	0	0	1	0	1	!=1111	Rt	1	1	1	0	imm8
												Rn						

T1 variant

```
LDRT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then SEE "LDR (literal)";
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rt>` For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.

For encoding A2 and T1: is the general-purpose register to be transferred, encoded in the "Rt" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`+/-` For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`<Rm>` Is the general-purpose index register, encoded in the "Rm" field.

`<shift>` The shift to apply to the value read from `<Rm>`. If absent, no shift is applied. Otherwise, see *Shifts applied to a register* on page F2-4083.

`+` Specifies the offset is added to the base register.

`<imm>` For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

Operation for all encodings

```
if ConditionPassed() then
    if PSTATE.EL == EL2 then UNPREDICTABLE;               // Hyp mode
    EncodingSpecificOperations();
    offset = if register_form then Shift(R[m], shift_t, shift_n, PSTATE.C) else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    data = MemU_unpriv[address,4];
    if postindex then R[n] = offset_addr;
    R[t] = data;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `LDR` (immediate).

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.99 LSL (immediate)

Logical Shift Left (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This instruction is an alias of the MOV, MOVS (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

A1

31–28	27 26 25 24	23 22 21 20	19 18 17 16	15–12	11–7	6 5	4	3–0
!=1111	0 0 0 1 1	0 1 0	(0) (0) (0) (0)	Rd	!=00000	0 0	0	Rm
cond		S			imm5	stype		

MOV, shift or rotate by value variant

```
LSL{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>
```

and is always the preferred disassembly.

T2

15 14 13	12 11	10–6	5–3	2–0
0 0 0	0 0	!=00000	Rm	Rd
	op	imm5		

T2 variant

```
LSL<c>{<q>} {<Rd>,} <Rm>, #<imm> // Inside IT block
```

is equivalent to

```
MOV<c>{<q>} <Rd>, <Rm>, LSL #<imm>
```

and is the preferred disassembly when `InITBlock()`.

T3

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 14	12–11	8–7	6 5	4	3–0
1 1 1 0	1 0 1	0 0 1 0	0	1 1 1 1	(0)	imm3	Rd	imm2	0 0	Rm
			S						stype	

MOV, shift or rotate by value variant

```
LSL<c>.W {<Rd>,} <Rm>, #<imm> // Inside IT block, and <Rd>, <Rm>, <imm> can be represented in T2
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>
```

and is always the preferred disassembly.

```
LSL{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>
```

and is always the preferred disassembly.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.

For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.

`<imm>` For encoding A1: is the shift amount, in the range 0 to 31, encoded in the "imm5" field as <imm> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31, encoded in the "imm5" field as <amount> modulo 32.

For encoding T3: is the shift amount, in the range 0 to 31, encoded in the "imm3:imm2" field as <imm> modulo 32.

Operation for all encodings

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.100 LSL (register)

Logical Shift Left (register) shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

A1

31–28	27 26 25 24	23 22 21 20	19 18 17 16	15–12	11–8	7	6 5	4	3–0
!=1111	0 0 0 1 1	0 1 0	(0) (0) (0) (0)	Rd	Rs	0	0 0	1	Rm
cond		S					stype		

Not flag setting variant

```
LSL{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSL <Rs>
```

and is always the preferred disassembly.

T1

15 14 13 12	11 10	9–6	5–3	2–0
0 1 0 0	0 0	0 0 1 0	Rs	Rdm
		op		

Logical shift left variant

```
LSL<c>{<q>} {<Rdm>,} <Rdm>, <Rs> // Inside IT block
```

is equivalent to

```
MOV<c>{<q>} <Rdm>, <Rdm>, LSL <Rs>
```

and is the preferred disassembly when `InITBlock()`.

T2

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15 14 13 12	11–8	7 6 5 4	3–0
1 1 1 1	1 0 1 0	0 0 0	Rm	1 1 1 1	Rd	0 0 0 0	Rs
		stype S					

Not flag setting variant

```
LSL<c>.W {<Rd>,} <Rm>, <Rs> // Inside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSL <Rs>
```

and is always the preferred disassembly.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
LSL{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSL <Rs>
```

and is always the preferred disassembly.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rdm>`	Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<Rm>`	Is the first general-purpose source register, encoded in the "Rm" field.
`<Rs>`	Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation for all encodings

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.101 LSLS (immediate)

Logical Shift Left, setting flags (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

If the destination register is not the PC, this instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This instruction is an alias of the MOV, MOVS (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

A1

31–28	27 26 25 24	23 22 21 20	19 18 17 16	15–12	11–7	6 5	4	3–0
!=1111	0 0 0 1 1	0 1 1	(0) (0) (0) (0)	Rd	!=00000	0 0	0	Rm
cond		S			imm5	stype		

MOVS, shift or rotate by value variant

```
LSLS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>
```

and is always the preferred disassembly.

T2

15 14 13	12 11	10–6	5–3	2–0
0 0 0	0 0	!=00000	Rm	Rd
	op	imm5		

T2 variant

```
LSLS{<q>} {<Rd>,} <Rm>, #<imm> // Outside IT block
```

is equivalent to

```
MOVS{<q>} <Rd>, <Rm>, LSL #<imm>
```

and is the preferred disassembly when `!InITBlock()`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T3

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 14 12	11 8	7 6 5 4	3 0
1 1 1 0	1 0 1 0	0 1 1 0	1 1 1 1	(0) imm3	Rd	imm2 0 0	Rm
						stype	
		S					

MOVS, shift or rotate by value variant

```
LSLS.W {<Rd>,} <Rm>, #<imm> // Outside IT block, and <Rd>, <Rm>, <imm> can be represented in T2
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>
```

and is always the preferred disassembly.

```
LSLS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>
```

and is always the preferred disassembly.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.

`<imm>` For encoding A1: is the shift amount, in the range 0 to 31, encoded in the "imm5" field as <imm> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31, encoded in the "imm5" field as <amount> modulo 32.

For encoding T3: is the shift amount, in the range 0 to 31, encoded in the "imm3:imm2" field as <imm> modulo 32.

Operation for all encodings

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.102 LSLS (register)

Logical Shift Left, setting flags (register) shifts a register value left by a variable number of bits, shifting in zeros, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

A1

31–28	27–24	23–20		19–16	15–12	11–8	7	6–5	4	3–0
!=1111	0 0 0 1 1	0 1	1	(0) (0) (0) (0)	Rd	Rs	0	0 0	1	Rm
cond			S					stype		

Flag setting variant

```
LSLS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSL <Rs>
```

and is always the preferred disassembly.

T1

15–14	13–12	11–10	9–6	5–3	2–0
0 1	0 0	0 0	0 0 1 0	Rs	Rdm
			op		

Logical shift left variant

```
LSLS{<q>} {<Rdm>,} <Rdm>, <Rs> // Outside IT block
```

is equivalent to

```
MOVS{<q>} <Rdm>, <Rdm>, LSL <Rs>
```

and is the preferred disassembly when `!InITBlock()`.

T2

15–14	13–12	11–10	9–8	7–6	5	4	3–0	15–14	13–12	11–8	7–6	5–4	3–0
1 1	1 1	1 0	1 0	0 0		1	Rm	1 1	1 1	Rd	0 0	0 0	Rs
				stype		S							

Flag setting variant

```
LSLS.W {<Rd>,} <Rm>, <Rs> // Outside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSL <Rs>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

and is always the preferred disassembly.

```
LSLS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSL <Rs>
```

and is always the preferred disassembly.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rdm>`	Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<Rm>`	Is the first general-purpose source register, encoded in the "Rm" field.
`<Rs>`	Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation for all encodings

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.103 LSR (immediate)

Logical Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This instruction is an alias of the MOV, MOVS (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

A1

31–28	27–25	24–23	22–21	20	19–16	15–12	11–7	6–5	4	3–0
!=1111	0 0 0	1 1	0 1	0	(0) (0) (0) (0)	Rd	imm5	0 1	0	Rm
cond				S				stype		

MOV, shift or rotate by value variant

```
LSR{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSR #<imm>
```

and is always the preferred disassembly.

T2

15–14	13–12	11–10	6–5	3–0
0 0	0 1	imm5	Rm	Rd
	op			

T2 variant

```
LSR<c>{<q>} {<Rd>,} <Rm>, #<imm> // Inside IT block
```

is equivalent to

```
MOV<c>{<q>} <Rd>, <Rm>, LSR #<imm>
```

and is the preferred disassembly when `InITBlock()`.

T3

15–14	13–12	11–10	9–8	7–6	5–4	3–2	1–0	15–14	12–11	8–7	6–5	4–0
1 1	1 0	1 0	1	0 0	1 0	0	1 1	1 1	(0) imm3	Rd	imm2	0 1	Rm
						S						stype	

MOV, shift or rotate by value variant

```
LSR<c>.W {<Rd>,} <Rm>, #<imm> // Inside IT block, and <Rd>, <Rm>, <imm> can be represented in T2
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSR #<imm>
```

and is always the preferred disassembly.

```
LSR{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSR #<imm>
```

and is always the preferred disassembly.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.

For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.

`<imm>` For encoding A1 and T2: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as <imm> modulo 32.

For encoding T3: is the shift amount, in the range 1 to 32, encoded in the "imm3:imm2" field as <imm> modulo 32.

Operation for all encodings

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.104 LSR (register)

Logical Shift Right (register) shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

A1

31–28	27 26 25 24	23 22 21 20	19 18 17 16	15–12	11–8	7	6 5	4	3–0
!=1111	0 0 0 1	1 0 1 0	(0) (0) (0) (0)	Rd	Rs	0	0 1	1	Rm
cond		S					stype		

Not flag setting variant

```
LSR{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>
```

and is always the preferred disassembly.

T1

15 14 13 12	11 10	9–6	5–3	2–0
0 1 0 0	0 0	0 0 1 1	Rs	Rdm
		op		

Logical shift right variant

```
LSR<c>{<q>} {<Rdm>,} <Rdm>, <Rs> // Inside IT block
```

is equivalent to

```
MOV<c>{<q>} <Rdm>, <Rdm>, LSR <Rs>
```

and is the preferred disassembly when `InITBlock()`.

T2

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15 14 13 12	11–8	7 6 5 4	3–0
1 1 1 1	1 0 1 0	0 1 0	Rm	1 1 1 1	Rd	0 0 0 0	Rs
		stype S					

Not flag setting variant

```
LSR<c>.W {<Rd>,} <Rm>, <Rs> // Inside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>
```

and is always the preferred disassembly.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
LSR{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>
```

and is always the preferred disassembly.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rdm>`	Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<Rm>`	Is the first general-purpose source register, encoded in the "Rm" field.
`<Rs>`	Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation for all encodings

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.105 LSRS (immediate)

Logical Shift Right, setting flags (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

If the destination register is not the PC, this instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This instruction is an alias of the MOV, MOVS (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

A1

31–28	27 26 25 24	23 22 21 20	19 18 17 16	15–12	11–7	6 5	4	3–0
!=1111	0 0 0 1 1	0 1 1	(0) (0) (0) (0)	Rd	imm5	0 1	0	Rm
cond		S				stype		

MOVS, shift or rotate by value variant

```
LSRS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSR #<imm>
```

and is always the preferred disassembly.

T2

15 14 13 12	11 10	6 5	3 2 0
0 0 0	0 1	imm5	Rm	Rd
	op			

T2 variant

```
LSRS{<q>} {<Rd>,} <Rm>, #<imm> // Outside IT block
```

is equivalent to

```
MOVS{<q>} <Rd>, <Rm>, LSR #<imm>
```

and is the preferred disassembly when `!InITBlock()`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T3

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 14 12	11 8	7 6 5 4	3 0
1 1 1 0	1 0 1 0	0 0 1 0	1 1 1 1 1	(0) imm3	Rd	imm2 0 1	Rm
			S			stype	

MOVS, shift or rotate by value variant

```
LSRS.W {<Rd>,} <Rm>, #<imm> // Outside IT block, and <Rd>, <Rm>, <imm> can be represented in T2
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSR #<imm>
```

and is always the preferred disassembly.

```
LSRS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSR #<imm>
```

and is always the preferred disassembly.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.

`<imm>` For encoding A1 and T2: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as <imm> modulo 32.

For encoding T3: is the shift amount, in the range 1 to 32, encoded in the "imm3:imm2" field as <imm> modulo 32.

Operation for all encodings

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.106 LSRS (register)

Logical Shift Right, setting flags (register) shifts a register value right by an immediate number of bits, shifting in zeros, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

A1

31–28	27 26 25 24	23 22 21 20	19 18 17 16	15–12	11–8	7	6 5	4	3–0
!=1111	0 0 0 1 1	0 1 1	(0) (0) (0) (0)	Rd	Rs	0	0 1	1	Rm
cond		S					stype		

Flag setting variant

```
LSRS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>
```

and is always the preferred disassembly.

T1

15 14 13 12	11 10 9	6 5	3 2 0
0 1 0 0 0 0	0 0 1 1	Rs	Rdm
	op		

Logical shift right variant

```
LSRS{<q>} {<Rdm>,} <Rdm>, <Rs> // Outside IT block
```

is equivalent to

```
MOVS{<q>} <Rdm>, <Rdm>, LSR <Rs>
```

and is the preferred disassembly when `!InITBlock()`.

T2

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15 14 13 12	11–8	7 6 5 4	3–0
1 1 1 1	1 0 1 0	0 1 1	Rm	1 1 1 1	Rd	0 0 0 0	Rs
		stype S					

Flag setting variant

```
LSRS.W {<Rd>,} <Rm>, <Rs> // Outside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

and is always the preferred disassembly.

```
LSRS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>
```

and is always the preferred disassembly.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rdm>`	Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<Rm>`	Is the first general-purpose source register, encoded in the "Rm" field.
`<Rs>`	Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation for all encodings

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.107 MCR

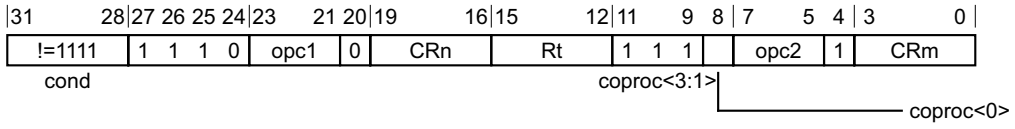
Move to System register from general-purpose register or execute a System instruction. This instruction copies the value of a general-purpose register to a System register, or executes a System instruction.

The System register and System instruction descriptions identify valid encodings for this instruction. Other encodings are UNDEFINED. For more information see *About the AArch32 System register interface* on page E1-3971 and *General behavior of System registers* on page G8-6096.

In an implementation that includes EL2, `MCR` accesses to System registers can be trapped to Hyp mode, meaning that an attempt to execute an `MCR` instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see *EL2 configurable controls* on page G1-5787.

Because of the range of possible traps to Hyp mode, the `MCR` pseudocode does not show these possible traps.

A1



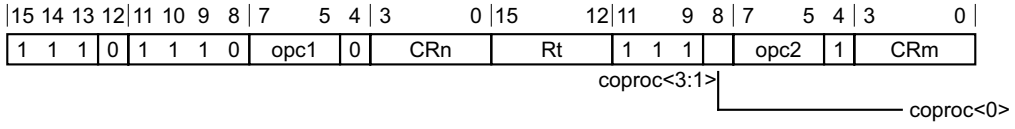
A1 variant

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

Decode for this encoding

```
t = UInt(Rt);  cp = if coproc<0> == '0' then 14 else 15;
if t == 15  then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

T1



T1 variant

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

Decode for this encoding

```
t = UInt(Rt);  cp = if coproc<0> == '0' then 14 else 15;
if t == 15  then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<coproc>` Is the System register encoding space, encoded in the "coproc<0>" field. It can have the following values:

`p14` when `coproc<0> = 0`

`p15` when `coproc<0> = 1`

`<opc1>` Is the opc1 parameter within the System register encoding space, in the range 0 to7, encoded in the "opc1" field.

`<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.

`<CRn>` Is the CRn parameter within the System register encoding space, in the range c0 to c15, encoded in the "CRn" field.

`<CRm>` Is the CRm parameter within the System register encoding space, in the range c0 to c15, encoded in the "CRm" field.

`<opc2>` Is the opc2 parameter within the System register encoding space, in the range 0 to7, encoded in the "opc2" field.

The possible values of { `<coproc>`, `<opc1>`, `<CRn>`, `<CRm>`, `<opc2>` } encode the entire System register and System instruction encoding space. Not all of this space is allocated, and the System register and System instruction descriptions identify the allocated encodings.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    AArch32.SysRegWrite(cp, ThisInstr(), R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.108 MCRR

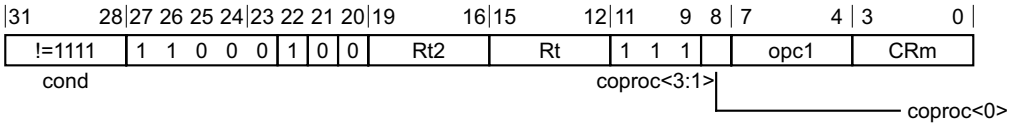
Move to System register from two general-purpose registers. This instruction copies the values of two general-purpose registers to a System register.

The System register descriptions identify valid encodings for this instruction. Other encodings are UNDEFINED. For more information see *About the AArch32 System register interface* on page E1-3971 and *General behavior of System registers* on page G8-6096.

In an implementation that includes EL2, `MCRR` accesses to System registers can be trapped to Hyp mode, meaning that an attempt to execute an `MCRR` instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see *EL2 configurable controls* on page G1-5787.

Because of the range of possible traps to Hyp mode, the `MCRR` pseudocode does not show these possible traps.

A1



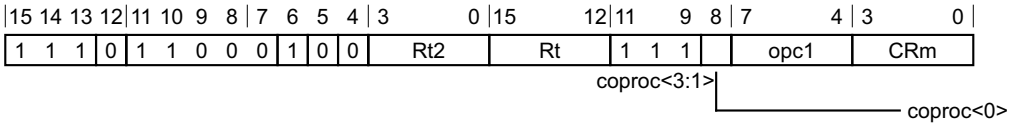
A1 variant

```
MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

Decode for this encoding

```
t = UInt(Rt);  t2 = UInt(Rt2);  cp = if coproc<0> == '0' then 14 else 15;
if t == 15 || t2 == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
```

T1



T1 variant

```
MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

Decode for this encoding

```
t = UInt(Rt);  t2 = UInt(Rt2);  cp = if coproc<0> == '0' then 14 else 15;
if t == 15 || t2 == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<coproc>` Is the System register encoding space, encoded in the "coproc<0>" field. It can have the following values:

`p14` when `coproc<0> = 0`

`p15` when `coproc<0> = 1`

`<opc1>` Is the opc1 parameter within the System register encoding space, in the range 0 to 15, encoded in the "opc1" field.

`<Rt>` Is the first general-purpose register that is transferred into, encoded in the "Rt" field.

`<Rt2>` Is the second general-purpose register that is transferred into, encoded in the "Rt2" field.

`<CRm>` Is the CRm parameter within the System register encoding space, in the range c0 to c15, encoded in the "CRm" field.

The possible values of { `<coproc>`, `<opc1>`, `<CRm>` } encode the entire System register encoding space. Not all of this space is allocated, and the System register descriptions identify the allocated encodings.

For the permitted uses of these instructions, as described in this manual, `<Rt2>` transfers bits[63:32] of the selected System register, while `<Rt>` transfers bits[31:0].

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    value = R[t2]:R[t];
    AArch32.SysRegWrite64(cp, ThisInstr(), value);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.109 MLA, MLAS

Multiply Accumulate multiplies two register values, and adds a third register value. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether the source register values are considered to be signed values or unsigned values.

In an A32 instruction, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

A1

31–28	27–24	23–20		19–16	15–12	11–8	7–4	3–0
!=1111 (cond)	0 0 0 0	0 0 1	S	Rd	Ra	Rm	1 0 0 1	Rn

Flag setting variant

Applies when `S == 1`.

```
MLAS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

Not flag setting variant

Applies when `S == 0`.

```
MLA{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  setflags = (S == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
```

T1

15–12	11–8	7–4	3–0	15–12	11–8	7–4	3–0
1 1 1 1	1 0 1 1	0 0 0	Rn	!=1111 (Ra)	Rd	0 0 0 0	Rm

T1 variant

```
MLA{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for this encoding

```
if Ra == '1111' then SEE "MUL";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  setflags = FALSE;
if d == 15 || n == 15 || m  == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Rn>` Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

`<Ra>` Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = SInt(R[n]);  // operand1 = UInt(R[n]) produces the same final results
    operand2 = SInt(R[m]);  // operand2 = UInt(R[m]) produces the same final results
    addend   = SInt(R[a]);  // addend   = UInt(R[a]) produces the same final results
    result = operand1 * operand2 + addend;
    R[d] = result<31:0>;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result<31:0>);
        // PSTATE.C, PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.110 MLS

Multiply and Subtract multiplies two register values, and subtracts the product from a third register value. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether the source register values are considered to be signed values or unsigned values.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–8	7 6 5 4	3–0
!=1111 (cond)	0 0 0 0	0 1 1 0	Rd	Ra	Rm	1 0 0 1	Rn

A1 variant

```
MLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15–12	11–8	7 6	5 4	3–0
1 1 1 1	1 0 1 1	0 0 0	Rn	Ra	Rd	0 0	0 1	Rm

T1 variant

```
MLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Ra>	Is the third general-purpose source register holding the minuend, encoded in the "Ra" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = SInt(R[n]);  // operand1 = UInt(R[n]) produces the same final results
    operand2 = SInt(R[m]);  // operand2 = UInt(R[m]) produces the same final results
    addend   = SInt(R[a]);  // addend   = UInt(R[a]) produces the same final results
    result = addend - operand1 * operand2;
    R[d] = result<31:0>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.111 MOV, MOVS (immediate)

Move (immediate) writes an immediate value to the destination register.

If the destination register is not the PC, the MOVS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The MOV variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- The MOVS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

31–28	27 26 25 24	23 22 21 20	19 18 17 16	15–12	11–0
!=1111 (cond)	0 0 1 1 1	0 1 S	(0) (0) (0) (0)	Rd	imm12

MOV variant

Applies when `S == 0`.

```
MOV{<c>}{<q>} <Rd>, #<const>
```

MOVS variant

Applies when `S == 1`.

```
MOVS{<c>}{<q>} <Rd>, #<const>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  setflags = (S == '1');  (imm32, carry) = A32ExpandImm_C(imm12, PSTATE.C);
```

A2

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–0
!=1111 (cond)	0 0 1 1 0	0 0 0	imm4	Rd	imm12

A2 variant

```
MOV{<c>}{<q>} <Rd>, #<imm16> // <imm16> can not be represented in A1
MOVW{<c>}{<q>} <Rd>, #<imm16> // <imm16> can be represented in A1
```

Decode for this encoding

```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(imm4:imm12, 32);
if d == 15 then UNPREDICTABLE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15 14 13	12 11	10 8	7 0
0 0 1	0 0	Rd	imm8

T1 variant

```
MOV<c>{<q>} <Rd>, #<imm8> // Inside IT block
MOVS{<q>} <Rd>, #<imm8> // Outside IT block
```

Decode for this encoding

```
d = UInt(Rd);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);  carry = PSTATE.C;
```

T2

15 14 13 12	11	10	9 8	7 6 5 4	3 2 1 0	15	14 12	11 8	7 0
1 1 1 1 0	i	0	0 0	1 0 S	1 1 1 1	0	imm3	Rd	imm8

MOV variant

Applies when `S == 0`.

```
MOV<c>.W <Rd>, #<const> // Inside IT block, and <Rd>, <const> can be represented in T1
MOV{<c>}{<q>} <Rd>, #<const>
```

MOVS variant

Applies when `S == 1`.

```
MOVS.W <Rd>, #<const> // Outside IT block, and <Rd>, <const> can be represented in T1
MOVS{<c>}{<q>} <Rd>, #<const>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  setflags = (S == '1');  (imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

T3

15 14 13 12	11	10	9 8	7	6 5 4	3 0	15	14 12	11 8	7 0
1 1 1 1 0	i	1	0	0	1 0 0	imm4	0	imm3	Rd	imm8

T3 variant

```
MOV{<c>}{<q>} <Rd>, #<imm16> // <imm16> cannot be represented in T1 or T2
MOVW{<c>}{<q>} <Rd>, #<imm16> // <imm16> can be represented in T1 or T2
```

Decode for this encoding

```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(imm4:i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the MOV variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- For the MOVS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding A2, T1, T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.

`<imm8>` Is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.

`<imm16>` For encoding A2: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:imm12" field.

For encoding T3: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:i:imm3:imm8" field.

`<const>` For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* on page F2-4096 for the range of values.

For encoding T2: an immediate value. See *Modified immediate constants in T32 instructions* on page F2-4095 for the range of values.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = imm32;
    if d == 15 then          // Can only occur for encoding A1
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.112 MOV, MOVS (register)

Move (register) copies a value from a register to the destination register.

If the destination register is not the PC, the MOVS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The MOV variant of the instruction is a branch. In the T32 instruction set (encoding T1) this is a simple branch, and in the A32 instruction set it is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- The MOVS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This instruction is used by the aliases ASRS (immediate), ASR (immediate), LSLS (immediate), LSL (immediate), LSRS (immediate), LSR (immediate), RORS (immediate), ROR (immediate), RRXS, and RRX. See *Alias conditions* on page F5-4519 for details of when each alias is preferred.

A1

31–28	27 26 25 24	23 22 21 20	19 18 17 16	15–12	11–7	6 5	4	3–0
!=1111 (cond)	0 0 0 1 1	0 1 S	(0) (0) (0) (0)	Rd	imm5	stype	0	Rm

MOV, rotate right with extend variant

Applies when `S == 0 && imm5 == 00000 && stype == 11`.

```
MOV{<c>}{<q>} <Rd>, <Rm>, RRX
```

MOV, shift or rotate by value variant

Applies when `S == 0 && !(imm5 == 00000 && stype == 11)`.

```
MOV{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
```

MOVS, rotate right with extend variant

Applies when `S == 1 && imm5 == 00000 && stype == 11`.

```
MOVS{<c>}{<q>} <Rd>, <Rm>, RRX
```

MOVS, shift or rotate by value variant

Applies when `S == 1 && !(imm5 == 00000 && stype == 11)`.

```
MOVS{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15 14 13 12	11 10	9 8	7	6	3 2 0
0 1 0 0	0 1	1 0	D	Rm	Rd

T1 variant

```
MOV{<c>}{<q>} <Rd>, <Rm>
```

Decode for this encoding

```
d = UInt(D:Rd);  m = UInt(Rm);  setflags = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

T2

15 14 13 12	11 10	6 5	3 2 0
0 0 0	!=11	imm5	Rm	Rd
	op		

T2 variant

```
MOV<c>{<q>} <Rd>, <Rm> {, <shift> #<amount>} // Inside IT block
MOVS{<q>} <Rd>, <Rm> {, <shift> #<amount>} // Outside IT block
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = DecodeImmShift(op, imm5);
if op == '00' && imm5 == '00000' && InITBlock() then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `op == '00' && imm5 == '00000' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passed its condition code check.
- The instruction executes as `NOP`, as if it failed its condition code check.
- The instruction executes as `MOV` Rd, Rm.

T3

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 14 12	11 8	7 6 5 4	3 0
1 1 1 0	1 0 1	0 0 1 0	S	1 1 1 1	(0) imm3	Rd	imm2 stype	Rm

MOV, rotate right with extend variant

Applies when `S == 0 && imm3 == 000 && imm2 == 00 && stype == 11`.

```
MOV{<c>}{<q>} <Rd>, <Rm>, RRX
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MOV, shift or rotate by value variant

Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11)`.

```
MOV{<c>}.W <Rd>, <Rm> {, LSL #0} // <Rd>, <Rm> can be represented in T1
MOV<c>.W <Rd>, <Rm> {, <shift> #<amount>} // Inside IT block, and <Rd>, <Rm>, <shift>, <amount> can be
represented in T2
MOV{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
```

MOVS, rotate right with extend variant

Applies when `S == 1 && imm3 == 000 && imm2 == 00 && stype == 11`.

```
MOVS{<c>}{<q>} <Rd>, <Rm>, RRX
```

MOVS, shift or rotate by value variant

Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11)`.

```
MOVS.W <Rd>, <Rm> {, <shift> #<amount>} // Outside IT block, and <Rd>, <Rm>, <shift>, <amount> can be
represented in T1 or T2
MOVS{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Alias conditions

Alias	of variant	is preferred when
ASRS (immediate)	T3 (MOVS, shift or rotate by value), A1 (MOVS, shift or rotate by value)	`S == '1' && stype == '10'`
ASRS (immediate)	T2	`op == '10' && !InITBlock()`
ASR (immediate)	T3 (MOV, shift or rotate by value), A1 (MOV, shift or rotate by value)	`S == '0' && stype == '10'`
ASR (immediate)	T2	`op == '10' && InITBlock()`
LSLS (immediate)	T3 (MOVS, shift or rotate by value)	`S == '1' && imm3:Rd:imm2 != '000xxxx00' && stype == '00'`
LSLS (immediate)	A1 (MOVS, shift or rotate by value)	`S == '1' && imm5 != '00000' && stype == '00'`
LSLS (immediate)	T2	`op == '00' && imm5 != '00000' && !InITBlock()`
LSL (immediate)	T3 (MOV, shift or rotate by value)	`S == '0' && imm3:Rd:imm2 != '000xxxx00' && stype == '00'`
LSL (immediate)	A1 (MOV, shift or rotate by value)	`S == '0' && imm5 != '00000' && stype == '00'`
LSL (immediate)	T2	`op == '00' && imm5 != '00000' && InITBlock()`
LSRS (immediate)	T3 (MOVS, shift or rotate by value), A1 (MOVS, shift or rotate by value)	`S == '1' && stype == '01'`
LSRS (immediate)	T2	`op == '01' && !InITBlock()`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Alias	of variant	is preferred when
LSR (immediate)	T3 (MOV, shift or rotate by value), A1 (MOV, shift or rotate by value)	`S == '0' && stype == '01'`
LSR (immediate)	T2	`op == '01' && InITBlock()`
RORS (immediate)	T3 (MOVS, shift or rotate by value)	`S == '1' && imm3:Rd:imm2 != '000xxxx00' && stype == '11'`
RORS (immediate)	A1 (MOVS, shift or rotate by value)	`S == '1' && imm5 != '00000' && stype == '11'`
ROR (immediate)	T3 (MOV, shift or rotate by value)	`S == '0' && imm3:Rd:imm2 != '000xxxx00' && stype == '11'`
ROR (immediate)	A1 (MOV, shift or rotate by value)	`S == '0' && imm5 != '00000' && stype == '11'`
RRXS	T3 (MOVS, rotate right with extend)	`S == '1' && imm3 == '000' && imm2 == '00' && stype == '11'`
RRXS	A1 (MOVS, rotate right with extend)	`S == '1' && imm5 == '00000' && stype == '11'`
RRX	T3 (MOV, rotate right with extend)	`S == '0' && imm3 == '000' && imm2 == '00' && stype == '11'`
RRX	A1 (MOV, rotate right with extend)	`S == '0' && imm5 == '00000' && stype == '11'`

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If the PC is used:

- For the MOV variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953. Arm deprecates use of the instruction if `<Rn>` is the PC.
- For the MOVS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>. Arm deprecates use of the instruction if <Rn> is not the LR, or if the optional shift or `RRX` argument is specified.

For encoding T1: is the general-purpose destination register, encoded in the "D:Rd" field. If the PC is used:

- The instruction causes a branch to the address moved to the PC. This is a simple branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- The instruction must either be outside an IT block or the last instruction of an IT block.

For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` For encoding A1 and T1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used. Arm deprecates use of the instruction if `<Rd>` is the PC.

For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.

`<shift>` For encoding A1 and T3: is the type of shift to be applied to the source register, encoded in the "stype" field. It can have the following values:

`LSL`	when `stype = 00`
`LSR`	when `stype = 01`
`ASR`	when `stype = 10`
`ROR`	when `stype = 11`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For encoding T2: is the type of shift to be applied to the source register, encoded in the "op" field. It can have the following values:

`LSL` when `op = 00`

`LSR` when `op = 01`

`ASR` when `op = 10`

`<amount>` For encoding A1: is the shift amount, in the range 0 to 31 (when <shift> = LSL), or 1 to 31 (when <shift> = ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm5" field as <amount> modulo 32.

For encoding T3: is the shift amount, in the range 0 to 31 (when <shift> = LSL) or 1 to 31 (when <shift> = ROR), or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = shifted;
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.113 MOV, MOVS (register-shifted register)

Move (register-shifted register) copies a register-shifted register value to the destination register. It can optionally update the condition flags based on the value.

This instruction is used by the aliases ASRS (register), ASR (register), LSLS (register), LSL (register), LSRS (register), LSR (register), RORS (register), and ROR (register). See *Alias conditions* on page F5-4524 for details of when each alias is preferred.

A1

31–28	27–24	23–20		19–16	15–12	11–8	7	6–5	4	3–0
!=1111 (cond)	0 0 0 1	1 0 1	S	(0) (0) (0) (0)	Rd	Rs	0	stype	1	Rm

Flag setting variant

Applies when `S == 1`.

```
MOVS{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>
```

Not flag setting variant

Applies when `S == 0`.

```
MOV{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(stype);
if d == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

T1

15–14	13–12	11–10	9–6	5–3	2–0
0 1	0 0	0 0	0 x x x (op)	Rs	Rdm

Arithmetic shift right variant

Applies when `op == 0100`.

```
MOV<c>{<q>} <Rdm>, <Rdm>, ASR <Rs> // Inside IT block
MOVS{<q>} <Rdm>, <Rdm>, ASR <Rs> // Outside IT block
```

Logical shift left variant

Applies when `op == 0010`.

```
MOV<c>{<q>} <Rdm>, <Rdm>, LSL <Rs> // Inside IT block
MOVS{<q>} <Rdm>, <Rdm>, LSL <Rs> // Outside IT block
```

Logical shift right variant

Applies when `op == 0011`.

```
MOV<c>{<q>} <Rdm>, <Rdm>, LSR <Rs> // Inside IT block
MOVS{<q>} <Rdm>, <Rdm>, LSR <Rs> // Outside IT block
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Rotate right variant

Applies when `op == 0111`.

```
MOV<c>{<q>} <Rdm>, <Rdm>, ROR <Rs> // Inside IT block
MOVS{<q>} <Rdm>, <Rdm>, ROR <Rs> // Outside IT block
```

Decode for all variants of this encoding

```
if !(op IN {'0010', '0011', '0100', '0111'}) then SEE "Related encodings";
d = UInt(Rdm);  m = UInt(Rdm);  s = UInt(Rs);
setflags = !InITBlock();  shift_t = DecodeRegShift(op<2>:op<0>);
```

T2

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 14 13 12	11 8	7 6 5 4	3 0
1 1 1 1	1 0 1 0 0	stype	S	Rm	1 1 1 1	Rd	0 0 0 0	Rs

Flag setting variant

Applies when `S == 1`.

```
MOVS.W <Rd>, <Rm>, <shift> <Rs> // Outside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in
T1
MOVS{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>
```

Not flag setting variant

Applies when `S == 0`.

```
MOV<c>.W <Rd>, <Rm>, <shift> <Rs> // Inside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in
T1
MOV{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(stype);
if d == 15 || m == 15 || s == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

Related encodings: In encoding T1, for an `op` field value that is not described above, see *Data-processing (two low registers)* on page F3-4109.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Alias conditions

Alias	of variant	is preferred when
ASRS (register)	A1 (flag setting)	`S == '1' && stype == '10'`
ASRS (register)	T1 (arithmetic shift right)	`op == '0100' && !InITBlock()`
ASRS (register)	T2 (flag setting)	`stype == '10' && S == '1'`
ASR (register)	A1 (not flag setting)	`S == '0' && stype == '10'`
ASR (register)	T1 (arithmetic shift right)	`op == '0100' && InITBlock()`
ASR (register)	T2 (not flag setting)	`stype == '10' && S == '0'`
LSLS (register)	A1 (flag setting)	`S == '1' && stype == '00'`
LSLS (register)	T1 (logical shift left)	`op == '0010' && !InITBlock()`
LSLS (register)	T2 (flag setting)	`stype == '00' && S == '1'`
LSL (register)	A1 (not flag setting)	`S == '0' && stype == '00'`
LSL (register)	T1 (logical shift left)	`op == '0010' && InITBlock()`
LSL (register)	T2 (not flag setting)	`stype == '00' && S == '0'`
LSRS (register)	A1 (flag setting)	`S == '1' && stype == '01'`
LSRS (register)	T1 (logical shift right)	`op == '0011' && !InITBlock()`
LSRS (register)	T2 (flag setting)	`stype == '01' && S == '1'`
LSR (register)	A1 (not flag setting)	`S == '0' && stype == '01'`
LSR (register)	T1 (logical shift right)	`op == '0011' && InITBlock()`
LSR (register)	T2 (not flag setting)	`stype == '01' && S == '0'`
RORS (register)	A1 (flag setting)	`S == '1' && stype == '11'`
RORS (register)	T1 (rotate right)	`op == '0111' && !InITBlock()`
RORS (register)	T2 (flag setting)	`stype == '11' && S == '1'`
ROR (register)	A1 (not flag setting)	`S == '0' && stype == '11'`
ROR (register)	T1 (rotate right)	`op == '0111' && InITBlock()`
ROR (register)	T2 (not flag setting)	`stype == '11' && S == '0'`

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rdm>` Is the general-purpose source register and the destination register, encoded in the "Rdm" field.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` Is the general-purpose source register, encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

`LSL`	when `stype = 00`
`LSR`	when `stype = 01`
`ASR`	when `stype = 10`
`ROR`	when `stype = 11`

`<Rs>` Is the general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (result, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.114 MOVT

Move Top writes an immediate value to the top halfword of the destination register. It does not affect the contents of the bottom halfword.

A1

31–28	27 26 25 24	23	22 21 20	19–16	15–12	11–0
!=1111 (cond)	0 0 1 1 0	1	0 0	imm4	Rd	imm12

A1 variant

```
MOVT{<c>}{<q>} <Rd>, #<imm16>
```

Decode for this encoding

```
d = UInt(Rd);  imm16 = imm4:imm12;
if d == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11	10 9 8	7	6 5 4	3–0	15	14–12	11–8	7–0
1 1 1 1 0	i	1 0	1	1 0 0	imm4	0	imm3	Rd	imm8

T1 variant

```
MOVT{<c>}{<q>} <Rd>, #<imm16>
```

Decode for this encoding

```
d = UInt(Rd);  imm16 = imm4:i:imm3:imm8;
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<imm16>	For encoding A1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:imm12" field. For encoding T1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:i:imm3:imm8" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R[d]<31:16> = imm16;
    // R[d]<15:0> unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.115 MRC

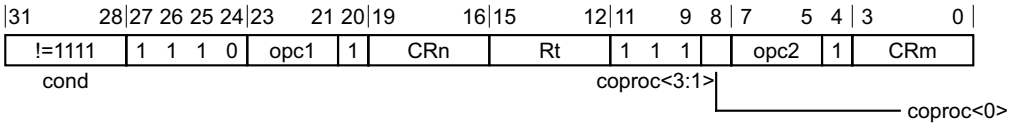
Move to general-purpose register from System register. This instruction copies the value of a System register to a general-purpose register.

The System register descriptions identify valid encodings for this instruction. Other encodings are UNDEFINED. For more information see *About the AArch32 System register interface* on page E1-3971 and *General behavior of System registers* on page G8-6096.

In an implementation that includes EL2, `MRC` accesses to system control registers can be trapped to Hyp mode, meaning that an attempt to execute an `MRC` instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see *EL2 configurable controls* on page G1-5787.

Because of the range of possible traps to Hyp mode, the `MRC` pseudocode does not show these possible traps.

A1



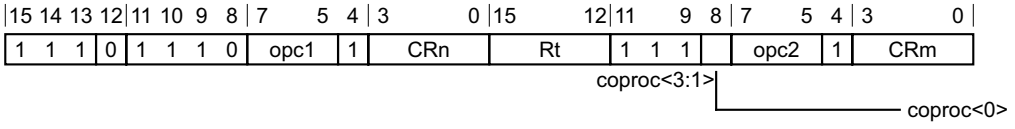
A1 variant

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

Decode for this encoding

```
t = UInt(Rt);  cp = if coproc<0> == '0' then 14 else 15;
// Armv8-A removes UNPREDICTABLE for R13
```

T1



T1 variant

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

Decode for this encoding

```
t = UInt(Rt);  cp = if coproc<0> == '0' then 14 else 15;
// Armv8-A removes UNPREDICTABLE for R13
```

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<coproc>` Is the System register encoding space, encoded in the "coproc<0>" field. It can have the following values:

- `p14` when `coproc<0> = 0`
- `p15` when `coproc<0> = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<opc1>` Is the opc1 parameter within the System register encoding space, in the range 0 to7, encoded in the "opc1" field.

`<Rt>` Is the general-purpose register to be transferred or `APSR_nzcv` (encoded as `0b1111`), encoded in the "Rt" field. If `APSR_nzcv` is used, bits [31:28] of the transferred value are written to the PSTATE condition flags.

`<CRn>` Is the CRn parameter within the System register encoding space, in the range c0 to c15, encoded in the "CRn" field.

`<CRm>` Is the CRm parameter within the System register encoding space, in the range c0 to c15, encoded in the "CRm" field.

`<opc2>` Is the opc2 parameter within the System register encoding space, in the range 0 to7, encoded in the "opc2" field.

The possible values of { `<coproc>`, `<opc1>`, `<CRn>`, `<CRm>`, `<opc2>` } encode the entire System register and System instruction encoding space. Not all of this space is allocated, and the System register and System instruction descriptions identify the allocated encodings.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) value = AArch32.SysRegRead(cp, ThisInstr());
    if t != 15 then
        R[t] = value;
    elsif AArch32.SysRegReadCanWriteAPSR(cp, ThisInstr()) then
        PSTATE.<N,Z,C,V> = value<31:28>;
        // value<27:0> are not used.
    else
        PSTATE.<N,Z,C,V> = bits(4) UNKNOWN;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.116 MRRC

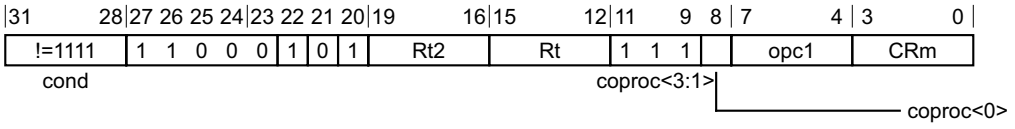
Move to two general-purpose registers from System register. This instruction copies the value of a System register to two general-purpose registers.

The System register descriptions identify valid encodings for this instruction. Other encodings are UNDEFINED. For more information see *About the AArch32 System register interface* on page E1-3971 and *General behavior of System registers* on page G8-6096.

In an implementation that includes EL2, `MRRC` accesses to System registers can be trapped to Hyp mode, meaning that an attempt to execute an `MRRC` instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see *EL2 configurable controls* on page G1-5787.

Because of the range of possible traps to Hyp mode, the `MRRC` pseudocode does not show these possible traps.

A1



A1 variant

```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

Decode for this encoding

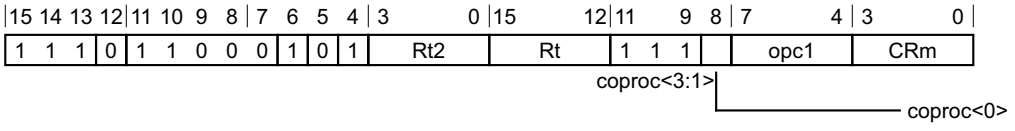
```
t = UInt(Rt);  t2 = UInt(Rt2);  cp = if coproc<0> == '0' then 14 else 15;
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

T1



T1 variant

```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

Decode for this encoding

```
t = UInt(Rt);  t2 = UInt(Rt2);  cp = if coproc<0> == '0' then 14 else 15;
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<coproc>` Is the System register encoding space, encoded in the "coproc<0>" field. It can have the following values:

`p14`	when `coproc<0> = 0`
`p15`	when `coproc<0> = 1`

`<opc1>` Is the opc1 parameter within the System register encoding space, in the range 0 to 15, encoded in the "opc1" field.

`<Rt>` Is the first general-purpose register that is transferred into, encoded in the "Rt" field.

`<Rt2>` Is the second general-purpose register that is transferred into, encoded in the "Rt2" field.

`<CRm>` Is the CRm parameter within the System register encoding space, in the range c0 to c15, encoded in the "CRm" field.

The possible values of { `<coproc>`, `<opc1>`, `<CRm>` } encode the entire System register encoding space. Not all of this space is allocated, and the System register descriptions identify the allocated encodings.

For the permitted uses of these instructions, as described in this manual, `<Rt2>` transfers bits[63:32] of the selected System register, while `<Rt>` transfers bits[31:0].

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    value = AArch32.SysRegRead64(cp, ThisInstr());
    R[t] = value<31:0>;
    R[t2] = value<63:32>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.117 MRS

Move Special register to general-purpose register moves the value of *The Application Program Status Register, APSR* on page E1-3955, CPSR, or SPSR_<current_mode> into a general-purpose register.

Arm recommends the `APSR` form when only the N, Z, C, V, Q, and GE[3:0] bits are being written. For more information, see *The Application Program Status Register, APSR* on page E1-3955.

An `MRS` that accesses the SPSR is UNPREDICTABLE if executed in User mode or System mode.

An `MRS` that is executed in User mode and accesses the CPSR returns an UNKNOWN value for the CPSR.{E, A, I, F, M} fields.

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3	2	1	0
!=1111	0	0	0	1	0	R	0	0	(1)	(1)	(1)	(1)	Rd	(0)	(0)	0	(0)	0	0	0	0	(0)	(0)	(0)	(0)
cond																									

A1 variant

```
MRS{<c>}{<q>} <Rd>, <spec_reg>
```

Decode for this encoding

```
d = UInt(Rd);  read_spsr = (R == '1');
if d == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11–8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	1	1	R	(1)	(1)	(1)	(1)	1	0	(0)	0	Rd	(0)	(0)	0	(0)	(0)	(0)	(0)	(0)

T1 variant

```
MRS{<c>}{<q>} <Rd>, <spec_reg>
```

Decode for this encoding

```
d = UInt(Rd);  read_spsr = (R == '1');
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<spec_reg>` Is the special register to be accessed, encoded in the "R" field. It can have the following values:

`CPSR|APSR` when `R = 0`

`SPSR` when `R = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if read_spsr then
        if PSTATE.M IN {M32_User,M32_System} then
            UNPREDICTABLE;
        else
            R[d] = SPSR[];
    else
        // CPSR has same bit assignments as SPSR, but with the IT, J, SS, IL, and T bits masked out.
        bits(32) mask = '11111000 00001111 00000011 11011111';
        if HavePANExt() then
            mask<22> = '1';

        if HaveDITExt() then
            mask<21> = '1';
        psr_val = GetPSRFromPSTATE() AND mask;
        if PSTATE.EL == EL0 then
            // If accessed from User mode return UNKNOWN values for E, A, I, F bits, bits<9:6>,
            // and for the M field, bits<4:0>
            psr_val<22> = bits(1) UNKNOWN;
            psr_val<9:6> = bits(4) UNKNOWN;
            psr_val<4:0> = bits(5) UNKNOWN;
        R[d] = psr_val;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.M IN {M32_User, M32_System} && read_spsr`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.118 MRS (Banked register)

Move to Register from Banked or Special register moves the value from the Banked general-purpose register or SPSR of the specified mode, or the value of *ELR_hyp* on page G1-5694, to a general-purpose register.

`MRS` (Banked register) is UNPREDICTABLE if executed in User mode.

When EL3 is using AArch64, if an MRS (Banked register) instruction that is executed in a Secure EL1 mode would access SPSR_mon, SP_mon, or LR_mon, it is trapped to EL3.

The effect of using an `MRS` (Banked register) instruction with a register argument that is not valid for the current mode is UNPREDICTABLE. For more information see *Usage restrictions on the banked register transfer instructions* on page F5-4952.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3	2	1	0
!=1111	0	0	0	1	0	R	0	0	M1	Rd	(0)	(0)	1	M	0	0	0	0	(0)	(0)	(0)	(0)
cond																						

A1 variant

```
MRS{<c>}{<q>} <Rd>, <banked_reg>
```

Decode for this encoding

```
d = UInt(Rd);  read_spsr = (R == '1');
if d == 15 then UNPREDICTABLE;
SYSm = M:M1;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15	14	13	12	11–8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	1	1	R	M1	1	0	(0)	0	Rd	(0)	(0)	1	M	(0)	(0)	(0)	(0)

T1 variant

```
MRS{<c>}{<q>} <Rd>, <banked_reg>
```

Decode for this encoding

```
d = UInt(Rd);  read_spsr = (R == '1');
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
SYSm = M:M1;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<banked_reg> Is the name of the banked register to be transferred to or from, encoded in the "R:M:M1" field. It can have the following values:

R8_usr	when R = 0, M = 0, M1 = 0000
R9_usr	when R = 0, M = 0, M1 = 0001
R10_usr	when R = 0, M = 0, M1 = 0010
R11_usr	when R = 0, M = 0, M1 = 0011
R12_usr	when R = 0, M = 0, M1 = 0100
SP_usr	when R = 0, M = 0, M1 = 0101
LR_usr	when R = 0, M = 0, M1 = 0110
R8_fiq	when R = 0, M = 0, M1 = 1000
R9_fiq	when R = 0, M = 0, M1 = 1001
R10_fiq	when R = 0, M = 0, M1 = 1010
R11_fiq	when R = 0, M = 0, M1 = 1011
R12_fiq	when R = 0, M = 0, M1 = 1100
SP_fiq	when R = 0, M = 0, M1 = 1101
LR_fiq	when R = 0, M = 0, M1 = 1110
LR_irq	when R = 0, M = 1, M1 = 0000
SP_irq	when R = 0, M = 1, M1 = 0001
LR_svc	when R = 0, M = 1, M1 = 0010
SP_svc	when R = 0, M = 1, M1 = 0011
LR_abt	when R = 0, M = 1, M1 = 0100
SP_abt	when R = 0, M = 1, M1 = 0101
LR_und	when R = 0, M = 1, M1 = 0110
SP_und	when R = 0, M = 1, M1 = 0111
LR_mon	when R = 0, M = 1, M1 = 1100
SP_mon	when R = 0, M = 1, M1 = 1101
ELR_hyp	when R = 0, M = 1, M1 = 1110
SP_hyp	when R = 0, M = 1, M1 = 1111
SPSR_fiq	when R = 1, M = 0, M1 = 1110
SPSR_irq	when R = 1, M = 1, M1 = 0000
SPSR_svc	when R = 1, M = 1, M1 = 0010
SPSR_abt	when R = 1, M = 1, M1 = 0100
SPSR_und	when R = 1, M = 1, M1 = 0110
SPSR_mon	when R = 1, M = 1, M1 = 1100
SPSR_hyp	when R = 1, M = 1, M1 = 1110

The following encodings are UNPREDICTABLE:

- R = 0, M = 0, M1 = 0111.
- R = 0, M = 0, M1 = 1111.
- R = 0, M = 1, M1 = 10xx.
- R = 1, M = 0, M1 = 0xxx.
- R = 1, M = 0, M1 = 10xx.
- R = 1, M = 0, M1 = 110x.
- R = 1, M = 0, M1 = 1111.
- R = 1, M = 1, M1 = 0001.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- R = 1, M = 1, M1 = 0011.
- R = 1, M = 1, M1 = 0101.
- R = 1, M = 1, M1 = 0111.
- R = 1, M = 1, M1 = 10xx.
- R = 1, M = 1, M1 = 1101.
- R = 1, M = 1, M1 = 1111.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL0 then
       UNPREDICTABLE;
    else
        mode = PSTATE.M;
        if read_spsr then
            SPSRaccessValid(SYSm, mode);           // Check for UNPREDICTABLE cases
            case SYSm of
                when '01110'  R[d] = SPSR_fiq;
                when '10000'  R[d] = SPSR_irq;
                when '10010'  R[d] = SPSR_svc;
                when '10100'  R[d] = SPSR_abt;
                when '10110'  R[d] = SPSR_und;
                when '11100'
                    if !ELUsingAArch32(EL3) then AArch64.MonitorModeTrap();
                    R[d] = SPSR_mon;
                when '11110'  R[d] = SPSR_hyp;
        else
            BankedRegisterAccessValid(SYSm, mode); // Check for UNPREDICTABLE cases
            case SYSm of
                when '00xxx'                       // Access the User mode registers
                    m = UInt(SYSm<2:0>) + 8;
                    R[d] = Rmode[m,M32_User];
                when '01xxx'                       // Access the FIQ mode registers
                    m = UInt(SYSm<2:0>) + 8;
                    R[d] = Rmode[m,M32_FIQ];
                when '1000x'                       // Access the IRQ mode registers
                    m = 14 - UInt(SYSm<0>);        // LR when SYSm<0> == 0, otherwise SP
                    R[d] = Rmode[m,M32_IRQ];
                when '1001x'                       // Access the Supervisor mode registers
                    m = 14 - UInt(SYSm<0>);        // LR when SYSm<0> == 0, otherwise SP
                    R[d] = Rmode[m,M32_Svc];
                when '1010x'                       // Access the Abort mode registers
                    m = 14 - UInt(SYSm<0>);        // LR when SYSm<0> == 0, otherwise SP
                    R[d] = Rmode[m,M32_Abort];
                when '1011x'                       // Access the Undefined mode registers
                    m = 14 - UInt(SYSm<0>);        // LR when SYSm<0> == 0, otherwise SP
                    R[d] = Rmode[m,M32_Undef];
                when '1110x'                       // Access Monitor registers
                    if !ELUsingAArch32(EL3) then AArch64.MonitorModeTrap();
                    m = 14 - UInt(SYSm<0>);        // LR when SYSm<0> == 0, otherwise SP
                    R[d] = Rmode[m,M32_Monitor];
                when '11110'                       // Access ELR_hyp register
                    R[d] = ELR_hyp;
                when '11111'                       // Access SP_hyp register
                    R[d] = Rmode[13,M32_Hyp];
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The instruction executes as `NOP`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.119 MSR (Banked register)

Move to Banked or Special register from general-purpose register moves the value of a general-purpose register to the Banked general-purpose register or SPSR of the specified mode, or to *ELR_hyp* on page G1-5694.

`MSR` (Banked register) is UNPREDICTABLE if executed in User mode.

When EL3 is using AArch64, if an MSR (Banked register) instruction that is executed in a Secure EL1 mode would access SPSR_mon, SP_mon, or LR_mon, it is trapped to EL3.

The effect of using an `MSR` (Banked register) instruction with a register argument that is not valid for the current mode is UNPREDICTABLE. For more information see *Usage restrictions on the banked register transfer instructions* on page F5-4952.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11	10	9	8	7	6	5	4	3–0
!=1111	0	0	0	1	0	R	1	0	M1	(1)	(1)	(1)	(1)	(0)	(0)	1	M	0	0	0	0	Rn
cond																						

A1 variant

```
MSR{<c>}{<q>} <banked_reg>, <Rn>
```

Decode for this encoding

```
n = UInt(Rn);  write_spsr = (R == '1');
if n == 15 then UNPREDICTABLE;
SYSm = M:M1;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15	14	13	12	11–8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	0	R	Rn	1	0	(0)	0	M1	(0)	(0)	1	M	(0)	(0)	(0)	(0)

T1 variant

```
MSR{<c>}{<q>} <banked_reg>, <Rn>
```

Decode for this encoding

```
n = UInt(Rn);  write_spsr = (R == '1');
if n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
SYSm = M:M1;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<banked_reg> Is the name of the banked register to be transferred to or from, encoded in the "R:M:M1" field. It can have the following values:

R8_usr	when R = 0, M = 0, M1 = 0000
R9_usr	when R = 0, M = 0, M1 = 0001
R10_usr	when R = 0, M = 0, M1 = 0010
R11_usr	when R = 0, M = 0, M1 = 0011
R12_usr	when R = 0, M = 0, M1 = 0100
SP_usr	when R = 0, M = 0, M1 = 0101
LR_usr	when R = 0, M = 0, M1 = 0110
R8_fiq	when R = 0, M = 0, M1 = 1000
R9_fiq	when R = 0, M = 0, M1 = 1001
R10_fiq	when R = 0, M = 0, M1 = 1010
R11_fiq	when R = 0, M = 0, M1 = 1011
R12_fiq	when R = 0, M = 0, M1 = 1100
SP_fiq	when R = 0, M = 0, M1 = 1101
LR_fiq	when R = 0, M = 0, M1 = 1110
LR_irq	when R = 0, M = 1, M1 = 0000
SP_irq	when R = 0, M = 1, M1 = 0001
LR_svc	when R = 0, M = 1, M1 = 0010
SP_svc	when R = 0, M = 1, M1 = 0011
LR_abt	when R = 0, M = 1, M1 = 0100
SP_abt	when R = 0, M = 1, M1 = 0101
LR_und	when R = 0, M = 1, M1 = 0110
SP_und	when R = 0, M = 1, M1 = 0111
LR_mon	when R = 0, M = 1, M1 = 1100
SP_mon	when R = 0, M = 1, M1 = 1101
ELR_hyp	when R = 0, M = 1, M1 = 1110
SP_hyp	when R = 0, M = 1, M1 = 1111
SPSR_fiq	when R = 1, M = 0, M1 = 1110
SPSR_irq	when R = 1, M = 1, M1 = 0000
SPSR_svc	when R = 1, M = 1, M1 = 0010
SPSR_abt	when R = 1, M = 1, M1 = 0100
SPSR_und	when R = 1, M = 1, M1 = 0110
SPSR_mon	when R = 1, M = 1, M1 = 1100
SPSR_hyp	when R = 1, M = 1, M1 = 1110

The following encodings are UNPREDICTABLE:

- R = 0, M = 0, M1 = 0111.
- R = 0, M = 0, M1 = 1111.
- R = 0, M = 1, M1 = 10xx.
- R = 1, M = 0, M1 = 0xxx.
- R = 1, M = 0, M1 = 10xx.
- R = 1, M = 0, M1 = 110x.
- R = 1, M = 0, M1 = 1111.
- R = 1, M = 1, M1 = 0001.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- R = 1, M = 1, M1 = 0011.
- R = 1, M = 1, M1 = 0101.
- R = 1, M = 1, M1 = 0111.
- R = 1, M = 1, M1 = 10xx.
- R = 1, M = 1, M1 = 1101.
- R = 1, M = 1, M1 = 1111.

<Rn> Is the general-purpose source register, encoded in the "Rn" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL0 then
        UNPREDICTABLE;
    else
        mode = PSTATE.M;
        if write_spsr then
            SPSRaccessValid(SYSm, mode);             // Check for UNPREDICTABLE cases
            case SYSm of
                when '01110'  SPSR_fiq = R[n];
                when '10000'  SPSR_irq = R[n];
                when '10010'  SPSR_svc = R[n];
                when '10100'  SPSR_abt = R[n];
                when '10110'  SPSR_und = R[n];
                when '11100'
                    if !ELUsingAArch32(EL3) then AArch64.MonitorModeTrap();
                    SPSR_mon = R[n];
                when '11110'  SPSR_hyp = R[n];
        else
            BankedRegisterAccessValid(SYSm, mode);  // Check for UNPREDICTABLE cases
            case SYSm of
                when '00xxx'                      // Access the User mode registers
                    m = UInt(SYSm<2:0>) + 8;
                    Rmode[m,M32_User] = R[n];
                when '01xxx'                      // Access the FIQ mode registers
                    m = UInt(SYSm<2:0>) + 8;
                    Rmode[m,M32_FIQ] = R[n];
                when '1000x'                      // Access the IRQ mode registers
                    m = 14 - UInt(SYSm<0>);       // LR when SYSm<0> == 0, otherwise SP
                    Rmode[m,M32_IRQ] = R[n];
                when '1001x'                      // Access the Supervisor mode registers
                    m = 14 - UInt(SYSm<0>);       // LR when SYSm<0> == 0, otherwise SP
                    Rmode[m,M32_Svc] = R[n];
                when '1010x'                      // Access the Abort mode registers
                    m = 14 - UInt(SYSm<0>);       // LR when SYSm<0> == 0, otherwise SP
                    Rmode[m,M32_Abort] = R[n];
                when '1011x'                      // Access the Undefined mode registers
                    m = 14 - UInt(SYSm<0>);       // LR when SYSm<0> == 0, otherwise SP
                    Rmode[m,M32_Undef] = R[n];
                when '1110x'                      // Access Monitor registers
                    if !ELUsingAArch32(EL3) then AArch64.MonitorModeTrap();
                    m = 14 - UInt(SYSm<0>);       // LR when SYSm<0> == 0, otherwise SP
                    Rmode[m,M32_Monitor] = R[n];
                when '11110'                      // Access ELR_hyp register
                    ELR_hyp = R[n];
                when '11111'                      // Access SP_hyp register
                    Rmode[13,M32_Hyp] = R[n];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.120 MSR (immediate)

Move immediate value to Special register moves selected bits of an immediate value to the corresponding bits in the *The Application Program Status Register, APSR* on page E1-3955, CPSR, or SPSR_<current_mode>.

Because of the Do-Not-Modify nature of its reserved bits, the immediate form of `MSR` is normally only useful at the Application level for writing to `APSR_nzcvq` (`CPSR_f`).

If an `MSR` (immediate) moves selected bits of an immediate value to the CPSR, the PE checks whether the value being written to PSTATE.M is legal. See *Illegal changes to PSTATE.M* on page G1-5699.

An `MSR` (immediate) executed in User mode:

- Is CONSTRAINED UNPREDICTABLE if it attempts to update the SPSR.
- Otherwise, does not update any CPSR field that is accessible only at EL1 or higher,

An `MSR` (immediate) executed in System mode is CONSTRAINED UNPREDICTABLE if it attempts to update the SPSR.

The CPSR.E bit is writable from any mode using an `MSR` instruction. Arm deprecates using this to change its value.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11–0
!=1111 (cond)	0	0	1	1	0	R	1	0	mask	(1)	(1)	(1)	(1)	imm12

A1 variant

Applies when `!(R == 0 && mask == 0000)`.

```
MSR{<c>}{<q>} <spec_reg>, #<imm>
```

Decode for this encoding

```
if mask == '0000' && R == '0' then SEE "Related encodings";
imm32 = A32ExpandImm(imm12);  write_spsr = (R == '1');
if mask == '0000' then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `mask == '0000' && R == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Related encodings: *Move Special Register and Hints (immediate)* on page F4-4195.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<spec_reg>` Is one of:

- `APSR_<bits>`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- `CPSR_<fields>`.
- `SPSR_<fields>`.

For CPSR and SPSR, <fields> is a sequence of one or more of the following:

`c` mask<0> = '1' to enable writing of bits<7:0> of the destination PSR.

`x` mask<1> = '1' to enable writing of bits<15:8> of the destination PSR.

`s` mask<2> = '1' to enable writing of bits<23:16> of the destination PSR.

`f` mask<3> = '1' to enable writing of bits<31:24> of the destination PSR.

For APSR, <bits> is one of `nzcvq`, `g`, or `nzcvqg`. These map to the following CPSR_<fields> values:

- `APSR_nzcvq` is the same as `CPSR_f` (mask== '1000').
- `APSR_g` is the same as `CPSR_s` (mask == '0100').
- `APSR_nzcvqg` is the same as `CPSR_fs` (mask == '1100').

Arm recommends the `APSR_<bits>` forms when only the N, Z, C, V, Q, and GE[3:0] bits are being written. For more information, see *The Application Program Status Register, APSR* on page E1-3955.

`<imm>` Is an immediate value. See *Modified immediate constants in A32 instructions* on page F2-4096 for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if write_spsr then
        if PSTATE.M IN {M32_User,M32_System} then
            UNPREDICTABLE;
        else
            SPSRWriteByInstr(imm32, mask);
    else
        // Attempts to change to an illegal mode will invoke the Illegal Execution state mechanism
        CPSRWriteByInstr(imm32, mask);
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.M IN {M32_User,M32_System} && write_spsr`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.121 MSR (register)

Move general-purpose register to Special register moves selected bits of a general-purpose register to the *The Application Program Status Register, APSR* on page E1-3955, CPSR or SPSR_<current_mode>.

Because of the Do-Not-Modify nature of its reserved bits, a read-modify-write sequence is normally required when the `MSR` instruction is being used at Application level and its destination is not `APSR_nzcvq` (`CPSR_f`).

If an `MSR` (register) moves selected bits of an immediate value to the CPSR, the PE checks whether the value being written to PSTATE.M is legal. See *Illegal changes to PSTATE.M* on page G1-5699.

An `MSR` (register) executed in User mode:

- Is UNPREDICTABLE if it attempts to update the SPSR.
- Otherwise, does not update any CPSR field that is accessible only at EL1 or higher.

An `MSR` (register) executed in System mode is UNPREDICTABLE if it attempts to update the SPSR.

The CPSR.E bit is writable from any mode using an `MSR` instruction. Arm deprecates using this to change its value.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11	10	9	8	7	6	5	4	3–0
!=1111	0	0	0	1	0	R	1	0	mask	(1)	(1)	(1)	(1)	(0)	(0)	0	(0)	0	0	0	0	Rn
cond																						

A1 variant

```
MSR{<c>}{<q>} <spec_reg>, <Rn>
```

Decode for this encoding

```
n = UInt(Rn);  write_spsr = (R == '1');
if mask == '0000' then UNPREDICTABLE;
if n == 15 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `mask == '0000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15	14	13	12	11–8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	0	R	Rn	1	0	(0)	0	mask	(0)	(0)	0	(0)	(0)	(0)	(0)	(0)

T1 variant

```
MSR{<c>}{<q>} <spec_reg>, <Rn>
```

Decode for this encoding

```
n = UInt(Rn);  write_spsr = (R == '1');
if mask == '0000' then UNPREDICTABLE;
if n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CONSTRAINED UNPREDICTABLE behavior

If `mask == '0000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<spec_reg>` Is one of:

- `APSR_<bits>`.
- `CPSR_<fields>`.
- `SPSR_<fields>`.

For CPSR and SPSR, <fields> is a sequence of one or more of the following:

`c` mask<0> = '1' to enable writing of bits<7:0> of the destination PSR.

`x` mask<1> = '1' to enable writing of bits<15:8> of the destination PSR.

`s` mask<2> = '1' to enable writing of bits<23:16> of the destination PSR.

`f` mask<3> = '1' to enable writing of bits<31:24> of the destination PSR.

For APSR, <bits> is one of `nzcvq`, `g`, or `nzcvqg`. These map to the following CPSR_<fields> values:

- `APSR_nzcvq` is the same as `CPSR_f` (mask== '1000').
- `APSR_g` is the same as `CPSR_s` (mask == '0100').
- `APSR_nzcvqg` is the same as `CPSR_fs` (mask == '1100').

Arm recommends the `APSR_<bits>` forms when only the N, Z, C, V, Q, and GE[3:0] bits are being written. For more information, see *The Application Program Status Register, APSR* on page E1-3955.

`<Rn>` Is the general-purpose source register, encoded in the "Rn" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if write_spsr then
        if PSTATE.M IN {M32_User,M32_System} then
            UNPREDICTABLE;
        else
            SPSRWriteByInstr(R[n], mask);
    else
        // Attempts to change to an illegal mode will invoke the Illegal Execution state mechanism
        CPSRWriteByInstr(R[n], mask);
```

CONSTRAINED UNPREDICTABLE behavior

If `write_spsr && PSTATE.M IN {M32_User,M32_System}`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The instruction executes as `NOP`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.122 MUL, MULS

Multiply multiplies two register values. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether the source register values are considered to be signed values or unsigned values.

Optionally, it can update the condition flags based on the result. In the T32 instruction set, this option is limited to only a few forms of the instruction. Use of this option adversely affects performance on many implementations.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15 14 13 12	11–8	7 6 5 4	3–0
!=1111 (cond)	0 0 0 0	0 0 0 S	Rd	(0) (0) (0) (0)	Rm	1 0 0 1	Rn

Flag setting variant

Applies when `S == 1`.

```
MULS{<c>}{<q>} <Rd>, <Rn>{, <Rm>}
```

Not flag setting variant

Applies when `S == 0`.

```
MUL{<c>}{<q>} <Rd>, <Rn>{, <Rm>}
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7 6 5	3 2 0
0 1 0 0 0 0	1 1 0 1	Rn	Rdm

T1 variant

```
MUL<c>{<q>} <Rdm>, <Rn>{, <Rdm>} // Inside IT block
MULS{<q>} <Rdm>, <Rn>{, <Rdm>} // Outside IT block
```

Decode for this encoding

```
d = UInt(Rdm);  n = UInt(Rn);  m = UInt(Rdm);  setflags = !InITBlock();
```

T2

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15 14 13 12	11–8	7 6	5 4	3–0
1 1 1 1	1 0 1 1	0 0 0	Rn	1 1 1 1	Rd	0 0	0 0	Rm

T2 variant

```
MUL<c>.W <Rd>, <Rn>{, <Rm>} // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
MUL{<c>}{<q>} <Rd>, <Rn>{, <Rm>}
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rdm>`	Is the second general-purpose source register holding the multiplier and the destination register, encoded in the "Rdm" field.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<Rn>`	Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
`<Rm>`	Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field. If omitted, `<Rd>` is used.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = SInt(R[n]);  // operand1 = UInt(R[n]) produces the same final results
    operand2 = SInt(R[m]);  // operand2 = UInt(R[m]) produces the same final results
    result = operand1 * operand2;
    R[d] = result<31:0>;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result<31:0>);
        // PSTATE.C, PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.123 MVN, MVNS (immediate)

Bitwise NOT (immediate) writes the bitwise inverse of an immediate value to the destination register.

If the destination register is not the PC, the MVNS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The MVN variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- The MVNS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
!=1111 (cond)	0	0	1	1	1	1	1	S	(0)(0)(0)(0)	Rd	imm12

MVN variant

Applies when `S == 0`.

```
MVN{<c>}{<q>} <Rd>, #<const>
```

MVNS variant

Applies when `S == 1`.

```
MVNS{<c>}{<q>} <Rd>, #<const>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  setflags = (S == '1');
(imm32, carry) = A32ExpandImm_C(imm12, PSTATE.C);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14–12	11–8	7–0
1	1	1	1	0	i	0	0	0	1	1	S	1	1	1	1	0	imm3	Rd	imm8

MVN variant

Applies when `S == 0`.

```
MVN{<c>}{<q>} <Rd>, #<const>
```

MVNS variant

Applies when `S == 1`.

```
MVNS{<c>}{<q>} <Rd>, #<const>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
d = UInt(Rd);  setflags = (S == '1');
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the MVN variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- For the MVNS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field.

<const> For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* on page F2-4096 for the range of values.

For encoding T1: an immediate value. See *Modified immediate constants in T32 instructions* on page F2-4095 for the range of values.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = NOT(imm32);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.124 MVN, MVNS (register)

Bitwise NOT (register) writes the bitwise inverse of a register value to the destination register.

If the destination register is not the PC, the MVNS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The MVN variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- The MVNS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

31–28	27 26 25 24	23 22 21 20	19 18 17 16	15–12	11–7	6 5	4	3–0
!=1111 (cond)	0 0 0 1	1 1 S	(0) (0) (0) (0)	Rd	imm5	stype	0	Rm

MVN, rotate right with extend variant

Applies when `S == 0 && imm5 == 00000 && stype == 11`.

```
MVN{<c>}{<q>} <Rd>, <Rm>, RRX
```

MVN, shift or rotate by value variant

Applies when `S == 0 && !(imm5 == 00000 && stype == 11)`.

```
MVN{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
```

MVNS, rotate right with extend variant

Applies when `S == 1 && imm5 == 00000 && stype == 11`.

```
MVNS{<c>}{<q>} <Rd>, <Rm>, RRX
```

MVNS, shift or rotate by value variant

Applies when `S == 1 && !(imm5 == 00000 && stype == 11)`.

```
MVNS{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```

T1

15 14 13 12	11 10	9 8	7 6 5	3 2 0
0 1 0 0	0 0	1 1	1 1	Rm	Rd

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1 variant

```
MVN<c>{<q>} <Rd>, <Rm> // Inside IT block
MVNS{<q>} <Rd>, <Rm> // Outside IT block
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14 12	11 8	7 6	5 4	3 0
1	1	1	0	1	0	1	0	0	1	1	S	1	1	1	1	(0)	imm3	Rd	imm2	stype	Rm

MVN, rotate right with extend variant

Applies when `S == 0 && imm3 == 000 && imm2 == 00 && stype == 11`.

```
MVN{<c>}{<q>} <Rd>, <Rm>, RRX
```

MVN, shift or rotate by value variant

Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11)`.

```
MVN<c>.W <Rd>, <Rm> // Inside IT block, and <Rd>, <Rm> can be represented in T1
MVN{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
```

MVNS, rotate right with extend variant

Applies when `S == 1 && imm3 == 000 && imm2 == 00 && stype == 11`.

```
MVNS{<c>}{<q>} <Rd>, <Rm>, RRX
```

MVNS, shift or rotate by value variant

Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11)`.

```
MVNS.W <Rd>, <Rm> // Outside IT block, and <Rd>, <Rm> can be represented in T1
MVNS{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the MVN variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- For the MVNS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1 and T2: is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the source register, encoded in the "stype" field. It can have the following values:

`LSL`	when `stype = 00`
`LSR`	when `stype = 01`
`ASR`	when `stype = 10`
`ROR`	when `stype = 11`

`<amount>` For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = NOT(shifted);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.125 MVN, MVNS (register-shifted register)

Bitwise NOT (register-shifted register) writes the bitwise inverse of a register-shifted register value to the destination register. It can optionally update the condition flags based on the result.

A1

31–28	27 26 25 24	23 22 21 20		19 18 17 16	15–12	11–8	7	6 5	4	3–0
!=1111 (cond)	0 0 0 1	1 1	S	(0) (0) (0) (0)	Rd	Rs	0	stype	1	Rm

Flag setting variant

Applies when `S == 1`.

```
MVNS{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>
```

Not flag setting variant

Applies when `S == 0`.

```
MVN{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(stype);
if d == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` Is the general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

- `LSL` when `stype = 00`
- `LSR` when `stype = 01`
- `ASR` when `stype = 10`
- `ROR` when `stype = 11`

`<Rs>` Is the general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
result = NOT(shifted);
R[d] = result;
if setflags then
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.126 NOP

No Operation does nothing. This instruction can be used for instruction alignment purposes.

Note

The timing effects of including a `NOP` instruction in a program are not guaranteed. It can increase execution time, leave it unchanged, or even reduce it. Therefore, `NOP` instructions are not suitable for timing loops.

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!=1111	0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	0	0	0	0
cond																												

A1 variant

```
NOP{<c>}{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0

T1 variant

```
NOP{<c>}{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	0	0	0

T2 variant

```
NOP{<c>}.W
```

Decode for this encoding

```
// No additional decoding required
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    // Do nothing
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.127 ORN, ORNS (immediate)

Bitwise OR NOT (immediate) performs a bitwise (inclusive) OR of a register value and the complement of an immediate value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

T1

15 14 13 12	11	10	9	8	7 6 5 4	3 0	15	14 12	11 8	7 0
1 1 1 1 0	i	0	0	0	1 1 S	!=1111	0	imm3	Rd	imm8
						Rn				

Flag setting variant

Applies when `S == 1`.

```
ORNS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

Not flag setting variant

Applies when `S == 0`.

```
ORN{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

Decode for all variants of this encoding

```
if Rn == '1111' then SEE "MVN (immediate)";
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`.

`<Rn>` Is the general-purpose source register, encoded in the "Rn" field.

`<const>` An immediate value. See *Modified immediate constants in T32 instructions* on page F2-4095 for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] OR NOT(imm32);
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.128 ORN, ORNS (register)

Bitwise OR NOT (register) performs a bitwise (inclusive) OR of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

T1

15 14 13 12	11 10 9	8	7 6 5	4	3 0	15 14 12	11 8	7 6 5 4	3 0
1 1 1 0	1 0 1	0	0 1 1	S	!=1111	(0) imm3	Rd	imm2 stype	Rm
					Rn				

ORN, rotate right with extend variant

Applies when `S == 0 && imm3 == 000 && imm2 == 00 && stype == 11`.

```
ORN{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

ORN, shift or rotate by value variant

Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11)`.

```
ORN{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

ORNS, rotate right with extend variant

Applies when `S == 1 && imm3 == 000 && imm2 == 00 && stype == 11`.

```
ORNS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

ORNS, shift or rotate by value variant

Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11)`.

```
ORNS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if Rn == '1111' then SEE "MVN (register)";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<Rn>`	Is the first general-purpose source register, encoded in the "Rn" field.
`<Rm>`	Is the second general-purpose source register, encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

`LSL`	when `stype = 00`
`LSR`	when `stype = 01`
`ASR`	when `stype = 10`
`ROR`	when `stype = 11`

`<amount>` Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] OR NOT(shifted);
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.129 ORR, ORRS (immediate)

Bitwise OR (immediate) performs a bitwise (inclusive) OR of a register value and an immediate value, and writes the result to the destination register.

If the destination register is not the PC, the ORRS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The ORR variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- The ORRS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–0
!=1111 (cond)	0 0 1 1 1	0 0 S	Rn	Rd	imm12

ORR variant

Applies when `S == 0`.

```
ORR{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

ORRS variant

Applies when `S == 1`.

```
ORRS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = A32ExpandImm_C(imm12, PSTATE.C);
```

T1

15 14 13 12	11	10	9	8	7 6 5 4	3–0	15	14–12	11–8	7–0
1 1 1 1 0	i	0	0	0	0 1 0 S	!=1111 (Rn)	0	imm3	Rd	imm8

ORR variant

Applies when `S == 0`.

```
ORR{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

ORRS variant

Applies when `S == 1`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
ORRS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

Decode for all variants of this encoding

```
if Rn == '1111' then SEE "MOV (immediate)";
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the ORR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- For the ORRS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`.

`<Rn>` For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

`<const>` For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* on page F2-4096 for the range of values.

For encoding T1: an immediate value. See *Modified immediate constants in T32 instructions* on page F2-4095 for the range of values.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] OR imm32;
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.130 ORR, ORRS (register)

Bitwise OR (register) performs a bitwise (inclusive) OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the ORRS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The ORR variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- The ORRS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–7	6 5	4	3–0
!=1111 (cond)	0 0 0 1 1	0 0 S	Rn	Rd	imm5	stype	0	Rm

ORR, rotate right with extend variant

Applies when `S == 0 && imm5 == 00000 && stype == 11`.

```
ORR{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

ORR, shift or rotate by value variant

Applies when `S == 0 && !(imm5 == 00000 && stype == 11)`.

```
ORR{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

ORRS, rotate right with extend variant

Applies when `S == 1 && imm5 == 00000 && stype == 11`.

```
ORRS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

ORRS, shift or rotate by value variant

Applies when `S == 1 && !(imm5 == 00000 && stype == 11)`.

```
ORRS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15	14	13	12	11	10	9	8	7	6	5	3	2	0
0	1	0	0	0	0	1	1	0	0	Rm		Rdn	

T1 variant

```
ORR<c>{<q>} {<Rdn>,} <Rdn>, <Rm> // Inside IT block
ORRS{<q>} {<Rdn>,} <Rdn>, <Rm> // Outside IT block
```

Decode for this encoding

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	0	15	14	12	11	8	7	6	5	4	3	0
1	1	1	0	1	0	1	0	0	1	0	S	!=1111		(0)	imm3		Rd		imm2		stype		Rm	
												Rn												

ORR, rotate right with extend variant

Applies when `S == 0 && imm3 == 000 && imm2 == 00 && stype == 11`.

```
ORR{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

ORR, shift or rotate by value variant

Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11)`.

```
ORR<c>.W {<Rd>,} <Rn>, <Rm> // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
ORR{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

ORRS, rotate right with extend variant

Applies when `S == 1 && imm3 == 000 && imm2 == 00 && stype == 11`.

```
ORRS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

ORRS, shift or rotate by value variant

Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11)`.

```
ORRS.W {<Rd>,} <Rn>, <Rm> // Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
ORRS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if Rn == '1111' then SEE "Related encodings";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Related encodings: *Data-processing (shifted register)* on page F3-4120

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rdn>` Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the ORR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- For the ORRS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`.

`<Rn>` For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T2: is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

`LSL`	when `stype = 00`
`LSR`	when `stype = 01`
`ASR`	when `stype = 10`
`ROR`	when `stype = 11`

`<amount>` For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

In T32 assembly:

- Outside an IT block, if `ORRS <Rd>, <Rn>, <Rd>` is written with `<Rd>` and `<Rn>` both in the range R0-R7, it is assembled using encoding T1 as though `ORRS <Rd>, <Rn>` had been written.
- Inside an IT block, if `ORR<c> <Rd>, <Rn>, <Rd>` is written with `<Rd>` and `<Rn>` both in the range R0-R7, it is assembled using encoding T1 as though `ORR<c> <Rd>, <Rn>` had been written.

To prevent either of these happening, use the `.W` qualifier.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] OR shifted;
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
else
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.131 ORR, ORRS (register-shifted register)

Bitwise OR (register-shifted register) performs a bitwise (inclusive) OR of a register value and a register-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

A1

31–28	27 26 25 24	23 22 21 20		19–16	15–12	11–8	7	6 5	4	3–0
!=1111	0 0 0 1	1 0 0	S	Rn	Rd	Rs	0	stype	1	Rm
cond										

Flag setting variant

Applies when `S == 1`.

```
ORRS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

Not flag setting variant

Applies when `S == 0`.

```
ORR{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

- `LSL` when `stype = 00`
- `LSR` when `stype = 01`
- `ASR` when `stype = 10`
- `ROR` when `stype = 11`

`<Rs>` Is the general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] OR shifted;
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.132 PKHBT, PKHTB

Pack Halfword combines one halfword of its first operand with the other halfword of its shifted second operand.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–7	6	5	4	3–0
!=1111	0 1 1 0	1 0 0 0	Rn	Rd	imm5	tb	0	1	Rm
cond									

PKHBT variant

Applies when `tb == 0`.

```
PKHBT{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, LSL #<imm>}
```

PKHTB variant

Applies when `tb == 1`.

```
PKHTB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ASR #<imm>}
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  tbform = (tb == '1');
(shift_t, shift_n) = DecodeImmShift(tb:'0', imm5);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15 14–12	11–8	7 6 5 4	3–0
1 1 1 0	1 0 1 0	1 1 0 0	Rn	(0) imm3	Rd	imm2 tb 0	Rm
		S				T	

PKHBT variant

Applies when `tb == 0`.

```
PKHBT{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, LSL #<imm>} // tbform == FALSE
```

PKHTB variant

Applies when `tb == 1`.

```
PKHTB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ASR #<imm>} // tbform == TRUE
```

Decode for all variants of this encoding

```
if S == '1' || T == '1' then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  tbform = (tb == '1');
(shift_t, shift_n) = DecodeImmShift(tb:'0', imm3:imm2);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<imm>` For encoding A1: the shift to apply to the value read from `<Rm>`, encoded in the "imm5" field. Is one of:

`omitted` No shift, encoded as `0b00000`.

`1-31` Left shift by specified number of bits, encoded as a binary number.

For encoding A1: the shift to apply to the value read from `<Rm>`, encoded in the "imm5" field. Is one of:

`omitted` Instruction is a pseudo-instruction and is assembled as though `PKHBT{<c>}{<q>} <Rd>, <Rm>, <Rn>` had been written.

`1-32` Arithmetic right shift by specified number of bits. A shift by 32 bits is encoded as `0b00000`. Other shift amounts are encoded as binary numbers.

Note

An assembler can permit `<imm>` = 0 to mean the same thing as omitting the shift, but this is not standard UAL and must not be used for disassembly.

For encoding T1: the shift to apply to the value read from `<Rm>`, encoded in the "imm3:imm2" field.

For `PKHBT`, it is one of:

`omitted` No shift, encoded as `0b00000`.

`1-31` Left shift by specified number of bits, encoded as a binary number.

For `PKHTB`, it is one of:

`omitted` Instruction is a pseudo-instruction and is assembled as though `PKHBT{<c>}{<q>} <Rd>, <Rm>, <Rn>` had been written.

`1-32` Arithmetic right shift by specified number of bits. A shift by 32 bits is encoded as `0b00000`. Other shift amounts are encoded as binary numbers.

Note

An assembler can permit `<imm>` = 0 to mean the same thing as omitting the shift, but this is not standard UAL and must not be used for disassembly.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = Shift(R[m], shift_t, shift_n, PSTATE.C);  // PSTATE.C ignored
    R[d]<15:0>  = if tbform then operand2<15:0> else R[n]<15:0>;
    R[d]<31:16> = if tbform then R[n]<31:16>    else operand2<31:16>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.133 PLD, PLDW (immediate)

Preload Data (immediate) signals the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into the data cache.

The `PLD` instruction signals that the likely memory access is a read, and the `PLDW` instruction signals that it is a write.

The effect of a `PLD` or `PLDW` instruction is IMPLEMENTATION DEFINED. For more information, see *Preloading caches* on page E2-4002.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15	14	13	12	11 0
1 1 1 1	0 1 0 1	U	R	0	1	!=1111	(1)	(1)	(1)	(1)	imm12
						Rn					

Preload read variant

Applies when `R == 1`.

```
PLD{<c>}{<q>} [<Rn> {, #{+/-}<imm>}]
```

Preload write variant

Applies when `R == 0`.

```
PLDW{<c>}{<q>} [<Rn> {, #{+/-}<imm>}]
```

Decode for all variants of this encoding

```
if Rn == '1111' then SEE "PLD (literal)";
n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');  is_pldw = (R == '0');
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 14 13 12	11 0
1 1 1 1	1 0 0 0	1	0	W	1	!=1111	1 1 1 1	imm12
						Rn		

Preload read variant

Applies when `W == 0`.

```
PLD{<c>}{<q>} [<Rn> {, #{+}<imm>}]
```

Preload write variant

Applies when `W == 1`.

```
PLDW{<c>}{<q>} [<Rn> {, #{+}<imm>}]
```

Decode for all variants of this encoding

```
if Rn == '1111' then SEE "PLD (literal)";
n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);  add = TRUE;  is_pldw = (W == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T2

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15	14	13	12	11	10	9	8	7 … 0
1	1	1	1	1	0	0	0	0	0	W	1	!=1111 (Rn)	1	1	1	1	1	1	0	0	imm8

Preload read variant

Applies when `W == 0`.

```
PLD{<c>}{<q>} [<Rn> {, #-<imm>}]
```

Preload write variant

Applies when `W == 1`.

```
PLDW{<c>}{<q>} [<Rn> {, #-<imm>}]
```

Decode for all variants of this encoding

```
if Rn == '1111' then SEE "PLD (literal)";
n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);  add = FALSE;  is_pldw = (W == '1');
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. Must be `AL` or omitted.

For encoding T1 and T2: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field. If the PC is used, see PLD (literal).

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`+` Specifies the offset is added to the base register.

`<imm>` For encoding A1: is the optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T1: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T2: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = if add then (R[n] + imm32) else (R[n] - imm32);
    if is_pldw then
        Hint_PreloadDataForWrite(address);
    else
        Hint_PreloadData(address);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.134 PLD (literal)

Preload Data (literal) signals the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into the data cache.

The effect of a PLD instruction is IMPLEMENTATION DEFINED. For more information, see *Preloading caches* on page E2-4002.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11			0
1	1	1	1	0	1	0	1	U	(1)	0	1	1	1	1	1	(1)	(1)	(1)	(1)	imm12			

A1 variant

```
PLD{<c>}{<q>} <label> // Normal form
PLD{<c>}{<q>} [PC, #{+/-}<imm>] // Alternative form
```

Decode for this encoding

```
imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11			0
1	1	1	1	1	0	0	0	U	0	(0)	1	1	1	1	1	1	1	1	1	imm12			

T1 variant

```
PLD{<c>}{<q>} <label> // Preferred syntax
PLD{<c>}{<q>} [PC, #{+/-}<imm>] // Alternative syntax
```

Decode for this encoding

```
imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-4080. Must be AL or omitted.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

<label> The label of the literal data item that is likely to be accessed in the near future. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. The offset must be in the range –4095 to 4095.

If the offset is zero or positive, imm32 is equal to the offset and add == TRUE.

If the offset is negative, imm32 is equal to minus the offset and add == FALSE.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`<imm>` For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

For encoding T1: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page F1-4037.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
    Hint_PreloadData(address);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.135 PLD, PLDW (register)

Preload Data (register) signals the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into the data cache.

The `PLD` instruction signals that the likely memory access is a read, and the `PLDW` instruction signals that it is a write.

The effect of a `PLD` or `PLDW` instruction is IMPLEMENTATION DEFINED. For more information, see *Preloading caches* on page E2-4002.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15	14	13	12	11 7	6 5	4	3 0
1 1 1 1	0 1 1 1	U	R	0	1	Rn	(1)	(1)	(1)	(1)	imm5	stype	0	Rm

Preload read, optional shift or rotate variant

Applies when `R == 1 && !(imm5 == 00000 && stype == 11)`.

```
PLD{<c>}{<q>} [<Rn>, {+/-}<Rm> {, <shift> #<amount>}]
```

Preload read, rotate right with extend variant

Applies when `R == 1 && imm5 == 00000 && stype == 11`.

```
PLD{<c>}{<q>} [<Rn>, {+/-}<Rm> , RRX]
```

Preload write, optional shift or rotate variant

Applies when `R == 0 && !(imm5 == 00000 && stype == 11)`.

```
PLDW{<c>}{<q>} [<Rn>, {+/-}<Rm> {, <shift> #<amount>}]
```

Preload write, rotate right with extend variant

Applies when `R == 0 && imm5 == 00000 && stype == 11`.

```
PLDW{<c>}{<q>} [<Rn>, {+/-}<Rm> , RRX]
```

Decode for all variants of this encoding

```
n = UInt(Rn);  m = UInt(Rm);  add = (U == '1');  is_pldw = (R == '0');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
if m == 15 || (n == 15 && is_pldw) then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 14 13 12	11 10 9 8	7 6 5 4	3 0
1 1 1 1	1 0 0 0	0 W 1	!=1111	1 1 1 1	0 0 0 0	0 0 imm2	Rm
			Rn				

Preload read variant

Applies when `W == 0`.

```
PLD{<c>}{<q>} [<Rn>, {+}<Rm> {, LSL #<amount>}]
```

Preload write variant

Applies when `W == 1`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
PLDW{<c>}{<q>} [<Rn>, {+}<Rm> {, LSL #<amount>}]
```

Decode for all variants of this encoding

```
if Rn == '1111' then SEE "PLD (literal)";
n = UInt(Rn);  m = UInt(Rm);  add = TRUE;  is_pldw = (W == '1');
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. `<c>` must be `AL` or omitted.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rn>` For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used.

For encoding T1: is the general-purpose base register, encoded in the "Rn" field.

`+/-` Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`+` Specifies the index register is added to the base register.

`<Rm>` Is the general-purpose index register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the index register, encoded in the "stype" field. It can have the following values:

- `LSL` when `stype = 00`
- `LSR` when `stype = 01`
- `ASR` when `stype = 10`
- `ROR` when `stype = 11`

`<amount>` For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T1: is the shift amount, in the range 0 to 3, defaulting to 0 and encoded in the "imm2" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    address = if add then (R[n] + offset) else (R[n] - offset);
    if is_pldw then
        Hint_PreloadDataForWrite(address);
    else
        Hint_PreloadData(address);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.136 PLI (immediate, literal)

Preload Instruction signals the memory system that instruction memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the instruction cache.

The effect of a `PLI` instruction is IMPLEMENTATION DEFINED. For more information, see *Preloading caches* on page E2-4002.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15	14	13	12	11 0
1 1 1 1	0 1 0	0	U	1	0	1	Rn	(1)	(1)	(1)	(1)	imm12

A1 variant

```
PLI{<c>}{<q>} [<Rn> {, #{+/-}<imm>}]
PLI{<c>}{<q>} <label> // Normal form
PLI{<c>}{<q>} [PC, #{+/-}<imm>] // Alternative form
```

Decode for this encoding

```
n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 14 13 12	11 0
1 1 1 1	1 0 0 1	1	0	0	1	!=1111	1 1 1 1	imm12
						Rn		

T1 variant

```
PLI{<c>}{<q>} [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then SEE "encoding T3";
n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);  add = TRUE;
```

T2

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 14 13 12	11 10 9 8	7 0
1 1 1 1	1 0 0 1	0	0	0	1	!=1111	1 1 1 1	1 1 0 0	imm8
						Rn			

T2 variant

```
PLI{<c>}{<q>} [<Rn> {, #-<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then SEE "encoding T3";
n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);  add = FALSE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11			0
1	1	1	1	1	0	0	1	U	0	0	1	1	1	1	1	1	1	1	1	imm12			

T3 variant

```
PLI{<c>}{<q>} <label> // Preferred syntax
PLI{<c>}{<q>} [PC, #{+/-}<imm>] // Alternative syntax
```

Decode for this encoding

```
n = 15;  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. Must be `AL` or omitted.

For encoding T1, T2 and T3: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<label>` The label of the instruction that is likely to be accessed in the near future. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. The offset must be in the range –4095 to 4095.

If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`.

If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`+` Specifies the offset is added to the base register.

`<imm>` For encoding A1: is the optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T1: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T2: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

For encoding T3: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

For the literal forms of the instruction, encoding T3 is used, or Rn is encoded as `0b1111` in encoding A1, to indicate that the PC is the base register.

The alternative literal syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page F1-4037.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = if n == 15 then Align(PC,4) else R[n];
    address = if add then (base + imm32) else (base - imm32);
    Hint_PreloadInstr(address);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.137 PLI (register)

Preload Instruction signals the memory system that instruction memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the instruction cache.

The effect of a PLI instruction is IMPLEMENTATION DEFINED. For more information, see *Preloading caches* on page E2-4002.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11–7	6	5	4	3–0
1	1	1	1	0	1	1	0	U	1	0	1	Rn	(1)	(1)	(1)	(1)	imm5	stype		0	Rm

Rotate right with extend variant

Applies when `imm5 == 00000 && stype == 11`.

```
PLI{<c>}{<q>} [<Rn>, {+/-}<Rm> , RRX]
```

Shift or rotate by value variant

Applies when `!(imm5 == 00000 && stype == 11)`.

```
PLI{<c>}{<q>} [<Rn>, {+/-}<Rm> {, <shift> #<amount>}]
```

Decode for all variants of this encoding

```
n = UInt(Rn);  m = UInt(Rm);  add = (U == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
if m == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15	14	13	12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	0	0	1	0	0	0	1	!=1111 (Rn)	1	1	1	1	0	0	0	0	0	0	imm2		Rm

T1 variant

```
PLI{<c>}{<q>} [<Rn>, {+}<Rm> {, LSL #<amount>}]
```

Decode for this encoding

```
if Rn == '1111' then SEE "PLI (immediate, literal)";
n = UInt(Rn);  m = UInt(Rm);  add = TRUE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-4080. <c> must be AL or omitted.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`+/-` Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`+` Specifies the index register is added to the base register.

`<Rm>` Is the general-purpose index register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the index register, encoded in the "stype" field. It can have the following values:

- `LSL` when `stype = 00`
- `LSR` when `stype = 01`
- `ASR` when `stype = 10`
- `ROR` when `stype = 11`

`<amount>` For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T1: is the shift amount, in the range 0 to 3, defaulting to 0 and encoded in the "imm2" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    address = if add then (R[n] + offset) else (R[n] - offset);
    Hint_PreloadInstr(address);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.138 POP

Pop Multiple Registers from Stack loads multiple general-purpose registers from the stack, loading from consecutive memory locations starting at the address in SP, and updates SP to point just above the loaded data.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC* on page F2-4086.

The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.

T1

15	14	13	12	11	10	9	8	7 … 0
1	0	1	1	1	1	0	P	register_list

T1 variant

```
POP{<c>}{<q>} <registers> // Preferred syntax
LDM{<c>}{<q>} SP!, <registers> // Alternate syntax
```

Decode for this encoding

```
registers = P:'0000000':register_list;   UnalignedAllowed = FALSE;
if BitCount(registers) < 1 then UNPREDICTABLE;
if registers<15> == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction targets an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<registers>` Is a list of one or more registers to be loaded, separated by commas and surrounded by { and }.

The registers in the list must be in the range R0-R7, encoded in the "register_list" field, and can optionally include the PC. If the PC is in the list, the "P" field is set to 1, otherwise this field defaults to 0.

If the PC is in the list, the instruction must be either outside any IT block, or the last instruction in an IT block.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = SP;
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = if UnalignedAllowed then MemU[address,4] else MemA[address,4];
            address = address + 4;
    if registers<15> == '1' then
        if UnalignedAllowed then
            if address<1:0> == '00' then
                LoadWritePC(MemU[address,4]);
            else
                UNPREDICTABLE;
        else
            LoadWritePC(MemA[address,4]);
    if registers<13> == '0' then SP = SP + 4*BitCount(registers);
    if registers<13> == '1' then SP = bits(32) UNKNOWN;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

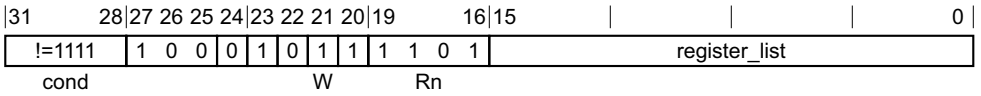
F5.1.139 POP (multiple registers)

Pop Multiple Registers from Stack loads multiple general-purpose registers from the stack, loading from consecutive memory locations starting at the address in SP, and updates SP to point just above the loaded data

This instruction is an alias of the LDM, LDMIA, LDMFD instruction. This means that:

- The encodings in this description are named to match the encodings of LDM, LDMIA, LDMFD.
- The description of LDM, LDMIA, LDMFD gives the operational pseudocode for this instruction.

A1



A1 variant

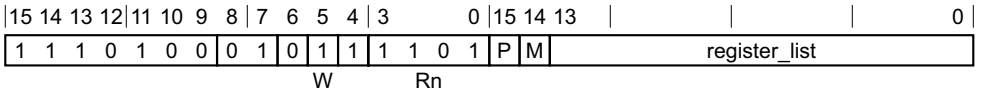
```
POP{<c>}{<q>} <registers>
```

is equivalent to

```
LDM{<c>}{<q>} SP!, <registers>
```

and is the preferred disassembly when `BitCount(register_list) > 1`.

T2



T2 variant

```
POP{<c>}.W <registers> // All registers in R0-R7, PC
```

is equivalent to

```
LDM{<c>}{<q>} SP!, <registers>
```

and is the preferred disassembly when `BitCount(P:M:register_list) > 1`.

```
POP{<c>}{<q>} <registers>
```

is equivalent to

```
LDM{<c>}{<q>} SP!, <registers>
```

and is the preferred disassembly when `BitCount(P:M:register_list) > 1`.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<registers>` For encoding A1: is a list of two or more registers to be loaded, separated by commas and surrounded by { and }. The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC* on page F2-4086.

If the SP is in the list, the value of the SP after such an instruction is UNKNOWN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The PC can be in the list. If it is, the instruction branches to the address loaded to the PC. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.

Arm deprecates the use of this instruction with both the LR and the PC in the list.

For encoding T2: is a list of two or more registers to be loaded, separated by commas and surrounded by { and }. The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC* on page F2-4086.

The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain one of the LR or the PC. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0. If the PC is in the list, the "P" field is set to 1, otherwise it defaults to 0.

The PC can be in the list. If it is, the instruction branches to the address loaded to the PC. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953. If the PC is in the list:

- The LR must not be in the list.
- The instruction must be either outside any IT block, or the last instruction in an IT block.

Operation for all encodings

The description of LDM, LDMIA, LDMFD gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.140 POP (single register)

Pop Single Register from Stack loads a single general-purpose register from the stack, loading from the address in SP, and updates SP to point just above the loaded data

This instruction is an alias of the LDR (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of LDR (immediate).
- The description of LDR (immediate) gives the operational pseudocode for this instruction.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11									1	0
!=1111	0	1	0	0	1	0	0	1	1 1 0 1	Rt	0	0	0	0	0	0	0	0	0	1	0	0
cond				P	U		W		Rn								imm12					

Post-indexed variant

```
POP{<c>}{<q>} <single_register_list>
```

is equivalent to

```
LDR{<c>}{<q>} <Rt>, [SP], #4
```

and is always the preferred disassembly.

T4

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7						1	0
1	1	1	1	1	0	0	0	0	1	0	1	1 1 0 1	Rt	1	0	1	1	0	0	0	0	0	1	0	0
												Rn			P	U	W	imm8							

Post-indexed variant

```
POP{<c>}{<q>} <single_register_list>
```

is equivalent to

```
LDR{<c>}{<q>} <Rt>, [SP], #4
```

and is always the preferred disassembly.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<single_register_list>`
Is the general-purpose register `<Rt>` to be loaded surrounded by { and }.

`<Rt>` For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For encoding T4: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.

Operation for all encodings

The description of LDR (immediate) gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.141 PSSBB

Physical Speculative Store Bypass Barrier is a memory barrier which prevents speculative loads from bypassing earlier stores to the same physical address.

The semantics of the Physical Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the PSSBB, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order before the PSSBB.
- When a load to a location appears in program order before the PSSBB, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order after the PSSBB.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	1	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	1	0	0	0	1	0	0

A1 variant

```
PSSBB{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	1	0	0	0	1	0	0

T1 variant

```
PSSBB{<q>}
```

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
```

Assembler symbols

<q> See *Standard assembler syntax fields* on page F2-4080.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    SpeculativeStoreBypassBarrierToPA();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.142 PUSH

Push Multiple Registers to Stack stores multiple general-purpose registers to the stack, storing to consecutive memory locations ending just below the address in SP, and updates SP to point to the start of the stored data.

The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address. See also *Encoding of lists of general-purpose registers and the PC* on page F2-4086.

T1

15 14 13 12	11	10	9	8	7 – 0
1 0 1 1	0	1	0	M	register_list

T1 variant

```
PUSH{<c>}{<q>} <registers> // Preferred syntax
STMDB{<c>}{<q>} SP!, <registers> // Alternate syntax
```

Decode for this encoding

```
registers = '0':M:'000000':register_list;  UnalignedAllowed = FALSE;
if BitCount(registers) < 1 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction targets an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<registers>` Is a list of one or more registers to be stored, separated by commas and surrounded by { and }.

The registers in the list must be in the range R0-R7, encoded in the "register_list" field, and can optionally include the LR. If the LR is in the list, the "M" field is set to 1, otherwise this field defaults to 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = SP - 4*BitCount(registers);
    for i = 0 to 14
        if registers<i> == '1' then
            if i == 13 && i != LowestSetBit(registers) then  // Only possible for encoding A1
                MemA[address,4] = bits(32) UNKNOWN;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        else
            if UnalignedAllowed then
                MemU[address,4] = R[i];
            else
                MemA[address,4] = R[i];
        address = address + 4;
if registers<15> == '1' then  // Only possible for encoding A1 or A2
    if UnalignedAllowed then
        MemU[address,4] = PCStoreValue();
    else
        MemA[address,4] = PCStoreValue();
SP = SP - 4*BitCount(registers);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.143 PUSH (multiple registers)

Push multiple registers to Stack stores multiple general-purpose registers to the stack, storing to consecutive memory locations ending just below the address in SP, and updates SP to point to the start of the stored data

This instruction is an alias of the STMDB, STMFD instruction. This means that:

- The encodings in this description are named to match the encodings of STMDB, STMFD.
- The description of STMDB, STMFD gives the operational pseudocode for this instruction.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–0
!=1111	1	0	0	1	0	0	1	0	1 1 0 1	register_list
cond							W		Rn	

A1 variant

```
PUSH{<c>}{<q>} <registers>
```

is equivalent to

```
STMDB{<c>}{<q>} SP!, <registers>
```

and is the preferred disassembly when `BitCount(register_list) > 1`.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15	14	13–0
1	1	1	0	1	0	0	1	0	0	1	0	1 1 0 1	(0)	M	register_list
										W		Rn	P		

T1 variant

```
PUSH{<c>}.W <registers> // All registers in R0-R7, LR
```

is equivalent to

```
STMDB{<c>}{<q>} SP!, <registers>
```

and is the preferred disassembly when `BitCount(M:register_list) > 1`.

```
PUSH{<c>}{<q>} <registers>
```

is equivalent to

```
STMDB{<c>}{<q>} SP!, <registers>
```

and is the preferred disassembly when `BitCount(M:register_list) > 1`.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<registers>` For encoding A1: is a list of two or more registers to be stored, separated by commas and surrounded by { and }. The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address. See also *Encoding of lists of general-purpose registers and the PC* on page F2-4086.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The SP and PC can be in the list. However:

- Arm deprecates the use of instructions that include the PC in the list.
- If the SP is in the list, and it is not the lowest-numbered register in the list, the instruction stores an UNKNOWN value for the SP.

For encoding T1: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address. See also *Encoding of lists of general-purpose registers and the PC* on page F2-4086.

The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain the LR. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0.

Operation for all encodings

The description of STMDB, STMFD gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.144 PUSH (single register)

Push Single Register to Stack stores a single general-purpose register to the stack, storing to the 32-bit word below the address in SP, and updates SP to point to the start of the stored data

This instruction is an alias of the STR (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of STR (immediate).
- The description of STR (immediate) gives the operational pseudocode for this instruction.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
!=1111	0	1	0	1	0	0	1	0	1 1 0 1	Rt	0 0 0 0 0 0 0 0 0 1 0 0
cond				P	U		W		Rn		imm12

Pre-indexed variant

```
PUSH{<c>}{<q>} <single_register_list>
```

is equivalent to

```
STR{<c>}{<q>} <Rt>, [SP, #-4]!
```

and is always the preferred disassembly.

T4

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7–0
1	1	1	1	1	0	0	0	0	1	0	0	1 1 0 1	Rt	1	1	0	1	0 0 0 0 0 1 0 0
												Rn			P	U	W	imm8

Pre-indexed variant

```
PUSH{<c>}{<q>} <single_register_list> // Standard syntax
```

is equivalent to

```
STR{<c>}{<q>} <Rt>, [SP, #-4]!
```

and is always the preferred disassembly.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<single_register_list>`
Is the general-purpose register `<Rt>` to be stored surrounded by { and }.

`<Rt>` For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.

For encoding T4: is the general-purpose register to be transferred, encoded in the "Rt" field.

Operation for all encodings

The description of STR (immediate) gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.145 QADD

Saturating Add adds two register values, saturates the result to the 32-bit signed integer range -2^{31} to $(2^{31} - 1)$, and writes the result to the destination register. If saturation occurs, it sets PSTATE.Q to 1.

A1

31–28	27–24	23–20	19–16	15–12	11–8	7–4	3–0
!=1111 (cond)	0 0 0 1 0	0 0 0	Rn	Rd	(0)(0)(0)(0)	0 1 0 1	Rm

A1 variant

```
QADD{<c>}{<q>} {<Rd>,} <Rm>, <Rn>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15–12	11–8	7–4	3–0	15–12	11–8	7–4	3–0
1 1 1 1	1 0 1 0	1 0 0 0	Rn	1 1 1 1	Rd	1 0 0 0	Rm

T1 variant

```
QADD{<c>}{<q>} {<Rd>,} <Rm>, <Rn>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	Is the first general-purpose source register, encoded in the "Rm" field.
<Rn>	Is the second general-purpose source register, encoded in the "Rn" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (R[d], sat) = SignedSatQ(SInt(R[m]) + SInt(R[n]), 32);
    if sat then
        PSTATE.Q = '1';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.146 QADD16

Saturating Add 16 performs two 16-bit integer additions, saturates the results to the 16-bit signed integer range $-2^{15} <= x <= 2^{15} - 1$, and writes the results to the destination register.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111 (cond)	0 1 1 0	0 1 0	Rn	Rd	(1)	(1)	(1)	(1)	0	0	0	1	Rm

A1 variant

```
QADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15 14 13 12	11–8	7	6	5	4	3–0
1 1 1 1	1 0 1 0	1	0 0 1	Rn	1 1 1 1	Rd	0	0	0	1	Rm

T1 variant

```
QADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<15:0>) + SInt(R[m]<15:0>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
sum2 = SInt(R[n]<31:16>) + SInt(R[m]<31:16>);
R[d]<15:0>  = SignedSat(sum1, 16);
R[d]<31:16> = SignedSat(sum2, 16);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.147 QADD8

Saturating Add 8 performs four 8-bit integer additions, saturates the results to the 8-bit signed integer range $-2^7 <= x <= 2^7 - 1$, and writes the results to the destination register.

A1

31–28	27–20	19–16	15–12	11–8	7–4	3–0
!=1111 (cond)	0 1 1 0 0 0 1 0	Rn	Rd	(1)(1)(1)(1)	1 0 0 1	Rm

A1 variant

```
QADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15–12	11–8	7–4	3–0	15–12	11–8	7–4	3–0
1 1 1 1	1 0 1 0	1	0 0 0	Rn	1 1 1 1	Rd	0 0 0 1	Rm

T1 variant

```
QADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<7:0>) + SInt(R[m]<7:0>);
    sum2 = SInt(R[n]<15:8>) + SInt(R[m]<15:8>);
    sum3 = SInt(R[n]<23:16>) + SInt(R[m]<23:16>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
sum4 = SInt(R[n]<31:24>) + SInt(R[m]<31:24>);
R[d]<7:0>   = SignedSat(sum1, 8);
R[d]<15:8>  = SignedSat(sum2, 8);
R[d]<23:16> = SignedSat(sum3, 8);
R[d]<31:24> = SignedSat(sum4, 8);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.148 QASX

Saturating Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer addition and one 16-bit subtraction, saturates the results to the 16-bit signed integer range $-2^{15} <= x <= 2^{15} - 1$, and writes the results to the destination register.

A1

31–28	27–20	19–16	15–12	11–8	7–4	3–0
!=1111 (cond)	0 1 1 0 0 0 1 0	Rn	Rd	(1)(1)(1)(1)	0 0 1 1	Rm

A1 variant

```
QASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15–12	11–8	7–4	3–0	15–12	11–8	7–4	3–0
1 1 1 1	1 0 1 0	1	0 1 0	Rn	1 1 1 1	Rd	0 0 0 1	Rm

T1 variant

```
QASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = SInt(R[n]<15:0>) - SInt(R[m]<31:16>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
sum  = SInt(R[n]<31:16>) + SInt(R[m]<15:0>);
R[d]<15:0>  = SignedSat(diff, 16);
R[d]<31:16> = SignedSat(sum, 16);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.149 QDADD

Saturating Double and Add adds a doubled register value to another register value, and writes the result to the destination register. Both the doubling and the addition have their results saturated to the 32-bit signed integer range $-2^{31} <= x <= 2^{31} - 1$. If saturation occurs in either operation, it sets PSTATE.Q to 1.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11 10 9 8	7 6 5 4	3–0
!=1111 (cond)	0 0 0 1 0	1 0 0	Rn	Rd	(0)(0)(0)(0)	0 1 0 1	Rm

A1 variant

```
QDADD{<c>}{<q>} {<Rd>,} <Rm>, <Rn>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15 14 13 12	11–8	7 6	5 4	3–0
1 1 1 1	1 0 1 0 1	0 0 0	Rn	1 1 1 1	Rd	1 0	0 1	Rm

T1 variant

```
QDADD{<c>}{<q>} {<Rd>,} <Rm>, <Rn>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	Is the first general-purpose source register, encoded in the "Rm" field.
<Rn>	Is the second general-purpose source register, encoded in the "Rn" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (doubled, sat1) = SignedSatQ(2 * SInt(R[n]), 32);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
(R[d], sat2)  = SignedSatQ(SInt(R[m]) + SInt(doubled), 32);
if sat1 || sat2 then
    PSTATE.Q = '1';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.150 QDSUB

Saturating Double and Subtract subtracts a doubled register value from another register value, and writes the result to the destination register. Both the doubling and the subtraction have their results saturated to the 32-bit signed integer range $-2^{31} <= x <= 2^{31} - 1$. If saturation occurs in either operation, it sets PSTATE.Q to 1.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11 10 9 8	7 6 5 4	3–0
!=1111 (cond)	0 0 0 1 0	1 1 0	Rn	Rd	(0)(0)(0)(0)	0 1 0 1	Rm

A1 variant

```
QDSUB{<c>}{<q>} {<Rd>,} <Rm>, <Rn>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15 14 13 12	11–8	7 6	5 4	3–0
1 1 1 1	1 0 1 0 1	0 0 0	Rn	1 1 1 1	Rd	1 0	1 1	Rm

T1 variant

```
QDSUB{<c>}{<q>} {<Rd>,} <Rm>, <Rn>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	Is the first general-purpose source register, encoded in the "Rm" field.
<Rn>	Is the second general-purpose source register, encoded in the "Rn" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (doubled, sat1) = SignedSatQ(2 * SInt(R[n]), 32);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
(R[d], sat2)  = SignedSatQ(SInt(R[m]) - SInt(doubled), 32);
if sat1 || sat2 then
    PSTATE.Q = '1';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.151 QSAX

Saturating Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer subtraction and one 16-bit addition, saturates the results to the 16-bit signed integer range $-2^{15} <= x <= 2^{15} - 1$, and writes the results to the destination register.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111 (cond)	0 1 1 0	0 1 0	Rn	Rd	(1)	(1)	(1)	(1)	0	1	0	1	Rm

A1 variant

```
QSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15 14 13 12	11–8	7	6	5	4	3–0
1 1 1 1	1 0 1 0 1	1 1 0	Rn	1 1 1 1	Rd	0	0	0	1	Rm

T1 variant

```
QSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = SInt(R[n]<15:0>) + SInt(R[m]<31:16>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
diff = SInt(R[n]<31:16>) - SInt(R[m]<15:0>);
R[d]<15:0>  = SignedSat(sum, 16);
R[d]<31:16> = SignedSat(diff, 16);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.152 QSUB

Saturating Subtract subtracts one register value from another register value, saturates the result to the 32-bit signed integer range $-2^{31} <= x <= 2^{31} - 1$, and writes the result to the destination register. If saturation occurs, it sets PSTATE.Q to 1.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11 10 9 8	7 6 5 4	3–0
!=1111 (cond)	0 0 0 1 0	0 1 0	Rn	Rd	(0)(0)(0)(0)	0 1 0 1	Rm

A1 variant

```
QSUB{<c>}{<q>} {<Rd>,} <Rm>, <Rn>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15 14 13 12	11–8	7 6 5 4	3–0
1 1 1 1	1 0 1 0 1	0 0 0	Rn	1 1 1 1	Rd	1 0 1 0	Rm

T1 variant

```
QSUB{<c>}{<q>} {<Rd>,} <Rm>, <Rn>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	Is the first general-purpose source register, encoded in the "Rm" field.
<Rn>	Is the second general-purpose source register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (R[d], sat) = SignedSatQ(SInt(R[m]) - SInt(R[n]), 32);
    if sat then
        PSTATE.Q = '1';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.153 QSUB16

Saturating Subtract 16 performs two 16-bit integer subtractions, saturates the results to the 16-bit signed integer range $-2^{15} <= x <= 2^{15} - 1$, and writes the results to the destination register.

A1

31–28	27–20	19–16	15–12	11–8	7–4	3–0
!=1111 (cond)	0 1 1 0 0 0 1 0	Rn	Rd	(1)(1)(1)(1)	0 1 1 1	Rm

A1 variant

```
QSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15–12	11–8	7–4	3–0	15–12	11–8	7–4	3–0
1 1 1 1	1 0 1 0	1 1 0 1	Rn	1 1 1 1	Rd	0 0 0 1	Rm

T1 variant

```
QSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<15:0>) - SInt(R[m]<15:0>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
diff2 = SInt(R[n]<31:16>) - SInt(R[m]<31:16>);
R[d]<15:0>  = SignedSat(diff1, 16);
R[d]<31:16> = SignedSat(diff2, 16);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.154 QSUB8

Saturating Subtract 8 performs four 8-bit integer subtractions, saturates the results to the 8-bit signed integer range $-2^7 <= x <= 2^7 - 1$, and writes the results to the destination register.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111 (cond)	0 1 1 0	0 1 0	Rn	Rd	(1)	(1)	(1)	(1)	1	1	1	1	Rm

A1 variant

```
QSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15 14 13 12	11–8	7	6	5	4	3–0
1 1 1 1	1 0 1 0	1 0 0	Rn	1 1 1 1	Rd	0	0	0	1	Rm

T1 variant

```
QSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<7:0>) - SInt(R[m]<7:0>);
    diff2 = SInt(R[n]<15:8>) - SInt(R[m]<15:8>);
    diff3 = SInt(R[n]<23:16>) - SInt(R[m]<23:16>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
diff4 = SInt(R[n]<31:24>) - SInt(R[m]<31:24>);
R[d]<7:0>   = SignedSat(diff1, 8);
R[d]<15:8>  = SignedSat(diff2, 8);
R[d]<23:16> = SignedSat(diff3, 8);
R[d]<31:24> = SignedSat(diff4, 8);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.155 RBIT

Reverse Bits reverses the bit order in a 32-bit register.

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	0	1	1	0	1	1	1	1	(1)	(1)	(1)	(1)	Rd	(1)	(1)	(1)	(1)	0	0	1	1	Rm
cond																						

A1 variant

```
RBIT{<c>}{<q>} <Rd>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15	14	13	12	11–8	7	6	5	4	3–0
1	1	1	1	1	0	1	0	1	0	0	1	Rn	1	1	1	1	Rd	1	0	1	0	Rm

T1 variant

```
RBIT{<c>}{<q>} <Rd>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);  n = UInt(Rn);
if m != n || d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `m != n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: m = UInt(Rn);.
- The instruction executes with the additional decode: m = UInt(Rm);.
- The value in the destination register is UNKNOWN.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Rm>` For encoding A1: is the general-purpose source register, encoded in the "Rm" field.

For encoding T1: is the general-purpose source register, encoded in the "Rm" field. It must be encoded with an identical value in the "Rn" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    for i = 0 to 31
        result<31-i> = R[m]<i>;
    R[d] = result;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.156 REV

Byte-Reverse Word reverses the byte order in a 32-bit register.

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	0	1	1	0	1	0	1	1	(1)	(1)	(1)	(1)	Rd	(1)	(1)	(1)	(1)	0	0	1	1	Rm
cond																						

A1 variant

```
REV{<c>}{<q>} <Rd>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5–3	2–0
1	0	1	1	1	0	1	0	0	0	Rm	Rd

T1 variant

```
REV{<c>}{<q>} <Rd>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15	14	13	12	11–8	7	6	5	4	3–0
1	1	1	1	1	0	1	0	1	0	0	1	Rn	1	1	1	1	Rd	1	0	0	0	Rm

T2 variant

```
REV{<c>}.W <Rd>, <Rm> // <Rd>, <Rm> can be represented in T1
REV{<c>}{<q>} <Rd>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);  n = UInt(Rn);
if m != n || d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `m != n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: m = UInt(Rn);.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The instruction executes with the additional decode: m = UInt(Rm);.
- The value in the destination register is UNKNOWN.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` For encoding A1 and T1: is the general-purpose source register, encoded in the "Rm" field.

For encoding T2: is the general-purpose source register, encoded in the "Rm" field. It must be encoded with an identical value in the "Rn" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    result<31:24> = R[m]<7:0>;
    result<23:16> = R[m]<15:8>;
    result<15:8>  = R[m]<23:16>;
    result<7:0>   = R[m]<31:24>;
    R[d] = result;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.157 REV16

Byte-Reverse Packed Halfword reverses the byte order in each16-bit halfword of a 32-bit register.

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111 (cond)	0	1	1	0	1	0	1	1	(1)	(1)	(1)	(1)	Rd	(1)	(1)	(1)	(1)	1	0	1	1	Rm

A1 variant

```
REV16{<c>}{<q>} <Rd>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5–3	2–0
1	0	1	1	1	0	1	0	0	1	Rm	Rd

T1 variant

```
REV16{<c>}{<q>} <Rd>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15	14	13	12	11–8	7	6	5	4	3–0
1	1	1	1	1	0	1	0	1	0	0	1	Rn	1	1	1	1	Rd	1	0	0	1	Rm

T2 variant

```
REV16{<c>}.W <Rd>, <Rm> // <Rd>, <Rm> can be represented in T1
REV16{<c>}{<q>} <Rd>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);  n = UInt(Rn);
if m != n || d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `m != n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: m = UInt(Rn);.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The instruction executes with the additional decode: m = UInt(Rm);.
- The value in the destination register is UNKNOWN.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` For encoding A1 and T1: is the general-purpose source register, encoded in the "Rm" field.

For encoding T2: is the general-purpose source register, encoded in the "Rm" field. It must be encoded with an identical value in the "Rn" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    result<31:24> = R[m]<23:16>;
    result<23:16> = R[m]<31:24>;
    result<15:8>  = R[m]<7:0>;
    result<7:0>   = R[m]<15:8>;
    R[d] = result;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.158 REVSH

Byte-Reverse Signed Halfword reverses the byte order in the lower 16-bit halfword of a 32-bit register, and sign-extends the result to 32 bits.

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111 (cond)	0	1	1	0	1	1	1	1	(1)	(1)	(1)	(1)	Rd	(1)	(1)	(1)	(1)	1	0	1	1	Rm

A1 variant

```
REVSH{<c>}{<q>} <Rd>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5–3	2–0
1	0	1	1	1	0	1	0	1	1	Rm	Rd

T1 variant

```
REVSH{<c>}{<q>} <Rd>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15	14	13	12	11–8	7	6	5	4	3–0
1	1	1	1	1	0	1	0	1	0	0	1	Rn	1	1	1	1	Rd	1	0	1	1	Rm

T2 variant

```
REVSH{<c>}.W <Rd>, <Rm> // <Rd>, <Rm> can be represented in T1
REVSH{<c>}{<q>} <Rd>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);  n = UInt(Rn);
if m != n || d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `m != n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The instruction executes with the additional decode: m = UInt(Rn);.
- The instruction executes with the additional decode: m = UInt(Rm);.
- The value in the destination register is UNKNOWN.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` For encoding A1 and T1: is the general-purpose source register, encoded in the "Rm" field.

For encoding T2: is the general-purpose source register, encoded in the "Rm" field. It must be encoded with an identical value in the "Rn" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    result<31:8>  = SignExtend(R[m]<7:0>, 24);
    result<7:0>   = R[m]<15:8>;
    R[d] = result;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.159 RFE, RFEDA, RFEDB, RFEIA, RFEIB

Return From Exception loads two consecutive memory locations using an address in a base register:

- The word loaded from the lower address is treated as an instruction address. The PE branches to it.
- The word loaded from the higher address is used to restore PSTATE. This word must be in the format of an SPSR.

An address adjusted by the size of the data loaded can optionally be written back to the base register.

The PE checks the value of the word loaded from the higher address for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.

`RFE` is UNDEFINED in Hyp mode and CONSTRAINED UNPREDICTABLE in User mode.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	P	U	0	W	1	Rn	(0)	(0)	(0)	(0)	(1)	(0)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

Decrement After variant

Applies when `P == 0 && U == 0`.

```
RFEDA{<c>}{<q>} <Rn>{!}
```

Decrement Before variant

Applies when `P == 1 && U == 0`.

```
RFEDB{<c>}{<q>} <Rn>{!}
```

Increment After variant

Applies when `P == 0 && U == 1`.

```
RFE{IA}{<c>}{<q>} <Rn>{!}
```

Increment Before variant

Applies when `P == 1 && U == 1`.

```
RFEIB{<c>}{<q>} <Rn>{!}
```

Decode for all variants of this encoding

```
n = UInt(Rn);
wback = (W == '1');  increment = (U == '1');  wordhigher = (P == U);
if n == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	0	0	W	1	Rn	(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

T1 variant

```
RFEDB{<c>}{<q>} <Rn>{!} // Outside or last in IT block
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
n = UInt(Rn);  wback = (W == '1');  increment = FALSE;  wordhigher = FALSE;
if n == 15 then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	1	1	0	W	1	Rn	(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

T2 variant

```
RFE{IA}{<c>}{<q>} <Rn>{!} // Outside or last in IT block
```

Decode for this encoding

```
n = UInt(Rn);  wback = (W == '1');  increment = TRUE;  wordhigher = FALSE;
if n == 15 then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

IA	For encoding A1: is an optional suffix to indicate the Increment After variant. For encoding T2: is an optional suffix for the Increment After form.
<c>	For encoding A1: see *Standard assembler syntax fields* on page F2-4080. <c> must be AL or omitted. For encoding T1 and T2: see *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.
!	The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

RFEFA, RFEEA, RFEFD, and RFEED are pseudo-instructions for RFEDA, RFEDB, RFEIA, and RFEIB respectively, referring to their use for popping data from Full Ascending, Empty Ascending, Full Descending, and Empty Descending stacks.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then
        UNDEFINED;
    elsif PSTATE.EL == EL0 then
        UNPREDICTABLE;                        // UNDEFINED or NOP
    else
        address = if increment then R[n] else R[n]-8;
        if wordhigher then address = address+4;
        new_pc_value = MemA[address,4];
        spsr = MemA[address+4,4];
        if wback then R[n] = if increment then R[n]+8 else R[n]-8;
        AArch32.ExceptionReturn(new_pc_value, spsr);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.160 ROR (immediate)

Rotate Right (immediate) provides the value of the contents of a register rotated by a constant value. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

This instruction is an alias of the MOV, MOVS (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

A1

31–28	27–25	24–23	22–21	20	19–16	15–12	11–7	6–5	4	3–0
!=1111	0 0 0	1 1	0 1	0	(0) (0) (0) (0)	Rd	!=00000	1 1	0	Rm
cond				S			imm5	stype		

MOV, shift or rotate by value variant

```
ROR{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ROR #<imm>
```

and is always the preferred disassembly.

T3

15–12	11–8	7–4	3–0	15–12	11–8	7–6	5–4	3–0
1 1 1 0	1 0 1 0	0 0 1 0	0	1 1 1 1	(0) imm3	Rd	imm2	1 1	Rm
			S					stype	

MOV, shift or rotate by value variant

Applies when `!(imm3 == 000 && imm2 == 00)`.

```
ROR{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ROR #<imm>
```

and is always the preferred disassembly.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.

For encoding T3: is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T3: is the general-purpose source register, encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<imm>` For encoding A1: is the shift amount, in the range 1 to 31, encoded in the "imm5" field.

For encoding T3: is the shift amount, in the range 1 to 31, encoded in the "imm3:imm2" field.

Operation for all encodings

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.161 ROR (register)

Rotate Right (register) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The variable number of bits is read from the bottom byte of a register

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

A1

31–28	27 26 25 24	23 22 21 20	19 18 17 16	15–12	11–8	7	6 5	4	3–0
!=1111	0 0 0 1	1 0 1 0	(0) (0) (0) (0)	Rd	Rs	0	1 1	1	Rm
cond		S				stype			

Not flag setting variant

```
ROR{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>
```

and is always the preferred disassembly.

T1

15 14 13 12	11 10	9–6	5–3	2–0
0 1 0 0	0 0	0 1 1 1	Rs	Rdm
		op		

Rotate right variant

```
ROR<c>{<q>} {<Rdm>,} <Rdm>, <Rs> // Inside IT block
```

is equivalent to

```
MOV<c>{<q>} <Rdm>, <Rdm>, ROR <Rs>
```

and is the preferred disassembly when `InITBlock()`.

T2

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15 14 13 12	11–8	7 6 5 4	3–0
1 1 1 1	1 0 1 0	0 1 1	0	Rm	1 1 1 1	Rd	0 0 0 0	Rs
		stype	S				

Not flag setting variant

```
ROR<c>.W {<Rd>,} <Rm>, <Rs> // Inside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

and is always the preferred disassembly.

```
ROR{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>
```

and is always the preferred disassembly.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rdm>`	Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<Rm>`	Is the first general-purpose source register, encoded in the "Rm" field.
`<Rs>`	Is the second general-purpose source register holding a rotate amount in its bottom 8 bits, encoded in the "Rs" field.

Operation for all encodings

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.162 RORS (immediate)

Rotate Right, setting flags (immediate) provides the value of the contents of a register rotated by a constant value. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

If the destination register is not the PC, this instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This instruction is an alias of the MOV, MOVS (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

A1

31–28	27 26 25 24	23 22 21 20	19 18 17 16	15–12	11–7	6 5	4	3–0
!=1111	0 0 0 1 1	0 1 1	(0) (0) (0) (0)	Rd	!=00000	1 1	0	Rm
cond		S			imm5	stype		

MOVS, shift or rotate by value variant

```
RORS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ROR #<imm>
```

and is always the preferred disassembly.

T3

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 14	12–11	8–7	6 5	4	3–0
1 1 1 0	1 0 1 0	0 0 1 0	1 1 1 1 1	(0)	imm3	Rd	imm2	1 1	Rm
		S						stype	

MOVS, shift or rotate by value variant

Applies when `!(imm3 == 000 && imm2 == 00)`.

```
RORS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ROR #<imm>
```

and is always the preferred disassembly.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T3: is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T3: is the general-purpose source register, encoded in the "Rm" field.

`<imm>` For encoding A1: is the shift amount, in the range 1 to 31, encoded in the "imm5" field.

For encoding T3: is the shift amount, in the range 1 to 31, encoded in the "imm3:imm2" field.

Operation for all encodings

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.163 RORS (register)

Rotate Right, setting flags (register) provides the value of the contents of a register rotated by a variable number of bits, and updates the condition flags based on the result. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The variable number of bits is read from the bottom byte of a register

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

A1

31–28	27–20								19–16	15–12	11–8	7	6–5	4	3–0
!=1111	0	0	0	1	1	0	1	1	(0)(0)(0)(0)	Rd	Rs	0	1 1	1	Rm
cond								S					stype		

Flag setting variant

```
RORS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>
```

and is always the preferred disassembly.

T1

15–10	9–6	5–3	2–0
0 1 0 0 0 0	0 1 1 1	Rs	Rdm
	op		

Rotate right variant

```
RORS{<q>} {<Rdm>,} <Rdm>, <Rs> // Outside IT block
```

is equivalent to

```
MOVS{<q>} <Rdm>, <Rdm>, ROR <Rs>
```

and is the preferred disassembly when `!InITBlock()`.

T2

15–11	10–9	8	7–6	5	4	3–0	15–12	11–8	7–4	3–0
1 1 1 1 1	0 1	0 0	1 1		1	Rm	1 1 1 1	Rd	0 0 0 0	Rs
			stype		S					

Flag setting variant

```
RORS.W {<Rd>,} <Rm>, <Rs> // Outside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

and is always the preferred disassembly.

```
RORS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>
```

and is always the preferred disassembly.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rdm>`	Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<Rm>`	Is the first general-purpose source register, encoded in the "Rm" field.
`<Rs>`	Is the second general-purpose source register holding a rotate amount in its bottom 8 bits, encoded in the "Rs" field.

Operation for all encodings

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.164 RRX

Rotate Right with Extend provides the value of the contents of a register shifted right by one place, with the Carry flag shifted into bit[31].

This instruction is an alias of the MOV, MOVS (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

A1

31–28	27–25	24–23	22–21	20	19–16	15–12	11–7	6–5	4	3–0
!=1111	0 0 0	1 1	0 1	0	(0) (0) (0) (0)	Rd	0 0 0 0 0	1 1	0	Rm
cond				S			imm5	stype		

MOV, rotate right with extend variant

```
RRX{<c>}{<q>} {<Rd>,} <Rm>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, RRX
```

and is always the preferred disassembly.

T3

15–12	11–10	9–8	7–6	5–4	3–0	15	14–12	11–8	7–6	5–4	3–0
1 1 1 0	1 0	1	0 0	1 0	0	1 1 1 1	(0)	0 0 0	Rd	0 0	1 1	Rm
				S				imm3		imm2	stype	

MOV, rotate right with extend variant

```
RRX{<c>}{<q>} {<Rd>,} <Rm>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, RRX
```

and is always the preferred disassembly.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.

For encoding T3: is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T3: is the general-purpose source register, encoded in the "Rm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.165 RRXS

Rotate Right with Extend, setting flags provides the value of the contents of a register shifted right by one place, with the Carry flag shifted into bit[31].

If the destination register is not the PC, this instruction updates the condition flags based on the result, and bit[0] is shifted into the Carry flag.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This instruction is an alias of the MOV, MOVS (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

A1

31–28	27 26 25 24	23 22 21 20	19 18 17 16	15–12	11–7	7 6	5	4	3–0
!=1111	0 0 0 1 1	0 1 1	(0) (0) (0) (0)	Rd	0 0 0 0 0	1 1	0		Rm
cond		S			imm5	stype			

MOVS, rotate right with extend variant

```
RRXS{<c>}{<q>} {<Rd>,} <Rm>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, RRX
```

and is always the preferred disassembly.

T3

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15	14–12	11–8	7 6	5	4	3–0
1 1 1 0	1 0 1 0	0 0 1 0	1 1 1 1	(0)	0 0 0	Rd	0 0	1	1	Rm
		S			imm3		imm2	stype		

MOVS, rotate right with extend variant

```
RRXS{<c>}{<q>} {<Rd>,} <Rm>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, RRX
```

and is always the preferred disassembly.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T3: is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T3: is the general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.166 RSB, RSBS (immediate)

Reverse Subtract (immediate) subtracts a register value from an immediate value, and writes the result to the destination register.

If the destination register is not the PC, the RSBS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The RSB variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- The RSBS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–0
!=1111 (cond)	0 0 1 0	0 1 1 S	Rn	Rd	imm12

RSB variant

Applies when `S == 0`.

```
RSB{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

RSBS variant

Applies when `S == 1`.

```
RSBS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = A32ExpandImm(imm12);
```

T1

15 14 13 12	11 10 9 8	7 6 5	3 2 0
0 1 0 0 0 0	1 0 0 1	Rn	Rd

T1 variant

```
RSB<c>{<q>} {<Rd>, }<Rn>, #0 // Inside IT block
RSBS{<q>} {<Rd>, }<Rn>, #0 // Outside IT block
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = Zeros(32); // immediate = #0
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T2

15 14 13 12	11	10	9	8	7	6	5	4	3 … 0	15	14 … 12	11 … 8	7 … 0
1 1 1 1 0	i	0	1	1	1	0	S	Rn	0	imm3	Rd	imm8

RSB variant

Applies when `S == 0`.

```
RSB<c>.W {<Rd>,} <Rn>, #0 // Inside IT block
RSB{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

RSBS variant

Applies when `S == 1`.

```
RSBS.W {<Rd>,} <Rn>, #0 // Outside IT block
RSBS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = T32ExpandImm(i:imm3:imm8);
if d == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the RSB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- For the RSBS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1 and T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`.

`<Rn>` For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the general-purpose source register, encoded in the "Rn" field.

`<const>` For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* on page F2-4096 for the range of values.

For encoding T2: an immediate value. See *Modified immediate constants in T32 instructions* on page F2-4095 for the range of values.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, nzcv) = AddWithCarry(NOT(R[n]), imm32, '1');
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.167 RSB, RSBS (register)

Reverse Subtract (register) subtracts a register value from an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the RSBS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The RSB variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- The RSBS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–7	6 5	4	3–0
!=1111 (cond)	0 0 0 0	0 1 1 S	Rn	Rd	imm5	stype	0	Rm

RSB, rotate right with extend variant

Applies when `S == 0 && imm5 == 00000 && stype == 11`.

```
RSB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

RSB, shift or rotate by value variant

Applies when `S == 0 && !(imm5 == 00000 && stype == 11)`.

```
RSB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

RSBS, rotate right with extend variant

Applies when `S == 1 && imm5 == 00000 && stype == 11`.

```
RSBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

RSBS, shift or rotate by value variant

Applies when `S == 1 && !(imm5 == 00000 && stype == 11)`.

```
RSBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15 14 13 12	11 10 9	8	7 6 5 4	3 0	15 14 12	11 8	7 6 5 4	3 0
1 1 1 0	1 0 1	1	1 1 1 0	S	Rn	(0) imm3	Rd	imm2 stype	Rm

RSB, rotate right with extend variant

Applies when `S == 0 && imm3 == 000 && imm2 == 00 && stype == 11`.

```
RSB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

RSB, shift or rotate by value variant

Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11)`.

```
RSB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

RSBS, rotate right with extend variant

Applies when `S == 1 && imm3 == 000 && imm2 == 00 && stype == 11`.

```
RSBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

RSBS, shift or rotate by value variant

Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11)`.

```
RSBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the RSB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- For the RSBS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`.

`<Rn>` For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1: is the first general-purpose source register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Rm>` For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1: is the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

`LSL`	when `stype = 00`
`LSR`	when `stype = 01`
`ASR`	when `stype = 10`
`ROR`	when `stype = 11`

`<amount>` For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(NOT(R[n]), shifted, '1');
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.168 RSB, RSBS (register-shifted register)

Reverse Subtract (register-shifted register) subtracts a register value from a register-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–8	7	6 5	4	3–0
!=1111 (cond)	0 0 0 0	0 1 1 S	Rn	Rd	Rs	0	stype	1	Rm

Flag setting variant

Applies when `S == 1`.

```
RSBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

Not flag setting variant

Applies when `S == 0`.

```
RSB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

- `LSL` when `stype = 00`
- `LSR` when `stype = 01`
- `ASR` when `stype = 10`
- `ROR` when `stype = 11`

`<Rs>` Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(NOT(R[n]), shifted, '1');
    R[d] = result;
    if setflags then
        PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.169 RSC, RSCS (immediate)

Reverse Subtract with Carry (immediate) subtracts a register value and the value of NOT (Carry flag) from an immediate value, and writes the result to the destination register.

If the destination register is not the PC, the RSCS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The RSC variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- The RSCS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–0
!=1111	0 0 1 0	1 1 1 S	Rn	Rd	imm12
cond					

RSC variant

Applies when `S == 0`.

```
RSC{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

RSCS variant

Applies when `S == 1`.

```
RSCS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = A32ExpandImm(imm12);
```

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the RSC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- For the RSCS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

`<Rn>` Is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<const>` An immediate value. See *Modified immediate constants in A32 instructions* on page F2-4096 for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, nzcv) = AddWithCarry(NOT(R[n]), imm32, PSTATE.C);
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.170 RSC, RSCS (register)

Reverse Subtract with Carry (register) subtracts a register value and the value of NOT (Carry flag) from an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the RSCS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The RSC variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- The RSCS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–7	6 5	4	3–0
!=1111 (cond)	0 0 0 0	1 1 1 S	Rn	Rd	imm5	stype	0	Rm

RSC, rotate right with extend variant

Applies when `S == 0 && imm5 == 00000 && stype == 11`.

```
RSC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

RSC, shift or rotate by value variant

Applies when `S == 0 && !(imm5 == 00000 && stype == 11)`.

```
RSC{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

RSCS, rotate right with extend variant

Applies when `S == 1 && imm5 == 00000 && stype == 11`.

```
RSCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

RSCS, shift or rotate by value variant

Applies when `S == 1 && !(imm5 == 00000 && stype == 11)`.

```
RSCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the RSC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- For the RSCS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

`LSL`	when `stype = 00`
`LSR`	when `stype = 01`
`ASR`	when `stype = 10`
`ROR`	when `stype = 11`

`<amount>` Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(NOT(R[n]), shifted, PSTATE.C);
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.171 RSC, RSCS (register-shifted register)

Reverse Subtract (register-shifted register) subtracts a register value and the value of NOT (Carry flag) from a register-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

A1

31–28	27–24	23–20	19–16	15–12	11–8	7	6–5	4	3–0
!=1111 (cond)	0 0 0 0	1 1 1 S	Rn	Rd	Rs	0	stype	1	Rm

Flag setting variant

Applies when `S == 1`.

```
RSCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

Not flag setting variant

Applies when `S == 0`.

```
RSC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

- `LSL` when `stype = 00`
- `LSR` when `stype = 01`
- `ASR` when `stype = 10`
- `ROR` when `stype = 11`

`<Rs>` Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(NOT(R[n]), shifted, PSTATE.C);
    R[d] = result;
    if setflags then
        PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.172 SADD16

Signed Add 16 performs two 16-bit signed integer additions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the additions.

A1

31–28	27–20	19–16	15–12	11–8	7–4	3–0
!=1111 (cond)	0 1 1 0 0 0 0 1	Rn	Rd	(1)(1)(1)(1)	0 0 0 1	Rm

A1 variant

```
SADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15–12	11–4	3–0	15–12	11–8	7–4	3–0
1 1 1 1	1 0 1 0 1 0 0 1	Rn	1 1 1 1	Rd	0 0 0 0	Rm

T1 variant

```
SADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<15:0>) + SInt(R[m]<15:0>);
    sum2 = SInt(R[n]<31:16>) + SInt(R[m]<31:16>);
    R[d]<15:0>  = sum1<15:0>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
R[d]<31:16> = sum2<15:0>;
PSTATE.GE<1:0> = if sum1 >= 0 then '11' else '00';
PSTATE.GE<3:2> = if sum2 >= 0 then '11' else '00';
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.173 SADD8

Signed Add 8 performs four 8-bit signed integer additions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the additions.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11 10 9 8	7	6 5	4	3–0
!=1111 (cond)	0 1 1 0	0 0 1	Rn	Rd	(1)(1)(1)(1)	1	0 0	1	Rm

A1 variant

```
SADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15 14 13 12	11–8	7	6	5	4	3–0
1 1 1 1	1 0 1 0	1	0 0 0	Rn	1 1 1 1	Rd	0	0	0	0	Rm

T1 variant

```
SADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<7:0>) + SInt(R[m]<7:0>);
    sum2 = SInt(R[n]<15:8>) + SInt(R[m]<15:8>);
    sum3 = SInt(R[n]<23:16>) + SInt(R[m]<23:16>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
sum4 = SInt(R[n]<31:24>) + SInt(R[m]<31:24>);
R[d]<7:0>   = sum1<7:0>;
R[d]<15:8>  = sum2<7:0>;
R[d]<23:16> = sum3<7:0>;
R[d]<31:24> = sum4<7:0>;
PSTATE.GE<0>  = if sum1 >= 0 then '1' else '0';
PSTATE.GE<1>  = if sum2 >= 0 then '1' else '0';
PSTATE.GE<2>  = if sum3 >= 0 then '1' else '0';
PSTATE.GE<3>  = if sum4 >= 0 then '1' else '0';
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.174 SASX

Signed Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer addition and one 16-bit subtraction, and writes the results to the destination register. It sets PSTATE.GE according to the results.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11 10 9 8	7	6 5 4	3–0
!=1111 (cond)	0 1 1 0 0	0 0 1	Rn	Rd	(1)(1)(1)(1)	0	0 1 1	Rm

A1 variant

```
SASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15 14 13 12	11–8	7 6 5 4	3–0
1 1 1 1	1 0 1 0	1	0 1 0	Rn	1 1 1 1	Rd	0 0 0 0	Rm

T1 variant

```
SASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = SInt(R[n]<15:0>) - SInt(R[m]<31:16>);
    sum  = SInt(R[n]<31:16>) + SInt(R[m]<15:0>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
R[d]<15:0>  = diff<15:0>;
R[d]<31:16> = sum<15:0>;
PSTATE.GE<1:0> = if diff >= 0 then '11' else '00';
PSTATE.GE<3:2> = if sum  >= 0 then '11' else '00';
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.175 SB

Speculation Barrier is a barrier that controls speculation.

The semantics of the Speculation Barrier are that the execution, until the barrier completes, of any instruction that appears later in the program order than the barrier:

- Cannot be performed speculatively to the extent that such speculation can be observed through side-channels as a result of control flow speculation or data value speculation.
- Can be speculatively executed as a result of predicting that a potentially exception generating instruction has not generated an exception.

In particular, any instruction that appears later in the program order than the barrier cannot cause a speculative allocation into any caching structure where the allocation of that entry could be indicative of any data value present in memory or in the registers.

The SB instruction:

- Cannot be speculatively executed as a result of control flow speculation or data value speculation.
- Can be speculatively executed as a result of predicting that a potentially exception generating instruction has not generated an exception. The potentially exception generating instruction can complete once it is known not to be speculative, and all data values generated by instructions appearing in program order before the SB instruction have their predicted values confirmed.

When the prediction of the instruction stream is not informed by data taken from the register outputs of the speculative execution of instructions appearing in program order after an uncompleted SB instruction, the SB instruction has no effect on the use of prediction resources to predict the instruction stream that is being fetched.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	1	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	1	1	1	(0)	(0)	(0)	(0)

A1 variant

```
SB{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	1	1	1	(0)	(0)	(0)	(0)

T1 variant

```
SB{<q>}
```

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<q> See *Standard assembler syntax fields* on page F2-4080.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    SpeculationBarrier();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.176 SBC, SBCS (immediate)

Subtract with Carry (immediate) subtracts an immediate value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register.

If the destination register is not the PC, the SBCS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The SBC variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- The SBCS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–0
!=1111 (cond)	0 0 1 0	1 1 0 S	Rn	Rd	imm12

SBC variant

Applies when `S == 0`.

```
SBC{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

SBCS variant

Applies when `S == 1`.

```
SBCS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = A32ExpandImm(imm12);
```

T1

15 14 13 12	11	10	9	8	7 6 5 4	3–0	15	14–12	11–8	7–0
1 1 1 1 0	i	0	1	0	1 1 S	Rn	0	imm3	Rd	imm8

SBC variant

Applies when `S == 0`.

```
SBC{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

SBCS variant

Applies when `S == 1`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
SBCS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = T32ExpandImm(i:imm3:imm8);
if d == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the SBC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- For the SBCS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`.

`<Rn>` For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

`<const>` For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* on page F2-4096 for the range of values.

For encoding T1: an immediate value. See *Modified immediate constants in T32 instructions* on page F2-4095 for the range of values.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, nzcv) = AddWithCarry(R[n], NOT(imm32), PSTATE.C);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - — The values of the data supplied in any of its registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.177 SBC, SBCS (register)

Subtract with Carry (register) subtracts an optionally-shifted register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register.

If the destination register is not the PC, the SBCS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The SBC variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- The SBCS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–7	6 5	4	3–0
!=1111	0 0 0 0	1 1 0 S	Rn	Rd	imm5	stype	0	Rm
cond								

SBC, rotate right with extend variant

Applies when `S == 0 && imm5 == 00000 && stype == 11`.

```
SBC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

SBC, shift or rotate by value variant

Applies when `S == 0 && !(imm5 == 00000 && stype == 11)`.

```
SBC{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

SBCS, rotate right with extend variant

Applies when `S == 1 && imm5 == 00000 && stype == 11`.

```
SBCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

SBCS, shift or rotate by value variant

Applies when `S == 1 && !(imm5 == 00000 && stype == 11)`.

```
SBCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15 14 13 12	11 10	9 8	7 6	5 3	2 0
0 1 0 0	0 0	0 1	1 0	Rm	Rdn

T1 variant

```
SBC<c>{<q>} {<Rdn>,} <Rdn>, <Rm> // Inside IT block
SBCS{<q>} {<Rdn>,} <Rdn>, <Rm> // Outside IT block
```

Decode for this encoding

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2

15 14 13 12	11 10 9	8	7 6 5	4	3 0	15 14 12	11 8	7 6 5 4	3 0
1 1 1 0	1 0 1	1	0 1 1	S	Rn	(0) imm3	Rd	imm2 stype	Rm

SBC, rotate right with extend variant

Applies when `S == 0 && imm3 == 000 && imm2 == 00 && stype == 11`.

```
SBC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

SBC, shift or rotate by value variant

Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11)`.

```
SBC<c>.W {<Rd>,} <Rn>, <Rm> // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
SBC{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

SBCS, rotate right with extend variant

Applies when `S == 1 && imm3 == 000 && imm2 == 00 && stype == 11`.

```
SBCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

SBCS, shift or rotate by value variant

Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11)`.

```
SBCS.W {<Rd>,} <Rn>, <Rm> // Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
SBCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rdn>` Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the SBC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- For the SBCS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`.

`<Rn>` For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T2: is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

`LSL`	when `stype = 00`
`LSR`	when `stype = 01`
`ASR`	when `stype = 10`
`ROR`	when `stype = 11`

`<amount>` For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], NOT(shifted), PSTATE.C);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.178 SBC, SBCS (register-shifted register)

Subtract with Carry (register-shifted register) subtracts a register-shifted register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

A1

31–28	27 26 25 24	23 22 21	20	19–16	15–12	11–8	7	6 5	4	3–0
!=1111 (cond)	0 0 0 0	1 1 0	S	Rn	Rd	Rs	0	stype	1	Rm

Flag setting variant

Applies when `S == 1`.

```
SBCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

Not flag setting variant

Applies when `S == 0`.

```
SBC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

- `LSL` when `stype = 00`
- `LSR` when `stype = 01`
- `ASR` when `stype = 10`
- `ROR` when `stype = 11`

`<Rs>` Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], NOT(shifted), PSTATE.C);
    R[d] = result;
    if setflags then
        PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.179 SBFX

Signed Bit Field Extract extracts any number of adjacent bits at any position from a register, sign-extends them to 32 bits, and writes the result to the destination register.

A1

31–28	27	26	25	24	23	22	21	20–16	15–12	11–7	6	5	4	3–0
!=1111	0	1	1	1	1	0	1	widthm1	Rd	lsb	1	0	1	Rn
cond														

A1 variant

```
SBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);
lsbit = UInt(lsb);  widthminus1 = UInt(widthm1);
if d == 15 || n == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15	14–12	11–8	7	6	5	4	0
1	1	1	1	0	(0)	1	1	0	1	0	0	Rn	0	imm3	Rd	imm2	(0)	widthm1		

T1 variant

```
SBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);
lsbit = UInt(imm3:imm2);  widthminus1 = UInt(widthm1);
if d == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rn>` Is the general-purpose source register, encoded in the "Rn" field.

`<lsb>` For encoding A1: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the "lsb" field.

For encoding T1: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the "imm3:imm2" field.

`<width>` Is the width of the field, in the range 1 to 32-<lsb>, encoded in the "widthm1" field as <width>-1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    msbit = lsbit + widthminus1;
    if msbit <= 31 then
        R[d] = SignExtend(R[n]<msbit:lsbit>, 32);
    else
        UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `msbit > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.180 SDIV

Signed Divide divides a 32-bit signed integer register value by a 32-bit signed integer register value, and writes the result to the destination register. The condition flags are not affected.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–8	7 6 5 4	3–0
!=1111	0 1 1 1 0	0 0 1	Rd	(1) (1) (1) (1)	Rm	0 0 0 1	Rn
cond				Ra			

A1 variant

```
SDIV{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
if d == 15 || n == 15 || m == 15 || a != 15 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `Ra != '1111'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction executes as described, and the register specified by Ra becomes UNKNOWN.

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15–12	11–8	7 6 5 4	3–0
1 1 1 1	1 0 1 1 1	0 0 1	Rn	(1) (1) (1) (1)	Rd	1 1 1 1	Rm
				Ra			

T1 variant

```
SDIV{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
if d == 15 || n == 15 || m == 15 || a != 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `Ra != '1111'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction executes as described, and the register specified by Ra becomes UNKNOWN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rn>` Is the first general-purpose source register holding the dividend, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register holding the divisor, encoded in the "Rm" field.

Overflow

If the signed integer division `0x80000000` / `0xFFFFFFFF` is performed, the pseudocode produces the intermediate integer result $+2^{31}$, that overflows the 32-bit signed integer range. No indication of this overflow case is produced, and the 32-bit result written to `<Rd>` must be the bottom 32 bits of the binary representation of $+2^{31}$. So the result of the division is `0x80000000`.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if SInt(R[m]) == 0 then
        result = 0;
    else
        result = RoundTowardsZero(Real(SInt(R[n])) / Real(SInt(R[m])));
    R[d] = result<31:0>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.181 SEL

Select Bytes selects each byte of its result from either its first operand or its second operand, according to the values of the PSTATE.GE flags.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	0 1 1 0	1 0 0 0	Rn	Rd	(1)	(1)	(1)	(1)	1	0	1	1	Rm
cond													

A1 variant

```
SEL{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15 14 13 12	11–8	7 6	5 4	3–0
1 1 1 1	1 0 1 0	1	0 1 0	Rn	1 1 1 1	Rd	1 0	0 0	Rm

T1 variant

```
SEL{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R[d]<7:0>   = if PSTATE.GE<0> == '1' then R[n]<7:0>   else R[m]<7:0>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
R[d]<15:8>  = if PSTATE.GE<1> == '1' then R[n]<15:8>  else R[m]<15:8>;
R[d]<23:16> = if PSTATE.GE<2> == '1' then R[n]<23:16> else R[m]<23:16>;
R[d]<31:24> = if PSTATE.GE<3> == '1' then R[n]<31:24> else R[m]<31:24>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.182 SETEND

Set Endianness writes a new value to PSTATE.E.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	1	0	0	0	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)	(0)	(0)	E	(0)	0	0	0	0	(0)	(0)	(0)	(0)

A1 variant

```
SETEND{<q>} <endian_specifier> // Cannot be conditional
```

Decode for this encoding

```
set_bigend = (E == '1');
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	1	0	0	1	0	(1)	E	(0)	(0)	(0)

T1 variant

```
SETEND{<q>} <endian_specifier> // Not permitted in IT block
```

Decode for this encoding

```
set_bigend = (E == '1');
if InITBlock() then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<endian_specifier>` Is the endianness to be selected, and the value to be set in PSTATE.E, encoded in the "E" field. It can have the following values:

`LE` when `E = 0`

`BE` when `E = 1`

Operation for all encodings

```
EncodingSpecificOperations();
AArch32.CheckSETENDEnabled();
PSTATE.E = if set_bigend then '1' else '0';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

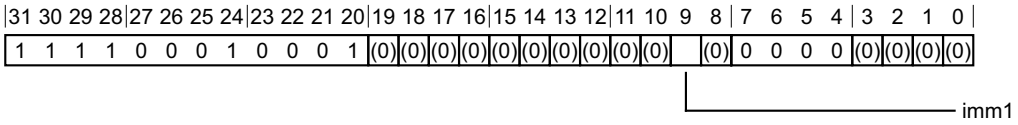
F5.1.183 SETPAN

Set Privileged Access Never writes a new value to PSTATE.PAN.

This instruction is available only in privileged mode and it is a NOP when executed in User mode.

A1

ARMv8.1



A1 variant

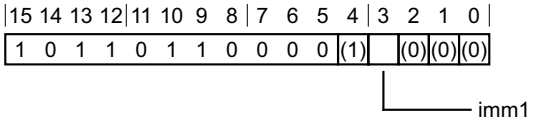
```
SETPAN{<q>} #<imm> // Cannot be conditional
```

Decode for this encoding

```
if !HavePANExt() then UNDEFINED;
value = imm1;
```

T1

ARMv8.1



T1 variant

```
SETPAN{<q>} #<imm> // Not permitted in IT block
```

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HavePANExt() then UNDEFINED;
value = imm1;
```

Assembler symbols

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<imm>` Is the unsigned immediate 0 or 1, encoded in the "imm1" field.

Operation for all encodings

```
EncodingSpecificOperations();
if PSTATE.EL != EL0 then
    PSTATE.PAN = value;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.184 SEV

Send Event is a hint instruction. It causes an event to be signaled to all PEs in the multiprocessor system. For more information, see *Wait For Event and Send Event* on page G1-5764.

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!=1111	0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	0	1	0	0
cond																												

A1 variant

```
SEV{<c>}{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0

T1 variant

```
SEV{<c>}{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	1	0	0

T2 variant

```
SEV{<c>}.W
```

Decode for this encoding

```
// No additional decoding required
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    SendEvent();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.185 SEVL

Send Event Local is a hint instruction that causes an event to be signaled locally without requiring the event to be signaled to other PEs in the multiprocessor system. It can prime a wait-loop which starts with a `WFE` instruction.

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!=1111 (cond)	0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	0	1	0	1

A1 variant

```
SEVL{<c>}{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	1	0	1	0	0	0	0

T1 variant

```
SEVL{<c>}{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	1	0	1

T2 variant

```
SEVL{<c>}.W
```

Decode for this encoding

```
// No additional decoding required
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    SendEventLocal();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.186 SHADD16

Signed Halving Add 16 performs two signed 16-bit integer additions, halves the results, and writes the results to the destination register.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11 10 9 8	7	6 5 4	3–0
!=1111 (cond)	0 1 1 0 0	0 1 1	Rn	Rd	(1)(1)(1)(1)	0	0 0 1	Rm

A1 variant

```
SHADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7	6 5 4	3–0	15 14 13 12	11–8	7	6	5	4	3–0
1 1 1 1	1 0 1 0	1	0 0 1	Rn	1 1 1 1	Rd	0	0	1	0	Rm

T1 variant

```
SHADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<15:0>) + SInt(R[m]<15:0>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
sum2 = SInt(R[n]<31:16>) + SInt(R[m]<31:16>);
R[d]<15:0>  = sum1<16:1>;
R[d]<31:16> = sum2<16:1>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.187 SHADD8

Signed Halving Add 8 performs four signed 8-bit integer additions, halves the results, and writes the results to the destination register.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11 10 9 8	7	6 5	4	3–0
!=1111 (cond)	0 1 1 0	0 1 1	Rn	Rd	(1)(1)(1)(1)	1	0 0	1	Rm

A1 variant

```
SHADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15 14 13 12	11–8	7	6	5	4	3–0
1 1 1 1	1 0 1 0	1	0 0 0	Rn	1 1 1 1	Rd	0	0	1	0	Rm

T1 variant

```
SHADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<7:0>) + SInt(R[m]<7:0>);
    sum2 = SInt(R[n]<15:8>) + SInt(R[m]<15:8>);
    sum3 = SInt(R[n]<23:16>) + SInt(R[m]<23:16>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
sum4 = SInt(R[n]<31:24>) + SInt(R[m]<31:24>);
R[d]<7:0>   = sum1<8:1>;
R[d]<15:8>  = sum2<8:1>;
R[d]<23:16> = sum3<8:1>;
R[d]<31:24> = sum4<8:1>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.188 SHASX

Signed Halving Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one signed 16-bit integer addition and one signed 16-bit subtraction, halves the results, and writes the results to the destination register.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	0 1 1 0	0 1 1	Rn	Rd	(1)	(1)	(1)	(1)	0	0	1	1	Rm
cond													

A1 variant

```
SHASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15 14 13 12	11–8	7	6	5	4	3–0
1 1 1 1	1 0 1 0	1	0 1 0	Rn	1 1 1 1	Rd	0	0	1	0	Rm

T1 variant

```
SHASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = SInt(R[n]<15:0>) - SInt(R[m]<31:16>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
sum  = SInt(R[n]<31:16>) + SInt(R[m]<15:0>);
R[d]<15:0>  = diff<16:1>;
R[d]<31:16> = sum<16:1>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.189 SHSAX

Signed Halving Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one signed 16-bit integer subtraction and one signed 16-bit addition, halves the results, and writes the results to the destination register.

A1

31–28	27–20	19–16	15–12	11–8	7–4	3–0
!=1111 (cond)	0 1 1 0 0 0 1 1	Rn	Rd	(1)(1)(1)(1)	0 1 0 1	Rm

A1 variant

```
SHSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15–12	11–8	7–4	3–0	15–12	11–8	7–4	3–0
1 1 1 1	1 0 1 0	1 1 0 1	Rn	1 1 1 1	Rd	0 0 1 0	Rm

T1 variant

```
SHSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = SInt(R[n]<15:0>) + SInt(R[m]<31:16>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
diff = SInt(R[n]<31:16>) - SInt(R[m]<15:0>);
R[d]<15:0>  = sum<16:1>;
R[d]<31:16> = diff<16:1>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.190 SHSUB16

Signed Halving Subtract 16 performs two signed 16-bit integer subtractions, halves the results, and writes the results to the destination register.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11 10 9 8	7	6 5 4	3–0
!=1111 (cond)	0 1 1 0 0	0 1 1	Rn	Rd	(1)(1)(1)(1)	0	1 1 1	Rm

A1 variant

```
SHSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7	6 5 4	3–0	15 14 13 12	11–8	7	6	5	4	3–0
1 1 1 1	1 0 1 0	1	1 0 1	Rn	1 1 1 1	Rd	0	0	1	0	Rm

T1 variant

```
SHSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<15:0>) - SInt(R[m]<15:0>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
diff2 = SInt(R[n]<31:16>) - SInt(R[m]<31:16>);
R[d]<15:0>  = diff1<16:1>;
R[d]<31:16> = diff2<16:1>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.191 SHSUB8

Signed Halving Subtract 8 performs four signed 8-bit integer subtractions, halves the results, and writes the results to the destination register.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11 10 9 8	7	6 5 4	3–0
!=1111 (cond)	0 1 1 0 0	0 1 1	Rn	Rd	(1)(1)(1)(1)	1	1 1 1	Rm

A1 variant

```
SHSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15 14 13 12	11–8	7	6	5	4	3–0
1 1 1 1	1 0 1 0 1	1 0 0	Rn	1 1 1 1	Rd	0	0	1	0	Rm

T1 variant

```
SHSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<7:0>) - SInt(R[m]<7:0>);
    diff2 = SInt(R[n]<15:8>) - SInt(R[m]<15:8>);
    diff3 = SInt(R[n]<23:16>) - SInt(R[m]<23:16>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
diff4 = SInt(R[n]<31:24>) - SInt(R[m]<31:24>);
R[d]<7:0>   = diff1<8:1>;
R[d]<15:8>  = diff2<8:1>;
R[d]<23:16> = diff3<8:1>;
R[d]<31:24> = diff4<8:1>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.192 SMC

Secure Monitor Call causes a Secure Monitor Call exception. For more information see *Secure Monitor Call (SMC) exception* on page G1-5743.

`SMC` is available only for software executing at EL1 or higher. It is UNDEFINED in User mode.

If the values of HCR.TSC and SCR.SCD are both 0, execution of an `SMC` instruction at EL1 or higher generates a Secure Monitor Call exception that is taken to EL3. When EL3 is using AArch32 this exception is taken to Monitor mode. When EL3 is using AArch64, it is the SCR_EL3.SMD bit, rather than the SCR.SCD bit, that can change the effect of executing an SMC instruction.

If the value of HCR.TSC is 1, execution of an `SMC` instruction in a Non-secure EL1 mode generates an exception that is taken to EL2, regardless of the value of SCR.SCD. When EL2 is using AArch32, this is a Hyp Trap exception that is taken to Hyp mode. For more information see *Traps to Hyp mode of Non-secure EL1 execution of SMC instructions* on page G1-5794.

If the value of HCR.TSC is 0 and the value of SCR.SCD is 1, the SMC instruction is:

- UNDEFINED in Non-secure state.
- CONSTRAINED UNPREDICTABLE if executed in Secure state at EL1 or higher.

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3–0
!=1111	0	0	0	1	0	1	1	0	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	0	1	1	1	imm4
cond																									

A1 variant

```
SMC{<c>}{<q>} {#}<imm4>
```

Decode for this encoding

```
// imm4 is for assembly/disassembly only and is ignored by hardware
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	1	1	1	1	1	imm4	1	0	0	0	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

T1 variant

```
SMC{<c>}{<q>} {#}<imm4>
```

Decode for this encoding

```
// imm4 is for assembly/disassembly only and is ignored by hardware
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

<imm4> Is a 4-bit unsigned immediate value, in the range 0 to 15, encoded in the "imm4" field. This is ignored by the PE. The Secure Monitor Call exception handler (Secure Monitor code) can use this value to determine what service is being requested, but Arm does not recommend this.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();

    AArch32.CheckForSMCUndefOrTrap();

    if !ELUsingAArch32(EL3) then
        if SCR_EL3.SMD == '1' then
            // SMC disabled.
            UNDEFINED;
    else
        if SCR.SCD == '1' then
            // SMC disabled
            if IsSecure() then
                // Executes either as a NOP or UNALLOCATED.
                c = ConstrainUnpredictable();
                assert c IN {Constraint_NOP, Constraint_UNDEF};
                if c == Constraint_NOP then EndOfInstruction();
            UNDEFINED;

    if !ELUsingAArch32(EL3) then
        AArch64.CallSecureMonitor(Zeros(16));
    else
        AArch32.TakeSMCException();
```

CONSTRAINED UNPREDICTABLE behavior

If `SCR.SCD == '1' && IsSecure()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.193 SMLABB, SMLABT, SMLATB, SMLATT

Signed Multiply Accumulate (halfwords) performs a signed multiply accumulate operation. The multiply acts on two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is added to a 32-bit accumulate value and the result is written to the destination register.

If overflow occurs during the addition of the accumulate value, the instruction sets PSTATE.Q to 1. It is not possible for overflow to occur during the multiplication.

A1

31–28	27–20	19–16	15–12	11–8	7	6	5	4	3–0
!=1111 (cond)	0 0 0 1 0 0 0 0	Rd	Ra	Rm	1	M	N	0	Rn

SMLABB variant

Applies when `M == 0 && N == 0`.

```
SMLABB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMLABT variant

Applies when `M == 1 && N == 0`.

```
SMLABT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMLATB variant

Applies when `M == 0 && N == 1`.

```
SMLATB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMLATT variant

Applies when `M == 1 && N == 1`.

```
SMLATT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
n_high = (N == '1');  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
```

T1

15–12	11–8	7	6	5	4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1 1	1 0 1 1	0	0	0	1	Rn	!=1111 (Ra)	Rd	0	0	N	M	Rm

SMLABB variant

Applies when `N == 0 && M == 0`.

```
SMLABB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMLABT variant

Applies when `N == 0 && M == 1`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
SMLABT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMLATB variant

Applies when `N == 1 && M == 0`.

```
SMLATB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMLATT variant

Applies when `N == 1 && M == 1`.

```
SMLATT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for all variants of this encoding

```
if Ra == '1111' then SEE "SMULBB, SMULBT, SMULTB, SMULTT";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
n_high = (N == '1');  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<Rn>`	Is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by `<x>`), encoded in the "Rn" field.
`<Rm>`	Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by `<y>`), encoded in the "Rm" field.
`<Ra>`	Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = if n_high then R[n]<31:16> else R[n]<15:0>;
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    result = SInt(operand1) * SInt(operand2) + SInt(R[a]);
    R[d] = result<31:0>;
    if result != SInt(result<31:0>) then  // Signed overflow
        PSTATE.Q = '1';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.194 SMLAD, SMLADX

Signed Multiply Accumulate Dual performs two signed 16 x 16-bit multiplications. It adds the products to a 32-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

This instruction sets PSTATE.Q to 1 if the accumulate operation overflows. Overflow cannot occur during the multiplications.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–8	7 6	5	4	3–0
!=1111	0 1 1 1	0 0 0	Rd	!=1111	Rm	0 0	M	1	Rn
cond				Ra					

SMLAD variant

Applies when `M == 0`.

```
SMLAD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMLADX variant

Applies when `M == 1`.

```
SMLADX{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for all variants of this encoding

```
if Ra == '1111' then SEE "SMUAD";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15–12	11–8	7 6	5	4	3–0
1 1 1 1	1 0 1 1	0 1 0	Rn	!=1111	Rd	0 0	0	M	Rm
				Ra					

SMLAD variant

Applies when `M == 0`.

```
SMLAD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMLADX variant

Applies when `M == 1`.

```
SMLADX{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for all variants of this encoding

```
if Ra == '1111' then SEE "SMUAD";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.
<Ra>	Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 + product2 + SInt(R[a]);
    R[d] = result<31:0>;
    if result != SInt(result<31:0>) then  // Signed overflow
        PSTATE.Q = '1';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.195 SMLAL, SMLALS

Signed Multiply Accumulate Long multiplies two signed 32-bit values to produce a 64-bit value, and accumulates this with a 64-bit value.

In A32 instructions, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

A1

31–28	27–24	23–20	19–16	15–12	11–8	7–4	3–0
!=1111 (cond)	0 0 0 0	1 1 1 S	RdHi	RdLo	Rm	1 0 0 1	Rn

Flag setting variant

Applies when `S == 1`.

```
SMLALS{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Not flag setting variant

Applies when `S == 0`.

```
SMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Decode for all variants of this encoding

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

T1

15–12	11–8	7–4	3–0	15–12	11–8	7–4	3–0
1 1 1 1	1 0 1 1	1 0 0	Rn	RdLo	RdHi	0 0 0 0	Rm

T1 variant

```
SMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Decode for this encoding

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<RdLo>`	Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
`<RdHi>`	Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.
`<Rn>`	Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
`<Rm>`	Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = SInt(R[n]) * SInt(R[m]) + SInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
    if setflags then
        PSTATE.N = result<63>;
        PSTATE.Z = IsZeroBit(result<63:0>);
        // PSTATE.C, PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.196 SMLALBB, SMLALBT, SMLALTB, SMLALTT

Signed Multiply Accumulate Long (halfwords) multiplies two signed 16-bit values to produce a 32-bit value, and accumulates this with a 64-bit value. The multiply acts on two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is sign-extended and accumulated with a 64-bit accumulate value.

Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo 2^{64}.

A1

31–28	27–24	23–20	19–16	15–12	11–8	7	6	5	4	3–0
!=1111 (cond)	0 0 0 1 0	1 0 0	RdHi	RdLo	Rm	1	M	N	0	Rn

SMLALBB variant

Applies when `M == 0 && N == 0`.

```
SMLALBB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

SMLALBT variant

Applies when `M == 1 && N == 0`.

```
SMLALBT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

SMLALTB variant

Applies when `M == 0 && N == 1`.

```
SMLALTB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

SMLALTT variant

Applies when `M == 1 && N == 1`.

```
SMLALTT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Decode for all variants of this encoding

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);
n_high = (N == '1');  m_high = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 8	7 6	5	4	3 0
1 1 1 1	1 0 1 1	1 0 0	Rn	RdLo	RdHi	1 0	N	M	Rm

SMLALBB variant

Applies when `N == 0 && M == 0`.

```
SMLALBB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

SMLALBT variant

Applies when `N == 0 && M == 1`.

```
SMLALBT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

SMLALTB variant

Applies when `N == 1 && M == 0`.

```
SMLALTB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

SMLALTT variant

Applies when `N == 1 && M == 1`.

```
SMLALTT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Decode for all variants of this encoding

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);
n_high = (N == '1');  m_high = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<RdLo>`	Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
`<RdHi>`	Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Rn> For encoding A1: is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by <x>), encoded in the "Rn" field.

For encoding T1: is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by <x>), encoded in the "Rn" field.

<Rm> For encoding A1: is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <y>), encoded in the "Rm" field.

For encoding T1: is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <x>), encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = if n_high then R[n]<31:16> else R[n]<15:0>;
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    result = SInt(operand1) * SInt(operand2) + SInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.197 SMLALD, SMLALDX

Signed Multiply Accumulate Long Dual performs two signed 16 x 16-bit multiplications. It adds the products to a 64-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo 2^{64}.

A1

31–28	27–20	19–16	15–12	11–8	7–6	5	4	3–0
!=1111 (cond)	0 1 1 1 0 1 0 0	RdHi	RdLo	Rm	0 0	M	1	Rn

SMLALD variant

Applies when `M == 0`.

```
SMLALD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

SMLALDX variant

Applies when `M == 1`.

```
SMLALDX{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Decode for all variants of this encoding

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

T1

15–12	11–8	7–4	3–0	15–12	11–8	7–6	5	4	3–0
1 1 1 1	1 0 1 1	1 0 0 1	Rn	RdLo	RdHi	1 1	0	M	Rm

SMLALD variant

Applies when `M == 0`.

```
SMLALD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

SMLALDX variant

Applies when `M == 1`.

```
SMLALDX{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<RdLo>`	Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
`<RdHi>`	Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.
`<Rn>`	Is the first general-purpose source register, encoded in the "Rn" field.
`<Rm>`	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 + product2 + SInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.198 SMLAWB, SMLAWT

Signed Multiply Accumulate (word by halfword) performs a signed multiply accumulate operation. The multiply acts on a signed 32-bit quantity and a signed 16-bit quantity. The signed 16-bit quantity is taken from either the bottom or the top half of its source register. The other half of the second source register is ignored. The top 32 bits of the 48-bit product are added to a 32-bit accumulate value and the result is written to the destination register. The bottom 16 bits of the 48-bit product are ignored.

If overflow occurs during the addition of the accumulate value, the instruction sets PSTATE.Q to 1. No overflow can occur during the multiplication.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–8	7	6	5	4	3–0
!=1111 (cond)	0 0 0 1 0	0 1 0	Rd	Ra	Rm	1	M	0	0	Rn

SMLAWB variant

Applies when `M == 0`.

```
SMLAWB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMLAWT variant

Applies when `M == 1`.

```
SMLAWT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1 1	1 0 1 1 0	0 1 1	Rn	!=1111 (Ra)	Rd	0	0	0	M	Rm

SMLAWB variant

Applies when `M == 0`.

```
SMLAWB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMLAWT variant

Applies when `M == 1`.

```
SMLAWT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for all variants of this encoding

```
if Ra == '1111' then SEE "SMULWB, SMULWT";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <y>), encoded in the "Rm" field.
<Ra>	Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    result = SInt(R[n]) * SInt(operand2) + (SInt(R[a]) << 16);
    R[d] = result<47:16>;
    if (result >> 16) != SInt(R[d]) then  // Signed overflow
        PSTATE.Q = '1';
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.199 SMLSD, SMLSDX

Signed Multiply Subtract Dual performs two signed 16 x 16-bit multiplications. It adds the difference of the products to a 32-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

This instruction sets PSTATE.Q to 1 if the accumulate operation overflows. Overflow cannot occur during the multiplications or subtraction.

A1

31–28	27–20	19–16	15–12	11–8	7	6	5	4	3–0
!=1111 (cond)	0 1 1 1 0 0 0 0	Rd	!=1111 (Ra)	Rm	0	1	M	1	Rn

SMLSD variant

Applies when `M == 0`.

```
SMLSD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMLSDX variant

Applies when `M == 1`.

```
SMLSDX{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for all variants of this encoding

```
if Ra == '1111' then SEE "SMUSD";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15–12	11–8	7	6	5	4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1 1	1 0 1 1	0	1	0	0	Rn	!=1111 (Ra)	Rd	0	0	0	M	Rm

SMLSD variant

Applies when `M == 0`.

```
SMLSD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMLSDX variant

Applies when `M == 1`.

```
SMLSDX{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for all variants of this encoding

```
if Ra == '1111' then SEE "SMUSD";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<Rn>`	Is the first general-purpose source register, encoded in the "Rn" field.
`<Rm>`	Is the second general-purpose source register, encoded in the "Rm" field.
`<Ra>`	Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 - product2 + SInt(R[a]);
    R[d] = result<31:0>;
    if result != SInt(result<31:0>) then  // Signed overflow
        PSTATE.Q = '1';
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.200 SMLSLD, SMLSLDX

Signed Multiply Subtract Long Dual performs two signed 16 x 16-bit multiplications. It adds the difference of the products to a 64-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo 2^{64}.

A1

31–28	27–20	19–16	15–12	11–8	7	6	5	4	3–0
!=1111 (cond)	0 1 1 1 0 1 0 0	RdHi	RdLo	Rm	0	1	M	1	Rn

SMLSLD variant

Applies when `M == 0`.

```
SMLSLD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

SMLSLDX variant

Applies when `M == 1`.

```
SMLSLDX{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Decode for all variants of this encoding

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

T1

15–12	11–8	7	6	5	4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1 1	1 0 1 1	1	1	0	1	Rn	RdLo	RdHi	1	1	0	M	Rm

SMLSLD variant

Applies when `M == 0`.

```
SMLSLD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

SMLSLDX variant

Applies when `M == 1`.

```
SMLSLDX{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<RdLo>`	Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
`<RdHi>`	Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.
`<Rn>`	Is the first general-purpose source register, encoded in the "Rn" field.
`<Rm>`	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 - product2 + SInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.201 SMMLA, SMMLAR

Signed Most Significant Word Multiply Accumulate multiplies two signed 32-bit values, extracts the most significant 32 bits of the result, and adds an accumulate value.

Optionally, the instruction can specify that the result is rounded instead of being truncated. In this case, the constant `0x80000000` is added to the product before the high word is extracted.

A1

31–28	27–20	19–16	15–12	11–8	7	6	5	4	3–0
!=1111	0 1 1 1 0 1 0 1	Rd	!=1111	Rm	0	0	R	1	Rn
cond			Ra						

SMMLA variant

Applies when `R == 0`.

```
SMMLA{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMMLAR variant

Applies when `R == 1`.

```
SMMLAR{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for all variants of this encoding

```
if Ra == '1111' then SEE "SMMUL";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  round = (R == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15–12	11–10	9–8	7	6	5	4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1 1	1 0	1 1	0	1	0	1	Rn	!=1111	Rd	0	0	0	R	Rm
								Ra						

SMMLA variant

Applies when `R == 0`.

```
SMMLA{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMMLAR variant

Applies when `R == 1`.

```
SMMLAR{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for all variants of this encoding

```
if Ra == '1111' then SEE "SMMUL";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  round = (R == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<Rn>`	Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
`<Rm>`	Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
`<Ra>`	Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = (SInt(R[a]) << 32) + SInt(R[n]) * SInt(R[m]);
    if round then result = result + 0x80000000;
    R[d] = result<63:32>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.202 SMMLS, SMMLSR

Signed Most Significant Word Multiply Subtract multiplies two signed 32-bit values, subtracts the result from a 32-bit accumulate value that is shifted left by 32 bits, and extracts the most significant 32 bits of the result of that subtraction.

Optionally, the instruction can specify that the result of the instruction is rounded instead of being truncated. In this case, the constant `0x80000000` is added to the result of the subtraction before the high word is extracted.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–8	7 6	5	4	3–0
!=1111	0 1 1 1 0	1 0 1	Rd	Ra	Rm	1 1	R	1	Rn
cond									

SMMLS variant

Applies when `R == 0`.

```
SMMLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMMLSR variant

Applies when `R == 1`.

```
SMMLSR{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  round = (R == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15–12	11–8	7 6	5	4	3–0
1 1 1 1	1 0 1 1 0	1 1 0	Rn	Ra	Rd	0 0	0	R	Rm

SMMLS variant

Applies when `R == 0`.

```
SMMLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMMLSR variant

Applies when `R == 1`.

```
SMMLSR{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  round = (R == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<Rn>`	Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
`<Rm>`	Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
`<Ra>`	Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = (SInt(R[a]) << 32) - SInt(R[n]) * SInt(R[m]);
    if round then result = result + 0x80000000;
    R[d] = result<63:32>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.203 SMMUL, SMMULR

Signed Most Significant Word Multiply multiplies two signed 32-bit values, extracts the most significant 32 bits of the result, and writes those bits to the destination register.

Optionally, the instruction can specify that the result is rounded instead of being truncated. In this case, the constant `0x80000000` is added to the product before the high word is extracted.

A1

31–28	27–20	19–16	15–12	11–8	7	6	5	4	3–0
!=1111 (cond)	0 1 1 1 0 1 0 1	Rd	1 1 1 1	Rm	0	0	R	1	Rn

SMMUL variant

Applies when `R == 0`.

```
SMMUL{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMMULR variant

Applies when `R == 1`.

```
SMMULR{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  round = (R == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15–12	11–8	7	6	5	4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1 1	1 0 1 1 0	1	0	1		Rn	1 1 1 1	Rd	0	0	0	R	Rm

SMMUL variant

Applies when `R == 0`.

```
SMMUL{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMMULR variant

Applies when `R == 1`.

```
SMMULR{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  round = (R == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<Rn>`	Is the first general-purpose source register, encoded in the "Rn" field.
`<Rm>`	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = SInt(R[n]) * SInt(R[m]);
    if round then result = result + 0x80000000;
    R[d] = result<63:32>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.204 SMUAD, SMUADX

Signed Dual Multiply Add performs two signed 16 x 16-bit multiplications. It adds the products together, and writes the result to the destination register.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

This instruction sets PSTATE.Q to 1 if the addition overflows. The multiplications cannot overflow.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15 14 13 12	11–8	7 6	5	4	3–0
!=1111 (cond)	0 1 1 1 0	0 0 0	Rd	1 1 1 1	Rm	0 0	M	1	Rn

SMUAD variant

Applies when `M == 0`.

```
SMUAD{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMUADX variant

Applies when `M == 1`.

```
SMUADX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15 14 13 12	11–8	7 6	5	4	3–0
1 1 1 1	1 0 1 1	0 1 0	Rn	1 1 1 1	Rd	0 0	0	M	Rm

SMUAD variant

Applies when `M == 0`.

```
SMUAD{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMUADX variant

Applies when `M == 1`.

```
SMUADX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<Rn>`	Is the first general-purpose source register, encoded in the "Rn" field.
`<Rm>`	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 + product2;
    R[d] = result<31:0>;
    if result != SInt(result<31:0>) then  // Signed overflow
        PSTATE.Q = '1';
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.205 SMULBB, SMULBT, SMULTB, SMULTT

Signed Multiply (halfwords) multiplies two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is written to the destination register. No overflow is possible during this instruction.

A1

31–28	27–20				19–16	15–12	11–8	7	6	5	4	3–0
!=1111 (cond)	0 0 0 1 0	1 1	0		Rd	(0) (0) (0) (0)	Rm	1	M	N	0	Rn

SMULBB variant

Applies when `M == 0 && N == 0`.

```
SMULBB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMULBT variant

Applies when `M == 1 && N == 0`.

```
SMULBT{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMULTB variant

Applies when `M == 0 && N == 1`.

```
SMULTB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMULTT variant

Applies when `M == 1 && N == 1`.

```
SMULTT{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
n_high = (N == '1');  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15–12	11–8	7–4		3–0	15–12	11–8	7	6	5	4	3–0
1 1 1 1	1 0 1 1 0	0 0 1		Rn	1 1 1 1	Rd	0	0	N	M	Rm

SMULBB variant

Applies when `N == 0 && M == 0`.

```
SMULBB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMULBT variant

Applies when `N == 0 && M == 1`.

```
SMULBT{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

SMULTB variant

Applies when `N == 1 && M == 0`.

```
SMULTB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMULTT variant

Applies when `N == 1 && M == 1`.

```
SMULTT{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
n_high = (N == '1');  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<Rn>`	Is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by `<x>`), encoded in the "Rn" field.
`<Rm>`	Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by `<y>`), encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = if n_high then R[n]<31:16> else R[n]<15:0>;
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    result = SInt(operand1) * SInt(operand2);
    R[d] = result<31:0>;
    // Signed overflow cannot occur
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.206 SMULL, SMULLS

Signed Multiply Long multiplies two 32-bit signed values to produce a 64-bit result.

In A32 instructions, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

A1

31–28	27–24	23–20	19–16	15–12	11–8	7–4	3–0
!=1111 (cond)	0 0 0 0	1 1 0 S	RdHi	RdLo	Rm	1 0 0 1	Rn

Flag setting variant

Applies when `S == 1`.

```
SMULLS{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Not flag setting variant

Applies when `S == 0`.

```
SMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Decode for all variants of this encoding

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

T1

15–12	11–8	7–4	3–0	15–12	11–8	7–4	3–0
1 1 1 1	1 0 1 1	1 0 0 0	Rn	RdLo	RdHi	0 0 0 0	Rm

T1 variant

```
SMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Decode for this encoding

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<RdLo>`	Is the general-purpose destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
`<RdHi>`	Is the general-purpose destination register for the upper 32 bits of the result, encoded in the "RdHi" field.
`<Rn>`	Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
`<Rm>`	Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = SInt(R[n]) * SInt(R[m]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
    if setflags then
        PSTATE.N = result<63>;
        PSTATE.Z = IsZeroBit(result<63:0>);
        // PSTATE.C, PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.207 SMULWB, SMULWT

Signed Multiply (word by halfword) multiplies a signed 32-bit quantity and a signed 16-bit quantity. The signed 16-bit quantity is taken from either the bottom or the top half of its source register. The other half of the second source register is ignored. The top 32 bits of the 48-bit product are written to the destination register. The bottom 16 bits of the 48-bit product are ignored. No overflow is possible during this instruction.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15 14 13 12	11–8	7	6	5	4	3–0
!=1111	0 0 0 1 0	0 1 0	Rd	(0) (0) (0) (0)	Rm	1	M	1	0	Rn
cond										

SMULWB variant

Applies when `M == 0`.

```
SMULWB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMULWT variant

Applies when `M == 1`.

```
SMULWT{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15 14 13 12	11–8	7	6	5	4	3–0
1 1 1 1	1 0 1 1 0	0 1 1	Rn	1 1 1 1	Rd	0	0	0	M	Rm

SMULWB variant

Applies when `M == 0`.

```
SMULWB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMULWT variant

Applies when `M == 1`.

```
SMULWT{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <y>), encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    product = SInt(R[n]) * SInt(operand2);
    R[d] = product<47:16>;
    // Signed overflow cannot occur
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.208 SMUSD, SMUSDX

Signed Multiply Subtract Dual performs two signed 16 x 16-bit multiplications. It subtracts one of the products from the other, and writes the result to the destination register.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

Overflow cannot occur.

A1

31–28	27–20	19–16	15–12	11–8	7	6	5	4	3–0
!=1111 (cond)	0 1 1 1 0 0 0 0	Rd	1 1 1 1	Rm	0	1	M	1	Rn

SMUSD variant

Applies when `M == 0`.

```
SMUSD{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMUSDX variant

Applies when `M == 1`.

```
SMUSDX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15–12	11–8	7	6	5	4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1 1	1 0 1 1	0	1	0	0	Rn	1 1 1 1	Rd	0	0	0	M	Rm

SMUSD variant

Applies when `M == 0`.

```
SMUSD{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMUSDX variant

Applies when `M == 1`.

```
SMUSDX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 - product2;
    R[d] = result<31:0>;
    // Signed overflow cannot occur
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.209 SRS, SRSDA, SRSDB, SRSIA, SRSIB

Store Return State stores the LR_<current_mode> and SPSR_<current_mode> to the stack of a specified mode. For information about memory accesses see *Memory accesses* on page F2-4085.

SRS is UNDEFINED in Hyp mode.

SRS is CONSTRAINED UNPREDICTABLE if it is executed in User or System mode, or if the specified mode is any of the following:

- Not implemented.
- A mode that Table G1-5 on page G1-5686 does not show.
- Hyp mode.
- Monitor mode, if the SRS instruction is executed in Non-secure state.

If EL3 is using AArch64 and an SRS instruction that is executed in a Secure EL1 mode specifies Monitor mode, it is trapped to EL3.

See *Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32* on page D1-2391.

A1

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	0
1 1 1 1	1 0 0	P	U	1	W	0	(1) (1) (0) (1)	(0) (0) (0) (0)	(0) (1) (0) (1)	(0) (0) (0) (0)	mode

Decrement After variant

Applies when `P == 0 && U == 0`.

```
SRSDA{<c>}{<q>} SP{!}, #<mode>
```

Decrement Before variant

Applies when `P == 1 && U == 0`.

```
SRSDB{<c>}{<q>} SP{!}, #<mode>
```

Increment After variant

Applies when `P == 0 && U == 1`.

```
SRS{IA}{<c>}{<q>} SP{!}, #<mode>
```

Increment Before variant

Applies when `P == 1 && U == 1`.

```
SRSIB{<c>}{<q>} SP{!}, #<mode>
```

Decode for all variants of this encoding

```
wback = (W == '1');  increment = (U == '1');  wordhigher = (P == U);
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15	14	13 12	11 10 9 8	7 6 5 4	0
1 1 1 0	1 0 0	0 0	0 W 0	(1) (1) (0) (1)	(1)	(1)	(0) (0) (0) (0)	(0) (0) (0) (0)	mode

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1 variant

```
SRSDB{<c>}{<q>} SP{!}, #<mode>
```

Decode for this encoding

```
wback = (W == '1');  increment = FALSE;  wordhigher = FALSE;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	0
1	1	1	0	1	0	0	1	1	0	W	0	(1)	(1)	(0)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	mode

T2 variant

```
SRS{IA}{<c>}{<q>} SP{!}, #<mode>
```

Decode for this encoding

```
wback = (W == '1');  increment = TRUE;  wordhigher = FALSE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *SRS (T32)* on page K1-7843 and *SRS (A32)* on page K1-7843.

Assembler symbols

IA	For encoding A1: is an optional suffix to indicate the Increment After variant. For encoding T2: is an optional suffix for the Increment After form.
<c>	For encoding A1: see *Standard assembler syntax fields* on page F2-4080. <c> must be AL or omitted. For encoding T1 and T2: see *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
!	The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.
<mode>	Is the number of the mode whose Banked SP is used as the base register, encoded in the "mode" field. For details of PE modes and their numbers see *AArch32 state PE mode descriptions* on page G1-5686.

SRSFA, SRSEA, SRSFD, and SRSED are pseudo-instructions for SRSIB, SRSIA, SRSDB, and SRSDA respectively, referring to their use for pushing data onto Full Ascending, Empty Ascending, Full Descending, and Empty Descending stacks.

Operation for all encodings

```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        if PSTATE.EL == EL2 then          // UNDEFINED at EL2
            UNDEFINED;

        // Check for UNPREDICTABLE cases. The definition of UNPREDICTABLE does not permit these
        // to be security holes
        if PSTATE.M IN {M32_User,M32_System} then
            UNPREDICTABLE;
        elsif mode == M32_Hyp then        // Check for attempt to access Hyp mode SP
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            UNPREDICTABLE;
        elsif mode == M32_Monitor then    // Check for attempt to access Monitor mode SP
            if !HaveEL(EL3) || !IsSecure()  then
                UNPREDICTABLE;
            elsif !ELUsingAArch32(EL3) then
                AArch64.MonitorModeTrap();
        elsif BadMode(mode) then
            UNPREDICTABLE;

        base = Rmode[13,mode];
        address = if increment then base else base-8;
        if wordhigher then address = address+4;
        MemA[address,4]   = LR;
        MemA[address+4,4] = SPSR[];
        if wback then Rmode[13,mode] = if increment then base+8 else base-8;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        if PSTATE.EL == EL2 then          // UNDEFINED at EL2
            UNDEFINED;

        // Check for UNPREDICTABLE cases. The definition of UNPREDICTABLE does not permit these
        // to be security holes
        if PSTATE.M IN {M32_User,M32_System} then
            UNPREDICTABLE;
        elsif mode == M32_Hyp then        // Check for attempt to access Hyp mode SP
            UNPREDICTABLE;
        elsif mode == M32_Monitor then    // Check for attempt to access Monitor mode SP
            if !HaveEL(EL3) || !IsSecure()  then
                UNPREDICTABLE;
            elsif !ELUsingAArch32(EL3) then
                AArch64.MonitorModeTrap();
        elsif BadMode(mode) then
            UNPREDICTABLE;

        base = Rmode[13,mode];
        address = if increment then base else base-8;
        if wordhigher then address = address+4;
        MemA[address,4]   = LR;
        MemA[address+4,4] = SPSR[];
        if wback then Rmode[13,mode] = if increment then base+8 else base-8;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.M IN {M32_User,M32_System}`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

If `mode == M32_Hyp`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

If `mode == M32_Monitor && (!HaveEL(EL3) || !IsSecure())`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

If `BadMode(mode)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The instruction stores to the stack of the mode in which it is executed.
- The instruction stores to an UNKNOWN address, and if the instruction specifies writeback then any general-purpose register that can be accessed from the current Exception level without a privilege violation becomes UNKNOWN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.210 SSAT

Signed Saturate saturates an optionally-shifted signed value to a selectable signed range.

This instruction sets PSTATE.Q to 1 if the operation saturates.

A1

31–28	27	26	25	24	23	22	21	20–16	15–12	11–7	6	5	4	3–0
!=1111 (cond)	0	1	1	0	1	0	1	sat_imm	Rd	imm5	sh	0	1	Rn

Arithmetic shift right variant

Applies when `sh == 1`.

```
SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>
```

Logical shift left variant

Applies when `sh == 0`.

```
SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm)+1;
(shift_t, shift_n) = DecodeImmShift(sh:'0', imm5);
if d == 15 || n == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15	14–12	11–8	7	6	5	4	3–0
1	1	1	1	0	(0)	1	1	0	0	sh	0	Rn	0	imm3	Rd	imm2	(0)	sat_imm		

Arithmetic shift right variant

Applies when `sh == 1 && !(imm3 == 000 && imm2 == 00)`.

```
SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>
```

Logical shift left variant

Applies when `sh == 0`.

```
SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}
```

Decode for all variants of this encoding

```
if sh == '1' && (imm3:imm2) == '00000' then SEE "SSAT16";
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm)+1;
(shift_t, shift_n) = DecodeImmShift(sh:'0', imm3:imm2);
if d == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<imm>`	Is the bit position for saturation, in the range 1 to 32, encoded in the "sat_imm" field as <imm>-1.
`<Rn>`	Is the general-purpose source register, encoded in the "Rn" field.
`<amount>`	For encoding A1: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm5" field. For encoding A1: is the shift amount, in the range 1 to 32 encoded in the "imm5" field as <amount> modulo 32. For encoding T1: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm3:imm2" field. For encoding T1: is the shift amount, in the range 1 to 31 encoded in the "imm3:imm2" field as <amount>.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand = Shift(R[n], shift_t, shift_n, PSTATE.C);  // PSTATE.C ignored
    (result, sat) = SignedSatQ(SInt(operand), saturate_to);
    R[d] = SignExtend(result, 32);
    if sat then
        PSTATE.Q = '1';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.211 SSAT16

Signed Saturate 16 saturates two signed 16-bit values to a selected signed range.

This instruction sets PSTATE.Q to 1 if the operation saturates.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	0	1	1	0	1	0	1	0	sat_imm	Rd	(1)	(1)	(1)	(1)	0	0	1	1	Rn
cond																			

A1 variant

```
SSAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm)+1;
if d == 15 || n == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15	14	13	12	11–8	7	6	5	4	3–0
1	1	1	1	0	(0)	1	1	0	0	1	0	Rn	0	0	0	0	Rd	0	0	(0)	(0)	sat_imm

T1 variant

```
SSAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm)+1;
if d == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<imm>	Is the bit position for saturation, in the range 1 to 16, encoded in the "sat_imm" field as <imm>-1.
<Rn>	Is the general-purpose source register, encoded in the "Rn" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result1, sat1) = SignedSatQ(SInt(R[n]<15:0>), saturate_to);
    (result2, sat2) = SignedSatQ(SInt(R[n]<31:16>), saturate_to);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
R[d]<15:0> = SignExtend(result1, 16);
R[d]<31:16> = SignExtend(result2, 16);
if sat1 || sat2 then
    PSTATE.Q = '1';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.212 SSAX

Signed Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer subtraction and one 16-bit addition, and writes the results to the destination register. It sets PSTATE.GE according to the results.

A1

31–28	27–20	19–16	15–12	11–8	7–4	3–0
!=1111 (cond)	0 1 1 0 0 0 0 1	Rn	Rd	(1)(1)(1)(1)	0 1 0 1	Rm

A1 variant

```
SSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15–12	11–8	7–4	3–0	15–12	11–8	7–4	3–0
1 1 1 1	1 0 1 0 1	1 1 0	Rn	1 1 1 1	Rd	0 0 0 0	Rm

T1 variant

```
SSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = SInt(R[n]<15:0>) + SInt(R[m]<31:16>);
    diff = SInt(R[n]<31:16>) - SInt(R[m]<15:0>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
R[d]<15:0>  = sum<15:0>;
R[d]<31:16> = diff<15:0>;
PSTATE.GE<1:0> = if sum  >= 0 then '11' else '00';
PSTATE.GE<3:2> = if diff >= 0 then '11' else '00';
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.213 SSBB

Speculative Store Bypass Barrier is a memory barrier which prevents speculative loads from bypassing earlier stores to the same virtual address under certain conditions.

The semantics of the Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the SSBB, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store uses the same virtual address as the load.
  - The store appears in program order before the SSBB.
- When a load to a location appears in program order before the SSBB, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store uses the same virtual address as the load.
  - The store appears in program order after the SSBB.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	1	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	1	0	0	0	0	0	0

A1 variant

```
SSBB{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	1	0	0	0	0	0	0

T1 variant

```
SSBB{<q>}
```

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<q> See *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    SpeculativeStoreBypassBarrierToVA();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.214 SSUB16

Signed Subtract 16 performs two 16-bit signed integer subtractions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the subtractions.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11 10 9 8	7	6 5 4	3–0
!=1111 (cond)	0 1 1 0 0	0 0 1	Rn	Rd	(1)(1)(1)(1)	0	1 1 1	Rm

A1 variant

```
SSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7	6 5 4	3–0	15 14 13 12	11–8	7	6	5	4	3–0
1 1 1 1	1 0 1 0	1	0 1	Rn	1 1 1 1	Rd	0	0	0	0	Rm

T1 variant

```
SSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<15:0>) - SInt(R[m]<15:0>);
    diff2 = SInt(R[n]<31:16>) - SInt(R[m]<31:16>);
    R[d]<15:0>  = diff1<15:0>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
R[d]<31:16> = diff2<15:0>;
PSTATE.GE<1:0> = if diff1 >= 0 then '11' else '00';
PSTATE.GE<3:2> = if diff2 >= 0 then '11' else '00';
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.215 SSUB8

Signed Subtract 8 performs four 8-bit signed integer subtractions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the subtractions.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11 10 9 8	7	6 5 4	3–0
!=1111	0 1 1 0	0 0 1	Rn	Rd	(1)(1)(1)(1)	1	1 1 1	Rm
cond								

A1 variant

```
SSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15 14 13 12	11–8	7	6	5	4	3–0
1 1 1 1	1 0 1 0	1 0 0	Rn	1 1 1 1	Rd	0	0	0	0	Rm

T1 variant

```
SSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<7:0>) - SInt(R[m]<7:0>);
    diff2 = SInt(R[n]<15:8>) - SInt(R[m]<15:8>);
    diff3 = SInt(R[n]<23:16>) - SInt(R[m]<23:16>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
diff4 = SInt(R[n]<31:24>) - SInt(R[m]<31:24>);
R[d]<7:0>   = diff1<7:0>;
R[d]<15:8>  = diff2<7:0>;
R[d]<23:16> = diff3<7:0>;
R[d]<31:24> = diff4<7:0>;
PSTATE.GE<0>  = if diff1 >= 0 then '1' else '0';
PSTATE.GE<1>  = if diff2 >= 0 then '1' else '0';
PSTATE.GE<2>  = if diff3 >= 0 then '1' else '0';
PSTATE.GE<3>  = if diff4 >= 0 then '1' else '0';
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.216 STC

Store data to System register calculates an address from a base register value and an immediate offset, and stores a word from the DBGDTRRXint System register to memory. It can use offset, post-indexed, pre-indexed, or unindexed addressing. For information about memory accesses see *Memory accesses* on page F2-4085.

In an implementation that includes EL2, the permitted STC access to DBGDTRRXint can be trapped to Hyp mode, meaning that an attempt to execute an STC instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see *Trapping general Non-secure System register accesses to debug registers* on page G1-5803.

For simplicity, the STC pseudocode does not show this possible trap to Hyp mode.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11	10	9	8	7–0
!=1111	1	1	0	P	U	0	W	0	Rn	0	1	0	1	1	1	1	0	imm8
cond																		

Offset variant

Applies when P == 1 && W == 0.

```
STC{<c>}{<q>} p14, c5, [<Rn>{, #{+/-}<imm>}]
```

Post-indexed variant

Applies when P == 0 && W == 1.

```
STC{<c>}{<q>} p14, c5, [<Rn>], #{+/-}<imm>
```

Pre-indexed variant

Applies when P == 1 && W == 1.

```
STC{<c>}{<q>} p14, c5, [<Rn>, #{+/-}<imm>]!
```

Unindexed variant

Applies when P == 0 && U == 1 && W == 0.

```
STC{<c>}{<q>} p14, c5, [<Rn>], <option>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then UNDEFINED;
n = UInt(Rn);  cp = 14;
imm32 = ZeroExtend(imm8:'00', 32);  index = (P == '1');  add = (U == '1');  wback = (W == '1');
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If n == 15 && wback, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction executes with writeback to the PC. The instruction is handled as described in *Using R15* on page K1-7831.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 14 13 12	11 10 9 8	7 0
1 1 1 0	1 1 0 P	U	0	W	0	Rn	0 1 0 1	1 1 1 0	imm8

Offset variant

Applies when `P == 1 && W == 0`.

```
STC{<c>}{<q>} p14, c5, [<Rn>{, #{+/-}<imm>}]
```

Post-indexed variant

Applies when `P == 0 && W == 1`.

```
STC{<c>}{<q>} p14, c5, [<Rn>], #{+/-}<imm>
```

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
STC{<c>}{<q>} p14, c5, [<Rn>, #{+/-}<imm>]!
```

Unindexed variant

Applies when `P == 0 && U == 1 && W == 0`.

```
STC{<c>}{<q>} p14, c5, [<Rn>], <option>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then UNDEFINED;
n = UInt(Rn);  cp = 14;
imm32 = ZeroExtend(imm8:'00', 32);  index = (P == '1');  add = (U == '1');  wback = (W == '1');
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction executes with writeback to the PC. The instruction is handled as described in *Using R15* on page K1-7831.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rn>`	For the offset or unindexed variant: is the general-purpose base register, encoded in the "Rn" field. The PC can be used, but this is deprecated. For the offset, post-indexed or pre-indexed variant: is the general-purpose base register, encoded in the "Rn" field.
`<option>`	Is an 8-bit immediate, in the range 0 to 255 enclosed in { }, encoded in the "imm8" field. The value of this field is ignored when executing this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

+/-
: Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

  - when U = 0
  + when U = 1

<imm>
: Is the immediate offset used for forming the address, a multiple of 4 in the range 0-1020, defaulting to 0 and encoded in the "imm8" field, as <imm>/4.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];

    // System register read from DBGDTRRXint.
    MemA[address,4] = DBGDTR_EL0[];

    if wback then R[n] = offset_addr;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.217 STL

Store-Release Word stores a word from a register to memory. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page E2-3996.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-4023. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15 14 13 12	11 10	9	8	7 6 5 4	3–0
!=1111	0 0 0 1	1 0 0 0	Rn	(1) (1) (1) (1)	(1) (1)	0	0	1 0 0 1	Rt
cond									

A1 variant

```
STL{<c>}{<q>} <Rt>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15–12	11 10 9 8	7	6	5 4	3 2 1 0
1 1 1 0	1 0 0 0	1 1 0 0	Rn	Rt	(1) (1) (1) (1)	1	0	1 0	(1) (1) (1) (1)

T1 variant

```
STL{<c>}{<q>} <Rt>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    MemO[address, 4] = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.218 STLB

Store-Release Byte stores a byte from a register to memory. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page E2-3996.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-4023. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11	10	9	8	7	6	5	4	3–0
!=1111	0	0	0	1	1	1	0	0	Rn	(1)	(1)	(1)	(1)	(1)	(1)	0	0	1	0	0	1	Rt
cond																						

A1 variant

```
STLB{<c>}{<q>} <Rt>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	0	Rn	Rt	(1)	(1)	(1)	(1)	1	0	0	0	(1)	(1)	(1)	(1)

T1 variant

```
STLB{<c>}{<q>} <Rt>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    MemO[address, 1] = R[t]<7:0>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.219 STLEX

Store-Release Exclusive Word stores a word from a register to memory if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page E2-3996.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-4023. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11 10	9	8	7 6 5 4	3–0
!=1111	0 0 0 1 1	0 0	0	Rn	Rd	(1) (1)	1	0	1 0 0 1	Rt
cond									

A1 variant

```
STLEX{<c>}{<q>} <Rd>, <Rt>, [<Rn>]
```

Decode for this encoding

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15–12	11 10 9 8	7	6	5 4	3–0
1 1 1 0	1 0 0 0	1 1 0	0	Rn	Rt	(1) (1) (1) (1)	1	1	1 0	Rd

T1 variant

```
STLEX{<c>}{<q>} <Rd>, <Rt>, [<Rn>]
```

Decode for this encoding

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

- `0` If the operation updates memory.
- `1` If the operation fails to update memory.

`<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Rd>` is not updated.

A non word-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch32.ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch32.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    if AArch32.ExclusiveMonitorsPass(address,4) then
        MemO[address, 4] = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.220 STLEXB

Store-Release Exclusive Byte stores a byte from a register to memory if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page E2-3996.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-4023. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11 10	9	8	7 6 5 4	3–0
!=1111 (cond)	0 0 0 1 1	1 0 0	Rn	Rd	(1) (1)	1	0	1 0 0 1	Rt

A1 variant

```
STLEXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]
```

Decode for this encoding

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15–12	11 10 9 8	7	6	5 4	3–0
1 1 1 0	1 0 0 0	1 1 0	0	Rn	Rt	(1) (1) (1) (1)	1	1	0 0	Rd

T1 variant

```
STLEXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]
```

Decode for this encoding

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

- `0` If the operation updates memory.
- `1` If the operation fails to update memory.

`<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Rd>` is not updated.

If `AArch32.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    if AArch32.ExclusiveMonitorsPass(address,1) then
        MemO[address, 1] = R[t]<7:0>;
        R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.221 STLEXD

Store-Release Exclusive Doubleword stores a doubleword from two registers to memory if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page E2-3996.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-4023. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	0	0	0	1	1	0	1	0	Rn	Rd	(1)	(1)	1	0	1	0	0	1	Rt
cond																			

A1 variant

```
STLEXD{<c>}{<q>} <Rd>, <Rt>, <Rt2>, [<Rn>]
```

Decode for this encoding

```
d = UInt(Rd);  t = UInt(Rt);  t2 = t+1;  n = UInt(Rn);
if d == 15 || Rt<0> == '1' || t2 == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t || d == t2 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

If `Rt<0> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: Rt<0> = '0'.
- The instruction executes with the additional decode: t2 = t.
- The instruction executes as described, with no change to its behavior and no additional side effects.

If `Rt == '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction is handled as described in *Using R15* on page K1-7831.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11 … 8	7	6	5	4	3 … 0
1	1	1	0	1	0	0	0	1	1	0	0	Rn	Rt	Rt2	1	1	1	1	Rd

T1 variant

```
STLEXD{<c>}{<q>} <Rd>, <Rt>, <Rt2>, [<Rn>]
```

Decode for this encoding

```
d = UInt(Rd);  t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);
if d == 15 || t == 15 || t2 == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t || d == t2 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

- `0` If the operation updates memory.
- `1` If the operation fails to update memory.

`<Rt>` For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. `<Rt>` must be even-numbered and not R14.

For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.

`<Rt2>` For encoding A1: is the second general-purpose register to be transferred. `<Rt2>` must be `<R(t+1)>`.

For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

Aborts and alignment

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Rd>` is not updated.

A non word-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch32.ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch32.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    // Create doubleword to store such that R[t] will be stored at address and R[t2] at address+4.
    value = if BigEndian() then R[t]:R[t2] else R[t2]:R[t];
    if AArch32.ExclusiveMonitorsPass(address, 8) then
        MemO[address, 8] = value;
        R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.222 STLEXH

Store-Release Exclusive Halfword stores a halfword from a register to memory if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page E2-3996.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-4023. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11 10	9	8	7 6 5 4	3–0
!=1111 (cond)	0 0 0 1	1 1 0	Rn	Rd	(1) (1)	1	0	1 0 0 1	Rt

A1 variant

```
STLEXH{<c>}{<q>} <Rd>, <Rt>, [<Rn>]
```

Decode for this encoding

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15–12	11 10 9 8	7 6	5 4	3–0
1 1 1 0	1 0 0 0	1 1 0	0	Rn	Rt	(1) (1) (1) (1)	1 1	0 1	Rd

T1 variant

```
STLEXH{<c>}{<q>} <Rd>, <Rt>, [<Rn>]
```

Decode for this encoding

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

- `0` If the operation updates memory.
- `1` If the operation fails to update memory.

`<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated
- `<Rd>` is not updated.

A non word-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch32.ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch32.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    if AArch32.ExclusiveMonitorsPass(address,2) then
        MemO[address, 2] = R[t]<15:0>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.223 STLH

Store-Release Halfword stores a halfword from a register to memory. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page E2-3996.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-4023. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11	10	9	8	7	6	5	4	3–0
!=1111	0	0	0	1	1	1	1	0	Rn	(1)	(1)	(1)	(1)	(1)	(1)	0	0	1	0	0	1	Rt
cond																						

A1 variant

```
STLH{<c>}{<q>} <Rt>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	0	Rn	Rt	(1)	(1)	(1)	(1)	1	0	0	1	(1)	(1)	(1)	(1)

T1 variant

```
STLH{<c>}{<q>} <Rt>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    MemO[address, 2] = R[t]<15:0>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.224 STM, STMIA, STMEA

Store Multiple (Increment After, Empty Ascending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations start at this address, and the address just above the last of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC* on page F2-4086.

Armv8.2 permits the deprecation of some Store Multiple ordering behaviors in AArch32 state, for more information see ARMv8.2-LSMAOC. For details of related system instructions see STM (User registers).

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–0
!=1111	1	0	0	0	1	0	W	0	Rn	register_list
cond										

A1 variant

```
STM{IA}{<c>}{<q>} <Rn>{!}, <registers> // Preferred syntax
STMEA{<c>}{<q>} <Rn>{!}, <registers> // Alternate syntax, Empty Ascending stack
```

Decode for this encoding

```
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If `n == 15 && wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction executes with writeback to the PC. The instruction is handled as described in *Using R15* on page K1-7831.

T1

15 14 13 12	11	10–8	7–0
1 1 0 0	0	Rn	register_list

T1 variant

```
STM{IA}{<c>}{<q>} <Rn>!, <registers> // Preferred syntax
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
STMEA{<c>}{<q>} <Rn>!, <registers> // Alternate syntax, Empty Ascending stack
```

Decode for this encoding

```
n = UInt(Rn);  registers = '00000000':register_list;  wback = TRUE;
if BitCount(registers) < 1 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If `n == 15 && wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction executes with writeback to the PC. The instruction is handled as described in *Using R15* on page K1-7831.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15	14	13 … 0
1	1	1	0	1	0	0	0	1	0	W	0	Rn	(0)	M	register_list
													P		

T2 variant

```
STM{IA}{<c>}.W <Rn>{!}, <registers> // Preferred syntax, if <Rn>, '!' and <registers> can be represented
in T1
STMEA{<c>}.W <Rn>{!}, <registers> // Alternate syntax, Empty Ascending stack, if <Rn>, '!' and
<registers> can be represented in T1
STM{IA}{<c>}{<q>} <Rn>{!}, <registers> // Preferred syntax
STMEA{<c>}{<q>} <Rn>{!}, <registers> // Alternate syntax, Empty Ascending stack
```

Decode for this encoding

```
n = UInt(Rn);  registers = P:M:register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 2 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
if registers<13> == '1' then UNPREDICTABLE;
if registers<15> == '1' then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If `BitCount(registers) == 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored for the base register is UNKNOWN.

If `registers<13> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs all of the stores using the specified addressing mode but the value of R13 is UNKNOWN.

If `registers<15> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs all of the stores using the specified addressing mode but the value of R15 is UNKNOWN.

If `n == 15 && wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction executes with writeback to the PC. The instruction is handled as described in *Using R15* on page K1-7831.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`IA`	Is an optional suffix for the Increment After form.
`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rn>`	Is the general-purpose base register, encoded in the "Rn" field.
`!`	The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<registers>` For encoding A1: is a list of one or more registers to be stored, separated by commas and surrounded by { and }.

The PC can be in the list. However, Arm deprecates the use of instructions that include the PC in the list.

If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

For encoding T1: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R7, encoded in the "register_list" field. If the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

For encoding T2: is a list of one or more registers to be stored, separated by commas and surrounded by { and }.

The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain the LR. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    for i = 0 to 14
        if registers<i> == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemA[address,4] = bits(32) UNKNOWN;  // Only possible for encodings T1 and A1
            else
                MemA[address,4] = R[i];
            address = address + 4;
    if registers<15> == '1' then  // Only possible for encoding A1
        MemA[address,4] = PCStoreValue();
    if wback then R[n] = R[n] + 4*BitCount(registers);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

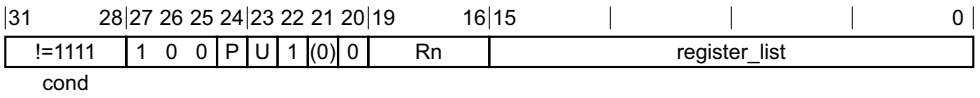
F5.1.225 STM (User registers)

In an EL1 mode other than System mode, Store Multiple (User registers) stores multiple User mode registers to consecutive memory locations using an address from a base register. The PE reads the base register value normally, using the current mode to determine the correct Banked version of the register. This instruction cannot writeback to the base register.

Store Multiple (User registers) is UNDEFINED in Hyp mode, and CONSTRAINED UNPREDICTABLE in User or System modes.

Armv8.2 permits the deprecation of some Store Multiple ordering behaviors in AArch32 state, for more information see ARMv8.2-LSMAOC.

A1



A1 variant

```
STM{<amode>}{<c>}{<q>} <Rn>, <registers>^
```

Decode for this encoding

```
n = UInt(Rn);  registers = register_list;  increment = (U == '1');  wordhigher = (P == U);
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<amode>` is one of:

`DA`	Decrement After. The consecutive memory addresses end at the address in the base register. Encoded as P = 0, U = 0.
`ED`	Empty Descending. For this instruction, a synonym for `DA`.
`DB`	Decrement Before. The consecutive memory addresses end one word below the address in the base register. Encoded as P = 1, U = 0.
`FD`	Full Descending. For this instruction, a synonym for `DB`.
`IA`	Increment After. The consecutive memory addresses start at the address in the base register. This is the default. Encoded as P = 0, U = 1.
`EA`	Empty Ascending. For this instruction, a synonym for `IA`.
`IB`	Increment Before. The consecutive memory addresses start one word above the address in the base register. Encoded as P = 1, U = 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

FA Full Ascending. For this instruction, a synonym for IB.

<c> See *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<registers> Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be stored by the STM instruction. The registers are stored with the lowest-numbered register to the lowest memory address, through to the highest-numbered register to the highest memory address. See also *Encoding of lists of general-purpose registers and the PC* on page F2-4086.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then
        UNDEFINED;
    elsif PSTATE.M IN {M32_User,M32_System} then
        UNPREDICTABLE;
    else
        length = 4*BitCount(registers);
        address = if increment then R[n] else R[n]-length;
        if wordhigher then address = address+4;
        for i = 0 to 14
            if registers<i> == '1' then  // Store User mode register
                MemA[address,4] = Rmode[i, M32_User];
                address = address + 4;
        if registers<15> == '1' then
            MemA[address,4] = PCStoreValue();
```

CONSTRAINED UNPREDICTABLE behavior

If PSTATE.M IN {M32_User,M32_System}, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.226 STMDA, STMED

Store Multiple Decrement After (Empty Descending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations end at this address, and the address just below the lowest of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC* on page F2-4086.

Armv8.2 permits the deprecation of some Store Multiple ordering behaviors in AArch32 state, for more information see ARMv8.2-LSMAOC. For details of related system instructions see STM (User registers).

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–0
!=1111	1	0	0	0	0	0	W	0	Rn	register_list
cond										

A1 variant

```
STMDA{<c>}{<q>} <Rn>{!}, <registers> // Preferred syntax
STMED{<c>}{<q>} <Rn>{!}, <registers> // Alternate syntax, Empty Descending stack
```

Decode for this encoding

```
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction targets an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If `n == 15 && wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`!` The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

`<registers>` Is a list of one or more registers to be stored, separated by commas and surrounded by { and }.

The PC can be in the list. However, Arm deprecates the use of instructions that include the PC in the list.

If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] - 4*BitCount(registers) + 4;
    for i = 0 to 14
        if registers<i> == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemA[address,4] = bits(32) UNKNOWN;
            else
                MemA[address,4] = R[i];
            address = address + 4;
    if registers<15> == '1' then
        MemA[address,4] = PCStoreValue();
    if wback then R[n] = R[n] - 4*BitCount(registers);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.227 STMDB, STMFD

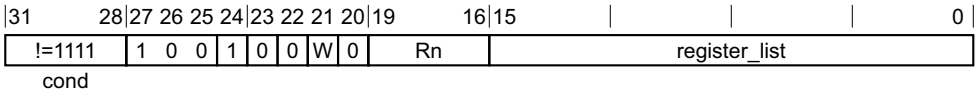
Store Multiple Decrement Before (Full Descending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations end just below this address, and the address of the first of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC* on page F2-4086.

Armv8.2 permits the deprecation of some Store Multiple ordering behaviors in AArch32 state, for more information see ARMv8.2-LSMAOC. For details of related system instructions see STM (User registers).

This instruction is used by the alias PUSH (multiple registers). See *Alias conditions* on page F5-4775 for details of when each alias is preferred.

A1



A1 variant

```
STMDB{<c>}{<q>} <Rn>{!}, <registers> // Preferred syntax
STMFD{<c>}{<q>} <Rn>{!}, <registers> // Alternate syntax, Full Descending stack
```

Decode for this encoding

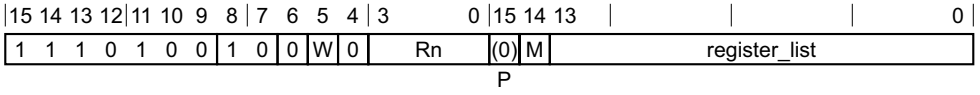
```
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

T1



T1 variant

```
STMDB{<c>}{<q>} <Rn>{!}, <registers> // Preferred syntax
STMFD{<c>}{<q>} <Rn>{!}, <registers> // Alternate syntax, Full Descending stack
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
n = UInt(Rn);  registers = P:M:register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 2 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
if registers<13> == '1' then UNPREDICTABLE;
if registers<15> == '1' then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored for the base register is UNKNOWN.

If `BitCount(registers) == 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

If `registers<13> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The store instruction performs all of the stores using the specified addressing mode but the value of R13 is UNKNOWN.

If `registers<15> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs all of the stores using the specified addressing mode but the value of R15 is UNKNOWN.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Alias conditions

Alias	of variant	is preferred when
PUSH (multiple registers)	T1	`W == '1' && Rn == '1101' && BitCount(M:register_list) > 1`
PUSH (multiple registers)	A1	`W == '1' && Rn == '1101' && BitCount(register_list) > 1`

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`!` The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

`<registers>` For encoding A1: is a list of one or more registers to be stored, separated by commas and surrounded by { and }.

The PC can be in the list. However, Arm deprecates the use of instructions that include the PC in the list.

If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

For encoding T1: is a list of one or more registers to be stored, separated by commas and surrounded by { and }.

The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain the LR. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] - 4*BitCount(registers);
    for i = 0 to 14
        if registers<i> == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemA[address,4] = bits(32) UNKNOWN;  // Only possible for encoding A1
            else
                MemA[address,4] = R[i];
            address = address + 4;
    if registers<15> == '1' then  // Only possible for encoding A1
        MemA[address,4] = PCStoreValue();
    if wback then R[n] = R[n] - 4*BitCount(registers);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.228 STMIB, STMFA

Store Multiple Increment Before (Full Ascending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations start just above this address, and the address of the last of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC* on page F2-4086.

Armv8.2 permits the deprecation of some Store Multiple ordering behaviors in AArch32 state, for more information see ARMv8.2-LSMAOC. For details of related system instructions see STM (User registers).

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–0
!=1111	1	0	0	1	1	0	W	0	Rn	register_list
cond										

A1 variant

```
STMIB{<c>}{<q>} <Rn>{!}, <registers> // Preferred syntax
STMFA{<c>}{<q>} <Rn>{!}, <registers> // Alternate syntax, Full Ascending stack
```

Decode for this encoding

```
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

If `n == 15 && wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

!
: The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

<registers>
: Is a list of one or more registers to be stored, separated by commas and surrounded by { and }.

  The PC can be in the list. However, Arm deprecates the use of instructions that include the PC in the list.

  If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + 4;
    for i = 0 to 14
        if registers<i> == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemA[address,4] = bits(32) UNKNOWN;
            else
                MemA[address,4] = R[i];
            address = address + 4;
    if registers<15> == '1' then
        MemA[address,4] = PCStoreValue();
    if wback then R[n] = R[n] + 4*BitCount(registers);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.229 STR (immediate)

Store Register (immediate) calculates an address from a base register value and an immediate offset, and stores a word from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page F2-4085.

This instruction is used by the alias PUSH (single register). See *Alias conditions* on page F5-4781 for details of when each alias is preferred.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
!=1111 (cond)	0	1	0	P	U	0	W	0	Rn	Rt	imm12

Offset variant

Applies when `P == 1 && W == 0`.

```
STR{<c>}{<q>} <Rt>, [<Rn> {, #{+/-}<imm>}]
```

Post-indexed variant

Applies when `P == 0 && W == 0`.

```
STR{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
STR{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if P == '0' && W == '1' then SEE "STRT";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15 14 13 12	11	10 … 6	5 … 3	2 … 0
0 1 1 0	0	imm5	Rn	Rt

T1 variant

```
STR{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

T2

15 14 13 12	11	10 … 8	7 … 0
1 0 0 1	0	Rt	imm8

T2 variant

```
STR{<c>}{<q>} <Rt>, [SP{, #{+}<imm>}]
```

Decode for this encoding

```
t = UInt(Rt);  n = 13;  imm32 = ZeroExtend(imm8:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

T3

15 14 13 12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11 … 0
1 1 1 1	1	0	0	0	1	1	0	0	!=1111	Rt	imm12
									Rn		

T3 variant

```
STR{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}] // <Rt>, <Rn>, <imm> can be represented in T1 or T2
STR{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 15 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T4

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7 … 0
1	1	1	1	1	0	0	0	0	1	0	0	!=1111 (Rn)	Rt	1	P	U	W	imm8

Offset variant

Applies when `P == 1 && U == 0 && W == 0`.

```
STR{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

Post-indexed variant

Applies when `P == 0 && W == 1`.

```
STR{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
STR{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if P == '1' && U == '1' && W == '0' then SEE "STRT";
if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Alias conditions

Alias	of variant	is preferred when
PUSH (single register)	A1 (pre-indexed)	`P == '1' && U == '0' && W == '1' && Rn == '1101' && imm12 == '000000000100'`
PUSH (single register)	T4 (pre-indexed)	`Rn == '1101' && U == '0' && imm8 == '00000100'`

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rt>` For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.

For encoding T1, T2, T3 and T4: is the general-purpose register to be transferred, encoded in the "Rt" field.

`<Rn>` For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.

For encoding T1, T3 and T4: is the general-purpose base register, encoded in the "Rn" field.

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

`-` when `U = 0`

`+` when `U = 1`

`+` Specifies the offset is added to the base register.

`<imm>` For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

For encoding T1: is the optional positive unsigned immediate byte offset, a multiple of 4, in the range 0 to 124, defaulting to 0 and encoded in the "imm5" field as <imm>/4.

For encoding T2: is the optional positive unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 and encoded in the "imm8" field as <imm>/4.

For encoding T3: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T4: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

Operation for all encodings

```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        MemU[address,4] = if t == 15 then PCStoreValue() else R[t];
        if wback then R[n] = offset_addr;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        MemU[address,4] = R[t];
        if wback then R[n] = offset_addr;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.230 STR (register)

Store Register (register) calculates an address from a base register value and an offset register value, stores a word from a register to memory. The offset register value can optionally be shifted. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–7	6–5	4	3–0
!=1111	0	1	1	P	U	0	W	0	Rn	Rt	imm5	stype	0	Rm
cond														

Offset variant

Applies when `P == 1 && W == 0`.

```
STR{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]
```

Post-indexed variant

Applies when `P == 0 && W == 0`.

```
STR{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
STR{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]!
```

Decode for all variants of this encoding

```
if P == '0' && W == '1' then SEE "STRT";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
if m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15 14 13 12	11	10	9	8	6 5	3 2 0
0 1 0 1	0	0	0	Rm	Rn	Rt

T1 variant

```
STR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 10 9 8	7 6 5 4	3 0
1 1 1 1	1 0 0 0	0	1 0	!=1111 Rn	Rt	0 0 0 0	0 0	imm2	Rm

T2 variant

```
STR{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // <Rt>, <Rn>, <Rm> can be represented in T1
STR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if t == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rt>` For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the general-purpose register to be transferred, encoded in the "Rt" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Rn>` For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.

For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.

`+/-` Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`+` Specifies the index register is added to the base register.

`<Rm>` Is the general-purpose index register, encoded in the "Rm" field.

`<shift>` The shift to apply to the value read from `<Rm>`. If absent, no shift is applied. Otherwise, see *Shifts applied to a register* on page F2-4083.

`<imm>` If present, the size of the left shift to apply to the value from `<Rm>`, in the range 1-3. `<imm>` is encoded in imm2. If absent, no shift is specified and imm2 is encoded as `0b00`.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    if t == 15 then  // Only possible for encoding A1
        data = PCStoreValue();
    else
        data = R[t];
    MemU[address,4] = data;
    if wback then R[n] = offset_addr;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.231 STRB (immediate)

Store Register Byte (immediate) calculates an address from a base register value and an immediate offset, and stores a byte from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
!=1111	0	1	0	P	U	1	W	0	Rn	Rt	imm12
cond											

Offset variant

Applies when `P == 1 && W == 0`.

```
STRB{<c>}{<q>} <Rt>, [<Rn> {, #{+/-}<imm>}]
```

Post-indexed variant

Applies when `P == 0 && W == 0`.

```
STRB{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
STRB{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if P == '0' && W == '1' then SEE "STRBT";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

T1

15 14 13 12	11	10 … 6	5 … 3	2 … 0
0 1 1 1	0	imm5	Rn	Rt

T1 variant

```
STRB{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

T2

15 14 13 12	11 10 9 8	7 6 5 4	3 … 0	15 … 12	11 … 0
1 1 1 1	1 0 0 0 1	0 0 0	!=1111	Rt	imm12
			Rn		

T2 variant

```
STRB{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}] // <Rt>, <Rn>, <imm> can be represented in T1
STRB{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

T3

15 14 13 12	11 10 9 8	7 6 5 4	3 … 0	15 … 12	11	10	9	8	7 … 0
1 1 1 1	1 0 0 0 0	0 0 0	!=1111	Rt	1	P	U	W	imm8
			Rn						

Offset variant

Applies when `P == 1 && U == 0 && W == 0`.

```
STRB{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Post-indexed variant

Applies when `P == 0 && W == 1`.

```
STRB{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
STRB{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if P == '1' && U == '1' && W == '0' then SEE "STRBT";
if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rt>`	Is the general-purpose register to be transferred, encoded in the "Rt" field.
`<Rn>`	For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated. For encoding T1, T2 and T3: is the general-purpose base register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

+/-
: Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

  - when U = 0
  + when U = 1

+
: Specifies the offset is added to the base register.

<imm>
: For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

  For encoding T1: is an optional 5-bit unsigned immediate byte offset, in the range 0 to 31, defaulting to 0 and encoded in the "imm5" field.

  For encoding T2: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

  For encoding T3: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

Operation for all encodings

```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        MemU[address,1] = R[t]<7:0>;
        if wback then R[n] = offset_addr;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        MemU[address,1] = R[t]<7:0>;
        if wback then R[n] = offset_addr;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.232 STRB (register)

Store Register Byte (register) calculates an address from a base register value and an offset register value, and stores a byte from a register to memory. The offset register value can optionally be shifted. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–7	6–5	4	3–0
!=1111 (cond)	0	1	1	P	U	1	W	0	Rn	Rt	imm5	stype	0	Rm

Offset variant

Applies when `P == 1 && W == 0`.

```
STRB{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]
```

Post-indexed variant

Applies when `P == 0 && W == 0`.

```
STRB{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
STRB{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]!
```

Decode for all variants of this encoding

```
if P == '0' && W == '1' then SEE "STRBT";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

T1

15 14 13 12	11	10	9	8 | 6	5 | 3	2 | 0
0 1 0 1	0	1	0	Rm	Rn	Rt

T1 variant

```
STRB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2

15 14 13 12	11 10 9 8	7 6 5 4	3 | 0	15 | 12	11 10 9 8	7 6 5 4	3 | 0
1 1 1 1	1 0 0 0 0	0 0 0	!=1111	Rt	0 0 0 0 0 0	imm2	Rm
			Rn				

T2 variant

```
STRB{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // <Rt>, <Rn>, <Rm> can be represented in T1
STRB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if t == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rt>`	Is the general-purpose register to be transferred, encoded in the "Rt" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Rn>` For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.

For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.

`+/-` Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`+` Specifies the index register is added to the base register.

`<Rm>` Is the general-purpose index register, encoded in the "Rm" field.

`<shift>` The shift to apply to the value read from `<Rm>`. If absent, no shift is applied. Otherwise, see *Shifts applied to a register* on page F2-4083.

`<imm>` If present, the size of the left shift to apply to the value from `<Rm>`, in the range 1-3. `<imm>` is encoded in imm2. If absent, no shift is specified and imm2 is encoded as `0b00`.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    MemU[address,1] = R[t]<7:0>;
    if wback then R[n] = offset_addr;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.233 STRBT

Store Register Byte Unprivileged stores a byte from a register to memory. For information about memory accesses see *Memory accesses* on page F2-4085.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

STRBT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
!=1111	0	1	0	0	U	1	1	0	Rn	Rt	imm12
cond											

A1 variant

```
STRBT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/-}<imm>}
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm12, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in *Using R15* on page K1-7831.
- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A2

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–7	6–5	4	3–0
!=1111 (cond)	0	1	1	0	U	1	1	0	Rn	Rt	imm5	stype	0	Rm

A2 variant

```
STRBT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;  (shift_t, shift_n) = DecodeImmShift(stype, imm5);
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in *Using R15* on page K1-7831.
- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7–0
1	1	1	1	1	0	0	0	0	0	0	0	!=1111 (Rn)	Rt	1	1	1	0	imm8

T1 variant

```
STRBT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rt>` For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.

For encoding A2 and T1: is the general-purpose register to be transferred, encoded in the "Rt" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`+/-` For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`<Rm>` Is the general-purpose index register, encoded in the "Rm" field.

`<shift>` The shift to apply to the value read from `<Rm>`. If absent, no shift is applied. Otherwise, see *Shifts applied to a register* on page F2-4083.

`+` Specifies the offset is added to the base register.

`<imm>` For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

Operation for all encodings

```
if ConditionPassed() then
    if PSTATE.EL == EL2 then UNPREDICTABLE;              // Hyp mode
    EncodingSpecificOperations();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
offset = if register_form then Shift(R[m], shift_t, shift_n, PSTATE.C) else imm32;
offset_addr = if add then (R[n] + offset) else (R[n] - offset);
address = if postindex then R[n] else offset_addr;
MemU_unpriv[address,1] = R[t]<7:0>;
if postindex then R[n] = offset_addr;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `STRB` (immediate).

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.234 STRD (immediate)

Store Register Dual (immediate) calculates an address from a base register value and an immediate offset, and stores two words from two registers to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
!=1111	0	0	0	P	U	1	W	0	Rn	Rt	imm4H	1	1	1	1	imm4L
cond																

Offset variant

Applies when `P == 1 && W == 0`.

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn> {, #{+/-}<imm>}]
```

Post-indexed variant

Applies when `P == 0 && W == 0`.

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], #{+/-}<imm>
```

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if Rt<0> == '1' then UNPREDICTABLE;
t = UInt(Rt);  t2 = t+1;  n = UInt(Rn);  imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if P == '0' && W == '1' then UNPREDICTABLE;
if wback && (n == 15 || n == t || n == t2) then UNPREDICTABLE;
if t2 == 15 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15 || t2 == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `wback && (n == t || n == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

If `Rt<0> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: t<0> = '0'.
- The instruction executes with the additional decode: t2 = t.
- The instruction executes as described, with no change to its behavior and no additional side-effects. This does not apply when Rt == '1111'.

If `P == '0' && W == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as an `LDRD` using one of offset, post-indexed, or pre-indexed addressing.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11 … 8	7 … 0
1	1	1	0	1	0	0	P	U	1	W	0	!=1111 (Rn)	Rt	Rt2	imm8

Offset variant

Applies when `P == 1 && W == 0`.

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn> {, #{+/-}<imm>}]
```

Post-indexed variant

Applies when `P == 0 && W == 1`.

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], #{+/-}<imm>
```

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if P == '0' && W == '0' then SEE "Related encodings";
t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if wback && (n == t || n == t2) then UNPREDICTABLE;
if n == 15 || t == 15 || t2 == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15 || t2 == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If `wback && (n == t || n == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Related encodings: *Load/store dual, load/store exclusive, load-acquire/store-release, and table branch* on page F3-4150.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rt>` For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14.

For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.

`<Rt2>` For encoding A1: is the second general-purpose register to be transferred. This register must be `<R(t+1)>`.

For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.

`<Rn>` For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.

For encoding T1: is the general-purpose base register, encoded in the "Rn" field.

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

`-` when `U = 0`

`+` when `U = 1`

`<imm>` For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is the unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 if omitted, and encoded in the "imm8" field as <imm>/4.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    if address == Align(address, 8) then
        bits(64) data;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        if BigEndian() then
            data<63:32> = R[t];
            data<31:0> = R[t2];
        else
            data<31:0> = R[t];
            data<63:32> = R[t2];
        MemA[address,8] = data;
    else
        MemA[address,4] = R[t];
        MemA[address+4,4] = R[t2];
    if wback then R[n] = offset_addr;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.235 STRD (register)

Store Register Dual (register) calculates an address from a base register value and a register offset, and stores two words from two registers to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111 (cond)	0	0	0	P	U	0	W	0	Rn	Rt	(0)	(0)	(0)	(0)	1	1	1	1	Rm

Offset variant

Applies when `P == 1 && W == 0`.

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, {+/-}<Rm>]
```

Post-indexed variant

Applies when `P == 0 && W == 0`.

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], {+/-}<Rm>
```

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, {+/-}<Rm>]!
```

Decode for all variants of this encoding

```
if Rt<0> == '1' then UNPREDICTABLE;
t = UInt(Rt);  t2 = t+1;  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if P == '0' && W == '1' then UNPREDICTABLE;
if t2 == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t || n == t2) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15 || t2 == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `wback && (n == t || n == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

If `Rt<0> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: t<0> = '0'.
- The instruction executes with the additional decode: t2 = t.
- The instruction executes as described, with no change to its behavior and no additional side-effects. This does not apply when Rt == '1111'.

If `P == '0' && W == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: P = '1'; W = '0'.
- The instruction executes with the additional decode: P = '1'; W = '1'.
- The instruction executes with the additional decode: P = '0'; W = '0'.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rt>` Is the first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14.

`<Rt2>` Is the second general-purpose register to be transferred. This register must be `<R(t+1)>`.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.

`+/-` Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`<Rm>` Is the general-purpose index register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + R[m]) else (R[n] - R[m]);
    address = if index then offset_addr else R[n];
    if address == Align(address, 8) then
        bits(64) data;
        if BigEndian() then
            data<63:32> = R[t];
            data<31:0> = R[t2];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        else
            data<31:0> = R[t];
            data<63:32> = R[t2];
        MemA[address,8] = data;
    else
        MemA[address,4] = R[t];
        MemA[address+4,4] = R[t2];
    if wback then R[n] = offset_addr;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.236 STREX

Store Register Exclusive calculates an address from a base register value and an immediate offset, stores a word from a register to the calculated address if the PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-4023. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11	10	9	8	7 6 5 4	3–0
!=1111 (cond)	0 0 0 1 1	0 0 0	Rn	Rd	(1)	(1)	1	1	1 0 0 1	Rt

A1 variant

```
STREX{<c>}{<q>} <Rd>, <Rt>, [<Rn> {, {#}<imm>}]
```

Decode for this encoding

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);  imm32 = Zeros(32); // Zero offset
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15–12	11–8	7–0
1 1 1 0	1 0 0 0	0 1 0	0	Rn	Rt	Rd	imm8

T1 variant

```
STREX{<c>}{<q>} <Rd>, <Rt>, [<Rn> {, #<imm>}]
```

Decode for this encoding

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if d == n || d == t then UNPREDICTABLE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

- `0` If the operation updates memory.
- `1` If the operation fails to update memory.

`<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`<imm>` For encoding A1: the immediate offset added to the value of `<Rn>` to calculate the address. `<imm>` can only be 0 or omitted.

For encoding T1: the immediate offset added to the value of `<Rn>` to calculate the address. `<imm>` can be omitted, meaning an offset of 0. Values are multiples of 4 in the range 0-1020.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Rd>` is not updated.

A non word-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch32.ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch32.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    if AArch32.ExclusiveMonitorsPass(address,4) then
        MemA[address,4] = R[t];
        R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.237 STREXB

Store Register Exclusive Byte derives an address from a base register value, stores a byte from a register to the derived address if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-4023. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	0	0	0	1	1	1	0	0	Rn	Rd	(1)	(1)	1	1	1	0	0	1	Rt
cond																			

A1 variant

```
STREXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]
```

Decode for this encoding

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	0	1	0	0	0	1	1	0	0	Rn	Rt	(1)	(1)	(1)	(1)	0	1	0	0	Rd

T1 variant

```
STREXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]
```

Decode for this encoding

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if d == n || d == t then UNPREDICTABLE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

- `0` If the operation updates memory.
- `1` If the operation fails to update memory.

`<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Rd>` is not updated.

If `AArch32.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    if AArch32.ExclusiveMonitorsPass(address,1) then
        MemA[address,1] = R[t]<7:0>;
        R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.238 STREXD

Store Register Exclusive Doubleword derives an address from a base register value, stores a 64-bit doubleword from two registers to the derived address if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-4023. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	0	0	0	1	1	0	1	0	Rn	Rd	(1)	(1)	1	1	1	0	0	1	Rt
cond																			

A1 variant

```
STREXD{<c>}{<q>} <Rd>, <Rt>, <Rt2>, [<Rn>]
```

Decode for this encoding

```
d = UInt(Rd);  t = UInt(Rt);  t2 = t+1;  n = UInt(Rn);
if d == 15 || Rt<0> == '1' || t2 == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t || d == t2 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

If `Rt<0> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: Rt<0> = '0'.
- The instruction executes with the additional decode: t2 = t.
- The instruction executes as described, with no change to its behavior and no additional side effects.

If `Rt == '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction is handled as described in *Using R15* on page K1-7831.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 8	7 6	5 4	3 0
1 1 1 0	1 0 0 0	1 1 0	0	Rn	Rt	Rt2	0 1	1 1	Rd

T1 variant

```
STREXD{<c>}{<q>} <Rd>, <Rt>, <Rt2>, [<Rn>]
```

Decode for this encoding

```
d = UInt(Rd);  t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);
if d == 15 || t == 15 || t2 == 15 || n == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if d == n || d == t || d == t2 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

- `0` If the operation updates memory.
- `1` If the operation fails to update memory.

`<Rd>` must not be the same as `<Rn>`, `<Rt>`, or `<Rt2>`.

`<Rt>` For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. `<Rt>` must be even-numbered and not R14.

For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.

`<Rt2>` For encoding A1: is the second general-purpose register to be transferred. `<Rt2>` must be `<R(t+1)>`.

For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Rd>` is not updated.

A non doubleword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch32.ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch32.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    // Create doubleword to store such that R[t] will be stored at address and R[t2] at address+4.
    value = if BigEndian() then R[t]:R[t2] else R[t2]:R[t];
    if AArch32.ExclusiveMonitorsPass(address,8) then
        MemA[address,8] = value;  R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.239 STREXH

Store Register Exclusive Halfword derives an address from a base register value, stores a halfword from a register to the derived address if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-4023. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	0	0	0	1	1	1	1	0	Rn	Rd	(1)	(1)	1	1	1	0	0	1	Rt
cond																			

A1 variant

```
STREXH{<c>}{<q>} <Rd>, <Rt>, [<Rn>]
```

Decode for this encoding

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	0	1	0	0	0	1	1	0	0	Rn	Rt	(1)	(1)	(1)	(1)	0	1	0	1	Rd

T1 variant

```
STREXH{<c>}{<q>} <Rd>, <Rt>, [<Rn>]
```

Decode for this encoding

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if d == n || d == t then UNPREDICTABLE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

- `0` If the operation updates memory.
- `1` If the operation fails to update memory.

`<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Rd>` is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch32.ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch32.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    if AArch32.ExclusiveMonitorsPass(address,2) then
        MemA[address,2] = R[t]<15:0>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.240 STRH (immediate)

Store Register Halfword (immediate) calculates an address from a base register value and an immediate offset, and stores a halfword from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
!=1111 (cond)	0	0	0	P	U	1	W	0	Rn	Rt	imm4H	1	0	1	1	imm4L

Offset variant

Applies when `P == 1 && W == 0`.

```
STRH{<c>}{<q>} <Rt>, [<Rn> {, #{+/-}<imm>}]
```

Post-indexed variant

Applies when `P == 0 && W == 0`.

```
STRH{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
STRH{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if P == '0' && W == '1' then SEE "STRHT";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

T1

15 14 13 12	11	10 … 6	5 … 3	2 … 0
1 0 0 0	0	imm5	Rn	Rt

T1 variant

```
STRH{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'0', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

T2

15 14 13 12	11 10 9 8	7 6 5 4	3 … 0	15 … 12	11 … 0
1 1 1 1	1 0 0 0 1	0 1 0	!=1111	Rt	imm12
			Rn		

T2 variant

```
STRH{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}] // <Rt>, <Rn>, <imm> can be represented in T1
STRH{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

T3

15 14 13 12	11 10 9 8	7 6 5 4	3 … 0	15 … 12	11	10	9	8	7 … 0
1 1 1 1	1 0 0 0 0	0 1 0	!=1111	Rt	1	P	U	W	imm8
			Rn						

Offset variant

Applies when `P == 1 && U == 0 && W == 0`.

```
STRH{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Post-indexed variant

Applies when `P == 0 && W == 1`.

```
STRH{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
STRH{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if P == '1' && U == '1' && W == '0' then SEE "STRHT";
if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.

`<Rn>` For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.

For encoding A1, T1, T2, T3: is the general-purpose base register, encoded in the "Rn" field.

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`+` Specifies the offset is added to the base register.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<imm> For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is the optional positive unsigned immediate byte offset, a multiple of 2, in the range 0 to 62, defaulting to 0 and encoded in the "imm5" field as <imm>/2.

For encoding T2: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T3: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

Operation for all encodings

```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        MemU[address,2] = R[t]<15:0>;
        if wback then R[n] = offset_addr;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        MemU[address,2] = R[t]<15:0>;
        if wback then R[n] = offset_addr;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.241 STRH (register)

Store Register Halfword (register) calculates an address from a base register value and an offset register value, and stores a halfword from a register to memory. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see *Memory accesses* on page F2-4085.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	0	0	0	P	U	0	W	0	Rn	Rt	(0)	(0)	(0)	(0)	1	0	1	1	Rm
cond																			

Offset variant

Applies when `P == 1 && W == 0`.

```
STRH{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]
```

Post-indexed variant

Applies when `P == 0 && W == 0`.

```
STRH{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>
```

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
STRH{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]!
```

Decode for all variants of this encoding

```
if P == '0' && W == '1' then SEE "STRHT";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = (SRType_LSL, 0);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

T1

15 14 13 12	11	10	9	8 | 6	5 | 3	2 | 0
0 1 0 1	0	0	1	Rm	Rn	Rt

T1 variant

```
STRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2

15 14 13 12	11 10 9 8	7 6 5 4	3 | 0	15 | 12	11 10 9 8	7 6 5 4	3 | 0
1 1 1 1	1 0 0 0 0	0 1 0	!=1111	Rt	0 0 0 0 0 0	imm2	Rm
			Rn				

T2 variant

```
STRH{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // <Rt>, <Rn>, <Rm> can be represented in T1
STRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if t == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rt>`	Is the general-purpose register to be transferred, encoded in the "Rt" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Rn>` For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.

For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.

`+/-` Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`+` Specifies the index register is added to the base register.

`<Rm>` Is the general-purpose index register, encoded in the "Rm" field.

`<imm>` If present, the size of the left shift to apply to the value from `<Rm>`, in the range 1-3. `<imm>` is encoded in imm2. If absent, no shift is specified and imm2 is encoded as `0b00`.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    MemU[address,2] = R[t]<15:0>;
    if wback then R[n] = offset_addr;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.242 STRHT

Store Register Halfword Unprivileged stores a halfword from a register to memory. For information about memory accesses see *Memory accesses* on page F2-4085.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

`STRHT` is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
!=1111	0	0	0	0	U	1	1	0	Rn	Rt	imm4H	1	0	1	1	imm4L
cond																

A1 variant

```
STRHT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/-}<imm>}
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm4H:imm4L, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in *Using R15* on page K1-7831.
- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A2

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	0	0	0	0	U	0	1	0	Rn	Rt	(0)	(0)	(0)	(0)	1	0	1	1	Rm
cond																			

A2 variant

```
STRHT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in *Using R15* on page K1-7831.
- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7–0
1	1	1	1	1	0	0	0	0	0	1	0	!=1111	Rt	1	1	1	0	imm8
												Rn						

T1 variant

```
STRHT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`+/-` For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`<Rm>` Is the general-purpose index register, encoded in the "Rm" field.

`+` Specifies the offset is added to the base register.

`<imm>` For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

Operation for all encodings

```
if ConditionPassed() then
    if PSTATE.EL == EL2 then UNPREDICTABLE;              // Hyp mode
    EncodingSpecificOperations();
    offset = if register_form then R[m] else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    MemU_unpriv[address,2] = R[t]<15:0>;
    if postindex then R[n] = offset_addr;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `STRH` (immediate).

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.243 STRT

Store Register Unprivileged stores a word from a register to memory. For information about memory accesses see *Memory accesses* on page F2-4085.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

STRT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
!=1111	0	1	0	0	U	0	1	0	Rn	Rt	imm12
cond											

A1 variant

```
STRT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/-}<imm>}
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm12, 32);
if n == 15 || n == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in *Using R15* on page K1-7831.
- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

A2

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–7	6–5	4	3–0
!=1111	0	1	1	0	U	0	1	0	Rn	Rt	imm5	stype	0	Rm
cond														

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A2 variant

```
STRT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;  (shift_t, shift_n) = DecodeImmShift(stype, imm5);
if n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in *Using R15* on page K1-7831.
- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

T1

15 14 13 12	11 10 9 8	7 6 5 4	3 … 0	15 … 12	11 10 9 8	7 … 0
1 1 1 1	1 0 0 0	1 0 0	!=1111 (Rn)	Rt	1 1 1 0	imm8

T1 variant

```
STRT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rt>` For encoding A1 and A2: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.

For encoding T1: is the general-purpose register to be transferred, encoded in the "Rt" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`+/-` For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`<Rm>` Is the general-purpose index register, encoded in the "Rm" field.

`<shift>` The shift to apply to the value read from `<Rm>`. If absent, no shift is applied. Otherwise, see *Shifts applied to a register* on page F2-4083.

`+` Specifies the offset is added to the base register.

`<imm>` For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

Operation for all encodings

```
if ConditionPassed() then
    if PSTATE.EL == EL2 then UNPREDICTABLE;               // Hyp mode
    EncodingSpecificOperations();
    offset = if register_form then Shift(R[m], shift_t, shift_n, PSTATE.C) else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    if t == 15 then  // Only possible for encodings A1 and A2
        data = PCStoreValue();
    else
        data = R[t];
    MemU_unpriv[address,4] = data;
    if postindex then R[n] = offset_addr;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `STR` (immediate).

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.244 SUB (immediate, from PC)

Subtract from PC subtracts an immediate value from the Align(PC, 4) value to form a PC-relative address, and writes the result to the destination register. Arm recommends that, where possible, software avoids using this alias

This instruction is an alias of the ADR instruction. This means that:

- The encodings in this description are named to match the encodings of ADR.
- The description of ADR gives the operational pseudocode for this instruction.

A2

31–28	27 26 25 24	23 22 21 20	19 18 17 16	15–12	11–0
!=1111 (cond)	0 0 1 0	0 1 0 0	1 1 1 1	Rd	imm12

A2 variant

```
SUB{<c>}{<q>} <Rd>, PC, #<const>
```

is equivalent to

```
ADR{<c>}{<q>} <Rd>, <label>
```

and is the preferred disassembly when `imm12 == '000000000000'`.

T2

15 14 13 12	11	10 9 8	7	6	5	4	3 2 1 0	15	14–12	11–8	7–0
1 1 1 1 0	i	1 0	1	0	1	0	1 1 1 1	0	imm3	Rd	imm8

T2 variant

```
SUB{<c>}{<q>} <Rd>, PC, #<imm12>
```

is equivalent to

```
ADR{<c>}{<q>} <Rd>, <label>
```

and is the preferred disassembly when `i:imm3:imm8 == '000000000000'`.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` For encoding A2: is the general-purpose destination register, encoded in the "Rd" field. If the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.

For encoding T2: is the general-purpose destination register, encoded in the "Rd" field.

`<label>` For encoding A2: the label of an instruction or literal data item whose address is to be loaded into `<Rd>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the `ADR` instruction to this label.

If the offset is zero or positive, encoding A1 is used, with `imm32` equal to the offset.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the offset is negative, encoding A2 is used, with `imm32` equal to the size of the offset. That is, the use of encoding A2 indicates that the required offset is minus the value of `imm32`.

Permitted values of the size of the offset are any of the constants described in *Modified immediate constants in A32 instructions* on page F2-4096.

For encoding T2: the label of an instruction or literal data item whose address is to be loaded into `<Rd>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the `ADR` instruction to this label.

If the offset is zero or positive, encoding T3 is used, with `imm32` equal to the offset.

If the offset is negative, encoding T2 is used, with `imm32` equal to the size of the offset. That is, the use of encoding T2 indicates that the required offset is minus the value of `imm32`.

Permitted values of the size of the offset are 0-4095.

`<imm12>` Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.

`<const>` An immediate value. See *Modified immediate constants in A32 instructions* on page F2-4096 for the range of values.

Operation for all encodings

The description of ADR gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.245 SUB, SUBS (immediate)

Subtract (immediate) subtracts an immediate value from a register value, and writes the result to the destination register.

If the destination register is not the PC, the SUBS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The SUB variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- The SUBS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.
  - The instruction is UNDEFINED in Hyp mode, except for encoding T5 with `<imm8>` set to zero, which is the encoding for the `ERET` instruction, see ERET.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–0
!=1111 (cond)	0 0 1 0	0 1 0 S	Rn	Rd	imm12

SUB variant

Applies when `S == 0 && Rn != 11x1`.

```
SUB{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

SUBS variant

Applies when `S == 1 && Rn != 1101`.

```
SUBS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

Decode for all variants of this encoding

```
if Rn == '1111' && S == '0' then SEE "ADR";
if Rn == '1101' then SEE "SUB (SP minus immediate)";
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = A32ExpandImm(imm12);
```

T1

15 14 13 12	11 10 9	8	6 5	3 2 0
0 0 0 1	1 1 1	1	imm3	Rn	Rd

T1 variant

```
SUB<c>{<q>} <Rd>, <Rn>, #<imm3> // Inside IT block
SUBS{<q>} <Rd>, <Rn>, #<imm3> // Outside IT block
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm3, 32);
```

T2

15 14 13	12 11	10 8	7 0
0 0 1	1 1	Rdn	imm8

T2 variant

```
SUB<c>{<q>} <Rdn>, #<imm8> // Inside IT block, and <Rdn>, <imm8> can be represented in T1
SUB<c>{<q>} {<Rdn>,} <Rdn>, #<imm8> // Inside IT block, and <Rdn>, <imm8> cannot be represented in T1
SUBS{<q>} <Rdn>, #<imm8> // Outside IT block, and <Rdn>, <imm8> can be represented in T1
SUBS{<q>} {<Rdn>,} <Rdn>, #<imm8> // Outside IT block, and <Rdn>, <imm8> cannot be represented in T1
```

Decode for this encoding

```
d = UInt(Rdn);  n = UInt(Rdn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);
```

T3

15 14 13 12	11	10	9 8	7 6 5 4	3 0	15	14 12	11 8	7 0
1 1 1 1 0	i	0	1 1 0 1	S	!=1101	0	imm3	Rd	imm8
					Rn				

SUB variant

Applies when `S == 0`.

```
SUB<c>.W {<Rd>,} <Rn>, #<const> // Inside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or
T2
SUB{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

SUBS variant

Applies when `S == 1 && Rd != 1111`.

```
SUBS.W {<Rd>,} <Rn>, #<const> // Outside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or T2
SUBS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then SEE "CMP (immediate)";
if Rn == '1101' then SEE "SUB (SP minus immediate)";
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = T32ExpandImm(i:imm3:imm8);
if (d == 15 && !setflags) || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

T4

15 14 13 12	11	10	9 8	7	6	5	4	3 0	15	14 12	11 8	7 0
1 1 1 1 0	i	1	0	1	0	1	0	!=11x1	0	imm3	Rd	imm8
								Rn				

T4 variant

```
SUB{<c>}{<q>} {<Rd>,} <Rn>, #<imm12> // <imm12> cannot be represented in T1, T2, or T3
SUBW{<c>}{<q>} {<Rd>,} <Rn>, #<imm12> // <imm12> can be represented in T1, T2, or T3
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
if Rn == '1111' then SEE "ADR";
if Rn == '1101' then SEE "SUB (SP minus immediate)";
d = UInt(Rd);  n = UInt(Rn);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

T5

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15	14	13	12	11	10	9	8	7 … 0
1	1	1	1	0	0	1	1	1	1	0	1	(1)(1)(1)(0)	1	0	(0)	0	(1)	(1)	(1)	(1)	!=00000000
												Rn									imm8

T5 variant

```
SUBS{<c>}{<q>} PC, LR, #<imm8>
```

Decode for this encoding

```
if Rn == '1110' && IsZero(imm8) then SEE "ERET";
d = 15;  n = UInt(Rn);  setflags = TRUE;  imm32 = ZeroExtend(imm8, 32);
if n != 14 then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *SUBS PC. LR and related instructions (A32)* on page K1-7844 and *SUBS PC, LR and related instructions (T32)* on page K1-7844.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rdn>` Is the general-purpose source and destination register, encoded in the "Rdn" field.

`<imm8>` For encoding T2: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.

For encoding T5: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field. If `<Rn>` is the LR, and zero is used, see ERET.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. If the PC is used:

- For the SUB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- For the SUBS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>. Arm deprecates use of this instruction unless `<Rn>` is the LR.

For encoding T1, T3 and T4: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`.

`<Rn>` For encoding A1 and T4: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see SUB, SUBS (SP minus immediate). If the PC is used, see ADR.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

For encoding T3: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see SUB, SUBS (SP minus immediate).

`<imm3>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "imm3" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<imm12>` Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.

`<const>` For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* on page F2-4096 for the range of values.

For encoding T3: an immediate value. See *Modified immediate constants in T32 instructions* on page F2-4095 for the range of values.

In the T32 instruction set, `MOVS{<c>}{<q>} PC, LR` is a pseudo-instruction for `SUBS{<c>}{<q>} PC, LR, #0`.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, nzcv) = AddWithCarry(R[n], NOT(imm32), '1');
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.246 SUB, SUBS (register)

Subtract (register) subtracts an optionally-shifted register value from a register value, and writes the result to the destination register.

If the destination register is not the PC, the SUBS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. However, when the destination register is the PC:

- The SUB variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- The SUBS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–7	6 5	4	3–0
!=1111	0 0 0 0	0 1 0 S	!=1101	Rd	imm5	stype	0	Rm
cond			Rn					

SUB, rotate right with extend variant

Applies when `S == 0 && imm5 == 00000 && stype == 11`.

```
SUB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

SUB, shift or rotate by value variant

Applies when `S == 0 && !(imm5 == 00000 && stype == 11)`.

```
SUB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

SUBS, rotate right with extend variant

Applies when `S == 1 && imm5 == 00000 && stype == 11`.

```
SUBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

SUBS, shift or rotate by value variant

Applies when `S == 1 && !(imm5 == 00000 && stype == 11)`.

```
SUBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if Rn == '1101' then SEE "SUB (SP minus register)";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15 14 13 12	11 10	9	8	6 5	3 2 0
0 0 0 1	1 0	1	Rm	Rn	Rd

T1 variant

```
SUB<c>{<q>} <Rd>, <Rn>, <Rm> // Inside IT block
SUBS{<q>} {<Rd>,} <Rn>, <Rm> // Outside IT block
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 14 12	11 8	7 6 5 4	3 0
1 1 1 0	1 0 1	1 1 0 1	S	!=1101 Rn	(0) imm3	Rd	imm2 stype	Rm

SUB, rotate right with extend variant

Applies when `S == 0 && imm3 == 000 && imm2 == 00 && stype == 11`.

```
SUB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

SUB, shift or rotate by value variant

Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11)`.

```
SUB<c>.W {<Rd>,} <Rn>, <Rm> // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
SUB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

SUBS, rotate right with extend variant

Applies when `S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11`.

```
SUBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

SUBS, shift or rotate by value variant

Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11) && Rd != 1111`.

```
SUBS.W {<Rd>,} <Rn>, <Rm> // Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
SUBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then SEE "CMP (register)";
if Rn == '1101' then SEE "SUB (SP minus register)";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if (d == 15 && !setflags) || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the SUB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- For the SUBS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1 and T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`.

`<Rn>` For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated. If the SP is used, see SUB, SUBS (SP minus register).

For encoding T1: is the first general-purpose source register, encoded in the "Rn" field.

For encoding T2: is the first general-purpose source register, encoded in the "Rn" field. If the SP is used, see SUB, SUBS (SP minus register).

`<Rm>` For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

`LSL`	when `stype = 00`
`LSR`	when `stype = 01`
`ASR`	when `stype = 10`
`ROR`	when `stype = 11`

`<amount>` For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], NOT(shifted), '1');
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.247 SUB, SUBS (register-shifted register)

Subtract (register-shifted register) subtracts a register-shifted register value from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–8	7	6 5	4	3–0
!=1111 (cond)	0 0 0 0	0 1 0 S	Rn	Rd	Rs	0	stype	1	Rm

Flag setting variant

Applies when `S == 1`.

```
SUBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

Not flag setting variant

Applies when `S == 0`.

```
SUB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

- `LSL` when `stype = 00`
- `LSR` when `stype = 01`
- `ASR` when `stype = 10`
- `ROR` when `stype = 11`

`<Rs>` Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], NOT(shifted), '1');
    R[d] = result;
    if setflags then
        PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.248 SUB, SUBS (SP minus immediate)

Subtract from SP (immediate) subtracts an immediate value from the SP value, and writes the result to the destination register.

If the destination register is not the PC, the SUBS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The SUB variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- The SUBS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

31–28	27 26 25 24	23 22 21 20	19 18 17 16	15–12	11–0
!=1111 (cond)	0 0 1 0	0 1 0 S	1 1 0 1	Rd	imm12

SUB variant

Applies when `S == 0`.

```
SUB{<c>}{<q>} {<Rd>,} SP, #<const>
```

SUBS variant

Applies when `S == 1`.

```
SUBS{<c>}{<q>} {<Rd>,} SP, #<const>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  setflags = (S == '1');  imm32 = A32ExpandImm(imm12);
```

T1

15 14 13 12	11 10 9 8	7	6–0
1 0 1 1	0 0 0 0	1	imm7

T1 variant

```
SUB{<c>}{<q>} {SP,} SP, #<imm7>
```

Decode for this encoding

```
d = 13;  setflags = FALSE;  imm32 = ZeroExtend(imm7:'00', 32);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14 12	11 8	7 0
1	1	1	1	0	i	0	1	1	0	1	S	1	1	0	1	0	imm3	Rd	imm8

SUB variant

Applies when `S == 0`.

```
SUB{<c>}.W {<Rd>,} SP, #<const> // <Rd>, <const> can be represented in T1
SUB{<c>}{<q>} {<Rd>,} SP, #<const>
```

SUBS variant

Applies when `S == 1 && Rd != 1111`.

```
SUBS{<c>}{<q>} {<Rd>,} SP, #<const>
```

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then SEE "CMP (immediate)";
d = UInt(Rd);  setflags = (S == '1');  imm32 = T32ExpandImm(i:imm3:imm8);
if d == 15 && !setflags then UNPREDICTABLE;
```

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14 12	11 8	7 0
1	1	1	1	0	i	1	0	1	0	1	0	1	1	0	1	0	imm3	Rd	imm8

T3 variant

```
SUB{<c>}{<q>} {<Rd>,} SP, #<imm12> // <imm12> cannot be represented in T1, T2, or T3
SUBW{<c>}{<q>} {<Rd>,} SP, #<imm12> // <imm12> can be represented in T1, T2, or T3
```

Decode for this encoding

```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`SP,`	Is the stack pointer.
`<imm7>`	Is the unsigned immediate, a multiple of 4, in the range 0 to 508, encoded in the "imm7" field as <imm7>/4.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP. If the PC is used:

- For the SUB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- For the SUBS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>. Arm deprecates use of this instruction unless `<Rn>` is the LR.

For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP.

`<imm12>` Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.

`<const>` For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* on page F2-4096 for the range of values.

For encoding T2: an immediate value. See *Modified immediate constants in T32 instructions* on page F2-4095 for the range of values.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, nzcv) = AddWithCarry(SP, NOT(imm32), '1');
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.249 SUB, SUBS (SP minus register)

Subtract from SP (register) subtracts an optionally-shifted register value from the SP value, and writes the result to the destination register.

If the destination register is not the PC, the SUBS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The SUB variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- The SUBS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-5726.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

31–28	27 26 25 24	23 22 21 20	19 18 17 16	15–12	11–7	6 5	4	3–0
!=1111 (cond)	0 0 0 0	0 1 0 S	1 1 0 1	Rd	imm5	stype	0	Rm

SUB, rotate right with extend variant

Applies when `S == 0 && imm5 == 00000 && stype == 11`.

```
SUB{<c>}{<q>} {<Rd>,} SP, <Rm> , RRX
```

SUB, shift or rotate by value variant

Applies when `S == 0 && !(imm5 == 00000 && stype == 11)`.

```
SUB{<c>}{<q>} {<Rd>,} SP, <Rm> {, <shift> #<amount>}
```

SUBS, rotate right with extend variant

Applies when `S == 1 && imm5 == 00000 && stype == 11`.

```
SUBS{<c>}{<q>} {<Rd>,} SP, <Rm> , RRX
```

SUBS, shift or rotate by value variant

Applies when `S == 1 && !(imm5 == 00000 && stype == 11)`.

```
SUBS{<c>}{<q>} {<Rd>,} SP, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15 14 13 12	11 10 9	8	7 6 5 4	3 2 1 0	15 14 12	11 8	7 6 5 4	3 0
1 1 1 0	1 0 1	1	1 0 1	S	1 1 0 1	(0) imm3	Rd	imm2 stype	Rm

SUB, rotate right with extend variant

Applies when `S == 0 && imm3 == 000 && imm2 == 00 && stype == 11`.

```
SUB{<c>}{<q>} {<Rd>,} SP, <Rm>, RRX
```

SUB, shift or rotate by value variant

Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11)`.

```
SUB{<c>}.W {<Rd>,} SP, <Rm> // <Rd>, <Rm> can be represented in T1 or T2
SUB{<c>}{<q>} {<Rd>,} SP, <Rm> {, <shift> #<amount>}
```

SUBS, rotate right with extend variant

Applies when `S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11`.

```
SUBS{<c>}{<q>} {<Rd>,} SP, <Rm>, RRX
```

SUBS, shift or rotate by value variant

Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11) && Rd != 1111`.

```
SUBS{<c>}{<q>} {<Rd>,} SP, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then SEE "CMP (register)";
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if (d == 15 && !setflags) || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the SUB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-3953.
- For the SUBS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP.

`<Rm>` For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For encoding T1: is the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

`LSL`	when `stype = 00`
`LSR`	when `stype = 01`
`ASR`	when `stype = 10`
`ROR`	when `stype = 11`

`<amount>` For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(SP, NOT(shifted), '1');
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.250 SVC

Supervisor Call causes a Supervisor Call exception. For more information, see *Supervisor Call (SVC) exception* on page G1-5742.

Note

SVC was previously called SWI, Software Interrupt, and this name is still found in some documentation.

Software can use this instruction as a call to an operating system to provide a service.

In the following cases, the Supervisor Call exception generated by the SVC instruction is taken to Hyp mode:

- If the SVC is executed in Hyp mode.
- If HCR.TGE is set to 1, and the SVC is executed in Non-secure User mode. For more information, see *Supervisor Call exception, when the value of HCR.TGE is 1* on page G1-5719

In these cases, the HSR identifies that the exception entry was caused by a Supervisor Call exception, EC value 0x11, see *Use of the HSR* on page G5-6038. The immediate field in the HSR:

- If the SVC is unconditional:
  - For the T32 instruction, is the zero-extended value of the imm8 field.
  - For the A32 instruction, is the least-significant 16 bits the imm24 field.
- If the SVC is conditional, is UNKNOWN.

A1

31–28	27 26 25 24	23–0
!=1111	1 1 1 1	imm24
cond		

A1 variant

```
SVC{<c>}{<q>} {#}<imm>
```

Decode for this encoding

```
imm32 = ZeroExtend(imm24, 32);
```

T1

15 14 13 12	11 10 9 8	7–0
1 1 0 1	1 1 1 1	imm8

T1 variant

```
SVC{<c>}{<q>} {#}<imm>
```

Decode for this encoding

```
imm32 = ZeroExtend(imm8, 32);
```

Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<imm>` For encoding A1: is a 24-bit unsigned immediate, in the range 0 to 16777215, encoded in the "imm24" field. This value is for assembly and disassembly only. SVC handlers in some systems interpret imm24 in software, for example to determine the required service.

For encoding T1: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field. This value is for assembly and disassembly only. SVC handlers in some systems interpret imm8 in software, for example to determine the required service.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    AArch32.CheckForSVCTrap(imm32<15:0>);
    AArch32.CallSupervisor(imm32<15:0>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.251 SXTAB

Signed Extend and Add Byte extracts an 8-bit value from a register, sign-extends it to 32 bits, adds the result to the value in another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

A1

31–28	27–20	19–16	15–12	11–10	9	8	7–4	3–0
!=1111	0 1 1 0 1 0 1 0	!=1111	Rd	rotate	(0)	(0)	0 1 1 1	Rm
cond		Rn						

A1 variant

```
SXTAB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if Rn == '1111' then SEE "SXTB";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

T1

15–12	11–8	7–6	5	4	3–0	15–12	11–8	7	6	5–4	3–0
1 1 1 1	1 0 1 0	0 1	0	0	!=1111	1 1 1 1	Rd	1	(0)	rotate	Rm
					Rn						

T1 variant

```
SXTAB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if Rn == '1111' then SEE "SXTB";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<amount>` Is the rotate amount, encoded in the "rotate" field. It can have the following values:

`(omitted)` when `rotate = 00`

`8` when `rotate = 01`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

16	when `rotate = 10`
24	when `rotate = 11`

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = R[n] + SignExtend(rotated<7:0>, 32);
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.252 SXTAB16

Signed Extend and Add Byte 16 extracts two 8-bit values from a register, sign-extends them to 16 bits each, adds the results to two 16-bit values from another register, and writes the final results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

A1

31–28	27 26 25 24	23	22 21 20	19–16	15–12	11 10	9	8	7 6 5 4	3–0
!=1111	0 1 1 0 1	0	0 0	!=1111	Rd	rotate	(0)	(0)	0 1 1 1	Rm
cond				Rn						

A1 variant

```
SXTAB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if Rn == '1111' then SEE "SXTB16";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7	6 5 4	3–0	15 14 13 12	11–8	7	6 5 4	3–0
1 1 1 1	1 0 1 0 0	0	1 0	!=1111	1 1 1 1	Rd	1	(0) rotate	Rm
				Rn					

T1 variant

```
SXTAB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if Rn == '1111' then SEE "SXTB16";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<amount>` Is the rotate amount, encoded in the "rotate" field. It can have the following values:

`(omitted)` when `rotate = 00`

`8` when `rotate = 01`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

16	when rotate = 10
24	when rotate = 11

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d]<15:0>  = R[n]<15:0> + SignExtend(rotated<7:0>, 16);
    R[d]<31:16> = R[n]<31:16> + SignExtend(rotated<23:16>, 16);
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.253 SXTAH

Signed Extend and Add Halfword extracts a 16-bit value from a register, sign-extends it to 32 bits, adds the result to a value from another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

A1

31–28	27–24	23–20	19–16	15–12	11–8	7–4	3–0
!=1111	0 1 1 0 1	0 1 1	!=1111	Rd	rotate (0) (0)	0 1 1 1	Rm
cond			Rn				

A1 variant

```
SXTAH{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if Rn == '1111' then SEE "SXTH";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

T1

15–12	11–8	7–4	3–0	15–12	11–8	7–4	3–0
1 1 1 1	1 0 1 0 0	0 0 0	!=1111	1 1 1 1	Rd	1 (0) rotate	Rm
			Rn				

T1 variant

```
SXTAH{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if Rn == '1111' then SEE "SXTH";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<amount>` Is the rotate amount, encoded in the "rotate" field. It can have the following values:

`(omitted)` when `rotate = 00`

`8` when `rotate = 01`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

16	when `rotate = 10`
24	when `rotate = 11`

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = R[n] + SignExtend(rotated<15:0>, 32);
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.254 SXTB

Signed Extend Byte extracts an 8-bit value from a register, sign-extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

A1

31–28	27 26 25 24	23	22	21 20	19 18 17 16	15–12	11 10	9	8	7 6 5 4	3–0
!=1111 (cond)	0 1 1 0	1	0	1 0	1 1 1 1	Rd	rotate	(0)	(0)	0 1 1 1	Rm

A1 variant

```
SXTB{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7	6	5–3	2–0
1 0 1 1	0 0 1 0	0	1	Rm	Rd

T1 variant

```
SXTB{<c>}{<q>} {<Rd>,} <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

T2

15 14 13 12	11 10 9 8	7	6	5	4	3 2 1 0	15 14 13 12	11–8	7	6	5 4	3–0
1 1 1 1	1 0 1 0	0	1	0	0	1 1 1 1	1 1 1 1	Rd	1	(0)	rotate	Rm

T2 variant

```
SXTB{<c>}.W {<Rd>,} <Rm> // <Rd>, <Rm> can be represented in T1
SXTB{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<Rm>`	Is the general-purpose source register, encoded in the "Rm" field.
`<amount>`	Is the rotate amount, encoded in the "rotate" field. It can have the following values:

`(omitted)`	when `rotate = 00`
`8`	when `rotate = 01`
`16`	when `rotate = 10`
`24`	when `rotate = 11`

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = SignExtend(rotated<7:0>, 32);
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.255 SXTB16

Signed Extend Byte 16 extracts two 8-bit values from a register, sign-extends them to 16 bits each, and writes the results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

A1

31–28	27–20	19–16	15–12	11–8	7–4	3–0
!=1111 (cond)	0 1 1 0 1 0 0 0	1 1 1 1	Rd	rotate (0) (0)	0 1 1 1	Rm

A1 variant

```
SXTB16{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

T1

15–12	11–8	7–4	3–0	15–12	11–8	7–4	3–0
1 1 1 1	1 0 1 0 0	0 1 0	1 1 1 1	1 1 1 1	Rd	1 (0) rotate	Rm

T1 variant

```
SXTB16{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` Is the general-purpose source register, encoded in the "Rm" field.

`<amount>` Is the rotate amount, encoded in the "rotate" field. It can have the following values:

(omitted)	when rotate = 00
8	when rotate = 01
16	when rotate = 10
24	when rotate = 11

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d]<15:0>  = SignExtend(rotated<7:0>, 16);
    R[d]<31:16> = SignExtend(rotated<23:16>, 16);
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.256 SXTH

Signed Extend Halfword extracts a 16-bit value from a register, sign-extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

A1

31–28	27–24	23	22	21–20	19–16	15–12	11–10	9	8	7–4	3–0
!=1111 (cond)	0 1 1 0	1	0	1 1	1 1 1 1	Rd	rotate	(0)	(0)	0 1 1 1	Rm

A1 variant

```
SXTH{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

T1

15–14	13–12	11–10	9–8	7	6	5–3	2–0
1 0	1 1	0 0	1 0	0	0	Rm	Rd

T1 variant

```
SXTH{<c>}{<q>} {<Rd>,} <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

T2

15–14	13–12	11–10	9–8	7	6	5	4	3–2	1–0	15–14	13–12	11–8	7	6	5–4	3–0
1 1	1 1	1 0	1 0	0	0	0	0	1 1	1 1	1 1	1 1	Rd	1	(0)	rotate	Rm

T2 variant

```
SXTH{<c>}.W {<Rd>,} <Rm> // <Rd>, <Rm> can be represented in T1
SXTH{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` Is the general-purpose source register, encoded in the "Rm" field.

`<amount>` Is the rotate amount, encoded in the "rotate" field. It can have the following values:

`(omitted)`	when `rotate = 00`
`8`	when `rotate = 01`
`16`	when `rotate = 10`
`24`	when `rotate = 11`

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = SignExtend(rotated<15:0>, 32);
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.257 TBB, TBH

Table Branch Byte or Halfword causes a PC-relative forward branch using a table of single byte or halfword offsets. A base register provides a pointer to the table, and a second register supplies an index into the table. The branch length is twice the value returned from the table.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15	14	13	12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	0	1	0	0	0	1	1	0	1	Rn	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	H	Rm

Byte variant

Applies when `H == 0`.

```
TBB{<c>}{<q>} [<Rn>, <Rm>] // Outside or last in IT block
```

Halfword variant

Applies when `H == 1`.

```
TBH{<c>}{<q>} [<Rn>, <Rm>, LSL #1] // Outside or last in IT block
```

Decode for all variants of this encoding

```
n = UInt(Rn);  m = UInt(Rm);  is_tbh = (H == '1');
if m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rn>` Is the general-purpose base register holding the address of the table of branch lengths, encoded in the "Rn" field. The PC can be used. If it is, the table immediately follows this instruction.

`<Rm>` For the byte variant: is the general-purpose index register, encoded in the "Rm" field. This register contains an integer pointing to a single byte in the table. The offset in the table is the value of the index.

For the halfword variant: is the general-purpose index register, encoded in the "Rm" field. This register contains an integer pointing to a halfword in the table. The offset in the table is twice the value of the index.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if is_tbh then
        halfwords = UInt(MemU[R[n]+LSL(R[m],1), 2]);
    else
        halfwords = UInt(MemU[R[n]+R[m], 1]);
    BranchWritePC(PC + 2*halfwords, BranchType_INDIR);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.258 TEQ (immediate)

Test Equivalence (immediate) performs a bitwise exclusive OR operation on a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15 14 13 12	11–0
!=1111 (cond)	0 0 1 1 0	0 1 1	Rn	(0)(0)(0)(0)	imm12

A1 variant

```
TEQ{<c>}{<q>} <Rn>, #<const>
```

Decode for this encoding

```
n = UInt(Rn);
(imm32, carry) = A32ExpandImm_C(imm12, PSTATE.C);
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15 14–12	11 10 9 8	7–0
1 1 1 1 0	i 0	0 1 0 0 1	Rn	0 imm3	1 1 1 1	imm8

T1 variant

```
TEQ{<c>}{<q>} <Rn>, #<const>
```

Decode for this encoding

```
n = UInt(Rn);
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
if n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rn>` For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

`<const>` For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* on page F2-4096 for the range of values.

For encoding T1: an immediate value. See *Modified immediate constants in T32 instructions* on page F2-4095 for the range of values.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] EOR imm32;
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.259 TEQ (register)

Test Equivalence (register) performs a bitwise exclusive OR operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

A1

31–28	27 26 25 24	23 22 21	20	19–16	15	14	13	12	11–7	6 5	4	3–0
!=1111 (cond)	0 0 0 1 0	0 1	1	Rn	(0)	(0)	(0)	(0)	imm5	stype	0	Rm

Rotate right with extend variant

Applies when `imm5 == 00000 && stype == 11`.

```
TEQ{<c>}{<q>} <Rn>, <Rm>, RRX
```

Shift or rotate by value variant

Applies when `!(imm5 == 00000 && stype == 11)`.

```
TEQ{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15	14–12	11 10 9 8	7 6	5 4	3–0
1 1 1 0	1 0 1	0 1 0 0	1	Rn	(0)	imm3	1 1 1 1	imm2	stype	Rm

Rotate right with extend variant

Applies when `imm3 == 000 && imm2 == 00 && stype == 11`.

```
TEQ{<c>}{<q>} <Rn>, <Rm>, RRX
```

Shift or rotate by value variant

Applies when `!(imm3 == 000 && imm2 == 00 && stype == 11)`.

```
TEQ{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rn>` For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1: is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1: is the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

`LSL` when `stype = 00`

`LSR` when `stype = 01`

`ASR` when `stype = 10`

`ROR` when `stype = 11`

`<amount>` For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] EOR shifted;
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.260 TEQ (register-shifted register)

Test Equivalence (register-shifted register) performs a bitwise exclusive OR operation on a register value and a register-shifted register value. It updates the condition flags based on the result, and discards the result.

A1

31–28	27–20	19–16	15–12	11–8	7	6–5	4	3–0
!=1111 (cond)	0 0 0 1 0 0 1 1	Rn	(0)(0)(0)(0)	Rs	0	stype	1	Rm

A1 variant

```
TEQ{<c>}{<q>} <Rn>, <Rm>, <type> <Rs>
```

Decode for this encoding

```
n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
shift_t = DecodeRegShift(stype);
if n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<type>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

`LSL`	when `stype = 00`
`LSR`	when `stype = 01`
`ASR`	when `stype = 10`
`ROR`	when `stype = 11`

`<Rs>` Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] EOR shifted;
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.261 TSB CSYNC

Trace Synchronization Barrier. This instruction is a barrier that synchronizes the trace operations of instructions.

If ARMv8.4-Trace is not implemented, this instruction executes as a `NOP`.

A1

ARMv8.4

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!=1111	0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	1	0	0	1	0
cond																												

A1 variant

```
TSB{<c>}{<q>} CSYNC
```

Decode for this encoding

```
if !HaveSelfHostedTrace() then EndOfInstruction(); // Instruction executes as NOP
if cond != '1110' then UNPREDICTABLE;              // ESB must be encoded with AL condition
```

CONSTRAINED UNPREDICTABLE behavior

If `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1

ARMv8.4

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	1	0	0	1	0

T1 variant

```
TSB{<c>}{<q>} CSYNC
```

Decode for this encoding

```
if !HaveSelfHostedTrace() then EndOfInstruction(); // Instruction executes as NOP
if InITBlock() then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes unconditionally.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The instruction executes conditionally.

Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    TraceSynchronizationBarrier();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.262 TST (immediate)

Test (immediate) performs a bitwise AND operation on a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11–0
!=1111 (cond)	0	0	1	1	0	0	0	1	Rn	(0)	(0)	(0)	(0)	imm12

A1 variant

```
TST{<c>}{<q>} <Rn>, #<const>
```

Decode for this encoding

```
n = UInt(Rn);
(imm32, carry) = A32ExpandImm_C(imm12, PSTATE.C);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15	14–12	11	10	9	8	7–0
1	1	1	1	0	i	0	0	0	0	0	1	Rn	0	imm3	1	1	1	1	imm8

T1 variant

```
TST{<c>}{<q>} <Rn>, #<const>
```

Decode for this encoding

```
n = UInt(Rn);
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
if n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rn>` For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

`<const>` For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* on page F2-4096 for the range of values.

For encoding T1: an immediate value. See *Modified immediate constants in T32 instructions* on page F2-4095 for the range of values.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] AND imm32;
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.263 TST (register)

Test (register) performs a bitwise AND operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15 14 13 12	11–7	6 5	4	3–0
!=1111	0 0 0 1 0	0 0 1	Rn	(0)(0)(0)(0)	imm5	stype	0	Rm
cond								

Rotate right with extend variant

Applies when `imm5 == 00000 && stype == 11`.

```
TST{<c>}{<q>} <Rn>, <Rm>, RRX
```

Shift or rotate by value variant

Applies when `!(imm5 == 00000 && stype == 11)`.

```
TST{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```

T1

15 14 13 12	11 10 9 8	7 6 5	3 2 0
0 1 0 0 0 0	1 0 0 0	Rm	Rn

T1 variant

```
TST{<c>}{<q>} <Rn>, <Rm>
```

Decode for this encoding

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15 14–12	11 10 9 8	7 6 5 4	3–0
1 1 1 0 1 0 1	0 0 0 0	1	Rn	(0) imm3	1 1 1 1	imm2 stype	Rm

Rotate right with extend variant

Applies when `imm3 == 000 && imm2 == 00 && stype == 11`.

```
TST{<c>}{<q>} <Rn>, <Rm>, RRX
```

Shift or rotate by value variant

Applies when `!(imm3 == 000 && imm2 == 00 && stype == 11)`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
TST{<c>}.W <Rn>, <Rm> // <Rn>, <Rm> can be represented in T1
TST{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rn>` For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

`LSL`	when `stype = 00`
`LSR`	when `stype = 01`
`ASR`	when `stype = 10`
`ROR`	when `stype = 11`

`<amount>` For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] AND shifted;
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.264 TST (register-shifted register)

Test (register-shifted register) performs a bitwise AND operation on a register value and a register-shifted register value. It updates the condition flags based on the result, and discards the result.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15 14 13 12	11–8	7	6 5	4	3–0
!=1111 (cond)	0 0 0 1 0	0 0 1	Rn	(0)(0)(0)(0)	Rs	0	stype	1	Rm

A1 variant

```
TST{<c>}{<q>} <Rn>, <Rm>, <type> <Rs>
```

Decode for this encoding

```
n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
shift_t = DecodeRegShift(stype);
if n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<type>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

`LSL`	when `stype = 00`
`LSR`	when `stype = 01`
`ASR`	when `stype = 10`
`ROR`	when `stype = 11`

`<Rs>` Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] AND shifted;
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.265 UADD16

Unsigned Add 16 performs two 16-bit unsigned integer additions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the additions.

A1

31–28	27–20	19–16	15–12	11–8	7–4	3–0
!=1111 (cond)	0 1 1 0 0 1 0 1	Rn	Rd	(1)(1)(1)(1)	0 0 0 1	Rm

A1 variant

```
UADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15–12	11–8	7–4	3–0	15–12	11–8	7–4	3–0
1 1 1 1	1 0 1 0	1 0 0 1	Rn	1 1 1 1	Rd	0 1 0 0	Rm

T1 variant

```
UADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<15:0>) + UInt(R[m]<15:0>);
    sum2 = UInt(R[n]<31:16>) + UInt(R[m]<31:16>);
    R[d]<15:0>  = sum1<15:0>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
R[d]<31:16> = sum2<15:0>;
PSTATE.GE<1:0> = if sum1 >= 0x10000 then '11' else '00';
PSTATE.GE<3:2> = if sum2 >= 0x10000 then '11' else '00';
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.266 UADD8

Unsigned Add 8 performs four unsigned 8-bit integer additions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the additions.

A1

31–28	27–20	19–16	15–12	11–8	7–4	3–0
!=1111 (cond)	0 1 1 0 0 1 0 1	Rn	Rd	(1)(1)(1)(1)	1 0 0 1	Rm

A1 variant

```
UADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15–12	11–8	7–4	3–0	15–12	11–8	7–4	3–0
1 1 1 1	1 0 1 0 1	0 0 0	Rn	1 1 1 1	Rd	0 1 0 0	Rm

T1 variant

```
UADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m  == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<7:0>) + UInt(R[m]<7:0>);
    sum2 = UInt(R[n]<15:8>) + UInt(R[m]<15:8>);
    sum3 = UInt(R[n]<23:16>) + UInt(R[m]<23:16>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
sum4 = UInt(R[n]<31:24>) + UInt(R[m]<31:24>);
R[d]<7:0>   = sum1<7:0>;
R[d]<15:8>  = sum2<7:0>;
R[d]<23:16> = sum3<7:0>;
R[d]<31:24> = sum4<7:0>;
PSTATE.GE<0>  = if sum1 >= 0x100 then '1' else '0';
PSTATE.GE<1>  = if sum2 >= 0x100 then '1' else '0';
PSTATE.GE<2>  = if sum3 >= 0x100 then '1' else '0';
PSTATE.GE<3>  = if sum4 >= 0x100 then '1' else '0';
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.267 UASX

Unsigned Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, and writes the results to the destination register. It sets PSTATE.GE according to the results.

A1

31–28	27–20	19–16	15–12	11–8	7–4	3–0
!=1111 (cond)	0 1 1 0 0 1 0 1	Rn	Rd	(1)(1)(1)(1)	0 0 1 1	Rm

A1 variant

```
UASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15–12	11–8	7–4	3–0	15–12	11–8	7–4	3–0
1 1 1 1	1 0 1 0	1	0 1 0	Rn	1 1 1 1	Rd	0 1 0 0	Rm

T1 variant

```
UASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = UInt(R[n]<15:0>) - UInt(R[m]<31:16>);
    sum  = UInt(R[n]<31:16>) + UInt(R[m]<15:0>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
R[d]<15:0>  = diff<15:0>;
R[d]<31:16> = sum<15:0>;
PSTATE.GE<1:0> = if diff >= 0 then '11' else '00';
PSTATE.GE<3:2> = if sum  >= 0x10000 then '11' else '00';
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.268 UBFX

Unsigned Bit Field Extract extracts any number of adjacent bits at any position from a register, zero-extends them to 32 bits, and writes the result to the destination register.

A1

31–28	27	26	25	24	23	22	21	20–16	15–12	11–7	6	5	4	3–0
!=1111	0	1	1	1	1	1	1	widthm1	Rd	lsb	1	0	1	Rn
cond														

A1 variant

```
UBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);
lsbit = UInt(lsb);  widthminus1 = UInt(widthm1);
if d == 15 || n == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15	14–12	11–8	7	6	5	4	0
1	1	1	1	0	(0)	1	1	1	1	0	0	Rn	0	imm3	Rd	imm2	(0)	widthm1		

T1 variant

```
UBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);
lsbit = UInt(imm3:imm2);  widthminus1 = UInt(widthm1);
if d == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rn>` Is the general-purpose source register, encoded in the "Rn" field.

`<lsb>` For encoding A1: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the "lsb" field.

For encoding T1: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the "imm3:imm2" field.

`<width>` Is the width of the field, in the range 1 to 32-<lsb>, encoded in the "widthm1" field as <width>-1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    msbit = lsbit + widthminus1;
    if msbit <= 31 then
        R[d] = ZeroExtend(R[n]<msbit:lsbit>, 32);
    else
        UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `msbit > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.269 UDF

Permanently Undefined generates an Undefined Instruction exception.

The encodings for `UDF` used in this section are defined as permanently UNDEFINED in the Armv8-A architecture. However:

- With the T32 instruction set, Arm deprecates using the `UDF` instruction in an IT block.
- In the A32 instruction set, `UDF` is not conditional.

A1

31–28	27–20	19–8	7–4	3–0
1 1 1 0	0 1 1 1 1 1 1 1	imm12	1 1 1 1	imm4
cond				

A1 variant

```
UDF{<c>}{<q>} {#}<imm>
```

Decode for this encoding

```
imm32 = ZeroExtend(imm12:imm4, 32);
// imm32 is for assembly and disassembly only, and is ignored by hardware.
```

T1

15–12	11–8	7–0
1 1 0 1	1 1 1 1	0	imm8

T1 variant

```
UDF{<c>}{<q>} {#}<imm>
```

Decode for this encoding

```
imm32 = ZeroExtend(imm8, 32);
// imm32 is for assembly and disassembly only, and is ignored by hardware.
```

T2

15–12	11–8	7–4	3–0	15–12	11–0
1 1 1 1	0 1 1 1 1 1 1	1	imm4	1 0 1 0	imm12

T2 variant

```
UDF{<c>}.W {#}<imm> // <imm> can be represented in T1
UDF{<c>}{<q>} {#}<imm>
```

Decode for this encoding

```
imm32 = ZeroExtend(imm4:imm12, 32);
// imm32 is for assembly and disassembly only, and is ignored by hardware.
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-4080. <c> must be AL or omitted.

For encoding T1 and T2: see *Standard assembler syntax fields* on page F2-4080. Arm deprecates using any <c> value other than AL.

<q> See *Standard assembler syntax fields* on page F2-4080.

<imm> For encoding A1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm12:imm4" field. The PE ignores the value of this constant.

For encoding T1: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field. The PE ignores the value of this constant.

For encoding T2: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:imm12" field. The PE ignores the value of this constant.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.270 UDIV

Unsigned Divide divides a 32-bit unsigned integer register value by a 32-bit unsigned integer register value, and writes the result to the destination register. The condition flags are not affected.

See *Divide instructions* on page F1-4047 for more information about this instruction.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–8	7 6 5 4	3–0
!=1111 (cond)	0 1 1 1 0	0 1 1	Rd	(1) (1) (1) (1) (Ra)	Rm	0 0 0 1	Rn

A1 variant

```
UDIV{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
if d == 15 || n == 15 || m == 15 || a != 15 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `Ra != '1111'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction performs a divide and the register specified by Ra becomes UNKNOWN.

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15–12	11–8	7 6 5 4	3–0
1 1 1 1	1 0 1 1	1 0 1 1	Rn	(1) (1) (1) (1) (Ra)	Rd	1 1 1 1	Rm

T1 variant

```
UDIV{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
if d == 15 || n == 15 || m == 15 || a != 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `Ra != '1111'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction performs a divide and the register specified by Ra becomes UNKNOWN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<Rn>`	Is the first general-purpose source register holding the dividend, encoded in the "Rn" field.
`<Rm>`	Is the second general-purpose source register holding the divisor, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if UInt(R[m]) == 0 then
        result = 0;
    else
        result = RoundTowardsZero(Real(UInt(R[n])) / Real(UInt(R[m])));
    R[d] = result<31:0>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.271 UHADD16

Unsigned Halving Add 16 performs two unsigned 16-bit integer additions, halves the results, and writes the results to the destination register.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11 10 9 8	7	6 5 4	3–0
!=1111 (cond)	0 1 1 0 0	1 1 1	Rn	Rd	(1)(1)(1)(1)	0	0 0 1	Rm

A1 variant

```
UHADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7	6 5 4	3–0	15 14 13 12	11–8	7	6	5	4	3–0
1 1 1 1	1 0 1 0	1	0 0 1	Rn	1 1 1 1	Rd	0	1	1	0	Rm

T1 variant

```
UHADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<15:0>) + UInt(R[m]<15:0>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
sum2 = UInt(R[n]<31:16>) + UInt(R[m]<31:16>);
R[d]<15:0>  = sum1<16:1>;
R[d]<31:16> = sum2<16:1>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.272 UHADD8

Unsigned Halving Add 8 performs four unsigned 8-bit integer additions, halves the results, and writes the results to the destination register.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11 10 9 8	7	6 5	4	3–0
!=1111 (cond)	0 1 1 0	0 1 1 1	Rn	Rd	(1)(1)(1)(1)	1	0 0	1	Rm

A1 variant

```
UHADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15 14 13 12	11–8	7	6	5	4	3–0
1 1 1 1	1 0 1 0	1	0 0 0	Rn	1 1 1 1	Rd	0	1	1	0	Rm

T1 variant

```
UHADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<7:0>) + UInt(R[m]<7:0>);
    sum2 = UInt(R[n]<15:8>) + UInt(R[m]<15:8>);
    sum3 = UInt(R[n]<23:16>) + UInt(R[m]<23:16>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
sum4 = UInt(R[n]<31:24>) + UInt(R[m]<31:24>);
R[d]<7:0>   = sum1<8:1>;
R[d]<15:8>  = sum2<8:1>;
R[d]<23:16> = sum3<8:1>;
R[d]<31:24> = sum4<8:1>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.273 UHASX

Unsigned Halving Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, halves the results, and writes the results to the destination register.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111 (cond)	0 1 1 0	0 1 1 1	Rn	Rd	(1)	(1)	(1)	(1)	0	0	1	1	Rm

A1 variant

```
UHASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15 14 13 12	11–8	7	6	5	4	3–0
1 1 1 1	1 0 1 0	1 0 1 0	Rn	1 1 1 1	Rd	0	1	1	0	Rm

T1 variant

```
UHASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = UInt(R[n]<15:0>) - UInt(R[m]<31:16>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
sum  = UInt(R[n]<31:16>) + UInt(R[m]<15:0>);
R[d]<15:0>  = diff<16:1>;
R[d]<31:16> = sum<16:1>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.274 UHSAX

Unsigned Halving Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, halves the results, and writes the results to the destination register.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111 (cond)	0 1 1 0	0 1 1 1	Rn	Rd	(1)	(1)	(1)	(1)	0	1	0	1	Rm

A1 variant

```
UHSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15 14 13 12	11–8	7	6	5	4	3–0
1 1 1 1	1 0 1 0	1 1 1 0	Rn	1 1 1 1	Rd	0	1	1	0	Rm

T1 variant

```
UHSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = UInt(R[n]<15:0>) + UInt(R[m]<31:16>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
diff = UInt(R[n]<31:16>) - UInt(R[m]<15:0>);
R[d]<15:0>  = sum<16:1>;
R[d]<31:16> = diff<16:1>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.275 UHSUB16

Unsigned Halving Subtract 16 performs two unsigned 16-bit integer subtractions, halves the results, and writes the results to the destination register.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11 10 9 8	7	6 5 4	3–0
!=1111 (cond)	0 1 1 0	0 1 1 1	Rn	Rd	(1)(1)(1)(1)	0	1 1 1	Rm

A1 variant

```
UHSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7	6 5 4	3–0	15 14 13 12	11–8	7	6 5 4	3–0
1 1 1 1	1 0 1 0	1	1 0 1	Rn	1 1 1 1	Rd	0	1 1 0	Rm

T1 variant

```
UHSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<15:0>) - UInt(R[m]<15:0>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
diff2 = UInt(R[n]<31:16>) - UInt(R[m]<31:16>);
R[d]<15:0>  = diff1<16:1>;
R[d]<31:16> = diff2<16:1>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.276 UHSUB8

Unsigned Halving Subtract 8 performs four unsigned 8-bit integer subtractions, halves the results, and writes the results to the destination register.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11 10 9 8	7	6 5 4	3–0
!=1111 (cond)	0 1 1 0	0 1 1 1	Rn	Rd	(1)(1)(1)(1)	1	1 1 1	Rm

A1 variant

```
UHSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15 14 13 12	11–8	7	6	5	4	3–0
1 1 1 1	1 0 1 0	1 0 0 1	Rn	1 1 1 1	Rd	0	1	1	0	Rm

T1 variant

```
UHSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<7:0>) - UInt(R[m]<7:0>);
    diff2 = UInt(R[n]<15:8>) - UInt(R[m]<15:8>);
    diff3 = UInt(R[n]<23:16>) - UInt(R[m]<23:16>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
diff4 = UInt(R[n]<31:24>) - UInt(R[m]<31:24>);
R[d]<7:0>   = diff1<8:1>;
R[d]<15:8>  = diff2<8:1>;
R[d]<23:16> = diff3<8:1>;
R[d]<31:24> = diff4<8:1>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.277 UMAAL

Unsigned Multiply Accumulate Accumulate Long multiplies two unsigned 32-bit values to produce a 64-bit value, adds two unsigned 32-bit values, and writes the 64-bit result to two registers.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–8	7 6 5 4	3–0
!=1111 (cond)	0 0 0 0	0 1 0 0	RdHi	RdLo	Rm	1 0 0 1	Rn

A1 variant

```
UMAAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Decode for this encoding

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15–12	11–8	7 6 5 4	3–0
1 1 1 1	1 0 1 1 1	1 1 0	Rn	RdLo	RdHi	0 1 1 0	Rm

T1 variant

```
UMAAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Decode for this encoding

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<RdLo>` Is the general-purpose source register holding the first addend and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.

`<RdHi>` Is the general-purpose source register holding the second addend and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.

`<Rn>` Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = UInt(R[n]) * UInt(R[m]) + UInt(R[dHi]) + UInt(R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.278 UMLAL, UMLALS

Unsigned Multiply Accumulate Long multiplies two unsigned 32-bit values to produce a 64-bit value, and accumulates this with a 64-bit value.

In A32 instructions, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

A1

31–28	27–24	23–21	20	19–16	15–12	11–8	7–4	3–0
!=1111 (cond)	0 0 0 0	1 0 1	S	RdHi	RdLo	Rm	1 0 0 1	Rn

Flag setting variant

Applies when `S == 1`.

```
UMLALS{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Not flag setting variant

Applies when `S == 0`.

```
UMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Decode for all variants of this encoding

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

T1

15–8	7–4	3–0	15–12	11–8	7–4	3–0
1 1 1 1 1 0 1 1 1	1 1 0	Rn	RdLo	RdHi	0 0 0 0	Rm

T1 variant

```
UMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Decode for this encoding

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<RdLo>`	Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
`<RdHi>`	Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.
`<Rn>`	Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
`<Rm>`	Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = UInt(R[n]) * UInt(R[m]) + UInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
    if setflags then
        PSTATE.N = result<63>;
        PSTATE.Z = IsZeroBit(result<63:0>);
        // PSTATE.C, PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.279 UMULL, UMULLS

Unsigned Multiply Long multiplies two 32-bit unsigned values to produce a 64-bit result.

In A32 instructions, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

A1

31–28	27–24	23–20	19–16	15–12	11–8	7–4	3–0
!=1111 (cond)	0 0 0 0	1 0 0 S	RdHi	RdLo	Rm	1 0 0 1	Rn

Flag setting variant

Applies when `S == 1`.

```
UMULLS{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Not flag setting variant

Applies when `S == 0`.

```
UMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Decode for all variants of this encoding

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

T1

15–12	11–8	7–4	3–0	15–12	11–8	7–4	3–0
1 1 1 1	1 0 1 1	1 0 1 0	Rn	RdLo	RdHi	0 0 0 0	Rm

T1 variant

```
UMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Decode for this encoding

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<RdLo>`	Is the general-purpose destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
`<RdHi>`	Is the general-purpose destination register for the upper 32 bits of the result, encoded in the "RdHi" field.
`<Rn>`	Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
`<Rm>`	Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = UInt(R[n]) * UInt(R[m]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
    if setflags then
        PSTATE.N = result<63>;
        PSTATE.Z = IsZeroBit(result<63:0>);
        // PSTATE.C, PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.280 UQADD16

Unsigned Saturating Add 16 performs two unsigned 16-bit integer additions, saturates the results to the 16-bit unsigned integer range 0 <= x <= 2^{16} - 1, and writes the results to the destination register.

A1

31–28	27–20	19–16	15–12	11–8	7–4	3–0
!=1111 (cond)	0 1 1 0 0 1 1 0	Rn	Rd	(1)(1)(1)(1)	0 0 0 1	Rm

A1 variant

```
UQADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15–12	11–8	7–4	3–0	15–12	11–8	7–4	3–0
1 1 1 1	1 0 1 0	1	0 0 1	Rn	1 1 1 1	Rd	0 1 0 1	Rm

T1 variant

```
UQADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<15:0>) + UInt(R[m]<15:0>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
sum2 = UInt(R[n]<31:16>) + UInt(R[m]<31:16>);
R[d]<15:0>  = UnsignedSat(sum1, 16);
R[d]<31:16> = UnsignedSat(sum2, 16);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.281 UQADD8

Unsigned Saturating Add 8 performs four unsigned 8-bit integer additions, saturates the results to the 8-bit unsigned integer range 0 <= x <= 2^8 - 1, and writes the results to the destination register.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	0 1 1 0	0 1 1 0	Rn	Rd	(1)	(1)	(1)	(1)	1	0	0	1	Rm
cond													

A1 variant

```
UQADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15 14 13 12	11–8	7	6	5	4	3–0
1 1 1 1	1 0 1 0	1	0 0 0	Rn	1 1 1 1	Rd	0	1	0	1	Rm

T1 variant

```
UQADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<7:0>) + UInt(R[m]<7:0>);
    sum2 = UInt(R[n]<15:8>) + UInt(R[m]<15:8>);
    sum3 = UInt(R[n]<23:16>) + UInt(R[m]<23:16>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
sum4 = UInt(R[n]<31:24>) + UInt(R[m]<31:24>);
R[d]<7:0>   = UnsignedSat(sum1, 8);
R[d]<15:8>  = UnsignedSat(sum2, 8);
R[d]<23:16> = UnsignedSat(sum3, 8);
R[d]<31:24> = UnsignedSat(sum4, 8);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.282 UQASX

Unsigned Saturating Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, saturates the results to the 16-bit unsigned integer range 0 <= x <= 2^{16} - 1, and writes the results to the destination register.

A1

31–28	27–20	19–16	15–12	11–8	7	6–4	3–0
!=1111 (cond)	0 1 1 0 0 1 1 0	Rn	Rd	(1)(1)(1)(1)	0	0 1 1	Rm

A1 variant

```
UQASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15–12	11–8	7–4	3–0	15–12	11–8	7–4	3–0
1 1 1 1	1 0 1 0	1	0 1 0	Rn	1 1 1 1	Rd	0 1 0 1	Rm

T1 variant

```
UQASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = UInt(R[n]<15:0>) - UInt(R[m]<31:16>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
sum  = UInt(R[n]<31:16>) + UInt(R[m]<15:0>);
R[d]<15:0>  = UnsignedSat(diff, 16);
R[d]<31:16> = UnsignedSat(sum, 16);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.283 UQSAX

Unsigned Saturating Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, saturates the results to the 16-bit unsigned integer range 0 <= x <= 2^{16} - 1, and writes the results to the destination register.

A1

31–28	27–20	19–16	15–12	11–8	7–4	3–0
!=1111 (cond)	0 1 1 0 0 1 1 0	Rn	Rd	(1)(1)(1)(1)	0 1 0 1	Rm

A1 variant

```
UQSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15–12	11–8	7–4	3–0	15–12	11–8	7–4	3–0
1 1 1 1	1 0 1 0	1 1 0	Rn	1 1 1 1	Rd	0 1 0 1	Rm

T1 variant

```
UQSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = UInt(R[n]<15:0>) + UInt(R[m]<31:16>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
diff = UInt(R[n]<31:16>) - UInt(R[m]<15:0>);
R[d]<15:0>  = UnsignedSat(sum, 16);
R[d]<31:16> = UnsignedSat(diff, 16);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.284 UQSUB16

Unsigned Saturating Subtract 16 performs two unsigned 16-bit integer subtractions, saturates the results to the 16-bit unsigned integer range 0 <= x <= 2^{16} - 1, and writes the results to the destination register.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11 10 9 8	7	6 5	4	3–0
!=1111 (cond)	0 1 1 0	0 1 1 0	Rn	Rd	(1)(1)(1)(1)	0	1 1	1	Rm

A1 variant

```
UQSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7	6 5	4	3–0	15 14 13 12	11–8	7	6	5	4	3–0
1 1 1 1	1 0 1 0	1	1 0	1	Rn	1 1 1 1	Rd	0	1	0	1	Rm

T1 variant

```
UQSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<15:0>) - UInt(R[m]<15:0>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
diff2 = UInt(R[n]<31:16>) - UInt(R[m]<31:16>);
R[d]<15:0>  = UnsignedSat(diff1, 16);
R[d]<31:16> = UnsignedSat(diff2, 16);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.285 UQSUB8

Unsigned Saturating Subtract 8 performs four unsigned 8-bit integer subtractions, saturates the results to the 8-bit unsigned integer range 0 <= x <= 2^8 - 1, and writes the results to the destination register.

A1

31–28	27–20	19–16	15–12	11–8	7	6–4	3–0
!=1111	0 1 1 0 0 1 1 0	Rn	Rd	(1)(1)(1)(1)	1	1 1 1	Rm
cond							

A1 variant

```
UQSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15–12	11–8	7–4	3–0	15–12	11–8	7–4	3–0
1 1 1 1	1 0 1 0	1 0 0 1	Rn	1 1 1 1	Rd	0 1 0 1	Rm

T1 variant

```
UQSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<7:0>) - UInt(R[m]<7:0>);
    diff2 = UInt(R[n]<15:8>) - UInt(R[m]<15:8>);
    diff3 = UInt(R[n]<23:16>) - UInt(R[m]<23:16>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
diff4 = UInt(R[n]<31:24>) - UInt(R[m]<31:24>);
R[d]<7:0>   = UnsignedSat(diff1, 8);
R[d]<15:8>  = UnsignedSat(diff2, 8);
R[d]<23:16> = UnsignedSat(diff3, 8);
R[d]<31:24> = UnsignedSat(diff4, 8);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.286 USAD8

Unsigned Sum of Absolute Differences performs four unsigned 8-bit subtractions, and adds the absolute values of the differences together.

A1

31–28	27–20	19–16	15–12	11–8	7–4	3–0
!=1111 (cond)	0 1 1 1 1 0 0 0	Rd	1 1 1 1	Rm	0 0 0 1	Rn

A1 variant

```
USAD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15–12	11–8	7–4	3–0	15–12	11–8	7–4	3–0
1 1 1 1	1 0 1 1 0	1 1 1	Rn	1 1 1 1	Rd	0 0 0 0	Rm

T1 variant

```
USAD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    absdiff1 = Abs(UInt(R[n]<7:0>)   - UInt(R[m]<7:0>));
    absdiff2 = Abs(UInt(R[n]<15:8>)  - UInt(R[m]<15:8>));
    absdiff3 = Abs(UInt(R[n]<23:16>) - UInt(R[m]<23:16>));
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
absdiff4 = Abs(UInt(R[n]<31:24>) - UInt(R[m]<31:24>));
result = absdiff1 + absdiff2 + absdiff3 + absdiff4;
R[d] = result<31:0>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.287 USADA8

Unsigned Sum of Absolute Differences and Accumulate performs four unsigned 8-bit subtractions, and adds the absolute values of the differences to a 32-bit accumulate operand.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–8	7 6 5 4	3–0
!=1111	0 1 1 1	1 0 0 0	Rd	!=1111	Rm	0 0 0 1	Rn
cond				Ra			

A1 variant

```
USADA8{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for this encoding

```
if Ra == '1111' then SEE "USAD8";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15–12	11–8	7 6	5 4	3–0
1 1 1 1	1 0 1 1	0 1 1 1	Rn	!=1111	Rd	0 0	0 0	Rm
				Ra				

T1 variant

```
USADA8{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for this encoding

```
if Ra == '1111' then SEE "USAD8";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.
<Ra>	Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    absdiff1 = Abs(UInt(R[n]<7:0>)   - UInt(R[m]<7:0>));
    absdiff2 = Abs(UInt(R[n]<15:8>)  - UInt(R[m]<15:8>));
    absdiff3 = Abs(UInt(R[n]<23:16>) - UInt(R[m]<23:16>));
    absdiff4 = Abs(UInt(R[n]<31:24>) - UInt(R[m]<31:24>));
    result = UInt(R[a]) + absdiff1 + absdiff2 + absdiff3 + absdiff4;
    R[d] = result<31:0>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.288 USAT

Unsigned Saturate saturates an optionally-shifted signed value to a selected unsigned range.

This instruction sets PSTATE.Q to 1 if the operation saturates.

A1

31–28	27	26	25	24	23	22	21	20–16	15–12	11–7	6	5	4	3–0
!=1111	0	1	1	0	1	1	1	sat_imm	Rd	imm5	sh	0	1	Rn
cond														

Arithmetic shift right variant

Applies when `sh == 1`.

```
USAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>
```

Logical shift left variant

Applies when `sh == 0`.

```
USAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm);
(shift_t, shift_n) = DecodeImmShift(sh:'0', imm5);
if d == 15 || n == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15	14–12	11–8	7	6	5	4	0
1	1	1	1	0	(0)	1	1	1	0	sh	0	Rn	0	imm3	Rd	imm2	(0)	sat_imm		

Arithmetic shift right variant

Applies when `sh == 1 && !(imm3 == 000 && imm2 == 00)`.

```
USAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>
```

Logical shift left variant

Applies when `sh == 0`.

```
USAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}
```

Decode for all variants of this encoding

```
if sh == '1' && (imm3:imm2) == '00000' then SEE "USAT16";
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm);
(shift_t, shift_n) = DecodeImmShift(sh:'0', imm3:imm2);
if d == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<imm>` Is the bit position for saturation, in the range 0 to 31, encoded in the "sat_imm" field.

`<Rn>` Is the general-purpose source register, encoded in the "Rn" field.

`<amount>` For encoding A1: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm5" field.

For encoding A1: is the shift amount, in the range 1 to 32 encoded in the "imm5" field as <amount> modulo 32.

For encoding T1: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm3:imm2" field.

For encoding T1: is the shift amount, in the range 1 to 31 encoded in the "imm3:imm2" field as <amount>.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand = Shift(R[n], shift_t, shift_n, PSTATE.C);  // PSTATE.C ignored
    (result, sat) = UnsignedSatQ(SInt(operand), saturate_to);
    R[d] = ZeroExtend(result, 32);
    if sat then
        PSTATE.Q = '1';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.289 USAT16

Unsigned Saturate 16 saturates two signed 16-bit values to a selected unsigned range.

This instruction sets PSTATE.Q to 1 if the operation saturates.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	0	1	1	0	1	1	1	0	sat_imm	Rd	(1)	(1)	(1)	(1)	0	0	1	1	Rn
cond																			

A1 variant

```
USAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm);
if d == 15 || n == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15	14	13	12	11–8	7	6	5	4	3–0
1	1	1	1	0	(0)	1	1	1	0	1	0	Rn	0	0	0	0	Rd	0	0	(0)	(0)	sat_imm

T1 variant

```
USAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm);
if d == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<imm>	Is the bit position for saturation, in the range 0 to 15, encoded in the "sat_imm" field.
<Rn>	Is the general-purpose source register, encoded in the "Rn" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result1, sat1) = UnsignedSatQ(SInt(R[n]<15:0>), saturate_to);
    (result2, sat2) = UnsignedSatQ(SInt(R[n]<31:16>), saturate_to);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
R[d]<15:0> = ZeroExtend(result1, 16);
R[d]<31:16> = ZeroExtend(result2, 16);
if sat1 || sat2 then
    PSTATE.Q = '1';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.290 USAX

Unsigned Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, and writes the results to the destination register. It sets PSTATE.GE according to the results.

A1

31–28	27–20	19–16	15–12	11–8	7–4	3–0
!=1111 (cond)	0 1 1 0 0 1 0 1	Rn	Rd	(1)(1)(1)(1)	0 1 0 1	Rm

A1 variant

```
USAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15–12	11–8	7–4	3–0	15–12	11–8	7–4	3–0
1 1 1 1	1 0 1 0 1	1 1 0	Rn	1 1 1 1	Rd	0 1 0 0	Rm

T1 variant

```
USAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = UInt(R[n]<15:0>) + UInt(R[m]<31:16>);
    diff = UInt(R[n]<31:16>) - UInt(R[m]<15:0>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
R[d]<15:0>  = sum<15:0>;
R[d]<31:16> = diff<15:0>;
PSTATE.GE<1:0> = if sum  >= 0x10000 then '11' else '00';
PSTATE.GE<3:2> = if diff >= 0 then '11' else '00';
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.291 USUB16

Unsigned Subtract 16 performs two 16-bit unsigned integer subtractions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the subtractions.

A1

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11 10 9 8	7 6 5 4	3–0
!=1111 (cond)	0 1 1 0 0	1 0 1	Rn	Rd	(1)(1)(1)(1)	0 1 1 1	Rm

A1 variant

```
USUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15 14 13 12	11–8	7 6 5 4	3–0
1 1 1 1	1 0 1 0	1 0 1	Rn	1 1 1 1	Rd	0 1 0 0	Rm

T1 variant

```
USUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<15:0>) - UInt(R[m]<15:0>);
    diff2 = UInt(R[n]<31:16>) - UInt(R[m]<31:16>);
    R[d]<15:0>  = diff1<15:0>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
R[d]<31:16> = diff2<15:0>;
PSTATE.GE<1:0> = if diff1 >= 0 then '11' else '00';
PSTATE.GE<3:2> = if diff2 >= 0 then '11' else '00';
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.292 USUB8

Unsigned Subtract 8 performs four 8-bit unsigned integer subtractions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the subtractions.

A1

31–28	27–20	19–16	15–12	11–8	7–4	3–0
!=1111 (cond)	0 1 1 0 0 1 0 1	Rn	Rd	(1)(1)(1)(1)	1 1 1 1	Rm

A1 variant

```
USUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15–12	11–8	7–4	3–0	15–12	11–8	7–4	3–0
1 1 1 1	1 0 1 0 1	1 0 0	Rn	1 1 1 1	Rd	0 1 0 0	Rm

T1 variant

```
USUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<7:0>) - UInt(R[m]<7:0>);
    diff2 = UInt(R[n]<15:8>) - UInt(R[m]<15:8>);
    diff3 = UInt(R[n]<23:16>) - UInt(R[m]<23:16>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
diff4 = UInt(R[n]<31:24>) - UInt(R[m]<31:24>);
R[d]<7:0>   = diff1<7:0>;
R[d]<15:8>  = diff2<7:0>;
R[d]<23:16> = diff3<7:0>;
R[d]<31:24> = diff4<7:0>;
PSTATE.GE<0>  = if diff1 >= 0 then '1' else '0';
PSTATE.GE<1>  = if diff2 >= 0 then '1' else '0';
PSTATE.GE<2>  = if diff3 >= 0 then '1' else '0';
PSTATE.GE<3>  = if diff4 >= 0 then '1' else '0';
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.293 UXTAB

Unsigned Extend and Add Byte extracts an 8-bit value from a register, zero-extends it to 32 bits, adds the result to the value in another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

A1

31–28	27–20	19–16	15–12	11–10	9	8	7–4	3–0
!=1111 (cond)	0 1 1 0 1 1 1 0	!=1111 (Rn)	Rd	rotate	(0)	(0)	0 1 1 1	Rm

A1 variant

```
UXTAB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if Rn == '1111' then SEE "UXTB";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

T1

15–12	11–8	7–4	3–0	15–12	11–8	7	6	5–4	3–0
1 1 1 1	1 0 1 0	0 1 0 1	!=1111 (Rn)	1 1 1 1	Rd	1	(0)	rotate	Rm

T1 variant

```
UXTAB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if Rn == '1111' then SEE "UXTB";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<amount>` Is the rotate amount, encoded in the "rotate" field. It can have the following values:

`(omitted)` when `rotate = 00`

`8` when `rotate = 01`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

16	when `rotate = 10`
24	when `rotate = 11`

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = R[n] + ZeroExtend(rotated<7:0>, 32);
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.294 UXTAB16

Unsigned Extend and Add Byte 16 extracts two 8-bit values from a register, zero-extends them to 16 bits each, adds the results to two 16-bit values from another register, and writes the final results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

A1

31–28	27 26 25 24	23	22 21 20	19–16	15–12	11 10	9	8	7 6 5 4	3–0
!=1111	0 1 1 0 1	1	0 0	!=1111	Rd	rotate	(0)	(0)	0 1 1 1	Rm
cond				Rn						

A1 variant

```
UXTAB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if Rn == '1111' then SEE "UXTB16";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3–0	15 14 13 12	11–8	7	6	5 4	3–0
1 1 1 1	1 0 1 0	0 1	1	!=1111	1 1 1 1	Rd	1	(0)	rotate	Rm
				Rn						

T1 variant

```
UXTAB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if Rn == '1111' then SEE "UXTB16";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<amount>` Is the rotate amount, encoded in the "rotate" field. It can have the following values:

`(omitted)` when `rotate = 00`

`8` when `rotate = 01`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

16	when `rotate = 10`
24	when `rotate = 11`

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d]<15:0>  = R[n]<15:0> + ZeroExtend(rotated<7:0>, 16);
    R[d]<31:16> = R[n]<31:16> + ZeroExtend(rotated<23:16>, 16);
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.295 UXTAH

Unsigned Extend and Add Halfword extracts a 16-bit value from a register, zero-extends it to 32 bits, adds the result to a value from another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

A1

31–28	27–20	19–16	15–12	11–10	9	8	7–4	3–0
!=1111 (cond)	0 1 1 0 1 1 1 1	!=1111 (Rn)	Rd	rotate	(0)	(0)	0 1 1 1	Rm

A1 variant

```
UXTAH{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if Rn == '1111' then SEE "UXTH";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

T1

15–12	11–8	7–4	3–0	15–12	11–8	7	6	5–4	3–0
1 1 1 1	1 0 1 0	0 0 1	!=1111 (Rn)	1 1 1 1	Rd	1	(0)	rotate	Rm

T1 variant

```
UXTAH{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if Rn == '1111' then SEE "UXTH";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<amount>` Is the rotate amount, encoded in the "rotate" field. It can have the following values:

`(omitted)` when `rotate = 00`

`8` when `rotate = 01`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

16	when `rotate = 10`
24	when `rotate = 11`

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = R[n] + ZeroExtend(rotated<15:0>, 32);
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.296 UXTB

Unsigned Extend Byte extracts an 8-bit value from a register, zero-extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111 (cond)	0	1	1	0	1	1	1	0	1	1	1	1	Rd	rotate		(0)	(0)	0	1	1	1	Rm

A1 variant

```
UXTB{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5–3	2–0
1	0	1	1	0	0	1	0	1	1	Rm	Rd

T1 variant

```
UXTB{<c>}{<q>} {<Rd>,} <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11–8	7	6	5	4	3–0
1	1	1	1	1	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	Rd	1	(0)	rotate		Rm

T2 variant

```
UXTB{<c>}.W {<Rd>,} <Rm> // <Rd>, <Rm> can be represented in T1
UXTB{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<Rm>`	Is the general-purpose source register, encoded in the "Rm" field.
`<amount>`	Is the rotate amount, encoded in the "rotate" field. It can have the following values:

`(omitted)`	when `rotate = 00`
`8`	when `rotate = 01`
`16`	when `rotate = 10`
`24`	when `rotate = 11`

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = ZeroExtend(rotated<7:0>, 32);
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.297 UXTB16

Unsigned Extend Byte 16 extracts two 8-bit values from a register, zero-extends them to 16 bits each, and writes the results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

A1

31–28	27–20	19–16	15–12	11–8	7–4	3–0
!=1111 (cond)	0 1 1 0 1 1 0 0	1 1 1 1	Rd	rotate (0) (0)	0 1 1 1	Rm

A1 variant

```
UXTB16{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

T1

15–12	11–8	7–4	3–0	15–12	11–8	7–4	3–0
1 1 1 1	1 0 1 0 0	0 1 1	1 1 1 1	1 1 1 1	Rd	1 (0) rotate	Rm

T1 variant

```
UXTB16{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` For encoding A1: is the general-purpose source register, encoded in the "Rm" field.

For encoding T1: is the second general-purpose source register, encoded in the "Rm" field.

`<amount>` Is the rotate amount, encoded in the "rotate" field. It can have the following values:

`(omitted)`	when `rotate = 00`
`8`	when `rotate = 01`
`16`	when `rotate = 10`
`24`	when `rotate = 11`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d]<15:0>  = ZeroExtend(rotated<7:0>, 16);
    R[d]<31:16> = ZeroExtend(rotated<23:16>, 16);
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.298 UXTH

Unsigned Extend Halfword extracts a 16-bit value from a register, zero-extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

A1

31–28	27–24	23	22	21–20	19–16	15–12	11–10	9	8	7–4	3–0
!=1111 (cond)	0 1 1 0	1	1	1 1	1 1 1 1	Rd	rotate	(0)	(0)	0 1 1 1	Rm

A1 variant

```
UXTH{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

T1

15–12	11–8	7	6	5–3	2–0
1 0 1 1	0 0 1 0	1	0	Rm	Rd

T1 variant

```
UXTH{<c>}{<q>} {<Rd>,} <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

T2

15–12	11–8	7–4	3–0	15–12	11–8	7	6–5	4	3–0
1 1 1 1	1 0 1 0	0 0 1	1 1 1 1	1 1 1 1	Rd	1	(0)	rotate	Rm

T2 variant

```
UXTH{<c>}.W {<Rd>,} <Rm> // <Rd>, <Rm> can be represented in T1
UXTH{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` Is the general-purpose source register, encoded in the "Rm" field.

`<amount>` Is the rotate amount, encoded in the "rotate" field. It can have the following values:

`(omitted)`	when `rotate = 00`
`8`	when `rotate = 01`
`16`	when `rotate = 10`
`24`	when `rotate = 11`

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = ZeroExtend(rotated<15:0>, 32);
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.299 WFE

Wait For Event is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. Wakeup events include the event signaled as a result of executing the `SEV` instruction on any PE in the multiprocessor system. For more information, see *Wait For Event and Send Event* on page G1-5764.

As described in *Wait For Event and Send Event* on page G1-5764, the execution of a `WFE` instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level, see:

- *Traps to Undefined mode of EL0 execution of WFE and WFI instructions* on page G1-5781.
- *Traps to Hyp mode of Non-secure EL0 and EL1 execution of WFE and WFI instructions* on page G1-5797.
- *Traps to Monitor mode of the execution of WFE and WFI instructions in modes other than Monitor mode* on page G1-5808.

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!=1111	0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	0	0	1	0
cond																												

A1 variant

```
WFE{<c>}{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	1	0	0	0	0	0

T1 variant

```
WFE{<c>}{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	0	1	0

T2 variant

```
WFE{<c>}.W
```

Decode for this encoding

```
// No additional decoding required
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if IsEventRegisterSet() then
        ClearEventRegister();
    else
        if PSTATE.EL == EL0 then
            // Check for traps described by the OS which may be EL1 or EL2.
            AArch32.CheckForWFxTrap(EL1, TRUE);
        if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
            // Check for traps described by the Hypervisor.
            AArch32.CheckForWFxTrap(EL2, TRUE);
        if HaveEL(EL3) && PSTATE.M != M32_Monitor then
            // Check for traps described by the Secure Monitor.
            AArch32.CheckForWFxTrap(EL3, TRUE);
        WaitForEvent();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.300 WFI

Wait For Interrupt is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. For more information, see *Wait For Interrupt* on page G1-5767.

As described in *Wait For Interrupt* on page G1-5767, the execution of a `WFI` instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level, see:

- *Traps to Undefined mode of EL0 execution of WFE and WFI instructions* on page G1-5781.
- *Traps to Hyp mode of Non-secure EL0 and EL1 execution of WFE and WFI instructions* on page G1-5797.
- *Traps to Monitor mode of the execution of WFE and WFI instructions in modes other than Monitor mode* on page G1-5808.

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!=1111	0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	0	0	1	1
cond																												

A1 variant

```
WFI{<c>}{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	1	1	0	0	0	0

T1 variant

```
WFI{<c>}{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	0	1	1

T2 variant

```
WFI{<c>}.W
```

Decode for this encoding

```
// No additional decoding required
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !InterruptPending() then
        if PSTATE.EL == EL0 then
            // Check for traps described by the OS which may be EL1 or EL2.
            AArch32.CheckForWFxTrap(EL1, FALSE);
        if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
            // Check for traps described by the Hypervisor.
            AArch32.CheckForWFxTrap(EL2, FALSE);
        if HaveEL(EL3) && PSTATE.M != M32_Monitor then
            // Check for traps described by the Secure Monitor.
            AArch32.CheckForWFxTrap(EL3, FALSE);
        WaitForInterrupt();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.1.301 YIELD

`YIELD` is a hint instruction. Software with a multithreading capability can use a `YIELD` instruction to indicate to the PE that it is performing a task, for example a spin-lock, that could be swapped out to improve overall system performance. The PE can use this hint to suspend and resume multiple software threads if it supports the capability.

For more information about the recommended use of this instruction see *The Yield instruction* on page F1-4053.

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!=1111	0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	0	0	0	1
cond																												

A1 variant

```
YIELD{<c>}{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	0	1	0	0	0	0

T1 variant

```
YIELD{<c>}{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	0	0	1

T2 variant

```
YIELD{<c>}.W
```

Decode for this encoding

```
// No additional decoding required
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Hint_Yield();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F5.2 Encoding and use of banked register transfer instructions

Software executing at EL1 or higher can use the `MRS` (banked register) and `MSR` (banked register) instructions to transfer values between the general-purpose registers and Special-purpose registers. One particular use of these instructions is for a hypervisor to save or restore the register values of a Guest OS. The following sections give more information about these instructions:

- *Register arguments in the banked register transfer instructions*.
- *Usage restrictions on the banked register transfer instructions* on page F5-4952.
- *Encoding the register argument in the banked register transfer instructions* on page F5-4953.
- *Pseudocode support for the banked register transfer instructions* on page F5-4954.

For descriptions of the instructions see *MRS (Banked register)* on page F5-4534 and *MSR (Banked register)* on page F5-4538.

F5.2.1 Register arguments in the banked register transfer instructions

Figure F5-1 shows the banked general-purpose registers and Special-purpose registers:

	Associated PE mode							
	User or System	Hyp	Supervisor	Abort	Undefined	Monitor	IRQ	FIQ
General-purpose registers	R8_usr							R8_fiq
	R9_usr							R9_fiq
	R10_usr							R10_fiq
	R11_usr							R11_fiq
	R12_usr							R12_fiq
	SP_usr	SP_hyp	SP_svc	SP_abt	SP_und	SP_mon	SP_irq	SP_fiq
	LR_usr		LR_svc	LR_abt	LR_und	LR_mon	LR_irq	LR_fiq
Special-purpose registers		SPSR_hyp	SPSR_svc	SPSR_abt	SPSR_und	SPSR_mon	SPSR_irq	SPSR_fiq
		ELR_hyp						

For the general-purpose registers, if no other register is shown, the *current mode register* is the _usr register. So, for example, the full set of current mode registers, including the registers that are not banked:

- For Hyp mode, is {R0_usr - R12_usr, SP_hyp, LR_usr, SPSR_hyp, ELR_hyp}.
- For Abort mode, is {R0_usr - R12_usr, SP_abt, LR_abt, SPSR_abt}.

Figure F5-1 Banking of general-purpose and Special-purpose registers

Figure F5-1 is based on Figure G1-2 on page G1-5689, that shows the complete set of general-purpose registers and Special-purpose registers accessible in each mode.

Note

- System mode uses the same set of registers as User mode. Neither of these modes can access an SPSR, except that System mode can use the `MRS` (banked register) and `MSR` (banked register) instructions to access some SPSRs, as described in *Usage restrictions on the banked register transfer instructions* on page F5-4952.
- General-purpose registers R0-R7, that are not banked, cannot be accessed using the `MRS` (banked register) and `MSR` (banked register) instructions.
- In addition to the registers shown in Figure F5-1, the DLR and DSPSR are AArch32 System registers that map onto the AArch64 Special-purpose registers DLR_EL0 and DSPSR_EL0. However, DLR and DSPSR are not accessible using the `MRS` (banked register) and `MSR` (banked register) instructions.

Software using an `MRS` (banked register) or `MSR` (banked register) instruction specifies one of these registers using a name shown in Figure F5-1, or an alternative name for SP or LR. These registers can be grouped as follows:

R8-R12 Each of these registers has two banked copies, _usr and _fiq, for example R8_usr and R8_fiq.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

SP There is a banked copy of SP for every mode except System mode. For example, SP_svc is the SP for Supervisor mode.

LR There is a banked copy of LR for every mode except System mode and Hyp mode. For example, LR_svc is the LR for Supervisor mode.

SPSR There is a banked copy of SPSR for every mode except System mode and User mode.

ELR_hyp Except for the operations provided by `MRS` (banked register) and `MSR` (banked register), ELR_hyp is accessible only from Hyp mode. It is not banked.

F5.2.2 Usage restrictions on the banked register transfer instructions

`MRS` (banked register) and `MSR` (banked register) instructions are CONSTRAINED UNPREDICTABLE if any of the following applies:

- The instruction is executed in User mode.
- The instruction accesses a banked register that is not implemented, or that either:
  - Is not accessible from the current Privilege level and Security state.
  - Can be accessed from the current mode by using a different instruction.

MSR (banked register) and MRS (banked register) on page K1-7853 describes the permitted CONSTRAINED UNPREDICTABLE behavior.

An `MRS` (banked register) instruction or an `MSR` (banked register) instruction executed:

- At Non-secure EL1 cannot access any Hyp mode banked registers.
- At Non-secure EL1 or EL2 cannot access any Monitor mode banked registers.
- In a Secure mode other than Monitor mode cannot access any Hyp banked registers.

This means that the banked registers that `MRS` (banked register) and `MSR` (banked register) instructions cannot access are:

From Monitor mode

- The current mode registers R8_usr-R12_usr, SP_mon, LR_mon, and SPSR_mon.

From Hyp mode

- The Monitor mode registers SP_mon, LR_mon, and SPSR_mon.
- The current mode registers R8_usr-R12_usr, SP_hyp, LR_usr, and SPSR_hyp.

Note

`MRS` (banked register) and `MSR` (banked register) instructions can access the current mode register ELR_hyp.

From FIQ mode

- From Non-secure EL1, the Monitor mode registers SP_mon, LR_mon, and SPSR_mon.
- The Hyp mode registers SP_hyp, SPSR_hyp, and ELR_hyp.
- The current mode registers R8_fiq-R12_fiq, SP_fiq, LR_fiq, and SPSR_fiq.

From System mode

- From Non-secure EL1, the Monitor mode registers SP_mon, LR_mon, and SPSR_mon.
- The Hyp mode registers SP_hyp, SPSR_hyp, and ELR_hyp.
- The current mode registers R8_usr-R12_usr, SP_usr, and LR_usr.

From Supervisor mode, Abort mode, Undefined mode, and IRQ mode

- From Non-secure EL1, the Monitor mode registers SP_mon, LR_mon, and SPSR_mon.
- The Hyp mode registers SP_hyp, SPSR_hyp, and ELR_hyp.
- The current mode registers R8_usr-R12_usr, SP_<current_mode>, LR_<current_mode>, and SPSR_<current_mode>.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If EL3 is using AArch64, all MRS (banked register) and MSR (banked register) accesses to the Monitor mode registers from Secure EL1 modes are trapped to EL3. See *Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32* on page D1-2391.

For more information, see:

- *Encoding the register argument in the banked register transfer instructions*.
- *Pseudocode support for the banked register transfer instructions* on page F5-4954.
- *MRS (Banked register)* on page F5-4534.
- *MSR (Banked register)* on page F5-4538.

Note

CONSTRAINED UNPREDICTABLE behavior must not give access to registers that are not accessible from the current Privilege level and Security state.

F5.2.3 Encoding the register argument in the banked register transfer instructions

The MRS (banked register) and MSR (banked register) instructions include a 5-bit field, SYSm, and an R bit, that together encode the register argument for the instruction.

When the R bit is set to 0, the argument is a register other than a banked copy of the SPSR, and Table F5-1 shows how the SYSm field defines the required register argument. In this table, CONST. UNPREDICTABLE indicates that behavior is CONSTRAINED UNPREDICTABLE.

Table F5-1 Banked register encodings when R==0

SYSm<2:0>	SYSm<4:3>			
	0b00	0b01	0b10	0b11
0b000	R8_usr	R8_fiq	LR_irq	CONST. UNPREDICTABLE
0b001	R9_usr	R9_fiq	SP_irq	CONST. UNPREDICTABLE
0b010	R10_usr	R10_fiq	LR_svc	CONST. UNPREDICTABLE
0b011	R11_usr	R11_fiq	SP_svc	CONST. UNPREDICTABLE
0b100	R12_usr	R12_fiq	LR_abt	LR_mon
0b101	SP_usr	SP_fiq	SP_abt	SP_mon
0b110	LR_usr	LR_fiq	LR_und	ELR_hyp
0b111	CONST. UNPREDICTABLE	CONST. UNPREDICTABLE	SP_und	SP_hyp

When the R bit is set to 1, the argument is a banked copy of the SPSR, and Table F5-2 shows how the SYSm field defines the required register argument. In this table, CONST. UNPREDICTABLE indicates that behavior is CONSTRAINED UNPREDICTABLE.

Table F5-2 Banked register encodings when R==1

SYSm<2:0>	SYSm<4:3>			
	0b00	0b01	0b10	0b11
0b000	CONST. UNPREDICTABLE	CONST. UNPREDICTABLE	SPSR_irq	CONST. UNPREDICTABLE
0b001	CONST. UNPREDICTABLE	CONST. UNPREDICTABLE	CONST. UNPREDICTABLE	CONST. UNPREDICTABLE
0b010	CONST. UNPREDICTABLE	CONST. UNPREDICTABLE	SPSR_svc	CONST. UNPREDICTABLE

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table F5-2 Banked register encodings when R==1 (continued)

SYSm<2:0>	SYSm<4:3>			
	0b00	0b01	0b10	0b11
0b011	CONST. UNPREDICTABLE	CONST. UNPREDICTABLE	CONST. UNPREDICTABLE	CONST. UNPREDICTABLE
0b100	CONST. UNPREDICTABLE	CONST. UNPREDICTABLE	SPSR_abt	SPSR_mon
0b101	CONST. UNPREDICTABLE	CONST. UNPREDICTABLE	CONST. UNPREDICTABLE	CONST. UNPREDICTABLE
0b110	CONST. UNPREDICTABLE	SPSR_fiq	SPSR_und	SPSR_hyp
0b111	CONST. UNPREDICTABLE	CONST. UNPREDICTABLE	CONST. UNPREDICTABLE	CONST. UNPREDICTABLE

F5.2.4 Pseudocode support for the banked register transfer instructions

The pseudocode functions `BankedRegisterAccessValid()` and `SPSRaccessValid()` check the validity of `MRS` (banked register) and `MSR` (banked register) accesses. That is, they filter the accesses that are CONSTRAINED UNPREDICTABLE either because:

- They attempt to access a register that *Usage restrictions on the banked register transfer instructions* on page F5-4952 shows is not accessible.
- They use an `SYSm<4:0>` encoding that *Encoding the register argument in the banked register transfer instructions* on page F5-4953 shows as CONSTRAINED UNPREDICTABLE.

`BankedRegisterAccessValid()` applies to accesses to the banked general-purpose registers, or to ELR_hyp, and `SPSRaccessValid()` applies to accesses to the SPSRs.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter F6
T32 and A32 Advanced SIMD and Floating-point Instruction Descriptions

This chapter describes each instruction. It contains the following sections:

- *Alphabetical list of Advanced SIMD and floating-point instructions* on page F6-4956.

Note

Some headings in this chapter use the term *floating-point register*. This is an abbreviated description, and means a register in the Advanced SIMD and floating-point register file.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1 Alphabetical list of Advanced SIMD and floating-point instructions

This section lists every Advanced SIMD and floating-point instruction in the T32 and A32 instruction sets. For details of the format used see *Format of instruction descriptions* on page F2-4076.

This section is formatted so that each instruction description starts on a new page.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.1 AESD

AES single round decryption.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19 18	17	16	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	1	0	0	1	1	1	D	1	1	size	0	0	Vd	0	0	1	1	0	1	M	0	Vm

A1 variant

```
AESD.<dt> <Qd>, <Qm>
```

Decode for this encoding

```
if !HaveAESExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 2	1	0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	1	1	1	1	1	1	D	1	1	size	0	0	Vd	0	0	1	1	0	1	M	0	Vm

T1 variant

```
AESD.<dt> <Qd>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveAESExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<dt> Is the data type, encoded in the "size" field. It can have the following values:

8 when `size = 00`

The following encodings are reserved:

- `size = 01`.
- `size = 1x`.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    op1 = Q[d>>1]; op2 = Q[m>>1];
    Q[d>>1] = AESInvSubBytes(AESInvShiftRows(op1 EOR op2));
```

Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.2 AESE

AES single round encryption.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19 18	17	16	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	1	0	0	1	1	1	D	1	1	size	0	0	Vd	0	0	1	1	0	0	M	0	Vm

A1 variant

```
AESE.<dt> <Qd>, <Qm>
```

Decode for this encoding

```
if !HaveAESExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd	0	0	1	1	0	0	M	0	Vm

T1 variant

```
AESE.<dt> <Qd>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveAESExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<dt>` Is the data type, encoded in the "size" field. It can have the following values:

`8` when `size = 00`

The following encodings are reserved:

- `size = 01`.
- `size = 1x`.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    op1 = Q[d>>1]; op2 = Q[m>>1];
    Q[d>>1] = AESSubBytes(AESShiftRows(op1 EOR op2));
```

Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.3 AESIMC

AES inverse mix columns.

A1

31 30 29 28	27 26 25 24	23	22	21 20	19 18	17 16	15 12	11	10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	1 1	size	0 0	Vd	0	0 1 1	1	1	M	0	Vm

A1 variant

```
AESIMC.<dt> <Qd>, <Qm>
```

Decode for this encoding

```
if !HaveAESExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

T1

15 14 13 12	11 10 9 8	7	6	5 4	3 2	1 0	15 12	11	10 9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	1	D	1 1	size	0 0	Vd	0	0 1 1	1	1	M	0	Vm

T1 variant

```
AESIMC.<dt> <Qd>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveAESExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

Assembler symbols

`<dt>` Is the data type, encoded in the "size" field. It can have the following values:

`8` when `size = 00`

The following encodings are reserved:

- `size = 01`.
- `size = 1x`.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    Q[d>>1] = AESInvMixColumns(Q[m>>1]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.4 AESMC

AES mix columns.

A1

31 30 29 28	27 26 25 24	23	22	21 20	19 18	17 16	15 12	11	10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	1 1	size	0 0	Vd	0	0 1 1	1	0	M	0	Vm

A1 variant

```
AESMC.<dt> <Qd>, <Qm>
```

Decode for this encoding

```
if !HaveAESExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

T1

15 14 13 12	11 10 9 8	7	6	5 4	3 2	1 0	15 12	11	10 9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	1	D	1 1	size	0 0	Vd	0	0 1 1	1	0	M	0	Vm

T1 variant

```
AESMC.<dt> <Qd>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveAESExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<dt>` Is the data type, encoded in the "size" field. It can have the following values:

`8` when `size = 00`

The following encodings are reserved:

- `size = 01`.
- `size = 1x`.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    Q[d>>1] = AESMixColumns(Q[m>>1]);
```

Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.5 FLDM*X (FLDMDBX, FLDMIAX)

FLDMDBX is the Decrement Before variant of this instruction, and FLDMIAX is the Increment After variant. FLDM*X loads multiple SIMD&FP registers from consecutive locations in the Advanced SIMD and floating-point register file using an address from a general-purpose register.

Arm deprecates use of FLDMDBX and FLDMIAX, except for disassembly purposes, and reassembly of disassembled code.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7–1	0
!=1111	1	1	0	P	U	D	W	1	Rn	Vd	1	0	1	1	imm8<7:1>	1
cond																imm8<0>

Decrement Before variant

Applies when `P == 1 && U == 0 && W == 1`.

```
FLDMDBX{<c>}{<q>} <Rn>!, <dreglist>
```

Increment After variant

Applies when `P == 0 && U == 1`.

```
FLDMIAX{<c>}{<q>} <Rn>{!}, <dreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE "VLDR";
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = FALSE;  add = (U == '1');  wback = (W == '1');
d = UInt(D:Vd);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
regs = UInt(imm8) DIV 2;  // If UInt(imm8) is odd, see "FLDM*X".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
if imm8<0> == '1' && (d+regs) > 16 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as a `VLDM` with the same addressing mode but loads no registers.

If `regs > 16 || (d+regs) > 16`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7 … 1	0
1	1	1	0	1	1	0	P	U	D	W	1	Rn	Vd	1	0	1	1	imm8<7:1>	1
																			imm8<0>

Decrement Before variant

Applies when `P == 1 && U == 0 && W == 1`.

```
FLDMDBX{<c>}{<q>} <Rn>!, <dreglist>
```

Increment After variant

Applies when `P == 0 && U == 1`.

```
FLDMIAX{<c>}{<q>} <Rn>{!}, <dreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE "VLDR";
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = FALSE;  add = (U == '1');  wback = (W == '1');
d = UInt(D:Vd);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
regs = UInt(imm8) DIV 2;  // If UInt(imm8) is odd, see "FLDM*X".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
if imm8<0> == '1' && (d+regs) > 16 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as a `VLDM` with the same addressing mode but loads no registers.

If `regs > 16 || (d+regs) > 16`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Related encodings: See *Advanced SIMD and floating-point 64-bit move* on page F3-4147 for the T32 instruction set, or *Advanced SIMD and floating-point 64-bit move* on page F4-4220 for the A32 instruction set.

Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field. If writeback is not specified, the PC can be used.

`!` Specifies base register writeback. Encoded in the "W" field as 1 if present, otherwise 0.

`<dreglist>` Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list plus one. The list must contain at least one register, all registers must be in the range D0-D15, and must not contain more than 16 registers.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    address = if add then R[n] else R[n]-imm32;
    for r = 0 to regs-1
        if single_regs then
            S[d+r] = MemA[address,4];  address = address+4;
        else
            word1 = MemA[address,4];  word2 = MemA[address+4,4];  address = address+8;
            // Combine the word-aligned words in the correct order for current endianness.
            D[d+r] = if BigEndian() then word1:word2 else word2:word1;
    if wback then R[n] = if add then R[n]+imm32 else R[n]-imm32;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.6 FSTMDBX, FSTMIAX

FSTMX stores multiple SIMD&FP registers from the Advanced SIMD and floating-point register file to consecutive locations in using an address from a general-purpose register.

Arm deprecates use of FLDMDBX and FLDMIAX, except for disassembly purposes, and reassembly of disassembled code.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7–1	0
!=1111	1	1	0	P	U	D	W	0	Rn	Vd	1	0	1	1	imm8<7:1>	1
cond																imm8<0>

Decrement Before variant

Applies when `P == 1 && U == 0 && W == 1`.

```
FSTMDBX{<c>}{<q>} <Rn>!, <dreglist>
```

Increment After variant

Applies when `P == 0 && U == 1`.

```
FSTMIAX{<c>}{<q>} <Rn>{!}, <dreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE "VSTR";
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = FALSE;  add = (U == '1');  wback = (W == '1');
d = UInt(D:Vd);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
regs = UInt(imm8) DIV 2;  // If UInt(imm8) is odd, see "FSTMX".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
if imm8<0> == '1' && (d+regs) > 16 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as a `VSTM` with the same addressing mode but stores no registers.

If `regs > 16 || (d+regs) > 16`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10	9 8	7 1	0
1 1 1 0	1 1 0 P	U	D	W	0	Rn	Vd	1 0	1 1	imm8<7:1>	1
											imm8<0>

Decrement Before variant

Applies when `P == 1 && U == 0 && W == 1`.

```
FSTMDBX{<c>}{<q>} <Rn>!, <dreglist>
```

Increment After variant

Applies when `P == 0 && U == 1`.

```
FSTMIAX{<c>}{<q>} <Rn>{!}, <dreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE "VSTR";
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = FALSE;  add = (U == '1');  wback = (W == '1');
d = UInt(D:Vd);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
regs = UInt(imm8) DIV 2;  // If UInt(imm8) is odd, see "FSTMX".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
if imm8<0> == '1' && (d+regs) > 16 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as a `VSTM` with the same addressing mode but stores no registers.

If `regs > 16 || (d+regs) > 16`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Related encodings: See *Advanced SIMD and floating-point 64-bit move* on page F3-4147 for the T32 instruction set, or *Advanced SIMD and floating-point 64-bit move* on page F4-4220 for the A32 instruction set.

Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field. If writeback is not specified, the PC can be used. However, Arm deprecates use of the PC.

`!` Specifies base register writeback. Encoded in the "W" field as 1 if present, otherwise 0.

`<dreglist>` Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list plus one. The list must contain at least one register, all registers must be in the range D0-D15, and must not contain more than 16 registers.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    address = if add then R[n] else R[n]-imm32;
    for r = 0 to regs-1
        if single_regs then
            MemA[address,4] = S[d+r];  address = address+4;
        else
            // Store as two word-aligned words in the correct order for current endianness.
            MemA[address,4] = if BigEndian() then D[d+r]<63:32> else D[d+r]<31:0>;
            MemA[address+4,4] = if BigEndian() then D[d+r]<31:0> else D[d+r]<63:32>;
            address = address+8;
    if wback then R[n] = if add then R[n]+imm32 else R[n]-imm32;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.7 SHA1C

SHA1 hash update (choose).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	0	0	D	0	0	Vn	Vd	1	1	0	0	N	Q	M	0	Vm

A1 variant

```
SHA1C.32 <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	0	1	1	1	1	0	D	0	0	Vn	Vd	1	1	0	0	N	Q	M	0	Vm

T1 variant

```
SHA1C.32 <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    X = Q[d>>1];
    Y = Q[n>>1]<31:0>; // Note: 32 bits wide
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
W = Q[m>>1];
for e = 0 to 3
    t = SHAchoose(X<63:32>, X<95:64>, X<127:96>);
    Y = Y + ROL(X<31:0>, 5) + t + Elem[W, e, 32];
    X<63:32> = ROL(X<63:32>, 30);
    <Y, X> = ROL(Y:X, 32);
Q[d>>1] = X;
```

Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.8 SHA1H

SHA1 fixed rotate.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19 18	17	16	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	1	0	0	1	1	1	D	1	1	size	0	1	Vd	0	0	1	0	1	1	M	0	Vm

A1 variant

```
SHA1H.32 <Qd>, <Qm>
```

Decode for this encoding

```
if !HaveSHA1Ext() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 2	1	0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	1	1	1	1	1	1	D	1	1	size	0	1	Vd	0	0	1	0	1	1	M	0	Vm

T1 variant

```
SHA1H.32 <Qd>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveSHA1Ext() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    Q[d>>1] = ZeroExtend(ROL(Q[m>>1]<31:0>, 30), 128);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.9 SHA1M

SHA1 hash update (majority).

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 0	0	D	1	0	Vn	Vd	1 1 0 0	N	Q	M	0	Vm

A1 variant

```
SHA1M.32 <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 0	1 1 1 1	0	D	1	0	Vn	Vd	1 1 0 0	N	Q	M	0	Vm

T1 variant

```
SHA1M.32 <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    X = Q[d>>1];
    Y = Q[n>>1]<31:0>; // Note: 32 bits wide
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
W = Q[m>>1];
for e = 0 to 3
    t = SHAmajority(X<63:32>, X<95:64>, X<127:96>);
    Y = Y + ROL(X<31:0>, 5) + t + Elem[W, e, 32];
    X<63:32> = ROL(X<63:32>, 30);
    <Y, X> = ROL(Y:X, 32);
Q[d>>1] = X;
```

Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.10 SHA1P

SHA1 hash update (parity).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19 – 16	15 – 12	11	10	9	8	7	6	5	4	3 – 0
1	1	1	1	0	0	1	0	0	D	0	1	Vn	Vd	1	1	0	0	N	Q	M	0	Vm

A1 variant

```
SHA1P.32 <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 – 0	15 – 12	11	10	9	8	7	6	5	4	3 – 0
1	1	1	0	1	1	1	1	0	D	0	1	Vn	Vd	1	1	0	0	N	Q	M	0	Vm

T1 variant

```
SHA1P.32 <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<Qd>`	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
`<Qn>`	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
`<Qm>`	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    X = Q[d>>1];
    Y = Q[n>>1]<31:0>; // Note: 32 bits wide
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
W = Q[m>>1];
for e = 0 to 3
    t = SHAparity(X<63:32>, X<95:64>, X<127:96>);
    Y = Y + ROL(X<31:0>, 5) + t + Elem[W, e, 32];
    X<63:32> = ROL(X<63:32>, 30);
    <Y, X> = ROL(Y:X, 32);
Q[d>>1] = X;
```

Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.11 SHA1SU0

SHA1 schedule update 0.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	0	0	D	1	1	Vn	Vd	1	1	0	0	N	Q	M	0	Vm

A1 variant

```
SHA1SU0.32 <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	0	1	1	1	1	0	D	1	1	Vn	Vd	1	1	0	0	N	Q	M	0	Vm

T1 variant

```
SHA1SU0.32 <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<Qd>`	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
`<Qn>`	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
`<Qm>`	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    op1 = Q[d>>1]; op2 = Q[n>>1]; op3 = Q[m>>1];
    op2 = op2<63:0> : op1<127:64>;
    Q[d>>1] = op1 EOR op2 EOR op3;
```

Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.12 SHA1SU1

SHA1 schedule update 1.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19 18	17	16	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	1	0	0	1	1	1	D	1	1	size	1	0	Vd	0	0	1	1	1	0	M	0	Vm

A1 variant

```
SHA1SU1.32 <Qd>, <Qm>
```

Decode for this encoding

```
if !HaveSHA1Ext() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 2	1	0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	1	1	1	1	1	1	D	1	1	size	1	0	Vd	0	0	1	1	1	0	M	0	Vm

T1 variant

```
SHA1SU1.32 <Qd>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveSHA1Ext() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qm>	Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    X = Q[d>>1]; Y = Q[m>>1];
    T = X EOR LSR(Y, 32);
    W0 = ROL(T<31:0>, 1);
    W1 = ROL(T<63:32>, 1);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
W2 = ROL(T<95:64>, 1);
W3 = ROL(T<127:96>, 1) EOR ROL(T<31:0>, 2);
Q[d>>1] = W3:W2:W1:W0;
```

Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.13 SHA256H

SHA256 hash update part 1.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	1	0	D	0	0	Vn	Vd	1	1	0	0	N	Q	M	0	Vm

A1 variant

```
SHA256H.32 <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if !HaveSHA256Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	1	1	0	D	0	0	Vn	Vd	1	1	0	0	N	Q	M	0	Vm

T1 variant

```
SHA256H.32 <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveSHA256Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<Qd>`	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
`<Qn>`	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
`<Qm>`	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    X = Q[d>>1]; Y = Q[n>>1]; W = Q[m>>1]; part1 = TRUE;
    Q[d>>1] = SHA256hash(X, Y, W, part1);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.14 SHA256H2

SHA256 hash update part 2.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	1	0	D	0	1	Vn	Vd	1	1	0	0	N	Q	M	0	Vm

A1 variant

```
SHA256H2.32 <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if !HaveSHA256Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	1	1	0	D	0	1	Vn	Vd	1	1	0	0	N	Q	M	0	Vm

T1 variant

```
SHA256H2.32 <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveSHA256Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<Qd>`	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
`<Qn>`	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
`<Qm>`	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    X = Q[n>>1]; Y = Q[d>>1]; W = Q[m>>1]; part1 = FALSE;
    Q[d>>1] = SHA256hash(X, Y, W, part1);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.15 SHA256SU0

SHA256 schedule update 0.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19 18	17	16	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	1	0	0	1	1	1	D	1	1	size	1	0	Vd	0	0	1	1	1	1	M	0	Vm

A1 variant

```
SHA256SU0.32 <Qd>, <Qm>
```

Decode for this encoding

```
if !HaveSHA256Ext() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 2	1	0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	1	1	1	1	1	1	D	1	1	size	1	0	Vd	0	0	1	1	1	1	M	0	Vm

T1 variant

```
SHA256SU0.32 <Qd>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveSHA256Ext() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation for all encodings

```
if ConditionPassed() then
    bits(128) result;
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    X = Q[d>>1]; Y = Q[m>>1];
    T = Y<31:0> : X<127:32>;
    for e = 0 to 3
        elt = Elem[T, e, 32];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        elt = ROR(elt, 7) EOR ROR(elt, 18) EOR LSR(elt, 3);
        Elem[result, e, 32] = elt + Elem[X, e, 32];
    Q[d>>1] = result;
```

Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.16 SHA256SU1

SHA256 schedule update 1.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	1	0	D	1	0	Vn	Vd	1	1	0	0	N	Q	M	0	Vm

A1 variant

```
SHA256SU1.32 <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if !HaveSHA256Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	1	1	0	D	1	0	Vn	Vd	1	1	0	0	N	Q	M	0	Vm

T1 variant

```
SHA256SU1.32 <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveSHA256Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<Qd>`	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
`<Qn>`	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
`<Qm>`	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation for all encodings

```
if ConditionPassed() then
    bits(128) result;
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    X = Q[d>>1]; Y = Q[n>>1]; Z = Q[m>>1];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
T0 = Z<31:0> : Y<127:32>;

T1 = Z<127:64>;
for e = 0 to 1
    elt = Elem[T1, e, 32];
    elt = ROR(elt, 17) EOR ROR(elt, 19) EOR LSR(elt, 10);
    elt = elt + Elem[X, e, 32] + Elem[T0, e, 32];
    Elem[result, e, 32] = elt;

T1 = result<63:0>;
for e = 2 to 3
    elt = Elem[T1, e - 2, 32];
    elt = ROR(elt, 17) EOR ROR(elt, 19) EOR LSR(elt, 10);
    elt = elt + Elem[X, e, 32] + Elem[T0, e, 32];
    Elem[result, e, 32] = elt;

Q[d>>1] = result;
```

Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.17 VABA

Vector Absolute Difference and Accumulate subtracts the elements of one vector from the corresponding elements of another vector, and accumulates the absolute values of the results into the elements of the destination vector.

Operand and result elements are all integers of the same length.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 U	0	D	size		Vn	Vd	0 1 1 1	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VABA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VABA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');  long_destination = FALSE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 U	1 1 1 1	0	D	size		Vn	Vd	0 1 1 1	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VABA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VABA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');  long_destination = FALSE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

<dt> Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

S8	when U = 0, size = 00
S16	when U = 0, size = 01
S32	when U = 0, size = 10
U8	when U = 1, size = 00
U16	when U = 1, size = 01
U32	when U = 1, size = 10

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];
            op2 = Elem[Din[m+r],e,esize];
            absdiff = Abs(Int(op1,unsigned) - Int(op2,unsigned));
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + absdiff;
            else
                Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + absdiff;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.18 VABAL

Vector Absolute Difference and Accumulate Long subtracts the elements of one vector from the corresponding elements of another vector, and accumulates the absolute values of the results into the elements of the destination vector.

Operand elements are all integers of the same length, and the result elements are double the length of the operands.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21 20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 U	1	D	!=11	Vn	Vd	0 1 0 1	N	0	M	0	Vm
				size								

A1 variant

```
VABAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  long_destination = TRUE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;
```

T1

15 14 13 12	11 10 9 8	7	6	5 4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 U	1 1 1 1	1	D	!=11	Vn	Vd	0 1 0 1	N	0	M	0	Vm
				size								

T1 variant

```
VABAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  long_destination = TRUE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;
```

Notes for all encodings

Related encodings: See *Advanced SIMD data-processing* on page F3-4137 for the T32 instruction set, or *Advanced SIMD data-processing* on page F4-4225 for the A32 instruction set.

Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

`S8`	when `U = 0`, `size = 00`
`S16`	when `U = 0`, `size = 01`
`S32`	when `U = 0`, `size = 10`
`U8`	when `U = 1`, `size = 00`
`U16`	when `U = 1`, `size = 01`
`U32`	when `U = 1`, `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];
            op2 = Elem[Din[m+r],e,esize];
            absdiff = Abs(Int(op1,unsigned) - Int(op2,unsigned));
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + absdiff;
            else
                Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + absdiff;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.19 VABD (floating-point)

Vector Absolute Difference (floating-point) subtracts the elements of one vector from the corresponding elements of another vector, and places the absolute values of the results in the elements of the destination vector.

Operand and result elements are floating-point numbers of the same size.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	1	0	D	1	sz	Vn	Vd	1	1	0	1	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VABD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VABD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	1	1	0	D	1	sz	Vn	Vd	1	1	0	1	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VABD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VABD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
case sz of
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

`F32` when `sz = 0`

`F16` when `sz = 1`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op2 = Elem[D[m+r],e,esize];
            Elem[D[d+r],e,esize] = FPAbs(FPSub(op1,op2,StandardFPSCRValue()));
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.20 VABD (integer)

Vector Absolute Difference (integer) subtracts the elements of one vector from the corresponding elements of another vector, and places the absolute values of the results in the elements of the destination vector.

Operand and result elements are all integers of the same length.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 U	0	D	size		Vn	Vd	0 1 1 1	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VABD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VABD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');  long_destination = FALSE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 U	1 1 1 1	0	D	size		Vn	Vd	0 1 1 1	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VABD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VABD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');  long_destination = FALSE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

<dt> Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

S8	when U = 0, size = 00
S16	when U = 0, size = 01
S32	when U = 0, size = 10
U8	when U = 1, size = 00
U16	when U = 1, size = 01
U32	when U = 1, size = 10

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];
            op2 = Elem[Din[m+r],e,esize];
            absdiff = Abs(Int(op1,unsigned) - Int(op2,unsigned));
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = absdiff<2*esize-1:0>;
            else
                Elem[D[d+r],e,esize] = absdiff<esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.21 VABDL (integer)

Vector Absolute Difference Long (integer) subtracts the elements of one vector from the corresponding elements of another vector, and places the absolute values of the results in the elements of the destination vector.

Operand elements are all integers of the same length, and the result elements are double the length of the operands.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21 20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 U	1	D	!=11	Vn	Vd	0 1 1 1	N	0	M	0	Vm
				size								

A1 variant

```
VABDL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  long_destination = TRUE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;
```

T1

15 14 13	12	11 10 9 8	7	6	5 4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1	U	1 1 1 1	1	D	!=11	Vn	Vd	0 1 1 1	N	0	M	0	Vm
					size								

T1 variant

```
VABDL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  long_destination = TRUE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;
```

Notes for all encodings

Related encodings: See *Advanced SIMD data-processing* on page F3-4137 for the T32 instruction set, or *Advanced SIMD data-processing* on page F4-4225 for the A32 instruction set.

Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<dt>` Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

`S8`	when `U = 0, size = 00`
`S16`	when `U = 0, size = 01`
`S32`	when `U = 0, size = 10`
`U8`	when `U = 1, size = 00`
`U16`	when `U = 1, size = 01`
`U32`	when `U = 1, size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];
            op2 = Elem[Din[m+r],e,esize];
            absdiff = Abs(Int(op1,unsigned) - Int(op2,unsigned));
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = absdiff<2*esize-1:0>;
            else
                Elem[D[d+r],e,esize] = absdiff<esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.22 VABS

Vector Absolute takes the absolute value of each element in a vector, and places the results in a second vector. The floating-point version only clears the sign bit.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 18	17 16	15 12	11	10	9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	1	1	size	0 1	Vd	0	F	1 1	0	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VABS{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VABS{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
advsimd = TRUE;  floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

A2

31 28	27 26 25 24	23	22	21	20	19	18 17 16	15 12	11 10	9 8	7	6	5	4	3 0
!=1111	1 1 1 0	1	D	1	1	0	0 0 0	Vd	1 0	size	1	1	M	0	Vm
cond															

Half-precision scalar variant

Applies when `size == 01`.

```
VABS{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VABS{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VABS{<c>}{<q>}.F64 <Dd>, <Dm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
advsimd = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15 – 12	11	10	9	8	7	6	5	4	3 – 0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	1	Vd	0	F	1	1	0	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VABS{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VABS{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
advsimd = TRUE;  floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	0	1	1	1	0	1	D	1	1	0	0	0	0	Vd	1	0	size		1	1	M	0	Vm

Half-precision scalar variant

Applies when `size == 01`.

```
VABS{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VABS{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VABS{<c>}{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
advsimd = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding A2, T1 and T2: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "F:size" field. It can have the following values:

`S8`	when `F = 0`, `size = 00`
`S16`	when `F = 0`, `size = 01`
`S32`	when `F = 0`, `size = 10`
`F16`	when `F = 1`, `size = 01`
`F32`	when `F = 1`, `size = 10`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

`<Sm>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                if floating_point then
                    Elem[D[d+r],e,esize] = FPAbs(Elem[D[m+r],e,esize]);
                else
                    result = Abs(SInt(Elem[D[m+r],e,esize]));
                    Elem[D[d+r],e,esize] = result<esize-1:0>;
    else             // VFP instruction
        case esize of
            when 16 S[d] = Zeros(16) : FPAbs(S[m]<15:0>);
            when 32 S[d] = FPAbs(S[m]);
            when 64 D[d] = FPAbs(D[m]);
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.23 VACGE

Vector Absolute Compare Greater Than or Equal takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the first is greater than or equal to the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operands and result can be quadword or doubleword vectors. They must all be the same size.

The operand vector elements are floating-point numbers. The result vector elements are the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

This instruction is used by the pseudo-instruction VACLE. The pseudo-instruction is never the preferred disassembly.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	0	D	0	sz	Vn	Vd	1 1 1 0	N	Q	M	1	Vm
				op									

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VACGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VACGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
or_equal = (op == '0');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 1 0	D	0	sz		Vn	Vd	1 1 1 0	N	Q	M	1	Vm
			op										

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VACGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VACGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
or_equal = (op == '0');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

`F32`	when `sz = 0`
`F16`	when `sz = 1`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = FPAbs(Elem[D[n+r],e,esize]);  op2 = FPAbs(Elem[D[m+r],e,esize]);
            if or_equal then
                test_passed = FPCompareGE(op1, op2, StandardFPSCRValue());
            else
                test_passed = FPCompareGT(op1, op2, StandardFPSCRValue());
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.24 VACLE

Vector Absolute Compare Less Than or Equal takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the first is less than or equal to the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros

This instruction is a pseudo-instruction of the VACGE instruction. This means that:

- The encodings in this description are named to match the encodings of VACGE.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VACGE gives the operational pseudocode for this instruction.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	0	D	0	sz	Vn	Vd	1 1 1 0	N	Q	M	1	Vm
				op									

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VACLE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VACGE{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>
```

and is never the preferred disassembly.

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VACLE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VACGE{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>
```

and is never the preferred disassembly.

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	0	D	0	sz	Vn	Vd	1 1 1 0	N	Q	M	1	Vm
				op									

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VACLE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VACGE{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>
```

and is never the preferred disassembly.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VACLE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VACGE{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>
```

and is never the preferred disassembly.

Assembler symbols

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

`F32`	when `sz = 0`
`F16`	when `sz = 1`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Operation for all encodings

The description of VACGE gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.25 VACGT

Vector Absolute Compare Greater Than takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the first is greater than the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operands and result can be quadword or doubleword vectors. They must all be the same size.

The operand vector elements are floating-point numbers. The result vector elements are the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

This instruction is used by the pseudo-instruction VACLT. The pseudo-instruction is never the preferred disassembly.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	0	D	1	sz	Vn	Vd	1 1 1 0	N	Q	M	1	Vm
				op									

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VACGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VACGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
or_equal = (op == '0');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	0	D	1	sz	Vn	Vd	1 1 1 0	N	Q	M	1	Vm
				op									

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VACGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VACGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
or_equal = (op == '0');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

`F32`	when `sz = 0`
`F16`	when `sz = 1`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = FPAbs(Elem[D[n+r],e,esize]);  op2 = FPAbs(Elem[D[m+r],e,esize]);
            if or_equal then
                test_passed = FPCompareGE(op1, op2, StandardFPSCRValue());
            else
                test_passed = FPCompareGT(op1, op2, StandardFPSCRValue());
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.26 VACLT

Vector Absolute Compare Less Than takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the first is less than the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros

This instruction is a pseudo-instruction of the VACGT instruction. This means that:

- The encodings in this description are named to match the encodings of VACGT.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VACGT gives the operational pseudocode for this instruction.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	0	D	1	sz	Vn	Vd	1 1 1 0	N	Q	M	1	Vm
				op									

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VACLT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VACGT{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>
```

and is never the preferred disassembly.

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VACLT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VACGT{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>
```

and is never the preferred disassembly.

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	0	D	1	sz	Vn	Vd	1 1 1 0	N	Q	M	1	Vm
				op									

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VACLT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VACGT{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>
```

and is never the preferred disassembly.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VACLT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VACGT{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>
```

and is never the preferred disassembly.

Assembler symbols

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

`F32`	when `sz = 0`
`F16`	when `sz = 1`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Operation for all encodings

The description of VACGT gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.27 VADD (floating-point)

Vector Add (floating-point) adds corresponding elements in two vectors, and places the results in the destination vector.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	0	0	D	0	sz	Vn	Vd	1	1	0	1	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
advsimd = TRUE;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

A2

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	1	1	1	0	0	D	1	1	Vn	Vd	1	0	size		N	0	M	0	Vm
cond																			

Half-precision scalar variant

Applies when `size == 01`.

```
VADD{<c>}{<q>}.F16 {<Sd>,} <Sn>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VADD{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VADD{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
advsimd = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	0	1	1	1	1	0	D	0	sz	Vn	Vd	1	1	0	1	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
advsimd = TRUE;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T2

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	0	1	1	1	0	0	D	1	1	Vn	Vd	1	0	size		N	0	M	0	Vm

Half-precision scalar variant

Applies when `size == 01`.

```
VADD{<c>}{<q>}.F16 {<Sd>,} <Sn>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VADD{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VADD{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
advsimd = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding A2, T1 and T2: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

`F32`	when `sz = 0`
`F16`	when `sz = 1`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

`<Sn>` Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

`<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                Elem[D[d+r],e,esize] = FPAdd(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize],
                                             StandardFPSCRValue());
    else             // VFP instruction
        case esize of
            when 16
                S[d] = Zeros(16) : FPAdd(S[n]<15:0>, S[m]<15:0>, FPSCR);
            when 32
                S[d] = FPAdd(S[n], S[m], FPSCR);
            when 64
                D[d] = FPAdd(D[n], D[m], FPSCR);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.28 VADD (integer)

Vector Add (integer) adds corresponding elements in two vectors, and places the results in the destination vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 0	0	D	size		Vn	Vd	1 0 0 0	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 0	1 1 1 1	0	D	size		Vn	Vd	1 0 0 0	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:

`I8`	when `size = 00`
`I16`	when `size = 01`
`I32`	when `size = 10`
`I64`	when `size = 11`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = Elem[D[n+r],e,esize] + Elem[D[m+r],e,esize];
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.29 VADDHN

Vector Add and Narrow, returning High Half adds corresponding elements in two quadword vectors, and places the most significant half of each result in a doubleword vector. The results are truncated. For rounded results, see VRADDHN.

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21 20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 0	1	D	!=11 (size)	Vn	Vd	0 1 0 0	N	0	M	0	Vm

A1 variant

```
VADDHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1	0	1 1 1 1 1	D	!=11 (size)	Vn		Vd	0 1 0 0	N	0	M	0	Vm

T1 variant

```
VADDHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

Notes for all encodings

Related encodings: See *Advanced SIMD data-processing* on page F3-4137 for the T32 instruction set, or *Advanced SIMD data-processing* on page F4-4225 for the A32 instruction set.

Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<dt>` Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:

`I16` when `size = 00`

`I32` when `size = 01`

`I64` when `size = 10`

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        result = Elem[Qin[n>>1],e,2*esize] + Elem[Qin[m>>1],e,2*esize];
        Elem[D[d],e,esize] = result<2*esize-1:esize>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.30 VADDL

Vector Add Long adds corresponding elements in two doubleword vectors, and places the results in a quadword vector. Before adding, it sign-extends or zero-extends the elements of both operands.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	U	1	D	!=11		Vn	Vd	0	0	0	0	N	0	M	0	Vm
										size							op					

A1 variant

```
VADDL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;  is_vaddw = (op == '1');
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	U	1	1	1	1	1	D	!=11		Vn	Vd	0	0	0	0	N	0	M	0	Vm
										size							op					

T1 variant

```
VADDL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;  is_vaddw = (op == '1');
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

Notes for all encodings

Related encodings: See *Advanced SIMD data-processing* on page F3-4137 for the T32 instruction set, or *Advanced SIMD data-processing* on page F4-4225 for the A32 instruction set.

Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<dt>` Is the data type for the elements of the second operand vector, encoded in the "U:size" field. It can have the following values:

`S8`	when `U = 0`, `size = 00`
`S16`	when `U = 0`, `size = 01`
`S32`	when `U = 0`, `size = 10`
`U8`	when `U = 1`, `size = 00`
`U16`	when `U = 1`, `size = 01`
`U32`	when `U = 1`, `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        if is_vaddw then
            op1 = Int(Elem[Qin[n>>1],e,2*esize], unsigned);
        else
            op1 = Int(Elem[Din[n],e,esize], unsigned);
        result = op1 + Int(Elem[Din[m],e,esize],unsigned);
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.31 VADDW

Vector Add Wide adds corresponding elements in one quadword and one doubleword vector, and places the results in a quadword vector. Before adding, it sign-extends or zero-extends the elements of the doubleword operand.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21 20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 U	1	D	!=11	Vn	Vd	0 0 0 1	N	0	M	0	Vm
				size			op					

A1 variant

```
VADDW{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;  is_vaddw = (op == '1');
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

T1

15 14 13 12	11 10 9 8	7	6	5 4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 U	1 1 1 1	1	D	!=11	Vn	Vd	0 0 0 1	N	0	M	0	Vm
				size			op					

T1 variant

```
VADDW{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;  is_vaddw = (op == '1');
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

Notes for all encodings

Related encodings: See *Advanced SIMD data-processing* on page F3-4137 for the T32 instruction set, or *Advanced SIMD data-processing* on page F4-4225 for the A32 instruction set.

Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<dt>` Is the data type for the elements of the second operand vector, encoded in the "U:size" field. It can have the following values:

`S8`	when `U = 0`, `size = 00`
`S16`	when `U = 0`, `size = 01`
`S32`	when `U = 0`, `size = 10`
`U8`	when `U = 1`, `size = 00`
`U16`	when `U = 1`, `size = 01`
`U32`	when `U = 1`, `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        if is_vaddw then
            op1 = Int(Elem[Qin[n>>1],e,2*esize], unsigned);
        else
            op1 = Int(Elem[Din[n],e,esize], unsigned);
        result = op1 + Int(Elem[Din[m],e,esize],unsigned);
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.32 VAND (immediate)

Vector Bitwise AND (immediate) performs a bitwise AND between a register value and an immediate value, and returns the result into the destination vector

This instruction is a pseudo-instruction of the VBIC (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of VBIC (immediate).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VBIC (immediate) gives the operational pseudocode for this instruction.

A1

31 30 29 28	27 26 25 24	23	22	21 20	19 18 16	15 12	11 8	7	6	5	4	3 0
1 1 1 1	0 0 0 1	i	1	D	0 0 0	imm3	Vd	0 x x 1	0	Q	1	1	imm4
							cmode					

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VAND{<c>}{<q>}.I16 {<Dd>,} <Dd>, #<imm>
```

is equivalent to

```
VBIC{<c>}{<q>}.I16 <Dd>, #~<imm>
```

and is never the preferred disassembly.

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VAND{<c>}{<q>}.I16 {<Qd>,} <Qd>, #<imm>
```

is equivalent to

```
VBIC{<c>}{<q>}.I16 <Qd>, #~<imm>
```

and is never the preferred disassembly.

A2

31 30 29 28	27 26 25 24	23	22	21 20	19 18 16	15 12	11 8	7	6	5	4	3 0
1 1 1 1	0 0 0 1	i	1	D	0 0 0	imm3	Vd	1 0 x 1	0	Q	1	1	imm4
							cmode					

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VAND{<c>}{<q>}.I32 {<Dd>,} <Dd>, #<imm>
```

is equivalent to

```
VBIC{<c>}{<q>}.I32 <Dd>, #~<imm>
```

and is never the preferred disassembly.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VAND{<c>}{<q>}.I32 {<Qd>,} <Qd>, #<imm>
```

is equivalent to

```
VBIC{<c>}{<q>}.I32 <Qd>, #~<imm>
```

and is never the preferred disassembly.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	0	15	12	11	8	7	6	5	4	3	0
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3		Vd		0 x x 1 (cmode)		0	Q	1	1	imm4	

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VAND{<c>}{<q>}.I16 {<Dd>,} <Dd>, #<imm>
```

is equivalent to

```
VBIC{<c>}{<q>}.I16 <Dd>, #~<imm>
```

and is never the preferred disassembly.

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VAND{<c>}{<q>}.I16 {<Qd>,} <Qd>, #<imm>
```

is equivalent to

```
VBIC{<c>}{<q>}.I16 <Qd>, #~<imm>
```

and is never the preferred disassembly.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	0	15	12	11	8	7	6	5	4	3	0
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3		Vd		1 0 x 1 (cmode)		0	Q	1	1	imm4	

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VAND{<c>}{<q>}.I32 {<Dd>,} <Dd>, #<imm>
```

is equivalent to

```
VBIC{<c>}{<q>}.I32 <Dd>, #~<imm>
```

and is never the preferred disassembly.

128-bit SIMD vector variant

Applies when `Q == 1`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VAND{<c>}{<q>}.I32 {<Qd>,} <Qd>, #<imm>
```

is equivalent to

```
VBIC{<c>}{<q>}.I32 <Qd>, #~<imm>
```

and is never the preferred disassembly.

Assembler symbols

`<c>` For encoding A1 and A2: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1 and T2: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<imm>` Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of `<imm>`, see *Modified immediate constants in T32 and A32 Advanced SIMD instructions* on page F2-4097.

Operation for all encodings

The description of VBIC (immediate) gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.33 VAND (register)

Vector Bitwise AND (register) performs a bitwise AND operation between two registers, and places the result in the destination register.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	0	0	D	0	0	Vn	Vd	0	0	0	1	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VAND{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VAND{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	0	1	1	1	1	0	D	0	0	Vn	Vd	0	0	0	1	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VAND{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VAND{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` An optional data type. It is ignored by assemblers, and does not affect the encoding.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[n+r] AND D[m+r];
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.34 VBIC (immediate)

Vector Bitwise Bit Clear (immediate) performs a bitwise AND between a register value and the complement of an immediate value, and returns the result into the destination vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

This instruction is used by the pseudo-instruction VAND (immediate). The pseudo-instruction is never the preferred disassembly.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	16	15	12	11	8	7	6	5	4	3	0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3		Vd		0 x x 1 (cmode)		0	Q	1	1	imm4	

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VBIC{<c>}{<q>}.I32 {<Dd>,} <Dd>, #<imm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VBIC{<c>}{<q>}.I32 {<Qd>,} <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if cmode<0> == '0' || cmode<3:2> == '11' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	16	15	12	11	8	7	6	5	4	3	0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3		Vd		1 0 x 1 (cmode)		0	Q	1	1	imm4	

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VBIC{<c>}{<q>}.I16 {<Dd>,} <Dd>, #<imm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VBIC{<c>}{<q>}.I16 {<Qd>,} <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if cmode<0> == '0' || cmode<3:2> == '11' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15 14 13	12	11 10 9 8	7	6	5 4	3 2 0	15 12	11 8	7	6	5	4	3 0
1 1 1	i	1 1 1 1	1	D	0 0	0 imm3	Vd	0 x x 1 (cmode)	0	Q	1	1	imm4

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VBIC{<c>}{<q>}.I32 {<Dd>,} <Dd>, #<imm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VBIC{<c>}{<q>}.I32 {<Qd>,} <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if cmode<0> == '0' || cmode<3:2> == '11' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

T2

15 14 13	12	11 10 9 8	7	6	5 4	3 2 0	15 12	11 8	7	6	5	4	3 0
1 1 1	i	1 1 1 1	1	D	0 0	0 imm3	Vd	1 0 x 1 (cmode)	0	Q	1	1	imm4

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VBIC{<c>}{<q>}.I16 {<Dd>,} <Dd>, #<imm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VBIC{<c>}{<q>}.I16 {<Qd>,} <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if cmode<0> == '0' || cmode<3:2> == '11' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

Notes for all encodings

Related encodings: See *Advanced SIMD one register and modified immediate* on page F3-4146 for the T32 instruction set, or *Advanced SIMD one register and modified immediate* on page F4-4233 for the A32 instruction set.

Assembler symbols

<c> For encoding A1 and A2: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For encoding T1 and T2: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<imm>` Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of `<imm>`, see *Modified immediate constants in T32 and A32 Advanced SIMD instructions* on page F2-4097.

The I8, I64, and F32 data types are permitted as pseudo-instructions, if the immediate can be represented by this instruction, and are encoded using a permitted encoding of the I16 or I32 data type.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[d+r] AND NOT(imm64);
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.35 VBIC (register)

Vector Bitwise Bit Clear (register) performs a bitwise AND between a register value and the complement of a register value, and places the result in the destination register.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 0	0	D	0	1	Vn	Vd	0 0 0 1	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VBIC{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VBIC{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 0	1 1 1 1	0	D	0	1	Vn	Vd	0 0 0 1	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VBIC{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VBIC{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` An optional data type. It is ignored by assemblers, and does not affect the encoding.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[n+r] AND NOT(D[m+r]);
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.36 VBIF

Vector Bitwise Insert if False inserts each bit from the first source register into the destination register if the corresponding bit of the second source register is 0, otherwise leaves the bit in the destination register unchanged.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21 20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 0 1	1	0	D	1 1	Vn	Vd	0 0 0 1	N	Q	M	1	Vm
				op								

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VBIF{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VBIF{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if op == '00' then SEE "VEOR";
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5 4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	0	D	1 1	Vn	Vd	0 0 0 1	N	Q	M	1	Vm
				op								

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VBIF{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VBIF{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if op == '00' then SEE "VEOR";
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` An optional data type. It is ignored by assemblers, and does not affect the encoding.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
enumeration VBitOps {VBitOps_VBIF, VBitOps_VBIT, VBitOps_VBSL};

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        case operation of
            when VBitOps_VBIF  D[d+r] = (D[d+r] AND D[m+r]) OR (D[n+r] AND NOT(D[m+r]));
            when VBitOps_VBIT  D[d+r] = (D[n+r] AND D[m+r]) OR (D[d+r] AND NOT(D[m+r]));
            when VBitOps_VBSL  D[d+r] = (D[n+r] AND D[d+r]) OR (D[m+r] AND NOT(D[d+r]));
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.37 VBIT

Vector Bitwise Insert if True inserts each bit from the first source register into the destination register if the corresponding bit of the second source register is 1, otherwise leaves the bit in the destination register unchanged.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21 20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 0 1	1	0	D	1 0	Vn	Vd	0 0 0 1	N	Q	M	1	Vm
				op								

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VBIT{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VBIT{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if op == '00' then SEE "VEOR";
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5 4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 1 0	D	1	0	Vn	Vd	0 0 0 1	N	Q	M	1	Vm
				op								

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VBIT{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VBIT{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if op == '00' then SEE "VEOR";
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` An optional data type. It is ignored by assemblers, and does not affect the encoding.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
enumeration VBitOps {VBitOps_VBIF, VBitOps_VBIT, VBitOps_VBSL};

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        case operation of
            when VBitOps_VBIF  D[d+r] = (D[d+r] AND D[m+r]) OR (D[n+r] AND NOT(D[m+r]));
            when VBitOps_VBIT  D[d+r] = (D[n+r] AND D[m+r]) OR (D[d+r] AND NOT(D[m+r]));
            when VBitOps_VBSL  D[d+r] = (D[n+r] AND D[d+r]) OR (D[m+r] AND NOT(D[d+r]));
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.38 VBSL

Vector Bitwise Select sets each bit in the destination to the corresponding bit from the first source operand when the original destination bit was 1, otherwise from the second source operand.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	1	0	D	0	1	Vn	Vd	0	0	0	1	N	Q	M	1	Vm
										op												

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VBSL{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VBSL{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if op == '00' then SEE "VEOR";
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	1	1	0	D	0	1	Vn	Vd	0	0	0	1	N	Q	M	1	Vm
										op												

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VBSL{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VBSL{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if op == '00' then SEE "VEOR";
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` An optional data type. It is ignored by assemblers, and does not affect the encoding.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
enumeration VBitOps {VBitOps_VBIF, VBitOps_VBIT, VBitOps_VBSL};

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        case operation of
            when VBitOps_VBIF  D[d+r] = (D[d+r] AND D[m+r]) OR (D[n+r] AND NOT(D[m+r]));
            when VBitOps_VBIT  D[d+r] = (D[n+r] AND D[m+r]) OR (D[d+r] AND NOT(D[m+r]));
            when VBitOps_VBSL  D[d+r] = (D[n+r] AND D[d+r]) OR (D[m+r] AND NOT(D[d+r]));
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.39 VCADD

Vector Complex Add.

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following computation on the corresponding complex number element pairs from the two source registers:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 90 or 270 degrees.
- The rotated complex number is added to the complex number from the first source register.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

ARMv8.3

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	rot	1	D	0	S	Vn	Vd	1	0	0	0	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCADD{<q>}.<dt> <Dd>, <Dn>, <Dm>, #<rotate>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCADD{<q>}.<dt> <Qd>, <Qn>, <Qm>, #<rotate>
```

Decode for all variants of this encoding

```
if !HaveFCADDExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 16 << UInt(S);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
elements = 64 DIV esize;
regs = if Q == '0' then 1 else 2;
```

T1

ARMv8.3

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	rot	1	D	0	S	Vn	Vd	1	0	0	0	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCADD{<q>}.<dt> <Dd>, <Dn>, <Dm>, #<rotate>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCADD{<q>}.<dt> <Qd>, <Qn>, <Qm>, #<rotate>
```

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveFCADDExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 16 << UInt(S);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
elements = 64 DIV esize;
regs = if Q == '0' then 1 else 2;
```

Assembler symbols

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "S" field. It can have the following values:

`F16`	when `S = 0`
`F32`	when `S = 1`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

`<rotate>` Is the rotation to be applied to elements in the second SIMD&FP source register, encoded in the "rot" field. It can have the following values:

`90`	when `rot = 0`
`270`	when `rot = 1`

Operation for all encodings

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
for r = 0 to regs-1
    operand1 = D[n+r];
    operand2 = D[m+r];
    operand3 = D[d+r];
    for e = 0 to (elements DIV 2)-1
        case rot of
            when '0'
                element1 = FPNeg(Elem[operand2,e*2+1,esize]);
                element3 = Elem[operand2,e*2,esize];
            when '1'
                element1 = Elem[operand2,e*2+1,esize];
                element3 = FPNeg(Elem[operand2,e*2,esize]);
        result1 = FPAdd(Elem[operand1,e*2,esize],element1,StandardFPSCRValue());
        result2 = FPAdd(Elem[operand1,e*2+1,esize],element3,StandardFPSCRValue());
        Elem[D[d+r],e*2,esize] = result1;
        Elem[D[d+r],e*2+1,esize] = result2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.40 VCEQ (immediate #0)

Vector Compare Equal to Zero takes each element in a vector, and compares it with zero. If it is equal to zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 18	17 16	15 12	11	10	9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	1	1	size	0 1	Vd	0	F	0 1 0	Q	M	0		Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCEQ{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCEQ{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 2	1 0	15 12	11	10	9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	1	D	1	1	size	0 1	Vd	0	F	0 1 0	Q	M	0		Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCEQ{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCEQ{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the operands, encoded in the "F:size" field. It can have the following values:

`I8`	when `F = 0`, `size = 00`
`I16`	when `F = 0`, `size = 01`
`I32`	when `F = 0`, `size = 10`
`F16`	when `F = 1`, `size = 01`
`F32`	when `F = 1`, `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                bits(esize) zero = FPZero('0');
                test_passed = FPCompareEQ(Elem[D[m+r],e,esize], zero, StandardFPSCRValue());
            else
                test_passed = (Elem[D[m+r],e,esize] == Zeros(esize));
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.41 VCEQ (register)

Vector Compare Equal takes each element in a vector, and compares it with the corresponding element of a second vector. If they are equal, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	1	0	D	size		Vn	Vd	1	0	0	0	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCEQ{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCEQ{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
int_operation = TRUE;  esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	0	0	D	0	sz	Vn	Vd	1	1	1	0	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCEQ{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCEQ{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
int_operation = FALSE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	1	1	1	1	1	0	D	size		Vn	Vd	1	0	0	0	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCEQ{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCEQ{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
int_operation = TRUE;  esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	0	1	1	1	1	0	D	0	sz	Vn	Vd	1	1	1	0	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCEQ{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCEQ{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
int_operation = FALSE;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>` For encoding A1 and A2: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1 and T2: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` For encoding A1 and T1: is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:

`I8` when `size = 00`

`I16` when `size = 01`

`I32` when `size = 10`

For encoding A2 and T2: is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

`F32` when `sz = 0`

`F16` when `sz = 1`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op2 = Elem[D[m+r],e,esize];
            if int_operation then
                test_passed = (op1 == op2);
            else
                test_passed = FPCompareEQ(op1, op2, StandardFPSCRValue());
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.42 VCGE (immediate #0)

Vector Compare Greater Than or Equal to Zero takes each element in a vector, and compares it with zero. If it is greater than or equal to zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	1	Vd		0	F	0	0	1	Q	M	0	Vm	

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCGE{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCGE{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	1	Vd		0	F	0	0	1	Q	M	0	Vm	

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCGE{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCGE{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the operands, encoded in the "F:size" field. It can have the following values:

`S8`	when `F = 0`, `size = 00`
`S16`	when `F = 0`, `size = 01`
`S32`	when `F = 0`, `size = 10`
`F16`	when `F = 1`, `size = 01`
`F32`	when `F = 1`, `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                bits(esize) zero = FPZero('0');
                test_passed = FPCompareGE(Elem[D[m+r],e,esize], zero, StandardFPSCRValue());
            else
                test_passed = (SInt(Elem[D[m+r],e,esize]) >= 0);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.43 VCGE (register)

Vector Compare Greater Than or Equal takes each element in a vector, and compares it with the corresponding element of a second vector. If the first is greater than or equal to the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers, unsigned integers, or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

This instruction is used by the pseudo-instruction VCLE (register). The pseudo-instruction is never the preferred disassembly.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 0 1	U	0	D	size	Vn	Vd	0 0 1 1	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
vtype = if U == '1' then VCGEtype_unsigned else VCGEtype_signed;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

A2

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 0 1	1	0	D	0	sz	Vn	Vd	1 1 1 0	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
vtype = VCGEtype_fp;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13	12	11 10 9 8	7	6	5 4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1	U	1 1 1 1	0	D	size	Vn	Vd	0 0 1 1	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
vtype = if U == '1' then VCGEtype_unsigned else VCGEtype_signed;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T2

15 14 13	12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1	1	1 1 1 1	0	D	0	sz	Vn	Vd	1 1 1 0	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
vtype = VCGEtype_fp;
case sz of
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>` For encoding A1 and A2: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1 and T2: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` For encoding A1 and T1: is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

`S8`	when `U = 0`, `size = 00`
`S16`	when `U = 0`, `size = 01`
`S32`	when `U = 0`, `size = 10`
`U8`	when `U = 1`, `size = 00`
`U16`	when `U = 1`, `size = 01`
`U32`	when `U = 1`, `size = 10`

For encoding A2 and T2: is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

`F32`	when `sz = 0`
`F16`	when `sz = 1`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
enumeration VCGEtype {VCGEtype_signed, VCGEtype_unsigned, VCGEtype_fp};

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op2 = Elem[D[m+r],e,esize];
            case vtype of
                when VCGEtype_signed    test_passed = (SInt(op1) >= SInt(op2));
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        when VCGEtype_unsigned  test_passed = (UInt(op1) >= UInt(op2));
        when VCGEtype_fp        test_passed = FPCompareGE(op1, op2, StandardFPSCRValue());
    Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.44 VCGT (immediate #0)

Vector Compare Greater Than Zero takes each element in a vector, and compares it with zero. If it is greater than zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 18	17 16	15 12	11	10	9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	1	1	size	0 1	Vd	0	F	0 0	0	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCGT{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCGT{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 12	11	10	9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	1	D	1	1	size	0 1	Vd	0	F	0 0	0	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCGT{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCGT{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the operands, encoded in the "F:size" field. It can have the following values:

`S8`	when `F = 0, size = 00`
`S16`	when `F = 0, size = 01`
`S32`	when `F = 0, size = 10`
`F16`	when `F = 1, size = 01`
`F32`	when `F = 1, size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                bits(esize) zero = FPZero('0');
                test_passed = FPCompareGT(Elem[D[m+r],e,esize], zero, StandardFPSCRValue());
            else
                test_passed = (SInt(Elem[D[m+r],e,esize]) > 0);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.45 VCGT (register)

Vector Compare Greater Than takes each element in a vector, and compares it with the corresponding element of a second vector. If the first is greater than the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers, unsigned integers, or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

This instruction is used by the pseudo-instruction VCLT (register). The pseudo-instruction is never the preferred disassembly.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	U	0	D	size		Vn	Vd	0	0	1	1	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
vtype = if U == '1' then VCGTtype_unsigned else VCGTtype_signed;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	1	0	D	1	sz	Vn	Vd	1	1	1	0	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
vtype = VCGTtype_fp;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	U	1	1	1	1	0	D	size		Vn	Vd	0	0	1	1	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
vtype = if U == '1' then VCGTtype_unsigned else VCGTtype_signed;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	1	1	1	1	1	0	D	1	sz	Vn	Vd	1	1	1	0	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
vtype = VCGTtype_fp;
case sz of
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>` For encoding A1 and A2: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1 and T2: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` For encoding A1 and T1: is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

`S8`	when `U = 0`, `size = 00`
`S16`	when `U = 0`, `size = 01`
`S32`	when `U = 0`, `size = 10`
`U8`	when `U = 1`, `size = 00`
`U16`	when `U = 1`, `size = 01`
`U32`	when `U = 1`, `size = 10`

For encoding A2 and T2: is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

`F32`	when `sz = 0`
`F16`	when `sz = 1`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
enumeration VCGTtype {VCGTtype_signed, VCGTtype_unsigned, VCGTtype_fp};

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op2 = Elem[D[m+r],e,esize];
            case vtype of
                when VCGTtype_signed    test_passed = (SInt(op1) > SInt(op2));
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    when VCGTtype_unsigned  test_passed = (UInt(op1) > UInt(op2));
    when VCGTtype_fp        test_passed = FPCompareGT(op1, op2, StandardFPSCRValue());
Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.46 VCLE (immediate #0)

Vector Compare Less Than or Equal to Zero takes each element in a vector, and compares it with zero. If it is less than or equal to zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	1	Vd		0	F	0	1	1	Q	M	0	Vm	

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCLE{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCLE{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	1	Vd		0	F	0	1	1	Q	M	0	Vm	

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCLE{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCLE{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the operands, encoded in the "F:size" field. It can have the following values:

`S8`	when `F = 0, size = 00`
`S16`	when `F = 0, size = 01`
`S32`	when `F = 0, size = 10`
`F16`	when `F = 1, size = 01`
`F32`	when `F = 1, size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                bits(esize) zero = FPZero('0');
                test_passed = FPCompareGE(zero, Elem[D[m+r],e,esize], StandardFPSCRValue());
            else
                test_passed = (SInt(Elem[D[m+r],e,esize]) <= 0);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.47 VCLE (register)

Vector Compare Less Than or Equal takes each element in a vector, and compares it with the corresponding element of a second vector. If the first is less than or equal to the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros

This instruction is a pseudo-instruction of the VCGE (register) instruction. This means that:

- The encodings in this description are named to match the encodings of VCGE (register).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VCGE (register) gives the operational pseudocode for this instruction.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 0 1	U	0	D	size	Vn	Vd	0 0 1 1	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCLE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VCGE{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>
```

and is never the preferred disassembly.

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCLE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VCGE{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>
```

and is never the preferred disassembly.

A2

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 0 1	1	0	D	0	sz	Vn	Vd	1 1 1 0	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCLE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VCGE{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>
```

and is never the preferred disassembly.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCLE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VCGE{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>
```

and is never the preferred disassembly.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	U	1	1	1	1	0	D	size		Vn	Vd	0	0	1	1	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCLE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VCGE{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>
```

and is never the preferred disassembly.

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCLE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VCGE{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>
```

and is never the preferred disassembly.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	1	1	1	1	1	0	D	0	sz	Vn	Vd	1	1	1	0	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCLE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VCGE{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>
```

and is never the preferred disassembly.

128-bit SIMD vector variant

Applies when `Q == 1`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VCLE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VCGE{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>
```

and is never the preferred disassembly.

Assembler symbols

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<c>` For encoding A1 and A2: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1 and T2: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` For encoding A1 and T1: is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

`S8` when `U = 0`, `size = 00`

`S16` when `U = 0`, `size = 01`

`S32` when `U = 0`, `size = 10`

`U8` when `U = 1`, `size = 00`

`U16` when `U = 1`, `size = 01`

`U32` when `U = 1`, `size = 10`

For encoding A2 and T2: is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

`F32` when `sz = 0`

`F16` when `sz = 1`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Operation for all encodings

The description of VCGE (register) gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.48 VCLS

Vector Count Leading Sign Bits counts the number of consecutive bits following the topmost bit, that are the same as the topmost bit, in each element in a vector, and places the results in a second vector. The count does not include the topmost bit itself.

The operand vector elements can be any one of 8-bit, 16-bit, or 32-bit signed integers.

The result vector elements are the same data type as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 18	17 16	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	1	1	size	0 0	Vd	0	1	0	0	0	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCLS{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCLS{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 2	1 0	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	1	D	1	1	size	0 0	Vd	0	1	0	0	0	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCLS{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCLS{<c>}{<q>}.<dt> <Qd>, <Qm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:

`S8`	when `size = 00`
`S16`	when `size = 01`
`S32`	when `size = 10`

The encoding `size = 11` is reserved.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = CountLeadingSignBits(Elem[D[m+r],e,esize])<esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.49 VCLT (immediate #0)

Vector Compare Less Than Zero takes each element in a vector, and compares it with zero. If it is less than zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 18	17 16	15 12	11	10	9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	1	1	size	0 1	Vd	0	F	1 0	0	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCLT{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCLT{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 12	11	10	9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	1	D	1	1	size	0 1	Vd	0	F	1 0	0	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCLT{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCLT{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the operands, encoded in the "F:size" field. It can have the following values:

`S8`	when `F = 0, size = 00`
`S16`	when `F = 0, size = 01`
`S32`	when `F = 0, size = 10`
`F16`	when `F = 1, size = 01`
`F32`	when `F = 1, size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                bits(esize) zero = FPZero('0');
                test_passed = FPCompareGT(zero, Elem[D[m+r],e,esize], StandardFPSCRValue());
            else
                test_passed = (SInt(Elem[D[m+r],e,esize]) < 0);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.50 VCLT (register)

Vector Compare Less Than takes each element in a vector, and compares it with the corresponding element of a second vector. If the first is less than the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros

This instruction is a pseudo-instruction of the VCGT (register) instruction. This means that:

- The encodings in this description are named to match the encodings of VCGT (register).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VCGT (register) gives the operational pseudocode for this instruction.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	U	0	D	size		Vn	Vd	0	0	1	1	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCLT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VCGT{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>
```

and is never the preferred disassembly.

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCLT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VCGT{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>
```

and is never the preferred disassembly.

A2

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	1	0	D	1	sz	Vn	Vd	1	1	1	0	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCLT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VCGT{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>
```

and is never the preferred disassembly.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCLT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VCGT{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>
```

and is never the preferred disassembly.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	U	1	1	1	1	0	D	size		Vn	Vd	0	0	1	1	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCLT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VCGT{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>
```

and is never the preferred disassembly.

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCLT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VCGT{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>
```

and is never the preferred disassembly.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	1	1	1	1	1	0	D	1	sz	Vn	Vd	1	1	1	0	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCLT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VCGT{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>
```

and is never the preferred disassembly.

128-bit SIMD vector variant

Applies when `Q == 1`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VCLT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VCGT{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>
```

and is never the preferred disassembly.

Assembler symbols

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<c>` For encoding A1 and A2: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1 and T2: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` For encoding A1 and T1: is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

`S8`	when `U = 0`, `size = 00`
`S16`	when `U = 0`, `size = 01`
`S32`	when `U = 0`, `size = 10`
`U8`	when `U = 1`, `size = 00`
`U16`	when `U = 1`, `size = 01`
`U32`	when `U = 1`, `size = 10`

For encoding A2 and T2: is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

`F32`	when `sz = 0`
`F16`	when `sz = 1`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Operation for all encodings

The description of VCGT (register) gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.51 VCLZ

Vector Count Leading Zeros counts the number of consecutive zeros, starting from the most significant bit, in each element in a vector, and places the results in a second vector.

The operand vector elements can be any one of 8-bit, 16-bit, or 32-bit integers. There is no distinction between signed and unsigned integers.

The result vector elements are the same data type as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 18	17 16	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	1	1	size	0	0	Vd	0	1	0	0	1	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCLZ{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCLZ{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 2	1	0	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	1	D	1	1	size	0	0	Vd	0	1	0	0	1	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCLZ{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCLZ{<c>}{<q>}.<dt> <Qd>, <Qm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:

`I8`	when `size = 00`
`I16`	when `size = 01`
`I32`	when `size = 10`

The encoding `size = 11` is reserved.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = CountLeadingZeroBits(Elem[D[m+r],e,esize])<esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.52 VCMLA

Vector Complex Multiply Accumulate.

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following computation on the corresponding complex number element pairs from the two source registers and the destination register:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees.
- The two elements of the transformed complex number are multiplied by:
  - The real element of the complex number from the first source register, if the transformation was a rotation by 0 or 180 degrees.
  - The imaginary element of the complex number from the first source register, if the transformation was a rotation by 90 or 270 degrees.
- The complex number resulting from that multiplication is added to the complex number from the destination register.

The multiplication and addition operations are performed as a fused multiply-add, without any intermediate rounding.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

ARMv8.3

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	rot	D	1	S	Vn	Vd	1	0	0	0	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCMLA{<q>}.<dt> <Dd>, <Dn>, <Dm>, #<rotate>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCMLA{<q>}.<dt> <Qd>, <Qn>, <Qm>, #<rotate>
```

Decode for all variants of this encoding

```
if !HaveFCADDExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
esize = 16 << UInt(S);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
elements = 64 DIV esize;
regs = if Q == '0' then 1 else 2;
```

T1

ARMv8.3

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	1	1	1	0	rot	D	1	S		Vn	Vd	1	0	0	0	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCMLA{<q>}.<dt> <Dd>, <Dn>, <Dm>, #<rotate>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCMLA{<q>}.<dt> <Qd>, <Qn>, <Qm>, #<rotate>
```

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveFCADDExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
esize = 16 << UInt(S);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
elements = 64 DIV esize;
regs = if Q == '0' then 1 else 2;
```

Assembler symbols

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "S" field. It can have the following values:

`F16`	when `S = 0`
`F32`	when `S = 1`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

`<rotate>` Is the rotation to be applied to elements in the second SIMD&FP source register, encoded in the "rot" field. It can have the following values:

`0`	when `rot = 00`
`90`	when `rot = 01`
`180`	when `rot = 10`
`270`	when `rot = 11`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
for r = 0 to regs-1
    operand1 = D[n+r];
    operand2 = D[m+r];
    operand3 = D[d+r];
    for e = 0 to (elements DIV 2)-1
        case rot of
            when '00'
               element1 = Elem[operand2,e*2,esize];
               element2 = Elem[operand1,e*2,esize];
               element3 = Elem[operand2,e*2+1,esize];
               element4 = Elem[operand1,e*2,esize];
            when '01'
               element1 = FPNeg(Elem[operand2,e*2+1,esize]);
               element2 = Elem[operand1,e*2+1,esize];
               element3 = Elem[operand2,e*2,esize];
               element4 = Elem[operand1,e*2+1,esize];
            when '10'
               element1 = FPNeg(Elem[operand2,e*2,esize]);
               element2 = Elem[operand1,e*2,esize];
               element3 = FPNeg(Elem[operand2,e*2+1,esize]);
               element4 = Elem[operand1,e*2,esize];
            when '11'
               element1 = Elem[operand2,e*2+1,esize];
               element2 = Elem[operand1,e*2+1,esize];
               element3 = FPNeg(Elem[operand2,e*2,esize]);
               element4 = Elem[operand1,e*2+1,esize];
        result1 = FPMulAdd(Elem[operand3,e*2,esize],element2,element1, StandardFPSCRValue());
        result2 = FPMulAdd(Elem[operand3,e*2+1,esize],element4,element3, StandardFPSCRValue());
        Elem[D[d+r],e*2,esize] = result1;
        Elem[D[d+r],e*2+1,esize] = result2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.53 VCMLA (by element)

Vector Complex Multiply Accumulate (by element).

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following computation on complex numbers from the first source register and the destination register with the specified complex number from the second source register:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees.
- The two elements of the transformed complex number are multiplied by:
  - The real element of the complex number from the first source register, if the transformation was a rotation by 0 or 180 degrees.
  - The imaginary element of the complex number from the first source register, if the transformation was a rotation by 90 or 270 degrees.
- The complex number resulting from that multiplication is added to the complex number from the destination register.

The multiplication and addition operations are performed as a fused multiply-add, without any intermediate rounding.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

ARMv8.3

31 30 29 28	27 26 25 24	23	22	21 20	19 16	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	S	D	rot	Vn	Vd	1	0	0	0	N	Q	M	0	Vm

64-bit SIMD vector of half-precision floating-point variant

Applies when `S == 0 && Q == 0`.

```
VCMLA{<q>}.F16 <Dd>, <Dn>, <Dm>[<index>], #<rotate>
```

64-bit SIMD vector of single-precision floating-point variant

Applies when `S == 1 && Q == 0`.

```
VCMLA{<q>}.F32 <Dd>, <Dn>, <Dm>[0], #<rotate>
```

128-bit SIMD vector of half-precision floating-point variant

Applies when `S == 0 && Q == 1`.

```
VCMLA{<q>}.F16 <Qd>, <Qn>, <Dm>[<index>], #<rotate>
```

128-bit SIMD vector of single-precision floating-point variant

Applies when `S == 1 && Q == 1`.

```
VCMLA{<q>}.F32 <Qd>, <Qn>, <Dm>[0], #<rotate>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if !HaveFCADDExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn);
m = if S=='1' then UInt(M:Vm) else UInt(Vm);
esize = 16 << UInt(S);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
elements = 64 DIV esize;
regs = if Q == '0' then 1 else 2;
index = if S=='1' then 0 else UInt(M);
```

T1

ARMv8.3

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	1	1	1	1	0	S	D	rot		Vn	Vd	1	0	0	0	N	Q	M	0	Vm

64-bit SIMD vector of half-precision floating-point variant

Applies when `S == 0 && Q == 0`.

```
VCMLA{<q>}.F16 <Dd>, <Dn>, <Dm>[<index>], #<rotate>
```

64-bit SIMD vector of single-precision floating-point variant

Applies when `S == 1 && Q == 0`.

```
VCMLA{<q>}.F32 <Dd>, <Dn>, <Dm>[0], #<rotate>
```

128-bit SIMD vector of half-precision floating-point variant

Applies when `S == 0 && Q == 1`.

```
VCMLA{<q>}.F16 <Qd>, <Qn>, <Dm>[<index>], #<rotate>
```

128-bit SIMD vector of single-precision floating-point variant

Applies when `S == 1 && Q == 1`.

```
VCMLA{<q>}.F32 <Qd>, <Qn>, <Dm>[0], #<rotate>
```

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveFCADDExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn);
m = if S=='1' then UInt(M:Vm) else UInt(Vm);
esize = 16 << UInt(S);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
elements = 64 DIV esize;
regs = if Q == '0' then 1 else 2;
index = if S=='1' then 0 else UInt(M);
```

Assembler symbols

<q>	See *Standard assembler syntax fields* on page F2-4080.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` For the half-precision scalar variant: is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm" field.

For the single-precision scalar variant: is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

`<index>` Is the element index in the range 0 to 1, encoded in the "M" field.

`<rotate>` Is the rotation to be applied to elements in the second SIMD&FP source register, encoded in the "rot" field. It can have the following values:

`0`	when `rot = 00`
`90`	when `rot = 01`
`180`	when `rot = 10`
`270`	when `rot = 11`

Operation for all encodings

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
for r = 0 to regs-1
    operand1 = D[n+r];
    operand2 = Din[m];
    operand3 = D[d+r];
    for e = 0 to (elements DIV 2)-1
        case rot of
            when '00'
                element1 = Elem[operand2,index*2,esize];
                element2 = Elem[operand1,e*2,esize];
                element3 = Elem[operand2,index*2+1,esize];
                element4 = Elem[operand1,e*2,esize];
            when '01'
                element1 = FPNeg(Elem[operand2,index*2+1,esize]);
                element2 = Elem[operand1,e*2+1,esize];
                element3 = Elem[operand2,index*2,esize];
                element4 = Elem[operand1,e*2+1,esize];
            when '10'
                element1 = FPNeg(Elem[operand2,index*2,esize]);
                element2 = Elem[operand1,e*2,esize];
                element3 = FPNeg(Elem[operand2,index*2+1,esize]);
                element4 = Elem[operand1,e*2,esize];
            when '11'
                element1 = Elem[operand2,index*2+1,esize];
                element2 = Elem[operand1,e*2+1,esize];
                element3 = FPNeg(Elem[operand2,index*2,esize]);
                element4 = Elem[operand1,e*2+1,esize];
        result1 = FPMulAdd(Elem[operand3,e*2,esize],element2,element1, StandardFPSCRValue());
        result2 = FPMulAdd(Elem[operand3,e*2+1,esize],element4,element3,StandardFPSCRValue());
        Elem[D[d+r],e*2,esize] = result1;
        Elem[D[d+r],e*2+1,esize] = result2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.54 VCMP

Vector Compare compares two floating-point registers, or one floating-point register and zero. It writes the result to the FPSCR flags. These are normally transferred to the PSTATE.{N, Z, C, V} Condition flags by a subsequent `VMRS` instruction.

It raises an Invalid Operation exception only if either operand is a signaling NaN.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	1	1	1	0	1	D	1	1	0	1	0	0	Vd	1	0	size		0	1	M	0	Vm
cond																		E				

Half-precision scalar variant

Applies when `size == 01`.

```
VCMP{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VCMP{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VCMP{<c>}{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
quiet_nan_exc = (E == '1');  with_zero = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

A2

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3	2	1	0
!=1111	1	1	1	0	1	D	1	1	0	1	0	1	Vd	1	0	size		0	1	(0)	0	(0)	(0)	(0)	(0)
cond																		E							

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Half-precision scalar variant

Applies when `size == 01`.

```
VCMP{<c>}{<q>}.F16 <Sd>, #0.0
```

Single-precision scalar variant

Applies when `size == 10`.

```
VCMP{<c>}{<q>}.F32 <Sd>, #0.0
```

Double-precision scalar variant

Applies when `size == 11`.

```
VCMP{<c>}{<q>}.F64 <Dd>, #0.0
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
quiet_nan_exc = (E == '1');  with_zero = TRUE;
case size of
    when '01' esize = 16; d = UInt(Vd:D);
    when '10' esize = 32; d = UInt(Vd:D);
    when '11' esize = 64; d = UInt(D:Vd);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	0	1	1	1	0	1	D	1	1	0	1	0	0	Vd	1	0	size		0	1	M	0	Vm
																					E				

Half-precision scalar variant

Applies when `size == 01`.

```
VCMP{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VCMP{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VCMP{<c>}{<q>}.F64 <Dd>, <Dm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
quiet_nan_exc = (E == '1');  with_zero = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15 … 12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	0	1	0	1	Vd	1	0	size		0	1	(0)	0	(0)	(0)	(0)	(0)
																					E							

Half-precision scalar variant

Applies when `size == 01`.

```
VCMP{<c>}{<q>}.F16 <Sd>, #0.0
```

Single-precision scalar variant

Applies when `size == 10`.

```
VCMP{<c>}{<q>}.F32 <Sd>, #0.0
```

Double-precision scalar variant

Applies when `size == 11`.

```
VCMP{<c>}{<q>}.F64 <Dd>, #0.0
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
quiet_nan_exc = (E == '1');  with_zero = TRUE;
case size of
    when '01' esize = 16; d = UInt(Vd:D);
    when '10' esize = 32; d = UInt(Vd:D);
    when '11' esize = 64; d = UInt(D:Vd);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

`<Sm>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

NaNs

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands are NaNs, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. This results in the FPSCR flags being set as N=0, Z=0, C=1 and V=1.

`VCMPE` raises an Invalid Operation exception if either operand is any type of NaN, and is suitable for testing for <, <=, >, >=, and other predicates that raise an exception when the operands are unordered.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    bits(4) nzcv;
    case esize of
        when 16
            bits(16) op16 = if with_zero then FPZero('0') else S[m]<15:0>;
            nzcv = FPCompare(S[d]<15:0>, op16, quiet_nan_exc, FPSCR);
        when 32
            bits(32) op32 = if with_zero then FPZero('0') else S[m];
            nzcv = FPCompare(S[d], op32, quiet_nan_exc, FPSCR);
        when 64
            bits(64) op64 = if with_zero then FPZero('0') else D[m];
            nzcv = FPCompare(D[d], op64, quiet_nan_exc, FPSCR);

    FPSCR.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.55 VCMPE

Vector Compare, raising Invalid Operation on NaN compares two floating-point registers, or one floating-point register and zero. It writes the result to the FPSCR flags. These are normally transferred to the PSTATE.{N, Z, C, V} Condition flags by a subsequent `VMRS` instruction.

It raises an Invalid Operation exception if either operand is any type of NaN.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31–28	27 26 25 24	23	22	21 20	19	18 17 16	15–12	11 10	9 8	7	6	5	4	3–0
!=1111	1 1 1 0	1	D	1 1	0	1 0 0	Vd	1 0	size	1	1	M	0	Vm
cond										E				

Half-precision scalar variant

Applies when `size == 01`.

```
VCMPE{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VCMPE{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VCMPE{<c>}{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
quiet_nan_exc = (E == '1');  with_zero = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

A2

31–28	27 26 25 24	23	22	21 20	19	18 17 16	15–12	11 10	9 8	7	6	5	4	3 2 1 0
!=1111	1 1 1 0	1	D	1 1	0	1 0 1	Vd	1 0	size	1	1	(0)	0	(0) (0) (0) (0)
cond										E				

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Half-precision scalar variant

Applies when `size == 01`.

```
VCMPE{<c>}{<q>}.F16 <Sd>, #0.0
```

Single-precision scalar variant

Applies when `size == 10`.

```
VCMPE{<c>}{<q>}.F32 <Sd>, #0.0
```

Double-precision scalar variant

Applies when `size == 11`.

```
VCMPE{<c>}{<q>}.F64 <Dd>, #0.0
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
quiet_nan_exc = (E == '1');  with_zero = TRUE;
case size of
    when '01' esize = 16; d = UInt(Vd:D);
    when '10' esize = 32; d = UInt(Vd:D);
    when '11' esize = 64; d = UInt(D:Vd);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	0	1	1	1	0	1	D	1	1	0	1	0	0	Vd	1	0	size		1	1	M	0	Vm
																					E				

Half-precision scalar variant

Applies when `size == 01`.

```
VCMPE{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VCMPE{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VCMPE{<c>}{<q>}.F64 <Dd>, <Dm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
quiet_nan_exc = (E == '1');  with_zero = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15–12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	0	1	0	1	Vd	1	0	size		1	1	(0)	0	(0)	(0)	(0)	(0)
																					E							

Half-precision scalar variant

Applies when `size == 01`.

```
VCMPE{<c>}{<q>}.F16 <Sd>, #0.0
```

Single-precision scalar variant

Applies when `size == 10`.

```
VCMPE{<c>}{<q>}.F32 <Sd>, #0.0
```

Double-precision scalar variant

Applies when `size == 11`.

```
VCMPE{<c>}{<q>}.F64 <Dd>, #0.0
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
quiet_nan_exc = (E == '1');  with_zero = TRUE;
case size of
    when '01' esize = 16; d = UInt(Vd:D);
    when '10' esize = 32; d = UInt(Vd:D);
    when '11' esize = 64; d = UInt(D:Vd);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

`<Sm>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

NaNs

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands are NaNs, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. This results in the FPSCR flags being set as N=0, Z=0, C=1 and V=1.

`VCMPE` raises an Invalid Operation exception if either operand is any type of NaN, and is suitable for testing for <, <=, >, >=, and other predicates that raise an exception when the operands are unordered.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    bits(4) nzcv;
    case esize of
        when 16
            bits(16) op16 = if with_zero then FPZero('0') else S[m]<15:0>;
            nzcv = FPCompare(S[d]<15:0>, op16, quiet_nan_exc, FPSCR);
        when 32
            bits(32) op32 = if with_zero then FPZero('0') else S[m];
            nzcv = FPCompare(S[d], op32, quiet_nan_exc, FPSCR);
        when 64
            bits(64) op64 = if with_zero then FPZero('0') else D[m];
            nzcv = FPCompare(D[d], op64, quiet_nan_exc, FPSCR);

    FPSCR.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.56 VCNT

Vector Count Set Bits counts the number of bits that are one in each element in a vector, and places the results in a second vector.

The operand vector elements must be 8-bit fields.

The result vector elements are 8-bit integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 18	17 16	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	1	1	size	0 0	Vd	0	1	0	1	0	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCNT{<c>}{<q>}.8 <Dd>, <Dm> // Encoded as Q = 0
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCNT{<c>}{<q>}.8 <Qd>, <Qm> // Encoded as Q = 1
```

Decode for all variants of this encoding

```
if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8;  elements = 8;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 2	1 0	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	1	D	1	1	size	0 0	Vd	0	1	0	1	0	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCNT{<c>}{<q>}.8 <Dd>, <Dm> // Encoded as Q = 0
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCNT{<c>}{<q>}.8 <Qd>, <Qm> // Encoded as Q = 1
```

Decode for all variants of this encoding

```
if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8;  elements = 8;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = BitCount(Elem[D[m+r],e,esize])<esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.57 VCVT (from single-precision to BFloat16, Advanced SIMD)

Vector Convert from single-precision to BFloat16 converts each 32-bit element in a vector from single-precision floating-point to BFloat16 format, and writes the result into a second vector. The result vector elements are half the width of the source vector elements.

Unlike the BFloat16 multiplication instructions, this instruction uses the Round to Nearest rounding mode, and can generate a floating-point exception that causes cumulative exception bits in the FPSCR to be set.

A1

ARMv8.6

31 30 29 28	27 26 25 24	23	22	21 20	19 18	17 16	15 12	11	10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	1 1	0 1	1 0	Vd	0	1 1 0	0	1	M	0	Vm

A1 variant

```
VCVT{<c>}{<q>}.BF16.F32 <Dd>, <Qm>
```

Decode for this encoding

```
if !HaveAArch32BF16Ext() then UNDEFINED;
if Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer m = UInt(M:Vm);
```

T1

ARMv8.6

15 14 13 12	11 10 9 8	7	6	5 4	3 2	1 0	15 12	11	10 9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	1	D	1 1	0 1	1 0	Vd	0	1 1 0	0	1	M	0	Vm

T1 variant

```
VCVT{<c>}{<q>}.BF16.F32 <Dd>, <Qm>
```

Decode for this encoding

```
if !HaveAArch32BF16Ext() then UNDEFINED;
if Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer m = UInt(M:Vm);
```

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
bits(128) operand;
bits(64) result;

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    operand = Q[m>>1];
    for e = 0 to 3
        bits(32) op = Elem[operand, e, 32];
        Elem[result, e, 16] = FPConvertBF(op, StandardFPSCRValue());
    D[d] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.58 VCVT (between double-precision and single-precision)

Convert between double-precision and single-precision does one of the following:

- Converts the value in a double-precision register to single-precision and writes the result to a single-precision register.
- Converts the value in a single-precision register to double-precision and writes the result to a double-precision register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	1	1	1	0	1	D	1	1	0	1	1	1	Vd	1	0	1	x	1	1	M	0	Vm
cond																size						

Single-precision to double-precision variant

Applies when `size == 10`.

```
VCVT{<c>}{<q>}.F64.F32 <Dd>, <Sm>
```

Double-precision to single-precision variant

Applies when `size == 11`.

```
VCVT{<c>}{<q>}.F32.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
double_to_single = (size == '11');
d = if double_to_single then UInt(Vd:D) else UInt(D:Vd);
m = if double_to_single then UInt(M:Vm) else UInt(Vm:M);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	0	1	1	1	0	1	D	1	1	0	1	1	1	Vd	1	0	1	x	1	1	M	0	Vm
																			size						

Single-precision to double-precision variant

Applies when `size == 10`.

```
VCVT{<c>}{<q>}.F64.F32 <Dd>, <Sm>
```

Double-precision to single-precision variant

Applies when `size == 11`.

```
VCVT{<c>}{<q>}.F32.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
double_to_single = (size == '11');
d = if double_to_single then UInt(Vd:D) else UInt(D:Vd);
m = if double_to_single then UInt(M:Vm) else UInt(Vm:M);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if double_to_single then
        S[d] = FPConvert(D[m], FPSCR);
    else
        D[d] = FPConvert(S[m], FPSCR);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.59 VCVT (between half-precision and single-precision, Advanced SIMD)

Vector Convert between half-precision and single-precision converts each element in a vector from single-precision to half-precision floating-point, or from half-precision to single-precision, and places the results in a second vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	0	Vd		0	1	1	op	0	0	M	0	Vm	

Half-precision to single-precision variant

Applies when `op == 1`.

```
VCVT{<c>}{<q>}.F32.F16 <Qd>, <Dm> // Encoded as op = 1
```

Single-precision to half-precision variant

Applies when `op == 0`.

```
VCVT{<c>}{<q>}.F16.F32 <Dd>, <Qm> // Encoded as op = 0
```

Decode for all variants of this encoding

```
if size != '01' then UNDEFINED;
half_to_single = (op == '1');
if half_to_single && Vd<0> == '1' then UNDEFINED;
if !half_to_single && Vm<0> == '1' then UNDEFINED;
esize = 16;  elements = 4;
m = UInt(M:Vm);  d = UInt(D:Vd);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	0	Vd		0	1	1	op	0	0	M	0	Vm	

Half-precision to single-precision variant

Applies when `op == 1`.

```
VCVT{<c>}{<q>}.F32.F16 <Qd>, <Dm> // Encoded as op = 1
```

Single-precision to half-precision variant

Applies when `op == 0`.

```
VCVT{<c>}{<q>}.F16.F32 <Dd>, <Qm> // Encoded as op = 0
```

Decode for all variants of this encoding

```
if size != '01' then UNDEFINED;
half_to_single = (op == '1');
if half_to_single && Vd<0> == '1' then UNDEFINED;
if !half_to_single && Vm<0> == '1' then UNDEFINED;
esize = 16;  elements = 4;
m = UInt(M:Vm);  d = UInt(D:Vd);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        if half_to_single then
            Elem[Q[d>>1],e,32] = FPConvert(Elem[Din[m],e,16], StandardFPSCRValue());
        else
            Elem[D[d],e,16] = FPConvert(Elem[Qin[m>>1],e,32], StandardFPSCRValue());
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.60 VCVT (between floating-point and integer, Advanced SIMD)

Vector Convert between floating-point and integer converts each element in a vector from floating-point to integer, or from integer to floating-point, and places the results in a second vector.

The vector elements are the same type, and are floating-point numbers or integers. Signed and unsigned integers are distinct.

The floating-point to integer operation uses the Round towards Zero rounding mode. The integer to floating-point operation uses the Round to Nearest rounding mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21 20	19 18	17 16	15 12	11	10 9	8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	1 1	size	1 1	Vd	0	1 1	op	Q	M	0		Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCVT{<c>}{<q>}.<dt1>.<dt2> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCVT{<c>}{<q>}.<dt1>.<dt2> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
to_integer = (op<1> == '1');  unsigned = (op<0> == '1');
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5 4	3 2 1 0	15 12	11	10 9	8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	1	D	1 1	size	1 1	Vd	0	1 1	op	Q	M	0		Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCVT{<c>}{<q>}.<dt1>.<dt2> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCVT{<c>}{<q>}.<dt1>.<dt2> <Qd>, <Qm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
to_integer = (op<1> == '1');  unsigned = (op<0> == '1');
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt1>` Is the data type for the elements of the destination vector, encoded in the "size:op" field. It can have the following values:

F16	when `size = 01`, `op = 0x`
S16	when `size = 01`, `op = 10`
U16	when `size = 01`, `op = 11`
F32	when `size = 10`, `op = 0x`
S32	when `size = 10`, `op = 10`
U32	when `size = 10`, `op = 11`

`<dt2>` Is the data type for the elements of the source vector, encoded in the "size:op" field. It can have the following values:

S16	when `size = 01`, `op = 00`
U16	when `size = 01`, `op = 01`
F16	when `size = 01`, `op = 1x`
S32	when `size = 10`, `op = 00`
U32	when `size = 10`, `op = 01`
F32	when `size = 10`, `op = 1x`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(esize) result;
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[m+r],e,esize];
            if to_integer then
                result = FPToFixed(op1, 0, unsigned, StandardFPSCRValue(), FPRounding_ZERO);
            else
                result = FixedToFP(op1, 0, unsigned, StandardFPSCRValue(), FPRounding_TIEEVEN);
            Elem[D[d+r],e,esize] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.61 VCVT (floating-point to integer, floating-point)

Convert floating-point to integer with Round towards Zero converts a value in a register from floating-point to a 32-bit integer, using the Round towards Zero rounding mode, and places the result in a second register.

VCVT (between floating-point and fixed-point, floating-point) describes conversions between floating-point and 16-bit integers.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	1	1	1	0	1	D	1	1	1	1	0	x	Vd	1	0	size		1	1	M	0	Vm
cond										opc2								op				

Half-precision scalar variant

Applies when `opc2 == 100 && size == 01`.

```
VCVT{<c>}{<q>}.U32.F16 <Sd>, <Sm>
```

Half-precision scalar variant

Applies when `opc2 == 101 && size == 01`.

```
VCVT{<c>}{<q>}.S32.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `opc2 == 100 && size == 10`.

```
VCVT{<c>}{<q>}.U32.F32 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `opc2 == 101 && size == 10`.

```
VCVT{<c>}{<q>}.S32.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `opc2 == 100 && size == 11`.

```
VCVT{<c>}{<q>}.U32.F64 <Sd>, <Dm>
```

Double-precision scalar variant

Applies when `opc2 == 101 && size == 11`.

```
VCVT{<c>}{<q>}.S32.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
if opc2 != '000' && opc2 != '10x' then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
to_integer = (opc2<2> == '1');
if to_integer then
    unsigned = (opc2<0> == '0');
    rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR);
    d = UInt(Vd:D);
    case size of
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        when '01' esize = 16; m = UInt(Vm:M);
        when '10' esize = 32; m = UInt(Vm:M);
        when '11' esize = 64; m = UInt(M:Vm);
else
    unsigned = (op == '0');
    rounding = FPRoundingMode(FPSCR);
    m = UInt(Vm:M);
    case size of
        when '01' esize = 16; d = UInt(Vd:D);
        when '10' esize = 32; d = UInt(Vd:D);
        when '11' esize = 64; d = UInt(D:Vd);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	0	1	1	1	0	1	D	1	1	1	1	0 x	Vd		1	0	size		1	1	M	0	Vm	
													opc2								op					

Half-precision scalar variant

Applies when `opc2 == 100 && size == 01`.

```
VCVT{<c>}{<q>}.U32.F16 <Sd>, <Sm>
```

Half-precision scalar variant

Applies when `opc2 == 101 && size == 01`.

```
VCVT{<c>}{<q>}.S32.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `opc2 == 100 && size == 10`.

```
VCVT{<c>}{<q>}.U32.F32 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `opc2 == 101 && size == 10`.

```
VCVT{<c>}{<q>}.S32.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `opc2 == 100 && size == 11`.

```
VCVT{<c>}{<q>}.U32.F64 <Sd>, <Dm>
```

Double-precision scalar variant

Applies when `opc2 == 101 && size == 11`.

```
VCVT{<c>}{<q>}.S32.F64 <Sd>, <Dm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if opc2 != '000' && opc2 != '10x' then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
to_integer = (opc2<2> == '1');
if to_integer then
    unsigned = (opc2<0> == '0');
    rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR);
    d = UInt(Vd:D);
    case size of
        when '01' esize = 16; m = UInt(Vm:M);
        when '10' esize = 32; m = UInt(Vm:M);
        when '11' esize = 64; m = UInt(M:Vm);
else
    unsigned = (op == '0');
    rounding = FPRoundingMode(FPSCR);
    m = UInt(Vm:M);
    case size of
        when '01' esize = 16; d = UInt(Vd:D);
        when '10' esize = 32; d = UInt(Vd:D);
        when '11' esize = 64; d = UInt(D:Vd);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings

Related encodings: See *Floating-point data-processing* on page F3-4131 for the T32 instruction set, or *Floating-point data-processing* on page F4-4214 for the A32 instruction set.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Sd>`	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
`<Sm>`	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
`<Dm>`	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_integer then
        case esize of
            when 16
                S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR, rounding);
            when 32
                S[d] = FPToFixed(S[m], 0, unsigned, FPSCR, rounding);
            when 64
                S[d] = FPToFixed(D[m], 0, unsigned, FPSCR, rounding);
    else
        case esize of
            when 16
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    bits(16) fp16 = FixedToFP(S[m], 0, unsigned, FPSCR, rounding);
    S[d] = Zeros(16):fp16;
when 32
    S[d] = FixedToFP(S[m], 0, unsigned, FPSCR, rounding);
when 64
    D[d] = FixedToFP(S[m], 0, unsigned, FPSCR, rounding);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.62 VCVT (integer to floating-point, floating-point)

Convert integer to floating-point converts a 32-bit integer to floating-point using the rounding mode specified by the FPSCR, and places the result in a second register.

VCVT (between floating-point and fixed-point, floating-point) describes conversions between floating-point and 16-bit integers.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31–28	27 26 25 24	23	22	21 20	19	18–16	15–12	11 10	9 8	7	6	5	4	3–0
!=1111	1 1 1 0	1	D	1 1	1	0 0 0	Vd	1 0	size	op	1	M	0	Vm
cond						opc2								

Half-precision scalar variant

Applies when `size == 01`.

```
VCVT{<c>}{<q>}.F16.<dt> <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VCVT{<c>}{<q>}.F32.<dt> <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VCVT{<c>}{<q>}.F64.<dt> <Dd>, <Sm>
```

Decode for all variants of this encoding

```
if opc2 != '000' && opc2 != '10x' then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
to_integer = (opc2<2> == '1');
if to_integer then
    unsigned = (opc2<0> == '0');
    rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR);
    d = UInt(Vd:D);
    case size of
        when '01' esize = 16; m = UInt(Vm:M);
        when '10' esize = 32; m = UInt(Vm:M);
        when '11' esize = 64; m = UInt(M:Vm);
else
    unsigned = (op == '0');
    rounding = FPRoundingMode(FPSCR);
    m = UInt(Vm:M);
    case size of
        when '01' esize = 16; d = UInt(Vd:D);
        when '10' esize = 32; d = UInt(Vd:D);
        when '11' esize = 64; d = UInt(D:Vd);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	0	1	1	1	0	1	D	1	1	1	0 0 0		Vd		1	0	size		op	1	M	0	Vm	
													opc2													

Half-precision scalar variant

Applies when `size == 01`.

```
VCVT{<c>}{<q>}.F16.<dt> <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VCVT{<c>}{<q>}.F32.<dt> <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VCVT{<c>}{<q>}.F64.<dt> <Dd>, <Sm>
```

Decode for all variants of this encoding

```
if opc2 != '000' && opc2 != '10x' then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
to_integer = (opc2<2> == '1');
if to_integer then
    unsigned = (opc2<0> == '0');
    rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR);
    d = UInt(Vd:D);
    case size of
        when '01' esize = 16; m = UInt(Vm:M);
        when '10' esize = 32; m = UInt(Vm:M);
        when '11' esize = 64; m = UInt(M:Vm);
else
    unsigned = (op == '0');
    rounding = FPRoundingMode(FPSCR);
    m = UInt(Vm:M);
    case size of
        when '01' esize = 16; d = UInt(Vd:D);
        when '10' esize = 32; d = UInt(Vd:D);
        when '11' esize = 64; d = UInt(D:Vd);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Notes for all encodings

Related encodings: See *Floating-point data-processing* on page F3-4131 for the T32 instruction set, or *Floating-point data-processing* on page F4-4214 for the A32 instruction set.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the operand, encoded in the "op" field. It can have the following values:

`U32`	when `op = 0`
`S32`	when `op = 1`

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Sm>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_integer then
        case esize of
            when 16
                S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR, rounding);
            when 32
                S[d] = FPToFixed(S[m], 0, unsigned, FPSCR, rounding);
            when 64
                S[d] = FPToFixed(D[m], 0, unsigned, FPSCR, rounding);
    else
        case esize of
            when 16
                bits(16) fp16 = FixedToFP(S[m], 0, unsigned, FPSCR, rounding);
                S[d] = Zeros(16):fp16;
            when 32
                S[d] = FixedToFP(S[m], 0, unsigned, FPSCR, rounding);
            when 64
                D[d] = FixedToFP(S[m], 0, unsigned, FPSCR, rounding);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.63 VCVT (between floating-point and fixed-point, Advanced SIMD)

Vector Convert between floating-point and fixed-point converts each element in a vector from floating-point to fixed-point, or from fixed-point to floating-point, and places the results in a second vector.

The vector elements are the same type, and are floating-point numbers or integers. Signed and unsigned integers are distinct.

The floating-point to fixed-point operation uses the Round towards Zero rounding mode. The fixed-point to floating-point operation uses the Round to Nearest rounding mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	16	15 12	11 10	9 8	7	6	5	4	3 0
1 1 1 1	0 0 1	U	1	D	imm6	Vd	1 1	op	0	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `imm6 != 000xxx && Q == 0`.

```
VCVT{<c>}{<q>}.<dt1>.<dt2> <Dd>, <Dm>, #<fbits>
```

128-bit SIMD vector variant

Applies when `imm6 != 000xxx && Q == 1`.

```
VCVT{<c>}{<q>}.<dt1>.<dt2> <Qd>, <Qm>, #<fbits>
```

Decode for all variants of this encoding

```
if imm6 == '000xxx' then SEE "Related encodings";
if op<1> == '0' && !HaveFP16Ext() then UNDEFINED;
if op<1> == '0' && imm6 == '10xxxx' then UNDEFINED;
if imm6 == '0xxxxx' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
to_fixed = (op<0> == '1');  frac_bits = 64 - UInt(imm6);
unsigned = (U == '1');
case op<1> of
     when '0' esize = 16; elements = 4;
     when '1' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13	12	11 10 9 8	7	6	5	0	15 12	11 10	9 8	7	6	5	4	3 0
1 1 1	U	1 1 1 1	1	D	imm6		Vd	1 1	op	0	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `imm6 != 000xxx && Q == 0`.

```
VCVT{<c>}{<q>}.<dt1>.<dt2> <Dd>, <Dm>, #<fbits>
```

128-bit SIMD vector variant

Applies when `imm6 != 000xxx && Q == 1`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VCVT{<c>}{<q>}.<dt1>.<dt2> <Qd>, <Qm>, #<fbits>
```

Decode for all variants of this encoding

```
if imm6 == '000xxx' then SEE "Related encodings";
if op<1> == '0' && !HaveFP16Ext() then UNDEFINED;
if op<1> == '0' && imm6 == '10xxxx' then UNDEFINED;
if imm6 == '0xxxxx' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
to_fixed = (op<0> == '1');  frac_bits = 64 - UInt(imm6);
unsigned = (U == '1');
case op<1> of
     when '0' esize = 16; elements = 4;
     when '1' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Notes for all encodings

Related encodings: See *Advanced SIMD one register and modified immediate* on page F3-4146 for the T32 instruction set, or *Advanced SIMD one register and modified immediate* on page F4-4233 for the A32 instruction set.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt1>` Is the data type for the elements of the destination vector, encoded in the "op:U" field. It can have the following values:

F16	when op = 00, U = x
S16	when op = 01, U = 0
U16	when op = 01, U = 1
F32	when op = 10, U = x
S32	when op = 11, U = 0
U32	when op = 11, U = 1

`<dt2>` Is the data type for the elements of the source vector, encoded in the "op:U" field. It can have the following values:

S16	when op = 00, U = 0
U16	when op = 00, U = 1
F16	when op = 01, U = x
S32	when op = 10, U = 0
U32	when op = 10, U = 1
F32	when op = 11, U = x

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<fbits> The number of fraction bits in the fixed point number, in the range 1 to 32 for 32-bit elements, or in the range 1 to 16 for 16-bit elements:

- (64 - <fbits>) is encoded in imm6.

An assembler can permit an <fbits> value of 0. This is encoded as floating-point to integer or integer to floating-point instruction, see VCVT (between floating-point and integer, Advanced SIMD).

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(esize) result;
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[m+r],e,esize];
            if to_fixed then
                result = FPToFixed(op1, frac_bits, unsigned, StandardFPSCRValue(),
                                   FPRounding_ZERO);
            else
                result = FixedToFP(op1, frac_bits, unsigned, StandardFPSCRValue(),
                                   FPRounding_TIEEVEN);
            Elem[D[d+r],e,esize] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.64 VCVT (between floating-point and fixed-point, floating-point)

Convert between floating-point and fixed-point converts a value in a register from floating-point to fixed-point, or from fixed-point to floating-point. Software can specify the fixed-point value as either signed or unsigned.

The fixed-point value can be 16-bit or 32-bit. Conversions from fixed-point values take their operand from the low-order bits of the source register and ignore any remaining bits. Signed conversions to fixed-point values sign-extend the result value to the destination register width. Unsigned conversions to fixed-point values zero-extend the result value to the destination register width.

The floating-point to fixed-point operation uses the Round towards Zero rounding mode. The fixed-point to floating-point operation uses the Round to Nearest rounding mode.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	1	1	1	0	1	D	1	1	1	op	1	U	Vd	1	0	sf		sx	1	i	0	imm4
cond																						

Half-precision scalar variant

Applies when `op == 0 && sf == 01`.

```
VCVT{<c>}{<q>}.F16.<dt> <Sdm>, <Sdm>, #<fbits>
```

Half-precision scalar variant

Applies when `op == 1 && sf == 01`.

```
VCVT{<c>}{<q>}.<dt>.F16 <Sdm>, <Sdm>, #<fbits>
```

Single-precision scalar variant

Applies when `op == 0 && sf == 10`.

```
VCVT{<c>}{<q>}.F32.<dt> <Sdm>, <Sdm>, #<fbits>
```

Single-precision scalar variant

Applies when `op == 1 && sf == 10`.

```
VCVT{<c>}{<q>}.<dt>.F32 <Sdm>, <Sdm>, #<fbits>
```

Double-precision scalar variant

Applies when `op == 0 && sf == 11`.

```
VCVT{<c>}{<q>}.F64.<dt> <Ddm>, <Ddm>, #<fbits>
```

Double-precision scalar variant

Applies when `op == 1 && sf == 11`.

```
VCVT{<c>}{<q>}.<dt>.F64 <Ddm>, <Ddm>, #<fbits>
```

Decode for all variants of this encoding

```
if sf == '00' || (sf == '01' && !HaveFP16Ext()) then UNDEFINED;
if sf == '01' && cond != '1110' then UNPREDICTABLE;
to_fixed = (op == '1');  unsigned = (U == '1');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
size = if sx == '0' then 16 else 32;
frac_bits = size - UInt(imm4:i);
case sf of
    when '01' fp_size = 16; d = UInt(Vd:D);
    when '10' fp_size = 32; d = UInt(Vd:D);
    when '11' fp_size = 64; d = UInt(D:Vd);

if frac_bits < 0 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `frac_bits < 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15 – 12	11	10	9	8	7	6	5	4	3 – 0
1	1	1	0	1	1	1	0	1	D	1	1	1	op	1	U	Vd	1	0	sf		sx	1	i	0	imm4

Half-precision scalar variant

Applies when `op == 0 && sf == 01`.

```
VCVT{<c>}{<q>}.F16.<dt> <Sdm>, <Sdm>, #<fbits>
```

Half-precision scalar variant

Applies when `op == 1 && sf == 01`.

```
VCVT{<c>}{<q>}.<dt>.F16 <Sdm>, <Sdm>, #<fbits>
```

Single-precision scalar variant

Applies when `op == 0 && sf == 10`.

```
VCVT{<c>}{<q>}.F32.<dt> <Sdm>, <Sdm>, #<fbits>
```

Single-precision scalar variant

Applies when `op == 1 && sf == 10`.

```
VCVT{<c>}{<q>}.<dt>.F32 <Sdm>, <Sdm>, #<fbits>
```

Double-precision scalar variant

Applies when `op == 0 && sf == 11`.

```
VCVT{<c>}{<q>}.F64.<dt> <Ddm>, <Ddm>, #<fbits>
```

Double-precision scalar variant

Applies when `op == 1 && sf == 11`.

```
VCVT{<c>}{<q>}.<dt>.F64 <Ddm>, <Ddm>, #<fbits>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if sf == '00' || (sf == '01' && !HaveFP16Ext()) then UNDEFINED;
if sf == '01' && InITBlock() then UNPREDICTABLE;
to_fixed = (op == '1');  unsigned = (U == '1');
size = if sx == '0' then 16 else 32;
frac_bits = size - UInt(imm4:i);
case sf of
    when '01' fp_size = 16; d = UInt(Vd:D);
    when '10' fp_size = 32; d = UInt(Vd:D);
    when '11' fp_size = 64; d = UInt(D:Vd);

if frac_bits < 0 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `frac_bits < 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *VCVT (between floating-point and fixed-point)* on page K1-7846.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the fixed-point number, encoded in the "U:sx" field. It can have the following values:

`S16`	when `U = 0`, `sx = 0`
`S32`	when `U = 0`, `sx = 1`
`U16`	when `U = 1`, `sx = 0`
`U32`	when `U = 1`, `sx = 1`

`<Sdm>` Is the 32-bit name of the SIMD&FP destination and source register, encoded in the "Vd:D" field.

`<Ddm>` Is the 64-bit name of the SIMD&FP destination and source register, encoded in the "D:Vd" field.

`<fbits>` The number of fraction bits in the fixed-point number:

- If `<dt>` is `S16` or `U16`, `<fbits>` must be in the range 0-16. (16 - `<fbits>`) is encoded in `[imm4, i]`
- If `<dt>` is `S32` or `U32`, `<fbits>` must be in the range 1-32. (32 - `<fbits>`) is encoded in `[imm4, i]`.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_fixed then
        bits(size) result;
        case fp_size of
            when 16
                result = FPToFixed(S[d]<15:0>, frac_bits, unsigned, FPSCR, FPRounding_ZERO);
                S[d] = Extend(result, 32, unsigned);
            when 32
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
                result = FPToFixed(S[d], frac_bits, unsigned, FPSCR, FPRounding_ZERO);
                S[d] = Extend(result, 32, unsigned);
            when 64
                result = FPToFixed(D[d], frac_bits, unsigned, FPSCR, FPRounding_ZERO);
                D[d] = Extend(result, 64, unsigned);
    else
        case fp_size of
            when 16
                bits(16) fp16 = FixedToFP(S[d]<size-1:0>, frac_bits, unsigned, FPSCR,
FPRounding_TIEEVEN);
                S[d] = Zeros(16):fp16;
            when 32
                S[d] = FixedToFP(S[d]<size-1:0>, frac_bits, unsigned, FPSCR, FPRounding_TIEEVEN);
            when 64
                D[d] = FixedToFP(D[d]<size-1:0>, frac_bits, unsigned, FPSCR, FPRounding_TIEEVEN);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.65 VCVTA (Advanced SIMD)

Vector Convert floating-point to integer with Round to Nearest with Ties to Away converts each element in a vector from floating-point to integer using the Round to Nearest with Ties to Away rounding mode, and places the results in a second vector.

The operand vector elements are floating-point numbers.

The result vector elements are integers, and the same size as the operand vector elements. Signed and unsigned integers are distinct.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21 20	19 18	17 16	15 12	11	10	9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 1 1	D		1 1	size	1 1	Vd	0	0	0 0	op	Q	M	0	Vm
										RM					

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCVTA{<q>}.<dt>.<dt2> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCVTA{<q>}.<dt>.<dt2> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '1');
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 2 1 0	15 12	11	10	9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 1 1	D		1	1	size	1 1	Vd	0	0	0 0	op	Q	M	0	Vm
											RM				

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCVTA{<q>}.<dt>.<dt2> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCVTA{<q>}.<dt>.<dt2> <Qd>, <Qm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '1');
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the destination, encoded in the "op:size" field. It can have the following values:

`S16`	when `op = 0`, `size = 01`
`S32`	when `op = 0`, `size = 10`
`U16`	when `op = 1`, `size = 01`
`U32`	when `op = 1`, `size = 10`

`<dt2>` Is the data type for the elements of the source vector, encoded in the "size" field. It can have the following values:

`F16`	when `size = 01`
`F32`	when `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
bits(esize) result;
for r = 0 to regs-1
    for e = 0 to elements-1
        Elem[D[d+r],e,esize] = FPToFixed(Elem[D[m+r],e,esize], 0, unsigned,
                                         StandardFPSCRValue(), rounding);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.66 VCVTA (floating-point)

Convert floating-point to integer with Round to Nearest with Ties to Away converts a value in a register from floating-point to a 32-bit integer using the Round to Nearest with Ties to Away rounding mode, and places the result in a second register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21 20	19	18 17	16	15 12	11 10	9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	1	D	1 1 1	1	0 0		Vd	1 0	!=00	op	1	M	0	Vm
						RM				size					

Half-precision scalar variant

Applies when `size == 01`.

```
VCVTA{<q>}.<dt>.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VCVTA{<q>}.<dt>.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VCVTA{<q>}.<dt>.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '0');
d = UInt(Vd:D);
case size of
    when '01' esize = 16; m = UInt(Vm:M);
    when '10' esize = 32; m = UInt(Vm:M);
    when '11' esize = 64; m = UInt(M:Vm);
```

T1

15 14 13 12	11 10 9 8	7	6 5 4	3	2 1 0	15 12	11 10	9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	1	D	1 1 1	1	0 0	Vd	1 0	!=00	op	1	M	0	Vm
						RM			size					

Half-precision scalar variant

Applies when `size == 01`.

```
VCVTA{<q>}.<dt>.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VCVTA{<q>}.<dt>.F32 <Sd>, <Sm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Double-precision scalar variant

Applies when `size == 11`.

```
VCVTA{<q>}.<dt>.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '0');
d = UInt(Vd:D);
case size of
    when '01' esize = 16; m = UInt(Vm:M);
    when '10' esize = 32; m = UInt(Vm:M);
    when '11' esize = 64; m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the destination, encoded in the "op" field. It can have the following values:

`U32`	when `op = 0`
`S32`	when `op = 1`

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

`<Sm>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
    when 16
        S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR, rounding);
    when 32
        S[d] = FPToFixed(S[m], 0, unsigned, FPSCR, rounding);
    when 64
        S[d] = FPToFixed(D[m], 0, unsigned, FPSCR, rounding);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.67 VCVTB

Convert to or from a half-precision value in the bottom half of a single-precision register does one of the following:

- Converts the half-precision value in the bottom half of a single-precision register to single-precision and writes the result to a single-precision register.
- Converts the half-precision value in the bottom half of a single-precision register to double-precision and writes the result to a double-precision register.
- Converts the single-precision value in a single-precision register to half-precision and writes the result into the bottom half of a single-precision register, preserving the other half of the destination register.
- Converts the double-precision value in a double-precision register to half-precision and writes the result into the bottom half of a single-precision register, preserving the other half of the destination register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	1	1	1	0	1	D	1	1	0	0	1	op	Vd	1	0	1	sz	0	1	M	0	Vm
cond																		T				

Half-precision to single-precision variant

Applies when `op == 0 && sz == 0`.

```
VCVTB{<c>}{<q>}.F32.F16 <Sd>, <Sm>
```

Half-precision to double-precision variant

Applies when `op == 0 && sz == 1`.

```
VCVTB{<c>}{<q>}.F64.F16 <Dd>, <Sm>
```

Single-precision to half-precision variant

Applies when `op == 1 && sz == 0`.

```
VCVTB{<c>}{<q>}.F16.F32 <Sd>, <Sm>
```

Double-precision to half-precision variant

Applies when `op == 1 && sz == 1`.

```
VCVTB{<c>}{<q>}.F16.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
uses_double = (sz == '1'); convert_from_half = (op == '0');
lowbit = (if T == '1' then 16 else 0);
if uses_double then
    if convert_from_half then
        d = UInt(D:Vd); m = UInt(Vm:M);
    else
        d = UInt(Vd:D); m = UInt(M:Vm);
else
    d = UInt(Vd:D); m = UInt(Vm:M);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	0	1	1	1	0	1	D	1	1	0	0	1	op	Vd	1	0	1	sz	0	1	M	0	Vm
																					T				

Half-precision to single-precision variant

Applies when `op == 0 && sz == 0`.

```
VCVTB{<c>}{<q>}.F32.F16 <Sd>, <Sm>
```

Half-precision to double-precision variant

Applies when `op == 0 && sz == 1`.

```
VCVTB{<c>}{<q>}.F64.F16 <Dd>, <Sm>
```

Single-precision to half-precision variant

Applies when `op == 1 && sz == 0`.

```
VCVTB{<c>}{<q>}.F16.F32 <Sd>, <Sm>
```

Double-precision to half-precision variant

Applies when `op == 1 && sz == 1`.

```
VCVTB{<c>}{<q>}.F16.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
uses_double = (sz == '1'); convert_from_half = (op == '0');
lowbit = (if T == '1' then 16 else 0);
if uses_double then
    if convert_from_half then
        d = UInt(D:Vd); m = UInt(Vm:M);
    else
        d = UInt(Vd:D); m = UInt(M:Vm);
else
    d = UInt(Vd:D); m = UInt(Vm:M);
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Sd>`	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
`<Dm>`	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
`<Sm>`	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    bits(16) hp;
    if convert_from_half then
        hp = S[m]<lowbit+15:lowbit>;
        if uses_double then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        D[d] = FPConvert(hp, FPSCR);
    else
        S[d] = FPConvert(hp, FPSCR);
else
    if uses_double then
        hp = FPConvert(D[m], FPSCR);
    else
        hp = FPConvert(S[m], FPSCR);
    S[d]<lowbit+15:lowbit> = hp;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.68 VCVTB (BFloat16)

Converts the single-precision value in a single-precision register to BFloat16 format and writes the result into the bottom half of a single precision register, preserving the top 16 bits of the destination register.

Unlike the BFloat16 multiplication instructions, this instruction honors all the control bits in the FPSCR that apply to single-precision arithmetic, including the rounding mode. This instruction can generate a floating-point exception which causes a cumulative exception bit in the FPSCR to be set, or a synchronous exception to be taken, depending on the enable bits in the FPSCR.

A1

ARMv8.6

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111 (cond)	1	1	1	0	1	D	1	1	0	0	1	1	Vd	1	0	0	1	0	1	M	0	Vm

A1 variant

```
VCVTB{<c>}{<q>}.BF16.F32 <Sd>, <Sm>
```

Decode for this encoding

```
if !HaveAArch32BF16Ext() then UNDEFINED;
integer d = UInt(Vd:D);
integer m = UInt(Vm:M);
```

T1

ARMv8.6

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	0	1	1	1	0	1	D	1	1	0	0	1	1	Vd	1	0	0	1	0	1	M	0	Vm

T1 variant

```
VCVTB{<c>}{<q>}.BF16.F32 <Sd>, <Sm>
```

Decode for this encoding

```
if !HaveAArch32BF16Ext() then UNDEFINED;
integer d = UInt(Vd:D);
integer m = UInt(Vm:M);
```

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);

    S[d]<15:0> = FPConvertBF(S[m], FPSCR);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.69 VCVTM (Advanced SIMD)

Vector Convert floating-point to integer with Round towards -Infinity converts each element in a vector from floating-point to integer using the Round towards -Infinity rounding mode, and places the results in a second vector.

The operand vector elements are floating-point numbers.

The result vector elements are integers, and the same size as the operand vector elements. Signed and unsigned integers are distinct.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 18	17 16	15 12	11	10	9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	1	1	size	1	1	Vd	0	0	1 1	op	Q	M	0	Vm
											RM					

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCVTM{<q>}.<dt>.<dt2> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCVTM{<q>}.<dt>.<dt2> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '1');
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 2	1	0	15 12	11	10	9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	1	D	1	1	size	1	1	Vd	0	0	1 1	op	Q	M	0	Vm
												RM					

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCVTM{<q>}.<dt>.<dt2> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCVTM{<q>}.<dt>.<dt2> <Qd>, <Qm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '1');
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the destination, encoded in the "op:size" field. It can have the following values:

`S16`	when `op = 0`, `size = 01`
`S32`	when `op = 0`, `size = 10`
`U16`	when `op = 1`, `size = 01`
`U32`	when `op = 1`, `size = 10`

`<dt2>` Is the data type for the elements of the source vector, encoded in the "size" field. It can have the following values:

`F16`	when `size = 01`
`F32`	when `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
bits(esize) result;
for r = 0 to regs-1
    for e = 0 to elements-1
        Elem[D[d+r],e,esize] = FPToFixed(Elem[D[m+r],e,esize], 0, unsigned,
                                         StandardFPSCRValue(), rounding);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.70 VCVTM (floating-point)

Convert floating-point to integer with Round towards -Infinity converts a value in a register from floating-point to a 32-bit integer using the Round towards -Infinity rounding mode, and places the result in a second register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	1	0	1	D	1	1	1	1	1	1	Vd	1	0	!=00		op	1	M	0	Vm
														RM					size						

Half-precision scalar variant

Applies when `size == 01`.

```
VCVTM{<q>}.<dt>.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VCVTM{<q>}.<dt>.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VCVTM{<q>}.<dt>.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '0');
d = UInt(Vd:D);
case size of
    when '01' esize = 16; m = UInt(Vm:M);
    when '10' esize = 32; m = UInt(Vm:M);
    when '11' esize = 64; m = UInt(M:Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	1	0	1	D	1	1	1	1	1	1	Vd	1	0	!=00		op	1	M	0	Vm
														RM					size						

Half-precision scalar variant

Applies when `size == 01`.

```
VCVTM{<q>}.<dt>.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VCVTM{<q>}.<dt>.F32 <Sd>, <Sm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Double-precision scalar variant

Applies when `size == 11`.

```
VCVTM{<q>}.<dt>.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '0');
d = UInt(Vd:D);
case size of
    when '01' esize = 16; m = UInt(Vm:M);
    when '10' esize = 32; m = UInt(Vm:M);
    when '11' esize = 64; m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the destination, encoded in the "op" field. It can have the following values:

`U32`	when `op = 0`
`S32`	when `op = 1`

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

`<Sm>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
    when 16
        S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR, rounding);
    when 32
        S[d] = FPToFixed(S[m], 0, unsigned, FPSCR, rounding);
    when 64
        S[d] = FPToFixed(D[m], 0, unsigned, FPSCR, rounding);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.71 VCVTN (Advanced SIMD)

Vector Convert floating-point to integer with Round to Nearest converts each element in a vector from floating-point to integer using the Round to Nearest rounding mode, and places the results in a second vector.

The operand vector elements are floating-point numbers.

The result vector elements are integers, and the same size as the operand vector elements. Signed and unsigned integers are distinct.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21 20	19 18	17 16	15 12	11	10	9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	1 1	size	1 1	Vd	0	0	0 1	op	Q	M	0	Vm
										RM					

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCVTN{<q>}.<dt>.<dt2> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCVTN{<q>}.<dt>.<dt2> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '1');
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5 4	3 2	1 0	15 12	11	10	9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	1	D	1 1	size	1 1	Vd	0	0	0 1	op	Q	M	0	Vm
										RM					

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCVTN{<q>}.<dt>.<dt2> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCVTN{<q>}.<dt>.<dt2> <Qd>, <Qm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '1');
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the destination, encoded in the "op:size" field. It can have the following values:

`S16`	when `op = 0`, `size = 01`
`S32`	when `op = 0`, `size = 10`
`U16`	when `op = 1`, `size = 01`
`U32`	when `op = 1`, `size = 10`

`<dt2>` Is the data type for the elements of the source vector, encoded in the "size" field. It can have the following values:

`F16`	when `size = 01`
`F32`	when `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
bits(esize) result;
for r = 0 to regs-1
    for e = 0 to elements-1
        Elem[D[d+r],e,esize] = FPToFixed(Elem[D[m+r],e,esize], 0, unsigned,
                                         StandardFPSCRValue(), rounding);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.72 VCVTN (floating-point)

Convert floating-point to integer with Round to Nearest converts a value in a register from floating-point to a 32-bit integer using the Round to Nearest rounding mode, and places the result in a second register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21 20	19	18	17 16	15 12	11 10	9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	1	D	1 1 1	1	1	0 1	Vd	1 0	!=00	op	1	M	0	Vm
							RM			size					

Half-precision scalar variant

Applies when `size == 01`.

```
VCVTN{<q>}.<dt>.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VCVTN{<q>}.<dt>.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VCVTN{<q>}.<dt>.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '0');
d = UInt(Vd:D);
case size of
    when '01' esize = 16; m = UInt(Vm:M);
    when '10' esize = 32; m = UInt(Vm:M);
    when '11' esize = 64; m = UInt(M:Vm);
```

T1

15 14 13 12	11 10 9 8	7	6	5 4	3	2	1 0	15 12	11 10	9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	1	D	1 1 1	1	1	0 1	Vd	1 0	!=00	op	1	M	0	Vm
							RM			size					

Half-precision scalar variant

Applies when `size == 01`.

```
VCVTN{<q>}.<dt>.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VCVTN{<q>}.<dt>.F32 <Sd>, <Sm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Double-precision scalar variant

Applies when `size == 11`.

```
VCVTN{<q>}.<dt>.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '0');
d = UInt(Vd:D);
case size of
    when '01' esize = 16; m = UInt(Vm:M);
    when '10' esize = 32; m = UInt(Vm:M);
    when '11' esize = 64; m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the destination, encoded in the "op" field. It can have the following values:

`U32`	when `op = 0`
`S32`	when `op = 1`

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

`<Sm>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
    when 16
        S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR, rounding);
    when 32
        S[d] = FPToFixed(S[m], 0, unsigned, FPSCR, rounding);
    when 64
        S[d] = FPToFixed(D[m], 0, unsigned, FPSCR, rounding);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.73 VCVTP (Advanced SIMD)

Vector Convert floating-point to integer with Round towards +Infinity converts each element in a vector from floating-point to integer using the Round towards +Infinity rounding mode, and places the results in a second vector.

The operand vector elements are floating-point numbers.

The result vector elements are integers, and the same size as the operand vector elements. Signed and unsigned integers are distinct.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 18	17 16	15 12	11	10	9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	1	1	size	1 1	Vd	0	0	1 0	op	Q	M	0	Vm
											RM					

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCVTP{<q>}.<dt>.<dt2> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCVTP{<q>}.<dt>.<dt2> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '1');
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 2	1 0	15 12	11	10	9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	1	D	1	1	size	1 1	Vd	0	0	1 0	op	Q	M	0	Vm
											RM					

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VCVTP{<q>}.<dt>.<dt2> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VCVTP{<q>}.<dt>.<dt2> <Qd>, <Qm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '1');
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the destination, encoded in the "op:size" field. It can have the following values:

`S16`	when `op = 0`, `size = 01`
`S32`	when `op = 0`, `size = 10`
`U16`	when `op = 1`, `size = 01`
`U32`	when `op = 1`, `size = 10`

`<dt2>` Is the data type for the elements of the source vector, encoded in the "size" field. It can have the following values:

`F16`	when `size = 01`
`F32`	when `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
bits(esize) result;
for r = 0 to regs-1
    for e = 0 to elements-1
        Elem[D[d+r],e,esize] = FPToFixed(Elem[D[m+r],e,esize], 0, unsigned,
                                         StandardFPSCRValue(), rounding);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.74 VCVTP (floating-point)

Convert floating-point to integer with Round towards +Infinity converts a value in a register from floating-point to a 32-bit integer using the Round towards +Infinity rounding mode, and places the result in a second register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	1	0	1	D	1	1	1	1	1	0	Vd	1	0	!=00		op	1	M	0	Vm
														RM					size						

Half-precision scalar variant

Applies when `size == 01`.

```
VCVTP{<q>}.<dt>.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VCVTP{<q>}.<dt>.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VCVTP{<q>}.<dt>.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '0');
d = UInt(Vd:D);
case size of
    when '01' esize = 16; m = UInt(Vm:M);
    when '10' esize = 32; m = UInt(Vm:M);
    when '11' esize = 64; m = UInt(M:Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	1	0	1	D	1	1	1	1	1	0	Vd	1	0	!=00		op	1	M	0	Vm
														RM					size						

Half-precision scalar variant

Applies when `size == 01`.

```
VCVTP{<q>}.<dt>.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VCVTP{<q>}.<dt>.F32 <Sd>, <Sm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Double-precision scalar variant

Applies when `size == 11`.

```
VCVTP{<q>}.<dt>.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '0');
d = UInt(Vd:D);
case size of
    when '01' esize = 16; m = UInt(Vm:M);
    when '10' esize = 32; m = UInt(Vm:M);
    when '11' esize = 64; m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the destination, encoded in the "op" field. It can have the following values:

`U32`	when `op = 0`
`S32`	when `op = 1`

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

`<Sm>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
    when 16
        S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR, rounding);
    when 32
        S[d] = FPToFixed(S[m], 0, unsigned, FPSCR, rounding);
    when 64
        S[d] = FPToFixed(D[m], 0, unsigned, FPSCR, rounding);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.75 VCVTR

Convert floating-point to integer converts a value in a register from floating-point to a 32-bit integer, using the rounding mode specified by the FPSCR and places the result in a second register.

VCVT (between floating-point and fixed-point, floating-point) describes conversions between floating-point and 16-bit integers.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31–28	27 26 25 24	23	22	21 20	19	18–16	15–12	11 10	9 8	7	6	5	4	3–0
!=1111	1 1 1 0	1	D	1 1	1	1 0 x	Vd	1 0	size	0	1	M	0	Vm
cond						opc2				op				

Half-precision scalar variant

Applies when `opc2 == 100 && size == 01`.

```
VCVTR{<c>}{<q>}.U32.F16 <Sd>, <Sm>
```

Half-precision scalar variant

Applies when `opc2 == 101 && size == 01`.

```
VCVTR{<c>}{<q>}.S32.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `opc2 == 100 && size == 10`.

```
VCVTR{<c>}{<q>}.U32.F32 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `opc2 == 101 && size == 10`.

```
VCVTR{<c>}{<q>}.S32.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `opc2 == 100 && size == 11`.

```
VCVTR{<c>}{<q>}.U32.F64 <Sd>, <Dm>
```

Double-precision scalar variant

Applies when `opc2 == 101 && size == 11`.

```
VCVTR{<c>}{<q>}.S32.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
if opc2 != '000' && opc2 != '10x' then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
to_integer = (opc2<2> == '1');
if to_integer then
    unsigned = (opc2<0> == '0');
    rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR);
    d = UInt(Vd:D);
    case size of
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        when '01' esize = 16; m = UInt(Vm:M);
        when '10' esize = 32; m = UInt(Vm:M);
        when '11' esize = 64; m = UInt(M:Vm);
else
    unsigned = (op == '0');
    rounding = FPRoundingMode(FPSCR);
    m = UInt(Vm:M);
    case size of
        when '01' esize = 16; d = UInt(Vd:D);
        when '10' esize = 32; d = UInt(Vd:D);
        when '11' esize = 64; d = UInt(D:Vd);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	0	1	1	1	0	1	D	1	1	1	1	0 x	Vd		1	0	size		0	1	M	0	Vm	
													opc2								op					

Half-precision scalar variant

Applies when `opc2 == 100 && size == 01`.

```
VCVTR{<c>}{<q>}.U32.F16 <Sd>, <Sm>
```

Half-precision scalar variant

Applies when `opc2 == 101 && size == 01`.

```
VCVTR{<c>}{<q>}.S32.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `opc2 == 100 && size == 10`.

```
VCVTR{<c>}{<q>}.U32.F32 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `opc2 == 101 && size == 10`.

```
VCVTR{<c>}{<q>}.S32.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `opc2 == 100 && size == 11`.

```
VCVTR{<c>}{<q>}.U32.F64 <Sd>, <Dm>
```

Double-precision scalar variant

Applies when `opc2 == 101 && size == 11`.

```
VCVTR{<c>}{<q>}.S32.F64 <Sd>, <Dm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if opc2 != '000' && opc2 != '10x' then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
to_integer = (opc2<2> == '1');
if to_integer then
    unsigned = (opc2<0> == '0');
    rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR);
    d = UInt(Vd:D);
    case size of
        when '01' esize = 16; m = UInt(Vm:M);
        when '10' esize = 32; m = UInt(Vm:M);
        when '11' esize = 64; m = UInt(M:Vm);
else
    unsigned = (op == '0');
    rounding = FPRoundingMode(FPSCR);
    m = UInt(Vm:M);
    case size of
        when '01' esize = 16; d = UInt(Vd:D);
        when '10' esize = 32; d = UInt(Vd:D);
        when '11' esize = 64; d = UInt(D:Vd);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings

Related encodings: See *Floating-point data-processing* on page F3-4131 for the T32 instruction set, or *Floating-point data-processing* on page F4-4214 for the A32 instruction set.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Sd>`	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
`<Sm>`	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
`<Dm>`	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_integer then
        case esize of
            when 16
                S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR, rounding);
            when 32
                S[d] = FPToFixed(S[m], 0, unsigned, FPSCR, rounding);
            when 64
                S[d] = FPToFixed(D[m], 0, unsigned, FPSCR, rounding);
    else
        case esize of
            when 16
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    bits(16) fp16 = FixedToFP(S[m], 0, unsigned, FPSCR, rounding);
    S[d] = Zeros(16):fp16;
when 32
    S[d] = FixedToFP(S[m], 0, unsigned, FPSCR, rounding);
when 64
    D[d] = FixedToFP(S[m], 0, unsigned, FPSCR, rounding);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.76 VCVTT

Convert to or from a half-precision value in the top half of a single-precision register does one of the following:

- Converts the half-precision value in the top half of a single-precision register to single-precision and writes the result to a single-precision register.
- Converts the half-precision value in the top half of a single-precision register to double-precision and writes the result to a double-precision register.
- Converts the single-precision value in a single-precision register to half-precision and writes the result into the top half of a single-precision register, preserving the other half of the destination register.
- Converts the double-precision value in a double-precision register to half-precision and writes the result into the top half of a single-precision register, preserving the other half of the destination register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	1	1	1	0	1	D	1	1	0	0	1	op	Vd	1	0	1	sz	1	1	M	0	Vm
cond																		T				

Half-precision to single-precision variant

Applies when `op == 0 && sz == 0`.

```
VCVTT{<c>}{<q>}.F32.F16 <Sd>, <Sm>
```

Half-precision to double-precision variant

Applies when `op == 0 && sz == 1`.

```
VCVTT{<c>}{<q>}.F64.F16 <Dd>, <Sm>
```

Single-precision to half-precision variant

Applies when `op == 1 && sz == 0`.

```
VCVTT{<c>}{<q>}.F16.F32 <Sd>, <Sm>
```

Double-precision to half-precision variant

Applies when `op == 1 && sz == 1`.

```
VCVTT{<c>}{<q>}.F16.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
uses_double = (sz == '1'); convert_from_half = (op == '0');
lowbit = (if T == '1' then 16 else 0);
if uses_double then
    if convert_from_half then
        d = UInt(D:Vd); m = UInt(Vm:M);
    else
        d = UInt(Vd:D); m = UInt(M:Vm);
else
    d = UInt(Vd:D); m = UInt(Vm:M);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	0	1	1	1	0	1	D	1	1	0	0	1	op	Vd	1	0	1	sz	1	1	M	0	Vm
																					T				

Half-precision to single-precision variant

Applies when `op == 0 && sz == 0`.

```
VCVTT{<c>}{<q>}.F32.F16 <Sd>, <Sm>
```

Half-precision to double-precision variant

Applies when `op == 0 && sz == 1`.

```
VCVTT{<c>}{<q>}.F64.F16 <Dd>, <Sm>
```

Single-precision to half-precision variant

Applies when `op == 1 && sz == 0`.

```
VCVTT{<c>}{<q>}.F16.F32 <Sd>, <Sm>
```

Double-precision to half-precision variant

Applies when `op == 1 && sz == 1`.

```
VCVTT{<c>}{<q>}.F16.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
uses_double = (sz == '1'); convert_from_half = (op == '0');
lowbit = (if T == '1' then 16 else 0);
if uses_double then
    if convert_from_half then
        d = UInt(D:Vd); m = UInt(Vm:M);
    else
        d = UInt(Vd:D); m = UInt(M:Vm);
else
    d = UInt(Vd:D); m = UInt(Vm:M);
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Sd>`	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
`<Dm>`	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
`<Sm>`	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    bits(16) hp;
    if convert_from_half then
        hp = S[m]<lowbit+15:lowbit>;
        if uses_double then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            D[d] = FPConvert(hp, FPSCR);
        else
            S[d] = FPConvert(hp, FPSCR);
    else
        if uses_double then
            hp = FPConvert(D[m], FPSCR);
        else
            hp = FPConvert(S[m], FPSCR);
        S[d]<lowbit+15:lowbit> = hp;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.77 VCVTT (BFloat16)

Converts the single-precision value in a single-precision register to BFloat16 format and writes the result in the top half of a single-precision register, preserving the bottom 16 bits of the register.

Unlike the BFloat16 multiplication instructions, this instruction honors all the control bits in the FPSCR that apply to single-precision arithmetic, including the rounding mode. This instruction can generate a floating-point exception which causes a cumulative exception bit in the FPSCR to be set, or a synchronous exception to be taken, depending on the enable bits in the FPSCR.

A1

ARMv8.6

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111 (cond)	1	1	1	0	1	D	1	1	0	0	1	1	Vd	1	0	0	1	1	1	M	0	Vm

A1 variant

```
VCVTT{<c>}{<q>}.BF16.F32 <Sd>, <Sm>
```

Decode for this encoding

```
if !HaveAArch32BF16Ext() then UNDEFINED;
integer d = UInt(Vd:D);
integer m = UInt(Vm:M);
```

T1

ARMv8.6

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	0	1	1	1	0	1	D	1	1	0	0	1	1	Vd	1	0	0	1	1	1	M	0	Vm

T1 variant

```
VCVTT{<c>}{<q>}.BF16.F32 <Sd>, <Sm>
```

Decode for this encoding

```
if !HaveAArch32BF16Ext() then UNDEFINED;
integer d = UInt(Vd:D);
integer m = UInt(Vm:M);
```

Assembler symbols

<c>	See *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);

    S[d]<31:16> = FPConvertBF(S[m], FPSCR);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.78 VDIV

Divide divides one floating-point value by another floating-point value and writes the result to a third floating-point register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9–8	7	6	5	4	3–0
!=1111	1	1	1	0	1	D	0	0	Vn	Vd	1	0	size	N	0	M	0	Vm
cond																		

Half-precision scalar variant

Applies when `size == 01`.

```
VDIV{<c>}{<q>}.F16 {<Sd>,} <Sn>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VDIV{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VDIV{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9–8	7	6	5	4	3–0
1	1	1	0	1	1	1	0	1	D	0	0	Vn	Vd	1	0	size	N	0	M	0	Vm

Half-precision scalar variant

Applies when `size == 01`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VDIV{<c>}{<q>}.F16 {<Sd>,} <Sn>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VDIV{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VDIV{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if size == '01' && InITBlock()  then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Sd>`	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
`<Sn>`	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
`<Sm>`	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
`<Dn>`	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
`<Dm>`	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    case esize of
        when 16
            S[d] = Zeros(16) : FPDiv(S[n]<15:0>, S[m]<15:0>, FPSCR);
        when 32
            S[d] = FPDiv(S[n], S[m], FPSCR);
        when 64
            D[d] = FPDiv(D[n], D[m], FPSCR);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.79 VDOT (vector)

BFloat16 floating-point (BF16) dot product (vector). This instruction delimits the source vectors into pairs of 16-bit BF16 elements. Within each pair, the elements in the first source vector are multiplied by the corresponding elements in the second source vector. The resulting single-precision products are then summed and added destructively to the single-precision element in the destination vector which aligns with the pair of BF16 values in the first source vector. The instruction does not update the FPSCR exception status.

A1

ARMv8.6

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	0	0	0	D	0	0	Vn	Vd	1	1	0	1	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VDOT{<q>}.BF16 <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VDOT{<q>}.BF16 <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if !HaveAArch32BF16Ext() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer regs = if Q == '1' then 2 else 1;
```

T1

ARMv8.6

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	0	0	0	D	0	0	Vn	Vd	1	1	0	1	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VDOT{<q>}.BF16 <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VDOT{<q>}.BF16 <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32BF16Ext() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer regs = if Q == '1' then 2 else 1;
```

Assembler symbols

<q>	See *Standard assembler syntax fields* on page F2-4080.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
bits(64) operand1;
bits(64) operand2;
bits(64) result;

CheckAdvSIMDEnabled();

for r = 0 to regs-1
    operand1 = Din[n+r];
    operand2 = Din[m+r];
    result = Din[d+r];
    for e = 0 to 1
        bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
        bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
        bits(16) elt2_a = Elem[operand2, 2 * e + 0, 16];
        bits(16) elt2_b = Elem[operand2, 2 * e + 1, 16];
        bits(32) sum = BFAdd(BFMul(elt1_a, elt2_a), BFMul(elt1_b, elt2_b));
        Elem[result, e, 32] = BFAdd(Elem[result, e, 32], sum);
    D[d+r] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.80 VDOT (by element)

BFloat16 floating-point indexed dot product (vector, by element). This instruction delimits the source vectors into pairs of 16-bit BF16 elements. Each pair of elements in the first source vector is multiplied by the indexed pair of elements in the second source vector. The resulting single-precision products are then summed and added destructively to the single-precision element in the destination vector which aligns with the pair of BFloat16 values in the first source vector. The instruction does not update the FPSCR exception status.

A1

ARMv8.6

31 30 29 28	27 26 25 24	23	22	21 20	19 16	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	0	D	0 0	Vn	Vd	1	1	0	1	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VDOT{<q>}.BF16 <Dd>, <Dn>, <Dm>[<index>]
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VDOT{<q>}.BF16 <Qd>, <Qn>, <Dm>[<index>]
```

Decode for all variants of this encoding

```
if !HaveAArch32BF16Ext() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm);
integer i = UInt(M);
integer regs = if Q == '1' then 2 else 1;
```

T1

ARMv8.6

15 14 13 12	11 10 9 8	7	6	5 4	3 0	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	0	D	0 0	Vn	Vd	1	1	0	1	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VDOT{<q>}.BF16 <Dd>, <Dn>, <Dm>[<index>]
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VDOT{<q>}.BF16 <Qd>, <Qn>, <Dm>[<index>]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32BF16Ext() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm);
integer i = UInt(M);
integer regs = if Q == '1' then 2 else 1;
```

Assembler symbols

<q>	See *Standard assembler syntax fields* on page F2-4080.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm" field.
<index>	Is the element index in the range 0 to 1, encoded in the "M" field.

Operation for all encodings

```
bits(64) operand1;
bits(64) operand2;
bits(64) result;

CheckAdvSIMDEnabled();

operand2 = Din[m];
for r = 0 to regs-1
    operand1 = Din[n+r];
    result = Din[d+r];
    for e = 0 to 1
        bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
        bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
        bits(16) elt2_a = Elem[operand2, 2 * i + 0, 16];
        bits(16) elt2_b = Elem[operand2, 2 * i + 1, 16];
        bits(32) sum = BFAdd(BFMul(elt1_a, elt2_a), BFMul(elt1_b, elt2_b));
        Elem[result, e, 32] = BFAdd(Elem[result, e, 32], sum);
    D[d+r] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.81 VDUP (general-purpose register)

Duplicate general-purpose register to vector duplicates an element from a general-purpose register into every element of the destination vector.

The destination vector elements can be 8-bit, 16-bit, or 32-bit fields. The source element is the least significant 8, 16, or 32 bits of the general-purpose register. There is no distinction between data types.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31–28	27 26 25 24	23	22	21	20	19–16	15–12	11 10 9 8	7	6	5	4	3	2	1	0
!=1111	1 1 1 0	1	B	Q	0	Vd	Rt	1 0 1 1	D	0	E	1	(0)	(0)	(0)	(0)
cond																

A1 variant

```
VDUP{<c>}{<q>}.<size> <Qd>, <Rt> // Encoded as Q = 1
VDUP{<c>}{<q>}.<size> <Dd>, <Rt> // Encoded as Q = 0
```

Decode for this encoding

```
if Q == '1' && Vd<0> == '1' then UNDEFINED;
d = UInt(D:Vd);  t = UInt(Rt);  regs = if Q == '0' then 1 else 2;
case B:E of
    when '00'  esize = 32;  elements = 2;
    when '01'  esize = 16;  elements = 4;
    when '10'  esize = 8;   elements = 8;
    when '11'  UNDEFINED;
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3–0	15–12	11 10 9 8	7	6	5	4	3	2	1	0
1 1 1 0	1 1 1 0	1	B	Q	0	Vd	Rt	1 0 1 1	D	0	E	1	(0)	(0)	(0)	(0)

T1 variant

```
VDUP{<c>}{<q>}.<size> <Qd>, <Rt> // Encoded as Q = 1
VDUP{<c>}{<q>}.<size> <Dd>, <Rt> // Encoded as Q = 0
```

Decode for this encoding

```
if Q == '1' && Vd<0> == '1' then UNDEFINED;
d = UInt(D:Vd);  t = UInt(Rt);  regs = if Q == '0' then 1 else 2;
case B:E of
    when '00'  esize = 32;  elements = 2;
    when '01'  esize = 16;  elements = 4;
    when '10'  esize = 8;   elements = 8;
    when '11'  UNDEFINED;
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080. Arm strongly recommends that any `VDUP` instruction is unconditional, see *Conditional execution* on page F2-4081.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<size>` The data size for the elements of the destination vector. It must be one of:

- `8` Encoded as [b, e] = `0b10`.
- `16` Encoded as [b, e] = `0b01`.
- `32` Encoded as [b, e] = `0b00`.

`<Qd>` The destination vector for a quadword operation.

`<Dd>` The destination vector for a doubleword operation.

`<Rt>` The Arm source register.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    scalar = R[t]<esize-1:0>;
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = scalar;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.82 VDUP (scalar)

Duplicate vector element to vector duplicates a single element of a vector into every element of the destination vector.

The scalar, and the destination vector elements, can be any one of 8-bit, 16-bit, or 32-bit fields. There is no distinction between data types.

For more information about scalars see *Advanced SIMD scalars* on page F2-4106.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23 22 21 20	19 16	15 12	11 10	9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 1 1	D 1 1	imm4	Vd	1 1	0 0	0	Q	M	0	Vm

Encoding

Applies when `Q == 0`.

```
VDUP{<c>}{<q>}.<size> <Dd>, <Dm[x]>
```

Encoding

Applies when `Q == 1`.

```
VDUP{<c>}{<q>}.<size> <Qd>, <Dm[x]>
```

Decode for all variants of this encoding

```
if imm4 == 'x000' then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;
case imm4 of
    when 'xxx1'  esize = 8;  elements = 8;  index = UInt(imm4<3:1>);
    when 'xx10'  esize = 16;  elements = 4;  index = UInt(imm4<3:2>);
    when 'x100'  esize = 32;  elements = 2;  index = UInt(imm4<3>);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 10	9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	1 D 1 1	imm4	Vd	1 1	0 0	0	Q	M	0	Vm

Encoding

Applies when `Q == 0`.

```
VDUP{<c>}{<q>}.<size> <Dd>, <Dm[x]>
```

Encoding

Applies when `Q == 1`.

```
VDUP{<c>}{<q>}.<size> <Qd>, <Dm[x]>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if imm4 == 'x000' then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;
case imm4 of
    when 'xxx1'  esize = 8;  elements = 8;  index = UInt(imm4<3:1>);
    when 'xx10'  esize = 16;  elements = 4;  index = UInt(imm4<3:2>);
    when 'x100'  esize = 32;  elements = 2;  index = UInt(imm4<3>);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<size>` The data size. It must be one of:

- `8` Encoded as imm4<0> = '1'. imm4<3:1> encodes the index`[x]` of the scalar.
- `16` Encoded as imm4<1:0> = '10'. imm4<3:2> encodes the index `[x]` of the scalar.
- `32` Encoded as imm4<2:0> = '100'. imm4<3> encodes the index `[x]` of the scalar.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm[x]>` The scalar. For details of how `[x]` is encoded, see the description of `<size>`.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    scalar = Elem[D[m],index,esize];
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = scalar;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.83 VEOR

Vector Bitwise Exclusive OR performs a bitwise Exclusive OR operation between two registers, and places the result in the destination register. The operand and result registers can be quadword or doubleword. They must all be the same size.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 0 1	1	0	D	0	0	Vn	Vd	0 0 0 1	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VEOR{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VEOR{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	0	D	0	0	Vn	Vd	0 0 0 1	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VEOR{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VEOR{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` An optional data type. It is ignored by assemblers, and does not affect the encoding.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[n+r] EOR D[m+r];
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

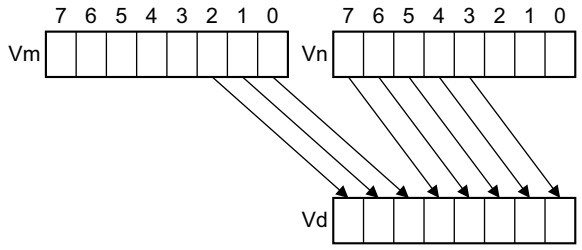
Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.84 VEXT (byte elements)

Vector Extract extracts elements from the bottom end of the second operand vector and the top end of the first, concatenates them and places the result in the destination vector.

The elements of the vectors are treated as being 8-bit fields. There is no distinction between data types.

The following figure shows the operation of VEXT doubleword operation for imm = 3.



Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

This instruction is used by the pseudo-instruction VEXT (multibyte elements). The pseudo-instruction is never the preferred disassembly.

A1

31 30 29 28	27 26 25 24	23	22	21 20	19 16	15 12	11 8	7	6	5	4	3 0
1 1 1 1	0 0 1 0	1	D	1 1	Vn	Vd	imm4	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VEXT{<c>}{<q>}.8 {<Dd>,} <Dn>, <Dm>, #<imm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VEXT{<c>}{<q>}.8 {<Qd>,} <Qn>, <Qm>, #<imm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if Q == '0' && imm4<3> == '1' then UNDEFINED;
quadword_operation = (Q == '1');  position = 8 * UInt(imm4);
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 8	7	6	5	4	3 0
1 1 1 0	1 1 1 1	1	D	1	1	Vn	Vd	imm4	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VEXT{<c>}{<q>}.8 {<Dd>,} <Dn>, <Dm>, #<imm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VEXT{<c>}{<q>}.8 {<Qd>,} <Qn>, <Qm>, #<imm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if Q == '0' && imm4<3> == '1' then UNDEFINED;
quadword_operation = (Q == '1');  position = 8 * UInt(imm4);
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

`<imm>` For the 64-bit SIMD vector variant: is the location of the extracted result in the concatenation of the operands, as a number of bytes from the least significant end, in the range 0 to 7, encoded in the "imm4" field.

For the 128-bit SIMD vector variant: is the location of the extracted result in the concatenation of the operands, as a number of bytes from the least significant end, in the range 0 to 15, encoded in the "imm4" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if quadword_operation then
        Q[d>>1] = (Q[m>>1]:Q[n>>1])<position+127:position>;
    else
        D[d] = (D[m]:D[n])<position+63:position>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.85 VEXT (multibyte elements)

Vector Extract extracts elements from the bottom end of the second operand vector and the top end of the first, concatenates them and places the result in the destination vector

This instruction is a pseudo-instruction of the VEXT (byte elements) instruction. This means that:

- The encodings in this description are named to match the encodings of VEXT (byte elements).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VEXT (byte elements) gives the operational pseudocode for this instruction.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
1 1 1 1	0 0 1 0	1	D	1	1	Vn	Vd	imm4	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VEXT{<c>}{<q>}.<size> {<Dd>,} <Dn>, <Dm>, #<imm>
```

is equivalent to

```
VEXT{<c>}{<q>}.8 {<Dd>,} <Dn>, <Dm>, #<imm*(size/8)>
```

and is never the preferred disassembly.

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VEXT{<c>}{<q>}.<size> {<Qd>,} <Qn>, <Qm>, #<imm>
```

is equivalent to

```
VEXT{<c>}{<q>}.8 {<Qd>,} <Qn>, <Qm>, #<imm*(size/8)>
```

and is never the preferred disassembly.

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 8	7	6	5	4	3 0
1 1 1 0	1 1 1 1	1	D	1	1	Vn	Vd	imm4	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VEXT{<c>}{<q>}.<size> {<Dd>,} <Dn>, <Dm>, #<imm>
```

is equivalent to

```
VEXT{<c>}{<q>}.8 {<Dd>,} <Dn>, <Dm>, #<imm*(size/8)>
```

and is never the preferred disassembly.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VEXT{<c>}{<q>}.<size> {<Qd>,} <Qn>, <Qm>, #<imm>
```

is equivalent to

```
VEXT{<c>}{<q>}.8 {<Qd>,} <Qn>, <Qm>, #<imm*(size/8)>
```

and is never the preferred disassembly.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<size>` For the 64-bit SIMD vector variant: is the size of the operation, and can be one of 16 or 32.

For the 128-bit SIMD vector variant: is the size of the operation, and can be one of 16, 32 or 64.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

`<imm>` For the 64-bit SIMD vector variant: is the location of the extracted result in the concatenation of the operands, as a number of bytes from the least significant end, in the range 0 to (128/`<size>`)-1.

For the 128-bit SIMD vector variant: is the location of the extracted result in the concatenation of the operands, as a number of bytes from the least significant end, in the range 0 to (64/`<size>`)-1.

Operation for all encodings

The description of VEXT (byte elements) gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.86 VFMA

Vector Fused Multiply Accumulate multiplies corresponding elements of two vectors, and accumulates the results into the elements of the destination vector. The instruction does not round the result of the multiply before the accumulation.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	0	0	D	0	sz	Vn	Vd	1	1	0	0	N	Q	M	1	Vm
										op												

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VFMA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VFMA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
advsimd = TRUE; op1_neg = (op == '1');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
regs = if Q == '0' then 1 else 2;
```

A2

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	1	1	1	0	1	D	1	0	Vn	Vd	1	0	size		N	0	M	0	Vm
cond																op			

Half-precision scalar variant

Applies when `size == 01`.

```
VFMA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VFMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VFMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
advsimd = FALSE; op1_neg = (op == '1');
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	0	1	1	1	1	0	D	0	sz	Vn	Vd	1	1	0	0	N	Q	M	1	Vm
										op												

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VFMA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VFMA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
advsimd = TRUE; op1_neg = (op == '1');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T2

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	0	1	1	1	0	1	D	1	0	Vn	Vd	1	0	size		N	0	M	0	Vm
																			op			

Half-precision scalar variant

Applies when `size == 01`.

```
VFMA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VFMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VFMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
advsimd = FALSE; op1_neg = (op == '1');
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding A2, T1 and T2: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

`F32`	when `sz = 0`
`F16`	when `sz = 1`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

`<Sn>` Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

`<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                bits(esize) op1 = Elem[D[n+r],e,esize];
                if op1_neg then op1 = FPNeg(op1);
                Elem[D[d+r],e,esize] = FPMulAdd(Elem[D[d+r],e,esize],
                                        op1, Elem[D[m+r],e,esize], StandardFPSCRValue());

    else // VFP instruction
        case esize of
            when 16
                op16 = if op1_neg then FPNeg(S[n]<15:0>) else S[n]<15:0>;
                S[d] = Zeros(16) : FPMulAdd(S[d]<15:0>, op16, S[m]<15:0>, FPSCR);
            when 32
                op32 = if op1_neg then FPNeg(S[n]) else S[n];
                S[d] = FPMulAdd(S[d], op32, S[m], FPSCR);
            when 64
                op64 = if op1_neg then FPNeg(D[n]) else D[n];
                D[d] = FPMulAdd(D[d], op64, D[m], FPSCR);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.87 VFMAB, VFMAT (BFloat16, vector)

The Bfloat16 floating-point widening multiply-add long instruction widens the even-numbered (bottom) or odd-numbered (top) 16-bit elements in the first and second source vectors from Bfloat16 to single-precision format. The instruction then multiplies and adds these values to the overlapping single-precision elements of the destination vector.

Unlike other BFloat16 multiplication instructions, this performs a fused multiply-add, without intermediate rounding that uses the Round to Nearest rounding mode and can generate a floating-point exception that causes cumulative exception bits in the FPSCR to be set.

A1

ARMv8.6

31 30 29 28	27 26 25 24	23 22 21 20	19 16	15 12	11 10 9 8	7 6 5 4	3 0
1 1 1 1	1 1 1 0	0 0 D 1 1	Vn	Vd	1 0 0 0	N Q M 1	Vm

A1 variant

```
VFMA<bt>{<q>}.BF16 <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if !HaveAArch32BF16Ext() then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer elements = 128 DIV 32;
integer sel = UInt(Q);
```

T1

ARMv8.6

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 10 9 8	7 6 5 4	3 0
1 1 1 1	1 1 1 0	0 0 D 1 1	Vn	Vd	1 0 0 0	N Q M 1	Vm

T1 variant

```
VFMA<bt>{<q>}.BF16 <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32BF16Ext() then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer elements = 128 DIV 32;
integer sel = UInt(Q);
```

Assembler symbols

<bt> Is the bottom or top element specifier, encoded in the "T" field. It can have the following values:

B when T = 0

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T when T = 1

<q> See *Standard assembler syntax fields* on page F2-4080.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation for all encodings

```
CheckAdvSIMDEnabled();
bits(128) operand1 = Q[n>>1];
bits(128) operand2 = Q[m>>1];
bits(128) operand3 = Q[d>>1];
bits(128) result;

for e = 0 to elements-1
    bits(32) element1 = Elem[operand1, 2 * e + sel, 16] : Zeros(16);
    bits(32) element2 = Elem[operand2, 2 * e + sel, 16] : Zeros(16);
    bits(32) addend = Elem[operand3, e, 32];
    Elem[result, e, 32] = FPMulAdd(addend, element1, element2,
                                   StandardFPSCRValue());

Q[d>>1] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.88 VFMAB, VFMAT (BFloat16, by scalar)

The BFloat16 floating-point widening multiply-add long instruction widens the even-numbered (bottom) or odd-numbered (top) 16-bit elements in the first source vector, and an indexed element in the second source vector from Bfloat16 to single-precision format. The instruction then multiplies and adds these values to the overlapping single-precision elements of the destination vector.

Unlike other BFloat16 multiplication instructions, this performs a fused multiply-add, without intermediate rounding that uses the Round to Nearest rounding mode and can generate a floating-point exception that causes cumulative exception bits in the FPSCR to be set.

A1

ARMv8.6

31 30 29 28	27 26 25 24	23	22	21 20	19 16	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	0	D	1 1	Vn	Vd	1	0	0	0	N	Q	M	1	Vm

A1 variant

```
VFMA<bt>{<q>}.BF16 <Qd>, <Qn>, <Dm>[<index>]
```

Decode for this encoding

```
if !HaveAArch32BF16Ext() then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm<2:0>);
integer i = UInt(M:Vm<3>);
integer elements = 128 DIV 32;
integer sel = UInt(Q);
```

T1

ARMv8.6

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	0	D	1	1	Vn	Vd	1	0	0	0	N	Q	M	1	Vm

T1 variant

```
VFMA<bt>{<q>}.BF16 <Qd>, <Qn>, <Dm>[<index>]
```

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32BF16Ext() then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm<2:0>);
integer i = UInt(M:Vm<3>);
integer elements = 128 DIV 32;
integer sel = UInt(Q);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<bt>` Is the bottom or top element specifier, encoded in the "T" field. It can have the following values:

- `B` when `T = 0`
- `T` when `T = 1`

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>" field.

`<index>` Is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field.

Operation for all encodings

```
CheckAdvSIMDEnabled();
bits(128) operand1 = Q[n>>1];
bits(64) operand2 = D[m];
bits(128) operand3 = Q[d>>1];
bits(128) result;

bits(32) element2 = Elem[operand2, i, 16] : Zeros(16);

for e = 0 to elements-1
    bits(32) element1 = Elem[operand1, 2 * e + sel, 16] : Zeros(16);
    bits(32) addend = Elem[operand3, e, 32];
    Elem[result, e, 32] = FPMulAdd(addend, element1, element2,
                                   StandardFPSCRValue());

Q[d>>1] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.89 VFMAL (vector)

Vector Floating-point Multiply-Add Long to accumulator (vector). This instruction multiplies corresponding values in the vectors in the two source SIMD&FP registers, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_ISAR6.FHM indicates whether this instruction is supported.

A1

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	0	0	0	D	1	0	Vn	Vd	1	0	0	0	N	Q	M	1	Vm
								S														

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VFMAL{<q>}.F16 <Dd>, <Sn>, <Sm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VFMAL{<q>}.F16 <Qd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;

integer d = UInt(D:Vd);
integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(M:Vm) else UInt(Vm:M);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datasize = if Q=='1' then 64 else 32;
boolean sub_op = S=='1';
```

T1

ARMv8.2

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	0	0	0	D	1	x	Vn	Vd	1	0	0	0	N	Q	M	1	Vm
								S		op2												

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

64-bit SIMD vector variant

Applies when `op2 == 10 && Q == 0`.

```
VFMAL{<q>}.F16 <Dd>, <Sn>, <Sm>
```

128-bit SIMD vector variant

Applies when `op2 == 11 && Q == 1`.

```
VFMAL{<q>}.F16 <Qd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;

integer d = UInt(D:Vd);
integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(M:Vm) else UInt(Vm:M);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datasize = if Q=='1' then 64 else 32;
boolean sub_op = S=='1';
```

Assembler symbols

`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Qd>`	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
`<Dn>`	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
`<Dm>`	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
`<Sn>`	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
`<Sm>`	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

Operation for all encodings

```
CheckAdvSIMDEnabled();
bits(datasize) operand1 ;
bits(datasize) operand2 ;
bits(64) operand3;
bits(64) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;

if Q=='0' then
    operand1 = S[n]<datasize-1:0>;
    operand2 = S[m]<datasize-1:0>;
else
    operand1 = D[n]<datasize-1:0>;
    operand2 = D[m]<datasize-1:0>;
for r = 0 to regs-1
    operand3 = D[d+r];
    for e = 0 to 1
        element1 = Elem[operand1, 2*r+e, esize DIV 2];
        element2 = Elem[operand2, 2*r+e, esize DIV 2];
        if sub_op then element1 = FPNeg(element1);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2,
StandardFPSCRValue());
    D[d+r] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.90 VFMAL (by scalar)

Vector Floating-point Multiply-Add Long to accumulator (by scalar). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_ISAR6.FHM indicates whether this instruction is supported.

A1

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	1	0	0	D	0	0	Vn	Vd	1	0	0	0	N	Q	M	1	Vm
											S											

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VFMAL{<q>}.F16 <Dd>, <Sn>, <Sm>[<index>]
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VFMAL{<q>}.F16 <Qd>, <Dn>, <Dm>[<index>]
```

Decode for all variants of this encoding

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;

integer d = UInt(D:Vd);
integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(Vm<2:0>) else UInt(Vm<2:0>:M);

integer index = if Q == '1' then UInt(M:Vm<3>) else UInt(Vm<3>);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datasize = if Q=='1' then 64 else 32;
boolean sub_op = S=='1';
```

T1

ARMv8.2

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	1	0	0	D	0	0	Vn	Vd	1	0	0	0	N	Q	M	1	Vm
											S											

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VFMAL{<q>}.F16 <Dd>, <Sn>, <Sm>[<index>]
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VFMAL{<q>}.F16 <Qd>, <Dn>, <Dm>[<index>]
```

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;

integer d = UInt(D:Vd);
integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(Vm<2:0>) else UInt(Vm<2:0>:M);

integer index = if Q == '1' then UInt(M:Vm<3>) else UInt(Vm<3>);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datasize = if Q=='1' then 64 else 32;
boolean sub_op = S=='1';
```

Assembler symbols

`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Qd>`	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
`<Dn>`	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
`<Dm>`	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>" field.
`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
`<Sn>`	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
`<Sm>`	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>:M" field.
`<index>`	For the 64-bit SIMD vector variant: is the element index in the range 0 to 1, encoded in the "Vm<3>" field. For the 128-bit SIMD vector variant: is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field.

Operation for all encodings

```
CheckAdvSIMDEnabled();
bits(datasize) operand1 ;
bits(datasize) operand2 ;
bits(64) operand3;
bits(64) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;

if Q=='0' then
    operand1 = S[n]<datasize-1:0>;
    operand2 = S[m]<datasize-1:0>;
else
    operand1 = D[n]<datasize-1:0>;
    operand2 = D[m]<datasize-1:0>;
element2 = Elem[operand2, index, esize DIV 2];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
for r = 0 to regs-1
    operand3 = D[d+r];
    for e = 0 to 1
        element1 = Elem[operand1, 2*r+e, esize DIV 2];
        if sub_op then element1 = FPNeg(element1);
        Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2,
StandardFPSCRValue());
    D[d+r] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.91 VFMS

Vector Fused Multiply Subtract negates the elements of one vector and multiplies them with the corresponding elements of another vector, adds the products to the corresponding elements of the destination vector, and places the results in the destination vector. The instruction does not round the result of the multiply before the addition.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	0	0	D	1	sz	Vn	Vd	1	1	0	0	N	Q	M	1	Vm
										op												

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VFMS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VFMS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
advsimd = TRUE; op1_neg = (op == '1');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
regs = if Q == '0' then 1 else 2;
```

A2

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	1	1	1	0	1	D	1	0	Vn	Vd	1	0	size		N	1	M	0	Vm
cond																op			

Half-precision scalar variant

Applies when `size == 01`.

```
VFMS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VFMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VFMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
advsimd = FALSE; op1_neg = (op == '1');
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	0	1	1	1	1	0	D	1	sz	Vn	Vd	1	1	0	0	N	Q	M	1	Vm
										op												

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VFMS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VFMS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
advsimd = TRUE; op1_neg = (op == '1');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T2

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	0	1	1	1	0	1	D	1	0	Vn	Vd	1	0	size		N	1	M	0	Vm
																			op			

Half-precision scalar variant

Applies when `size == 01`.

```
VFMS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VFMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VFMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
advsimd = FALSE; op1_neg = (op == '1');
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding A2, T1 and T2: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

`F32`	when `sz = 0`
`F16`	when `sz = 1`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

`<Sn>` Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

`<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                bits(esize) op1 = Elem[D[n+r],e,esize];
                if op1_neg then op1 = FPNeg(op1);
                Elem[D[d+r],e,esize] = FPMulAdd(Elem[D[d+r],e,esize],
                                        op1, Elem[D[m+r],e,esize], StandardFPSCRValue());

    else // VFP instruction
        case esize of
            when 16
                op16 = if op1_neg then FPNeg(S[n]<15:0>) else S[n]<15:0>;
                S[d] = Zeros(16) : FPMulAdd(S[d]<15:0>, op16, S[m]<15:0>, FPSCR);
            when 32
                op32 = if op1_neg then FPNeg(S[n]) else S[n];
                S[d] = FPMulAdd(S[d], op32, S[m], FPSCR);
            when 64
                op64 = if op1_neg then FPNeg(D[n]) else D[n];
                D[d] = FPMulAdd(D[d], op64, D[m], FPSCR);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.92 VFMSL (vector)

Vector Floating-point Multiply-Subtract Long from accumulator (vector). This instruction negates the values in the vector of one SIMD&FP register, multiplies these with the corresponding values in another vector, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_ISAR6.FHM indicates whether this instruction is supported.

A1

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	0	0	1	D	1	0	Vn	Vd	1	0	0	0	N	Q	M	1	Vm
								S														

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VFMSL{<q>}.F16 <Dd>, <Sn>, <Sm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VFMSL{<q>}.F16 <Qd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;

integer d = UInt(D:Vd);
integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(M:Vm) else UInt(Vm:M);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datasize = if Q=='1' then 64 else 32;
boolean sub_op = S=='1';
```

T1

ARMv8.2

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	0	0	1	D	1	0	Vn	Vd	1	0	0	0	N	Q	M	1	Vm
								S														

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VFMSL{<q>}.F16 <Dd>, <Sn>, <Sm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VFMSL{<q>}.F16 <Qd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;

integer d = UInt(D:Vd);
integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(M:Vm) else UInt(Vm:M);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datasize = if Q=='1' then 64 else 32;
boolean sub_op = S=='1';
```

Assembler symbols

`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Qd>`	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
`<Dn>`	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
`<Dm>`	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
`<Sn>`	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
`<Sm>`	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

Operation for all encodings

```
CheckAdvSIMDEnabled();
bits(datasize) operand1 ;
bits(datasize) operand2 ;
bits(64) operand3;
bits(64) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;

if Q=='0' then
    operand1 = S[n]<datasize-1:0>;
    operand2 = S[m]<datasize-1:0>;
else
    operand1 = D[n]<datasize-1:0>;
    operand2 = D[m]<datasize-1:0>;
for r = 0 to regs-1
    operand3 = D[d+r];
    for e = 0 to 1
        element1 = Elem[operand1, 2*r+e, esize DIV 2];
        element2 = Elem[operand2, 2*r+e, esize DIV 2];
        if sub_op then element1 = FPNeg(element1);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2,
StandardFPSCRValue());
    D[d+r] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.93 VFMSL (by scalar)

Vector Floating-point Multiply-Subtract Long from accumulator (by scalar). This instruction multiplies the negated vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_ISAR6.FHM indicates whether this instruction is supported.

A1

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	1	0	0	D	0	1	Vn	Vd	1	0	0	0	N	Q	M	1	Vm
											S											

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VFMSL{<q>}.F16 <Dd>, <Sn>, <Sm>[<index>]
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VFMSL{<q>}.F16 <Qd>, <Dn>, <Dm>[<index>]
```

Decode for all variants of this encoding

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;

integer d = UInt(D:Vd);
integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(Vm<2:0>) else UInt(Vm<2:0>:M);

integer index = if Q == '1' then UInt(M:Vm<3>) else UInt(Vm<3>);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datasize = if Q=='1' then 64 else 32;
boolean sub_op = S=='1';
```

T1

ARMv8.2

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	1	0	0	D	0	1	Vn	Vd	1	0	0	0	N	Q	M	1	Vm
											S											

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VFMSL{<q>}.F16 <Dd>, <Sn>, <Sm>[<index>]
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VFMSL{<q>}.F16 <Qd>, <Dn>, <Dm>[<index>]
```

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;

integer d = UInt(D:Vd);
integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(Vm<2:0>) else UInt(Vm<2:0>:M);

integer index = if Q == '1' then UInt(M:Vm<3>) else UInt(Vm<3>);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datasize = if Q=='1' then 64 else 32;
boolean sub_op = S=='1';
```

Assembler symbols

`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Qd>`	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
`<Dn>`	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
`<Dm>`	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>" field.
`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
`<Sn>`	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
`<Sm>`	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>:M" field.
`<index>`	For the 64-bit SIMD vector variant: is the element index in the range 0 to 1, encoded in the "Vm<3>" field. For the 128-bit SIMD vector variant: is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field.

Operation for all encodings

```
CheckAdvSIMDEnabled();
bits(datasize) operand1 ;
bits(datasize) operand2 ;
bits(64) operand3;
bits(64) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;

if Q=='0' then
    operand1 = S[n]<datasize-1:0>;
    operand2 = S[m]<datasize-1:0>;
else
    operand1 = D[n]<datasize-1:0>;
    operand2 = D[m]<datasize-1:0>;
element2 = Elem[operand2, index, esize DIV 2];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
for r = 0 to regs-1
    operand3 = D[d+r];
    for e = 0 to 1
        element1 = Elem[operand1, 2*r+e, esize DIV 2];
        if sub_op then element1 = FPNeg(element1);
        Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2,
StandardFPSCRValue());
    D[d+r] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.94 VFNMA

Vector Fused Negate Multiply Accumulate negates one floating-point register value and multiplies it by another floating-point register value, adds the negation of the floating-point value in the destination register to the product, and writes the result back to the destination register. The instruction does not round the result of the multiply before the addition.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31–28	27 26 25 24	23	22	21 20	19–16	15–12	11 10	9 8	7	6	5	4	3–0
!=1111	1 1 1 0	1	D	0 1	Vn	Vd	1 0	size	N	1	M	0	Vm
cond										op			

Half-precision scalar variant

Applies when `size == 01`.

```
VFNMA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VFNMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VFNMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
op1_neg = (op == '1');
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15 14 13 12	11 10 9 8	7	6	5 4	3–0	15–12	11 10	9 8	7	6	5	4	3–0
1 1 1 0	1 1 1 0	1	D	0 1	Vn	Vd	1 0	size	N	1	M	0	Vm
										op			

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Half-precision scalar variant

Applies when `size == 01`.

```
VFNMA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VFNMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VFNMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
op1_neg = (op == '1');
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Sd>`	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
`<Sn>`	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
`<Sm>`	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
`<Dn>`	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
`<Dm>`	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    case esize of
        when 16
            op16 = if op1_neg then FPNeg(S[n]<15:0>) else S[n]<15:0>;
            S[d] = Zeros(16) : FPMulAdd(FPNeg(S[d]<15:0>), op16, S[m]<15:0>, FPSCR);
        when 32
            op32 = if op1_neg then FPNeg(S[n]) else S[n];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    S[d] = FPMulAdd(FPNeg(S[d]), op32, S[m], FPSCR);
when 64
    op64 = if op1_neg then FPNeg(D[n]) else D[n];
    D[d] = FPMulAdd(FPNeg(D[d]), op64, D[m], FPSCR);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.95 VFNMS

Vector Fused Negate Multiply Subtract multiplies together two floating-point register values, adds the negation of the floating-point value in the destination register to the product, and writes the result back to the destination register. The instruction does not round the result of the multiply before the addition.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	1	1	1	0	1	D	0	1	Vn	Vd	1	0	size		N	0	M	0	Vm
cond																op			

Half-precision scalar variant

Applies when `size == 01`.

```
VFNMS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VFNMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VFNMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
op1_neg = (op == '1');
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	0	1	1	1	0	1	D	0	1	Vn	Vd	1	0	size		N	0	M	0	Vm
																			op			

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Half-precision scalar variant

Applies when `size == 01`.

```
VFNMS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VFNMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VFNMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
op1_neg = (op == '1');
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Sd>`	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
`<Sn>`	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
`<Sm>`	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
`<Dn>`	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
`<Dm>`	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    case esize of
        when 16
            op16 = if op1_neg then FPNeg(S[n]<15:0>) else S[n]<15:0>;
            S[d] = Zeros(16) : FPMulAdd(FPNeg(S[d]<15:0>), op16, S[m]<15:0>, FPSCR);
        when 32
            op32 = if op1_neg then FPNeg(S[n]) else S[n];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    S[d] = FPMulAdd(FPNeg(S[d]), op32, S[m], FPSCR);
when 64
    op64 = if op1_neg then FPNeg(D[n]) else D[n];
    D[d] = FPMulAdd(FPNeg(D[d]), op64, D[m], FPSCR);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.96 VHADD

Vector Halving Add adds corresponding elements in two vectors of integers, shifts each result right one bit, and places the final results in the destination vector. The results of the halving operations are truncated. For rounded results, see VRHADD).

The operand and result elements are all the same type, and can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25	24	23	22	21 20	19 16	15 12	11 10	9	8	7	6	5	4	3 0
1 1 1 1	0 0 1	U	0	D	size	Vn	Vd	0 0	0	0	N	Q	M	0	Vm
									op						

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VHADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VHADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
add = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13	12	11 10 9 8	7	6	5	4	3 0	15 12	11 10	9	8	7	6	5	4	3 0
1 1 1	U	1 1 1 1 0	D	size	Vn			Vd	0 0	0	0	N	Q	M	0	Vm
										op						

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VHADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VHADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
add = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

`S8`	when `U = 0`, `size = 00`
`S16`	when `U = 0`, `size = 01`
`S32`	when `U = 0`, `size = 10`
`U8`	when `U = 1`, `size = 00`
`U16`	when `U = 1`, `size = 01`
`U32`	when `U = 1`, `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Int(Elem[D[n+r],e,esize], unsigned);
            op2 = Int(Elem[D[m+r],e,esize], unsigned);
            result = if add then op1+op2 else op1-op2;
            Elem[D[d+r],e,esize] = result<esize:1>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.97 VHSUB

Vector Halving Subtract subtracts the elements of the second operand from the corresponding elements of the first operand, shifts each result right one bit, and places the final results in the destination vector. The results of the halving operations are truncated. There is no rounding version.

The operand and result elements are all the same type, and can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	U	0	D	size		Vn	Vd	0	0	1	0	N	Q	M	0	Vm
																op						

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VHSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VHSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
add = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	U	1	1	1	1	0	D	size		Vn	Vd	0	0	1	0	N	Q	M	0	Vm
																op						

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VHSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VHSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
add = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

`S8`	when `U = 0`, `size = 00`
`S16`	when `U = 0`, `size = 01`
`S32`	when `U = 0`, `size = 10`
`U8`	when `U = 1`, `size = 00`
`U16`	when `U = 1`, `size = 01`
`U32`	when `U = 1`, `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Int(Elem[D[n+r],e,esize], unsigned);
            op2 = Int(Elem[D[m+r],e,esize], unsigned);
            result = if add then op1+op2 else op1-op2;
            Elem[D[d+r],e,esize] = result<esize:1>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.98 VINS

Vector move Insertion. This instruction copies the lower 16 bits of the 32-bit source SIMD&FP register into the upper 16 bits of the 32-bit destination SIMD&FP register, while preserving the values in the remaining bits.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15 – 12	11	10	9	8	7	6	5	4	3 – 0
1	1	1	1	1	1	1	0	1	D	1	1	0	0	0	0	Vd	1	0	1	0	1	1	M	0	Vm

A1 variant

```
VINS{<q>}.F16 <Sd>, <Sm>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
d = UInt(Vd:D); m = UInt(Vm:M);
```

T1

ARMv8.2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15 – 12	11	10	9	8	7	6	5	4	3 – 0
1	1	1	1	1	1	1	0	1	D	1	1	0	0	0	0	Vd	1	0	1	0	1	1	M	0	Vm

T1 variant

```
VINS{<q>}.F16 <Sd>, <Sm>
```

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveFP16Ext() then UNDEFINED;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
d = UInt(Vd:D); m = UInt(Vm:M);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

<q>	See *Standard assembler syntax fields* on page F2-4080.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Sm>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    S[d] = S[m]<15:0> : S[d]<15:0>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.99 VJCVT

JavaScript Convert to signed fixed-point, rounding toward Zero. This instruction converts the double-precision floating-point value in the SIMD&FP source register to a 32-bit signed integer using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register. If the result is too large to be accommodated as a signed 32-bit integer, then the result is the integer modulo 2^{32}, as held in a 32-bit signed integer.

This instruction can generate a floating-point exception. Depending on the settings in FPSCR, the exception results in either a flag being set or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page E1-3963.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

ARMv8.3

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111 (cond)	1	1	1	0	1	D	1	1	1	0	0	1	Vd	1	0	1	1	1	1	M	0	Vm

A1 variant

```
VJCVT{<q>}.S32.F64 <Sd>, <Dm>
```

Decode for this encoding

```
if !HaveFJCVTZSExt() then UNDEFINED;
if cond != '1110' then UNPREDICTABLE;
d = UInt(Vd:D);  m = UInt(M:Vm);
```

T1

ARMv8.3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	0	1	1	1	0	1	D	1	1	1	0	0	1	Vd	1	0	1	1	1	1	M	0	Vm

T1 variant

```
VJCVT{<q>}.S32.F64 <Sd>, <Dm>
```

Decode for this encoding

```
if !HaveFJCVTZSExt() then UNDEFINED;
if InITBlock() then UNPREDICTABLE;
d = UInt(Vd:D);  m = UInt(M:Vm);
```

Assembler symbols

<q>	See *Standard assembler syntax fields* on page F2-4080.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
EncodingSpecificOperations();
CheckVFPEnabled(TRUE);
bits(64) fltval = D[m];
bits(32) intval;
bit      Z;
(intval, Z) = FPToFixedJS(fltval, FPSCR, FALSE);
FPSCR<31:28> = '0':Z:'00';
S[d] = intval;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.100 VLD1 (single element to one lane)

Load single 1-element structure to one lane of one register loads one element from memory into one element of a register. Elements of the register that are not loaded are unchanged. For details of the addressing mode see *The Advanced SIMD addressing mode* on page F2-4101.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10	9 8	7 4	3 0
1 1 1 1	0 1 0 0	1	D	1	0	Rn	Vd	0 0	0 0	index_align	Rm
								size			

Offset variant

Applies when `Rm == 1111`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then SEE "VLD1 (single element to all lanes)";
if index_align<0> != '0' then UNDEFINED;
ebytes = 1;  index = UInt(index_align<3:1>);  alignment = 1;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

A2

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10	9 8	7 4	3 0
1 1 1 1	0 1 0 0	1	D	1	0	Rn	Vd	0 1	0 0	index_align	Rm
								size			

Offset variant

Applies when `Rm == 1111`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then SEE "VLD1 (single element to all lanes)";
if index_align<1> != '0' then UNDEFINED;
ebytes = 2;  index = UInt(index_align<3:2>);
alignment = if index_align<0> == '0' then 1 else 2;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

A3

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10	9 8	7 4	3 0
1 1 1 1	0 1 0 0	1	D	1	0	Rn	Vd	1 0	0 0	index_align	Rm
								size			

Offset variant

Applies when `Rm == 1111`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then SEE "VLD1 (single element to all lanes)";
if index_align<2> != '0' then UNDEFINED;
if index_align<1:0> != '00' && index_align<1:0> != '11' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
alignment = if index_align<1:0> == '00' then 1 else 4;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10	9 8	7 4	3 0
1 1 1 1	1 0 0 1	1	D	1	0	Rn	Vd	0 0	0 0	index_align	Rm
								size			

Offset variant

Applies when `Rm == 1111`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then SEE "VLD1 (single element to all lanes)";
if index_align<0> != '0' then UNDEFINED;
ebytes = 1;  index = UInt(index_align<3:1>);  alignment = 1;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

T2

15 14 13 12	11 10 9 8	7	6	5	4	3 … 0	15 … 12	11 10	9 8	7 … 4	3 … 0
1 1 1 1	1 0 0 1	1	D	1	0	Rn	Vd	0 1	0 0	index_align	Rm
								size			

Offset variant

Applies when `Rm == 1111`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then SEE "VLD1 (single element to all lanes)";
if index_align<1> != '0' then UNDEFINED;
ebytes = 2;  index = UInt(index_align<3:2>);
alignment = if index_align<0> == '0' then 1 else 2;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

T3

15 14 13 12	11 10 9 8	7	6	5	4	3 … 0	15 … 12	11 10	9 8	7 … 4	3 … 0
1 1 1 1	1 0 0 1	1	D	1	0	Rn	Vd	1 0	0 0	index_align	Rm
								size			

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Offset variant

Applies when `Rm == 1111`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then SEE "VLD1 (single element to all lanes)";
if index_align<2> != '0' then UNDEFINED;
if index_align<1:0> != '00' && index_align<1:0> != '11' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
alignment = if index_align<1:0> == '00' then 1 else 4;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` For encoding A1, A2 and A3: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1, T2 and T3: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

`8`	when `size = 00`
`16`	when `size = 01`
`32`	when `size = 10`

`<list>` Is a list containing the single 64-bit name of the SIMD&FP register holding the element.

The list must be `{ <Dd>[<index>] }`.

The register `<Dd>` is encoded in the "D:Vd" field.

The permitted values and encoding of `<index>` depend on `<size>`:

`<size> == 8<index>` is in the range 0 to 7, encoded in the "index_align<3:1>" field.

`<size> == 16<index>` is in the range 0 to 3, encoded in the "index_align<3:2>" field.

`<size> == 32<index>` is 0 or 1, encoded in the "index_align<3>" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`<align>` When `<size>` == 8, `<align>` must be omitted, otherwise it is the optional alignment.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Whenever `<align>` is omitted, the standard alignment is used, see *Unaligned data access* on page E2-4004, and the encoding depends on `<size>`:

`<size> == 8`Encoded in the "index_align<0>" field as `0`.

`<size> == 16`Encoded in the "index_align<1:0>" field as `0b00`.

`<size> == 32`Encoded in the "index_align<2:0>" field as `0b000`.

Whenever `<align>` is present, the permitted values and encoding depend on `<size>`:

`<size> == 16``<align>` is 16, meaning 16-bit alignment, encoded in the "index_align<1:0>" field as `0b01`.

`<size> == 32``<align>` is 32, meaning 32-bit alignment, encoded in the "index_align<2:0>" field as `0b011`.

`:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see *The Advanced SIMD addressing mode* on page F2-4101.

`<Rm>` Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *The Advanced SIMD addressing mode* on page F2-4101.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    Elem[D[d],index] = MemU[address,ebytes];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + ebytes;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.101 VLD1 (single element to all lanes)

Load single 1-element structure and replicate to all lanes of one register loads one element from memory into every element of one or two vectors. For details of the addressing mode see *The Advanced SIMD addressing mode* on page F2-4101.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10	9 8	7 6	5	4	3 0
1 1 1 1	0 1 0 0	1	D	1	0	Rn	Vd	1 1	0 0	size	T	a	Rm

Offset variant

Applies when `Rm == 1111`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}],<Rm>
```

Decode for all variants of this encoding

```
if size == '11' || (size == '00' && a == '1') then UNDEFINED;
ebytes = 1 << UInt(size);  regs = if T == '0' then 1 else 2;
alignment = if a == '0' then 1 else ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10	9 8	7 6	5	4	3 0
1 1 1 1	1 0 0 1	1	D	1	0	Rn	Vd	1 1	0 0	size	T	a	Rm

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Offset variant

Applies when `Rm == 1111`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' || (size == '00' && a == '1') then UNDEFINED;
ebytes = 1 << UInt(size);  regs = if T == '0' then 1 else 2;
alignment = if a == '0' then 1 else ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *VLD1 (single element to all lanes)* on page K1-7846.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

`8`	when `size = 00`
`16`	when `size = 01`
`32`	when `size = 10`

The encoding `size = 11` is reserved.

`<list>` Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

`{ <Dd>[] }` Encoded in the "T" field as `0`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`{ <Dd>[], <Dd+1>[] }`Encoded in the "T" field as `1`.

The register `<Dd>` is encoded in the "D:Vd" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`<align>` When `<size>` == 8, `<align>` must be omitted, otherwise it is the optional alignment.

Whenever `<align>` is omitted, the standard alignment is used, see *Unaligned data access* on page E2-4004, and is encoded in the "a" field as `0`.

Whenever `<align>` is present, the permitted values and encoding depend on `<size>`:

`<size> == 16<align>` is 16, meaning 16-bit alignment, encoded in the "a" field as `1`.

`<size> == 32<align>` is 32, meaning 32-bit alignment, encoded in the "a" field as `1`.

`:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see *The Advanced SIMD addressing mode* on page F2-4101.

`<Rm>` Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *The Advanced SIMD addressing mode* on page F2-4101.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    bits(64) replicated_element = Replicate(MemU[address,ebytes]);
    for r = 0 to regs-1
        D[d+r] = replicated_element;
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + ebytes;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.102 VLD1 (multiple single elements)

Load multiple single 1-element structures to one, two, three, or four registers loads elements from memory into one, two, three, or four registers, without de-interleaving. Every element of each register is loaded. For details of the addressing mode see *The Advanced SIMD addressing mode* on page F2-4101.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8	7 6 5 4	3 0
1 1 1 1	0 1 0 0	0	D	1	0	Rn	Vd	0 1 1 1	size	align	Rm

Offset variant

Applies when `Rm == 1111`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
regs = 1;  if align<1> == '1' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A2

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8	7 6 5 4	3 0
1 1 1 1	0 1 0 0	0	D	1	0	Rn	Vd	1 0 1 0	size	align	Rm

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Offset variant

Applies when `Rm == 1111`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
regs = 2;  if align == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A3

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8	7 6	5 4	3 0
1 1 1 1	0 1 0 0	0	D	1	0	Rn	Vd	0 1 1 0	size	align	Rm

Offset variant

Applies when `Rm == 1111`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
regs = 3;  if align<1> == '1' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A4

31 30 29 28	27 26 25 24	23 22 21 20	19 16	15 12	11 10 9 8	7 6 5 4	3 0
1 1 1 1	0 1 0 0	0 D 1 0	Rn	Vd	0 0 1 0	size	align	Rm

Offset variant

Applies when `Rm == 1111`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
regs = 4;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7 6	5 4	3 0
1 1 1 1	1 0 0 1 0	D	1	0	Rn	Vd	0 1 1 1	size	align	Rm	

Offset variant

Applies when `Rm == 1111`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
regs = 1;  if align<1> == '1' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T2

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7 6	5 4	3 0
1 1 1 1	1 0 0 1 0	D	1	0	Rn	Vd	1 0 1 0	size	align	Rm	

Offset variant

Applies when `Rm == 1111`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
regs = 2;  if align == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T3

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	1	1	0	0	1	0	D	1	0	Rn	Vd	0	1	1	0	size		align		Rm

Offset variant

Applies when `Rm == 1111`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
regs = 3;  if align<1> == '1' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T4

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	1	1	0	0	1	0	D	1	0	Rn	Vd	0	0	1	0	size		align		Rm

Offset variant

Applies when `Rm == 1111`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
regs = 4;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *VLD1 (multiple single elements)* on page K1-7846.

Related encodings: See *Advanced SIMD element or structure load/store* on page F3-4160 for the T32 instruction set, or *Advanced SIMD element or structure load/store* on page F4-4237 for the A32 instruction set.

Assembler symbols

<c> For encoding A1, A2, A3 and A4: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For encoding T1, T2, T3 and T4: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

- `8` when `size = 00`
- `16` when `size = 01`
- `32` when `size = 10`
- `64` when `size = 11`

`<list>` Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

- `{ <Dd> }` Single register. Selects the A1 and T1 encodings of the instruction.
- `{ <Dd>, <Dd+1> }` Two single-spaced registers. Selects the A2 and T2 encodings of the instruction.
- `{ <Dd>, <Dd+1>, <Dd+2> }` Three single-spaced registers. Selects the A3 and T3 encodings of the instruction.
- `{ <Dd>, <Dd+1>, <Dd+2>, <Dd+3> }` Four single-spaced registers. Selects the A4 and T4 encodings of the instruction.

The register `<Dd>` is encoded in the "D:Vd" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`<align>` Is the optional alignment.

Whenever `<align>` is omitted, the standard alignment is used, see *Unaligned data access* on page E2-4004, and is encoded in the "align" field as `0b00`.

Whenever `<align>` is present, the permitted values are:

- `64` 64-bit alignment, encoded in the "align" field as `0b01`.
- `128` 128-bit alignment, encoded in the "align" field as `0b10`. Available only if `<list>` contains two or four registers.
- `256` 256-bit alignment, encoded in the "align" field as `0b11`. Available only if `<list>` contains four registers.

`:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see *The Advanced SIMD addressing mode* on page F2-4101.

`<Rm>` Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *The Advanced SIMD addressing mode* on page F2-4101.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    for r = 0 to regs-1
        for e = 0 to elements-1
            bits(ebytes*8) data;
            if ebytes != 8 then
                data = MemU[address,ebytes];
            else
                - = AArch32.CheckAlignment(address, ebytes, AccType_NORMAL, iswrite);
                data<31:0> = if BigEndian() then MemU[address+4,4] else MemU[address,4];
                data<63:32> = if BigEndian() then MemU[address,4] else MemU[address+4,4];
            Elem[D[d+r],e] = data;
            address = address + ebytes;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if wback then
    if register_index then
        R[n] = R[n] + R[m];
    else
        R[n] = R[n] + 8*regs;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.103 VLD2 (single 2-element structure to one lane)

Load single 2-element structure to one lane of two registers loads one 2-element structure from memory into corresponding elements of two registers. Elements of the registers that are not loaded are unchanged. For details of the addressing mode see *The Advanced SIMD addressing mode* on page F2-4101.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10	9 8	7 4	3 0
1 1 1 1	0 1 0 0	1	D	1	0	Rn	Vd	0 0	0 1	index_align	Rm
								size			

Offset variant

Applies when `Rm == 1111`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then SEE "VLD2 (single 2-element structure to all lanes)";
ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
alignment = if index_align<0> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A2

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10	9 8	7 4	3 0
1 1 1 1	0 1 0 0	1	D	1	0	Rn	Vd	0 1	0 1	index_align	Rm
								size			

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Offset variant

Applies when `Rm == 1111`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then SEE "VLD2 (single 2-element structure to all lanes)";
ebytes = 2;  index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 4;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A3

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10	9 8	7 4	3 0
1 1 1 1	0 1 0 0	1	D	1	0	Rn	Vd	1 0	0 1	index_align	Rm
								size			

Offset variant

Applies when `Rm == 1111`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if size == '11' then SEE "VLD2 (single 2-element structure to all lanes)";
if index_align<1> != '0' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7 … 4	3 … 0
1	1	1	1	1	0	0	1	1	D	1	0	Rn	Vd	0	0	0	1	index_align	Rm
														size					

Offset variant

Applies when `Rm == 1111`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then SEE "VLD2 (single 2-element structure to all lanes)";
ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
alignment = if index_align<0> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T2

15 14 13 12	11 10 9 8	7	6	5	4	3 … 0	15 … 12	11 10	9 8	7 … 4	3 … 0
1 1 1 1	1 0 0 1	1	D	1	0	Rn	Vd	0 1	0 1	index_align	Rm
								size			

Offset variant

Applies when `Rm == 1111`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then SEE "VLD2 (single 2-element structure to all lanes)";
ebytes = 2;  index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 4;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T3

15 14 13 12	11 10 9 8	7	6	5	4	3 … 0	15 … 12	11 10	9 8	7 … 4	3 … 0
1 1 1 1	1 0 0 1	1	D	1	0	Rn	Vd	1 0	0 1	index_align	Rm
								size			

Offset variant

Applies when `Rm == 1111`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then SEE "VLD2 (single 2-element structure to all lanes)";
if index_align<1> != '0' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *VLD2 (single 2-element structure to one lane)* on page K1-7846.

Assembler symbols

`<c>` For encoding A1, A2 and A3: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1, T2 and T3: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

`8`	when `size = 00`
`16`	when `size = 01`
`32`	when `size = 10`

`<list>` Is a list containing the 64-bit names of the two SIMD&FP registers holding the element.

The list must be one of:

`{ <Dd>[<index>], <Dd+1>[<index>] }`Single-spaced registers, encoded as "spacing" = `0`.

`{ <Dd>[<index>], <Dd+2>[<index>] }`Double-spaced registers, encoded as "spacing" = `1`. Not permitted when `<size>` == 8.

The encoding of "spacing" depends on `<size>`:

`<size> == 16`"spacing" is encoded in the "index_align<1>" field.

`<size> == 32`"spacing" is encoded in the "index_align<2>" field.

The register `<Dd>` is encoded in the "D:Vd" field.

The permitted values and encoding of `<index>` depend on `<size>`:

`<size> == 8``<index>` is in the range 0 to 7, encoded in the "index_align<3:1>" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<size>` == `16<index>` is in the range 0 to 3, encoded in the "index_align<3:2>" field.

`<size>` == `32<index>` is 0 or 1, encoded in the "index_align<3>" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`<align>` Is the optional alignment.

Whenever `<align>` is omitted, the standard alignment is used, see *Unaligned data access* on page E2-4004, and the encoding depends on `<size>`:

`<size>` == `8`Encoded in the "index_align<0>" field as `0`.

`<size>` == `16`Encoded in the "index_align<0>" field as `0`.

`<size>` == `32`Encoded in the "index_align<1:0>" field as `0b00`.

Whenever `<align>` is present, the permitted values and encoding depend on `<size>`:

`<size>` == `8<align>` is 16, meaning 16-bit alignment, encoded in the "index_align<0>" field as `1`.

`<size>` == `16<align>` is 32, meaning 32-bit alignment, encoded in the "index_align<0>" field as `1`.

`<size>` == `32<align>` is 64, meaning 64-bit alignment, encoded in the "index_align<1:0>" field as `0b01`.

`:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see *The Advanced SIMD addressing mode* on page F2-4101.

`<Rm>` Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *The Advanced SIMD addressing mode* on page F2-4101.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    Elem[D[d], index] = MemU[address,ebytes];
    Elem[D[d2],index] = MemU[address+ebytes,ebytes];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 2*ebytes;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.104 VLD2 (single 2-element structure to all lanes)

Load single 2-element structure and replicate to all lanes of two registers loads one 2-element structure from memory into all lanes of two registers. For details of the addressing mode see *The Advanced SIMD addressing mode* on page F2-4101.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10	9 8	7 6	5	4	3 0
1 1 1 1	0 1 0 0	1	D	1	0	Rn	Vd	1 1	0 1	size	T	a	Rm

Offset variant

Applies when `Rm == 1111`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}],<Rm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
ebytes = 1 << UInt(size);
alignment = if a == '0' then 1 else 2*ebytes;
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 10 9 8	7 6 5 4	3 0
1 1 1 1	1 0 0 1	1 D 1 0	Rn	Vd	1 1 0 1	size T a	Rm

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Offset variant

Applies when `Rm == 1111`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
ebytes = 1 << UInt(size);
alignment = if a == '0' then 1 else 2*ebytes;
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *VLD2 (single 2-element structure to all lanes)* on page K1-7847.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

`8`	when `size = 00`
`16`	when `size = 01`
`32`	when `size = 10`

The encoding `size = 11` is reserved.

`<list>` Is a list containing the 64-bit names of two SIMD&FP registers.

The list must be one of:

`{ <Dd>[], <Dd+1>[] }`Single-spaced registers, encoded in the "T" field as `0`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`{ <Dd>[], <Dd+2>[] }`Double-spaced registers, encoded in the "T" field as `1`.

The register `<Dd>` is encoded in the "D:Vd" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`<align>` Is the optional alignment.

Whenever `<align>` is omitted, the standard alignment is used, see *Unaligned data access* on page E2-4004, and is encoded in the "a" field as `0`.

Whenever `<align>` is present, the permitted values and encoding depend on `<size>`:

`<size> == 8<align>` is 16, meaning 16-bit alignment, encoded in the "a" field as `1`.

`<size> == 16<align>` is 32, meaning 32-bit alignment, encoded in the "a" field as `1`.

`<size> == 32<align>` is 64, meaning 64-bit alignment, encoded in the "a" field as `1`.

`:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see *The Advanced SIMD addressing mode* on page F2-4101.

`<Rm>` Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *The Advanced SIMD addressing mode* on page F2-4101.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    D[d] = Replicate(MemU[address,ebytes]);
    D[d2] = Replicate(MemU[address+ebytes,ebytes]);
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 2*ebytes;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.105 VLD2 (multiple 2-element structures)

Load multiple 2-element structures to two or four registers loads multiple 2-element structures from memory into two or four registers, with de-interleaving. For more information, see *Element and structure load/store instructions* on page F1-4058. Every element of each register is loaded. For details of the addressing mode see *The Advanced SIMD addressing mode* on page F2-4101.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 8	7 6 5 4	3 0
1 1 1 1	0 1 0 0	0	D	1	0	Rn	Vd	1 0 0 x (itype)	size	align	Rm

Offset variant

Applies when `Rm == 1111`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
regs = 1;  if align == '11' then UNDEFINED;
if size == '11' then UNDEFINED;
inc = if itype == '1001' then 2 else 1;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2+regs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A2

31 30 29 28	27 26 25 24	23 22 21 20	19 16	15 12	11 10 9 8	7 6 5 4	3 0
1 1 1 1	0 1 0 0	0 D 1 0	Rn	Vd	0 0 1 1	size align	Rm

Offset variant

Applies when `Rm == 1111`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
regs = 2;  inc = 2;
if size == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2+regs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 8	7 6 5 4	3 0
1 1 1 1	1 0 0 1	0 D 1 0	Rn	Vd	1 0 0 x	size align	Rm
					itype		

Offset variant

Applies when `Rm == 1111`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
regs = 1;  if align == '11' then UNDEFINED;
if size == '11' then UNDEFINED;
inc = if itype == '1001' then 2 else 1;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2+regs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	1	1	0	0	1	0	D	1	0	Rn	Vd	0	0	1	1	size		align		Rm

Offset variant

Applies when `Rm == 1111`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
regs = 2;  inc = 2;
if size == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2+regs > 32 then UNPREDICTABLE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CONSTRAINED UNPREDICTABLE behavior

If `d2+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *VLD2 (multiple 2-element structures)* on page K1-7846.

Related encodings: See *Advanced SIMD element or structure load/store* on page F3-4160 for the T32 instruction set, or *Advanced SIMD element or structure load/store* on page F4-4237 for the A32 instruction set.

Assembler symbols

`<c>` For encoding A1 and A2: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1 and T2: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

`8` when `size = 00`

`16` when `size = 01`

`32` when `size = 10`

The encoding `size = 11` is reserved.

`<list>` Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

`{ <Dd>, <Dd+1> }` Two single-spaced registers. Selects the A1 and T1 encodings of the instruction, and encoded in the "itype" field as `0b1000`.

`{ <Dd>, <Dd+2> }` Two double-spaced registers. Selects the A1 and T1 encodings of the instruction, and encoded in the "itype" field as `0b1001`.

`{ <Dd>, <Dd+1>, <Dd+2>, <Dd+3> }` Three single-spaced registers. Selects the A2 and T2 encodings of the instruction.

The register `<Dd>` is encoded in the "D:Vd" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`<align>` Is the optional alignment.

Whenever `<align>` is omitted, the standard alignment is used, see *Unaligned data access* on page E2-4004, and is encoded in the "align" field as `0b00`.

Whenever `<align>` is present, the permitted values are:

`64` 64-bit alignment, encoded in the "align" field as `0b01`.

`128` 128-bit alignment, encoded in the "align" field as `0b10`.

`256` 256-bit alignment, encoded in the "align" field as `0b11`. Available only if `<list>` contains four registers.

`:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see *The Advanced SIMD addressing mode* on page F2-4101.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Rm>` Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *The Advanced SIMD addressing mode* on page F2-4101.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r], e] = MemU[address,ebytes];
            Elem[D[d2+r],e] = MemU[address+ebytes,ebytes];
            address = address + 2*ebytes;
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 16*regs;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.106 VLD3 (single 3-element structure to one lane)

Load single 3-element structure to one lane of three registers loads one 3-element structure from memory into corresponding elements of three registers. Elements of the registers that are not loaded are unchanged. For details of the addressing mode see *The Advanced SIMD addressing mode* on page F2-4101.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10	9 8	7 4	3 0
1 1 1 1	0 1 0 0 1	D	1	1	0	Rn	Vd	0 0 (size)	1 0	index_align	Rm

Offset variant

Applies when `Rm == 1111`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then SEE "VLD3 (single 3-element structure to all lanes)";
if index_align<0> != '0' then UNDEFINED;
ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A2

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10	9 8	7 4	3 0
1 1 1 1	0 1 0 0 1	D	1	1	0	Rn	Vd	0 1 (size)	1 0	index_align	Rm

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Offset variant

Applies when `Rm == 1111`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then SEE "VLD3 (single 3-element structure to all lanes)";
if index_align<0> != '0' then UNDEFINED;
ebytes = 2;  index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A3

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10	9 8	7 4	3 0
1 1 1 1	0 1 0 0	1	D	1	0	Rn	Vd	1 0	1 0	index_align	Rm
								size			

Offset variant

Applies when `Rm == 1111`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if size == '11' then SEE "VLD3 (single 3-element structure to all lanes)";
if index_align<1:0> != '00' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7 … 4	3 … 0
1	1	1	1	1	0	0	1	1	D	1	0	Rn	Vd	0	0	1	0	index_align	Rm
														size					

Offset variant

Applies when `Rm == 1111`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then SEE "VLD3 (single 3-element structure to all lanes)";
if index_align<0> != '0' then UNDEFINED;
ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T2

15 14 13 12	11 10 9 8	7	6	5	4	3 … 0	15 … 12	11 10	9 8	7 … 4	3 … 0
1 1 1 1	1 0 0 1	1	D	1	0	Rn	Vd	0 1	1 0	index_align	Rm
								size			

Offset variant

Applies when `Rm == 1111`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then SEE "VLD3 (single 3-element structure to all lanes)";
if index_align<0> != '0' then UNDEFINED;
ebytes = 2;  index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T3

15 14 13 12	11 10 9 8	7	6	5	4	3 … 0	15 … 12	11 10	9 8	7 … 4	3 … 0
1 1 1 1	1 0 0 1	1	D	1	0	Rn	Vd	1 0	1 0	index_align	Rm
								size			

Offset variant

Applies when `Rm == 1111`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then SEE "VLD3 (single 3-element structure to all lanes)";
if index_align<1:0> != '00' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *VLD3 (single 3-element structure to one lane)* on page K1-7847.

Assembler symbols

`<c>` For encoding A1, A2 and A3: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1, T2 and T3: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

`8` when `size = 00`

`16` when `size = 01`

`32` when `size = 10`

`<list>` Is a list containing the 64-bit names of the three SIMD&FP registers holding the element.

The list must be one of:

`{ <Dd>[<index>], <Dd+1>[<index>], <Dd+2>[<index>] }` Single-spaced registers, encoded as "spacing" = `0`.

`{ <Dd>[<index>], <Dd+2>[<index>], <Dd+4>[<index>] }` Double-spaced registers, encoded as "spacing" = `1`. Not permitted when `<size>` == 8.

The encoding of "spacing" depends on `<size>`:

`<size> == 8` "spacing" is encoded in the "index_align<0>" field.

`<size> == 16` "spacing" is encoded in the "index_align<1>" field, and "index_align<0>" is set to `0`.

`<size> == 32` "spacing" is encoded in the "index_align<2>" field, and "index_align<1:0>" is set to `0b00`.

The register `<Dd>` is encoded in the "D:Vd" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The permitted values and encoding of `<index>` depend on `<size>`:

`<size> == 8``<index>` is in the range 0 to 7, encoded in the "index_align<3:1>" field.

`<size> == 16``<index>` is in the range 0 to 3, encoded in the "index_align<3:2>" field.

`<size> == 32``<index>` is 0 or 1, encoded in the "index_align<3>" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`<Rm>` Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *The Advanced SIMD addressing mode* on page F2-4101.

Alignment

Standard alignment rules apply, see *Alignment support* on page B2-148.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];
    Elem[D[d], index] = MemU[address,ebytes];
    Elem[D[d2],index] = MemU[address+ebytes,ebytes];
    Elem[D[d3],index] = MemU[address+2*ebytes,ebytes];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 3*ebytes;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.107 VLD3 (single 3-element structure to all lanes)

Load single 3-element structure and replicate to all lanes of three registers loads one 3-element structure from memory into all lanes of three registers. For details of the addressing mode see *The Advanced SIMD addressing mode* on page F2-4101.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10	9 8	7 6	5	4	3 0
1 1 1 1	0 1 0 0	1	D	1	0	Rn	Vd	1 1	1 0	size	T	0	Rm
												a	

Offset variant

Applies when `Rm == 1111`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' || a == '1' then UNDEFINED;
ebytes = 1 << UInt(size);
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 10 9 8	7 6 5	4	3 0
1 1 1 1	1 0 0 1	1 D 1 0	Rn	Vd	1 1 1 0	size	T	0 Rm
								a

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Offset variant

Applies when `Rm == 1111`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' || a == '1' then UNDEFINED;
ebytes = 1 << UInt(size);
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *VLD3 (single 3-element structure to all lanes)* on page K1-7847.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

`8`	when `size = 00`
`16`	when `size = 01`
`32`	when `size = 10`

The encoding `size = 11` is reserved.

`<list>` Is a list containing the 64-bit names of three SIMD&FP registers.

The list must be one of:

`{ <Dd>[], <Dd+1>[], <Dd+2>[] }`Single-spaced registers, encoded in the "T" field as `0`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`{ <Dd>[], <Dd+2>[], <Dd+4>[] }`Double-spaced registers, encoded in the "T" field as `1`.

The register `<Dd>` is encoded in the "D:Vd" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`<Rm>` Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *The Advanced SIMD addressing mode* on page F2-4101.

Alignment

Standard alignment rules apply, see *Alignment support* on page B2-148.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];
    D[d] = Replicate(MemU[address,ebytes]);
    D[d2] = Replicate(MemU[address+ebytes,ebytes]);
    D[d3] = Replicate(MemU[address+2*ebytes,ebytes]);
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 3*ebytes;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.108 VLD3 (multiple 3-element structures)

Load multiple 3-element structures to three registers loads multiple 3-element structures from memory into three registers, with de-interleaving. For more information, see *Element and structure load/store instructions* on page F1-4058. Every element of each register is loaded. For details of the addressing mode see *The Advanced SIMD addressing mode* on page F2-4101.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 8	7 6 5 4	3 0
1 1 1 1	0 1 0 0	0	D	1	0	Rn	Vd	0 1 0 x (itype)	size	align	Rm

Offset variant

Applies when `Rm == 1111`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
case itype of
    when '0100'
        inc = 1;
    when '0101'
        inc = 2;
    otherwise
        SEE "Related encodings";
if size == '11' || align<1> == '1' then UNDEFINED;
alignment = if align<0> == '0' then 1 else 8;
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 8	7 6 5 4	3 0
1 1 1 1	1 0 0 1 0	D 1 0	Rn	Vd	0 1 0 x (itype)	size	align	Rm

Offset variant

Applies when `Rm == 1111`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
case itype of
    when '0100'
        inc = 1;
    when '0101'
        inc = 2;
    otherwise
        SEE "Related encodings";
if size == '11' || align<1> == '1' then UNDEFINED;
alignment = if align<0> == '0' then 1 else 8;
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *VLD3 (multiple 3-element structures)* on page K1-7847.

Related encodings: See *Advanced SIMD element or structure load/store* on page F3-4160 for the T32 instruction set, or *Advanced SIMD element or structure load/store* on page F4-4237 for the A32 instruction set.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

`8` when `size = 00`

`16` when `size = 01`

`32` when `size = 10`

The encoding `size = 11` is reserved.

`<list>` Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

`{ <Dd>, <Dd+1>, <Dd+2> }`Single-spaced registers, encoded in the "itype" field as `0b0100`.

`{ <Dd>, <Dd+2>, <Dd+4> }`Double-spaced registers, encoded in the "itype" field as `0b0101`.

The register `<Dd>` is encoded in the "D:Vd" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`<align>` Is the optional alignment.

Whenever `<align>` is omitted, the standard alignment is used, see *Unaligned data access* on page E2-4004, and is encoded in the "align" field as `0b00`.

Whenever `<align>` is present, the only permitted values is 64, meaning 64-bit alignment, encoded in the "align" field as `0b01`.

`:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see *The Advanced SIMD addressing mode* on page F2-4101.

`<Rm>` Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about `<Rn>`, `!`, and `<Rm>`, see *The Advanced SIMD addressing mode* on page F2-4101.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    for e = 0 to elements-1
        Elem[D[d], e] = MemU[address,ebytes];
        Elem[D[d2],e] = MemU[address+ebytes,ebytes];
        Elem[D[d3],e] = MemU[address+2*ebytes,ebytes];
        address = address + 3*ebytes;
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 24;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.109 VLD4 (single 4-element structure to one lane)

Load single 4-element structure to one lane of four registers loads one 4-element structure from memory into corresponding elements of four registers. Elements of the registers that are not loaded are unchanged. For details of the addressing mode see *The Advanced SIMD addressing mode* on page F2-4101.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10	9 8	7 4	3 0
1 1 1 1	0 1 0 0	1	D	1	0	Rn	Vd	0 0	1 1	index_align	Rm
								size			

Offset variant

Applies when `Rm == 1111`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then SEE "VLD4 (single 4-element structure to all lanes)";
ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
alignment = if index_align<0> == '0' then 1 else 4;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A2

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10	9 8	7 4	3 0
1 1 1 1	0 1 0 0	1	D	1	0	Rn	Vd	0 1	1 1	index_align	Rm
								size			

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Offset variant

Applies when `Rm == 1111`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then SEE "VLD4 (single 4-element structure to all lanes)";
ebytes = 2;  index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A3

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10	9 8	7 4	3 0
1 1 1 1	0 1 0 0	1	D	1	0	Rn	Vd	1 0	1 1	index_align	Rm
								size			

Offset variant

Applies when `Rm == 1111`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if size == '11' then SEE "VLD4 (single 4-element structure to all lanes)";
if index_align<1:0> == '11' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<1:0> == '00' then 1 else 4 << UInt(index_align<1:0>);
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 – 0	15 – 12	11	10	9	8	7 – 4	3 – 0
1	1	1	1	1	0	0	1	1	D	1	0	Rn	Vd	0	0	1	1	index_align	Rm
														size					

Offset variant

Applies when `Rm == 1111`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then SEE "VLD4 (single 4-element structure to all lanes)";
ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
alignment = if index_align<0> == '0' then 1 else 4;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T2

15 14 13 12	11 10 9 8	7	6	5	4	3 … 0	15 … 12	11 10	9 8	7 … 4	3 … 0
1 1 1 1	1 0 0 1	1	D	1	0	Rn	Vd	0 1	1 1	index_align	Rm
								size			

Offset variant

Applies when `Rm == 1111`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then SEE "VLD4 (single 4-element structure to all lanes)";
ebytes = 2;  index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T3

15 14 13 12	11 10 9 8	7	6	5	4	3 … 0	15 … 12	11 10	9 8	7 … 4	3 … 0
1 1 1 1	1 0 0 1	1	D	1	0	Rn	Vd	1 0	1 1	index_align	Rm
								size			

Offset variant

Applies when `Rm == 1111`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then SEE "VLD4 (single 4-element structure to all lanes)";
if index_align<1:0> == '11' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<1:0> == '00' then 1 else 4 << UInt(index_align<1:0>);
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *VLD4 (single 4-element structure to one lane)* on page K1-7847.

Assembler symbols

`<c>` For encoding A1, A2 and A3: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1, T2 and T3: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

`8`	when `size = 00`
`16`	when `size = 01`
`32`	when `size = 10`

`<list>` Is a list containing the 64-bit names of the four SIMD&FP registers holding the element.

The list must be one of:

`{ <Dd>[<index>], <Dd+1>[<index>], <Dd+2>[<index>], <Dd+3>[<index>] }` Single-spaced registers, encoded as "spacing" = `0`.

`{ <Dd>[<index>], <Dd+2>[<index>], <Dd+4>[<index>], <Dd+6>[<index>] }` Double-spaced registers, encoded as "spacing" = `1`. Not permitted when `<size>` == 8.

The encoding of "spacing" depends on `<size>`:

`<size> == 16` "spacing" is encoded in the "index_align<1>" field.

`<size> == 32` "spacing" is encoded in the "index_align<2>" field.

The register `<Dd>` is encoded in the "D:Vd" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The permitted values and encoding of `<index>` depend on `<size>`:

`<size> == 8<index>` is in the range 0 to 7, encoded in the "index_align<3:1>" field.

`<size> == 16<index>` is in the range 0 to 3, encoded in the "index_align<3:2>" field.

`<size> == 32<index>` is 0 or 1, encoded in the "index_align<3>" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`<align>` Is the optional alignment.

Whenever `<align>` is omitted, the standard alignment is used, see *Unaligned data access* on page E2-4004, and the encoding depends on `<size>`:

`<size> == 8`Encoded in the "index_align<0>" field as `0`.

`<size> == 16`Encoded in the "index_align<0>" field as `0`.

`<size> == 32`Encoded in the "index_align<1:0>" field as `0b00`.

Whenever `<align>` is present, the permitted values and encoding depend on `<size>`:

`<size> == 8<align>` is 32, meaning 32-bit alignment, encoded in the "index_align<0>" field as `1`.

`<size> == 16<align>` is 64, meaning 64-bit alignment, encoded in the "index_align<0>" field as `1`.

`<size> == 32<align>` can be 64 or 128. 64-bit alignment is encoded in the "index_align<1:0>" field as `0b01`, and 128-bit alignment is encoded in the "index_align<1:0>" field as `0b10`.

`:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see *The Advanced SIMD addressing mode* on page F2-4101.

`<Rm>` Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *The Advanced SIMD addressing mode* on page F2-4101.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    Elem[D[d], index] = MemU[address,ebytes];
    Elem[D[d2],index] = MemU[address+ebytes,ebytes];
    Elem[D[d3],index] = MemU[address+2*ebytes,ebytes];
    Elem[D[d4],index] = MemU[address+3*ebytes,ebytes];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 4*ebytes;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.110 VLD4 (single 4-element structure to all lanes)

Load single 4-element structure and replicate to all lanes of four registers loads one 4-element structure from memory into all lanes of four registers. For details of the addressing mode see *The Advanced SIMD addressing mode* on page F2-4101.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10	9 8	7 6	5	4	3 0
1 1 1 1	0 1 0 0	1	D	1	0	Rn	Vd	1 1	1 1	size	T	a	Rm

Offset variant

Applies when `Rm == 1111`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}],<Rm>
```

Decode for all variants of this encoding

```
if size == '11' && a == '0' then UNDEFINED;
if size == '11' then
    ebytes = 4;  alignment = 16;
else
    ebytes = 1 << UInt(size);
    if size == '10' then
        alignment = if a == '0' then 1 else 8;
    else
        alignment = if a == '0' then 1 else 4*ebytes;
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	1	1	0	0	1	1	D	1	0	Rn	Vd	1	1	1	1	size		T	a	Rm

Offset variant

Applies when `Rm == 1111`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' && a == '0' then UNDEFINED;
if size == '11' then
    ebytes = 4;  alignment = 16;
else
    ebytes = 1 << UInt(size);
    if size == '10' then
        alignment = if a == '0' then 1 else 8;
    else
        alignment = if a == '0' then 1 else 4*ebytes;
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *VLD4 (single 4-element structure to all lanes)* on page K1-7848.

Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

`8` when `size = 00`

`16` when `size = 01`

`32` when `size = 1x`

`<list>` Is a list containing the 64-bit names of four SIMD&FP registers.

The list must be one of:

`{ <Dd>[], <Dd+1>[], <Dd+2>[], <Dd+3>[] }`Single-spaced registers, encoded in the "T" field as `0`.

`{ <Dd>[], <Dd+2>[], <Dd+4>[], <Dd+6>[] }`Double-spaced registers, encoded in the "T" field as `1`.

The register `<Dd>` is encoded in the "D:Vd" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`<align>` Is the optional alignment.

Whenever `<align>` is omitted, the standard alignment is used, see *Unaligned data access* on page E2-4004, and is encoded in the "a" field as `0`.

Whenever `<align>` is present, the permitted values and encoding depend on `<size>`:

`<size> == 8<align>` is 32, meaning 32-bit alignment, encoded in the "a" field as `1`.

`<size> == 16<align>` is 64, meaning 64-bit alignment, encoded in the "a" field as `1`.

`<size> == 32<align>` can be 64 or 128. 64-bit alignment is encoded in the "a:size<0>" field as `0b10`, and 128-bit alignment is encoded in the "a:size<0>" field as `0b11`.

`:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see *The Advanced SIMD addressing mode* on page F2-4101.

`<Rm>` Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *The Advanced SIMD addressing mode* on page F2-4101.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    D[d] = Replicate(MemU[address,ebytes]);
    D[d2] = Replicate(MemU[address+ebytes,ebytes]);
    D[d3] = Replicate(MemU[address+2*ebytes,ebytes]);
    D[d4] = Replicate(MemU[address+3*ebytes,ebytes]);
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 4*ebytes;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.111 VLD4 (multiple 4-element structures)

Load multiple 4-element structures to four registers loads multiple 4-element structures from memory into four registers, with de-interleaving. For more information, see *Element and structure load/store instructions* on page F1-4058. Every element of each register is loaded. For details of the addressing mode see *The Advanced SIMD addressing mode* on page F2-4101.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 8	7 6 5 4	3 0
1 1 1 1	0 1 0 0	0	D	1	0	Rn	Vd	0 0 0 x (itype)	size	align	Rm

Offset variant

Applies when `Rm == 1111`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
case itype of
    when '0000'
        inc = 1;
    when '0001'
        inc = 2;
    otherwise
        SEE "Related encodings";
if size == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 8	7 6	5 4	3 0
1 1 1 1	1 0 0 1	0	D	1	0	Rn	Vd	0 0 0 x (itype)	size	align	Rm

Offset variant

Applies when `Rm == 1111`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
case itype of
    when '0000'
        inc = 1;
    when '0001'
        inc = 2;
    otherwise
        SEE "Related encodings";
if size == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *VLD4 (multiple 4-element structures)* on page K1-7847.

Related encodings: See *Advanced SIMD element or structure load/store* on page F3-4160 for the T32 instruction set, or *Advanced SIMD element or structure load/store* on page F4-4237 for the A32 instruction set.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

`8` when `size = 00`

`16` when `size = 01`

`32` when `size = 10`

The encoding `size = 11` is reserved.

`<list>` Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

`{ <Dd>, <Dd+1>, <Dd+2>, <Dd+3> }`Single-spaced registers, encoded in the "itype" field as `0b0000`.

`{ <Dd>, <Dd+2>, <Dd+4>, <Dd+6> }`Double-spaced registers, encoded in the "itype" field as `0b0001`.

The register `<Dd>` is encoded in the "D:Vd" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`<align>` Is the optional alignment.

Whenever `<align>` is omitted, the standard alignment is used, see *Unaligned data access* on page E2-4004, and is encoded in the "align" field as `0b00`.

Whenever `<align>` is present, the permitted values are:

`64` 64-bit alignment, encoded in the "align" field as `0b01`.

`128` 128-bit alignment, encoded in the "align" field as `0b10`.

`256` 256-bit alignment, encoded in the "align" field as `0b11`.

`:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see *The Advanced SIMD addressing mode* on page F2-4101.

`<Rm>` Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *The Advanced SIMD addressing mode* on page F2-4101.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    for e = 0 to elements-1
        Elem[D[d], e] = MemU[address,ebytes];
        Elem[D[d2],e] = MemU[address+ebytes,ebytes];
        Elem[D[d3],e] = MemU[address+2*ebytes,ebytes];
        Elem[D[d4],e] = MemU[address+3*ebytes,ebytes];
        address = address + 4*ebytes;
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 32;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.112 VLDM, VLDMDB, VLDMIA

Load Multiple SIMD&FP registers loads multiple registers from consecutive locations in the Advanced SIMD and floating-point register file using an address from a general-purpose register.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

This instruction is used by the alias VPOP. See *Alias conditions* on page F6-5262 for details of when each alias is preferred.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7–1	0
!=1111	1	1	0	P	U	D	W	1	Rn	Vd	1	0	1	1	imm8<7:1>	0
cond																imm8<0>

Decrement Before variant

Applies when `P == 1 && U == 0 && W == 1`.

```
VLDMDB{<c>}{<q>}{.<size>} <Rn>!, <dreglist>
```

Increment After variant

Applies when `P == 0 && U == 1`.

```
VLDM{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
VLDMIA{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE "VLDR";
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = FALSE;  add = (U == '1');  wback = (W == '1');
d = UInt(D:Vd);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
regs = UInt(imm8) DIV 2;  // If UInt(imm8) is odd, see "FLDM*X".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
if imm8<0> == '1' && (d+regs) > 16 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as a `VLDM` with the same addressing mode but loads no registers.

If `regs > 16 || (d+regs) > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A2

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7–0
!=1111 (cond)	1	1	0	P	U	D	W	1	Rn	Vd	1	0	1	0	imm8

Decrement Before variant

Applies when `P == 1 && U == 0 && W == 1`.

```
VLDMDB{<c>}{<q>}{.<size>} <Rn>!, <sreglist>
```

Increment After variant

Applies when `P == 0 && U == 1`.

```
VLDM{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
VLDMIA{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE "VLDR";
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = TRUE;  add = (U == '1');  wback = (W == '1');  d = UInt(Vd:D);  n = UInt(Rn);
imm32 = ZeroExtend(imm8:'00', 32);  regs = UInt(imm8);
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || (d+regs) > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as a `VLDM` with the same addressing mode but loads no registers.

If `(d+regs) > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7–1	0
1	1	1	0	1	1	0	P	U	D	W	1	Rn	Vd	1	0	1	1	imm8<7:1>	0 (imm8<0>)

Decrement Before variant

Applies when `P == 1 && U == 0 && W == 1`.

```
VLDMDB{<c>}{<q>}{.<size>} <Rn>!, <dreglist>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Increment After variant

Applies when `P == 0 && U == 1`.

```
VLDM{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
VLDMIA{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE "VLDR";
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = FALSE;  add = (U == '1');  wback = (W == '1');
d = UInt(D:Vd);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
regs = UInt(imm8) DIV 2;  // If UInt(imm8) is odd, see "FLDM*X".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
if imm8<0> == '1' && (d+regs) > 16 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as a `VLDM` with the same addressing mode but loads no registers.

If `regs > 16 || (d+regs) > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T2

15 14 13 12	11 10 9 8	7	6	5	4	3 … 0	15 … 12	11 10	9 8	7 … 0
1 1 1 0	1 1 0	P	U	D	W	1	Rn	Vd	1 0	1 0	imm8

Decrement Before variant

Applies when `P == 1 && U == 0 && W == 1`.

```
VLDMDB{<c>}{<q>}{.<size>} <Rn>!, <sreglist>
```

Increment After variant

Applies when `P == 0 && U == 1`.

```
VLDM{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
VLDMIA{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE "VLDR";
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = TRUE;  add = (U == '1');  wback = (W == '1');  d = UInt(Vd:D);  n = UInt(Rn);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
imm32 = ZeroExtend(imm8:'00', 32);  regs = UInt(imm8);
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || (d+regs) > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as a `VLDM` with the same addressing mode but loads no registers.

If `(d+regs) > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *VLDM* on page K1-7848.

Related encodings: See *Advanced SIMD and floating-point 64-bit move* on page F3-4147 for the T32 instruction set, or *Advanced SIMD and floating-point 64-bit move* on page F4-4220 for the A32 instruction set.

Alias conditions

Alias	is preferred when
VPOP	`P == '0' && U == '1' && W == '1' && Rn == '1101'`

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<size>`	An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.
`<Rn>`	Is the general-purpose base register, encoded in the "Rn" field. If writeback is not specified, the PC can be used.
`!`	Specifies base register writeback. Encoded in the "W" field as 1 if present, otherwise 0.
`<sreglist>`	Is the list of consecutively numbered 32-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "Vd:D", and "imm8" is set to the number of registers in the list. The list must contain at least one register.
`<dreglist>`	Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    address = if add then R[n] else R[n]-imm32;
    for r = 0 to regs-1
        if single_regs then
            S[d+r] = MemA[address,4];  address = address+4;
        else
            word1 = MemA[address,4];  word2 = MemA[address+4,4];  address = address+8;
            // Combine the word-aligned words in the correct order for current endianness.
            D[d+r] = if BigEndian() then word1:word2 else word2:word1;
    if wback then R[n] = if add then R[n]+imm32 else R[n]-imm32;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.113 VLDR (immediate)

Load SIMD&FP register (immediate) loads a single register from the Advanced SIMD and floating-point register file, using an address from a general-purpose register, with an optional offset.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9–8	7–0
!=1111	1	1	0	1	U	D	0	1	!=1111	Vd	1	0	size	imm8
cond									Rn					

Half-precision scalar variant

Applies when `size == 01`.

```
VLDR{<c>}{<q>}.16 <Sd>, [<Rn> {, #{+/-}<imm>}]
```

Single-precision scalar variant

Applies when `size == 10`.

```
VLDR{<c>}{<q>}{.32} <Sd>, [<Rn> {, #{+/-}<imm>}]
```

Double-precision scalar variant

Applies when `size == 11`.

```
VLDR{<c>}{<q>}{.64} <Dd>, [<Rn> {, #{+/-}<imm>}]
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
esize = 8 << UInt(size);  add = (U == '1');
imm32 = if esize == 16 then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);
case size of
    when '01' d = UInt(Vd:D);
    when '10' d = UInt(Vd:D);
    when '11' d = UInt(D:Vd);
n = UInt(Rn);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9–8	7–0
1	1	1	0	1	1	0	1	U	D	0	1	!=1111	Vd	1	0	size	imm8
												Rn					

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Half-precision scalar variant

Applies when `size == 01`.

```
VLDR{<c>}{<q>}.16 <Sd>, [<Rn> {, #{+/-}<imm>}]
```

Single-precision scalar variant

Applies when `size == 10`.

```
VLDR{<c>}{<q>}{.32} <Sd>, [<Rn> {, #{+/-}<imm>}]
```

Double-precision scalar variant

Applies when `size == 11`.

```
VLDR{<c>}{<q>}{.64} <Dd>, [<Rn> {, #{+/-}<imm>}]
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
esize = 8 << UInt(size);  add = (U == '1');
imm32 = if esize == 16 then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);
case size of
    when '01' d = UInt(Vd:D);
    when '10' d = UInt(Vd:D);
    when '11' d = UInt(D:Vd);
n = UInt(Rn);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`.64`	Is an optional data size specifier for 64-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.
`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
`.32`	Is an optional data size specifier for 32-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.
`<Sd>`	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
`<Rn>`	Is the general-purpose base register, encoded in the "Rn" field.

`+/-`	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:
	`-` when `U = 0`
	`+` when `U = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<imm> For the single-precision scalar or double-precision scalar variants: is the optional unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0, and encoded in the "imm8" field as <imm>/4.

For the half-precision scalar variant: is the optional unsigned immediate byte offset, a multiple of 2, in the range 0 to 510, defaulting to 0, and encoded in the "imm8" field as <imm>/2.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    base = if n == 15 then Align(PC,4) else R[n];
    address = if add then (base + imm32) else (base - imm32);
    case esize of
        when 16
            S[d] = Zeros(16) : MemA[address,2];
        when 32
            S[d] = MemA[address,4];
        when 64
            word1 = MemA[address,4];  word2 = MemA[address+4,4];
            // Combine the word-aligned words in the correct order for current endianness.
            D[d] = if BigEndian() then word1:word2 else word2:word1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.114 VLDR (literal)

Load SIMD&FP register (literal) loads a single register from the Advanced SIMD and floating-point register file, using an address from the PC value and an immediate offset.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7–0
!=1111	1	1	0	1	U	D	0	1	1 1 1 1	Vd	1	0	size		imm8
cond									Rn						

Half-precision scalar variant

Applies when `size == 01`.

```
VLDR{<c>}{<q>}.16 <Sd>, <label>
VLDR{<c>}{<q>}.16 <Sd>, [PC, #{+/-}<imm>]
```

Single-precision scalar variant

Applies when `size == 10`.

```
VLDR{<c>}{<q>}{.32} <Sd>, <label>
VLDR{<c>}{<q>}{.32} <Sd>, [PC, #{+/-}<imm>]
```

Double-precision scalar variant

Applies when `size == 11`.

```
VLDR{<c>}{<q>}{.64} <Dd>, <label>
VLDR{<c>}{<q>}{.64} <Dd>, [PC, #{+/-}<imm>]
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
esize = 8 << UInt(size);  add = (U == '1');
imm32 = if esize == 16 then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);
case size of
    when '01' d = UInt(Vd:D);
    when '10' d = UInt(Vd:D);
    when '11' d = UInt(D:Vd);
n = UInt(Rn);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10	9 8	7 0
1 1 1 0	1 1 0 1	U	D	0	1	1 1 1 1	Vd	1 0	size	imm8
						Rn				

Half-precision scalar variant

Applies when `size == 01`.

```
VLDR{<c>}{<q>}.16 <Sd>, <label>
VLDR{<c>}{<q>}.16 <Sd>, [PC, #{+/-}<imm>]
```

Single-precision scalar variant

Applies when `size == 10`.

```
VLDR{<c>}{<q>}{.32} <Sd>, <label>
VLDR{<c>}{<q>}{.32} <Sd>, [PC, #{+/-}<imm>]
```

Double-precision scalar variant

Applies when `size == 11`.

```
VLDR{<c>}{<q>}{.64} <Dd>, <label>
VLDR{<c>}{<q>}{.64} <Dd>, [PC, #{+/-}<imm>]
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
esize = 8 << UInt(size);  add = (U == '1');
imm32 = if esize == 16 then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);
case size of
    when '01' d = UInt(Vd:D);
    when '10' d = UInt(Vd:D);
    when '11' d = UInt(D:Vd);
n = UInt(Rn);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`.64`	Is an optional data size specifier for 64-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.
`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
`.32`	Is an optional data size specifier for 32-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.
`<Sd>`	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<label>` The label of the literal data item to be loaded.

For the single-precision scalar or double-precision scalar variants: the assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values are multiples of 4 in the range -1020 to 1020.

For the half-precision scalar variant: the assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values are multiples of 2 in the range -510 to 510.

If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`.

If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`.

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

`-` when `U = 0`

`+` when `U = 1`

`<imm>` For the single-precision scalar or double-precision scalar variants: is the optional unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0, and encoded in the "imm8" field as <imm>/4.

For the half-precision scalar variant: is the optional unsigned immediate byte offset, a multiple of 2, in the range 0 to 510, defaulting to 0, and encoded in the "imm8" field as <imm>/2.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page F1-4037.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    base = if n == 15 then Align(PC,4) else R[n];
    address = if add then (base + imm32) else (base - imm32);
    case esize of
        when 16
            S[d] = Zeros(16) : MemA[address,2];
        when 32
            S[d] = MemA[address,4];
        when 64
            word1 = MemA[address,4];  word2 = MemA[address+4,4];
            // Combine the word-aligned words in the correct order for current endianness.
            D[d] = if BigEndian() then word1:word2 else word2:word1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.115 VMAX (floating-point)

Vector Maximum compares corresponding elements in two vectors, and copies the larger of each pair into the corresponding element in the destination vector.

The operand vector elements are floating-point numbers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	0	0	D	0	sz	Vn	Vd	1	1	1	1	N	Q	M	0	Vm
										op												

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMAX{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
maximum = (op == '0');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	0	1	1	1	1	0	D	0	sz	Vn	Vd	1	1	1	1	N	Q	M	0	Vm
										op												

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMAX{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
maximum = (op == '0');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

`F32`	when `sz = 0`
`F16`	when `sz = 1`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Floating-point maximum and minimum

- max(+0.0, -0.0) = +0.0
- If any input is a NaN, the corresponding result element is the default NaN.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op2 = Elem[D[m+r],e,esize];
            if maximum then
                Elem[D[d+r],e,esize] = FPMax(op1, op2, StandardFPSCRValue());
            else
                Elem[D[d+r],e,esize] = FPMin(op1, op2, StandardFPSCRValue());
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.116 VMAX (integer)

Vector Maximum compares corresponding elements in two vectors, and copies the larger of each pair into the corresponding element in the destination vector.

The operand vector elements can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

The result vector elements are the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21 20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 U	0	D	size	Vn	Vd	0 1 1 0	N	Q	M	0	Vm
											op	

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMAX{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
maximum = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5 4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 U	1 1 1 1 0	D	size	Vn	Vd		0 1 1 0	N	Q	M	0	Vm
											op	

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMAX{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
maximum = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

`S8`	when `U = 0`, `size = 00`
`S16`	when `U = 0`, `size = 01`
`S32`	when `U = 0`, `size = 10`
`U8`	when `U = 1`, `size = 00`
`U16`	when `U = 1`, `size = 01`
`U32`	when `U = 1`, `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Int(Elem[D[n+r],e,esize], unsigned);
            op2 = Int(Elem[D[m+r],e,esize], unsigned);
            result = if maximum then Max(op1,op2) else Min(op1,op2);
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.117 VMAXNM

This instruction determines the floating-point maximum number.

It handles NaNs in consistence with the IEEE754-2008 specification. It returns the numerical operand when one operand is numerical and the other is a quiet NaN, but otherwise the result is identical to floating-point `VMAX`.

This instruction is not conditional.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	1	0	D	0	sz	Vn	Vd	1	1	1	1	N	Q	M	1	Vm
										op												

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMAXNM{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMAXNM{<q>}.<dt> <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
maximum = (op == '0');
advsimd = TRUE;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	1	0	1	D	0	0	Vn	Vd	1	0	!=00		N	0	M	0	Vm
																size			op			

Half-precision scalar variant

Applies when `size == 01`.

```
VMAXNM{<q>}.F16 <Sd>, <Sn>, <Sm> // Cannot be conditional
```

Single-precision scalar variant

Applies when `size == 10`.

```
VMAXNM{<q>}.F32 <Sd>, <Sn>, <Sm> // Cannot be conditional
```

Double-precision scalar variant

Applies when `size == 11`.

```
VMAXNM{<q>}.F64 <Dd>, <Dn>, <Dm> // Cannot be conditional
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
advsimd = FALSE;
maximum = (op == '0');
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	1	1	1	1	1	0	D	0	sz	Vn	Vd	1	1	1	1	N	Q	M	1	Vm
										op												

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMAXNM{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMAXNM{<q>}.<dt> <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
maximum = (op == '0');
advsimd = TRUE;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	1	1	1	1	0	1	D	0	0	Vn	Vd	1	0	!=00		N	0	M	0	Vm
																size			op			

Half-precision scalar variant

Applies when `size == 01`.

```
VMAXNM{<q>}.F16 <Sd>, <Sn>, <Sm> // Not permitted in IT block
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Single-precision scalar variant

Applies when `size == 10`.

```
VMAXNM{<q>}.F32 <Sd>, <Sn>, <Sm> // Not permitted in IT block
```

Double-precision scalar variant

Applies when `size == 11`.

```
VMAXNM{<q>}.F64 <Dd>, <Dn>, <Dm> // Not permitted in IT block
```

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
advsimd = FALSE;
maximum = (op == '0');
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

`F32`	when `sz = 0`
`F16`	when `sz = 1`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

`<Sn>` Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

`<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
if advsimd then             // Advanced SIMD instruction
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r], e, esize]; op2 = Elem[D[m+r], e, esize];
            if maximum then
                Elem[D[d+r], e, esize] = FPMaxNum(op1, op2, StandardFPSCRValue());
            else
                Elem[D[d+r], e, esize] = FPMinNum(op1, op2, StandardFPSCRValue());
else                        // VFP instruction
    case esize of
        when 16
            if maximum then
                S[d] = Zeros(16) : FPMaxNum(S[n]<15:0>, S[m]<15:0>, FPSCR);
            else
                S[d] = Zeros(16) : FPMinNum(S[n]<15:0>, S[m]<15:0>, FPSCR);
        when 32
            if maximum then
                S[d] = FPMaxNum(S[n], S[m], FPSCR);
            else
                S[d] = FPMinNum(S[n], S[m], FPSCR);
        when 64
            if maximum then
                D[d] = FPMaxNum(D[n], D[m], FPSCR);
            else
                D[d] = FPMinNum(D[n], D[m], FPSCR);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.118 VMIN (floating-point)

Vector Minimum compares corresponding elements in two vectors, and copies the smaller of each pair into the corresponding element in the destination vector.

The operand vector elements are floating-point numbers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	0	0	D	1	sz	Vn	Vd	1	1	1	1	N	Q	M	0	Vm
										op												

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMIN{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
maximum = (op == '0');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	0	1	1	1	1	0	D	1	sz	Vn	Vd	1	1	1	1	N	Q	M	0	Vm
										op												

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMIN{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
maximum = (op == '0');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

`F32`	when `sz = 0`
`F16`	when `sz = 1`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Floating-point minimum

- min(+0.0, -0.0) = -0.0
- If any input is a NaN, the corresponding result element is the default NaN.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op2 = Elem[D[m+r],e,esize];
            if maximum then
                Elem[D[d+r],e,esize] = FPMax(op1, op2, StandardFPSCRValue());
            else
                Elem[D[d+r],e,esize] = FPMin(op1, op2, StandardFPSCRValue());
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.119 VMIN (integer)

Vector Minimum compares corresponding elements in two vectors, and copies the smaller of each pair into the corresponding element in the destination vector.

The operand vector elements can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

The result vector elements are the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21 20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 U	0	D	size	Vn	Vd	0 1 1 0	N	Q	M	1	Vm
											op	

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMIN{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
maximum = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5 4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 U	1 1 1 1 0	D	size	Vn	Vd		0 1 1 0	N	Q	M	1	Vm
											op	

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMIN{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
maximum = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

`S8`	when `U = 0`, `size = 00`
`S16`	when `U = 0`, `size = 01`
`S32`	when `U = 0`, `size = 10`
`U8`	when `U = 1`, `size = 00`
`U16`	when `U = 1`, `size = 01`
`U32`	when `U = 1`, `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Int(Elem[D[n+r],e,esize], unsigned);
            op2 = Int(Elem[D[m+r],e,esize], unsigned);
            result = if maximum then Max(op1,op2) else Min(op1,op2);
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.120 VMINNM

This instruction determines the floating point minimum number.

It handles NaNs in consistence with the IEEE754-2008 specification. It returns the numerical operand when one operand is numerical and the other is a quiet NaN, but otherwise the result is identical to floating-point `VMIN`.

This instruction is not conditional.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	1	0	D	1	sz	Vn	Vd	1	1	1	1	N	Q	M	1	Vm
										op												

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMINNM{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMINNM{<q>}.<dt> <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
maximum = (op == '0');
advsimd = TRUE;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	1	0	1	D	0	0	Vn	Vd	1	0	!=00		N	1	M	0	Vm
																size			op			

Half-precision scalar variant

Applies when `size == 01`.

```
VMINNM{<q>}.F16 <Sd>, <Sn>, <Sm> // Cannot be conditional
```

Single-precision scalar variant

Applies when `size == 10`.

```
VMINNM{<q>}.F32 <Sd>, <Sn>, <Sm> // Cannot be conditional
```

Double-precision scalar variant

Applies when `size == 11`.

```
VMINNM{<q>}.F64 <Dd>, <Dn>, <Dm> // Cannot be conditional
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
advsimd = FALSE;
maximum = (op == '0');
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	1	1	1	1	1	0	D	1	sz	Vn	Vd	1	1	1	1	N	Q	M	1	Vm
										op												

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMINNM{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMINNM{<q>}.<dt> <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
maximum = (op == '0');
advsimd = TRUE;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	1	1	1	1	0	1	D	0	0	Vn	Vd	1	0	!=00		N	1	M	0	Vm
																size			op			

Half-precision scalar variant

Applies when `size == 01`.

```
VMINNM{<q>}.F16 <Sd>, <Sn>, <Sm> // Not permitted in IT block
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Single-precision scalar variant

Applies when `size == 10`.

```
VMINNM{<q>}.F32 <Sd>, <Sn>, <Sm> // Not permitted in IT block
```

Double-precision scalar variant

Applies when `size == 11`.

```
VMINNM{<q>}.F64 <Dd>, <Dn>, <Dm> // Not permitted in IT block
```

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
advsimd = FALSE;
maximum = (op == '0');
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

`F32`	when `sz = 0`
`F16`	when `sz = 1`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

`<Sn>` Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

`<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
if advsimd then             // Advanced SIMD instruction
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r], e, esize]; op2 = Elem[D[m+r], e, esize];
            if maximum then
                Elem[D[d+r], e, esize] = FPMaxNum(op1, op2, StandardFPSCRValue());
            else
                Elem[D[d+r], e, esize] = FPMinNum(op1, op2, StandardFPSCRValue());
else                        // VFP instruction
    case esize of
        when 16
            if maximum then
                S[d] = Zeros(16) : FPMaxNum(S[n]<15:0>, S[m]<15:0>, FPSCR);
            else
                S[d] = Zeros(16) : FPMinNum(S[n]<15:0>, S[m]<15:0>, FPSCR);
        when 32
            if maximum then
                S[d] = FPMaxNum(S[n], S[m], FPSCR);
            else
                S[d] = FPMinNum(S[n], S[m], FPSCR);
        when 64
            if maximum then
                D[d] = FPMaxNum(D[n], D[m], FPSCR);
            else
                D[d] = FPMinNum(D[n], D[m], FPSCR);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.121 VMLA (floating-point)

Vector Multiply Accumulate multiplies corresponding elements in two vectors, and accumulates the results into the elements of the destination vector.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	0	0	D	0	sz	Vn	Vd	1	1	0	1	N	Q	M	1	Vm
										op												

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMLA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMLA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
advsimd = TRUE;  add = (op == '0');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

A2

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	1	1	1	0	0	D	0	0	Vn	Vd	1	0	size		N	0	M	0	Vm
cond																op			

Half-precision scalar variant

Applies when `size == 01`.

```
VMLA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VMLA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE; add = (op == '0');
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	0	1	1	1	1	0	D	0	sz	Vn	Vd	1	1	0	1	N	Q	M	1	Vm
										op												

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMLA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMLA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
advsimd = TRUE;  add = (op == '0');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T2

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	0	1	1	1	0	0	D	0	0	Vn	Vd	1	0	size		N	0	M	0	Vm
																			op			

Half-precision scalar variant

Applies when `size == 01`.

```
VMLA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VMLA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE; add = (op == '0');
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding A2, T1 and T2: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

`F32`	when `sz = 0`
`F16`	when `sz = 1`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

`<Sn>` Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

`<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                product = FPMul(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize], StandardFPSCRValue());
                addend = if add then product else FPNeg(product);
                Elem[D[d+r],e,esize] = FPAdd(Elem[D[d+r],e,esize], addend, StandardFPSCRValue());
    else             // VFP instruction
        case esize of
            when 16
                addend16 = if add then FPMul(S[n]<15:0>, S[m]<15:0>, FPSCR) else FPNeg(FPMul(S[n]<15:0>,
S[m]<15:0>, FPSCR));
                S[d] = Zeros(16) : FPAdd(S[d]<15:0>, addend16, FPSCR);
            when 32
                addend32 = if add then FPMul(S[n], S[m], FPSCR) else FPNeg(FPMul(S[n], S[m], FPSCR));
                S[d] = FPAdd(S[d], addend32, FPSCR);
            when 64
                addend64 = if add then FPMul(D[n], D[m], FPSCR) else FPNeg(FPMul(D[n], D[m], FPSCR));
                D[d] = FPAdd(D[d], addend64, FPSCR);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.122 VMLA (integer)

Vector Multiply Accumulate multiplies corresponding elements in two vectors, and adds the products to the corresponding elements of the destination vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	0	0	D	size		Vn	Vd	1	0	0	1	N	Q	M	0	Vm
							op															

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMLA{<c>}{<q>}.<type><size> <Dd>, <Dn>, <Dm> // Encoding T1/A1, encoded as Q = 0
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMLA{<c>}{<q>}.<type><size> <Qd>, <Qn>, <Qm> // Encoding T1/A1, encoded as Q = 1
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
add = (op == '0');  long_destination = FALSE;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	0	1	1	1	1	0	D	size		Vn	Vd	1	0	0	1	N	Q	M	0	Vm
			op																			

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMLA{<c>}{<q>}.<type><size> <Dd>, <Dn>, <Dm> // Encoding T1/A1, encoded as Q = 0
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMLA{<c>}{<q>}.<type><size> <Qd>, <Qn>, <Qm> // Encoding T1/A1, encoded as Q = 1
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
add = (op == '0');  long_destination = FALSE;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<type>` The data type for the elements of the operands. It must be one of:

- `S` Optional in encoding T1/A1. Encoded as U = 0 in encoding T2/A2.
- `U` Optional in encoding T1/A1. Encoded as U = 1 in encoding T2/A2.
- `I` Available only in encoding T1/A1.

`<size>` The data size for the elements of the operands. It must be one of:

- `8` Encoded as size = `0b00`.
- `16` Encoded as size = `0b01`.
- `32` Encoded as size = `0b10`.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            product = Int(Elem[Din[n+r],e,esize],unsigned) * Int(Elem[Din[m+r],e,esize],unsigned);
            addend = if add then product else -product;
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
            else
                Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.123 VMLA (by scalar)

Vector Multiply Accumulate multiplies elements of a vector by a scalar, and adds the products to corresponding elements of the destination vector.

For more information about scalars see *Advanced SIMD scalars* on page F2-4106.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	Q	1	D	size != 11		Vn	Vd	0	op 0	0	F	N	1	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMLA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMLA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>
```

Decode for all variants of this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || (F == '1' && size == '01' && !HaveFP16Ext()) then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
add = (op == '0');  floating_point = (F == '1');  long_destination = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	Q	1	1	1	1	1	D	size != 11		Vn	Vd	0	op 0	0	F	N	1	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMLA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMLA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || (F == '1' && size == '01' && !HaveFP16Ext()) then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
add = (op == '0');  floating_point = (F == '1');  long_destination = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings

Related encodings: See *Advanced SIMD data-processing* on page F3-4137 for the T32 instruction set, or *Advanced SIMD data-processing* on page F4-4225 for the A32 instruction set.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the scalar and the elements of the operand vector, encoded in the "F:size" field. It can have the following values:

`I16`	when `F = 0`, `size = 01`
`I32`	when `F = 0`, `size = 10`
`F16`	when `F = 1`, `size = 01`
`F32`	when `F = 1`, `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm[x]>` Is the 64-bit name of the second SIMD&FP source register holding the scalar. If `<dt>` is `I16` or `F16`, `Dm` is restricted to D0-D7. `Dm` is encoded in "Vm<2:0>", and `x` is encoded in "M:Vm<3>". If `<dt>` is `I32` or `F32`, `Dm` is restricted to D0-D15. `Dm` is encoded in "Vm", and `x` is encoded in "M".

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    op2 = Elem[Din[m],index,esize];  op2val = Int(op2, unsigned);
    for r = 0 to regs-1
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];  op1val = Int(op1, unsigned);
            if floating_point then
                fp_addend = if add then FPMul(op1,op2,StandardFPSCRValue()) else
FPNeg(FPMul(op1,op2,StandardFPSCRValue()));
                Elem[D[d+r],e,esize] = FPAdd(Elem[Din[d+r],e,esize], fp_addend, StandardFPSCRValue());
            else
                addend = if add then op1val*op2val else -op1val*op2val;
                if long_destination then
                    Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
                else
                    Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.124 VMLAL (integer)

Vector Multiply Accumulate Long multiplies corresponding elements in two vectors, and add the products to the corresponding element of the destination vector. The destination vector element is twice as long as the elements that are multiplied.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	U	1	D	!=11 (size)		Vn	Vd	1	0	0	0	N	0	M	0	Vm

op

A1 variant

```
VMLAL{<c>}{<q>}.<type><size> <Qd>, <Dn>, <Dm> // Encoding T2/A2
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
add = (op == '0');  long_destination = TRUE;  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	U	1	1	1	1	1	D	!=11 (size)		Vn	Vd	1	0	0	0	N	0	M	0	Vm

op

T1 variant

```
VMLAL{<c>}{<q>}.<type><size> <Qd>, <Dn>, <Dm> // Encoding T2/A2
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
add = (op == '0');  long_destination = TRUE;  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;
```

Notes for all encodings

Related encodings: See *Advanced SIMD data-processing* on page F3-4137 for the T32 instruction set, or *Advanced SIMD data-processing* on page F4-4225 for the A32 instruction set.

Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<type>` The data type for the elements of the operands. It must be one of:

- `S` Optional in encoding T1/A1. Encoded as U = 0 in encoding T2/A2.
- `U` Optional in encoding T1/A1. Encoded as U = 1 in encoding T2/A2.
- `I` Available only in encoding T1/A1.

`<size>` The data size for the elements of the operands. It must be one of:

- `8` Encoded as size = `0b00`.
- `16` Encoded as size = `0b01`.
- `32` Encoded as size = `0b10`.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            product = Int(Elem[Din[n+r],e,esize],unsigned) * Int(Elem[Din[m+r],e,esize],unsigned);
            addend = if add then product else -product;
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
            else
                Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.125 VMLAL (by scalar)

Vector Multiply Accumulate Long multiplies elements of a vector by a scalar, and adds the products to corresponding elements of the destination vector. The destination vector elements are twice as long as the elements that are multiplied.

For more information about scalars see *Advanced SIMD scalars* on page F2-4106.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	U	1	D	!=11		Vn	Vd	0	0	1	0	N	1	M	0	Vm
										size				op								

A1 variant

```
VMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  add = (op == '0');  floating_point = FALSE;  long_destination = TRUE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = 1;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	U	1	1	1	1	1	D	!=11		Vn	Vd	0	0	1	0	N	1	M	0	Vm
										size				op								

T1 variant

```
VMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  add = (op == '0');  floating_point = FALSE;  long_destination = TRUE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = 1;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

Notes for all encodings

Related encodings: See *Advanced SIMD data-processing* on page F3-4137 for the T32 instruction set, or *Advanced SIMD data-processing* on page F4-4225 for the A32 instruction set.

Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the scalar and the elements of the operand vector, encoded in the "U:size" field. It can have the following values:

- `S16` when `U = 0`, `size = 01`
- `S32` when `U = 0`, `size = 10`
- `U16` when `U = 1`, `size = 01`
- `U32` when `U = 1`, `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm[x]>` Is the 64-bit name of the second SIMD&FP source register holding the scalar. If `<dt>` is `S16` or `U16`, `Dm` is restricted to D0-D7. `Dm` is encoded in "Vm<2:0>", and `x` is encoded in "M:Vm<3>". If `<dt>` is `S32` or `U32`, `Dm` is restricted to D0-D15. `Dm` is encoded in "Vm", and `x` is encoded in "M".

Operation for all encodings

```
 if ConditionPassed() then
     EncodingSpecificOperations();  CheckAdvSIMDEnabled();
     op2 = Elem[Din[m],index,esize];  op2val = Int(op2, unsigned);
     for r = 0 to regs-1
         for e = 0 to elements-1
             op1 = Elem[Din[n+r],e,esize];  op1val = Int(op1, unsigned);
             if floating_point then
                 fp_addend = if add then FPMul(op1,op2,StandardFPSCRValue()) else
FPNeg(FPMul(op1,op2,StandardFPSCRValue()));
                 Elem[D[d+r],e,esize] = FPAdd(Elem[Din[d+r],e,esize], fp_addend, StandardFPSCRValue());
             else
                 addend = if add then op1val*op2val else -op1val*op2val;
                 if long_destination then
                     Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
                 else
                     Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.126 VMLS (floating-point)

Vector Multiply Subtract multiplies corresponding elements in two vectors, subtracts the products from corresponding elements of the destination vector, and places the results in the destination vector.

Note

Arm recommends that software does not use the `VMLS` instruction in the Round towards Plus Infinity and Round towards Minus Infinity rounding modes, because the rounding of the product and of the sum can change the result of the instruction in opposite directions, defeating the purpose of these rounding modes.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	0	0	D	1	sz	Vn	Vd	1	1	0	1	N	Q	M	1	Vm
										op												

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMLS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMLS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
advsimd = TRUE;  add = (op == '0');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

A2

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	1	1	1	0	0	D	0	0	Vn	Vd	1	0	size		N	1	M	0	Vm
cond																op			

Half-precision scalar variant

Applies when `size == 01`.

```
VMLS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VMLS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Double-precision scalar variant

Applies when `size == 11`.

```
VMLS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE; add = (op == '0');
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	0	1	1	1	1	0	D	1	sz	Vn	Vd	1	1	0	1	N	Q	M	1	Vm
										op												

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMLS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMLS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
advsimd = TRUE;  add = (op == '0');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T2

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	0	1	1	1	0	0	D	0	0	Vn	Vd	1	0	size		N	1	M	0	Vm
																			op			

Half-precision scalar variant

Applies when `size == 01`.

```
VMLS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VMLS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VMLS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE; add = (op == '0');
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding A2, T1 and T2: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

`F32`	when `sz = 0`
`F16`	when `sz = 1`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

`<Sn>` Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

`<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

Operation for all encodings

```
 if ConditionPassed() then
     EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
     if advsimd then  // Advanced SIMD instruction
         for r = 0 to regs-1
             for e = 0 to elements-1
                 product = FPMul(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize], StandardFPSCRValue());
                 addend = if add then product else FPNeg(product);
                 Elem[D[d+r],e,esize] = FPAdd(Elem[D[d+r],e,esize], addend, StandardFPSCRValue());
     else             // VFP instruction
         case esize of
             when 16
                 addend16 = if add then FPMul(S[n]<15:0>, S[m]<15:0>, FPSCR) else FPNeg(FPMul(S[n]<15:0>,
S[m]<15:0>, FPSCR));
                 S[d] = Zeros(16) : FPAdd(S[d]<15:0>, addend16, FPSCR);
             when 32
                 addend32 = if add then FPMul(S[n], S[m], FPSCR) else FPNeg(FPMul(S[n], S[m], FPSCR));
                 S[d] = FPAdd(S[d], addend32, FPSCR);
             when 64
                 addend64 = if add then FPMul(D[n], D[m], FPSCR) else FPNeg(FPMul(D[n], D[m], FPSCR));
                 D[d] = FPAdd(D[d], addend64, FPSCR);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.127 VMLS (integer)

Vector Multiply Subtract multiplies corresponding elements in two vectors, and subtracts the products from the corresponding elements of the destination vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	1	0	D	size		Vn	Vd	1	0	0	1	N	Q	M	0	Vm
							op															

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMLS{<c>}{<q>}.<type><size> <Dd>, <Dn>, <Dm> // Encoding T1/A1, encoded as Q = 0
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMLS{<c>}{<q>}.<type><size> <Qd>, <Qn>, <Qm> // Encoding T1/A1, encoded as Q = 1
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
add = (op == '0');  long_destination = FALSE;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	1	1	0	D	size		Vn	Vd	1	0	0	1	N	Q	M	0	Vm
			op																			

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMLS{<c>}{<q>}.<type><size> <Dd>, <Dn>, <Dm> // Encoding T1/A1, encoded as Q = 0
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMLS{<c>}{<q>}.<type><size> <Qd>, <Qn>, <Qm> // Encoding T1/A1, encoded as Q = 1
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
add = (op == '0');  long_destination = FALSE;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<type>` The data type for the elements of the operands. It must be one of:

- `S` Optional in encoding T1/A1. Encoded as U = 0 in encoding T2/A2.
- `U` Optional in encoding T1/A1. Encoded as U = 1 in encoding T2/A2.
- `I` Available only in encoding T1/A1.

`<size>` The data size for the elements of the operands. It must be one of:

- `8` Encoded as size = `0b00`.
- `16` Encoded as size = `0b01`.
- `32` Encoded as size = `0b10`.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            product = Int(Elem[Din[n+r],e,esize],unsigned) * Int(Elem[Din[m+r],e,esize],unsigned);
            addend = if add then product else -product;
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
            else
                Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.128 VMLS (by scalar)

Vector Multiply Subtract multiplies elements of a vector by a scalar, and either subtracts the products from corresponding elements of the destination vector.

For more information about scalars see *Advanced SIMD scalars* on page F2-4106.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	Q	1	D	!=11 (size)		Vn	Vd	0	1 (op)	0	F	N	1	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMLS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMLS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>
```

Decode for all variants of this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || (F == '1' && size == '01' && !HaveFP16Ext()) then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
add = (op == '0');  floating_point = (F == '1');  long_destination = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	Q	1	1	1	1	1	D	!=11 (size)		Vn	Vd	0	1 (op)	0	F	N	1	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMLS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMLS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || (F == '1' && size == '01' && !HaveFP16Ext()) then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
add = (op == '0');  floating_point = (F == '1');  long_destination = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings

Related encodings: See *Advanced SIMD data-processing* on page F3-4137 for the T32 instruction set, or *Advanced SIMD data-processing* on page F4-4225 for the A32 instruction set.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the scalar and the elements of the operand vector, encoded in the "F:size" field. It can have the following values:

`I16`	when `F = 0`, `size = 01`
`I32`	when `F = 0`, `size = 10`
`F16`	when `F = 1`, `size = 01`
`F32`	when `F = 1`, `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm[x]>` Is the 64-bit name of the second SIMD&FP source register holding the scalar. If `<dt>` is `I16` or `F16`, `Dm` is restricted to D0-D7. `Dm` is encoded in "Vm<2:0>", and `x` is encoded in "M:Vm<3>". If `<dt>` is `I32` or `F32`, `Dm` is restricted to D0-D15. `Dm` is encoded in "Vm", and `x` is encoded in "M".

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    op2 = Elem[Din[m],index,esize];  op2val = Int(op2, unsigned);
    for r = 0 to regs-1
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];  op1val = Int(op1, unsigned);
            if floating_point then
                fp_addend = if add then FPMul(op1,op2,StandardFPSCRValue()) else
FPNeg(FPMul(op1,op2,StandardFPSCRValue()));
                Elem[D[d+r],e,esize] = FPAdd(Elem[Din[d+r],e,esize], fp_addend, StandardFPSCRValue());
            else
                addend = if add then op1val*op2val else -op1val*op2val;
                if long_destination then
                    Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
                else
                    Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.129 VMLSL (integer)

Vector Multiply Subtract Long multiplies corresponding elements in two vectors, and subtract the products from the corresponding elements of the destination vector. The destination vector element is twice as long as the elements that are multiplied.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	U	1	D	!=11		Vn	Vd	1	0	1	0	N	0	M	0	Vm
										size						op						

A1 variant

```
VMLSL{<c>}{<q>}.<type><size> <Qd>, <Dn>, <Dm> // Encoding T2/A2
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
add = (op == '0');  long_destination = TRUE;  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	U	1	1	1	1	1	D	!=11		Vn	Vd	1	0	1	0	N	0	M	0	Vm
										size						op						

T1 variant

```
VMLSL{<c>}{<q>}.<type><size> <Qd>, <Dn>, <Dm> // Encoding T2/A2
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
add = (op == '0');  long_destination = TRUE;  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;
```

Notes for all encodings

Related encodings: See *Advanced SIMD data-processing* on page F3-4137 for the T32 instruction set, or *Advanced SIMD data-processing* on page F4-4225 for the A32 instruction set.

Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<type>` The data type for the elements of the operands. It must be one of:

- `S` Optional in encoding T1/A1. Encoded as U = 0 in encoding T2/A2.
- `U` Optional in encoding T1/A1. Encoded as U = 1 in encoding T2/A2.
- `I` Available only in encoding T1/A1.

`<size>` The data size for the elements of the operands. It must be one of:

- `8` Encoded as size = `0b00`.
- `16` Encoded as size = `0b01`.
- `32` Encoded as size = `0b10`.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            product = Int(Elem[Din[n+r],e,esize],unsigned) * Int(Elem[Din[m+r],e,esize],unsigned);
            addend = if add then product else -product;
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
            else
                Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.130 VMLSL (by scalar)

Vector Multiply Subtract Long multiplies elements of a vector by a scalar, and subtracts the products from corresponding elements of the destination vector. The destination vector elements are twice as long as the elements that are multiplied.

For more information about scalars see *Advanced SIMD scalars* on page F2-4106.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	U	1	D	!=11 (size)		Vn	Vd	0	1 (op)	1	0	N	1	M	0	Vm

A1 variant

```
VMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  add = (op == '0');  floating_point = FALSE;  long_destination = TRUE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = 1;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	U	1	1	1	1	1	D	!=11 (size)		Vn	Vd	0	1 (op)	1	0	N	1	M	0	Vm

T1 variant

```
VMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  add = (op == '0');  floating_point = FALSE;  long_destination = TRUE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = 1;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

Notes for all encodings

Related encodings: See *Advanced SIMD data-processing* on page F3-4137 for the T32 instruction set, or *Advanced SIMD data-processing* on page F4-4225 for the A32 instruction set.

Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the scalar and the elements of the operand vector, encoded in the "U:size" field. It can have the following values:

- `S16` when `U = 0`, `size = 01`
- `S32` when `U = 0`, `size = 10`
- `U16` when `U = 1`, `size = 01`
- `U32` when `U = 1`, `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm[x]>` Is the 64-bit name of the second SIMD&FP source register holding the scalar. If `<dt>` is `S16` or `U16`, `Dm` is restricted to D0-D7. `Dm` is encoded in "Vm<2:0>", and `x` is encoded in "M:Vm<3>". If `<dt>` is `S32` or `U32`, `Dm` is restricted to D0-D15. `Dm` is encoded in "Vm", and `x` is encoded in "M".

Operation for all encodings

```
 if ConditionPassed() then
     EncodingSpecificOperations();  CheckAdvSIMDEnabled();
     op2 = Elem[Din[m],index,esize];  op2val = Int(op2, unsigned);
     for r = 0 to regs-1
         for e = 0 to elements-1
             op1 = Elem[Din[n+r],e,esize];  op1val = Int(op1, unsigned);
             if floating_point then
                 fp_addend = if add then FPMul(op1,op2,StandardFPSCRValue()) else
FPNeg(FPMul(op1,op2,StandardFPSCRValue()));
                 Elem[D[d+r],e,esize] = FPAdd(Elem[Din[d+r],e,esize], fp_addend, StandardFPSCRValue());
             else
                 addend = if add then op1val*op2val else -op1val*op2val;
                 if long_destination then
                     Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
                 else
                     Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.131 VMMLA

BFloat16 floating-point matrix multiply-accumulate. This instruction multiplies the 2x4 matrix of BF16 values in the first 128-bit source vector by the 4x2 BF16 matrix in the second 128-bit source vector. The resulting 2x2 single-precision matrix product is then added destructively to the 2x2 single-precision matrix in the 128-bit destination vector. This is equivalent to performing a 4-way dot product per destination element. The instruction does not update the FPSCR exception status.

Note

Arm expects that the VMMLA instruction will deliver a peak BF16 multiply throughput that is at least as high as can be achieved using two VDOT instructions, with a goal that it should have significantly higher throughput.

A1

ARMv8.6

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	0	0	0	D	0	0	Vn	Vd	1	1	0	0	N	1	M	0	Vm

A1 variant

```
VMMLA{<q>}.BF16 <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if !HaveAArch32BF16Ext() then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer regs = 2;
```

T1

ARMv8.6

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	0	0	0	D	0	0	Vn	Vd	1	1	0	0	N	1	M	0	Vm

T1 variant

```
VMMLA{<q>}.BF16 <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32BF16Ext() then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer regs = 2;
```

Assembler symbols

<q> See *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation for all encodings

```
CheckAdvSIMDEnabled();

bits(128) op1 = Q[n>>1];
bits(128) op2 = Q[m>>1];
bits(128) acc = Q[d>>1];

Q[d>>1] = BFMatMulAdd(acc, op1, op2);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.132 VMOV (between two general-purpose registers and a doubleword floating-point register)

Copy two general-purpose registers to or from a SIMD&FP register copies two words from two general-purpose registers into a doubleword register in the Advanced SIMD and floating-point register file, or from a doubleword register in the Advanced SIMD and floating-point register file to two general-purpose registers.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	1	1	0	0	0	1	0	op	Rt2	Rt	1	0	1	1	0	0	M	1	Vm
cond																			

From general-purpose registers variant

Applies when `op == 0`.

```
VMOV{<c>}{<q>} <Dm>, <Rt>, <Rt2>
```

To general-purpose registers variant

Applies when `op == 1`.

```
VMOV{<c>}{<q>} <Rt>, <Rt2>, <Dm>
```

Decode for all variants of this encoding

```
to_arm_registers = (op == '1');  t = UInt(Rt);  t2 = UInt(Rt2);  m = UInt(M:Vm);
if t == 15 || t2 == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if to_arm_registers && t == t2 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `to_arm_registers && t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	0	1	1	0	0	0	1	0	op	Rt2	Rt	1	0	1	1	0	0	M	1	Vm

From general-purpose registers variant

Applies when `op == 0`.

```
VMOV{<c>}{<q>} <Dm>, <Rt>, <Rt2>
```

To general-purpose registers variant

Applies when `op == 1`.

```
VMOV{<c>}{<q>} <Rt>, <Rt2>, <Dm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
to_arm_registers = (op == '1');  t = UInt(Rt);  t2 = UInt(Rt2);  m = UInt(M:Vm);
if t == 15 || t2 == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if to_arm_registers && t == t2 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `to_arm_registers && t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *VMOV (between two general-purpose registers and a doubleword floating-point register)* on page K1-7848.

Assembler symbols

`<Dm>`	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "M:Vm" field.
`<Rt2>`	Is the second general-purpose register that `<Dm>`[63:32] will be transferred to or from, encoded in the "Rt2" field.
`<Rt>`	Is the first general-purpose register that `<Dm>`[31:0] will be transferred to or from, encoded in the "Rt" field.
`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_arm_registers then
        R[t] = D[m]<31:0>;
        R[t2] = D[m]<63:32>;
    else
        D[m]<31:0> = R[t];
        D[m]<63:32> = R[t2];
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.133 VMOV (between general-purpose register and half-precision)

Copy 16 bits of a general-purpose register to or from a 32-bit SIMD&FP register. This instruction transfers the value held in the bottom 16 bits of a 32-bit SIMD&FP register to the bottom 16 bits of a general-purpose register, or the value held in the bottom 16 bits of a general-purpose register to the bottom 16 bits of a 32-bit SIMD&FP register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

ARMv8.2

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3	2	1	0
!=1111	1	1	1	0	0	0	0	op	Vn	Rt	1	0	0	1	N	(0)	(0)	1	(0)	(0)	(0)	(0)
cond																						

From general-purpose register variant

Applies when `op == 0`.

```
VMOV{<c>}{<q>}.F16 <Sn>, <Rt>
```

To general-purpose register variant

Applies when `op == 1`.

```
VMOV{<c>}{<q>}.F16 <Rt>, <Sn>
```

Decode for all variants of this encoding

```
if !HaveFP16Ext() then UNDEFINED;
if cond != '1110' then UNPREDICTABLE;
to_arm_register = (op == '1');  t = UInt(Rt);  n = UInt(Vn:N);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

ARMv8.2

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	0	0	0	op	Vn	Rt	1	0	0	1	N	(0)	(0)	1	(0)	(0)	(0)	(0)

From general-purpose register variant

Applies when `op == 0`.

```
VMOV{<c>}{<q>}.F16 <Sn>, <Rt>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

To general-purpose register variant

Applies when `op == 1`.

```
VMOV{<c>}{<q>}.F16 <Rt>, <Sn>
```

Decode for all variants of this encoding

```
if !HaveFP16Ext() then UNDEFINED;
if InITBlock() then UNPREDICTABLE;
to_arm_register = (op == '1');  t = UInt(Rt);  n = UInt(Vn:N);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<Rt>`	Is the general-purpose register that `<Sn>` will be transferred to or from, encoded in the "Rt" field.
`<Sn>`	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Vn:N" field.
`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_arm_register then
        R[t] = Zeros(16) : S[n]<15:0>;
    else
        S[n] = Zeros(16) : R[t]<15:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.134 VMOV (immediate)

Copy immediate value to a SIMD&FP register places an immediate constant into every element of the destination register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	16	15	12	11	8	7	6	5	4	3	0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3		Vd		0 x x 0		0	Q	0	1	imm4	
																	cmode				op			

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMOV{<c>}{<q>}.I32 <Dd>, #<imm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMOV{<c>}{<q>}.I32 <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if op == '0' && cmode<0> == '1' && cmode<3:2> != '11' then SEE "VORR (immediate)";
if op == '1' && cmode != '1110' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
single_register = FALSE;  advsimd = TRUE;  imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

A2

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	10	9	8	7	6	5	4	3	0
!=1111		1	1	1	0	1	D	1	1	imm4H		Vd		1	0	size		(0)	0	(0)	0	imm4L	
cond																							

Half-precision scalar variant

Applies when `size == 01`.

```
VMOV{<c>}{<q>}.F16 <Sd>, #<imm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VMOV{<c>}{<q>}.F32 <Sd>, #<imm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VMOV{<c>}{<q>}.F64 <Dd>, #<imm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
single_register = (size != '11'); advsimd = FALSE;
bits(16) imm16;
bits(32) imm32;
bits(64) imm64;
case size of
    when '01' d = UInt(Vd:D);  imm16 = VFPExpandImm(imm4H:imm4L); imm32 = Zeros(16) : imm16;
    when '10' d = UInt(Vd:D);  imm32 = VFPExpandImm(imm4H:imm4L);
    when '11' d = UInt(D:Vd);  imm64 = VFPExpandImm(imm4H:imm4L);  regs = 1;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

A3

31	30	29	28	27	26	25	24	23	22	21	20	19 18	16	15 12	11 8	7	6	5	4	3 0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3	Vd	1 0 x 0	0	Q	0	1	imm4
															cmode			op		

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMOV{<c>}{<q>}.I16 <Dd>, #<imm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMOV{<c>}{<q>}.I16 <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if op == '0' && cmode<0> == '1' && cmode<3:2> != '11' then SEE "VORR (immediate)";
if op == '1' && cmode != '1110' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
single_register = FALSE;  advsimd = TRUE;  imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

A4

31	30	29	28	27	26	25	24	23	22	21	20	19 18	16	15 12	11 8	7	6	5	4	3 0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3	Vd	1 1 x x	0	Q	0	1	imm4
															cmode			op		

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMOV{<c>}{<q>}.<dt> <Dd>, #<imm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMOV{<c>}{<q>}.<dt> <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if op == '0' && cmode<0> == '1' && cmode<3:2> != '11' then SEE "VORR (immediate)";
if op == '1' && cmode != '1110' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
single_register = FALSE;  advsimd = TRUE;  imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

A5

31 30 29 28	27 26 25 24	23	22	21 20	19 18 16	15 12	11 8	7	6	5	4	3 0
1 1 1 1	0 0 1 i	1	D	0 0 0	imm3	Vd	1 1 1 0	0	Q	1	1	imm4
							cmode			op		

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMOV{<c>}{<q>}.I64 <Dd>, #<imm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMOV{<c>}{<q>}.I64 <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if op == '0' && cmode<0> == '1' && cmode<3:2> != '11' then SEE "VORR (immediate)";
if op == '1' && cmode != '1110' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
single_register = FALSE;  advsimd = TRUE;  imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13	12	11 10 9 8	7	6	5 4	3 2 0	15 12	11 8	7	6	5	4	3 0
1 1 1	i	1 1 1 1	1	D	0 0	imm3	Vd	0 x x 0	0	Q	0	1	imm4
								cmode			op		

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMOV{<c>}{<q>}.I32 <Dd>, #<imm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMOV{<c>}{<q>}.I32 <Qd>, #<imm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if op == '0' && cmode<0> == '1' && cmode<3:2> != '11' then SEE "VORR (immediate)";
if op == '1' && cmode != '1110' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
single_register = FALSE;  advsimd = TRUE;  imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	0	1	1	1	0	1	D	1	1	imm4H	Vd	1	0	size		(0)	0	(0)	0	imm4L

Half-precision scalar variant

Applies when `size == 01`.

```
VMOV{<c>}{<q>}.F16 <Sd>, #<imm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VMOV{<c>}{<q>}.F32 <Sd>, #<imm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VMOV{<c>}{<q>}.F64 <Dd>, #<imm>
```

Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
single_register = (size != '11'); advsimd = FALSE;
bits(16) imm16;
bits(32) imm32;
bits(64) imm64;
case size of
    when '01' d = UInt(Vd:D);  imm16 = VFPExpandImm(imm4H:imm4L); imm32 = Zeros(16) : imm16;
    when '10' d = UInt(Vd:D);  imm32 = VFPExpandImm(imm4H:imm4L);
    when '11' d = UInt(D:Vd);  imm64 = VFPExpandImm(imm4H:imm4L);  regs = 1;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3	Vd	1	0	x	0	0	Q	0	1	imm4
															cmode						op		

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMOV{<c>}{<q>}.I16 <Dd>, #<imm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMOV{<c>}{<q>}.I16 <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if op == '0' && cmode<0> == '1' && cmode<3:2> != '11' then SEE "VORR (immediate)";
if op == '1' && cmode != '1110' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
single_register = FALSE;  advsimd = TRUE;  imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

T4

15	14	13	12	11	10	9	8	7	6	5	4	3	2 – 0	15 – 12	11 – 8	7	6	5	4	3 – 0
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3	Vd	1 1 x x (cmode)	0	Q	0 (op)	1	imm4

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMOV{<c>}{<q>}.<dt> <Dd>, #<imm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMOV{<c>}{<q>}.<dt> <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if op == '0' && cmode<0> == '1' && cmode<3:2> != '11' then SEE "VORR (immediate)";
if op == '1' && cmode != '1110' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
single_register = FALSE;  advsimd = TRUE;  imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

T5

15	14	13	12	11	10	9	8	7	6	5	4	3	2 – 0	15 – 12	11 – 8	7	6	5	4	3 – 0
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3	Vd	1 1 1 0 (cmode)	0	Q	1 (op)	1	imm4

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMOV{<c>}{<q>}.I64 <Dd>, #<imm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VMOV{<c>}{<q>}.I64 <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if op == '0' && cmode<0> == '1' && cmode<3:2> != '11' then SEE "VORR (immediate)";
if op == '1' && cmode != '1110' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
single_register = FALSE;  advsimd = TRUE;  imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

Notes for all encodings

Related encodings: See *Advanced SIMD one register and modified immediate* on page F3-4146 for the T32 instruction set, or *Advanced SIMD one register and modified immediate* on page F4-4233 for the A32 instruction set.

Assembler symbols

`<c>` For encoding A1, A3, A4 and A5: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding A2, T1, T2, T3, T4 and T5: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` The data type, encoded in the "cmode" field. It can have the following values:

`I32`	when `cmode = 110x`
`I8`	when `cmode = 1110`
`F32`	when `cmode = 1111`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

`<imm>` For encoding A1, A3, A4, A5, T1, T3, T4 and T5: is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of `<imm>`, see *Modified immediate constants in T32 and A32 Advanced SIMD instructions* on page F2-4097.

For encoding A2 and T2: is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in "imm4H:imm4L". For details of the range of constants available and the encoding of `<imm>`, see *Modified immediate constants in T32 and A32 floating-point instructions* on page F2-4098.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if single_register then
        S[d] = imm32;
    else
        for r = 0 to regs-1
            D[d+r] = imm64;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.135 VMOV (register)

Copy between FP registers copies the contents of one FP register to another.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A2

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	1	1	1	0	1	D	1	1	0	0	0	0	Vd	1	0	1	x	0	1	M	0	Vm
cond																size						

Single-precision scalar variant

Applies when `size == 10`.

```
VMOV{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VMOV{<c>}{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
single_register = (size == '10');  advsimd = FALSE;
if single_register then
    d = UInt(Vd:D);  m = UInt(Vm:M);
else
    d = UInt(D:Vd);  m = UInt(M:Vm);  regs = 1;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	0	1	1	1	0	1	D	1	1	0	0	0	0	Vd	1	0	1	x	0	1	M	0	Vm
																			size						

Single-precision scalar variant

Applies when `size == 10`.

```
VMOV{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VMOV{<c>}{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
single_register = (size == '10');  advsimd = FALSE;
if single_register then
    d = UInt(Vd:D);  m = UInt(Vm:M);
else
    d = UInt(D:Vd);  m = UInt(M:Vm);  regs = 1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Sd>`	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
`<Sm>`	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
`<Dm>`	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if single_register then
        S[d] = S[m];
    else
        for r = 0 to regs-1
            D[d+r] = D[m+r];
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.136 VMOV (register, SIMD)

Copy between SIMD registers copies the contents of one SIMD register to another

This instruction is an alias of the VORR (register) instruction. This means that:

- The encodings in this description are named to match the encodings of VORR (register).
- The description of VORR (register) gives the operational pseudocode for this instruction.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 0 1	0	0	D	1	0	Vn	Vd	0 0 0 1	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMOV{<c>}{<q>}{.<dt>} <Dd>, <Dm>
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>} <Dd>, <Dm>, <Dm>
```

and is the preferred disassembly when `N:Vn == M:Vm`.

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMOV{<c>}{<q>}{.<dt>} <Qd>, <Qm>
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>} <Qd>, <Qm>, <Qm>
```

and is the preferred disassembly when `N:Vn == M:Vm`.

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 0	1 1 1 1 0	D	1	0	Vn	Vd	0 0 0 1	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMOV{<c>}{<q>}{.<dt>} <Dd>, <Dm>
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>} <Dd>, <Dm>, <Dm>
```

and is the preferred disassembly when `N:Vn == M:Vm`.

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMOV{<c>}{<q>}{.<dt>} <Qd>, <Qm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

is equivalent to

```
VORR{<c>}{<q>}{.<dt>} <Qd>, <Qm>, <Qm>
```

and is the preferred disassembly when `N:Vn == M:Vm`.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` An optional data type. `<dt>` must not be `F64`, but it is otherwise ignored.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field.

Operation for all encodings

The description of VORR (register) gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.137 VMOV (general-purpose register to scalar)

Copy a general-purpose register to a vector element copies a byte, halfword, or word from a general-purpose register into an Advanced SIMD scalar.

On a Floating-point-only system, this instruction transfers one word to the upper or lower half of a double-precision floating-point register from a general-purpose register. This is an identical operation to the Advanced SIMD single word transfer.

For more information about scalars see *Advanced SIMD scalars* on page F2-4106.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31–28	27	26	25	24	23	22–21	20	19–16	15–12	11	10	9	8	7	6–5	4	3	2	1	0
!=1111	1	1	1	0	0	opc1	0	Vd	Rt	1	0	1	1	D	opc2	1	(0)	(0)	(0)	(0)
cond																				

A1 variant

```
VMOV{<c>}{<q>}{.<size>} <Dd[x]>, <Rt>
```

Decode for this encoding

```
case opc1:opc2 of
    when '1xxx'  advsimd = TRUE;  esize = 8;  index = UInt(opc1<0>:opc2);
    when '0xx1'  advsimd = TRUE;  esize = 16;  index = UInt(opc1<0>:opc2<1>);
    when '0x00'  advsimd = FALSE;  esize = 32;  index = UInt(opc1<0>);
    when '0x10'  UNDEFINED;
d = UInt(D:Vd);  t = UInt(Rt);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

T1

15	14	13	12	11	10	9	8	7	6–5	4	3–0	15–12	11	10	9	8	7	6–5	4	3	2	1	0
1	1	1	0	1	1	1	0	0	opc1	0	Vd	Rt	1	0	1	1	D	opc2	1	(0)	(0)	(0)	(0)

T1 variant

```
VMOV{<c>}{<q>}{.<size>} <Dd[x]>, <Rt>
```

Decode for this encoding

```
case opc1:opc2 of
    when '1xxx'  advsimd = TRUE;  esize = 8;  index = UInt(opc1<0>:opc2);
    when '0xx1'  advsimd = TRUE;  esize = 16;  index = UInt(opc1<0>:opc2<1>);
    when '0x00'  advsimd = FALSE;  esize = 32;  index = UInt(opc1<0>);
    when '0x10'  UNDEFINED;
d = UInt(D:Vd);  t = UInt(Rt);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<size>` The data size. It must be one of:

- `8` Encoded as `opc1<1>` = 1. `[x]` is encoded in `opc1<0>`, `opc2`.
- `16` Encoded as `opc1<1>` = 0, `opc2<0>` = 1. `[x]` is encoded in `opc1<0>`, `opc2<1>`.
- `32` Encoded as `opc1<1>` = 0, `opc2` = `0b00`. `[x]` is encoded in `opc1<0>`.
- `omitted` Equivalent to `32`.

`<Dd[x]>` The scalar. The register `<Dd>` is encoded in D:Vd. For details of how `[x]` is encoded, see the description of `<size>`.

`<Rt>` The source general-purpose register.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    Elem[D[d],index,esize] = R[t]<esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.138 VMOV (between general-purpose register and single-precision)

Copy a general-purpose register to or from a 32-bit SIMD&FP register. This instruction transfers the value held in a 32-bit SIMD&FP register to a general-purpose register, or the value held in a general-purpose register to a 32-bit SIMD&FP register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3	2	1	0
!=1111 (cond)	1	1	1	0	0	0	0	op	Vn	Rt	1	0	1	0	N	(0)	(0)	1	(0)	(0)	(0)	(0)

From general-purpose register variant

Applies when `op == 0`.

```
VMOV{<c>}{<q>} <Sn>, <Rt>
```

To general-purpose register variant

Applies when `op == 1`.

```
VMOV{<c>}{<q>} <Rt>, <Sn>
```

Decode for all variants of this encoding

```
to_arm_register = (op == '1');  t = UInt(Rt);  n = UInt(Vn:N);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	0	0	0	op	Vn	Rt	1	0	1	0	N	(0)	(0)	1	(0)	(0)	(0)	(0)

From general-purpose register variant

Applies when `op == 0`.

```
VMOV{<c>}{<q>} <Sn>, <Rt>
```

To general-purpose register variant

Applies when `op == 1`.

```
VMOV{<c>}{<q>} <Rt>, <Sn>
```

Decode for all variants of this encoding

```
to_arm_register = (op == '1');  t = UInt(Rt);  n = UInt(Vn:N);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<Rt>`	Is the general-purpose register that `<Sn>` will be transferred to or from, encoded in the "Rt" field.
`<Sn>`	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Vn:N" field.
`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_arm_register then
        R[t] = S[n];
    else
        S[n] = R[t];
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.139 VMOV (scalar to general-purpose register)

Copy a vector element to a general-purpose register with sign or zero extension copies a byte, halfword, or word from an Advanced SIMD scalar to a general-purpose register. Bytes and halfwords can be either zero-extended or sign-extended.

On a Floating-point-only system, this instruction transfers one word from the upper or lower half of a double-precision floating-point register to a general-purpose register. This is an identical operation to the Advanced SIMD single word transfer.

For more information about scalars see *Advanced SIMD scalars* on page F2-4106.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31–28	27	26	25	24	23	22–21	20	19–16	15–12	11	10	9	8	7	6–5	4	3	2	1	0
!=1111 (cond)	1	1	1	0	U	opc1	1	Vn	Rt	1	0	1	1	N	opc2	1	(0)	(0)	(0)	(0)

A1 variant

```
VMOV{<c>}{<q>}{.<dt>} <Rt>, <Dn[x]>
```

Decode for this encoding

```
case U:opc1:opc2 of
    when 'x1xxx'  advsimd = TRUE;  esize = 8;  index = UInt(opc1<0>:opc2);
    when 'x0xx1'  advsimd = TRUE;  esize = 16;  index = UInt(opc1<0>:opc2<1>);
    when '00x00'  advsimd = FALSE;  esize = 32;  index = UInt(opc1<0>);
    when '10x00'  UNDEFINED;
    when 'x0x10'  UNDEFINED;
t = UInt(Rt);  n = UInt(N:Vn);  unsigned = (U == '1');
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

T1

15	14	13	12	11	10	9	8	7	6–5	4	3–0	15–12	11	10	9	8	7	6–5	4	3	2	1	0
1	1	1	0	1	1	1	0	U	opc1	1	Vn	Rt	1	0	1	1	N	opc2	1	(0)	(0)	(0)	(0)

T1 variant

```
VMOV{<c>}{<q>}{.<dt>} <Rt>, <Dn[x]>
```

Decode for this encoding

```
case U:opc1:opc2 of
    when 'x1xxx'  advsimd = TRUE;  esize = 8;  index = UInt(opc1<0>:opc2);
    when 'x0xx1'  advsimd = TRUE;  esize = 16;  index = UInt(opc1<0>:opc2<1>);
    when '00x00'  advsimd = FALSE;  esize = 32;  index = UInt(opc1<0>);
    when '10x00'  UNDEFINED;
    when 'x0x10'  UNDEFINED;
t = UInt(Rt);  n = UInt(N:Vn);  unsigned = (U == '1');
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` The data type. It must be one of:

- `S8` Encoded as U = 0, `opc1<1>` = 1. `[x]` is encoded in `opc1<0>`, `opc2`.
- `S16` Encoded as U = 0, `opc1<1>` = 0, `opc2<0>` = 1. `[x]` is encoded in `opc1<0>`, `opc2<1>`.
- `U8` Encoded as U = 1, `opc1<1>` = 1. `[x]` is encoded in `opc1<0>`, `opc2`.
- `U16` Encoded as U = 1, `opc1<1>` = 0, `opc2<0>` = 1. `[x]` is encoded in `opc1<0>`, `opc2<1>`.
- `32` Encoded as U = 0, `opc1<1>` = 0, `opc2` = `0b00`. `[x]` is encoded in `opc1<0>`.
- `omitted` Equivalent to `32`.

`<Rt>` The destination general-purpose register.

`<Dn[x]>` The scalar. For details of how `[x]` is encoded see the description of `<dt>`.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if unsigned then
        R[t] = ZeroExtend(Elem[D[n],index,esize], 32);
    else
        R[t] = SignExtend(Elem[D[n],index,esize], 32);
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.140 VMOV (between two general-purpose registers and two single-precision registers)

Copy two general-purpose registers to a pair of 32-bit SIMD&FP registers transfers the contents of two consecutively numbered single-precision Floating-point registers to two general-purpose registers, or the contents of two general-purpose registers to a pair of single-precision Floating-point registers. The general-purpose registers do not have to be contiguous.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	1	1	0	0	0	1	0	op	Rt2	Rt	1	0	1	0	0	0	M	1	Vm
cond																			

From general-purpose registers variant

Applies when `op == 0`.

```
VMOV{<c>}{<q>} <Sm>, <Sm1>, <Rt>, <Rt2>
```

To general-purpose registers variant

Applies when `op == 1`.

```
VMOV{<c>}{<q>} <Rt>, <Rt2>, <Sm>, <Sm1>
```

Decode for all variants of this encoding

```
to_arm_registers = (op == '1');  t = UInt(Rt);  t2 = UInt(Rt2);  m = UInt(Vm:M);
if t == 15 || t2 == 15 || m == 31 then UNPREDICTABLE;
if to_arm_registers && t == t2 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `to_arm_registers && t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

If `m == 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the single-precision registers become UNKNOWN for a move to the single-precision register. The general-purpose registers listed in the instruction become UNKNOWN for a move from the single-precision registers. This behavior does not affect any other general-purpose registers.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	0	1	1	0	0	0	1	0	op	Rt2	Rt	1	0	1	0	0	0	M	1	Vm

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

From general-purpose registers variant

Applies when `op == 0`.

```
VMOV{<c>}{<q>} <Sm>, <Sm1>, <Rt>, <Rt2>
```

To general-purpose registers variant

Applies when `op == 1`.

```
VMOV{<c>}{<q>} <Rt>, <Rt2>, <Sm>, <Sm1>
```

Decode for all variants of this encoding

```
to_arm_registers = (op == '1');  t = UInt(Rt);  t2 = UInt(Rt2);  m = UInt(Vm:M);
if t == 15 || t2 == 15 || m == 31 then UNPREDICTABLE;
if to_arm_registers && t == t2 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `to_arm_registers && t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

If `m == 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the single-precision registers become UNKNOWN for a move to the single-precision register. The general-purpose registers listed in the instruction become UNKNOWN for a move from the single-precision registers. This behavior does not affect any other general-purpose registers.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *VMOV (between two general-purpose registers and two single-precision registers)* on page K1-7848.

Assembler symbols

`<Rt2>`	Is the second general-purpose register that `<Sm1>` will be transferred to or from, encoded in the "Rt2" field.
`<Rt>`	Is the first general-purpose register that `<Sm>` will be transferred to or from, encoded in the "Rt" field.
`<Sm1>`	Is the 32-bit name of the second SIMD&FP register to be transferred. This is the next SIMD&FP register after `<Sm>`.
`<Sm>`	Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Vm:M" field.
`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_arm_registers then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        R[t] = S[m];
        R[t2] = S[m+1];
    else
        S[m] = R[t];
        S[m+1] = R[t2];
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.141 VMOVL

Vector Move Long takes each element in a doubleword vector, sign or zero-extends them to twice their original length, and places the results in a quadword vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	19 18 17 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 U	1	D	!=000	0 0 0	Vd	1 0 1 0	0	0	M	1	Vm
				imm3H								

A1 variant

```
VMOVL{<c>}{<q>}.<dt> <Qd>, <Dm>
```

Decode for this encoding

```
if imm3H == '000' then SEE "Related encodings";
if imm3H != '001' && imm3H != '010' && imm3H != '100' then SEE "VSHLL";
if Vd<0> == '1' then UNDEFINED;
esize = 8 * UInt(imm3H);
unsigned = (U == '1');  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);
```

T1

15 14 13 12	11 10 9 8	7	6	5	3 2 1 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 U	1 1 1 1	1	D	!=000	0 0 0	Vd	1 0 1 0	0	0	M	1	Vm
				imm3H								

T1 variant

```
VMOVL{<c>}{<q>}.<dt> <Qd>, <Dm>
```

Decode for this encoding

```
if imm3H == '000' then SEE "Related encodings";
if imm3H != '001' && imm3H != '010' && imm3H != '100' then SEE "VSHLL";
if Vd<0> == '1' then UNDEFINED;
esize = 8 * UInt(imm3H);
unsigned = (U == '1');  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);
```

Notes for all encodings

Related encodings: See *Advanced SIMD one register and modified immediate* on page F3-4146 for the T32 instruction set, or *Advanced SIMD one register and modified immediate* on page F4-4233 for the A32 instruction set.

Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the operand, encoded in the "U:imm3H" field. It can have the following values:

`S8`	when `U = 0`, `imm3H = 001`
`S16`	when `U = 0`, `imm3H = 010`
`S32`	when `U = 0`, `imm3H = 100`
`U8`	when `U = 1`, `imm3H = 001`
`U16`	when `U = 1`, `imm3H = 010`
`U32`	when `U = 1`, `imm3H = 100`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        result = Int(Elem[Din[m],e,esize], unsigned);
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.142 VMOVN

Vector Move and Narrow copies the least significant half of each element of a quadword vector into the corresponding elements of a doubleword vector.

The operand vector elements can be any one of 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

This instruction is used by the pseudo-instructions VRSHRN (zero) and VSHRN (zero). The pseudo-instruction is never the preferred disassembly.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 18	17 16	15 12	11	10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	1	1	size	1	0	Vd	0	0 1 0	0	0	M	0	Vm

A1 variant

```
VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

Decode for this encoding

```
if size == '11' then UNDEFINED;
if Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 2	1	0	15 12	11	10 9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	1	D	1	1	size	1	0	Vd	0	0 1 0	0	0	M	0	Vm

T1 variant

```
VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

Decode for this encoding

```
if size == '11' then UNDEFINED;
if Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);
```

Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<dt>` Is the data type for the elements of the operand, encoded in the "size" field. It can have the following values:

`I16` when `size = 00`

`I32` when `size = 01`

`I64` when `size = 10`

The encoding `size = 11` is reserved.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        Elem[D[d],e,esize] = Elem[Qin[m>>1],e,2*esize]<esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.143 VMOVX

Vector Move extraction. This instruction copies the upper 16 bits of the 32-bit source SIMD&FP register into the lower 16 bits of the 32-bit destination SIMD&FP register, while clearing the remaining bits to zero.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15 – 12	11	10	9	8	7	6	5	4	3 – 0
1	1	1	1	1	1	1	0	1	D	1	1	0	0	0	0	Vd	1	0	1	0	0	1	M	0	Vm

A1 variant

```
VMOVX{<q>}.F16 <Sd>, <Sm>
```

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
d = UInt(Vd:D); m = UInt(Vm:M);
```

T1

ARMv8.2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15 – 12	11	10	9	8	7	6	5	4	3 – 0
1	1	1	1	1	1	1	0	1	D	1	1	0	0	0	0	Vd	1	0	1	0	0	1	M	0	Vm

T1 variant

```
VMOVX{<q>}.F16 <Sd>, <Sm>
```

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveFP16Ext() then UNDEFINED;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
d = UInt(Vd:D); m = UInt(Vm:M);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

<q>	See *Standard assembler syntax fields* on page F2-4080.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Sm>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    S[d] = Zeros(16) : S[m]<31:16>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.144 VMRS

Move SIMD&FP Special register to general-purpose register moves the value of an Advanced SIMD and floating-point System register to a general-purpose register. When the specified System register is the FPSCR, a form of the instruction transfers the FPSCR.{N, Z, C, V} condition flags to the APSR.{N, Z, C, V} condition flags.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

When these settings permit the execution of floating-point and Advanced SIMD instructions, if the specified floating-point System register is not the FPSCR, the instruction is UNDEFINED if executed in User mode.

In an implementation that includes EL2, when HCR.TID0 is set to 1, any `VMRS` access to FPSID from a Non-secure EL1 mode that would be permitted if HCR.TID0 was set to 0 generates a Hyp Trap exception. For more information, see *ID group 0, Primary device identification registers* on page G1-5795.

For simplicity, the `VMRS` pseudocode does not show the possible trap to Hyp mode.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3	2	1	0
!=1111	1	1	1	0	1	1	1	1	reg	Rt	1	0	1	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)
cond																						

A1 variant

```
VMRS{<c>}{<q>} <Rt>, <spec_reg>
```

Decode for this encoding

```
t = UInt(Rt);
if !(reg IN {'000x', '0101', '011x', '1000'}) then UNPREDICTABLE;
if t == 15 && reg != '0001' then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `!(reg IN {'000x', '0101', '011x', '1000'})`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction transfers an UNKNOWN value to the specified target register. When the Rt field holds the value `0b1111`, the specified target register is the APSR.{N, Z, C, V} bits, and these bits become UNKNOWN. Otherwise, the specified target register is the register specified by the `Rt` field, R0 - R14.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	1	1	1	reg	Rt	1	0	1	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)

T1 variant

```
VMRS{<c>}{<q>} <Rt>, <spec_reg>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for this encoding

```
t = UInt(Rt);
if !(reg IN {'000x', '0101', '011x', '1000'}) then UNPREDICTABLE;
if t == 15 && reg != '0001' then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `!(reg IN {'000x', '0101', '011x', '1000'})`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction transfers an UNKNOWN value to the specified target register. When the Rt field holds the value `0b1111`, the specified target register is the APSR.{N, Z, C, V} bits, and these bits become UNKNOWN. Otherwise, the specified target register is the register specified by the `Rt` field, R0 - R14.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Rt>` Is the general-purpose destination register, encoded in the "Rt" field. Is one of:

`R0-R14`	General-purpose register.
`APSR_nzcv`	Permitted only when `<spec_reg>` is `FPSCR`. Encoded as `0b1111`. The instruction transfers the FPSCR.{N, Z, C, V} condition flags to the APSR.{N, Z, C, V} condition flags.

`<spec_reg>` Is the source Advanced SIMD and floating-point System register, encoded in the "reg" field. It can have the following values:

`FPSID`	when `reg = 0000`
`FPSCR`	when `reg = 0001`
`MVFR2`	when `reg = 0101`
`MVFR1`	when `reg = 0110`
`MVFR0`	when `reg = 0111`
`FPEXC`	when `reg = 1000`

The following encodings are UNPREDICTABLE:

- `reg = 001x`.
- `reg = 0100`.
- `reg = 1001`.
- `reg = 101x`.
- `reg = 11xx`.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if reg == '0001' then                // FPSCR
        CheckVFPEnabled(TRUE);
        if t == 15 then
            PSTATE.<N,Z,C,V> = FPSR.<N,Z,C,V>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        R[t] = FPSCR;
elsif PSTATE.EL == EL0 then
    UNDEFINED;                        // Non-FPSCR registers accessible only at PL1 or above
else
    CheckVFPEnabled(FALSE);           // Non-FPSCR registers are not affected by FPEXC.EN
    AArch32.CheckAdvSIMDOrFPRegisterTraps(reg);
    case reg of
        when '0000'  R[t] = FPSID;
        when '0101'  R[t] = MVFR2;
        when '0110'  R[t] = MVFR1;
        when '0111'  R[t] = MVFR0;
        when '1000'  R[t] = FPEXC;
        otherwise    Unreachable();   // Dealt with above or in encoding-specific pseudocode
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.145 VMSR

Move general-purpose register to SIMD&FP Special register moves the value of a general-purpose register to a floating-point System register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

When these settings permit the execution of floating-point and Advanced SIMD instructions:

- If the specified floating-point System register is FPSID or FPEXC, the instruction is UNDEFINED if executed in User mode.
- If the specified floating-point System register is the FPSID and the instruction is executed in a mode other than User mode, the instruction is ignored.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3	2	1	0
!=1111	1	1	1	0	1	1	1	0	reg	Rt	1	0	1	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)
cond																						

A1 variant

```
VMSR{<c>}{<q>} <spec_reg>, <Rt>
```

Decode for this encoding

```
t = UInt(Rt);
if reg != '000x' && reg != '1000' then UNPREDICTABLE;
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `reg != '000x' && reg != '1000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction transfers the value in the general-purpose register to one of the allocated registers accessible using `VMSR` at the same Exception level.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	1	1	0	reg	Rt	1	0	1	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)

T1 variant

```
VMSR{<c>}{<q>} <spec_reg>, <Rt>
```

Decode for this encoding

```
t = UInt(Rt);
if reg != '000x' && reg != '1000' then UNPREDICTABLE;
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CONSTRAINED UNPREDICTABLE behavior

If `reg != '000x' && reg != '1000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction transfers the value in the general-purpose register to one of the allocated registers accessible using `VMSR` at the same Exception level.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<spec_reg>` Is the destination Advanced SIMD and floating-point System register, encoded in the "reg" field. It can have the following values:

`FPSID`	when `reg = 0000`
`FPSCR`	when `reg = 0001`
`FPEXC`	when `reg = 1000`

The following encodings are UNPREDICTABLE:

- `reg = 001x`.
- `reg = 01xx`.
- `reg = 1001`.
- `reg = 101x`.
- `reg = 11xx`.

`<Rt>` Is the general-purpose source register, encoded in the "Rt" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if reg == '0001' then                // FPSCR
        CheckVFPEnabled(TRUE);
        FPSCR = R[t];
    elsif PSTATE.EL == EL0 then
        UNDEFINED;                       // Non-FPSCR registers accessible only at PL1 or above
    else
        CheckVFPEnabled(FALSE);          // Non-FPSCR registers are not affected by FPEXC.EN
        case reg of
            when '0000'                  // VMSR access to FPSID is ignored
            when '1000'  FPEXC = R[t];
            otherwise    Unreachable();  // Dealt with above or in encoding-specific pseudocode
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.146 VMUL (floating-point)

Vector Multiply multiplies corresponding elements in two vectors, and places the results in the destination vector.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	1	0	D	0	sz	Vn	Vd	1	1	0	1	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMUL{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMUL{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
advsimd = TRUE;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

A2

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9–8	7	6	5	4	3–0
!=1111	1	1	1	0	0	D	1	0	Vn	Vd	1	0	size	N	0	M	0	Vm
cond																		

Half-precision scalar variant

Applies when `size == 01`.

```
VMUL{<c>}{<q>}.F16 {<Sd>,} <Sn>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VMUL{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VMUL{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
advsimd = FALSE;

case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	1	1	1	1	1	0	D	0	sz	Vn	Vd	1	1	0	1	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMUL{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMUL{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if sz == '1' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
advsimd = TRUE;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T2

15 14 13 12	11 10 9 8	7	6	5 4	3 0	15 12	11 10	9 8	7	6	5	4	3 0
1 1 1 0	1 1 1 0	0	D	1 0	Vn	Vd	1 0	size	N	0	M	0	Vm

Half-precision scalar variant

Applies when `size == 01`.

```
VMUL{<c>}{<q>}.F16 {<Sd>,} <Sn>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VMUL{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VMUL{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if size == '01' && InITBlock()  then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
advsimd = FALSE;

case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding A2, T1 and T2: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

`F32`	when `sz = 0`
`F16`	when `sz = 1`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

`<Sn>` Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

`<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

Operation for all encodings

```
 if ConditionPassed() then
     EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
     if advsimd then  // Advanced SIMD instruction
         for r = 0 to regs-1
             for e = 0 to elements-1
                 Elem[D[d+r],e,esize] = FPMul(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize],
StandardFPSCRValue());
     else             // VFP instruction
         case esize of
             when 16
                 S[d] = Zeros(16) : FPMul(S[n]<15:0>, S[m]<15:0>, FPSCR);
             when 32
                 S[d] = FPMul(S[n], S[m], FPSCR);
             when 64
                 D[d] = FPMul(D[n], D[m], FPSCR);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.147 VMUL (integer and polynomial)

Vector Multiply multiplies corresponding elements in two vectors.

For information about multiplying polynomials see *Polynomial arithmetic over {0, 1}* on page A1-50.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21 20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 0 1	op	0	D	size	Vn	Vd	1 0 0 1	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMUL{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMUL{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' || (op == '1' && size != '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
polynomial = (op == '1');  long_destination = FALSE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1	op	1 1 1 1	0	D	size	Vn	Vd	1 0 0 1	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMUL{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMUL{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' || (op == '1' && size != '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
polynomial = (op == '1');  long_destination = FALSE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the operands, encoded in the "op:size" field. It can have the following values:

`I8`	when `op = 0`, `size = 00`
`I16`	when `op = 0`, `size = 01`
`I32`	when `op = 0`, `size = 10`
`P8`	when `op = 1`, `size = 00`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];  op1val = Int(op1, unsigned);
            op2 = Elem[Din[m+r],e,esize];  op2val = Int(op2, unsigned);
            if polynomial then
                product = PolynomialMult(op1,op2);
            else
                product = (op1val*op2val)<2*esize-1:0>;
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = product;
            else
                Elem[D[d+r],e,esize] = product<esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.148 VMUL (by scalar)

Vector Multiply multiplies each element in a vector by a scalar, and places the results in a second vector.

For more information about scalars see *Advanced SIMD scalars* on page F2-4106.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21 20	19 16	15 12	11 10 9	8	7	6	5	4	3 0
1 1 1 1	0 0 0 1	Q	1	D	!=11	Vn	Vd	1 0 0	F	N	1	M	0	Vm
				size									

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMUL{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>[<index>]
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMUL{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm>[<index>]
```

Decode for all variants of this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || (F == '1' && size == '01' && !HaveFP16Ext()) then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
floating_point = (F == '1');  long_destination = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

T1

15 14 13 12	11 10 9 8	7	6	5 4	3 0	15 12	11 10 9	8	7	6	5	4	3 0
1 1 1	Q	1 1 1 1 1	D	!=11	Vn	Vd	1 0 0	F	N	1	M	0	Vm
				size									

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMUL{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>[<index>]
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMUL{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm>[<index>]
```

Decode for all variants of this encoding

```
if size == '11' then SEE "Related encodings";
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if size == '00' || (F == '1' && size == '01' && !HaveFP16Ext()) then UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
floating_point = (F == '1');  long_destination = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings

Related encodings: See *Advanced SIMD data-processing* on page F3-4137 for the T32 instruction set, or *Advanced SIMD data-processing* on page F4-4225 for the A32 instruction set.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the scalar and the elements of the operand vector, encoded in the "F:size" field. It can have the following values:

`I16`	when `F = 0`, `size = 01`
`I32`	when `F = 0`, `size = 10`
`F16`	when `F = 1`, `size = 01`
`F32`	when `F = 1`, `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register. When `<dt>` is I16 or F16, this is encoded in the "Vm<2:0>" field. Otherwise it is encoded in the "Vm" field.

`<index>` Is the element index. When `<dt>` is I16 or F16, this is in the range 0 to 3 and is encoded in the "M:Vm<3>" field. Otherwise it is in the range 0 to 1 and is encoded in the "M" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    op2 = Elem[Din[m],index,esize];  op2val = Int(op2, unsigned);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];  op1val = Int(op1, unsigned);
            if floating_point then
                Elem[D[d+r],e,esize] = FPMul(op1, op2, StandardFPSCRValue());
            else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if long_destination then
    Elem[Q[d>>1],e,2*esize] = (op1val*op2val)<2*esize-1:0>;
else
    Elem[D[d+r],e,esize] = (op1val*op2val)<esize-1:0>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.149 VMULL (integer and polynomial)

Vector Multiply Long multiplies corresponding elements in two vectors. The destination vector elements are twice as long as the elements that are multiplied.

For information about multiplying polynomials see *Polynomial arithmetic over {0, 1}* on page A1-50.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	U	1	D	!=11 (size)		Vn	Vd	1	1	op	0	N	0	M	0	Vm

A1 variant

```
VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
unsigned = (U == '1'); polynomial = (op == '1'); long_destination = TRUE;
esize = 8 << UInt(size); elements = 64 DIV esize;
if polynomial then
    if U == '1' || size == '01' then UNDEFINED;
    if size == '10' then    // .p64
        if !HaveBit128PMULLExt() then UNDEFINED;
        esize = 64; elements = 1;
if Vd<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = 1;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	U	1	1	1	1	1	D	!=11 (size)		Vn	Vd	1	1	op	0	N	0	M	0	Vm

T1 variant

```
VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
unsigned = (U == '1'); polynomial = (op == '1'); long_destination = TRUE;
esize = 8 << UInt(size); elements = 64 DIV esize;
if polynomial then
    if U == '1' || size == '01' then UNDEFINED;
    if size == '10' then    // .p64
        if InITBlock() then UNPREDICTABLE;
        if !HaveBit128PMULLExt() then UNDEFINED;
        esize = 64; elements = 1;
if Vd<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = 1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CONSTRAINED UNPREDICTABLE behavior

If `op == '1' && size == '10' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Related encodings: See *Advanced SIMD data-processing* on page F3-4137 for the T32 instruction set, or *Advanced SIMD data-processing* on page F4-4225 for the A32 instruction set.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the operands, encoded in the "op:U:size" field. It can have the following values:

`S8`	when `op = 0, U = 0, size = 00`
`S16`	when `op = 0, U = 0, size = 01`
`S32`	when `op = 0, U = 0, size = 10`
`U8`	when `op = 0, U = 1, size = 00`
`U16`	when `op = 0, U = 1, size = 01`
`U32`	when `op = 0, U = 1, size = 10`
`P8`	when `op = 1, U = 0, size = 00`
`P64`	when `op = 1, U = 0, size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];  op1val = Int(op1, unsigned);
            op2 = Elem[Din[m+r],e,esize];  op2val = Int(op2, unsigned);
            if polynomial then
                product = PolynomialMult(op1,op2);
            else
                product = (op1val*op2val)<2*esize-1:0>;
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = product;
            else
                Elem[D[d+r],e,esize] = product<esize-1:0>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.150 VMULL (by scalar)

Vector Multiply Long multiplies each element in a vector by a scalar, and places the results in a second vector. The destination vector elements are twice as long as the elements that are multiplied.

For more information about scalars see *Advanced SIMD scalars* on page F2-4106.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21 20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 U	1	D	!=11	Vn	Vd	1 0 1 0	N	1	M	0	Vm
				size								

A1 variant

```
VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  long_destination = TRUE;  floating_point = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = 1;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 U	1 1 1 1 1	D	!=11	Vn	Vd	1 0 1 0	N	1	M	0	Vm		
			size										

T1 variant

```
VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  long_destination = TRUE;  floating_point = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = 1;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

Notes for all encodings

Related encodings: See *Advanced SIMD data-processing* on page F3-4137 for the T32 instruction set, or *Advanced SIMD data-processing* on page F4-4225 for the A32 instruction set.

Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the scalar and the elements of the operand vector, encoded in the "U:size" field. It can have the following values:

`S16` when `U = 0`, `size = 01`

`S32` when `U = 0`, `size = 10`

`U16` when `U = 1`, `size = 01`

`U32` when `U = 1`, `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>" field when `<dt>` is S16 or U16, otherwise the "Vm" field.

`<index>` Is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field when `<dt>` is S16 or U16, otherwise in range 0 to 1, encoded in the "M" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    op2 = Elem[Din[m],index,esize];  op2val = Int(op2, unsigned);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];  op1val = Int(op1, unsigned);
            if floating_point then
                Elem[D[d+r],e,esize] = FPMul(op1, op2, StandardFPSCRValue());
            else
                if long_destination then
                    Elem[Q[d>>1],e,2*esize] = (op1val*op2val)<2*esize-1:0>;
                else
                    Elem[D[d+r],e,esize] = (op1val*op2val)<esize-1:0>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.151 VMVN (immediate)

Vector Bitwise NOT (immediate) places the bitwise inverse of an immediate integer constant into every element of the destination register.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	16	15	12	11	8	7	6	5	4	3	0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3		Vd		0 x x 0 (cmode)		0	Q	1	1	imm4	

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMVN{<c>}{<q>}.I32 <Dd>, #<imm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMVN{<c>}{<q>}.I32 <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if (cmode<0> == '1' && cmode<3:2> != '11') || cmode<3:1> == '111' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	16	15	12	11	8	7	6	5	4	3	0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3		Vd		1 0 x 0 (cmode)		0	Q	1	1	imm4	

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMVN{<c>}{<q>}.I16 <Dd>, #<imm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMVN{<c>}{<q>}.I16 <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if (cmode<0> == '1' && cmode<3:2> != '11') || cmode<3:1> == '111' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A3

31	30	29	28	27	26	25	24	23	22	21	20	19 18 16	15 12	11 8	7	6	5	4	3 0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3	Vd	1 1 0 x cmode	0	Q	1	1	imm4

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMVN{<c>}{<q>}.I32 <Dd>, #<imm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMVN{<c>}{<q>}.I32 <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if (cmode<0> == '1' && cmode<3:2> != '11') || cmode<3:1> == '111' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 2 0	15 12	11 8	7	6	5	4	3 0
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3	Vd	0 x x 0 cmode	0	Q	1	1	imm4

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMVN{<c>}{<q>}.I32 <Dd>, #<imm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMVN{<c>}{<q>}.I32 <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if (cmode<0> == '1' && cmode<3:2> != '11') || cmode<3:1> == '111' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3 2 0	15 12	11 8	7	6	5	4	3 0
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3	Vd	1 0 x 0 cmode	0	Q	1	1	imm4

64-bit SIMD vector variant

Applies when `Q == 0`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VMVN{<c>}{<q>}.I16 <Dd>, #<imm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMVN{<c>}{<q>}.I16 <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if (cmode<0> == '1' && cmode<3:2> != '11') || cmode<3:1> == '111' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	0	15	12	11	8	7	6	5	4	3	0
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3		Vd		1 1 0 x (cmode)		0	Q	1	1	imm4	

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMVN{<c>}{<q>}.I32 <Dd>, #<imm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMVN{<c>}{<q>}.I32 <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if (cmode<0> == '1' && cmode<3:2> != '11') || cmode<3:1> == '111' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

Notes for all encodings

Related encodings: See *Advanced SIMD one register and modified immediate* on page F3-4146 for the T32 instruction set, or *Advanced SIMD one register and modified immediate* on page F4-4233 for the A32 instruction set.

Assembler symbols

`<c>` For encoding A1, A2 and A3: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1, T2 and T3: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<imm>` Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of `<imm>`, see *Modified immediate constants in T32 and A32 Advanced SIMD instructions* on page F2-4097.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = NOT(imm64);
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.152 VMVN (register)

Vector Bitwise NOT (register) takes a value from a register, inverts the value of each bit, and places the result in the destination register. The registers can be either doubleword or quadword.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 18	17 16	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	1	1	size	0 0	Vd	0	1	0	1	1	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMVN{<c>}{<q>}{.<dt>} <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMVN{<c>}{<q>}{.<dt>} <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 2	1 0	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	1	D	1	1	size	0 0	Vd	0	1	0	1	1	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VMVN{<c>}{<q>}{.<dt>} <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VMVN{<c>}{<q>}{.<dt>} <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` An optional data type. It is ignored by assemblers, and does not affect the encoding.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = NOT(D[m+r]);
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.153 VNEG

Vector Negate negates each element in a vector, and places the results in a second vector. The floating-point version only inverts the sign bit.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 18	17 16	15 12	11	10	9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	1	1	size	0 1	Vd	0	F	1 1	1	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VNEG{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VNEG{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
advsimd = TRUE;  floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

A2

31 28	27 26 25 24	23	22	21	20	19	18 17 16	15 12	11 10	9 8	7	6	5	4	3 0
!=1111	1 1 1 0	1	D	1	1	0	0 0 1	Vd	1 0	size	0	1	M	0	Vm
cond															

Half-precision scalar variant

Applies when `size == 01`.

```
VNEG{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VNEG{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VNEG{<c>}{<q>}.F64 <Dd>, <Dm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	1	Vd	0	F	1	1	1	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VNEG{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VNEG{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
advsimd = TRUE;  floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	0	1	1	1	0	1	D	1	1	0	0	0	1	Vd	1	0	size		0	1	M	0	Vm

Half-precision scalar variant

Applies when `size == 01`.

```
VNEG{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VNEG{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VNEG{<c>}{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding A2, T1 and T2: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "F:size" field. It can have the following values:

`S8`	when `F = 0`, `size = 00`
`S16`	when `F = 0`, `size = 01`
`S32`	when `F = 0`, `size = 10`
`F16`	when `F = 1`, `size = 01`
`F32`	when `F = 1`, `size = 10`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

`<Sm>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                if floating_point then
                    Elem[D[d+r],e,esize] = FPNeg(Elem[D[m+r],e,esize]);
                else
                    result = -SInt(Elem[D[m+r],e,esize]);
                    Elem[D[d+r],e,esize] = result<esize-1:0>;
    else             // VFP instruction
        case esize of
            when 16 S[d] = Zeros(16) : FPNeg(S[m]<15:0>);
            when 32 S[d] = FPNeg(S[m]);
            when 64 D[d] = FPNeg(D[m]);
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.154 VNMLA

Vector Negate Multiply Accumulate multiplies together two floating-point register values, adds the negation of the floating-point value in the destination register to the negation of the product, and writes the result back to the destination register.

Note

Arm recommends that software does not use the `VNMLA` instruction in the Round towards Plus Infinity and Round towards Minus Infinity rounding modes, because the rounding of the product and of the sum can change the result of the instruction in opposite directions, defeating the purpose of these rounding modes.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31–28	27 26 25 24	23	22	21	20	19–16	15–12	11	10	9 8	7	6	5	4	3–0
!=1111	1 1 1 0	0	D	0	1	Vn	Vd	1	0	size	N	1	M	0	Vm
cond												op			

Half-precision scalar variant

Applies when `size == 01`.

```
VNMLA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VNMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VNMLA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
vtype = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	0	1	1	1	0	0	D	0	1	Vn	Vd	1	0	size		N	1	M	0	Vm
																			op			

Half-precision scalar variant

Applies when `size == 01`.

```
VNMLA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VNMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VNMLA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
vtype = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Sd>`	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
`<Sn>`	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
`<Sm>`	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
`<Dn>`	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
`<Dm>`	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
enumeration VFPNegMul {VFPNegMul_VNMLA, VFPNegMul_VNMLS, VFPNegMul_VNMUL};

if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    case esize of
        when 16
            product16 = FPMul(S[n]<15:0>, S[m]<15:0>, FPSCR);
            case vtype of
                when VFPNegMul_VNMLA  S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), FPNeg(product16),
FPSCR);
                when VFPNegMul_VNMLS  S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), product16, FPSCR);
                when VFPNegMul_VNMUL  S[d] = Zeros(16) : FPNeg(product16);
        when 32
            product32 = FPMul(S[n], S[m], FPSCR);
            case vtype of
                when VFPNegMul_VNMLA  S[d] = FPAdd(FPNeg(S[d]), FPNeg(product32), FPSCR);
                when VFPNegMul_VNMLS  S[d] = FPAdd(FPNeg(S[d]), product32, FPSCR);
                when VFPNegMul_VNMUL  S[d] = FPNeg(product32);
        when 64
            product64 = FPMul(D[n], D[m], FPSCR);
            case vtype of
                when VFPNegMul_VNMLA  D[d] = FPAdd(FPNeg(D[d]), FPNeg(product64), FPSCR);
                when VFPNegMul_VNMLS  D[d] = FPAdd(FPNeg(D[d]), product64, FPSCR);
                when VFPNegMul_VNMUL  D[d] = FPNeg(product64);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.155 VNMLS

Vector Negate Multiply Subtract multiplies together two floating-point register values, adds the negation of the floating-point value in the destination register to the product, and writes the result back to the destination register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31–28	27 26 25 24	23	22	21	20	19–16	15–12	11 10	9 8	7	6	5	4	3–0
!=1111	1 1 1 0	0	D	0	1	Vn	Vd	1 0	size	N	0	M	0	Vm
cond											op			

Half-precision scalar variant

Applies when `size == 01`.

```
VNMLS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VNMLS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VNMLS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
vtype = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15 14 13 12	11 10 9 8	7	6	5	4	3–0	15–12	11 10	9 8	7	6	5	4	3–0
1 1 1 0	1 1 1 0	0	D	0	1	Vn	Vd	1 0	size	N	0	M	0	Vm
											op			

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Half-precision scalar variant

Applies when `size == 01`.

```
VNMLS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VNMLS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VNMLS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
vtype = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Sd>`	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
`<Sn>`	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
`<Sm>`	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
`<Dn>`	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
`<Dm>`	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
enumeration VFPNegMul {VFPNegMul_VNMLA, VFPNegMul_VNMLS, VFPNegMul_VNMUL};

if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    case esize of
        when 16
            product16 = FPMul(S[n]<15:0>, S[m]<15:0>, FPSCR);
            case vtype of
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
                when VFPNegMul_VNMLA  S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), FPNeg(product16),
FPSCR);
                when VFPNegMul_VNMLS  S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), product16, FPSCR);
                when VFPNegMul_VNMUL  S[d] = Zeros(16) : FPNeg(product16);
        when 32
            product32 = FPMul(S[n], S[m], FPSCR);
            case vtype of
                when VFPNegMul_VNMLA  S[d] = FPAdd(FPNeg(S[d]), FPNeg(product32), FPSCR);
                when VFPNegMul_VNMLS  S[d] = FPAdd(FPNeg(S[d]), product32, FPSCR);
                when VFPNegMul_VNMUL  S[d] = FPNeg(product32);
        when 64
            product64 = FPMul(D[n], D[m], FPSCR);
            case vtype of
                when VFPNegMul_VNMLA  D[d] = FPAdd(FPNeg(D[d]), FPNeg(product64), FPSCR);
                when VFPNegMul_VNMLS  D[d] = FPAdd(FPNeg(D[d]), product64, FPSCR);
                when VFPNegMul_VNMUL  D[d] = FPNeg(product64);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.156 VNMUL

Vector Negate Multiply multiplies together two floating-point register values, and writes the negation of the result to the destination register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31–28	27 26 25 24	23	22	21	20	19–16	15–12	11	10	9 8	7	6	5	4	3–0
!=1111	1 1 1 0	0	D	1	0	Vn	Vd	1	0	size	N	1	M	0	Vm
cond															

Half-precision scalar variant

Applies when `size == 01`.

```
VNMUL{<c>}{<q>}.F16 {<Sd>,} <Sn>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VNMUL{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VNMUL{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '01' && !HaveFP16Ext() then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
vtype = VFPNegMul_VNMUL;
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15 14 13 12	11 10 9 8	7	6	5	4	3–0	15–12	11	10	9 8	7	6	5	4	3–0
1 1 1 0	1 1 1 0	0	D	1	0	Vn	Vd	1	0	size	N	1	M	0	Vm

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Half-precision scalar variant

Applies when `size == 01`.

```
VNMUL{<c>}{<q>}.F16 {<Sd>,} <Sn>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VNMUL{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VNMUL{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '01' && !HaveFP16Ext() then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
vtype = VFPNegMul_VNMUL;
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Sd>`	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
`<Sn>`	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
`<Sm>`	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
`<Dn>`	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
`<Dm>`	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
enumeration VFPNegMul {VFPNegMul_VNMLA, VFPNegMul_VNMLS, VFPNegMul_VNMUL};

if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    case esize of
        when 16
            product16 = FPMul(S[n]<15:0>, S[m]<15:0>, FPSCR);
            case vtype of
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
                when VFPNegMul_VNMLA  S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), FPNeg(product16),
FPSCR);
                when VFPNegMul_VNMLS  S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), product16, FPSCR);
                when VFPNegMul_VNMUL  S[d] = Zeros(16) : FPNeg(product16);
            when 32
                product32 = FPMul(S[n], S[m], FPSCR);
                case vtype of
                when VFPNegMul_VNMLA  S[d] = FPAdd(FPNeg(S[d]), FPNeg(product32), FPSCR);
                when VFPNegMul_VNMLS  S[d] = FPAdd(FPNeg(S[d]), product32, FPSCR);
                when VFPNegMul_VNMUL  S[d] = FPNeg(product32);
            when 64
                product64 = FPMul(D[n], D[m], FPSCR);
                case vtype of
                when VFPNegMul_VNMLA  D[d] = FPAdd(FPNeg(D[d]), FPNeg(product64), FPSCR);
                when VFPNegMul_VNMLS  D[d] = FPAdd(FPNeg(D[d]), product64, FPSCR);
                when VFPNegMul_VNMUL  D[d] = FPNeg(product64);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.157 VORN (immediate)

Vector Bitwise OR NOT (immediate) performs a bitwise OR between a register value and the complement of an immediate value, and returns the result into the destination vector

This instruction is a pseudo-instruction of the VORR (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of VORR (immediate).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VORR (immediate) gives the operational pseudocode for this instruction.

A1

31 30 29 28	27 26 25 24	23	22	21 20	19 18 16	15 12	11 8	7	6	5	4	3 0
1 1 1 1	0 0 1 i	1	D	0 0 0	imm3	Vd	0 x x 1	0	Q	0	1	imm4
							cmode					

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VORN{<c>}{<q>}.I16 {<Dd>,} <Dd>, #<imm>
```

is equivalent to

```
VORR{<c>}{<q>}.I16 <Dd>, #~<imm>
```

and is never the preferred disassembly.

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VORN{<c>}{<q>}.I16 {<Qd>,} <Qd>, #<imm>
```

is equivalent to

```
VORR{<c>}{<q>}.I16 <Qd>, #~<imm>
```

and is never the preferred disassembly.

A2

31 30 29 28	27 26 25 24	23	22	21 20	19 18 16	15 12	11 8	7	6	5	4	3 0
1 1 1 1	0 0 1 i	1	D	0 0 0	imm3	Vd	1 0 x 1	0	Q	0	1	imm4
							cmode					

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VORN{<c>}{<q>}.I32 {<Dd>,} <Dd>, #<imm>
```

is equivalent to

```
VORR{<c>}{<q>}.I32 <Dd>, #~<imm>
```

and is never the preferred disassembly.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VORN{<c>}{<q>}.I32 {<Qd>,} <Qd>, #<imm>
```

is equivalent to

```
VORR{<c>}{<q>}.I32 <Qd>, #~<imm>
```

and is never the preferred disassembly.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	0	15	12	11	8	7	6	5	4	3	0
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3		Vd		0 x x 1 (cmode)		0	Q	0	1	imm4	

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VORN{<c>}{<q>}.I16 {<Dd>,} <Dd>, #<imm>
```

is equivalent to

```
VORR{<c>}{<q>}.I16 <Dd>, #~<imm>
```

and is never the preferred disassembly.

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VORN{<c>}{<q>}.I16 {<Qd>,} <Qd>, #<imm>
```

is equivalent to

```
VORR{<c>}{<q>}.I16 <Qd>, #~<imm>
```

and is never the preferred disassembly.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	0	15	12	11	8	7	6	5	4	3	0
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3		Vd		1 0 x 1 (cmode)		0	Q	0	1	imm4	

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VORN{<c>}{<q>}.I32 {<Dd>,} <Dd>, #<imm>
```

is equivalent to

```
VORR{<c>}{<q>}.I32 <Dd>, #~<imm>
```

and is never the preferred disassembly.

128-bit SIMD vector variant

Applies when `Q == 1`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VORN{<c>}{<q>}.I32 {<Qd>,} <Qd>, #<imm>
```

is equivalent to

```
VORR{<c>}{<q>}.I32 <Qd>, #~<imm>
```

and is never the preferred disassembly.

Assembler symbols

`<c>` For encoding A1 and A2: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1 and T2: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<imm>` Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of `<imm>`, see *Modified immediate constants in T32 and A32 Advanced SIMD instructions* on page F2-4097.

Operation for all encodings

The description of VORR (immediate) gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.158 VORN (register)

Vector bitwise OR NOT (register) performs a bitwise OR NOT operation between two registers, and places the result in the destination register. The operand and result registers can be quadword or doubleword. They must all be the same size.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 0 1	0	0	D	1	1	Vn	Vd	0 0 0 1	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VORN{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VORN{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 0	1 1 1 1	0	D	1	1	Vn	Vd	0 0 0 1	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VORN{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VORN{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` An optional data type. It is ignored by assemblers, and does not affect the encoding.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[n+r] OR NOT(D[m+r]);
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.159 VORR (immediate)

Vector Bitwise OR (immediate) performs a bitwise OR between a register value and an immediate value, and returns the result into the destination vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

This instruction is used by the pseudo-instruction VORN (immediate). The pseudo-instruction is never the preferred disassembly.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	16	15	12	11	8	7	6	5	4	3	0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3		Vd		0 x x 1 (cmode)		0	Q	0	1	imm4	

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VORR{<c>}{<q>}.I32 {<Dd>,} <Dd>, #<imm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VORR{<c>}{<q>}.I32 {<Qd>,} <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if cmode<0> == '0' || cmode<3:2> == '11' then SEE "VMOV (immediate)";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('0', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	16	15	12	11	8	7	6	5	4	3	0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3		Vd		1 0 x 1 (cmode)		0	Q	0	1	imm4	

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VORR{<c>}{<q>}.I16 {<Dd>,} <Dd>, #<imm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VORR{<c>}{<q>}.I16 {<Qd>,} <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if cmode<0> == '0' || cmode<3:2> == '11' then SEE "VMOV (immediate)";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('0', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	0	15	12	11	8	7	6	5	4	3	0
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3		Vd		0 x x 1 (cmode)		0	Q	0	1	imm4	

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VORR{<c>}{<q>}.I32 {<Dd>,} <Dd>, #<imm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VORR{<c>}{<q>}.I32 {<Qd>,} <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if cmode<0> == '0' || cmode<3:2> == '11' then SEE "VMOV (immediate)";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('0', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	0	15	12	11	8	7	6	5	4	3	0
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3		Vd		1 0 x 1 (cmode)		0	Q	0	1	imm4	

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VORR{<c>}{<q>}.I16 {<Dd>,} <Dd>, #<imm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VORR{<c>}{<q>}.I16 {<Qd>,} <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if cmode<0> == '0' || cmode<3:2> == '11' then SEE "VMOV (immediate)";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('0', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

Assembler symbols

<c>	For encoding A1 and A2: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional. For encoding T1 and T2: see *Standard assembler syntax fields* on page F2-4080.
<q>	See *Standard assembler syntax fields* on page F2-4080.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<imm>` Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of `<imm>`, see *Modified immediate constants in T32 and A32 Advanced SIMD instructions* on page F2-4097.

The I8, I64, and F32 data types are permitted as pseudo-instructions, if the immediate can be represented by this instruction, and are encoded using a permitted encoding of the I16 or I32 data type.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[d+r] OR imm64;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.160 VORR (register)

Vector bitwise OR (register) performs a bitwise OR operation between two registers, and places the result in the destination register. The operand and result registers can be quadword or doubleword. They must all be the same size.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

This instruction is used by the pseudo-instructions VMOV (register, SIMD), VRSHR (zero), and VSHR (zero). The pseudo-instruction is never the preferred disassembly.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 0	0	D	1	0	Vn	Vd	0 0 0 1	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VORR{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VORR{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 0	1 1 1 1	0	D	1	0	Vn	Vd	0 0 0 1	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VORR{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VORR{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Alias conditions

Alias or pseudo-instruction	is preferred when
VMOV (register, SIMD)	`N:Vn == M:Vm`
VRSHR (zero)	Never
VSHR (zero)	Never

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` An optional data type. It is ignored by assemblers, and does not affect the encoding.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[n+r] OR D[m+r];
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

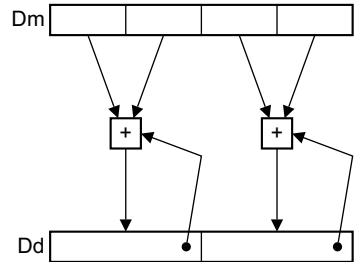
Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.161 VPADAL

Vector Pairwise Add and Accumulate Long adds adjacent pairs of elements of a vector, and accumulates the results into the elements of the destination vector.

The vectors can be doubleword or quadword. The operand elements can be 8-bit, 16-bit, or 32-bit integers. The result elements are twice the length of the operand elements.

The following figure shows the operation of VPADAL doubleword operation for data type S16.



Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15 – 12	11	10	9	8	7	6	5	4	3 – 0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd	0	1	1	0	op	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VPADAL{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VPADAL{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (op == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15 – 12	11	10	9	8	7	6	5	4	3 – 0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd	0	1	1	0	op	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VPADAL{<c>}{<q>}.<dt> <Dd>, <Dm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VPADAL{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (op == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "op:size" field. It can have the following values:

`S8`	when `op = 0`, `size = 00`
`S16`	when `op = 0`, `size = 01`
`S32`	when `op = 0`, `size = 10`
`U8`	when `op = 1`, `size = 00`
`U16`	when `op = 1`, `size = 01`
`U32`	when `op = 1`, `size = 10`

The following encodings are reserved:

- `op = 0`, `size = 11`.
- `op = 1`, `size = 11`.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    h = elements DIV 2;

    for r = 0 to regs-1
        for e = 0 to h-1
            op1 = Elem[D[m+r],2*e,esize];  op2 = Elem[D[m+r],2*e+1,esize];
            result = Int(op1, unsigned) + Int(op2, unsigned);
            Elem[D[d+r],e,2*esize] = Elem[D[d+r],e,2*esize] + result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.162 VPADD (floating-point)

Vector Pairwise Add (floating-point) adds adjacent pairs of elements of two vectors, and places the results in the destination vector.

The operands and result are doubleword vectors.

The operand and result elements are floating-point numbers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	0	D	0	sz	Vn	Vd	1 1 0 1	N	Q	M	0	Vm

A1 variant

```
VPADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

Decode for this encoding

```
if Q == '1' then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 1 0	D	0	sz		Vn	Vd	1 1 0 1	N	Q	M	0	Vm

T1 variant

```
VPADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

Decode for this encoding

```
if Q == '1' then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

`F32` when `sz = 0`

`F16` when `sz = 1`

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
 if ConditionPassed() then
     EncodingSpecificOperations();  CheckAdvSIMDEnabled();
     bits(64) dest;
     h = elements DIV 2;

     for e = 0 to h-1
         Elem[dest,e,esize]   = FPAdd(Elem[D[n],2*e,esize], Elem[D[n],2*e+1,esize],
StandardFPSCRValue());
         Elem[dest,e+h,esize] = FPAdd(Elem[D[m],2*e,esize], Elem[D[m],2*e+1,esize],
StandardFPSCRValue());

     D[d] = dest;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

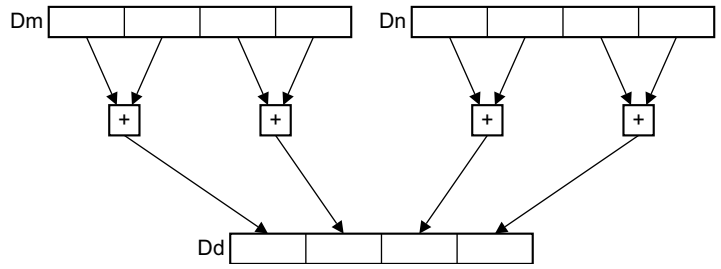
F6.1.163 VPADD (integer)

Vector Pairwise Add (integer) adds adjacent pairs of elements of two vectors, and places the results in the destination vector.

The operands and result are doubleword vectors.

The operand and result elements must all be the same type, and can be 8-bit, 16-bit, or 32-bit integers. There is no distinction between signed and unsigned integers.

The following figure shows the operation of VPADD doubleword operation for data type I16.



Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21 20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 0	0	D	size	Vn	Vd	1 0 1 1	N	Q	M	1	Vm

A1 variant

```
VPADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' || Q == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 0	1 1 1 1 0	D	size	Vn	Vd	1 0 1 1	N	Q	M	1	Vm

T1 variant

```
VPADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' || Q == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:

`I8`	when `size = 00`
`I16`	when `size = 01`
`I32`	when `size = 10`

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(64) dest;
    h = elements DIV 2;

    for e = 0 to h-1
        Elem[dest,e,esize]   = Elem[D[n],2*e,esize] + Elem[D[n],2*e+1,esize];
        Elem[dest,e+h,esize] = Elem[D[m],2*e,esize] + Elem[D[m],2*e+1,esize];

    D[d] = dest;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

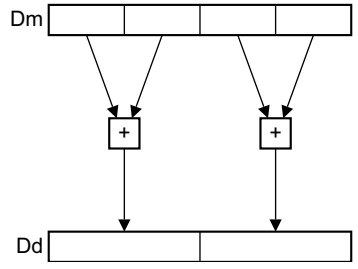
Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.164 VPADDL

Vector Pairwise Add Long adds adjacent pairs of elements of two vectors, and places the results in the destination vector.

The vectors can be doubleword or quadword. The operand elements can be 8-bit, 16-bit, or 32-bit integers. The result elements are twice the length of the operand elements.

The following figure shows the operation of VPADDL doubleword operation for data type S16.



Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd		0	0	1	0	op	Q	M	0	Vm	

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VPADDL{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VPADDL{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (op == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd		0	0	1	0	op	Q	M	0	Vm	

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VPADDL{<c>}{<q>}.<dt> <Dd>, <Dm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VPADDL{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (op == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "op:size" field. It can have the following values:

`S8`	when `op = 0`, `size = 00`
`S16`	when `op = 0`, `size = 01`
`S32`	when `op = 0`, `size = 10`
`U8`	when `op = 1`, `size = 00`
`U16`	when `op = 1`, `size = 01`
`U32`	when `op = 1`, `size = 10`

The following encodings are reserved:

- `op = 0`, `size = 11`.
- `op = 1`, `size = 11`.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    h = elements DIV 2;

    for r = 0 to regs-1
        for e = 0 to h-1
            op1 = Elem[D[m+r],2*e,esize];  op2 = Elem[D[m+r],2*e+1,esize];
            result = Int(op1, unsigned) + Int(op2, unsigned);
            Elem[D[d+r],e,2*esize] = result<2*esize-1:0>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.165 VPMAX (floating-point)

Vector Pairwise Maximum compares adjacent pairs of elements in two doubleword vectors, and copies the larger of each pair into the corresponding element in the destination doubleword vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	1	0	D	0	sz	Vn	Vd	1	1	1	1	N	0	M	0	Vm
										op												

A1 variant

```
VPMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

Decode for this encoding

```
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
maximum = (op == '0');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	1	1	0	D	0	sz	Vn	Vd	1	1	1	1	N	0	M	0	Vm
										op												

T1 variant

```
VPMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

Decode for this encoding

```
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
maximum = (op == '0');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

`F32` when `sz = 0`

`F16` when `sz = 1`

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

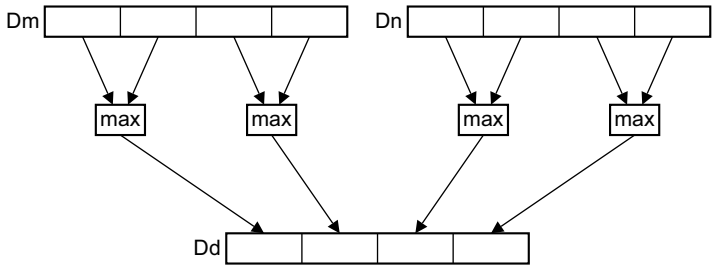
```
 if ConditionPassed() then
     EncodingSpecificOperations();  CheckAdvSIMDEnabled();
     bits(64) dest;
     h = elements DIV 2;

     for e = 0 to h-1
         op1 = Elem[D[n],2*e,esize];  op2 = Elem[D[n],2*e+1,esize];
         Elem[dest,e,esize] = if maximum then FPMax(op1,op2,StandardFPSCRValue()) else
FPMin(op1,op2,StandardFPSCRValue());
         op1 = Elem[D[m],2*e,esize];  op2 = Elem[D[m],2*e+1,esize];
         Elem[dest,e+h,esize] = if maximum then FPMax(op1,op2,StandardFPSCRValue()) else
FPMin(op1,op2,StandardFPSCRValue());

     D[d] = dest;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.166 VPMAX (integer)

Vector Pairwise Maximum compares adjacent pairs of elements in two doubleword vectors, and copies the larger of each pair into the corresponding element in the destination doubleword vector.

The following figure shows the operation of VPMAX doubleword operation for data type S16 or U16.



Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

1	1	1	1	0	0	1	U	0	D	size	Vn	Vd	1 0 1 0	N	0	M	0	Vm
																	op	

A1 variant

```
VPMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then UNDEFINED;
maximum = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 10 9 8	7 6 5 4	3 0
1 1 1 U	1 1 1 1 0	D size	Vn	Vd	1 0 1 0	N 0 M 0	Vm
						op	

T1 variant

```
VPMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then UNDEFINED;
maximum = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

`S8`	when `U = 0`, `size = 00`
`S16`	when `U = 0`, `size = 01`
`S32`	when `U = 0`, `size = 10`
`U8`	when `U = 1`, `size = 00`
`U16`	when `U = 1`, `size = 01`
`U32`	when `U = 1`, `size = 10`

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(64) dest;
    h = elements DIV 2;

    for e = 0 to h-1
        op1 = Int(Elem[D[n],2*e,esize], unsigned);
        op2 = Int(Elem[D[n],2*e+1,esize], unsigned);
        result = if maximum then Max(op1,op2) else Min(op1,op2);
        Elem[dest,e,esize] = result<esize-1:0>;
        op1 = Int(Elem[D[m],2*e,esize], unsigned);
        op2 = Int(Elem[D[m],2*e+1,esize], unsigned);
        result = if maximum then Max(op1,op2) else Min(op1,op2);
        Elem[dest,e+h,esize] = result<esize-1:0>;

    D[d] = dest;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.167 VPMIN (floating-point)

Vector Pairwise Minimum compares adjacent pairs of elements in two doubleword vectors, and copies the smaller of each pair into the corresponding element in the destination doubleword vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	1	0	D	1	sz	Vn	Vd	1	1	1	1	N	0	M	0	Vm
										op												

A1 variant

```
VPMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

Decode for this encoding

```
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
maximum = (op == '0');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	1	1	0	D	1	sz	Vn	Vd	1	1	1	1	N	0	M	0	Vm
										op												

T1 variant

```
VPMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

Decode for this encoding

```
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
maximum = (op == '0');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

`F32` when `sz = 0`

`F16` when `sz = 1`

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
 if ConditionPassed() then
     EncodingSpecificOperations();  CheckAdvSIMDEnabled();
     bits(64) dest;
     h = elements DIV 2;

     for e = 0 to h-1
         op1 = Elem[D[n],2*e,esize];  op2 = Elem[D[n],2*e+1,esize];
         Elem[dest,e,esize] = if maximum then FPMax(op1,op2,StandardFPSCRValue()) else
FPMin(op1,op2,StandardFPSCRValue());
         op1 = Elem[D[m],2*e,esize];  op2 = Elem[D[m],2*e+1,esize];
         Elem[dest,e+h,esize] = if maximum then FPMax(op1,op2,StandardFPSCRValue()) else
FPMin(op1,op2,StandardFPSCRValue());

     D[d] = dest;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.168 VPMIN (integer)

Vector Pairwise Minimum compares adjacent pairs of elements in two doubleword vectors, and copies the smaller of each pair into the corresponding element in the destination doubleword vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19 … 16	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	1	0	0	1	U	0	D	size		Vn	Vd	1	0	1	0	N	0	M	1	Vm
																					op	

A1 variant

```
VPMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then UNDEFINED;
maximum = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	U	1	1	1	1	0	D	size		Vn	Vd	1	0	1	0	N	0	M	1	Vm
																					op	

T1 variant

```
VPMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then UNDEFINED;
maximum = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

<dt> Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

S8	when U = 0, size = 00
S16	when U = 0, size = 01
S32	when U = 0, size = 10
U8	when U = 1, size = 00

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

U16	when U = 1, size = 01
U32	when U = 1, size = 10

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(64) dest;
    h = elements DIV 2;

    for e = 0 to h-1
        op1 = Int(Elem[D[n],2*e,esize], unsigned);
        op2 = Int(Elem[D[n],2*e+1,esize], unsigned);
        result = if maximum then Max(op1,op2) else Min(op1,op2);
        Elem[dest,e,esize] = result<esize-1:0>;
        op1 = Int(Elem[D[m],2*e,esize], unsigned);
        op2 = Int(Elem[D[m],2*e+1,esize], unsigned);
        result = if maximum then Max(op1,op2) else Min(op1,op2);
        Elem[dest,e+h,esize] = result<esize-1:0>;

    D[d] = dest;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.169 VPOP

Pop SIMD&FP registers from Stack loads multiple consecutive Advanced SIMD and floating-point register file registers from the stack

This instruction is an alias of the VLDM, VLDMDB, VLDMIA instruction. This means that:

- The encodings in this description are named to match the encodings of VLDM, VLDMDB, VLDMIA.
- The description of VLDM, VLDMDB, VLDMIA gives the operational pseudocode for this instruction.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7–1	0
!=1111	1	1	0	0	1	D	1	1	1 1 0 1	Vd	1	0	1	1	imm8<7:1>	0
cond				P	U		W		Rn							imm8<0>

Increment After variant

```
VPOP{<c>}{<q>}{.<size>} <dreglist>
```

is equivalent to

```
VLDM{<c>}{<q>}{.<size>} SP!, <dreglist>
```

and is always the preferred disassembly.

A2

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7–0
!=1111	1	1	0	0	1	D	1	1	1 1 0 1	Vd	1	0	1	0	imm8
cond				P	U		W		Rn						

Increment After variant

```
VPOP{<c>}{<q>}{.<size>} <sreglist>
```

is equivalent to

```
VLDM{<c>}{<q>}{.<size>} SP!, <sreglist>
```

and is always the preferred disassembly.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7–1	0
1	1	1	0	1	1	0	0	1	D	1	1	1 1 0 1	Vd	1	0	1	1	imm8<7:1>	0
							P	U		W		Rn							imm8<0>

Increment After variant

```
VPOP{<c>}{<q>}{.<size>} <dreglist>
```

is equivalent to

```
VLDM{<c>}{<q>}{.<size>} SP!, <dreglist>
```

and is always the preferred disassembly.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T2

15	14	13	12	11	10	9	8	7	6	5	4	3			0	15			12	11	10	9	8	7			0
1	1	1	0	1	1	0	0	1	D	1	1	1	1	0	1	Vd				1	0	1	0	imm8			
							P	U		W			Rn														

Increment After variant

```
VPOP{<c>}{<q>}{.<size>} <sreglist>
```

is equivalent to

```
VLDM{<c>}{<q>}{.<size>} SP!, <sreglist>
```

and is always the preferred disassembly.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<size>` An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.

`<sreglist>` Is the list of consecutively numbered 32-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "Vd:D", and "imm8" is set to the number of registers in the list. The list must contain at least one register.

`<dreglist>` Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers.

Operation for all encodings

The description of VLDM, VLDMDB, VLDMIA gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.170 VPUSH

Push SIMD&FP registers to Stack stores multiple consecutive registers from the Advanced SIMD and floating-point register file to the stack

This instruction is an alias of the VSTM, VSTMDB, VSTMIA instruction. This means that:

- The encodings in this description are named to match the encodings of VSTM, VSTMDB, VSTMIA.
- The description of VSTM, VSTMDB, VSTMIA gives the operational pseudocode for this instruction.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7–1	0
!=1111	1	1	0	1	0	D	1	0	1 1 0 1	Vd	1	0	1	1	imm8<7:1>	0
cond				P	U		W		Rn							imm8<0>

Decrement Before variant

```
VPUSH{<c>}{<q>}{.<size>} <dreglist>
```

is equivalent to

```
VSTMDB{<c>}{<q>}{.<size>} SP!, <dreglist>
```

and is always the preferred disassembly.

A2

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7–0
!=1111	1	1	0	1	0	D	1	0	1 1 0 1	Vd	1	0	1	0	imm8
cond				P	U		W		Rn						

Decrement Before variant

```
VPUSH{<c>}{<q>}{.<size>} <sreglist>
```

is equivalent to

```
VSTMDB{<c>}{<q>}{.<size>} SP!, <sreglist>
```

and is always the preferred disassembly.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7–1	0
1	1	1	0	1	1	0	1	0	D	1	0	1 1 0 1	Vd	1	0	1	1	imm8<7:1>	0
							P	U		W		Rn							imm8<0>

Decrement Before variant

```
VPUSH{<c>}{<q>}{.<size>} <dreglist>
```

is equivalent to

```
VSTMDB{<c>}{<q>}{.<size>} SP!, <dreglist>
```

and is always the preferred disassembly.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	1	0	D	1	0	1	1	0	1	Vd				1	0	1	0	imm8							
							P	U		W		Rn																			

Decrement Before variant

```
VPUSH{<c>}{<q>}{.<size>} <sreglist>
```

is equivalent to

```
VSTMDB{<c>}{<q>}{.<size>} SP!, <sreglist>
```

and is always the preferred disassembly.

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<size>` An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.

`<sreglist>` Is the list of consecutively numbered 32-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "Vd:D", and "imm8" is set to the number of registers in the list. The list must contain at least one register.

`<dreglist>` Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers.

Operation for all encodings

The description of VSTM, VSTMDB, VSTMIA gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.171 VQABS

Vector Saturating Absolute takes the absolute value of each element in a vector, and places the results in the destination vector.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-3951.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd		0	1	1	1	0	Q	M	0	Vm	

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VQABS{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VQABS{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd		0	1	1	1	0	Q	M	0	Vm	

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VQABS{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VQABS{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:

`S8`	when `size = 00`
`S16`	when `size = 01`
`S32`	when `size = 10`

The encoding `size = 11` is reserved.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = Abs(SInt(Elem[D[m+r],e,esize]));
            (Elem[D[d+r],e,esize], sat) = SignedSatQ(result, esize);
            if sat then FPSCR.QC = '1';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.172 VQADD

Vector Saturating Add adds the values of corresponding elements of two vectors, and places the results in the destination vector.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-3951.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 0 1	U	0	D	size	Vn	Vd	0 0 0 0	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VQADD{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VQADD{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1	U	1 1 1 1	0	D	size	Vn	Vd	0 0 0 0	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VQADD{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VQADD{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "U:size" field. It can have the following values:

`S8`	when `U = 0`, `size = 00`
`S16`	when `U = 0`, `size = 01`
`S32`	when `U = 0`, `size = 10`
`S64`	when `U = 0`, `size = 11`
`U8`	when `U = 1`, `size = 00`
`U16`	when `U = 1`, `size = 01`
`U32`	when `U = 1`, `size = 10`
`U64`	when `U = 1`, `size = 11`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            sum = Int(Elem[D[n+r],e,esize], unsigned) + Int(Elem[D[m+r],e,esize], unsigned);
            (Elem[D[d+r],e,esize], sat) = SatQ(sum, esize, unsigned);
            if sat then FPSCR.QC = '1';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.173 VQDMLAL

Vector Saturating Doubling Multiply Accumulate Long multiplies corresponding elements in two doubleword vectors, doubles the products, and accumulates the results into the elements of a quadword vector.

The second operand can be a scalar instead of a vector. For more information about scalars see *Advanced SIMD scalars* on page F2-4106.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-3951.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	0	1	D	!=11		Vn	Vd	1	0	0	1	N	0	M	0	Vm
										size						op						

A1 variant

```
VQDMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = FALSE;  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 8 << UInt(size);  elements = 64 DIV esize;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	0	1	D	!=11		Vn	Vd	0	0	1	1	N	1	M	0	Vm
										size					op							

A2 variant

```
VQDMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	0	1	1	1	1	1	D	!=11		Vn	Vd	1	0	0	1	N	0	M	0	Vm
										size						op						

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1 variant

```
VQDMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = FALSE;  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 8 << UInt(size);  elements = 64 DIV esize;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	0	1	1	1	1	1	D	!=11 (size)		Vn	Vd	0	0 (op)	1	1	N	1	M	0	Vm

T2 variant

```
VQDMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

Notes for all encodings

Related encodings: See *Advanced SIMD data-processing* on page F3-4137 for the T32 instruction set, or *Advanced SIMD data-processing* on page F4-4225 for the A32 instruction set.

Assembler symbols

`<c>` For encoding A1 and A2: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1 and T2: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:

`S16` when `size = 01`

`S32` when `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` For encoding A1 and T1: is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

For encoding A2 and T2: is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>" field when `<dt>` is S16, otherwise the "Vm" field.

`<index>` Is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field when `<dt>` is S16, otherwise in range 0 to 1, encoded in the "M" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if scalar_form then op2 = SInt(Elem[Din[m],index,esize]);
    for e = 0 to elements-1
        if !scalar_form then op2 = SInt(Elem[Din[m],e,esize]);
        op1 = SInt(Elem[Din[n],e,esize]);
        // The following only saturates if both op1 and op2 equal -(2^(esize-1))
        (product, sat1) = SignedSatQ(2*op1*op2, 2*esize);
        if add then
            result = SInt(Elem[Qin[d>>1],e,2*esize]) + SInt(product);
        else
            result = SInt(Elem[Qin[d>>1],e,2*esize]) - SInt(product);
        (Elem[Q[d>>1],e,2*esize], sat2) = SignedSatQ(result, 2*esize);
        if sat1 || sat2 then FPSCR.QC = '1';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.174 VQDMLSL

Vector Saturating Doubling Multiply Subtract Long multiplies corresponding elements in two doubleword vectors, subtracts double the products from corresponding elements of a quadword vector, and places the results in the same quadword vector.

The second operand can be a scalar instead of a vector. For more information about scalars see *Advanced SIMD scalars* on page F2-4106.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-3951.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	0	1	D	!=11		Vn	Vd	1	0	1	1	N	0	M	0	Vm
										size						op						

A1 variant

```
VQDMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = FALSE;  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 8 << UInt(size);  elements = 64 DIV esize;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	0	1	D	!=11		Vn	Vd	0	1	1	1	N	1	M	0	Vm
										size					op							

A2 variant

```
VQDMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	0	1	1	1	1	1	D	!=11		Vn	Vd	1	0	1	1	N	0	M	0	Vm
										size						op						

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1 variant

```
VQDMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = FALSE;  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 8 << UInt(size);  elements = 64 DIV esize;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	0	1	1	1	1	1	D	!=11		Vn	Vd	0	1	1	1	N	1	M	0	Vm
										size					op							

T2 variant

```
VQDMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

Notes for all encodings

Related encodings: See *Advanced SIMD data-processing* on page F3-4137 for the T32 instruction set, or *Advanced SIMD data-processing* on page F4-4225 for the A32 instruction set.

Assembler symbols

`<c>` For encoding A1 and A2: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1 and T2: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:

`S16` when `size = 01`

`S32` when `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` For encoding A1 and T1: is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

For encoding A2 and T2: is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>" field when `<dt>` is S16, otherwise the "Vm" field.

`<index>` Is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field when `<dt>` is S16, otherwise in range 0 to 1, encoded in the "M" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if scalar_form then op2 = SInt(Elem[Din[m],index,esize]);
    for e = 0 to elements-1
        if !scalar_form then op2 = SInt(Elem[Din[m],e,esize]);
        op1 = SInt(Elem[Din[n],e,esize]);
        // The following only saturates if both op1 and op2 equal -(2^(esize-1))
        (product, sat1) = SignedSatQ(2*op1*op2, 2*esize);
        if add then
            result = SInt(Elem[Qin[d>>1],e,2*esize]) + SInt(product);
        else
            result = SInt(Elem[Qin[d>>1],e,2*esize]) - SInt(product);
        (Elem[Q[d>>1],e,2*esize], sat2) = SignedSatQ(result, 2*esize);
        if sat1 || sat2 then FPSCR.QC = '1';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.175 VQDMULH

Vector Saturating Doubling Multiply Returning High Half multiplies corresponding elements in two vectors, doubles the results, and places the most significant half of the final results in the destination vector. The results are truncated, for rounded results see VQRDMULH.

The second operand can be a scalar instead of a vector. For more information about scalars see *Advanced SIMD scalars* on page F2-4106.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-3951.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	0	0	D	size		Vn	Vd	1	0	1	1	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VQDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VQDMULH{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
scalar_form = FALSE;  esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	Q	1	D	!=11		Vn	Vd	1	1	0	0	N	1	M	0	Vm
										size												

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VQDMULH{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm[x]>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VQDMULH{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm[x]>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

T1

15 14 13 12	11 10 9 8	7	6	5 4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1	0	1 1 1 1	0	D	size	Vn	Vd	1 0 1 1	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VQDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VQDMULH{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
scalar_form = FALSE;  esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T2

15 14 13 12	11 10 9 8	7	6	5 4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1	Q	1 1 1 1	1	D	!=11	Vn	Vd	1 1 0 0	N	1	M	0	Vm
					size								

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VQDMULH{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm[x]>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VQDMULH{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm[x]>
```

Decode for all variants of this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Notes for all encodings

Related encodings: See *Advanced SIMD data-processing* on page F3-4137 for the T32 instruction set, or *Advanced SIMD data-processing* on page F4-4225 for the A32 instruction set.

Assembler symbols

`<c>` For encoding A1 and A2: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1 and T2: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:

`S16` when `size = 01`

`S32` when `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm[x]>` Is the 64-bit name of the second SIMD&FP source register holding the scalar. If `<dt>` is `S16`, `Dm` is restricted to D0-D7. `Dm` is encoded in "Vm<2:0>", and `x` is encoded in "M:Vm<3>". If `<dt>` is `S32`, `Dm` is restricted to D0-D15. `Dm` is encoded in "Vm", and `x` is encoded in "M".

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if scalar_form then op2 = SInt(Elem[D[m],index,esize]);
    for r = 0 to regs-1
        for e = 0 to elements-1
            if !scalar_form then op2 = SInt(Elem[D[m+r],e,esize]);
            op1 = SInt(Elem[D[n+r],e,esize]);
            // The following only saturates if both op1 and op2 equal -(2^(esize-1))
            (result, sat) = SignedSatQ((2*op1*op2) >> esize, esize);
            Elem[D[d+r],e,esize] = result;
            if sat then FPSCR.QC = '1';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.176 VQDMULL

Vector Saturating Doubling Multiply Long multiplies corresponding elements in two doubleword vectors, doubles the products, and places the results in a quadword vector.

The second operand can be a scalar instead of a vector. For more information about scalars see *Advanced SIMD scalars* on page F2-4106.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-3951.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	0	1	D	!=11 (size)		Vn	Vd	1	1	0	1	N	0	M	0	Vm

A1 variant

```
VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
scalar_form = FALSE;  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 8 << UInt(size);  elements = 64 DIV esize;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	0	1	D	!=11 (size)		Vn	Vd	1	0	1	1	N	1	M	0	Vm

A2 variant

```
VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	0	1	1	1	1	1	D	!=11 (size)		Vn	Vd	1	1	0	1	N	0	M	0	Vm

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1 variant

```
VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
scalar_form = FALSE;  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 8 << UInt(size);  elements = 64 DIV esize;
```

T2

15 14 13 12	11 10 9 8	7	6	5 4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1	0	1 1 1 1 1	D	!=11 (size)	Vn	Vd	1 0 1 1	N	1	M	0	Vm

T2 variant

```
VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

Notes for all encodings

Related encodings: See *Advanced SIMD data-processing* on page F3-4137 for the T32 instruction set, or *Advanced SIMD data-processing* on page F4-4225 for the A32 instruction set.

Assembler symbols

`<c>` For encoding A1 and A2: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1 and T2: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:

`S16` when `size = 01`

`S32` when `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm[x]>` Is the 64-bit name of the second SIMD&FP source register holding the scalar. If `<dt>` is `S16`, `Dm` is restricted to D0-D7. `Dm` is encoded in "Vm<2:0>", and `x` is encoded in "M:Vm<3>". If `<dt>` is `S32`, `Dm` is restricted to D0-D15. `Dm` is encoded in "Vm", and `x` is encoded in "M".

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if scalar_form then op2 = SInt(Elem[Din[m],index,esize]);
    for e = 0 to elements-1
        if !scalar_form then op2 = SInt(Elem[Din[m],e,esize]);
        op1 = SInt(Elem[Din[n],e,esize]);
        // The following only saturates if both op1 and op2 equal -(2^(esize-1))
        (product, sat) = SignedSatQ(2*op1*op2, 2*esize);
        Elem[Q[d>>1],e,2*esize] = product;
        if sat then FPSCR.QC = '1';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.177 VQMOVN, VQMOVUN

Vector Saturating Move and Narrow copies each element of the operand vector to the corresponding element of the destination vector.

The operand is a quadword vector. The elements can be any one of:

- 16-bit, 32-bit, or 64-bit signed integers.
- 16-bit, 32-bit, or 64-bit unsigned integers.

The result is a doubleword vector. The elements are half the length of the operand vector elements. If the operand is unsigned, the results are unsigned. If the operand is signed, the results can be signed or unsigned.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-3951.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

This instruction is used by the pseudo-instructions VQRSHRN (zero), VQRSHRUN (zero), VQSHRN (zero), and VQSHRUN (zero). The pseudo-instruction is never the preferred disassembly.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19 18	17	16	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	1	0	0	1	1	1	D	1	1	size	1	0	Vd	0	0	1	0	op		M	0	Vm

Signed result variant

Applies when `op == 1x`.

```
VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

Unsigned result variant

Applies when `op == 01`.

```
VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

Decode for all variants of this encoding

```
if op == '00' then SEE "VMOVN";
if size == '11' || Vm<0> == '1' then UNDEFINED;
src_unsigned = (op == '11');  dest_unsigned = (op<0> == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 2	1	0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	1	1	1	1	1	1	D	1	1	size	1	0	Vd	0	0	1	0	op		M	0	Vm

Signed result variant

Applies when `op == 1x`.

```
VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Unsigned result variant

Applies when `op == 01`.

```
VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

Decode for all variants of this encoding

```
if op == '00' then SEE "VMOVN";
if size == '11' || Vm<0> == '1' then UNDEFINED;
src_unsigned = (op == '11');  dest_unsigned = (op<0> == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);
```

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` For the signed result variant: is the data type for the elements of the operand, encoded in the "op<0>:size" field. It can have the following values:

`S16`	when `op<0> = 0`, `size = 00`
`S32`	when `op<0> = 0`, `size = 01`
`S64`	when `op<0> = 0`, `size = 10`
`U16`	when `op<0> = 1`, `size = 00`
`U32`	when `op<0> = 1`, `size = 01`
`U64`	when `op<0> = 1`, `size = 10`

The following encodings are reserved:

- `op<0> = 0`, `size = 11`.
- `op<0> = 1`, `size = 11`.

For the unsigned result variant: is the data type for the elements of the operand, encoded in the "size" field. It can have the following values:

`S16`	when `size = 00`
`S32`	when `size = 01`
`S64`	when `size = 10`

The encoding `size = 11` is reserved.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        operand = Int(Elem[Qin[m>>1],e,2*esize], src_unsigned);
        (Elem[D[d],e,esize], sat) = SatQ(operand, esize, dest_unsigned);
        if sat then FPSCR.QC = '1';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.178 VQNEG

Vector Saturating Negate negates each element in a vector, and places the results in the destination vector.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-3951.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 18	17 16	15 12	11	10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	1	1	size	0 0	Vd	0	1 1 1	1	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VQNEG{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VQNEG{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 2 1 0	15 12	11	10 9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	D	1	1	size	0 0	Vd	0	1 1 1	1	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VQNEG{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VQNEG{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:

`S8`	when `size = 00`
`S16`	when `size = 01`
`S32`	when `size = 10`

The encoding `size = 11` is reserved.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = -SInt(Elem[D[m+r],e,esize]);
            (Elem[D[d+r],e,esize], sat) = SignedSatQ(result, esize);
            if sat then FPSCR.QC = '1';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.179 VQRDMLAH

Vector Saturating Rounding Doubling Multiply Accumulate Returning High Half. This instruction multiplies the vector elements of the first source SIMD&FP register with either the corresponding vector elements of the second source SIMD&FP register or the value of a vector element of the second source SIMD&FP register, without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-3951.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20	19 … 16	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	1	0	0	1	1	0	D	size		Vn	Vd	1	0	1	1	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VQRDMLAH{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VQRDMLAH{<q>}.<dt> <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if !HaveQRDMLAHExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
add = TRUE;  scalar_form = FALSE;  esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

A2

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20	19 … 16	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	1	0	0	1	Q	1	D	!=11		Vn	Vd	1	1	1	0	N	1	M	0	Vm
										size												

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VQRDMLAH{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VQRDMLAH{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if !HaveQRDMLAHExt() then UNDEFINED;
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
add = TRUE;  scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

T1

ARMv8.1

15 14 13 12	11 10 9 8	7	6	5 4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 1 0	D	size		Vn	Vd	1 0 1 1	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VQRDMLAH{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VQRDMLAH{<q>}.<dt> <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if !HaveQRDMLAHExt() then UNDEFINED;
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
add = TRUE;  scalar_form = FALSE;  esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

ARMv8.1

15 14 13 12	11 10 9 8	7	6	5 4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 Q	1 1 1 1 1	D	!=11		Vn	Vd	1 1 1 0	N	1	M	0	Vm
			size									

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VQRDMLAH{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VQRDMLAH{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>
```

Decode for all variants of this encoding

```
if !HaveQRDMLAHExt() then UNDEFINED;
if InITBlock() then UNPREDICTABLE;
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
add = TRUE;  scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings

Related encodings: See *Advanced SIMD data-processing* on page F3-4137 for the T32 instruction set, or *Advanced SIMD data-processing* on page F4-4225 for the A32 instruction set.

Assembler symbols

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:

`S16`	when `size = 01`
`S32`	when `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm[x]>` Is the 64-bit name of the second SIMD&FP source register holding the scalar. If `<dt>` is `S16`, `Dm` is restricted to D0-D7. `Dm` is encoded in "Vm<2:0>", and `x` is encoded in "M:Vm<3>". If `<dt>` is `S32`, `Dm` is restricted to D0-D15. `Dm` is encoded in "Vm", and `x` is encoded in "M".

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
EncodingSpecificOperations();  CheckAdvSIMDEnabled();
round_const = 1 << (esize-1);
if scalar_form then op2 = SInt(Elem[D[m],index,esize]);
for r = 0 to regs-1
    for e = 0 to elements-1
        op1 = SInt(Elem[D[n+r],e,esize]);
        op3 = SInt(Elem[D[d+r],e,esize]) << esize;
        if !scalar_form then op2 = SInt(Elem[D[m+r],e,esize]);
        (result, sat) = SignedSatQ((op3 + 2*(op1*op2) + round_const) >> esize, esize);
        Elem[D[d+r],e,esize] = result;
        if sat then FPSCR.QC = '1';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.180 VQRDMLSH

Vector Saturating Rounding Doubling Multiply Subtract Returning High Half. This instruction multiplies the vector elements of the first source SIMD&FP register with either the corresponding vector elements of the second source SIMD&FP register or the value of a vector element of the second source SIMD&FP register, without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-3951.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	1	0	D	size		Vn	Vd	1	1	0	0	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VQRDMLSH{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VQRDMLSH{<q>}.<dt> <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if !HaveQRDMLAHExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
add = FALSE;  scalar_form = FALSE;  esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

A2

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	Q	1	D	!=11		Vn	Vd	1	1	1	1	N	1	M	0	Vm
										size												

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VQRDMLSH{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VQRDMLSH{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if !HaveQRDMLAHExt() then UNDEFINED;
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
add = FALSE;  scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

T1

ARMv8.1

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	1	1	1	1	1	0	D	size		Vn	Vd	1	1	0	0	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VQRDMLSH{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VQRDMLSH{<q>}.<dt> <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if !HaveQRDMLAHExt() then UNDEFINED;
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
add = FALSE;  scalar_form = FALSE;  esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

ARMv8.1

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	Q	1	1	1	1	1	D	!=11		Vn	Vd	1	1	1	1	N	1	M	0	Vm
										size												

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VQRDMLSH{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VQRDMLSH{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>
```

Decode for all variants of this encoding

```
if !HaveQRDMLAHExt() then UNDEFINED;
if InITBlock() then UNPREDICTABLE;
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
add = FALSE;  scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings

Related encodings: See *Advanced SIMD data-processing* on page F3-4137 for the T32 instruction set, or *Advanced SIMD data-processing* on page F4-4225 for the A32 instruction set.

Assembler symbols

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:

`S16`	when `size = 01`
`S32`	when `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm[x]>` Is the 64-bit name of the second SIMD&FP source register holding the scalar. If `<dt>` is `S16`, `Dm` is restricted to D0-D7. `Dm` is encoded in "Vm<2:0>", and `x` is encoded in "M:Vm<3>". If `<dt>` is `S32`, `Dm` is restricted to D0-D15. `Dm` is encoded in "Vm", and `x` is encoded in "M".

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
EncodingSpecificOperations();  CheckAdvSIMDEnabled();
round_const = 1 << (esize-1);
if scalar_form then op2 = SInt(Elem[D[m],index,esize]);
for r = 0 to regs-1
    for e = 0 to elements-1
        op1 = SInt(Elem[D[n+r],e,esize]);
        op3 = SInt(Elem[D[d+r],e,esize]) << esize;
        if !scalar_form then op2 = SInt(Elem[D[m+r],e,esize]);
        (result, sat) = SignedSatQ((op3 - 2*(op1*op2) + round_const) >> esize, esize);
        Elem[D[d+r],e,esize] = result;
        if sat then FPSCR.QC = '1';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.181 VQRDMULH

Vector Saturating Rounding Doubling Multiply Returning High Half multiplies corresponding elements in two vectors, doubles the results, and places the most significant half of the final results in the destination vector. The results are rounded. For truncated results see VQDMULH.

The second operand can be a scalar instead of a vector. For more information about scalars see *Advanced SIMD scalars* on page F2-4106.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-3951.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21 20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	0	D	size	Vn	Vd	1 0 1 1	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VQRDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VQRDMULH{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
scalar_form = FALSE;  esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

A2

31 30 29 28	27 26 25 24	23	22	21 20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 Q	1	D	!=11	Vn	Vd	1 1 0 1	N	1	M	0	Vm
				size								

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VQRDMULH{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm[x]>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VQRDMULH{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm[x]>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

T1

15 14 13 12	11 10 9 8	7	6	5 4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 1 0	D	size		Vn	Vd	1 0 1 1	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VQRDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VQRDMULH{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
scalar_form = FALSE;  esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T2

15 14 13 12	11 10 9 8	7	6	5 4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 Q	1 1 1 1 1	D	!=11		Vn	Vd	1 1 0 1	N	1	M	0	Vm
			size									

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VQRDMULH{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm[x]>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VQRDMULH{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm[x]>
```

Decode for all variants of this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Notes for all encodings

Related encodings: See *Advanced SIMD data-processing* on page F3-4137 for the T32 instruction set, or *Advanced SIMD data-processing* on page F4-4225 for the A32 instruction set.

Assembler symbols

`<c>` For encoding A1 and A2: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1 and T2: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:

`S16` when `size = 01`

`S32` when `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm[x]>` Is the 64-bit name of the second SIMD&FP source register holding the scalar. If `<dt>` is `S16`, `Dm` is restricted to D0-D7. `Dm` is encoded in "Vm<2:0>", and `x` is encoded in "M:Vm<3>". If `<dt>` is `S32`, `Dm` is restricted to D0-D15. `Dm` is encoded in "Vm", and `x` is encoded in "M".

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (esize-1);
    if scalar_form then op2 = SInt(Elem[D[m],index,esize]);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = SInt(Elem[D[n+r],e,esize]);
            if !scalar_form then op2 = SInt(Elem[D[m+r],e,esize]);
            (result, sat) = SignedSatQ((2*op1*op2 + round_const) >> esize, esize);
            Elem[D[d+r],e,esize] = result;
            if sat then FPSCR.QC = '1';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.182 VQRSHL

Vector Saturating Rounding Shift Left takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift.

For truncated results see VQSHL (register).

The first operand and result elements are the same data type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The second operand is a signed integer of the same size.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-3951.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21 20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1	U	0	D	size	Vn	Vd	0 1 0 1	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VQRSHL{<c>}{<q>}.<dt> {<Dd>,} <Dm>, <Dn>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VQRSHL{<c>}{<q>}.<dt> {<Qd>,} <Qm>, <Qn>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1	U	1 1 1 1	0	D	size	Vn	Vd	0 1 0 1	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VQRSHL{<c>}{<q>}.<dt> {<Dd>,} <Dm>, <Dn>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VQRSHL{<c>}{<q>}.<dt> {<Qd>,} <Qm>, <Qn>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
```

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "U:size" field. It can have the following values:

`S8`	when `U = 0`, `size = 00`
`S16`	when `U = 0`, `size = 01`
`S32`	when `U = 0`, `size = 10`
`S64`	when `U = 0`, `size = 11`
`U8`	when `U = 1`, `size = 00`
`U16`	when `U = 1`, `size = 01`
`U32`	when `U = 1`, `size = 10`
`U64`	when `U = 1`, `size = 11`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            shift = SInt(Elem[D[n+r],e,esize]<7:0>);
            round_const = 1 << (-1-shift); // 0 for left shift, 2^(n-1) for right shift
            operand = Int(Elem[D[m+r],e,esize], unsigned);
            (result, sat) = SatQ((operand + round_const) << shift, esize, unsigned);
            Elem[D[d+r],e,esize] = result;
            if sat then FPSCR.QC = '1';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.183 VQRSHRN (zero)

Vector Saturating Rounding Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the signed rounded results in a doubleword vector

This instruction is a pseudo-instruction of the VQMOVN, VQMOVUN instruction. This means that:

- The encodings in this description are named to match the encodings of VQMOVN, VQMOVUN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	0	Vd		0	0	1	0	1	x	M	0	Vm	
																						op					

Signed result variant

```
VQRSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

and is never the preferred disassembly.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	0	Vd		0	0	1	0	1	x	M	0	Vm	
																						op					

Signed result variant

```
VQRSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

and is never the preferred disassembly.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the operand, encoded in the "op<0>:size" field. It can have the following values:

`S16`	when `op<0> = 0, size = 00`
`S32`	when `op<0> = 0, size = 01`
`S64`	when `op<0> = 0, size = 10`
`U16`	when `op<0> = 1, size = 00`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`U32` when `op<0> = 1`, `size = 01`

`U64` when `op<0> = 1`, `size = 10`

The following encodings are reserved:

- `op<0> = 0`, `size = 11`.
- `op<0> = 1`, `size = 11`.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation for all encodings

The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.184 VQRSHRN, VQRSHRUN

Vector Saturating Rounding Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the rounded results in a doubleword vector.

For truncated results, see VQSHL (register).

The operand elements must all be the same size, and can be any one of:

- 16-bit, 32-bit, or 64-bit signed integers.
- 16-bit, 32-bit, or 64-bit unsigned integers.

The result elements are half the width of the operand elements. If the operand elements are signed, the results can be either signed or unsigned. If the operand elements are unsigned, the result elements must also be unsigned.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-3951.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21 – 16	15 – 12	11	10	9	8	7	6	5	4	3 – 0
1	1	1	1	0	0	1	U	1	D	imm6	Vd	1	0	0	op	0	1	M	1	Vm

Signed result variant

Applies when `!(imm6 == 000xxx) && op == 1`.

```
VQRSHRN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>
```

Unsigned result variant

Applies when `U == 1 && !(imm6 == 000xxx) && op == 0`.

```
VQRSHRUN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>
```

Decode for all variants of this encoding

```
if imm6 == '000xxx' then SEE "Related encodings";
if U == '0' && op == '0' then SEE "VRSHRN";
if Vm<0> == '1' then UNDEFINED;
case imm6 of
    when '001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '01xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
src_unsigned = (U == '1' && op == '1');  dest_unsigned = (U == '1');
d = UInt(D:Vd);  m = UInt(M:Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5 – 0	15 – 12	11	10	9	8	7	6	5	4	3 – 0
1	1	1	U	1	1	1	1	1	D	imm6	Vd	1	0	0	op	0	1	M	1	Vm

Signed result variant

Applies when `!(imm6 == 000xxx) && op == 1`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VQRSHRN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>
```

Unsigned result variant

Applies when `U == 1 && !(imm6 == 000xxx) && op == 0`.

```
VQRSHRUN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>
```

Decode for all variants of this encoding

```
if imm6 == '000xxx' then SEE "Related encodings";
if U == '0' && op == '0' then SEE "VRSHRN";
if Vm<0> == '1' then UNDEFINED;
case imm6 of
    when '001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '01xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
src_unsigned = (U == '1' && op == '1');  dest_unsigned = (U == '1');
d = UInt(D:Vd);  m = UInt(M:Vm);
```

Notes for all encodings

Related encodings: See *Advanced SIMD one register and modified immediate* on page F3-4146 for the T32 instruction set, or *Advanced SIMD one register and modified immediate* on page F4-4233 for the A32 instruction set.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<type>` For the signed result variant: is the data type for the elements of the vectors, encoded in the "U" field. It can have the following values:

`S`	when `U = 0`
`U`	when `U = 1`

For the unsigned result variant: is the data type for the elements of the vectors, encoded in the "U" field. It can have the following values:

`S`	when `U = 1`

`<size>` Is the data size for the elements of the vectors, encoded in the "imm6<5:3>" field. It can have the following values:

`16`	when `imm6<5:3> = 001`
`32`	when `imm6<5:3> = 01x`
`64`	when `imm6<5:3> = 1xx`

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<imm>` Is an immediate value, in the range 1 to `<size>`/2, encoded in the "imm6" field as `<size>`/2 - `<imm>`.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (shift_amount - 1);
    for e = 0 to elements-1
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
operand = Int(Elem[Qin[m>>1],e,2*esize], src_unsigned);
(result, sat) = SatQ((operand + round_const) >> shift_amount, esize, dest_unsigned);
Elem[D[d],e,esize] = result;
if sat then FPSCR.QC = '1';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.185 VQRSHRUN (zero)

Vector Saturating Rounding Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the unsigned rounded results in a doubleword vector

This instruction is a pseudo-instruction of the VQMOVN, VQMOVUN instruction. This means that:

- The encodings in this description are named to match the encodings of VQMOVN, VQMOVUN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 18	17 16	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	1	1	size	1	0	Vd	0	0	1	0	0	1	M	0	Vm
														op			

Unsigned result variant

```
VQRSHRUN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

and is never the preferred disassembly.

T1

15 14 13 12	11 10 9 8	7	6	5	4	3	2	1	0	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	1	D	1	1	size		1	0	Vd	0	0	1	0	0	1	M	0	Vm
															op				

Unsigned result variant

```
VQRSHRUN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

and is never the preferred disassembly.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the operand, encoded in the "size" field. It can have the following values:

`S16` when `size = 00`

`S32` when `size = 01`

`S64` when `size = 10`

The encoding `size = 11` is reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation for all encodings

The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.186 VQSHL, VQSHLU (immediate)

Vector Saturating Shift Left (immediate) takes each element in a vector of integers, left shifts them by an immediate value, and places the results in a second vector.

The operand elements must all be the same size, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The result elements are the same size as the operand elements. If the operand elements are signed, the results can be either signed or unsigned. If the operand elements are unsigned, the result elements must also be unsigned.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-3951.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 U	1	D	imm6		Vd	0 1 1 op	L	Q	M	1	Vm

VQSHL,double,signed-result variant

Applies when `!(imm6 == 000xxx && L == 0) && op == 1 && Q == 0`.

```
VQSHL{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

VQSHL,quad,signed-result variant

Applies when `!(imm6 == 000xxx && L == 0) && op == 1 && Q == 1`.

```
VQSHL{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

VQSHLU,double,unsigned-result variant

Applies when `U == 1 && !(imm6 == 000xxx && L == 0) && op == 0 && Q == 0`.

```
VQSHLU{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

VQSHLU,quad,unsigned-result variant

Applies when `U == 1 && !(imm6 == 000xxx && L == 0) && op == 0 && Q == 1`.

```
VQSHLU{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

Decode for all variants of this encoding

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if U == '0' && op == '0' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = UInt(imm6);
src_unsigned = (U == '1' && op == '1');  dest_unsigned = (U == '1');
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15	14	13	12	11	10	9	8	7	6	5	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	U	1	1	1	1	1	D	imm6		Vd		0	1	1	op	L	Q	M	1	Vm	

VQSHL,double,signed-result variant

Applies when `!(imm6 == 000xxx && L == 0) && op == 1 && Q == 0`.

```
VQSHL{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

VQSHL,quad,signed-result variant

Applies when `!(imm6 == 000xxx && L == 0) && op == 1 && Q == 1`.

```
VQSHL{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

VQSHLU,double,unsigned-result variant

Applies when `U == 1 && !(imm6 == 000xxx && L == 0) && op == 0 && Q == 0`.

```
VQSHLU{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

VQSHLU,quad,unsigned-result variant

Applies when `U == 1 && !(imm6 == 000xxx && L == 0) && op == 0 && Q == 1`.

```
VQSHLU{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

Decode for all variants of this encoding

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if U == '0' && op == '0' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = UInt(imm6);
src_unsigned = (U == '1' && op == '1');  dest_unsigned = (U == '1');
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Notes for all encodings

Related encodings: See *Advanced SIMD one register and modified immediate* on page F3-4146 for the T32 instruction set, or *Advanced SIMD one register and modified immediate* on page F4-4233 for the A32 instruction set.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<type>` Is the data type for the elements of the vectors, encoded in the "U" field. It can have the following values:

`S`	when `U = 0`
`U`	when `U = 1`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<size>` Is the data size for the elements of the vectors, encoded in the "L:imm6<5:3>" field. It can have the following values:

8	when `L = 0`, `imm6<5:3> = 001`
16	when `L = 0`, `imm6<5:3> = 01x`
32	when `L = 0`, `imm6<5:3> = 1xx`
64	when `L = 1`, `imm6<5:3> = xxx`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

`<imm>` Is an immediate value, in the range 0 to `<size>`-1, encoded in the "imm6" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            operand = Int(Elem[D[m+r],e,esize], src_unsigned);
            (result, sat) = SatQ(operand << shift_amount, esize, dest_unsigned);
            Elem[D[d+r],e,esize] = result;
            if sat then FPSCR.QC = '1';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.187 VQSHL (register)

Vector Saturating Shift Left (register) takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift.

The results are truncated. For rounded results, see VQRSHL.

The first operand and result elements are the same data type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The second operand is a signed integer of the same size.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-3951.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21 20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 0 1	U	0	D	size	Vn	Vd	0 1 0 0	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VQSHL{<c>}{<q>}.<dt> {<Dd>,} <Dm>, <Dn>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VQSHL{<c>}{<q>}.<dt> {<Qd>,} <Qm>, <Qn>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1	U	1 1 1 1	0	D	size	Vn	Vd	0 1 0 0	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VQSHL{<c>}{<q>}.<dt> {<Dd>,} <Dm>, <Dn>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VQSHL{<c>}{<q>}.<dt> {<Qd>,} <Qm>, <Qn>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
```

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "U:size" field. It can have the following values:

`S8`	when `U = 0`, `size = 00`
`S16`	when `U = 0`, `size = 01`
`S32`	when `U = 0`, `size = 10`
`S64`	when `U = 0`, `size = 11`
`U8`	when `U = 1`, `size = 00`
`U16`	when `U = 1`, `size = 01`
`U32`	when `U = 1`, `size = 10`
`U64`	when `U = 1`, `size = 11`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            shift = SInt(Elem[D[n+r],e,esize]<7:0>);
            operand = Int(Elem[D[m+r],e,esize], unsigned);
            (result,sat) = SatQ(operand << shift, esize, unsigned);
            Elem[D[d+r],e,esize] = result;
            if sat then FPSCR.QC = '1';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.188 VQSHRN (zero)

Vector Saturating Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the signed truncated results in a doubleword vector

This instruction is a pseudo-instruction of the VQMOVN, VQMOVUN instruction. This means that:

- The encodings in this description are named to match the encodings of VQMOVN, VQMOVUN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	0	Vd		0	0	1	0	1	x	M	0	Vm	
																						op					

Signed result variant

```
VQSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

and is never the preferred disassembly.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	0	Vd		0	0	1	0	1	x	M	0	Vm	
																						op					

Signed result variant

```
VQSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

and is never the preferred disassembly.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the operand, encoded in the "op<0>:size" field. It can have the following values:

`S16`	when `op<0> = 0, size = 00`
`S32`	when `op<0> = 0, size = 01`
`S64`	when `op<0> = 0, size = 10`
`U16`	when `op<0> = 1, size = 00`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`U32` when `op<0> = 1`, `size = 01`

`U64` when `op<0> = 1`, `size = 10`

The following encodings are reserved:

- `op<0> = 0`, `size = 11`.
- `op<0> = 1`, `size = 11`.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation for all encodings

The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.189 VQSHRN, VQSHRUN

Vector Saturating Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the truncated results in a doubleword vector.

For rounded results, see VQRSHRN, VQRSHRUN.

The operand elements must all be the same size, and can be any one of:

- 16-bit, 32-bit, or 64-bit signed integers.
- 16-bit, 32-bit, or 64-bit unsigned integers.

The result elements are half the width of the operand elements. If the operand elements are signed, the results can be either signed or unsigned. If the operand elements are unsigned, the result elements must also be unsigned.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-3951.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21 – 16	15 – 12	11	10	9	8	7	6	5	4	3 – 0
1	1	1	1	0	0	1	U	1	D	imm6	Vd	1	0	0	op	0	0	M	1	Vm

Signed result variant

Applies when `!(imm6 == 000xxx) && op == 1`.

```
VQSHRN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>
```

Unsigned result variant

Applies when `U == 1 && !(imm6 == 000xxx) && op == 0`.

```
VQSHRUN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>
```

Decode for all variants of this encoding

```
if imm6 == '000xxx' then SEE "Related encodings";
if U == '0' && op == '0' then SEE "VSHRN";
if Vm<0> == '1' then UNDEFINED;
case imm6 of
    when '001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '01xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
src_unsigned = (U == '1' && op == '1');  dest_unsigned = (U == '1');
d = UInt(D:Vd);  m = UInt(M:Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5 – 0	15 – 12	11	10	9	8	7	6	5	4	3 – 0
1	1	1	U	1	1	1	1	1	D	imm6	Vd	1	0	0	op	0	0	M	1	Vm

Signed result variant

Applies when `!(imm6 == 000xxx) && op == 1`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VQSHRN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>
```

Unsigned result variant

Applies when `U == 1 && !(imm6 == 000xxx) && op == 0`.

```
VQSHRUN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>
```

Decode for all variants of this encoding

```
if imm6 == '000xxx' then SEE "Related encodings";
if U == '0' && op == '0' then SEE "VSHRN";
if Vm<0> == '1' then UNDEFINED;
case imm6 of
    when '001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '01xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
src_unsigned = (U == '1' && op == '1');  dest_unsigned = (U == '1');
d = UInt(D:Vd);  m = UInt(M:Vm);
```

Notes for all encodings

Related encodings: See *Advanced SIMD one register and modified immediate* on page F3-4146 for the T32 instruction set, or *Advanced SIMD one register and modified immediate* on page F4-4233 for the A32 instruction set.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<type>` For the signed result variant: is the data type for the elements of the vectors, encoded in the "U" field. It can have the following values:

`S`	when `U = 0`
`U`	when `U = 1`

For the unsigned result variant: is the data type for the elements of the vectors, encoded in the "U" field. It can have the following values:

`S`	when `U = 1`

`<size>` Is the data size for the elements of the vectors, encoded in the "imm6<5:3>" field. It can have the following values:

`16`	when `imm6<5:3> = 001`
`32`	when `imm6<5:3> = 01x`
`64`	when `imm6<5:3> = 1xx`

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<imm>` Is an immediate value, in the range 1 to `<size>`/2, encoded in the "imm6" field as `<size>`/2 - `<imm>`.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        operand = Int(Elem[Qin[m>>1],e,2*esize], src_unsigned);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
(result, sat) = SatQ(operand >> shift_amount, esize, dest_unsigned);
Elem[D[d],e,esize] = result;
if sat then FPSCR.QC = '1';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.190 VQSHRUN (zero)

Vector Saturating Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the unsigned truncated results in a doubleword vector

This instruction is a pseudo-instruction of the VQMOVN, VQMOVUN instruction. This means that:

- The encodings in this description are named to match the encodings of VQMOVN, VQMOVUN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 18	17 16	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	1	1	size	1	0	Vd	0	0	1	0	0	1	M	0	Vm
														op			

Unsigned result variant

```
VQSHRUN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

and is never the preferred disassembly.

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 2	1	0	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	1	D	1	1	size	1	0	Vd	0	0	1	0	0	1	M	0	Vm
														op				

Unsigned result variant

```
VQSHRUN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

and is never the preferred disassembly.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the operand, encoded in the "size" field. It can have the following values:

`S16` when `size = 00`

`S32` when `size = 01`

`S64` when `size = 10`

The encoding `size = 11` is reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation for all encodings

The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.191 VQSUB

Vector Saturating Subtract subtracts the elements of the second operand vector from the corresponding elements of the first operand vector, and places the results in the destination vector. Signed and unsigned operations are distinct.

The operand and result elements must all be the same type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-3951.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 0 1	U	0	D	size	Vn	Vd	0 0 1 0	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VQSUB{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VQSUB{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1	U	1 1 1 1	0	D	size	Vn	Vd	0 0 1 0	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VQSUB{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VQSUB{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Qm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "U:size" field. It can have the following values:

`S8`	when `U = 0`, `size = 00`
`S16`	when `U = 0`, `size = 01`
`S32`	when `U = 0`, `size = 10`
`S64`	when `U = 0`, `size = 11`
`U8`	when `U = 1`, `size = 00`
`U16`	when `U = 1`, `size = 01`
`U32`	when `U = 1`, `size = 10`
`U64`	when `U = 1`, `size = 11`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            diff = Int(Elem[D[n+r],e,esize], unsigned) - Int(Elem[D[m+r],e,esize], unsigned);
            (Elem[D[d+r],e,esize], sat) = SatQ(diff, esize, unsigned);
            if sat then FPSCR.QC = '1';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.192 VRADDHN

Vector Rounding Add and Narrow, returning High Half adds corresponding elements in two quadword vectors, and places the most significant half of each result in a doubleword vector. The results are rounded. For truncated results, see VADDHN.

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	!=11 (size)		Vn	Vd	0 1 0 0	N	0	M	0	Vm

A1 variant

```
VRADDHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	1	D	!=11 (size)		Vn	Vd	0 1 0 0	N	0	M	0	Vm

T1 variant

```
VRADDHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

Notes for all encodings

Related encodings: See *Advanced SIMD data-processing* on page F3-4137 for the T32 instruction set, or *Advanced SIMD data-processing* on page F4-4225 for the A32 instruction set.

Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<dt>` Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:

`I16` when `size = 00`

`I32` when `size = 01`

`I64` when `size = 10`

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (esize-1);
    for e = 0 to elements-1
        result = Elem[Qin[n>>1],e,2*esize] + Elem[Qin[m>>1],e,2*esize] + round_const;
        Elem[D[d],e,esize] = result<2*esize-1:esize>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.193 VRECPE

Vector Reciprocal Estimate finds an approximate reciprocal of each element in the operand vector, and places the results in the destination vector.

The operand and result elements are the same type, and can be floating-point numbers or unsigned integers.

For details of the operation performed by this instruction see *Floating-point reciprocal square root estimate and step* on page E1-3969.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 12	11 10 9 8	7 6 5 4	3 0
1 1 1 1	0 0 1 1 1	D 1 1	size 1 1	Vd	0 1 0 F	0 Q M 0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VRECPE{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VRECPE{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
floating_point = (F == '1');
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 12	11 10 9 8	7 6 5 4	3 0
1 1 1 1	1 1 1 1 1	D 1 1	size 1 1	Vd	0 1 0 F	0 Q M 0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VRECPE{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VRECPE{<c>}{<q>}.<dt> <Qd>, <Qm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
floating_point = (F == '1');
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "F:size" field. It can have the following values:

`U32`	when `F = 0`, `size = 10`
`F16`	when `F = 1`, `size = 01`
`F32`	when `F = 1`, `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Newton-Raphson iteration

For details of the operation performed and how it can be used in a Newton-Raphson iteration to calculate the reciprocal of a number, see *Floating-point reciprocal estimate and step* on page E1-3969.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                Elem[D[d+r],e,esize] = FPRecipEstimate(Elem[D[m+r],e,esize], StandardFPSCRValue());
            else
                Elem[D[d+r],e,esize] = UnsignedRecipEstimate(Elem[D[m+r],e,esize]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.194 VRECPS

Vector Reciprocal Step multiplies the elements of one vector by the corresponding elements of another vector, subtracts each of the products from 2.0, and places the results into the elements of the destination vector.

The operand and result elements are floating-point numbers.

For details of the operation performed by this instruction see *Floating-point reciprocal estimate and step* on page E1-3969.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 0	0	D	0	sz	Vn	Vd	1 1 1 1	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VRECPS{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VRECPS{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 0	1 1 1 1 0	D	0	sz		Vn	Vd	1 1 1 1	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VRECPS{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VRECPS{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

`F32`	when `sz = 0`
`F16`	when `sz = 1`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Newton-Raphson iteration

For details of the operation performed and how it can be used in a Newton-Raphson iteration to calculate the reciprocal of a number, see *Floating-point reciprocal estimate and step* on page E1-3969.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = FPRecipStep(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize]);
```

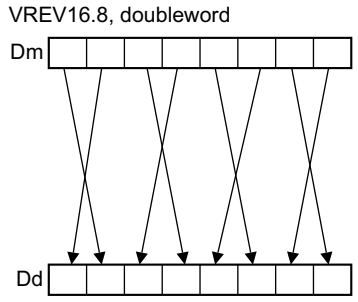

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.195 VREV16

Vector Reverse in halfwords reverses the order of 8-bit elements in each halfword of the vector, and places the result in the corresponding destination vector.

There is no distinction between data types, other than size.

The following figure shows the operation of VREV16 doubleword operation.



Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 18	17 16	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	1	1	size	0 0	Vd	0	0	0	1	0	Q	M	0	Vm
												op					

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VREV16{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VREV16{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if UInt(op)+UInt(size) >= 3 then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size);
integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;
integer containers = 64 DIV container_size;
integer elements_per_container = container_size DIV esize;

d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15 – 12	11	10	9	8	7	6	5	4	3 – 0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd	0	0	0	1	0	Q	M	0	Vm
																				op					

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VREV16{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VREV16{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if UInt(op)+UInt(size) >= 3 then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size);
integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;
integer containers = 64 DIV container_size;
integer elements_per_container = container_size DIV esize;

d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
```

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the operand, encoded in the "size" field. It can have the following values:

`8` when `size = 00`

The following encodings are reserved:

- `size = 01`.
- `size = 1x`.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();

    bits(64) result;
    integer element;
    integer rev_element;
    for r = 0 to regs-1
        element = 0;
        for c = 0 to containers-1
            rev_element = element + elements_per_container - 1;
            for e = 0 to elements_per_container-1
                Elem[result, rev_element, esize] = Elem[D[m+r], element, esize];
                element = element + 1;
                rev_element = rev_element - 1;
        D[d+r] = result;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

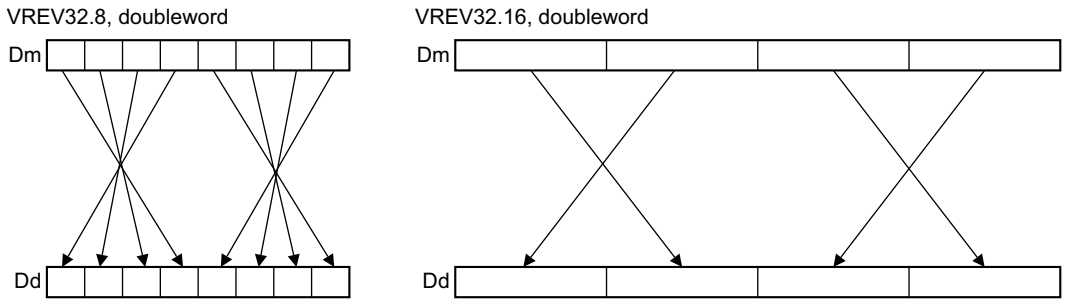
Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.196 VREV32

Vector Reverse in words reverses the order of 8-bit or 16-bit elements in each word of the vector, and places the result in the corresponding destination vector.

There is no distinction between data types, other than size.

The following figure shows the operation of VREV32 doubleword operations.



Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 18	17 16	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	1	1	size	0 0	Vd	0	0	0	0	1	Q	M	0	Vm
												op					

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VREV32{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VREV32{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if UInt(op)+UInt(size) >= 3 then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size);
integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;
integer containers = 64 DIV container_size;
integer elements_per_container = container_size DIV esize;

d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd		0	0	0	0	1	Q	M	0	Vm	
																					op						

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VREV32{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VREV32{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if UInt(op)+UInt(size) >= 3 then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size);
integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;
integer containers = 64 DIV container_size;
integer elements_per_container = container_size DIV esize;

d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
```

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the operand, encoded in the "size" field. It can have the following values:

`8`	when `size = 00`
`16`	when `size = 01`

The encoding `size = 1x` is reserved.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();

    bits(64) result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer element;
integer rev_element;
for r = 0 to regs-1
    element = 0;
    for c = 0 to containers-1
        rev_element = element + elements_per_container - 1;
        for e = 0 to elements_per_container-1
            Elem[result, rev_element, esize] = Elem[D[m+r], element, esize];
            element = element + 1;
            rev_element = rev_element - 1;
    D[d+r] = result;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

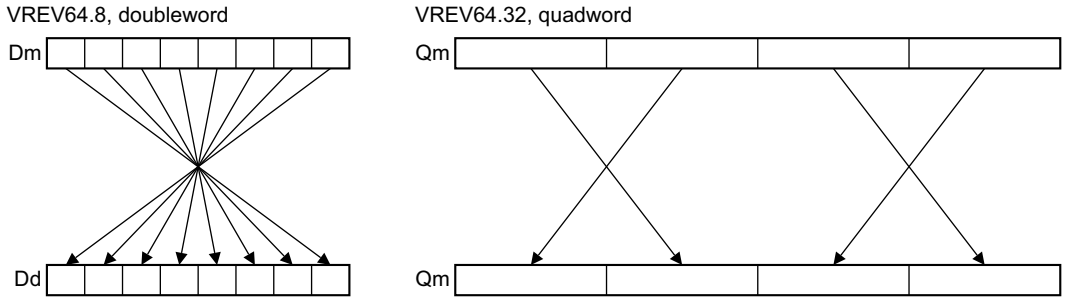
Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.197 VREV64

Vector Reverse in doublewords reverses the order of 8-bit, 16-bit, or 32-bit elements in each doubleword of the vector, and places the result in the corresponding destination vector.

There is no distinction between data types, other than size.

The following figure shows the operation of VREV64 doubleword operations.



Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 18	17 16	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	0 0 1 1 1	D		1	1	size	0 0	Vd	0	0	0	0	0	Q	M	0	Vm
												op					

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VREV64{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VREV64{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if UInt(op)+UInt(size) >= 3 then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size);
integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;
integer containers = 64 DIV container_size;
integer elements_per_container = container_size DIV esize;

d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd	0	0	0	0	0	Q	M	0	Vm
																				op					

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VREV64{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VREV64{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if UInt(op)+UInt(size) >= 3 then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size);
integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;
integer containers = 64 DIV container_size;
integer elements_per_container = container_size DIV esize;

d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
```

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the operand, encoded in the "size" field. It can have the following values:

`8`	when `size = 00`
`16`	when `size = 01`
`32`	when `size = 10`

The encoding `size = 11` is reserved.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();

    bits(64) result;
    integer element;
    integer rev_element;
    for r = 0 to regs-1
        element = 0;
        for c = 0 to containers-1
            rev_element = element + elements_per_container - 1;
            for e = 0 to elements_per_container-1
                Elem[result, rev_element, esize] = Elem[D[m+r], element, esize];
                element = element + 1;
                rev_element = rev_element - 1;
        D[d+r] = result;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.198 VRHADD

Vector Rounding Halving Add adds corresponding elements in two vectors of integers, shifts each result right one bit, and places the final results in the destination vector.

The operand and result elements are all the same type, and can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

The results of the halving operations are rounded. For truncated results, see VHADD.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21 20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 0 1	U	0	D	size	Vn	Vd	0 0 0 1	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VRHADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VRHADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13	12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1	U	1 1 1 1 0	D	size	Vn	Vd	0 0 0 1	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VRHADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VRHADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

`S8`	when `U = 0`, `size = 00`
`S16`	when `U = 0`, `size = 01`
`S32`	when `U = 0`, `size = 10`
`U8`	when `U = 1`, `size = 00`
`U16`	when `U = 1`, `size = 01`
`U32`	when `U = 1`, `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Int(Elem[D[n+r],e,esize], unsigned);
            op2 = Int(Elem[D[m+r],e,esize], unsigned);
            result = op1 + op2 + 1;
            Elem[D[d+r],e,esize] = result<esize:1>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.199 VRINTA (Advanced SIMD)

Vector Round floating-point to integer towards Nearest with Ties to Away rounds a vector of floating-point values to integral floating-point values of the same size using the Round to Nearest with Ties to Away rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 18	17 16	15 12	11	10	9 7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	1	1	size	1 0	Vd	0	1	0 1 0	Q	M	0	Vm
											op				

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VRINTA{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VRINTA{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if op<2> != op<0> then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>));  exact = FALSE;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 2	1 0	15 12	11	10	9 7	6	5	4	3 0
1 1 1 1	1 1 1 1	1	D	1	1	size	1 0	Vd	0	1	0 1 0	Q	M	0	Vm
											op				

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VRINTA{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VRINTA{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if op<2> != op<0> then SEE "Related encodings";
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>));  exact = FALSE;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings

Related encodings: See *Advanced SIMD two registers misc* on page F3-4141 for the T32 instruction set, or *Advanced SIMD two registers misc* on page F4-4228 for the A32 instruction set.

Assembler symbols

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:

`F16`	when `size = 01`
`F32`	when `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
for r = 0 to regs-1
    for e = 0 to elements-1
        op1 = Elem[D[m+r],e,esize];
        result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);
        Elem[D[d+r],e,esize] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.200 VRINTA (floating-point)

Round floating-point to integer to Nearest with Ties to Away rounds a floating-point value to an integral floating-point value of the same size using the Round to Nearest with Ties to Away rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A1

31 30 29 28	27 26 25 24	23	22 21 20	19	18 17 16	15 12	11 10	9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	1	D 1 1 1	0	0 0	Vd	1 0	!=00	0	1	M	0	Vm
					RM			size					

Half-precision scalar variant

Applies when `size == 01`.

```
VRINTA{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VRINTA{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VRINTA{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  exact = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

T1

15 14 13 12	11 10 9 8	7	6 5 4	3	2 1 0	15 12	11 10	9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	1	D 1 1 1	0	0 0	Vd	1 0	!=00	0	1	M	0	Vm
					RM			size					

Half-precision scalar variant

Applies when `size == 01`.

```
VRINTA{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VRINTA{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VRINTA{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  exact = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Sd>`	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
`<Sm>`	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
`<Dm>`	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
    when 16
        S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR, rounding, exact);
    when 32
        S[d] = FPRoundInt(S[m], FPSCR, rounding, exact);
    when 64
        D[d] = FPRoundInt(D[m], FPSCR, rounding, exact);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.201 VRINTM (Advanced SIMD)

Vector Round floating-point to integer towards -Infinity rounds a vector of floating-point values to integral floating-point values of the same size, using the Round towards -Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 18	17 16	15 12	11	10	9	8 7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	1	1	size	1	0	Vd	0	1	1 0 1	Q	M	0	Vm
												op				

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VRINTM{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VRINTM{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if op<2> != op<0> then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>));  exact = FALSE;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 2 1 0	15 12	11	10	9	8 7	6	5	4	3 0
1 1 1 1	1 1 1 1	1	D	1	1	size	1 0	Vd	0	1	1 0 1	Q	M	0	Vm
											op				

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VRINTM{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VRINTM{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if op<2> != op<0> then SEE "Related encodings";
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>));  exact = FALSE;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings

Related encodings: See *Advanced SIMD two registers misc* on page F3-4141 for the T32 instruction set, or *Advanced SIMD two registers misc* on page F4-4228 for the A32 instruction set.

Assembler symbols

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:

`F16`	when `size = 01`
`F32`	when `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
for r = 0 to regs-1
    for e = 0 to elements-1
        op1 = Elem[D[m+r],e,esize];
        result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);
        Elem[D[d+r],e,esize] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.202 VRINTM (floating-point)

Round floating-point to integer towards -Infinity rounds a floating-point value to an integral floating-point value of the same size using the Round towards -Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A1

31 30 29 28	27 26 25 24	23	22	21 20	19	18	17 16	15 12	11 10	9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	1	D	1 1 1	0	1	1	Vd	1 0	!=00	0	1	M	0	Vm
							RM			size					

Half-precision scalar variant

Applies when `size == 01`.

```
VRINTM{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VRINTM{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VRINTM{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  exact = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

T1

15 14 13 12	11 10 9 8	7	6	5 4	3	2	1 0	15 12	11 10	9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	1	D	1 1 1	0	1	1	Vd	1 0	!=00	0	1	M	0	Vm
							RM			size					

Half-precision scalar variant

Applies when `size == 01`.

```
VRINTM{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VRINTM{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VRINTM{<q>}.F64 <Dd>, <Dm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  exact = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Sd>`	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
`<Sm>`	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
`<Dm>`	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
    when 16
        S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR, rounding, exact);
    when 32
        S[d] = FPRoundInt(S[m], FPSCR, rounding, exact);
    when 64
        D[d] = FPRoundInt(D[m], FPSCR, rounding, exact);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.203 VRINTN (Advanced SIMD)

Vector Round floating-point to integer to Nearest rounds a vector of floating-point values to integral floating-point values of the same size using the Round to Nearest rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 18	17 16	15 12	11	10	9 7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	1	1	size	1 0	Vd	0	1	0 0 0	Q	M	0	Vm
											op				

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VRINTN{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VRINTN{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if op<2> != op<0> then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>));  exact = FALSE;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 2	1 0	15 12	11	10	9 7	6	5	4	3 0
1 1 1 1	1 1 1 1	1	D	1	1	size	1 0	Vd	0	1	0 0 0	Q	M	0	Vm
											op				

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VRINTN{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VRINTN{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if op<2> != op<0> then SEE "Related encodings";
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>));  exact = FALSE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings

Related encodings: See *Advanced SIMD two registers misc* on page F3-4141 for the T32 instruction set, or *Advanced SIMD two registers misc* on page F4-4228 for the A32 instruction set.

Assembler symbols

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:

`F16`	when `size = 01`
`F32`	when `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
for r = 0 to regs-1
    for e = 0 to elements-1
        op1 = Elem[D[m+r],e,esize];
        result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);
        Elem[D[d+r],e,esize] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.204 VRINTN (floating-point)

Round floating-point to integer to Nearest rounds a floating-point value to an integral floating-point value of the same size using the Round to Nearest rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A1

31 30 29 28	27 26 25 24	23	22 21 20	19	18	17 16	15 12	11 10	9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	1	D	1 1 1	0	0 1	Vd	1 0	!=00	0	1	M	0	Vm
						RM			size					

Half-precision scalar variant

Applies when `size == 01`.

```
VRINTN{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VRINTN{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VRINTN{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  exact = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

T1

15 14 13 12	11 10 9 8	7	6 5 4	3	2 1 0	15 12	11 10	9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	1	D	1 1 1	0	0 1	Vd	1 0	!=00	0	1	M	0	Vm
						RM			size					

Half-precision scalar variant

Applies when `size == 01`.

```
VRINTN{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VRINTN{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VRINTN{<q>}.F64 <Dd>, <Dm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  exact = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Sd>`	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
`<Sm>`	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
`<Dm>`	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
    when 16
        S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR, rounding, exact);
    when 32
        S[d] = FPRoundInt(S[m], FPSCR, rounding, exact);
    when 64
        D[d] = FPRoundInt(D[m], FPSCR, rounding, exact);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.205 VRINTP (Advanced SIMD)

Vector Round floating-point to integer towards +Infinity rounds a vector of floating-point values to integral floating-point values of the same size using the Round towards +Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A1

31 30 29 28	27 26 25 24	23	22	21 20	19 18	17 16	15 12	11	10	9 7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	1 1	size	1 0	Vd	0	1	1 1 1	Q	M	0	Vm
										op				

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VRINTP{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VRINTP{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if op<2> != op<0> then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>));  exact = FALSE;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5 4	3 2	1 0	15 12	11	10	9 7	6	5	4	3 0
1 1 1 1	1 1 1 1	1	D	1 1	size	1 0	Vd	0	1	1 1 1	Q	M	0	Vm
										op				

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VRINTP{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VRINTP{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if op<2> != op<0> then SEE "Related encodings";
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>));  exact = FALSE;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings

Related encodings: See *Advanced SIMD two registers misc* on page F3-4141 for the T32 instruction set, or *Advanced SIMD two registers misc* on page F4-4228 for the A32 instruction set.

Assembler symbols

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:

`F16`	when `size = 01`
`F32`	when `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
for r = 0 to regs-1
    for e = 0 to elements-1
        op1 = Elem[D[m+r],e,esize];
        result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);
        Elem[D[d+r],e,esize] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.206 VRINTP (floating-point)

Round floating-point to integer towards +Infinity rounds a floating-point value to an integral floating-point value of the same size using the Round towards +Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A1

31 30 29 28	27 26 25 24	23	22	21 20	19	18	17 16	15 12	11 10	9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	1	D	1 1 1	0	1 0		Vd	1 0	!=00	0	1	M	0	Vm
						RM				size					

Half-precision scalar variant

Applies when `size == 01`.

```
VRINTP{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VRINTP{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VRINTP{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  exact = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

T1

15 14 13 12	11 10 9 8	7	6	5 4	3	2	1 0	15 12	11 10	9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	1	D	1 1 1	0	1	0	Vd	1 0	!=00	0	1	M	0	Vm
							RM			size					

Half-precision scalar variant

Applies when `size == 01`.

```
VRINTP{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VRINTP{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VRINTP{<q>}.F64 <Dd>, <Dm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  exact = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Sd>`	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
`<Sm>`	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
`<Dm>`	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
    when 16
        S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR, rounding, exact);
    when 32
        S[d] = FPRoundInt(S[m], FPSCR, rounding, exact);
    when 64
        D[d] = FPRoundInt(D[m], FPSCR, rounding, exact);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.207 VRINTR

Round floating-point to integer rounds a floating-point value to an integral floating-point value of the same size using the rounding mode specified in the FPSCR. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A1

31–28	27–23	22	21–20	19–16	15–12	11–10	9–8	7	6	5	4	3–0
!=1111	1 1 1 0 1	D	1 1	0 1 1 0	Vd	1 0	size	0	1	M	0	Vm
cond								op				

Half-precision scalar variant

Applies when `size == 01`.

```
VRINTR{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VRINTR{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VRINTR{<c>}{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR);
exact = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

T1

15–12	11–8	7	6	5	4	3	2	1	0	15–12	11–10	9–8	7	6	5	4	3–0
1 1 1 0	1 1 1 0 1	D	1	1	0	1	1	1	0	Vd	1 0	size	0	1	M	0	Vm
													op				

Half-precision scalar variant

Applies when `size == 01`.

```
VRINTR{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VRINTR{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VRINTR{<c>}{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR);
exact = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Sd>`	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
`<Sm>`	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
`<Dm>`	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    case esize of
        when 16
            S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR, rounding, exact);
        when 32
            S[d] = FPRoundInt(S[m], FPSCR, rounding, exact);
        when 64
            D[d] = FPRoundInt(D[m], FPSCR, rounding, exact);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.208 VRINTX (Advanced SIMD)

Vector round floating-point to integer inexact rounds a vector of floating-point values to integral floating-point values of the same size, using the Round to Nearest rounding mode, and raises the Inexact exception when the result value is not numerically equal to the input value. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 18	17 16	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	1	1	size	1 0	Vd	0	1	0	0	1	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VRINTX{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VRINTX{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPRounding_TIEEVEN;  exact = TRUE;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 2	1 0	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	1	D	1	1	size	1 0	Vd	0	1	0	0	1	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VRINTX{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VRINTX{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPRounding_TIEEVEN;  exact = TRUE;
case size of
    when '01' esize = 16; elements = 4;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
if InITBlock() then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:

`F16`	when `size = 01`
`F32`	when `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
for r = 0 to regs-1
    for e = 0 to elements-1
        op1 = Elem[D[m+r],e,esize];
        result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);
        Elem[D[d+r],e,esize] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.209 VRINTX (floating-point)

Round floating-point to integer inexact rounds a floating-point value to an integral floating-point value of the same size, using the rounding mode specified in the FPSCR, and raises an Inexact exception when the result value is not numerically equal to the input value. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A1

31–28	27 26 25 24	23	22	21 20	19	18 17 16	15–12	11 10	9 8	7	6	5	4	3–0
!=1111	1 1 1 0	1	D	1 1	0	1 1 1	Vd	1 0	size	0	1	M	0	Vm
cond														

Half-precision scalar variant

Applies when `size == 01`.

```
VRINTX{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VRINTX{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VRINTX{<c>}{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
exact = TRUE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

T1

15 14 13 12	11 10 9 8	7	6	5 4	3	2 1 0	15–12	11 10	9 8	7	6	5	4	3–0
1 1 1 0	1 1 1 0 1	D	1	1	0	1 1 1	Vd	1 0	size	0	1	M	0	Vm

Half-precision scalar variant

Applies when `size == 01`.

```
VRINTX{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VRINTX{<c>}{<q>}.F32 <Sd>, <Sm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Double-precision scalar variant

Applies when `size == 11`.

```
VRINTX{<c>}{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
exact = TRUE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Sd>`	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
`<Sm>`	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
`<Dm>`	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    rounding = FPRoundingMode(FPSCR);
    case esize of
        when 16
            S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR, rounding, exact);
        when 32
            S[d] = FPRoundInt(S[m], FPSCR, rounding, exact);
        when 64
            D[d] = FPRoundInt(D[m], FPSCR, rounding, exact);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.210 VRINTZ (Advanced SIMD)

Vector round floating-point to integer towards Zero rounds a vector of floating-point values to integral floating-point values of the same size, using the Round towards Zero rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 18	17 16	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	1	1	size	1 0	Vd	0	1	0	1	1	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VRINTZ{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VRINTZ{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPRounding_ZERO;  exact = FALSE;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 2	1 0	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	1	D	1	1	size	1 0	Vd	0	1	0	1	1	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VRINTZ{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VRINTZ{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPRounding_ZERO;  exact = FALSE;
case size of
    when '01' esize = 16; elements = 4;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
if InITBlock() then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:

`F16`	when `size = 01`
`F32`	when `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
for r = 0 to regs-1
    for e = 0 to elements-1
        op1 = Elem[D[m+r],e,esize];
        result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);
        Elem[D[d+r],e,esize] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.211 VRINTZ (floating-point)

Round floating-point to integer towards Zero rounds a floating-point value to an integral floating-point value of the same size, using the Round towards Zero rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	1	1	1	0	1	D	1	1	0	1	1	0	Vd	1	0	size		1	1	M	0	Vm
cond																		op				

Half-precision scalar variant

Applies when `size == 01`.

```
VRINTZ{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VRINTZ{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VRINTZ{<c>}{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR);
exact = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	0	1	1	1	0	1	D	1	1	0	1	1	0	Vd	1	0	size		1	1	M	0	Vm
																					op				

Half-precision scalar variant

Applies when `size == 01`.

```
VRINTZ{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VRINTZ{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VRINTZ{<c>}{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR);
exact = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Sd>`	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
`<Sm>`	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
`<Dm>`	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    case esize of
        when 16
            S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR, rounding, exact);
        when 32
            S[d] = FPRoundInt(S[m], FPSCR, rounding, exact);
        when 64
            D[d] = FPRoundInt(D[m], FPSCR, rounding, exact);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.212 VRSHL

Vector Rounding Shift Left takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift. For a truncating shift, see VSHL.

The first operand and result elements are the same data type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The second operand is always a signed integer of the same size.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 U	0	D	size		Vn	Vd	0 1 0 1	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VRSHL{<c>}{<q>}.<dt> {<Dd>,} <Dm>, <Dn>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VRSHL{<c>}{<q>}.<dt> {<Qd>,} <Qm>, <Qn>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 U	1 1 1 1 0	D	size			Vn	Vd	0 1 0 1	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VRSHL{<c>}{<q>}.<dt> {<Dd>,} <Dm>, <Dn>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VRSHL{<c>}{<q>}.<dt> {<Qd>,} <Qm>, <Qn>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
```

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "U:size" field. It can have the following values:

`S8`	when `U = 0`, `size = 00`
`S16`	when `U = 0`, `size = 01`
`S32`	when `U = 0`, `size = 10`
`S64`	when `U = 0`, `size = 11`
`U8`	when `U = 1`, `size = 00`
`U16`	when `U = 1`, `size = 01`
`U32`	when `U = 1`, `size = 10`
`U64`	when `U = 1`, `size = 11`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            shift = SInt(Elem[D[n+r],e,esize]<7:0>);
            round_const = 1 << (-shift-1); // 0 for left shift, 2^(n-1) for right shift
            result = (Int(Elem[D[m+r],e,esize], unsigned) + round_const) << shift;
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.213 VRSHR

Vector Rounding Shift Right takes each element in a vector, right shifts them by an immediate value, and places the rounded results in the destination vector. For truncated results, see VSHR.

The operand and result elements must be the same size, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 U	1	D		imm6	Vd	0 0 1 0	L	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `!(imm6 == 000xxx && L == 0) && Q == 0`.

```
VRSHR{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

128-bit SIMD vector variant

Applies when `!(imm6 == 000xxx && L == 0) && Q == 1`.

```
VRSHR{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

Decode for all variants of this encoding

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 U	1 1 1 1	1	D		imm6	Vd	0 0 1 0	L	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `!(imm6 == 000xxx && L == 0) && Q == 0`.

```
VRSHR{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

128-bit SIMD vector variant

Applies when `!(imm6 == 000xxx && L == 0) && Q == 1`.

```
VRSHR{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Notes for all encodings

Related encodings: See *Advanced SIMD one register and modified immediate* on page F3-4146 for the T32 instruction set, or *Advanced SIMD one register and modified immediate* on page F4-4233 for the A32 instruction set.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<type>` Is the data type for the elements of the vectors, encoded in the "U" field. It can have the following values:

`S`	when `U = 0`
`U`	when `U = 1`

`<size>` Is the data size for the elements of the vectors, encoded in the "L:imm6<5:3>" field. It can have the following values:

`8`	when `L = 0`, `imm6<5:3> = 001`
`16`	when `L = 0`, `imm6<5:3> = 01x`
`32`	when `L = 0`, `imm6<5:3> = 1xx`
`64`	when `L = 1`, `imm6<5:3> = xxx`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

`<imm>` Is an immediate value, in the range 1 to `<size>`, encoded in the "imm6" field as `<size> - <imm>`.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (shift_amount - 1);
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = (Int(Elem[D[m+r],e,esize], unsigned) + round_const) >> shift_amount;
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.214 VRSHR (zero)

Vector Rounding Shift Right copies the contents of one SIMD register to another

This instruction is a pseudo-instruction of the VORR (register) instruction. This means that:

- The encodings in this description are named to match the encodings of VORR (register).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VORR (register) gives the operational pseudocode for this instruction.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 0	0	D	1	0	Vn	Vd	0 0 0 1	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VRSHR{<c>}{<q>}.<dt> <Dd>, <Dm>, #0
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>} <Dd>, <Dm>, <Dm>
```

and is never the preferred disassembly.

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VRSHR{<c>}{<q>}.<dt> <Qd>, <Qm>, #0
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>} <Qd>, <Qm>, <Qm>
```

and is never the preferred disassembly.

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1	0	1 1 1 1	0	D	1	0	Vn	Vd	0 0 0 1	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VRSHR{<c>}{<q>}.<dt> <Dd>, <Dm>, #0
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>} <Dd>, <Dm>, <Dm>
```

and is never the preferred disassembly.

128-bit SIMD vector variant

Applies when `Q == 1`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VRSHR{<c>}{<q>}.<dt> <Qd>, <Qm>, #0
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>} <Qd>, <Qm>, <Qm>
```

and is never the preferred disassembly.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, and must be one of: S8, S16, S32, S64, U8, U16, U32 or U64.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field.

Operation for all encodings

The description of VORR (register) gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.215 VRSHRN

Vector Rounding Shift Right and Narrow takes each element in a vector, right shifts them by an immediate value, and places the rounded results in the destination vector. For truncated results, see VSHRN.

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers. The destination elements are half the size of the source elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 0	1	D		imm6	Vd	1 0 0 0	0	1	M	1	Vm

A1 variant

Applies when `imm6 != 000xxx`.

```
VRSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>
```

Decode for this encoding

```
if imm6 == '000xxx' then SEE "Related encodings";
if Vm<0> == '1' then UNDEFINED;
case imm6 of
    when '001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '01xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);
```

T1

15 14 13 12	11 10 9 8	7	6	5	0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 0	1 1 1 1	1	D		imm6	Vd	1 0 0 0	0	1	M	1	Vm

T1 variant

Applies when `imm6 != 000xxx`.

```
VRSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>
```

Decode for this encoding

```
if imm6 == '000xxx' then SEE "Related encodings";
if Vm<0> == '1' then UNDEFINED;
case imm6 of
    when '001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '01xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);
```

Notes for all encodings

Related encodings: See *Advanced SIMD one register and modified immediate* on page F3-4146 for the T32 instruction set, or *Advanced SIMD one register and modified immediate* on page F4-4233 for the A32 instruction set.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<size>` Is the data size for the elements of the vectors, encoded in the "imm6<5:3>" field. It can have the following values:

`16` when `imm6<5:3> = 001`

`32` when `imm6<5:3> = 01x`

`64` when `imm6<5:3> = 1xx`

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<imm>` Is an immediate value, in the range 1 to `<size>`/2, encoded in the "imm6" field as `<size>`/2 - `<imm>`.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (shift_amount-1);
    for e = 0 to elements-1
        result = LSR(Elem[Qin[m>>1],e,2*esize] + round_const, shift_amount);
        Elem[D[d],e,esize] = result<esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.216 VRSHRN (zero)

Vector Rounding Shift Right and Narrow takes each element in a vector, right shifts them by an immediate value, and places the rounded results in the destination vector

This instruction is a pseudo-instruction of the VMOVN instruction. This means that:

- The encodings in this description are named to match the encodings of VMOVN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VMOVN gives the operational pseudocode for this instruction.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	0	Vd		0	0	1	0	0	0	M	0	Vm	

A1 variant

```
VRSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

and is never the preferred disassembly.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	0	Vd		0	0	1	0	0	0	M	0	Vm	

T1 variant

```
VRSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

and is never the preferred disassembly.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the operand, encoded in the "size" field. It can have the following values:

`I16` when `size = 00`

`I32` when `size = 01`

`I64` when `size = 10`

The encoding `size = 11` is reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation for all encodings

The description of VMOVN gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.217 VRSQRTE

Vector Reciprocal Square Root Estimate finds an approximate reciprocal square root of each element in a vector, and places the results in a second vector.

The operand and result elements are the same type, and can be floating-point numbers or unsigned integers.

For details of the operation performed by this instruction see *Floating-point reciprocal estimate and step* on page E1-3969.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 18	17 16	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	1	1	size	1	1	Vd	0	1	0	F	1	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VRSQRTE{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VRSQRTE{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
floating_point = (F == '1');
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 2	1	0	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	1	D	1	1	size	1	1	Vd	0	1	0	F	1	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VRSQRTE{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VRSQRTE{<c>}{<q>}.<dt> <Qd>, <Qm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
floating_point = (F == '1');
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "F:size" field. It can have the following values:

`U32`	when `F = 0`, `size = 10`
`F16`	when `F = 1`, `size = 01`
`F32`	when `F = 1`, `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Newton-Raphson iteration

For details of the operation performed and how it can be used in a Newton-Raphson iteration to calculate the reciprocal of the square root of a number, see *Floating-point reciprocal estimate and step* on page E1-3969.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                Elem[D[d+r],e,esize] = FPRSqrtEstimate(Elem[D[m+r],e,esize], StandardFPSCRValue());
            else
                Elem[D[d+r],e,esize] = UnsignedRSqrtEstimate(Elem[D[m+r],e,esize]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.218 VRSQRTS

Vector Reciprocal Square Root Step multiplies the elements of one vector by the corresponding elements of another vector, subtracts each of the products from 3.0, divides these results by 2.0, and places the results into the elements of the destination vector.

The operand and result elements are floating-point numbers.

For details of the operation performed by this instruction see *Floating-point reciprocal estimate and step* on page E1-3969.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	0	0	D	1	sz	Vn	Vd	1	1	1	1	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VRSQRTS{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VRSQRTS{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	0	1	1	1	1	0	D	1	sz	Vn	Vd	1	1	1	1	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VRSQRTS{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VRSQRTS{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

`F32`	when `sz = 0`
`F16`	when `sz = 1`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Newton-Raphson iteration

For details of the operation performed and how it can be used in a Newton-Raphson iteration to calculate the reciprocal of the square root of a number, see *Floating-point reciprocal estimate and step* on page E1-3969.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = FPRSqrtStep(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.219 VRSRA

Vector Rounding Shift Right and Accumulate takes each element in a vector, right shifts them by an immediate value, and accumulates the rounded results into the destination vector.For truncated results, see VSRA.

The operand and result elements must all be the same type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 U	1	D		imm6	Vd	0 0 1 1	L	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `!(imm6 == 000xxx && L == 0) && Q == 0`.

```
VRSRA{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

128-bit SIMD vector variant

Applies when `!(imm6 == 000xxx && L == 0) && Q == 1`.

```
VRSRA{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

Decode for all variants of this encoding

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 U	1 1 1 1	1	D		imm6	Vd	0 0 1 1	L	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `!(imm6 == 000xxx && L == 0) && Q == 0`.

```
VRSRA{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

128-bit SIMD vector variant

Applies when `!(imm6 == 000xxx && L == 0) && Q == 1`.

```
VRSRA{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Notes for all encodings

Related encodings: See *Advanced SIMD one register and modified immediate* on page F3-4146 for the T32 instruction set, or *Advanced SIMD one register and modified immediate* on page F4-4233 for the A32 instruction set.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<type>` Is the data type for the elements of the vectors, encoded in the "U" field. It can have the following values:

`S`	when `U = 0`
`U`	when `U = 1`

`<size>` Is the data size for the elements of the vectors, encoded in the "L:imm6<5:3>" field. It can have the following values:

`8`	when `L = 0`, `imm6<5:3> = 001`
`16`	when `L = 0`, `imm6<5:3> = 01x`
`32`	when `L = 0`, `imm6<5:3> = 1xx`
`64`	when `L = 1`, `imm6<5:3> = xxx`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

`<imm>` Is an immediate value, in the range 1 to `<size>`, encoded in the "imm6" field as `<size> - <imm>`.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (shift_amount - 1);
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = (Int(Elem[D[m+r],e,esize], unsigned) + round_const) >> shift_amount;
            Elem[D[d+r],e,esize] = Elem[D[d+r],e,esize] + result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.220 VRSUBHN

Vector Rounding Subtract and Narrow, returning High Half subtracts the elements of one quadword vector from the corresponding elements of another quadword vector, takes the most significant half of each result, and places the final results in a doubleword vector. The results are rounded. For truncated results, see VSUBHN.

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21 20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	!=11	Vn	Vd	0 1 1 0	N	0	M	0	Vm
				size								

A1 variant

```
VRSUBHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	1	D	!=11	Vn	Vd	0 1 1 0	N	0	M	0	Vm
				size									

T1 variant

```
VRSUBHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

Notes for all encodings

Related encodings: See *Advanced SIMD data-processing* on page F3-4137 for the T32 instruction set, or *Advanced SIMD data-processing* on page F4-4225 for the A32 instruction set.

Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<dt>` Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:

`I16` when `size = 00`

`I32` when `size = 01`

`I64` when `size = 10`

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (esize-1);
    for e = 0 to elements-1
        result = Elem[Qin[n>>1],e,2*esize] - Elem[Qin[m>>1],e,2*esize] + round_const;
        Elem[D[d],e,esize] = result<2*esize-1:esize>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.221 VSDOT (by element)

Dot Product index form with signed integers. This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_ISAR6.DP indicates whether this instruction is supported.

A1

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	1	0	0	D	1	0	Vn	Vd	1	1	0	1	N	Q	M	0 (U)	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>[<index>]
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VSDOT{<q>}.S8 <Qd>, <Qn>, <Dm>[<index>]
```

Decode for all variants of this encoding

```
if !HaveDOTPExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean signed = (U=='0');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm<3:0>);
integer index = UInt(M);
integer esize = 32;
integer regs = if Q == '1' then 2 else 1;
```

T1

ARMv8.2

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	1	0	0	D	1	0	Vn	Vd	1	1	0	1	N	Q	M	0 (U)	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>[<index>]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VSDOT{<q>}.S8 <Qd>, <Qn>, <Dm>[<index>]
```

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveDOTPExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean signed = (U=='0');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm<3:0>);
integer index = UInt(M);
integer esize = 32;
integer regs = if Q == '1' then 2 else 1;
```

Assembler symbols

`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Qd>`	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
`<Qn>`	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
`<Dn>`	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
`<Dm>`	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm" field.
`<index>`	Is the element index in the range 0 to 1, encoded in the "M" field.

Operation for all encodings

```
bits(64) operand1;
bits(64) operand2 = D[m];
bits(64) result;
CheckAdvSIMDEnabled();
for r = 0 to regs-1
    operand1 = D[n+r];
    result = D[d+r];
    integer element1, element2;
    for e = 0 to 1
        integer res = 0;
        for i = 0 to 3
            if signed then
                element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
                element2 = SInt(Elem[operand2, 4 * index + i, esize DIV 4]);
            else
                element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
                element2 = UInt(Elem[operand2, 4 * index + i, esize DIV 4]);
            res = res + element1 * element2;
        Elem[result, e, esize] = Elem[result, e, esize] + res;
    D[d+r] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.222 VSDOT (vector)

Dot Product vector form with signed integers. This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_ISAR6.DP indicates whether this instruction is supported.

A1

ARMv8.2

31 30 29 28	27 26 25 24	23 22 21 20	19 16	15 12	11 10 9 8	7 6 5 4	3 0
1 1 1 1	1 1 1 0	0 0 D 1 0	Vn	Vd	1 1 0 1	N Q M 0	Vm
						U	

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VSDOT{<q>}.S8 <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if !HaveDOTPExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
boolean signed = U=='0';
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer esize = 32;
integer regs = if Q == '1' then 2 else 1;
```

T1

ARMv8.2

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 10 9 8	7 6 5 4	3 0
1 1 1 1	1 1 1 0	0 0 D 1 0	Vn	Vd	1 1 0 1	N Q M 0	Vm
						U	

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VSDOT{<q>}.S8 <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveDOTPExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
boolean signed = U=='0';
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer esize = 32;
integer regs = if Q == '1' then 2 else 1;
```

Assembler symbols

`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Qd>`	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
`<Qn>`	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
`<Qm>`	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
`<Dn>`	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
`<Dm>`	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
bits(64) operand1;
bits(64) operand2;
bits(64) result;
CheckAdvSIMDEnabled();
for r = 0 to regs-1
    operand1 = D[n+r];
    operand2 = D[m+r];
    result = D[d+r];
    integer element1, element2;
    for e = 0 to 1
        integer res = 0;
        for i = 0 to 3
            if signed then
                element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
                element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            else
                element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
                element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            res = res + element1 * element2;
        Elem[result, e, esize] = Elem[result, e, esize] + res;
    D[d+r] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.223 VSELEQ, VSELGE, VSELGT, VSELVS

Floating-point conditional select allows the destination register to take the value in either one or the other source register according to the condition codes in the *The Application Program Status Register, APSR* on page E1-3955.

A1

31 30 29 28	27 26 25 24	23 22	21 20	19 16	15 12	11 10	9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	0 D	cc	Vn	Vd	1 0	!=00	N	0	M	0	Vm
							size					

VSELEQ,doubleprec variant

Applies when `cc == 00 && size == 11`.

```
VSELEQ.F64 <Dd>, <Dn>, <Dm> // Cannot be conditional
```

VSELEQ,halfprec variant

Applies when `cc == 00 && size == 01`.

```
VSELEQ.F16 <Sd>, <Sn>, <Sm> // Cannot be conditional
```

VSELEQ,singleprec variant

Applies when `cc == 00 && size == 10`.

```
VSELEQ.F32 <Sd>, <Sn>, <Sm> // Cannot be conditional
```

VSELGE,doubleprec variant

Applies when `cc == 10 && size == 11`.

```
VSELGE.F64 <Dd>, <Dn>, <Dm> // Cannot be conditional
```

VSELGE,halfprec variant

Applies when `cc == 10 && size == 01`.

```
VSELGE.F16 <Sd>, <Sn>, <Sm> // Cannot be conditional
```

VSELGE,singleprec variant

Applies when `cc == 10 && size == 10`.

```
VSELGE.F32 <Sd>, <Sn>, <Sm> // Cannot be conditional
```

VSELGT,doubleprec variant

Applies when `cc == 11 && size == 11`.

```
VSELGT.F64 <Dd>, <Dn>, <Dm> // Cannot be conditional
```

VSELGT,halfprec variant

Applies when `cc == 11 && size == 01`.

```
VSELGT.F16 <Sd>, <Sn>, <Sm> // Cannot be conditional
```

VSELGT,singleprec variant

Applies when `cc == 11 && size == 10`.

```
VSELGT.F32 <Sd>, <Sn>, <Sm> // Cannot be conditional
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

VSELVS,doubleprec variant

Applies when cc == 01 && size == 11.

```
VSELVS.F64 <Dd>, <Dn>, <Dm> // Cannot be conditional
```

VSELVS,halfprec variant

Applies when cc == 01 && size == 01.

```
VSELVS.F16 <Sd>, <Sn>, <Sm> // Cannot be conditional
```

VSELVS,singleprec variant

Applies when cc == 01 && size == 10.

```
VSELVS.F32 <Sd>, <Sn>, <Sm> // Cannot be conditional
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
cond = cc:(cc<1> EOR cc<0>):'0';
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	1	1	1	1	0	0	D	cc		Vn	Vd	1	0	!=00		N	0	M	0	Vm
																size						

VSELEQ,doubleprec variant

Applies when cc == 00 && size == 11.

```
VSELEQ.F64 <Dd>, <Dn>, <Dm> // Not permitted in IT block
```

VSELEQ,halfprec variant

Applies when cc == 00 && size == 01.

```
VSELEQ.F16 <Sd>, <Sn>, <Sm> // Not permitted in IT block
```

VSELEQ,singleprec variant

Applies when cc == 00 && size == 10.

```
VSELEQ.F32 <Sd>, <Sn>, <Sm> // Not permitted in IT block
```

VSELGE,doubleprec variant

Applies when cc == 10 && size == 11.

```
VSELGE.F64 <Dd>, <Dn>, <Dm> // Not permitted in IT block
```

VSELGE,halfprec variant

Applies when cc == 10 && size == 01.

```
VSELGE.F16 <Sd>, <Sn>, <Sm> // Not permitted in IT block
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

VSELGE,singleprec variant

Applies when `cc == 10 && size == 10`.

```
VSELGE.F32 <Sd>, <Sn>, <Sm> // Not permitted in IT block
```

VSELGT,doubleprec variant

Applies when `cc == 11 && size == 11`.

```
VSELGT.F64 <Dd>, <Dn>, <Dm> // Not permitted in IT block
```

VSELGT,halfprec variant

Applies when `cc == 11 && size == 01`.

```
VSELGT.F16 <Sd>, <Sn>, <Sm> // Not permitted in IT block
```

VSELGT,singleprec variant

Applies when `cc == 11 && size == 10`.

```
VSELGT.F32 <Sd>, <Sn>, <Sm> // Not permitted in IT block
```

VSELVS,doubleprec variant

Applies when `cc == 01 && size == 11`.

```
VSELVS.F64 <Dd>, <Dn>, <Dm> // Not permitted in IT block
```

VSELVS,halfprec variant

Applies when `cc == 01 && size == 01`.

```
VSELVS.F16 <Sd>, <Sn>, <Sm> // Not permitted in IT block
```

VSELVS,singleprec variant

Applies when `cc == 01 && size == 10`.

```
VSELVS.F32 <Sd>, <Sn>, <Sm> // Not permitted in IT block
```

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
cond = cc:(cc<1> EOR cc<0>):'0';
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
`<Dn>`	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

`<Sn>` Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

`<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

Operation for all encodings

```
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
    when 16
        S[d] = Zeros(16) : (if ConditionHolds(cond) then S[n] else S[m])<15:0>;
    when 32
        S[d] = if ConditionHolds(cond) then S[n] else S[m];
    when 64
        D[d] = if ConditionHolds(cond) then D[n] else D[m];
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.224 VSHL (immediate)

Vector Shift Left (immediate) takes each element in a vector of integers, left shifts them by an immediate value, and places the results in the destination vector.

Bits shifted out of the left of each element are lost.

The elements must all be the same size, and can be 8-bit, 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 0	1	D		imm6	Vd	0 1 0 1	L	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `!(imm6 == 000xxx && L == 0) && Q == 0`.

```
VSHL{<c>}{<q>}.I<size> {<Dd>,} <Dm>, #<imm>
```

128-bit SIMD vector variant

Applies when `!(imm6 == 000xxx && L == 0) && Q == 1`.

```
VSHL{<c>}{<q>}.I<size> {<Qd>,} <Qm>, #<imm>
```

Decode for all variants of this encoding

```
if L:imm6 == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 0	1 1 1 1	1	D		imm6	Vd	0 1 0 1	L	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `!(imm6 == 000xxx && L == 0) && Q == 0`.

```
VSHL{<c>}{<q>}.I<size> {<Dd>,} <Dm>, #<imm>
```

128-bit SIMD vector variant

Applies when `!(imm6 == 000xxx && L == 0) && Q == 1`.

```
VSHL{<c>}{<q>}.I<size> {<Qd>,} <Qm>, #<imm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if L:imm6 == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Notes for all encodings

Related encodings: See *Advanced SIMD one register and modified immediate* on page F3-4146 for the T32 instruction set, or *Advanced SIMD one register and modified immediate* on page F4-4233 for the A32 instruction set.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<size>` Is the data size for the elements of the vectors, encoded in the "L:imm6<5:3>" field. It can have the following values:

8	when `L = 0`, `imm6<5:3> = 001`
16	when `L = 0`, `imm6<5:3> = 01x`
32	when `L = 0`, `imm6<5:3> = 1xx`
64	when `L = 1`, `imm6<5:3> = xxx`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

`<imm>` Is an immediate value, in the range 0 to `<size>`-1, encoded in the "imm6" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = LSL(Elem[D[m+r],e,esize], shift_amount);
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.225 VSHL (register)

Vector Shift Left (register) takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift.

For a rounding shift, see VRSHL.

The first operand and result elements are the same data type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The second operand is always a signed integer of the same size.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 U	0	D	size		Vn	Vd	0 1 0 0	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VSHL{<c>}{<q>}.<dt> {<Dd>,} <Dm>, <Dn>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VSHL{<c>}{<q>}.<dt> {<Qd>,} <Qm>, <Qn>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 U	1 1 1 1 0	D	size			Vn	Vd	0 1 0 0	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VSHL{<c>}{<q>}.<dt> {<Dd>,} <Dm>, <Dn>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VSHL{<c>}{<q>}.<dt> {<Qd>,} <Qm>, <Qn>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
```

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "U:size" field. It can have the following values:

`S8`	when `U = 0`, `size = 00`
`S16`	when `U = 0`, `size = 01`
`S32`	when `U = 0`, `size = 10`
`S64`	when `U = 0`, `size = 11`
`U8`	when `U = 1`, `size = 00`
`U16`	when `U = 1`, `size = 01`
`U32`	when `U = 1`, `size = 10`
`U64`	when `U = 1`, `size = 11`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            shift = SInt(Elem[D[n+r],e,esize]<7:0>);
            result = Int(Elem[D[m+r],e,esize], unsigned) << shift;
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.226 VSHLL

Vector Shift Left Long takes each element in a doubleword vector, left shifts them by an immediate value, and places the results in a quadword vector.

The operand elements can be:

- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.
- 8-bit, 16-bit, or 32-bit untyped integers, maximum shift only.

The result elements are twice the length of the operand elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 U	1	D	imm6		Vd	1 0 1 0	0	0	M	1	Vm

A1 variant

Applies when `imm6 != 000xxx`.

```
VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>
```

Decode for this encoding

```
if imm6 == '000xxx' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
case imm6 of
    when '001xxx'  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when '01xxxx'  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
if shift_amount == 0 then SEE "VMOVL";
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);
```

A2

31 30 29 28	27 26 25 24	23	22	21 20	19 18	17 16	15 12	11	10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	1 1	size	1 0	Vd	0	0 1 1 0	0	0	M	0	Vm

A2 variant

```
VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>
```

Decode for this encoding

```
if size == '11' || Vd<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;  shift_amount = esize;
unsigned = FALSE;  // Or TRUE without change of functionality
d = UInt(D:Vd);  m = UInt(M:Vm);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15 14 13 12	11 10 9 8	7	6 5	0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 U	1 1 1 1 1	D	imm6		Vd	1 0 1 0	0	0	M	1	Vm

T1 variant

Applies when `imm6 != 000xxx`.

```
VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>
```

Decode for this encoding

```
if imm6 == '000xxx' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
case imm6 of
    when '001xxx'  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when '01xxxx'  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
if shift_amount == 0 then SEE "VMOVL";
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);
```

T2

15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0	15 12	11	10 9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 1 1	D	1 1	size 1 0	Vd	0	0 1 1 0	0	0	M	0	Vm

T2 variant

```
VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>
```

Decode for this encoding

```
if size == '11' || Vd<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;  shift_amount = esize;
unsigned = FALSE;  // Or TRUE without change of functionality
d = UInt(D:Vd);  m = UInt(M:Vm);
```

Notes for all encodings

Related encodings: See *Advanced SIMD one register and modified immediate* on page F3-4146 for the T32 instruction set, or *Advanced SIMD one register and modified immediate* on page F4-4233 for the A32 instruction set.

Assembler symbols

`<c>` For encoding A1 and A2: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1 and T2: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<type>` The data type for the elements of the operand. It must be one of:

- `S` Signed. In encoding T1/A1, encoded as U = 0.
- `U` Unsigned. In encoding T1/A1, encoded as U = 1.
- `I` Untyped integer, Available only in encoding T2/A2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<size>` The data size for the elements of the operand. The following table shows the permitted values and their encodings:

<size>	Encoding T1/A1	Encoding T2/A2
8	Encoded as imm6<5:3> = 0b001	Encoded as size = 0b00
16	Encoded as imm6<5:4> = 0b01	Encoded as size = 0b01
32	Encoded as imm6<5> = 1	Encoded as size = 0b10

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

`<imm>` The immediate value. `<imm>` must lie in the range 1 to `<size>`, and:

- If `<size>` == `<imm>`, the encoding is T2/A2.
- Otherwise, the encoding is T1/A1, and:
  - If `<size>` == 8, `<imm>` is encoded in imm6<2:0>.
  - If `<size>` == 16, `<imm>` is encoded in imm6<3:0>.
  - If `<size>` == 32, `<imm>` is encoded in imm6<4:0>.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        result = Int(Elem[Din[m],e,esize], unsigned) << shift_amount;
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.227 VSHR

Vector Shift Right takes each element in a vector, right shifts them by an immediate value, and places the truncated results in the destination vector. For rounded results, see VRSHR.

The operand and result elements must be the same size, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 U	1	D		imm6	Vd	0 0 0 0	L	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `!(imm6 == 000xxx && L == 0) && Q == 0`.

```
VSHR{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

128-bit SIMD vector variant

Applies when `!(imm6 == 000xxx && L == 0) && Q == 1`.

```
VSHR{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

Decode for all variants of this encoding

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 U	1 1 1 1	1	D		imm6	Vd	0 0 0 0	L	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `!(imm6 == 000xxx && L == 0) && Q == 0`.

```
VSHR{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

128-bit SIMD vector variant

Applies when `!(imm6 == 000xxx && L == 0) && Q == 1`.

```
VSHR{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Notes for all encodings

Related encodings: See *Advanced SIMD one register and modified immediate* on page F3-4146 for the T32 instruction set, or *Advanced SIMD one register and modified immediate* on page F4-4233 for the A32 instruction set.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<type>` Is the data type for the elements of the vectors, encoded in the "U" field. It can have the following values:

`S`	when `U = 0`
`U`	when `U = 1`

`<size>` Is the data size for the elements of the vectors, encoded in the "L:imm6<5:3>" field. It can have the following values:

`8`	when `L = 0`, `imm6<5:3> = 001`
`16`	when `L = 0`, `imm6<5:3> = 01x`
`32`	when `L = 0`, `imm6<5:3> = 1xx`
`64`	when `L = 1`, `imm6<5:3> = xxx`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

`<imm>` Is an immediate value, in the range 1 to `<size>`, encoded in the "imm6" field as `<size> - <imm>`.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = Int(Elem[D[m+r],e,esize], unsigned) >> shift_amount;
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.228 VSHR (zero)

Vector Shift Right copies the contents of one SIMD register to another

This instruction is a pseudo-instruction of the VORR (register) instruction. This means that:

- The encodings in this description are named to match the encodings of VORR (register).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VORR (register) gives the operational pseudocode for this instruction.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 0	0	D	1	0	Vn	Vd	0 0 0 1	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VSHR{<c>}{<q>}.<dt> <Dd>, <Dm>, #0
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>} <Dd>, <Dm>, <Dm>
```

and is never the preferred disassembly.

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VSHR{<c>}{<q>}.<dt> <Qd>, <Qm>, #0
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>} <Qd>, <Qm>, <Qm>
```

and is never the preferred disassembly.

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1	0	1 1 1 1	0	D	1	0	Vn	Vd	0 0 0 1	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VSHR{<c>}{<q>}.<dt> <Dd>, <Dm>, #0
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>} <Dd>, <Dm>, <Dm>
```

and is never the preferred disassembly.

128-bit SIMD vector variant

Applies when `Q == 1`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VSHR{<c>}{<q>}.<dt> <Qd>, <Qm>, #0
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>} <Qd>, <Qm>, <Qm>
```

and is never the preferred disassembly.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, and must be one of: S8, S16, S32, S64, U8, U16, U32 or U64.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field.

Operation for all encodings

The description of VORR (register) gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.229 VSHRN

Vector Shift Right Narrow takes each element in a vector, right shifts them by an immediate value, and places the truncated results in the destination vector. For rounded results, see VRSHRN.

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers. The destination elements are half the size of the source elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 0	1	D		imm6	Vd	1 0 0 0	0	0	M	1	Vm

A1 variant

Applies when `imm6 != 000xxx`.

```
VSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>
```

Decode for this encoding

```
if imm6 == '000xxx' then SEE "Related encodings";
if Vm<0> == '1' then UNDEFINED;
case imm6 of
    when '001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '01xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);
```

T1

15 14 13 12	11 10 9 8	7	6	5	0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 0	1 1 1 1	1	D		imm6	Vd	1 0 0 0	0	0	M	1	Vm

T1 variant

Applies when `imm6 != 000xxx`.

```
VSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>
```

Decode for this encoding

```
if imm6 == '000xxx' then SEE "Related encodings";
if Vm<0> == '1' then UNDEFINED;
case imm6 of
    when '001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '01xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);
```

Notes for all encodings

Related encodings: See *Advanced SIMD one register and modified immediate* on page F3-4146 for the T32 instruction set, or *Advanced SIMD one register and modified immediate* on page F4-4233 for the A32 instruction set.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<size>` Is the data size for the elements of the vectors, encoded in the "imm6<5:3>" field. It can have the following values:

`16` when `imm6<5:3> = 001`

`32` when `imm6<5:3> = 01x`

`64` when `imm6<5:3> = 1xx`

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<imm>` Is an immediate value, in the range 1 to `<size>`/2, encoded in the "imm6" field as `<size>`/2 - `<imm>`.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        result = LSR(Elem[Qin[m>>1],e,2*esize], shift_amount);
        Elem[D[d],e,esize] = result<esize-1:0>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.230 VSHRN (zero)

Vector Shift Right Narrow takes each element in a vector, right shifts them by an immediate value, and places the truncated results in the destination vector

This instruction is a pseudo-instruction of the VMOVN instruction. This means that:

- The encodings in this description are named to match the encodings of VMOVN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VMOVN gives the operational pseudocode for this instruction.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 18 17 16	15 12	11	10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	1	1	size	1 0	Vd	0	0 1 0	0	0	M	0	Vm

A1 variant

```
VSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

and is never the preferred disassembly.

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 2 1 0	15 12	11	10 9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	1	D	1	1	size	1 0	Vd	0	0 1 0	0	0	M	0	Vm

T1 variant

```
VSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

and is never the preferred disassembly.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the operand, encoded in the "size" field. It can have the following values:

`I16` when `size = 00`

`I32` when `size = 01`

`I64` when `size = 10`

The encoding `size = 11` is reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation for all encodings

The description of VMOVN gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.231 VSLI

Vector Shift Left and Insert takes each element in the operand vector, left shifts them by an immediate value, and inserts the results in the destination vector. Bits shifted out of the left of each element are lost.

The elements must all be the same size, and can be 8-bit, 16-bit, 32-bit, or 64-bit. There is no distinction between data types.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D		imm6	Vd	0 1 0 1	L	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `!(imm6 == 000xxx && L == 0) && Q == 0`.

```
VSLI{<c>}{<q>}.<size> {<Dd>,} <Dm>, #<imm>
```

128-bit SIMD vector variant

Applies when `!(imm6 == 000xxx && L == 0) && Q == 1`.

```
VSLI{<c>}{<q>}.<size> {<Qd>,} <Qm>, #<imm>
```

Decode for all variants of this encoding

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	1	D		imm6	Vd	0 1 0 1	L	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `!(imm6 == 000xxx && L == 0) && Q == 0`.

```
VSLI{<c>}{<q>}.<size> {<Dd>,} <Dm>, #<imm>
```

128-bit SIMD vector variant

Applies when `!(imm6 == 000xxx && L == 0) && Q == 1`.

```
VSLI{<c>}{<q>}.<size> {<Qd>,} <Qm>, #<imm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Notes for all encodings

Related encodings: See *Advanced SIMD one register and modified immediate* on page F3-4146 for the T32 instruction set, or *Advanced SIMD one register and modified immediate* on page F4-4233 for the A32 instruction set.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<size>` Is the data size for the elements of the vectors, encoded in the "L:imm6<5:3>" field. It can have the following values:

8	when L = 0, imm6<5:3> = 001
16	when L = 0, imm6<5:3> = 01x
32	when L = 0, imm6<5:3> = 1xx
64	when L = 1, imm6<5:3> = xxx

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

`<imm>` Is an immediate value, in the range 0 to `<size>`-1, encoded in the "imm6" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    mask = LSL(Ones(esize), shift_amount);
    for r = 0 to regs-1
        for e = 0 to elements-1
            shifted_op = LSL(Elem[D[m+r],e,esize], shift_amount);
            Elem[D[d+r],e,esize] = (Elem[D[d+r],e,esize] AND NOT(mask)) OR shifted_op;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.232 VSMMLA

The widening integer matrix multiply-accumulate instruction multiplies the 2x8 matrix of signed 8-bit integer values held in the first source vector by the 8x2 matrix of signed 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator held in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2, this is an OPTIONAL instruction. ID_ISAR6.I8MM indicates whether this instruction is supported in the T32 and A32 instruction sets.

A1

ARMv8.6

31 30 29 28	27 26 25 24	23	22	21 20	19 16	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	0	0	D	1 0	Vn	Vd	1	1	0	0	N	1	M	0	Vm
			B												U	

A1 variant

```
VSMMLA{<q>}.S8 <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
case B:U of
    when '00' op1_unsigned = FALSE; op2_unsigned = FALSE;
    when '01' op1_unsigned = TRUE;  op2_unsigned = TRUE;
    when '10' op1_unsigned = TRUE;  op2_unsigned = FALSE;
    when '11' UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
```

T1

ARMv8.6

15 14 13 12	11 10 9 8	7	6	5 4	3 0	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	0	0	D	1 0	Vn	Vd	1	1	0	0	N	1	M	0	Vm
			B												U	

T1 variant

```
VSMMLA{<q>}.S8 <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
case B:U of
    when '00' op1_unsigned = FALSE; op2_unsigned = FALSE;
    when '01' op1_unsigned = TRUE;  op2_unsigned = TRUE;
    when '10' op1_unsigned = TRUE;  op2_unsigned = FALSE;
    when '11' UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Qd>` Is the 128-bit name of the SIMD&FP third source and destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation for all encodings

```
CheckAdvSIMDEnabled();
bits(128) operand1 = Q[n>>1];
bits(128) operand2 = Q[m>>1];
bits(128) addend   = Q[d>>1];

Q[d>>1] = MatMulAdd(addend, operand1, operand2, op1_unsigned, op2_unsigned);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.233 VSQRT

Square Root calculates the square root of the value in a floating-point register and writes the result to another floating-point register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31–28	27–23	22	21–20	19–16	15–12	11–10	9–8	7	6	5	4	3–0
!=1111 (cond)	1 1 1 0 1	D	1 1	0	0 0 1	Vd	1 0	size	1	1	M	0	Vm

Half-precision scalar variant

Applies when `size == 01`.

```
VSQRT{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VSQRT{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VSQRT{<c>}{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15–14	13–12	11–10	9–8	7	6	5–4	3	2–1	0	15–12	11–10	9–8	7	6	5	4	3–0
1 1	1 0	1 1	1 0	1	D	1 1	0	0 0	1	Vd	1 0	size	1	1	M	0	Vm

Half-precision scalar variant

Applies when `size == 01`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VSQRT{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VSQRT{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VSQRT{<c>}{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Sd>`	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
`<Sm>`	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
`<Dm>`	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    case esize of
        when 16 S[d] = Zeros(16) : FPSqrt(S[m]<15:0>, FPSCR);
        when 32 S[d] = FPSqrt(S[m], FPSCR);
        when 64 D[d] = FPSqrt(D[m], FPSCR);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.234 VSRA

Vector Shift Right and Accumulate takes each element in a vector, right shifts them by an immediate value, and accumulates the truncated results into the destination vector. For rounded results, see VRSRA.

The operand and result elements must all be the same type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 0 1	U	1	D	imm6	Vd	0 0 0 1	L	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `!(imm6 == 000xxx && L == 0) && Q == 0`.

```
VSRA{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

128-bit SIMD vector variant

Applies when `!(imm6 == 000xxx && L == 0) && Q == 1`.

```
VSRA{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

Decode for all variants of this encoding

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1	U	1 1 1 1	1	D	imm6	Vd	0 0 0 1	L	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `!(imm6 == 000xxx && L == 0) && Q == 0`.

```
VSRA{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

128-bit SIMD vector variant

Applies when `!(imm6 == 000xxx && L == 0) && Q == 1`.

```
VSRA{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Notes for all encodings

Related encodings: See *Advanced SIMD one register and modified immediate* on page F3-4146 for the T32 instruction set, or *Advanced SIMD one register and modified immediate* on page F4-4233 for the A32 instruction set.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<type>` Is the data type for the elements of the vectors, encoded in the "U" field. It can have the following values:

`S`	when `U = 0`
`U`	when `U = 1`

`<size>` Is the data size for the elements of the vectors, encoded in the "L:imm6<5:3>" field. It can have the following values:

`8`	when `L = 0`, `imm6<5:3> = 001`
`16`	when `L = 0`, `imm6<5:3> = 01x`
`32`	when `L = 0`, `imm6<5:3> = 1xx`
`64`	when `L = 1`, `imm6<5:3> = xxx`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

`<imm>` Is an immediate value, in the range 1 to `<size>`, encoded in the "imm6" field as `<size> - <imm>`.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = Int(Elem[D[m+r],e,esize], unsigned) >> shift_amount;
            Elem[D[d+r],e,esize] = Elem[D[d+r],e,esize] + result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.235 VSRI

Vector Shift Right and Insert takes each element in the operand vector, right shifts them by an immediate value, and inserts the results in the destination vector. Bits shifted out of the right of each element are lost.

The elements must all be the same size, and can be 8-bit, 16-bit, 32-bit, or 64-bit. There is no distinction between data types.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	imm6		Vd	0 1 0 0	L	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `!(imm6 == 000xxx && L == 0) && Q == 0`.

```
VSRI{<c>}{<q>}.<size> {<Dd>,} <Dm>, #<imm>
```

128-bit SIMD vector variant

Applies when `!(imm6 == 000xxx && L == 0) && Q == 1`.

```
VSRI{<c>}{<q>}.<size> {<Qd>,} <Qm>, #<imm>
```

Decode for all variants of this encoding

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	1	D		imm6	Vd	0 1 0 0	L	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `!(imm6 == 000xxx && L == 0) && Q == 0`.

```
VSRI{<c>}{<q>}.<size> {<Dd>,} <Dm>, #<imm>
```

128-bit SIMD vector variant

Applies when `!(imm6 == 000xxx && L == 0) && Q == 1`.

```
VSRI{<c>}{<q>}.<size> {<Qd>,} <Qm>, #<imm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Notes for all encodings

Related encodings: See *Advanced SIMD one register and modified immediate* on page F3-4146 for the T32 instruction set, or *Advanced SIMD one register and modified immediate* on page F4-4233 for the A32 instruction set.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<size>` Is the data size for the elements of the vectors, encoded in the "L:imm6<5:3>" field. It can have the following values:

8	when L = 0, imm6<5:3> = 001
16	when L = 0, imm6<5:3> = 01x
32	when L = 0, imm6<5:3> = 1xx
64	when L = 1, imm6<5:3> = xxx

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

`<imm>` Is an immediate value, in the range 1 to `<size>`, encoded in the "imm6" field as `<size> - <imm>`.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    mask = LSR(Ones(esize), shift_amount);
    for r = 0 to regs-1
        for e = 0 to elements-1
            shifted_op = LSR(Elem[D[m+r],e,esize], shift_amount);
            Elem[D[d+r],e,esize] = (Elem[D[d+r],e,esize] AND NOT(mask)) OR shifted_op;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.236 VST1 (single element from one lane)

Store single element from one lane of one register stores one element to memory from one element of a register. For details of the addressing mode see *The Advanced SIMD addressing mode* on page F2-4101.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10	9 8	7 4	3 0
1 1 1 1	0 1 0 0 1	D	0	0	0	Rn	Vd	0 0	0 0	index_align	Rm
								size			

Offset variant

Applies when `Rm == 1111`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if index_align<0> != '0' then UNDEFINED;
ebytes = 1;  index = UInt(index_align<3:1>);  alignment = 1;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

A2

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10	9 8	7 4	3 0
1 1 1 1	0 1 0 0 1	D	0	0	0	Rn	Vd	0 1	0 0	index_align	Rm
								size			

Offset variant

Applies when `Rm == 1111`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if index_align<1> != '0' then UNDEFINED;
ebytes = 2;  index = UInt(index_align<3:2>);
alignment = if index_align<0> == '0' then 1 else 2;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

A3

31 30 29 28	27 26 25 24	23 22 21 20	19 16	15 12	11 10	9 8	7 4	3 0
1 1 1 1	0 1 0 0 1	D 0 0	Rn	Vd	1 0	0 0	index_align	Rm
					size			

Offset variant

Applies when `Rm == 1111`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if index_align<2> != '0' then UNDEFINED;
if index_align<1:0> != '00' && index_align<1:0> != '11' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
alignment = if index_align<1:0> == '00' then 1 else 4;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

T1

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 10	9 8	7 4	3 0
1 1 1 1	1 0 0 1 1	D 0 0	Rn	Vd	0 0	0 0	index_align	Rm
					size			

Offset variant

Applies when `Rm == 1111`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Post-indexed variant

Applies when `Rm == 1101`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if index_align<0> != '0' then UNDEFINED;
ebytes = 1;  index = UInt(index_align<3:1>);  alignment = 1;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3 – 0	15 – 12	11	10	9	8	7 – 4	3 – 0
1	1	1	1	1	0	0	1	1	D	0	0	Rn	Vd	0	1	0	0	index_align	Rm
														size					

Offset variant

Applies when `Rm == 1111`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if index_align<1> != '0' then UNDEFINED;
ebytes = 2;  index = UInt(index_align<3:2>);
alignment = if index_align<0> == '0' then 1 else 2;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

T3

15	14	13	12	11	10	9	8	7	6	5	4	3 – 0	15 – 12	11	10	9	8	7 – 4	3 – 0
1	1	1	1	1	0	0	1	1	D	0	0	Rn	Vd	1	0	0	0	index_align	Rm
														size					

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Offset variant

Applies when `Rm == 1111`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if index_align<2> != '0' then UNDEFINED;
if index_align<1:0> != '00' && index_align<1:0> != '11' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
alignment = if index_align<1:0> == '00' then 1 else 4;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` For encoding A1, A2 and A3: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1, T2 and T3: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

`8`	when `size = 00`
`16`	when `size = 01`
`32`	when `size = 10`

`<list>` Is a list containing the single 64-bit name of the SIMD&FP register holding the element.

The list must be `{ <Dd>[<index>] }`.

The register `<Dd>` is encoded in the "D:Vd" field.

The permitted values and encoding of `<index>` depend on `<size>`:

`<size> == 8``<index>` is in the range 0 to 7, encoded in the "index_align<3:1>" field.

`<size> == 16``<index>` is in the range 0 to 3, encoded in the "index_align<3:2>" field.

`<size> == 32``<index>` is 0 or 1, encoded in the "index_align<3>" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`<align>` When `<size>` == 8, `<align>` must be omitted, otherwise it is the optional alignment.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Whenever `<align>` is omitted, the standard alignment is used, see *Unaligned data access* on page E2-4004, and the encoding depends on `<size>`:

`<size> == 8`Encoded in the "index_align<0>" field as `0`.

`<size> == 16`Encoded in the "index_align<1:0>" field as `0b00`.

`<size> == 32`Encoded in the "index_align<2:0>" field as `0b000`.

Whenever `<align>` is present, the permitted values and encoding depend on `<size>`:

`<size> == 16``<align>` is 16, meaning 16-bit alignment, encoded in the "index_align<1:0>" field as `0b01`.

`<size> == 32``<align>` is 32, meaning 32-bit alignment, encoded in the "index_align<2:0>" field as `0b011`.

`:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see *The Advanced SIMD addressing mode* on page F2-4101.

`<Rm>` Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *The Advanced SIMD addressing mode* on page F2-4101.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = TRUE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    MemU[address,ebytes] = Elem[D[d],index];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + ebytes;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.237 VST1 (multiple single elements)

Store multiple single elements from one, two, three, or four registers stores elements to memory from one, two, three, or four registers, without interleaving. Every element of each register is stored. For details of the addressing mode see *The Advanced SIMD addressing mode* on page F2-4101.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8	7 6	5 4	3 0
1 1 1 1	0 1 0 0	0	D	0	0	Rn	Vd	0 1 1 1	size	align	Rm

Offset variant

Applies when `Rm == 1111`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
regs = 1;  if align<1> == '1' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A2

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8	7 6	5 4	3 0
1 1 1 1	0 1 0 0	0	D	0	0	Rn	Vd	1 0 1 0	size	align	Rm

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Offset variant

Applies when `Rm == 1111`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
regs = 2;  if align == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A3

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8	7 6 5 4	3 0
1 1 1 1	0 1 0 0	0	D	0	0	Rn	Vd	0 1 1 0	size	align	Rm

Offset variant

Applies when `Rm == 1111`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
regs = 3;  if align<1> == '1' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A4

31 30 29 28	27 26 25 24	23 22 21 20	19 16	15 12	11 10 9 8	7 6 5 4	3 0
1 1 1 1	0 1 0 0 0	D 0 0	Rn	Vd	0 0 1 0	size align	Rm

Offset variant

Applies when `Rm == 1111`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
regs = 4;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7 6	5 4	3 0
1 1 1 1	1 0 0 1 0	D	0	0	Rn	Vd	0 1 1 1	size	align	Rm	

Offset variant

Applies when `Rm == 1111`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
regs = 1;  if align<1> == '1' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T2

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7 6	5 4	3 0
1 1 1 1	1 0 0 1 0	D	0	0	Rn	Vd	1 0 1 0	size	align	Rm	

Offset variant

Applies when `Rm == 1111`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Post-indexed variant

Applies when `Rm == 1101`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
regs = 2;  if align == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T3

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	1	1	0	0	1	0	D	0	0	Rn	Vd	0	1	1	0	size	align	Rm		

Offset variant

Applies when `Rm == 1111`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
regs = 3;  if align<1> == '1' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T4

15	14	13	12	11	10	9	8	7	6	5	4	3 – 0	15 – 12	11	10	9	8	7 – 6	5 – 4	3 – 0
1	1	1	1	1	0	0	1	0	D	0	0	Rn	Vd	0	0	1	0	size	align	Rm

Offset variant

Applies when `Rm == 1111`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
regs = 4;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *VST1 (multiple single elements)* on page K1-7848.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Related encodings: See *Advanced SIMD element or structure load/store* on page F3-4160 for the T32 instruction set, or *Advanced SIMD element or structure load/store* on page F4-4237 for the A32 instruction set.

Assembler symbols

`<c>` For encoding A1, A2, A3 and A4: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1, T2, T3 and T4: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

- `8` when `size = 00`
- `16` when `size = 01`
- `32` when `size = 10`
- `64` when `size = 11`

`<list>` Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

- `{ <Dd> }` Single register. Selects the A1 and T1 encodings of the instruction.
- `{ <Dd>, <Dd+1> }` Two single-spaced registers. Selects the A2 and T2 encodings of the instruction.
- `{ <Dd>, <Dd+1>, <Dd+2> }` Three single-spaced registers. Selects the A3 and T3 encodings of the instruction.
- `{ <Dd>, <Dd+1>, <Dd+2>, <Dd+3> }` Four single-spaced registers. Selects the A4 and T4 encodings of the instruction.

The register `<Dd>` is encoded in the "D:Vd" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`<align>` Is the optional alignment.

Whenever `<align>` is omitted, the standard alignment is used, see *Unaligned data access* on page E2-4004, and is encoded in the "align" field as `0b00`.

Whenever `<align>` is present, the permitted values are:

- `64` 64-bit alignment, encoded in the "align" field as `0b01`.
- `128` 128-bit alignment, encoded in the "align" field as `0b10`. Available only if `<list>` contains two or four registers.
- `256` 256-bit alignment, encoded in the "align" field as `0b11`. Available only if `<list>` contains four registers.

`:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see *The Advanced SIMD addressing mode* on page F2-4101.

`<Rm>` Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about `<Rn>`, `!`, and `<Rm>`, see *The Advanced SIMD addressing mode* on page F2-4101.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = TRUE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    for r = 0 to regs-1
        for e = 0 to elements-1
            if ebytes != 8 then
                MemU[address,ebytes] = Elem[D[d+r],e];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        else
            - = AArch32.CheckAlignment(address, ebytes, AccType_NORMAL, iswrite);
            bits(64) data = Elem[D[d+r],e];
            MemU[address,4] = if BigEndian() then data<63:32> else data<31:0>;
            MemU[address+4,4] = if BigEndian() then data<31:0> else data<63:32>;
        address = address + ebytes;
if wback then
    if register_index then
        R[n] = R[n] + R[m];
    else
        R[n] = R[n] + 8*regs;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.238 VST2 (single 2-element structure from one lane)

Store single 2-element structure from one lane of two registers stores one 2-element structure to memory from corresponding elements of two registers. For details of the addressing mode see *The Advanced SIMD addressing mode* on page F2-4101.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10	9 8	7 4	3 0
1 1 1 1	0 1 0 0 1	D	0	0	0	Rn	Vd	0 0 (size)	0 1	index_align	Rm

Offset variant

Applies when `Rm == 1111`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
alignment = if index_align<0> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A2

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10	9 8	7 4	3 0
1 1 1 1	0 1 0 0 1	D	0	0	0	Rn	Vd	0 1 (size)	0 1	index_align	Rm

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Offset variant

Applies when `Rm == 1111`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
ebytes = 2;  index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 4;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A3

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10	9 8	7 4	3 0
1 1 1 1	0 1 0 0	1	D	0	0	Rn	Vd	1 0	0 1	index_align	Rm
								size			

Offset variant

Applies when `Rm == 1111`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if index_align<1> != '0' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7 … 4	3 … 0
1	1	1	1	1	0	0	1	1	D	0	0	Rn	Vd	0	0	0	1	index_align	Rm
														size					

Offset variant

Applies when `Rm == 1111`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
alignment = if index_align<0> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7 … 4	3 … 0
1	1	1	1	1	0	0	1	1	D	0	0	Rn	Vd	0	1	0	1	index_align	Rm
														size					

Offset variant

Applies when `Rm == 1111`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
ebytes = 2;  index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 4;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T3

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7 … 4	3 … 0
1	1	1	1	1	0	0	1	1	D	0	0	Rn	Vd	1	0	0	1	index_align	Rm
														size					

Offset variant

Applies when `Rm == 1111`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Post-indexed variant

Applies when `Rm == 1101`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if index_align<1> != '0' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *VST2 (single 2-element structure from one lane)* on page K1-7849.

Assembler symbols

`<c>` For encoding A1, A2 and A3: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1, T2 and T3: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

- `8` when `size = 00`
- `16` when `size = 01`
- `32` when `size = 10`

`<list>` Is a list containing the 64-bit names of the two SIMD&FP registers holding the element.

The list must be one of:

`{ <Dd>[<index>], <Dd+1>[<index>] }` Single-spaced registers, encoded as "spacing" = `0`.

`{ <Dd>[<index>], <Dd+2>[<index>] }` Double-spaced registers, encoded as "spacing" = `1`. Not permitted when `<size>` == 8.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The encoding of "spacing" depends on <size>:

<size> == 16"spacing" is encoded in the "index_align<1>" field.

<size> == 32"spacing" is encoded in the "index_align<2>" field.

The register <Dd> is encoded in the "D:Vd" field.

The permitted values and encoding of <index> depend on <size>:

<size> == 8<index> is in the range 0 to 7, encoded in the "index_align<3:1>" field.

<size> == 16<index> is in the range 0 to 3, encoded in the "index_align<3:2>" field.

<size> == 32<index> is 0 or 1, encoded in the "index_align<3>" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment.

Whenever <align> is omitted, the standard alignment is used, see *Unaligned data access* on page E2-4004, and the encoding depends on <size>:

<size> == 8Encoded in the "index_align<0>" field as 0.

<size> == 16Encoded in the "index_align<0>" field as 0.

<size> == 32Encoded in the "index_align<1:0>" field as 0b00.

Whenever <align> is present, the permitted values and encoding depend on <size>:

<size> == 8<align> is 16, meaning 16-bit alignment, encoded in the "index_align<0>" field as 1.

<size> == 16<align> is 32, meaning 32-bit alignment, encoded in the "index_align<0>" field as 1.

<size> == 32<align> is 64, meaning 64-bit alignment, encoded in the "index_align<1:0>" field as 0b01.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *The Advanced SIMD addressing mode* on page F2-4101.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *The Advanced SIMD addressing mode* on page F2-4101.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = TRUE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    MemU[address,       ebytes] = Elem[D[d], index];
    MemU[address+ebytes,ebytes] = Elem[D[d2],index];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 2*ebytes;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.239 VST2 (multiple 2-element structures)

Store multiple 2-element structures from two or four registers stores multiple 2-element structures from two or four registers to memory, with interleaving. For more information, see *Element and structure load/store instructions* on page F1-4058. Every element of each register is saved. For details of the addressing mode see *The Advanced SIMD addressing mode* on page F2-4101.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 8	7 6 5 4	3 0
1 1 1 1	0 1 0 0	0	D	0	0	Rn	Vd	1 0 0 x (itype)	size	align	Rm

Offset variant

Applies when `Rm == 1111`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
regs = 1;  if align == '11' then UNDEFINED;
if size == '11' then UNDEFINED;
inc = if itype == '1001' then 2 else 1;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2+regs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A2

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8	7 6	5 4	3 0
1 1 1 1	0 1 0 0	0	D	0	0	Rn	Vd	0 0 1 1	size	align	Rm

Offset variant

Applies when `Rm == 1111`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
regs = 2;  inc = 2;
if size == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2+regs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 8	7 6	5 4	3 0
1 1 1 1	1 0 0 1	0	D	0	0	Rn	Vd	1 0 0 x	size	align	Rm
								itype			

Offset variant

Applies when `Rm == 1111`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
regs = 1;  if align == '11' then UNDEFINED;
if size == '11' then UNDEFINED;
inc = if itype == '1001' then 2 else 1;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2+regs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	1	1	0	0	1	0	D	0	0	Rn	Vd	0	0	1	1	size		align		Rm

Offset variant

Applies when `Rm == 1111`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
regs = 2;  inc = 2;
if size == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2+regs > 32 then UNPREDICTABLE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CONSTRAINED UNPREDICTABLE behavior

If `d2+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *VST2 (multiple 2-element structures)* on page K1-7848.

Related encodings: See *Advanced SIMD element or structure load/store* on page F3-4160 for the T32 instruction set, or *Advanced SIMD element or structure load/store* on page F4-4237 for the A32 instruction set.

Assembler symbols

`<c>` For encoding A1 and A2: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1 and T2: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

`8` when `size = 00`

`16` when `size = 01`

`32` when `size = 10`

The encoding `size = 11` is reserved.

`<list>` Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

`{ <Dd>, <Dd+1> }` Two single-spaced registers. Selects the A1 and T1 encodings of the instruction, and encoded in the "itype" field as `0b1000`.

`{ <Dd>, <Dd+2> }` Two double-spaced registers. Selects the A1 and T1 encodings of the instruction, and encoded in the "itype" field as `0b1001`.

`{ <Dd>, <Dd+1>, <Dd+2>, <Dd+3> }` Three single-spaced registers. Selects the A2 and T2 encodings of the instruction.

The register `<Dd>` is encoded in the "D:Vd" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`<align>` Is the optional alignment.

Whenever `<align>` is omitted, the standard alignment is used, see *Unaligned data access* on page E2-4004, and is encoded in the "align" field as `0b00`.

Whenever `<align>` is present, the permitted values are:

`64` 64-bit alignment, encoded in the "align" field as `0b01`.

`128` 128-bit alignment, encoded in the "align" field as `0b10`.

`256` 256-bit alignment, encoded in the "align" field as `0b11`. Available only if `<list>` contains four registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *The Advanced SIMD addressing mode* on page F2-4101.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *The Advanced SIMD addressing mode* on page F2-4101.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = TRUE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    for r = 0 to regs-1
        for e = 0 to elements-1
            MemU[address,       ebytes] = Elem[D[d+r], e];
            MemU[address+ebytes,ebytes] = Elem[D[d2+r],e];
            address = address + 2*ebytes;
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 16*regs;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.240 VST3 (single 3-element structure from one lane)

Store single 3-element structure from one lane of three registers stores one 3-element structure to memory from corresponding elements of three registers. For details of the addressing mode see *The Advanced SIMD addressing mode* on page F2-4101.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10	9 8	7 4	3 0
1 1 1 1	0 1 0 0	1	D	0	0	Rn	Vd	0 0 (size)	1 0	index_align	Rm

Offset variant

Applies when `Rm == 1111`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if index_align<0> != '0' then UNDEFINED;
ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A2

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10	9 8	7 4	3 0
1 1 1 1	0 1 0 0	1	D	0	0	Rn	Vd	0 1 (size)	1 0	index_align	Rm

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Offset variant

Applies when `Rm == 1111`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if index_align<0> != '0' then UNDEFINED;
ebytes = 2;  index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A3

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10	9 8	7 4	3 0
1 1 1 1	0 1 0 0	1	D	0	0	Rn	Vd	1 0	1 0	index_align	Rm
								size			

Offset variant

Applies when `Rm == 1111`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if index_align<1:0> != '00' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7 … 4	3 … 0
1	1	1	1	1	0	0	1	1	D	0	0	Rn	Vd	0	0	1	0	index_align	Rm
														size					

Offset variant

Applies when `Rm == 1111`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if index_align<0> != '0' then UNDEFINED;
ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7 … 4	3 … 0
1	1	1	1	1	0	0	1	1	D	0	0	Rn	Vd	0	1	1	0	index_align	Rm
														size					

Offset variant

Applies when `Rm == 1111`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if index_align<0> != '0' then UNDEFINED;
ebytes = 2;  index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T3

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7 … 4	3 … 0
1	1	1	1	1	0	0	1	1	D	0	0	Rn	Vd	1	0	1	0	index_align	Rm
														size					

Offset variant

Applies when `Rm == 1111`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Post-indexed variant

Applies when `Rm == 1101`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if index_align<1:0> != '00' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *VST3 (single 3-element structure from one lane)* on page K1-7849.

Assembler symbols

`<c>` For encoding A1, A2 and A3: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1, T2 and T3: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

`8` when `size = 00`

`16` when `size = 01`

`32` when `size = 10`

`<list>` Is a list containing the 64-bit names of the three SIMD&FP registers holding the element.

The list must be one of:

`{ <Dd>[<index>], <Dd+1>[<index>], <Dd+2>[<index>] }` Single-spaced registers, encoded as "spacing" = `0`.

`{ <Dd>[<index>], <Dd+2>[<index>], <Dd+4>[<index>] }` Double-spaced registers, encoded as "spacing" = `1`. Not permitted when `<size>` == 8.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The encoding of "spacing" depends on <size>:

<size> == 8"spacing" is encoded in the "index_align<0>" field.

<size> == 16"spacing" is encoded in the "index_align<1>" field, and "index_align<0>" is set to 0.

<size> == 32"spacing" is encoded in the "index_align<2>" field, and "index_align<1:0>" is set to 0b00.

The register <Dd> is encoded in the "D:Vd" field.

The permitted values and encoding of <index> depend on <size>:

<size> == 8<index> is in the range 0 to 7, encoded in the "index_align<3:1>" field.

<size> == 16<index> is in the range 0 to 3, encoded in the "index_align<3:2>" field.

<size> == 32<index> is 0 or 1, encoded in the "index_align<3>" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *The Advanced SIMD addressing mode* on page F2-4101.

Alignment

Standard alignment rules apply, see *Alignment support* on page B2-148.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];
    MemU[address,         ebytes] = Elem[D[d], index];
    MemU[address+ebytes,  ebytes] = Elem[D[d2],index];
    MemU[address+2*ebytes,ebytes] = Elem[D[d3],index];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 3*ebytes;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.241 VST3 (multiple 3-element structures)

Store multiple 3-element structures from three registers stores multiple 3-element structures to memory from three registers, with interleaving. For more information, see *Element and structure load/store instructions* on page F1-4058. Every element of each register is saved. For details of the addressing mode see *The Advanced SIMD addressing mode* on page F2-4101.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 8	7 6 5 4	3 0
1 1 1 1	0 1 0 0	0	D	0	0	Rn	Vd	0 1 0 x (itype)	size	align	Rm

Offset variant

Applies when `Rm == 1111`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' || align<1> == '1' then UNDEFINED;
case itype of
    when '0100'
        inc = 1;
    when '0101'
        inc = 2;
    otherwise
        SEE "Related encodings";
alignment = if align<0> == '0' then 1 else 8;
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 8	7 6	5 4	3 0
1 1 1 1	1 0 0 1	0	D	0	0	Rn	Vd	0 1 0 x (itype)	size	align	Rm

Offset variant

Applies when `Rm == 1111`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' || align<1> == '1' then UNDEFINED;
case itype of
    when '0100'
        inc = 1;
    when '0101'
        inc = 2;
    otherwise
        SEE "Related encodings";
alignment = if align<0> == '0' then 1 else 8;
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *VST3 (multiple 3-element structures)* on page K1-7849.

Related encodings: See *Advanced SIMD element or structure load/store* on page F3-4160 for the T32 instruction set, or *Advanced SIMD element or structure load/store* on page F4-4237 for the A32 instruction set.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

`8` when `size = 00`

`16` when `size = 01`

`32` when `size = 10`

The encoding `size = 11` is reserved.

`<list>` Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

`{ <Dd>, <Dd+1>, <Dd+2> }`Single-spaced registers, encoded in the "itype" field as `0b0100`.

`{ <Dd>, <Dd+2>, <Dd+4> }`Double-spaced registers, encoded in the "itype" field as `0b0101`.

The register `<Dd>` is encoded in the "D:Vd" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`<align>` Is the optional alignment.

Whenever `<align>` is omitted, the standard alignment is used, see *Unaligned data access* on page E2-4004, and is encoded in the "align" field as `0b00`.

Whenever `<align>` is present, the only permitted values is 64, meaning 64-bit alignment, encoded in the "align" field as `0b01`.

`:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see *The Advanced SIMD addressing mode* on page F2-4101.

`<Rm>` Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *The Advanced SIMD addressing mode* on page F2-4101.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = TRUE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    for e = 0 to elements-1
        MemU[address,         ebytes] = Elem[D[d], e];
        MemU[address+ebytes,  ebytes] = Elem[D[d2],e];
        MemU[address+2*ebytes,ebytes] = Elem[D[d3],e];
        address = address + 3*ebytes;
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 24;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.242 VST4 (single 4-element structure from one lane)

Store single 4-element structure from one lane of four registers stores one 4-element structure to memory from corresponding elements of four registers. For details of the addressing mode see *The Advanced SIMD addressing mode* on page F2-4101.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10	9 8	7 4	3 0
1 1 1 1	0 1 0 0	1	D	0	0	Rn	Vd	0 0	1 1	index_align	Rm
								size			

Offset variant

Applies when `Rm == 1111`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if size != '00' then SEE "Related encodings";
ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
alignment = if index_align<0> == '0' then 1 else 4;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A2

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10	9 8	7 4	3 0
1 1 1 1	0 1 0 0	1	D	0	0	Rn	Vd	0 1	1 1	index_align	Rm
								size			

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Offset variant

Applies when `Rm == 1111`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if size != '01' then SEE "Related encodings";
ebytes = 2;  index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A3

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10	9 8	7 4	3 0
1 1 1 1	0 1 0 0	1	D	0	0	Rn	Vd	1 0	1 1	index_align	Rm
								size			

Offset variant

Applies when `Rm == 1111`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding

Applies when `Rm == 1101`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Reg-offset variant

Applies when `Rm != 11x1`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if size != '10' then SEE "Related encodings";
if index_align<1:0> == '11' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<1:0> == '00' then 1 else 4 << UInt(index_align<1:0>);
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7 … 4	3 … 0
1	1	1	1	1	0	0	1	1	D	0	0	Rn	Vd	0	0	1	1	index_align	Rm
														size					

Offset variant

Applies when `Rm == 1111`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if size != '00' then SEE "Related encodings";
ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
alignment = if index_align<0> == '0' then 1 else 4;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T2

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10	9 8	7 4	3 0
1 1 1 1	1 0 0 1	1	D	0	0	Rn	Vd	0 1	1 1	index_align	Rm
								size			

Offset variant

Applies when `Rm == 1111`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if size != '01' then SEE "Related encodings";
ebytes = 2;  index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T3

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10	9 8	7 4	3 0
1 1 1 1	1 0 0 1	1	D	0	0	Rn	Vd	1 0	1 1	index_align	Rm
								size			

Offset variant

Applies when `Rm == 1111`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if size != '10' then SEE "Related encodings";
if index_align<1:0> == '11' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<1:0> == '00' then 1 else 4 << UInt(index_align<1:0>);
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *VST4 (single 4-element structure from one lane)* on page K1-7849.

Assembler symbols

`<c>` For encoding A1, A2 and A3: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1, T2 and T3: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

`8`	when `size = 00`
`16`	when `size = 01`
`32`	when `size = 10`

`<list>` Is a list containing the 64-bit names of the four SIMD&FP registers holding the element.

The list must be one of:

`{ <Dd>[<index>], <Dd+1>[<index>], <Dd+2>[<index>], <Dd+3>[<index>] }` Single-spaced registers, encoded as "spacing" = `0`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`{ <Dd>[<index>], <Dd+2>[<index>], <Dd+4>[<index>], <Dd+6>[<index>] }`Double-spaced registers, encoded as "spacing" = 1. Not permitted when `<size>` == 8.

The encoding of "spacing" depends on `<size>`:

`<size> == 16`"spacing" is encoded in the "index_align<1>" field.

`<size> == 32`"spacing" is encoded in the "index_align<2>" field.

The register `<Dd>` is encoded in the "D:Vd" field.

The permitted values and encoding of `<index>` depend on `<size>`:

`<size> == 8``<index>` is in the range 0 to 7, encoded in the "index_align<3:1>" field.

`<size> == 16``<index>` is in the range 0 to 3, encoded in the "index_align<3:2>" field.

`<size> == 32``<index>` is 0 or 1, encoded in the "index_align<3>" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`<align>` Is the optional alignment.

Whenever `<align>` is omitted, the standard alignment is used, see *Unaligned data access* on page E2-4004, and the encoding depends on `<size>`:

`<size> == 8`Encoded in the "index_align<0>" field as `0`.

`<size> == 16`Encoded in the "index_align<0>" field as `0`.

`<size> == 32`Encoded in the "index_align<1:0>" field as `0b00`.

Whenever `<align>` is present, the permitted values and encoding depend on `<size>`:

`<size> == 8``<align>` is 32, meaning 32-bit alignment, encoded in the "index_align<0>" field as `1`.

`<size> == 16``<align>` is 64, meaning 64-bit alignment, encoded in the "index_align<0>" field as `1`.

`<size> == 32``<align>` can be 64 or 128. 64-bit alignment is encoded in the "index_align<1:0>" field as `0b01`, and 128-bit alignment is encoded in the "index_align<1:0>" field as `0b10`.

`:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see *The Advanced SIMD addressing mode* on page F2-4101.

`<Rm>` Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *The Advanced SIMD addressing mode* on page F2-4101.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = TRUE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    MemU[address,         ebytes] = Elem[D[d], index];
    MemU[address+ebytes,  ebytes] = Elem[D[d2],index];
    MemU[address+2*ebytes,ebytes] = Elem[D[d3],index];
    MemU[address+3*ebytes,ebytes] = Elem[D[d4],index];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 4*ebytes;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.243 VST4 (multiple 4-element structures)

Store multiple 4-element structures from four registers stores multiple 4-element structures to memory from four registers, with interleaving. For more information, see *Element and structure load/store instructions* on page F1-4058. Every element of each register is saved. For details of the addressing mode see *The Advanced SIMD addressing mode* on page F2-4101.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7–6	5–4	3–0
1	1	1	1	0	1	0	0	0	D	0	0	Rn	Vd	0 0 0 x (itype)	size	align	Rm

Offset variant

Applies when `Rm == 1111`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
case itype of
    when '0000'
        inc = 1;
    when '0001'
        inc = 2;
    otherwise
        SEE "Related encodings";
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 8	7 6	5 4	3 0
1 1 1 1	1 0 0 1	0	D	0	0	Rn	Vd	0 0 0 x (itype)	size	align	Rm

Offset variant

Applies when `Rm == 1111`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed variant

Applies when `Rm == 1101`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed variant

Applies when `Rm != 11x1`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
case itype of
    when '0000'
        inc = 1;
    when '0001'
        inc = 2;
    otherwise
        SEE "Related encodings";
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *VST4 (multiple 4-element structures)* on page K1-7849.

Related encodings: See *Advanced SIMD element or structure load/store* on page F3-4160 for the T32 instruction set, or *Advanced SIMD element or structure load/store* on page F4-4237 for the A32 instruction set.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

`8` when `size = 00`

`16` when `size = 01`

`32` when `size = 10`

The encoding `size = 11` is reserved.

`<list>` Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

`{ <Dd>, <Dd+1>, <Dd+2>, <Dd+3> }`Single-spaced registers, encoded in the "itype" field as `0b0000`.

`{ <Dd>, <Dd+2>, <Dd+4>, <Dd+6> }`Double-spaced registers, encoded in the "itype" field as `0b0001`.

The register `<Dd>` is encoded in the "D:Vd" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`<align>` Is the optional alignment.

Whenever `<align>` is omitted, the standard alignment is used, see *Unaligned data access* on page E2-4004, and is encoded in the "align" field as `0b00`.

Whenever `<align>` is present, the permitted values are:

`64` 64-bit alignment, encoded in the "align" field as `0b01`.

`128` 128-bit alignment, encoded in the "align" field as `0b10`.

`256` 256-bit alignment, encoded in the "align" field as `0b11`.

`:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see *The Advanced SIMD addressing mode* on page F2-4101.

`<Rm>` Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *The Advanced SIMD addressing mode* on page F2-4101.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = TRUE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    for e = 0 to elements-1
        MemU[address,         ebytes] = Elem[D[d], e];
        MemU[address+ebytes,  ebytes] = Elem[D[d2],e];
        MemU[address+2*ebytes,ebytes] = Elem[D[d3],e];
        MemU[address+3*ebytes,ebytes] = Elem[D[d4],e];
        address = address + 4*ebytes;
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 32;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.244 VSTM, VSTMDB, VSTMIA

Store multiple SIMD&FP registers stores multiple registers from the Advanced SIMD and floating-point register file to consecutive memory locations using an address from a general-purpose register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

This instruction is used by the alias VPUSH. See *Alias conditions* on page F6-5620 for details of when each alias is preferred.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7–1	0
!=1111	1	1	0	P	U	D	W	0	Rn	Vd	1	0	1	1	imm8<7:1>	0
cond																imm8<0>

Decrement Before variant

Applies when `P == 1 && U == 0 && W == 1`.

```
VSTMDB{<c>}{<q>}{.<size>} <Rn>!, <dreglist>
```

Increment After variant

Applies when `P == 0 && U == 1`.

```
VSTM{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
VSTMIA{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE "VSTR";
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = FALSE;  add = (U == '1');  wback = (W == '1');
d = UInt(D:Vd);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
regs = UInt(imm8) DIV 2;  // If UInt(imm8) is odd, see "FSTMX".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
if imm8<0> == '1' && (d+regs) > 16 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as a `VSTM` with the same addressing mode but stores no registers.

If `regs > 16 || (d+regs) > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A2

31–28	27 26 25	24	23	22	21	20	19–16	15–12	11 10	9 8	7–0
!=1111 (cond)	1 1 0	P	U	D	W	0	Rn	Vd	1 0	1 0	imm8

Decrement Before variant

Applies when `P == 1 && U == 0 && W == 1`.

```
VSTMDB{<c>}{<q>}{.<size>} <Rn>!, <sreglist>
```

Increment After variant

Applies when `P == 0 && U == 1`.

```
VSTM{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
VSTMIA{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE "VSTR";
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = TRUE;  add = (U == '1');  wback = (W == '1');  d = UInt(Vd:D);  n = UInt(Rn);
imm32 = ZeroExtend(imm8:'00', 32);  regs = UInt(imm8);
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || (d+regs) > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as a `VSTM` with the same addressing mode but stores no registers.

If `(d+regs) > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1

15 14 13 12	11 10 9 8	7	6	5	4	3–0	15–12	11 10	9 8	7–1	0
1 1 1 0	1 1 0	P	U	D	W	0	Rn	Vd	1 0	1 1	imm8<7:1>	imm8<0>

Decrement Before variant

Applies when `P == 1 && U == 0 && W == 1`.

```
VSTMDB{<c>}{<q>}{.<size>} <Rn>!, <dreglist>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Increment After variant

Applies when `P == 0 && U == 1`.

```
VSTM{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
VSTMIA{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE "VSTR";
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = FALSE;  add = (U == '1');  wback = (W == '1');
d = UInt(D:Vd);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
regs = UInt(imm8) DIV 2;  // If UInt(imm8) is odd, see "FSTMX".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
if imm8<0> == '1' && (d+regs) > 16 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as a `VSTM` with the same addressing mode but stores no registers.

If `regs > 16 || (d+regs) > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T2

15 14 13 12	11 10 9 8	7	6	5	4	3 … 0	15 … 12	11 10	9 8	7 … 0
1 1 1 0	1 1 0	P	U	D	W	0	Rn	Vd	1 0	1 0	imm8

Decrement Before variant

Applies when `P == 1 && U == 0 && W == 1`.

```
VSTMDB{<c>}{<q>}{.<size>} <Rn>!, <sreglist>
```

Increment After variant

Applies when `P == 0 && U == 1`.

```
VSTM{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
VSTMIA{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE "VSTR";
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
single_regs = TRUE;  add = (U == '1');  wback = (W == '1');  d = UInt(Vd:D);  n = UInt(Rn);
imm32 = ZeroExtend(imm8:'00', 32);  regs = UInt(imm8);
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || (d+regs) > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as a `VSTM` with the same addressing mode but stores no registers.

If `(d+regs) > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *VSTM* on page K1-7849.

Related encodings: See *Advanced SIMD and floating-point 64-bit move* on page F3-4147 for the T32 instruction set, or *Advanced SIMD and floating-point 64-bit move* on page F4-4220 for the A32 instruction set.

Alias conditions

Alias	is preferred when
VPUSH	`P == '1' && U == '0' && W == '1' && Rn == '1101'`

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<size>`	An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.
`<Rn>`	Is the general-purpose base register, encoded in the "Rn" field. If writeback is not specified, the PC can be used. However, Arm deprecates use of the PC.
`!`	Specifies base register writeback. Encoded in the "W" field as 1 if present, otherwise 0.
`<sreglist>`	Is the list of consecutively numbered 32-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "Vd:D", and "imm8" is set to the number of registers in the list. The list must contain at least one register.
`<dreglist>`	Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    address = if add then R[n] else R[n]-imm32;
    for r = 0 to regs-1
        if single_regs then
            MemA[address,4] = S[d+r];  address = address+4;
        else
            // Store as two word-aligned words in the correct order for current endianness.
            MemA[address,4] = if BigEndian() then D[d+r]<63:32> else D[d+r]<31:0>;
            MemA[address+4,4] = if BigEndian() then D[d+r]<31:0> else D[d+r]<63:32>;
            address = address+8;
    if wback then R[n] = if add then R[n]+imm32 else R[n]-imm32;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.245 VSTR

Store SIMD&FP register stores a single register from the Advanced SIMD and floating-point register file to memory, using an address from a general-purpose register, with an optional offset.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9–8	7–0
!=1111	1	1	0	1	U	D	0	0	Rn	Vd	1	0	size	imm8
cond														

Half-precision scalar variant

Applies when `size == 01`.

```
VSTR{<c>}{<q>}.16 <Sd>, [<Rn>{, #{+/-}<imm>}]
```

Single-precision scalar variant

Applies when `size == 10`.

```
VSTR{<c>}{<q>}{.32} <Sd>, [<Rn>{, #{+/-}<imm>}]
```

Double-precision scalar variant

Applies when `size == 11`.

```
VSTR{<c>}{<q>}{.64} <Dd>, [<Rn>{, #{+/-}<imm>}]
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
esize = 8 << UInt(size);  add = (U == '1');
imm32 = if esize == 16 then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);
case size of
    when '01' d = UInt(Vd:D);
    when '10' d = UInt(Vd:D);
    when '11' d = UInt(D:Vd);
n = UInt(Rn);
if n == 15 && CurrentInstrSet() != InstrSet_A32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9–8	7–0
1	1	1	0	1	1	0	1	U	D	0	0	Rn	Vd	1	0	size	imm8

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Half-precision scalar variant

Applies when `size == 01`.

```
VSTR{<c>}{<q>}.16 <Sd>, [<Rn>{, #{+/-}<imm>}]
```

Single-precision scalar variant

Applies when `size == 10`.

```
VSTR{<c>}{<q>}{.32} <Sd>, [<Rn>{, #{+/-}<imm>}]
```

Double-precision scalar variant

Applies when `size == 11`.

```
VSTR{<c>}{<q>}{.64} <Dd>, [<Rn>{, #{+/-}<imm>}]
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
esize = 8 << UInt(size);  add = (U == '1');
imm32 = if esize == 16 then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);
case size of
    when '01' d = UInt(Vd:D);
    when '10' d = UInt(Vd:D);
    when '11' d = UInt(D:Vd);
n = UInt(Rn);
if n == 15 && CurrentInstrSet() != InstrSet_A32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page F2-4080.
`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`.64`	Is an optional data size specifier for 64-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.
`<Dd>`	Is the 64-bit name of the SIMD&FP source register, encoded in the "D:Vd" field.
`.32`	Is an optional data size specifier for 32-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.
`<Sd>`	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vd:D" field.
`<Rn>`	Is the general-purpose base register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

`-` when `U = 0`

`+` when `U = 1`

`<imm>` For the single-precision scalar or double-precision scalar variants: is the optional unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0, and encoded in the "imm8" field as <imm>/4.

For the half-precision scalar variant: is the optional unsigned immediate byte offset, a multiple of 2, in the range 0 to 510, defaulting to 0, and encoded in the "imm8" field as <imm>/2.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    address = if add then (R[n] + imm32) else (R[n] - imm32);
    case esize of
        when 16
            MemA[address,2] = S[d]<15:0>;
        when 32
            MemA[address,4] = S[d];
        when 64
            // Store as two word-aligned words in the correct order for current endianness.
            MemA[address,4]   = if BigEndian() then D[d]<63:32> else D[d]<31:0>;
            MemA[address+4,4] = if BigEndian() then D[d]<31:0>  else D[d]<63:32>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.246 VSUB (floating-point)

Vector Subtract (floating-point) subtracts the elements of one vector from the corresponding elements of another vector, and places the results in the destination vector.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	0	0	D	1	sz	Vn	Vd	1	1	0	1	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
advsimd = TRUE;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

A2

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	1	1	1	0	0	D	1	1	Vn	Vd	1	0	size	N	1	M	0	Vm
cond																			

Half-precision scalar variant

Applies when `size == 01`.

```
VSUB{<c>}{<q>}.F16 {<Sd>,} <Sn>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VSUB{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VSUB{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
advsimd = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	0	1	1	1	1	0	D	1	sz	Vn	Vd	1	1	0	1	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
advsimd = TRUE;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T2

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	0	1	1	1	0	0	D	1	1	Vn	Vd	1	0	size		N	1	M	0	Vm

Half-precision scalar variant

Applies when `size == 01`.

```
VSUB{<c>}{<q>}.F16 {<Sd>,} <Sn>, <Sm>
```

Single-precision scalar variant

Applies when `size == 10`.

```
VSUB{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `size == 11`.

```
VSUB{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
advsimd = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding A2, T1 and T2: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

`F32`	when `sz = 0`
`F16`	when `sz = 1`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

`<Sn>` Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

`<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

Operation for all encodings

```
 if ConditionPassed() then
     EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
     if advsimd then  // Advanced SIMD instruction
         for r = 0 to regs-1
             for e = 0 to elements-1
                 Elem[D[d+r],e,esize] = FPSub(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize],
StandardFPSCRValue());
     else             // VFP instruction
         case esize of
             when 16
                 S[d] = Zeros(16) : FPSub(S[n]<15:0>, S[m]<15:0>, FPSCR);
             when 32
                 S[d] = FPSub(S[n], S[m], FPSCR);
             when 64
                 D[d] = FPSub(D[n], D[m], FPSCR);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.247 VSUB (integer)

Vector Subtract (integer) subtracts the elements of one vector from the corresponding elements of another vector, and places the results in the destination vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19 – 16	15 – 12	11	10	9	8	7	6	5	4	3 – 0
1	1	1	1	0	0	1	1	0	D	size		Vn	Vd	1	0	0	0	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3 – 0	15 – 12	11	10	9	8	7	6	5	4	3 – 0
1	1	1	1	1	1	1	1	0	D	size		Vn	Vd	1	0	0	0	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:

`I8`	when `size = 00`
`I16`	when `size = 01`
`I32`	when `size = 10`
`I64`	when `size = 11`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = Elem[D[n+r],e,esize] - Elem[D[m+r],e,esize];
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.248 VSUBHN

Vector Subtract and Narrow, returning High Half subtracts the elements of one quadword vector from the corresponding elements of another quadword vector, takes the most significant half of each result, and places the final results in a doubleword vector. The results are truncated. For rounded results, see VRSUBHN.

There is no distinction between signed and unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21 20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 0	1	D	!=11 (size)	Vn	Vd	0 1 1 0	N	0	M	0	Vm

A1 variant

```
VSUBHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

T1

15 14 13 12	11 10 9 8	7	6	5 4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 0	1 1 1 1 1	D	!=11 (size)	Vn	Vd	0 1 1 0	N	0	M	0	Vm

T1 variant

```
VSUBHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

Notes for all encodings

Related encodings: See *Advanced SIMD data-processing* on page F3-4137 for the T32 instruction set, or *Advanced SIMD data-processing* on page F4-4225 for the A32 instruction set.

Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<dt>` Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:

`I16` when `size = 00`

`I32` when `size = 01`

`I64` when `size = 10`

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        result = Elem[Qin[n>>1],e,2*esize] - Elem[Qin[m>>1],e,2*esize];
        Elem[D[d],e,esize] = result<2*esize-1:esize>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.249 VSUBL

Vector Subtract Long subtracts the elements of one doubleword vector from the corresponding elements of another doubleword vector, and places the results in a quadword vector. Before subtracting, it sign-extends or zero-extends the elements of both operands.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	0	0	1	U	1	D	!=11		Vn	Vd	0	0	1	0	N	0	M	0	Vm
										size							op					

A1 variant

```
VSUBL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;  is_vsubw = (op == '1');
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	U	1	1	1	1	1	D	!=11		Vn	Vd	0	0	1	0	N	0	M	0	Vm
										size							op					

T1 variant

```
VSUBL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;  is_vsubw = (op == '1');
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

Notes for all encodings

Related encodings: See *Advanced SIMD data-processing* on page F3-4137 for the T32 instruction set, or *Advanced SIMD data-processing* on page F4-4225 for the A32 instruction set.

Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<dt>` Is the data type for the elements of the second operand vector, encoded in the "U:size" field. It can have the following values:

`S8`	when `U = 0`, `size = 00`
`S16`	when `U = 0`, `size = 01`
`S32`	when `U = 0`, `size = 10`
`U8`	when `U = 1`, `size = 00`
`U16`	when `U = 1`, `size = 01`
`U32`	when `U = 1`, `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        if is_vsubw then
            op1 = Int(Elem[Qin[n>>1],e,2*esize], unsigned);
        else
            op1 = Int(Elem[Din[n],e,esize], unsigned);
        result = op1 - Int(Elem[Din[m],e,esize], unsigned);
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.250 VSUBW

Vector Subtract Wide subtracts the elements of a doubleword vector from the corresponding elements of a quadword vector, and places the results in another quadword vector. Before subtracting, it sign-extends or zero-extends the elements of the doubleword operand.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21 20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 U	1	D	!=11	Vn	Vd	0 0 1 1	N	0	M	0	Vm
				size			op					

A1 variant

```
VSUBW{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;  is_vsubw = (op == '1');
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

T1

15 14 13 12	11 10 9 8	7	6	5 4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 U	1 1 1 1	1	D	!=11	Vn	Vd	0 0 1 1	N	0	M	0	Vm
				size			op					

T1 variant

```
VSUBW{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;  is_vsubw = (op == '1');
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

Notes for all encodings

Related encodings: See *Advanced SIMD data-processing* on page F3-4137 for the T32 instruction set, or *Advanced SIMD data-processing* on page F4-4225 for the A32 instruction set.

Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

<q> See *Standard assembler syntax fields* on page F2-4080.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<dt>` Is the data type for the elements of the second operand vector, encoded in the "U:size" field. It can have the following values:

`S8`	when `U = 0`, `size = 00`
`S16`	when `U = 0`, `size = 01`
`S32`	when `U = 0`, `size = 10`
`U8`	when `U = 1`, `size = 00`
`U16`	when `U = 1`, `size = 01`
`U32`	when `U = 1`, `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        if is_vsubw then
            op1 = Int(Elem[Qin[n>>1],e,2*esize], unsigned);
        else
            op1 = Int(Elem[Din[n],e,esize], unsigned);
        result = op1 - Int(Elem[Din[m],e,esize], unsigned);
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.251 VSUDOT (by element)

Dot Product index form with signed and unsigned integers. This instruction performs the dot product of the four signed 8-bit integer values in each 32-bit element of the first source register with the four unsigned 8-bit integer values in an indexed 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2, this is an OPTIONAL instruction. ID_ISAR6.I8MM indicates whether this instruction is supported in the T32 and A32 instruction sets.

A1

ARMv8.6

31 30 29 28	27 26 25 24	23	22	21 20	19 16	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	1	D	0 0	Vn	Vd	1	1	0	1	N	Q	M	1	Vm
														U	

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VSUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>[<index>]
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VSUDOT{<q>}.U8 <Qd>, <Qn>, <Dm>[<index>]
```

Decode for all variants of this encoding

```
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean op1_unsigned = (U == '0');
boolean op2_unsigned = (U == '1');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm);
integer i = UInt(M);
integer regs = if Q == '1' then 2 else 1;
```

T1

ARMv8.6

15 14 13 12	11 10 9 8	7	6	5 4	3 0	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	1	D	0 0	Vn	Vd	1	1	0	1	N	Q	M	1	Vm
														U	

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VSUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>[<index>]
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VSUDOT{<q>}.U8 <Qd>, <Qn>, <Dm>[<index>]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean op1_unsigned = (U == '0');
boolean op2_unsigned = (U == '1');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm);
integer i = UInt(M);
integer regs = if Q == '1' then 2 else 1;
```

Assembler symbols

`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Qd>`	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
`<Qn>`	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
`<Dn>`	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
`<Dm>`	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm" field.
`<index>`	Is the element index in the range 0 to 1, encoded in the "M" field.

Operation for all encodings

```
CheckAdvSIMDEnabled();
bits(64) operand1;
bits(64) operand2;
bits(64) result;

operand2 = Din[m];
for r = 0 to regs-1
    operand1 = Din[n+r];
    result = Din[d+r];
    for e = 0 to 1
        bits(32) res = Elem[result, e, 32];
        for b = 0 to 3
            element1 = Int(Elem[operand1, 4 * e + b, 8], op1_unsigned);
            element2 = Int(Elem[operand2, 4 * i + b, 8], op2_unsigned);
            res = res + element1 * element2;
        Elem[result, e, 32] = res;
    D[d+r] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.252 VSWP

Vector Swap exchanges the contents of two vectors. The vectors can be either doubleword or quadword. There is no distinction between data types.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15 – 12	11	10	9	8	7	6	5	4	3 – 0
1	1	1	1	0	0	1	1	1	D	1	1	0	0	1	0	Vd	0	0	0	0	0	Q	M	0	Vm
												size													

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VSWP{<c>}{<q>}{.<dt>} <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VSWP{<c>}{<q>}{.<dt>} <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15 – 12	11	10	9	8	7	6	5	4	3 – 0
1	1	1	1	1	1	1	1	1	D	1	1	0	0	1	0	Vd	0	0	0	0	0	Q	M	0	Vm
												size													

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VSWP{<c>}{<q>}{.<dt>} <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VSWP{<c>}{<q>}{.<dt>} <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` An optional data type. It is ignored by assemblers, and does not affect the encoding.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        if d == m then
            D[d+r] = bits(64) UNKNOWN;
        else
            D[d+r] = Din[m+r];
            D[m+r] = Din[d+r];
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.253 VTBL, VTBX

Vector Table Lookup uses byte indexes in a control vector to look up byte values in a table and generate a new vector. Indexes out of range return 0.

Vector Table Extension works in the same way, except that indexes out of range leave the destination element unchanged.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23 22 21 20	19 16	15 12	11 10 9 8	7 6 5 4	3 0
1 1 1 1	0 0 1 1 1	D 1 1	Vn	Vd	1 0 len	N op M 0	Vm

VTBL variant

Applies when `op == 0`.

```
VTBL{<c>}{<q>}.8 <Dd>, <list>, <Dm>
```

VTBX variant

Applies when `op == 1`.

```
VTBX{<c>}{<q>}.8 <Dd>, <list>, <Dm>
```

Decode for all variants of this encoding

```
is_vtbl = (op == '0');  length = UInt(len)+1;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
if n+length > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `n + length > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. This behavior does not affect any general-purpose registers.

T1

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 10 9 8	7 6 5 4	3 0
1 1 1 1	1 1 1 1 1	D 1 1	Vn	Vd	1 0 len	N op M 0	Vm

VTBL variant

Applies when `op == 0`.

```
VTBL{<c>}{<q>}.8 <Dd>, <list>, <Dm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

VTBX variant

Applies when `op == 1`.

```
VTBX{<c>}{<q>}.8 <Dd>, <list>, <Dm>
```

Decode for all variants of this encoding

```
is_vtbl = (op == '0');  length = UInt(len)+1;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
if n+length > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `n + length > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. This behavior does not affect any general-purpose registers.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<list>` The vectors containing the table. It must be one of:

`{<Dn>}` Encoded as len = `0b00`.

`{<Dn>, <Dn+1>}` Encoded as len = `0b01`.

`{<Dn>, <Dn+1>, <Dn+2>}` Encoded as len = `0b10`.

`{<Dn>, <Dn+1>, <Dn+2>, <Dn+3>}` Encoded as len = `0b11`.

`<Dm>` Is the 64-bit name of the SIMD&FP source register holding the indices, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();

    // Create 256-bit = 32-byte table variable, with zeros in entries that will not be used.
    table3 = if length == 4 then D[n+3] else Zeros(64);
    table2 = if length >= 3 then D[n+2] else Zeros(64);
    table1 = if length >= 2 then D[n+1] else Zeros(64);
    table = table3 : table2 : table1 : D[n];

    for i = 0 to 7
        index = UInt(Elem[D[m],i,8]);
        if index < 8*length then
            Elem[D[d],i,8] = Elem[table,index,8];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
else
    if is_vtbl then
        Elem[D[d],i,8] = Zeros(8);
    // else Elem[D[d],i,8] unchanged
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

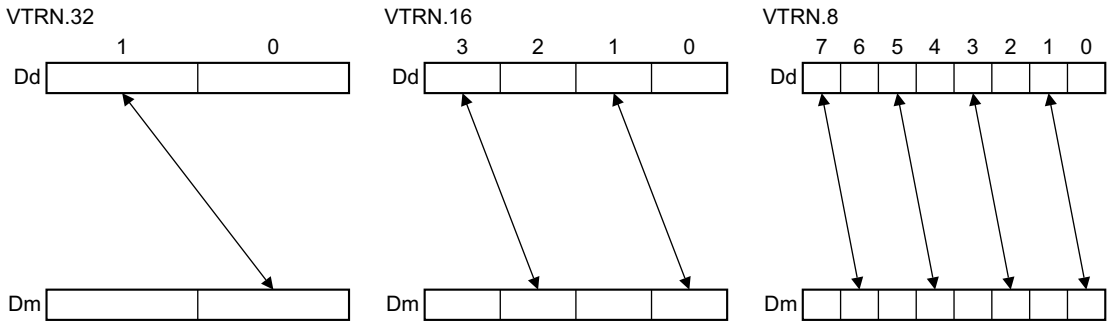
Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.254 VTRN

Vector Transpose treats the elements of its operand vectors as elements of 2 x 2 matrices, and transposes the matrices.

The elements of the vectors can be 8-bit, 16-bit, or 32-bit. There is no distinction between data types.

The following figure shows the operation of VTRN doubleword operations.



Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

This instruction is used by the pseudo-instructions VUZP (alias) and VZIP (alias). The pseudo-instruction is never the preferred disassembly.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	0	Vd		0	0	0	0	1	Q	M	0	Vm	

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VTRN{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VTRN{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	0	Vd		0	0	0	0	1	Q	M	0	Vm	

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VTRN{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VTRN{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:

`8`	when `size = 00`
`16`	when `size = 01`
`32`	when `size = 10`

The encoding `size = 11` is reserved.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    h = elements DIV 2;

    for r = 0 to regs-1
        if d == m then
            D[d+r] = bits(64) UNKNOWN;
        else
            for e = 0 to h-1
                Elem[D[d+r],2*e+1,esize] = Elem[Din[m+r],2*e,esize];
                Elem[D[m+r],2*e,esize] = Elem[Din[d+r],2*e+1,esize];
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.255 VTST

Vector Test Bits takes each element in a vector, and bitwise ANDs it with the corresponding element of a second vector. If the result is not zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements can be any one of:

- 8-bit, 16-bit, or 32-bit fields.

The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1	0 0 1 0	0	D	size		Vn	Vd	1 0 0 0	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VTST{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VTST{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 0	1 1 1 1 0	D	size			Vn	Vd	1 0 0 0	N	Q	M	1	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VTST{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VTST{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Qm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:

`8`	when `size = 00`
`16`	when `size = 01`
`32`	when `size = 10`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if !IsZero(Elem[D[n+r],e,esize] AND Elem[D[m+r],e,esize]) then
                Elem[D[d+r],e,esize] = Ones(esize);
            else
                Elem[D[d+r],e,esize] = Zeros(esize);
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.256 VUDOT (by element)

Dot Product index form with unsigned integers. This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_ISAR6.DP indicates whether this instruction is supported.

A1

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	1	0	0	D	1	0	Vn	Vd	1	1	0	1	N	Q	M	1 (U)	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>[<index>]
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VUDOT{<q>}.U8 <Qd>, <Qn>, <Dm>[<index>]
```

Decode for all variants of this encoding

```
if !HaveDOTPExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean signed = (U=='0');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm<3:0>);
integer index = UInt(M);
integer esize = 32;
integer regs = if Q == '1' then 2 else 1;
```

T1

ARMv8.2

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	1	0	0	D	1	0	Vn	Vd	1	1	0	1	N	Q	M	1 (U)	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>[<index>]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VUDOT{<q>}.U8 <Qd>, <Qn>, <Dm>[<index>]
```

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveDOTPExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean signed = (U=='0');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm<3:0>);
integer index = UInt(M);
integer esize = 32;
integer regs = if Q == '1' then 2 else 1;
```

Assembler symbols

`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Qd>`	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
`<Qn>`	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
`<Dn>`	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
`<Dm>`	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm" field.
`<index>`	Is the element index in the range 0 to 1, encoded in the "M" field.

Operation for all encodings

```
bits(64) operand1;
bits(64) operand2 = D[m];
bits(64) result;
CheckAdvSIMDEnabled();
for r = 0 to regs-1
    operand1 = D[n+r];
    result = D[d+r];
    integer element1, element2;
    for e = 0 to 1
        integer res = 0;
        for i = 0 to 3
            if signed then
                element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
                element2 = SInt(Elem[operand2, 4 * index + i, esize DIV 4]);
            else
                element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
                element2 = UInt(Elem[operand2, 4 * index + i, esize DIV 4]);
            res = res + element1 * element2;
        Elem[result, e, esize] = Elem[result, e, esize] + res;
    D[d+r] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.257 VUDOT (vector)

Dot Product vector form with unsigned integers. This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_ISAR6.DP indicates whether this instruction is supported.

A1

ARMv8.2

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	0	0	0	D	1	0	Vn	Vd	1	1	0	1	N	Q	M	1	Vm
																					U	

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VUDOT{<q>}.U8 <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if !HaveDOTPExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
boolean signed = U=='0';
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer esize = 32;
integer regs = if Q == '1' then 2 else 1;
```

T1

ARMv8.2

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	0	0	0	D	1	0	Vn	Vd	1	1	0	1	N	Q	M	1	Vm
																					U	

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VUDOT{<q>}.U8 <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveDOTPExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
boolean signed = U=='0';
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer esize = 32;
integer regs = if Q == '1' then 2 else 1;
```

Assembler symbols

`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Qd>`	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
`<Qn>`	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
`<Qm>`	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
`<Dn>`	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
`<Dm>`	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
bits(64) operand1;
bits(64) operand2;
bits(64) result;
CheckAdvSIMDEnabled();
for r = 0 to regs-1
    operand1 = D[n+r];
    operand2 = D[m+r];
    result = D[d+r];
    integer element1, element2;
    for e = 0 to 1
        integer res = 0;
        for i = 0 to 3
            if signed then
                element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
                element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            else
                element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
                element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            res = res + element1 * element2;
        Elem[result, e, esize] = Elem[result, e, esize] + res;
    D[d+r] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.258 VUMMLA

The widening integer matrix multiply-accumulate instruction multiplies the 2x8 matrix of unsigned 8-bit integer values held in the first source vector by the 8x2 matrix of unsigned 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator held in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2, this is an OPTIONAL instruction. ID_ISAR6.I8MM indicates whether this instruction is supported in the T32 and A32 instruction sets.

A1

ARMv8.6

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	1	0	0	D	1	0	Vn	Vd	1	1	0	0	N	1	M	1	Vm
								B													U	

A1 variant

```
VUMMLA{<q>}.U8 <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
case B:U of
    when '00' op1_unsigned = FALSE; op2_unsigned = FALSE;
    when '01' op1_unsigned = TRUE;  op2_unsigned = TRUE;
    when '10' op1_unsigned = TRUE;  op2_unsigned = FALSE;
    when '11' UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
```

T1

ARMv8.6

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	0	0	0	D	1	0	Vn	Vd	1	1	0	0	N	1	M	1	Vm
								B													U	

T1 variant

```
VUMMLA{<q>}.U8 <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
case B:U of
    when '00' op1_unsigned = FALSE; op2_unsigned = FALSE;
    when '01' op1_unsigned = TRUE;  op2_unsigned = TRUE;
    when '10' op1_unsigned = TRUE;  op2_unsigned = FALSE;
    when '11' UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Qd>` Is the 128-bit name of the SIMD&FP third source and destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation for all encodings

```
CheckAdvSIMDEnabled();
bits(128) operand1 = Q[n>>1];
bits(128) operand2 = Q[m>>1];
bits(128) addend   = Q[d>>1];

Q[d>>1] = MatMulAdd(addend, operand1, operand2, op1_unsigned, op2_unsigned);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.259 VUSDOT (by element)

Dot Product index form with unsigned and signed integers. This instruction performs the dot product of the four unsigned 8-bit integer values in each 32-bit element of the first source register with the four signed 8-bit integer values in an indexed 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2, this is an OPTIONAL instruction. ID_ISAR6.I8MM indicates whether this instruction is supported in the T32 and A32 instruction sets.

A1

ARMv8.6

31 30 29 28	27 26 25 24	23	22	21 20	19 16	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	1	D	0 0	Vn	Vd	1	1	0	1	N	Q	M	0	Vm
														U	

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VUSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>[<index>]
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VUSDOT{<q>}.S8 <Qd>, <Qn>, <Dm>[<index>]
```

Decode for all variants of this encoding

```
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean op1_unsigned = (U == '0');
boolean op2_unsigned = (U == '1');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm);
integer i = UInt(M);
integer regs = if Q == '1' then 2 else 1;
```

T1

ARMv8.6

15 14 13 12	11 10 9 8	7	6	5 4	3 0	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	1	D	0 0	Vn	Vd	1	1	0	1	N	Q	M	0	Vm
														U	

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VUSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>[<index>]
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VUSDOT{<q>}.S8 <Qd>, <Qn>, <Dm>[<index>]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean op1_unsigned = (U == '0');
boolean op2_unsigned = (U == '1');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm);
integer i = UInt(M);
integer regs = if Q == '1' then 2 else 1;
```

Assembler symbols

`<q>`	See *Standard assembler syntax fields* on page F2-4080.
`<Qd>`	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
`<Qn>`	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
`<Dd>`	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
`<Dn>`	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
`<Dm>`	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm" field.
`<index>`	Is the element index in the range 0 to 1, encoded in the "M" field.

Operation for all encodings

```
CheckAdvSIMDEnabled();
bits(64) operand1;
bits(64) operand2;
bits(64) result;

operand2 = Din[m];
for r = 0 to regs-1
    operand1 = Din[n+r];
    result = Din[d+r];
    for e = 0 to 1
        bits(32) res = Elem[result, e, 32];
        for b = 0 to 3
            element1 = Int(Elem[operand1, 4 * e + b, 8], op1_unsigned);
            element2 = Int(Elem[operand2, 4 * i + b, 8], op2_unsigned);
            res = res + element1 * element2;
        Elem[result, e, 32] = res;
    D[d+r] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.260 VUSDOT (vector)

Dot Product vector form with mixed-sign integers. This instruction performs the dot product of the four unsigned 8-bit integer values in each 32-bit element of the first source register with the four signed 8-bit integer values in the corresponding 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2, this is an OPTIONAL instruction. ID_ISAR6.I8MM indicates whether this instruction is supported in the T32 and A32 instruction sets.

A1

ARMv8.6

31 30 29 28	27 26 25 24	23	22	21 20	19 16	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	0	1	D	1 0	Vn	Vd	1	1	0	1	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VUSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VUSDOT{<q>}.S8 <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer regs = if Q == '1' then 2 else 1;
```

T1

ARMv8.6

15 14 13 12	11 10 9 8	7	6	5 4	3 0	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	0	1	D	1 0	Vn	Vd	1	1	0	1	N	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VUSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VUSDOT{<q>}.S8 <Qd>, <Qn>, <Qm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Decode for all variants of this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer regs = if Q == '1' then 2 else 1;
```

Assembler symbols

<q>	See *Standard assembler syntax fields* on page F2-4080.
<Qd>	Is the 128-bit name of the SIMD&FP third source and destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd>	Is the 64-bit name of the SIMD&FP third source and destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
CheckAdvSIMDEnabled();
bits(64) operand1;
bits(64) operand2;
bits(64) result;

for r = 0 to regs-1
    operand1 = Din[n+r];
    operand2 = Din[m+r];
    result = Din[d+r];
    for e = 0 to 1
        bits(32) res = Elem[result, e, 32];
        for b = 0 to 3
            element1 = UInt(Elem[operand1, 4 * e + b, 8]);
            element2 = SInt(Elem[operand2, 4 * e + b, 8]);
            res = res + element1 * element2;
        Elem[result, e, 32] = res;
    D[d+r] = result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.261 VUSMMLA

The widening integer matrix multiply-accumulate instruction multiplies the 2x8 matrix of unsigned 8-bit integer values held in the first source vector by the 8x2 matrix of signed 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator held in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2, this is an OPTIONAL instruction. ID_ISAR6.I8MM indicates whether this instruction is supported in the T32 and A32 instruction sets.

A1

ARMv8.6

31 30 29 28	27 26 25 24	23	22	21 20	19 16	15 12	11 10	9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	0	1	D 1 0	Vn	Vd	1 1	0 0	N	1	M	0	Vm
			B									U	

A1 variant

```
VUSMMLA{<q>}.S8 <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
case B:U of
    when '00' op1_unsigned = FALSE; op2_unsigned = FALSE;
    when '01' op1_unsigned = TRUE;  op2_unsigned = TRUE;
    when '10' op1_unsigned = TRUE;  op2_unsigned = FALSE;
    when '11' UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
```

T1

ARMv8.6

15 14 13 12	11 10 9 8	7	6	5 4	3 0	15 12	11 10	9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	0	1	D 1 0	Vn	Vd	1 1	0 0	N	1	M	0	Vm
			B									U	

T1 variant

```
VUSMMLA{<q>}.S8 <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
case B:U of
    when '00' op1_unsigned = FALSE; op2_unsigned = FALSE;
    when '01' op1_unsigned = TRUE;  op2_unsigned = TRUE;
    when '10' op1_unsigned = TRUE;  op2_unsigned = FALSE;
    when '11' UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Assembler symbols

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Qd>` Is the 128-bit name of the SIMD&FP third source and destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation for all encodings

```
CheckAdvSIMDEnabled();
bits(128) operand1 = Q[n>>1];
bits(128) operand2 = Q[m>>1];
bits(128) addend   = Q[d>>1];

Q[d>>1] = MatMulAdd(addend, operand1, operand2, op1_unsigned, op2_unsigned);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.262 VUZP

Vector Unzip de-interleaves the elements of two vectors.

The elements of the vectors can be 8-bit, 16-bit, or 32-bit. There is no distinction between data types.

The following figure shows the operation of VUZP doubleword operation for data type 8.

VUZP.8, doubleword

	Register state before operation								Register state after operation							
Dd	A7	A6	A5	A4	A3	A2	A1	A0	B6	B4	B2	B0	A6	A4	A2	A0
Dm	B7	B6	B5	B4	B3	B2	B1	B0	B7	B5	B3	B1	A7	A5	A3	A1

The following figure shows the operation of VUZP quadword operation for data type 32.

VUZP.32, quadword

	Register state before operation				Register state after operation			
Qd	A3	A2	A1	A0	B2	B0	A2	A0
Qm	B3	B2	B1	B0	B3	B1	A3	A1

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 18	17 16	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	1	1	size	1 0	Vd	0	0	0	1	0	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VUZP{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VUZP{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' || (Q == '0' && size == '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
quadword_operation = (Q == '1');  esize = 8 << UInt(size);
d = UInt(D:Vd);  m = UInt(M:Vm);
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 2 1 0	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	1	D	1	1	size	1 0	Vd	0	0	0	1	0	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VUZP{<c>}{<q>}.<dt> <Dd>, <Dm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VUZP{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' || (Q == '0' && size == '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
quadword_operation = (Q == '1');  esize = 8 << UInt(size);
d = UInt(D:Vd);  m = UInt(M:Vm);
```

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` For the 64-bit SIMD vector variant: is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:

`8` when `size = 00`

`16` when `size = 01`

The encoding `size = 1x` is reserved.

For the 128-bit SIMD vector variant: is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:

`8` when `size = 00`

`16` when `size = 01`

`32` when `size = 10`

The encoding `size = 11` is reserved.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if quadword_operation then
        if d == m then
            Q[d>>1] = bits(128) UNKNOWN;  Q[m>>1] = bits(128) UNKNOWN;
        else
            zipped_q = Q[m>>1]:Q[d>>1];
            for e = 0 to (128 DIV esize) - 1
                Elem[Q[d>>1],e,esize] = Elem[zipped_q,2*e,esize];
                Elem[Q[m>>1],e,esize] = Elem[zipped_q,2*e+1,esize];
    else
        if d == m then
            D[d] = bits(64) UNKNOWN;  D[m] = bits(64) UNKNOWN;
        else
            zipped_d = D[m]:D[d];
            for e = 0 to (64 DIV esize) - 1
                Elem[D[d],e,esize] = Elem[zipped_d,2*e,esize];
                Elem[D[m],e,esize] = Elem[zipped_d,2*e+1,esize];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.263 VUZP (alias)

Vector Unzip de-interleaves the elements of two vectors

This instruction is a pseudo-instruction of the VTRN instruction. This means that:

- The encodings in this description are named to match the encodings of VTRN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VTRN gives the operational pseudocode for this instruction.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 18 17 16	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	1	D	1	1	size	1 0	Vd	0	0	0	0	1	0	M	0	Vm
													Q			

64-bit SIMD vector variant

```
VUZP{<c>}{<q>}.32 <Dd>, <Dm>
```

is equivalent to

```
VTRN{<c>}{<q>}.32 <Dd>, <Dm>
```

and is never the preferred disassembly.

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 2 1 0	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	1	D	1	1	size	1 0	Vd	0	0	0	0	1	0	M	0	Vm
													Q			

64-bit SIMD vector variant

```
VUZP{<c>}{<q>}.32 <Dd>, <Dm>
```

is equivalent to

```
VTRN{<c>}{<q>}.32 <Dd>, <Dm>
```

and is never the preferred disassembly.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

The description of VTRN gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.264 VZIP

Vector Zip interleaves the elements of two vectors.

The elements of the vectors can be 8-bit, 16-bit, or 32-bit. There is no distinction between data types.

The following figure shows the operation of VZIP doubleword operation for data type 8.

VZIP.8, doubleword

	Register state before operation								Register state after operation							
Dd	A7	A6	A5	A4	A3	A2	A1	A0	B3	A3	B2	A2	B1	A1	B0	A0
Dm	B7	B6	B5	B4	B3	B2	B1	B0	B7	A7	B6	A6	B5	A5	B4	A4

The following figure shows the operation of VZIP quadword operation for data type

VZIP.32, quadword

	Register state before operation				Register state after operation			
Qd	A3	A2	A1	A0	B1	A1	B0	A0
Qm	B3	B2	B1	B0	B3	A3	B2	A2

32.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

A1

31 30 29 28	27 26 25 24	23	22	21	20	19 18	17 16	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	0 0 1 1	D	1	1		size	1 0	Vd	0	0	0	1	1	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VZIP{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VZIP{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' || (Q == '0' && size == '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
quadword_operation = (Q == '1');  esize = 8 << UInt(size);
d = UInt(D:Vd);  m = UInt(M:Vm);
```

T1

15 14 13 12	11 10 9 8	7	6	5	4	3 2	1 0	15 12	11	10	9	8	7	6	5	4	3 0
1 1 1 1	1 1 1 1	D	1	1		size	1 0	Vd	0	0	0	1	1	Q	M	0	Vm

64-bit SIMD vector variant

Applies when `Q == 0`.

```
VZIP{<c>}{<q>}.<dt> <Dd>, <Dm>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

128-bit SIMD vector variant

Applies when `Q == 1`.

```
VZIP{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' || (Q == '0' && size == '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
quadword_operation = (Q == '1');  esize = 8 << UInt(size);
d = UInt(D:Vd);  m = UInt(M:Vm);
```

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<dt>` For the 64-bit SIMD vector variant: is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:

`8` when `size = 00`

`16` when `size = 01`

The encoding `size = 1x` is reserved.

For the 128-bit SIMD vector variant: is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:

`8` when `size = 00`

`16` when `size = 01`

`32` when `size = 10`

The encoding `size = 11` is reserved.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if quadword_operation then
        if d == m then
            Q[d>>1] = bits(128) UNKNOWN;  Q[m>>1] = bits(128) UNKNOWN;
        else
            bits(256) zipped_q;
            for e = 0 to (128 DIV esize) - 1
                Elem[zipped_q,2*e,esize] = Elem[Q[d>>1],e,esize];
                Elem[zipped_q,2*e+1,esize] = Elem[Q[m>>1],e,esize];
            Q[d>>1] = zipped_q<127:0>;  Q[m>>1] = zipped_q<255:128>;
    else
        if d == m then
            D[d] = bits(64) UNKNOWN;  D[m] = bits(64) UNKNOWN;
        else
            bits(128) zipped_d;
            for e = 0 to (64 DIV esize) - 1
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        Elem[zipped_d,2*e,esize] = Elem[D[d],e,esize];
        Elem[zipped_d,2*e+1,esize] = Elem[D[m],e,esize];
    D[d] = zipped_d<63:0>;  D[m] = zipped_d<127:64>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

F6.1.265 VZIP (alias)

Vector Zip interleaves the elements of two vectors

This instruction is a pseudo-instruction of the VTRN instruction. This means that:

- The encodings in this description are named to match the encodings of VTRN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VTRN gives the operational pseudocode for this instruction.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	0	Vd		0	0	0	0	1	0	M	0	Vm	
																							Q				

64-bit SIMD vector variant

```
VZIP{<c>}{<q>}.32 <Dd>, <Dm>
```

is equivalent to

```
VTRN{<c>}{<q>}.32 <Dd>, <Dm>
```

and is never the preferred disassembly.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	0	Vd		0	0	0	0	1	0	M	0	Vm	
																							Q				

64-bit SIMD vector variant

```
VZIP{<c>}{<q>}.32 <Dd>, <Dm>
```

is equivalent to

```
VTRN{<c>}{<q>}.32 <Dd>, <Dm>
```

and is never the preferred disassembly.

Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-4080. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F2-4080.

`<q>` See *Standard assembler syntax fields* on page F2-4080.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

The description of VTRN gives the operational pseudocode for this instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Part G

The AArch32 System Level Architecture

Chapter G1
The AArch32 System Level Programmers' Model

This chapter gives a system level description of the programmers' model for execution in AArch32 state. It contains the following sections:

- *About the AArch32 System level programmers' model* on page G1-5672.
- *Exception levels* on page G1-5673.
- *Exception terminology* on page G1-5674.
- *Execution state* on page G1-5676.
- *Instruction Set state* on page G1-5678.
- *Security state* on page G1-5679.
- *Security state, Exception levels, and AArch32 execution privilege* on page G1-5682.
- *Virtualization* on page G1-5684.
- *AArch32 state PE modes, and general-purpose and Special-purpose registers* on page G1-5686.
- *Process state, PSTATE* on page G1-5695.
- *Instruction set states* on page G1-5701.
- *Handling exceptions that are taken to an Exception level using AArch32* on page G1-5703.
- *Routing of aborts taken to AArch32 state* on page G1-5722.
- *Exception return to an Exception level using AArch32* on page G1-5725.
- *Asynchronous exception behavior for exceptions taken from AArch32 state* on page G1-5730.
- *AArch32 state exception descriptions* on page G1-5738.
- *Reset into AArch32 state* on page G1-5760.
- *Mechanisms for entering a low-power state* on page G1-5764.
- *The AArch32 System register interface* on page G1-5769.
- *Advanced SIMD and floating-point support* on page G1-5772.
- *Configurable instruction enables and disables, and trap controls* on page G1-5778.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G1.1 About the AArch32 System level programmers' model

An application programmer has only a restricted view of the system. The System level programmers' model supports this application level view of the system, and includes features that are required for one or both of an operating system (OS) and a hypervisor to provide the programming environment seen by an application. This chapter describes the System level programmers' model when executing at EL1 or higher in an Exception level that is using AArch32.

The system level programmers' model includes all of the system features required to support operating systems and to handle hardware events.

The following sections give a system level introduction to the basic concepts of the Arm architecture AArch32 state, and the terminology that is used for describing the architecture when executing in this state:

- *Exception levels* on page G1-5673.
- *Exception terminology* on page G1-5674.
- *Execution state* on page G1-5676.
- *Instruction Set state* on page G1-5678.
- *Security state* on page G1-5679.
- *Virtualization* on page G1-5684.

The rest of this chapter describes the system level programmers' model when executing in AArch32 state.

The other chapters in this part describe:

- The memory system architecture, as seen when executing in an Exception level that is using AArch32:
  - Chapter G4 *The AArch32 System Level Memory Model* describes the general features of the Armv8 memory model, when executing in AArch32 state, that are not visible at the application level.

    Note

    Chapter E2 *The AArch32 Application Level Memory Model* describes the application level view of the memory model.

  - Chapter G5 *The AArch32 Virtual Memory System Architecture* describes the *Virtual Memory System Architecture* (VMSA) used in AArch32 state.
- The AArch32 System registers, see Chapter G8 *AArch32 System Register Descriptions*.

Note

The T32 and A32 instruction sets include instructions that provide system level functionality, such as returning from an exception. See for example, *ERET* on page F5-4369.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G1.2 Exception levels

The Armv8-A architecture defines a set of Exception levels, EL0 to EL3, where:

- If ELn is the Exception level, increased values of *n* indicate increased software execution privilege.
- Execution at EL0 is called *unprivileged execution*.
- EL2 provides support for virtualization.
- EL3 provides support for switching between two Security states, Secure state and Non-secure state.

An implementation might not include all of the Exception levels. All implementations must include EL0 and EL1. EL2 and EL3 are optional.

Note

A PE is not required to implement a contiguous set of Exception levels. For example, it is permissible for an implementation to include only EL0, EL1, and EL3.

The effect of implementation choices on the programmers' model on page D1-2419 provides information on implementations.

When executing in AArch32 state, execution can move between Exception levels only on taking an exception or on returning from an exception:

- On taking an exception, the Exception level can only increase or remain the same.
- On returning from an exception, the Exception level can only decrease or remain the same.

The Exception level that execution changes to or remains in on taking an exception is called the *target Exception level* of the exception.

Each exception type has a target Exception level that is either:

- Implicit in the nature of the exception.
- Defined by configuration bits in the System registers.

An exception cannot target EL0.

Exception levels exist within *Security states*. *The Armv8-A security model* on page G1-5679 describes this. When executing at an Exception level, the PE can access both of the following:

- The resources that are available for the combination of the current Exception level and the current Security state.
- The resources that are available at all lower Exception levels, provided that those resources are available to the current Security state.

This means that if the implementation includes EL3, then because EL3 is only implemented in Secure state, execution at EL3 can access all resources available at all Exception levels, for both Security states.

Each Exception level other than EL0 has its own translation regime and associated control registers. For information on the translation regimes, see Chapter G5 *The AArch32 Virtual Memory System Architecture*.

G1.2.1 Typical Exception level usage model

The architecture does not specify what software uses which Exception level. Such choices are outside the scope of the architecture. However, the following is a common usage model for the Exception levels:

EL0 Applications.

EL1 OS kernel and associated functions that are typically described as *privileged*.

EL2 Hypervisor.

EL3 Secure monitor.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G1.3 Exception terminology

The following subsections define the terms that are used when describing exceptions:

- *Terminology for taking an exception*.
- *Terminology for returning from an exception*.
- *Exception levels*.
- *Definition of a precise exception*.
- *Definitions of synchronous and asynchronous exceptions* on page G1-5675.

G1.3.1 Terminology for taking an exception

An exception is *generated* when the PE first responds to an exceptional condition. The PE state at this time is the state that the exception is *taken from*. The PE state immediately after taking the exception is the state that the exception is *taken to*.

G1.3.2 Terminology for returning from an exception

To return from an exception, the PE must execute an exception return instruction. The PE state when an exception return instruction is committed for execution is the state the exception *returns from*. The PE state immediately after the execution of that instruction is the state that the exception *returns to*.

G1.3.3 Exception levels

An Exception level, EL*n*, with a larger value of *n* than another Exception level, is described as being a *higher* Exception level than the other Exception level. For example, EL3 is a higher Exception level than EL1.

An Exception level with a smaller value of *n* than another Exception level is described as being a *lower* Exception level than the other Exception level. For example, EL0 is a lower Exception level than EL1.

An Exception level is described as:

- *Using AArch64* when execution in that Exception level is in the AArch64 Execution state.
- *Using AArch32* when execution in that Exception level is in the AArch32 Execution state.

G1.3.4 Definition of a precise exception

An exception is described as *precise* when the exception handler receives the PE state and memory system state that is consistent with the PE having executed all of the instructions up to but not including the point in the instruction stream where the exception was taken, and none afterwards.

Other than the SError interrupt all exceptions that are taken to AArch32 state are required to be precise. For each occurrence of an SError interrupt, whether the interrupt is precise or imprecise is IMPLEMENTATION DEFINED.

Where a synchronous exception that is taken to AArch32 state is generated as part of an instruction that performs more than one single-copy atomic memory access, the definition of precise permits that the values in registers or memory affected by those instructions can be UNKNOWN, provided that:

- The accesses affecting those registers or memory locations do not, themselves, generate exceptions.
- The registers are not involved in the calculation of the memory address that is used by the instruction.

In AArch32 state, examples of instructions that perform more than one single-copy atomic memory access are the `LDM` and `STM` instructions.

Note

- For the definition of a single-copy atomic access, see *Properties of single-copy atomic accesses* on page E2-3977.
- The SError interrupt replaces the Armv7 asynchronous abort.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G1.3.5 Definitions of synchronous and asynchronous exceptions

An exception is described as *synchronous* if all of the following apply:

- The exception is generated as a result of direct execution or attempted execution of an instruction.
- The return address presented to the exception handler is guaranteed to indicate the instruction that caused the exception.
- The exception is precise.

An exception is described as *asynchronous* if any of the following apply:

- The exception is not generated as a result of direct execution or attempted execution of the instruction stream.
- The return address presented to the exception handler is not guaranteed to indicate the instruction that caused the exception.
- The exception is imprecise.

For more information about exceptions, see *Handling exceptions that are taken to an Exception level using AArch32* on page G1-5703.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G1.4 Execution state

The Execution states are:

AArch64 The 64-bit Execution state.

AArch32 The 32-bit Execution state. Operation in this state is compatible with Armv7-A operation.

Execution state on page A1-37 gives more information about them.

Exception levels *use* Execution states. For example, EL0, EL1 and EL2 might all be using AArch32, under EL3 using AArch64.

This means that:

- Different software layers, such as an application, an operating system kernel, and a hypervisor, executing at different Exception levels, can execute in different Execution states.
- The PE can change Execution states only either:
  - At reset.
  - On a change of Exception level.

Note

- *Typical Exception level usage model* on page G1-5673 shows which Exception levels different software layers might typically use.
- *The effect of implementation choices on the programmers' model* on page D1-2419 gives information on supported configurations of Exception levels and Execution states.

The interaction between the AArch64 and AArch32 Execution states is called *interprocessing*. *Interprocessing* on page D1-2406 describes this.

G1.4.1 About the AArch32 PE modes

AArch32 state provides a set of *PE modes* that support normal software execution and handle exceptions. The current mode determines the set of registers that are available, as described in *AArch32 general-purpose registers, the PC, and the Special-purpose registers* on page G1-5691.

The AArch32 modes are:

- Monitor mode. This mode always executes at Secure EL3.
- Hyp mode. This mode always executes at Non-secure EL2.
- System, Supervisor, Abort, Undefined, IRQ, and FIQ modes. The Exception level these modes execute at depends on the Security state, as described in *Security state* on page G1-5679.
- User mode. This mode always executes at EL0.

Note

AArch64 state does not support modes. Modes are a concept that is specific to AArch32 state. Modes that execute at a particular Exception level are only implemented if that Exception level supports using AArch32 state.

For more information on modes, see *AArch32 state PE mode descriptions* on page G1-5686.

The mode in use immediately before an exception is taken is described as the mode the exception is *taken from*. The mode that is used on taking the exception is described as the mode the exception is *taken to*.

All of the following define the mode that an exception is taken to:

- The type of exception.
- The mode the exception is taken from.
- Configuration settings defined at EL2 and EL3.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Monitor mode and Hyp mode can create system traps that cause exceptions to EL3 or EL2 respectively. There is an architected hierarchy where EL2 and EL3 configuration settings affect a common condition, for example interrupt routing. When no traps are enabled for a particular condition, the AArch32 mode an exception is taken to is called the *default mode* for that exception.

In AArch32 state, a number of different modes can exist at the same Exception level. All modes at a particular Exception level have the same *execution privilege*, meaning they have the same access rights for accesses to memory and to System registers. However, the mapping of PE modes to Exception levels depends on the Security state, as described in *Security state* on page G1-5679. *Security state, Exception levels, and AArch32 execution privilege* on page G1-5682 gives more information about the PE modes, their associated execution privilege, and how this maps onto the Exception levels.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G1.5 Instruction Set state

In AArch32 state, the *Instruction Set state* determines the instruction set that the PE is executing. In an implementation that follows the Arm recommendations, the available Instruction Set states are:

T32 state The PE is executing T32 instructions.

A32 state The PE is executing A32 instructions.

Note

In previous versions of the Arm architecture:

- The T32 instruction set was called the Thumb instruction set.
- The A32 instruction set was called the ARM instruction set.

For more information, see *Process state, PSTATE* on page E1-3953.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G1.6 Security state

The Armv8-A architecture provides two Security states, each with an associated physical memory address space, as follows:

Secure state When in this state, the PE can access both the Secure physical address space and the Non-secure physical address space.

Non-secure state When in this state, the PE:
- Can access only the Non-secure physical address space.
- Cannot access the Secure system control resources.

For information on how virtual addresses translate onto Secure physical and Non-secure addresses, see *About VMSAv8-32* on page G5-5922.

G1.6.1 The Armv8-A security model

The principles of the Armv8-A security model are defined in *The Armv8-A security model* on page D1-2318.

The AArch32 security model, and execution privilege

The Exception level hierarchy of four Exception levels, EL0, EL1, EL2, and EL3, applies to execution in both Execution states. This section describes the mapping between Exception levels, AArch32 modes, and execution privilege.

The AArch32 modes Monitor, System, Supervisor, Abort, Undefined, IRQ, and FIQ all have the same execution privilege.

In Secure state:

- Monitor mode executes only at EL3, and is accessible only when EL3 is using AArch32.
- System mode, Supervisor mode, Abort mode, Undefined mode, IRQ mode, and FIQ mode all:
  - Execute at EL1 when EL3 is using AArch64.
  - Execute at EL3 when EL3 is using AArch32.

This means that there is a difference in the Secure state hierarchy that the PE is using, depending on which Execution state EL3 is using:

- If EL3 is using AArch64:
  - There is no support for Monitor mode.
  - If EL1 is using AArch32, System mode, Supervisor mode, Abort mode, Undefined mode, IRQ mode, and FIQ mode execute at Secure EL1.
- If EL3 is using AArch32:
  - Monitor mode is supported, and executes at Secure EL3.
  - System mode, Supervisor mode, Abort mode, Undefined mode, IRQ mode, and FIQ mode execute at Secure EL3.
  - There is no support for a Secure EL1 Exception level.

See *Security behavior in Exception levels using AArch32 when EL2 or EL3 are using AArch64* on page G1-5715 for more information about operation in a Secure EL1 mode when EL3 is using AArch64.

In Non-secure state, the PL1 modes System, Supervisor, Abort, Undefined, IRQ, and FIQ always execute at EL1.

User mode always executes at EL0 and has the lowest possible execution privilege.

Hyp mode always executes in Non-secure state at EL2 and has higher execution privilege than all of:
- User mode.
- System mode, Supervisor mode, Abort mode, Undefined mode, IRQ mode, and FIQ mode.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Limited use of Privilege level in Armv8 AArch32 state on page G1-5683 describes how, in some contexts, the concept of *Privilege levels* can be used to represent the execution privilege hierarchy.

For more information about the modes, see *About the AArch32 PE modes* on page G1-5676.

Figure G1-1 shows the security model when EL3 is using AArch32, and shows the expected use of the different Exception levels, and which modes execute at which Exception levels.

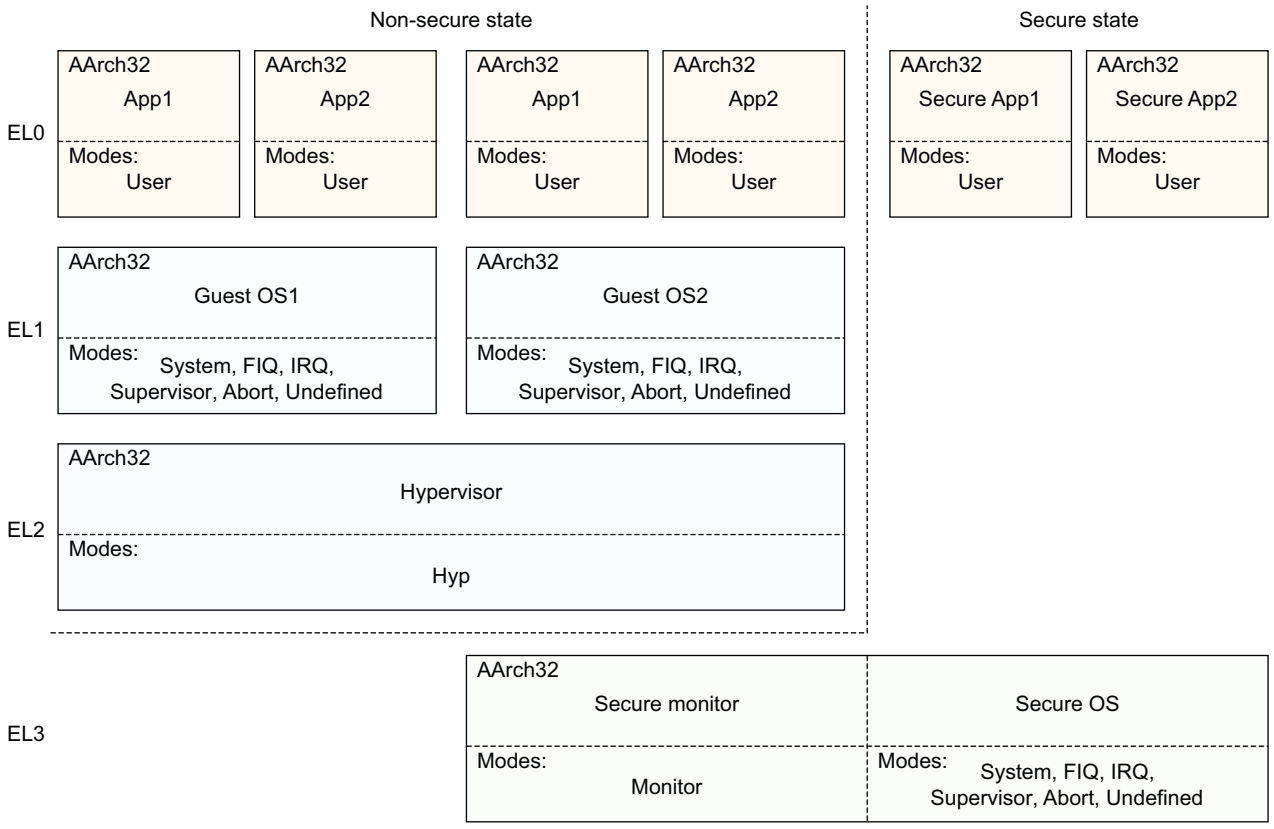


Figure G1-1 Armv8-A Security model when EL3 is using AArch32

Note

For an overview of the Security models when EL3 is using AArch64:

- See Figure G1-2 on page G1-5689 for the case where EL2, EL1, and EL0 are all using AArch32. This figure shows the implementation of the PE modes.
- See Figure D1-1 on page D1-2319 for an overview of the set of possible implementations.

Figure G1-1 shows that when EL3 is using AArch32, the Exception levels and modes available in each Security state are as follows:

Secure state

EL0 User mode.

EL3 Any mode that is available in Secure state, other than User mode.

Non-secure state

EL0 User mode.

EL1 Any mode that is available in Non-secure state, other than Hyp mode and User mode.

EL2 Hyp mode.

Execution at EL0 is described as *unprivileged execution*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A mode associated with a particular Exception level, EL*n*, is described as an EL*n* mode.

Note

The Exception level defines the ability to access resources in the current Security state, and does not imply anything about the ability to access resources in the other Security state.

When EL3 is using AArch32, many AArch32 System registers accessible at PL1 are *banked* between the Secure and Non-secure states.

When EL3 is using AArch64 and Secure EL1 is using AArch32, System registers accessible at PL1 are not banked between the Non-secure and Secure states. Software running at EL3 is expected to switch the content of the PL1-accessible System registers between the Secure and Non-secure context, in a similar manner to switching the contents of general purpose registers. For information on the relationship between AArch64 and AArch32 System registers in an interprocessing environment, see *Mapping of the System registers between the Execution states* on page D1-2409.

For more information on the System registers, see *The AArch32 System register interface* on page G1-5769.

The *Secure Monitor Call* (`SMC`) instruction provides software with a system call to EL3. When executing at a privileged Exception level, `SMC` instructions generates exceptions. For more information, see *Secure Monitor Call (SMC) exception* on page G1-5743 and *SMC* on page F5-4694.

Note

For more information about the Privilege level terminology, see *Security state, Exception levels, and AArch32 execution privilege* on page G1-5682.

Changing from Secure state to Non-secure state

Monitor mode is provided to support switching between Secure and Non-secure states. When executing in an Exception level that is using AArch32, except in Monitor mode and Hyp mode, the Security state is controlled:

- By the SCR.NS bit, when EL3 is using AArch32.
- By the SCR_EL3.NS bit, when EL3 is using AArch64.

The mapping of AArch32 privileged modes to the exception hierarchy means that it is possible when EL3 is using AArch32 to change from EL3 to Non-secure EL1 without an exception return. This can occur in one of the following ways:

- Using an `MSR` or `CPS` instruction to switch from Monitor mode to another privileged mode while SCR.NS is 1.
- Using an `MCR` instruction that writes SCR.NS to change from Secure to Non-secure state when in a privileged mode other than Monitor mode.

Arm strongly recommends that software executing at EL3 using AArch32 does not use either of these mechanisms to change from EL3 to Non-secure EL1 without an exception return. The use of both of these mechanisms is deprecated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G1.7 Security state, Exception levels, and AArch32 execution privilege

In Armv8, the hierarchy of software execution privilege, within a particular Security state, is defined by the Exception levels, with higher Exception level numbers indicating higher privilege. Table G1-1 shows this hierarchy for each Security state.

Table G1-1 Execution privilege and Exception levels, by Security state

Execution privilege	Secure state	Non-secure state	Typical use
Highest	EL3	-[a]	Secure monitor
-	EL2[b]	EL2	Hypervisor
-	EL1	EL1	Secure or Non-secure OS
Lowest, Unprivileged	EL0	EL0	Secure or Non-secure application

a. EL3 is never implemented in Non-secure state.
b. If ARMv8.4-SecEL2 is implemented in AArch64 state, EL2 can be enabled in Secure state.

When executing in AArch32 state, within a given Security state, the current PE state, including the execution privilege, is primarily indicated by the current *PE mode*. In Secure state, how the PE modes map onto the Exception levels depends on whether EL3 is using AArch32 or is using AArch64, and:

- Figure G1-1 on page G1-5680 shows this mapping when EL3 is using AArch32.
- Figure G1-2 on page G1-5689 shows this mapping when EL3 is using AArch64.

Table G1-2 shows this mapping. In interpreting this table:

- Monitor mode is implemented only in Secure state, and only if EL3 is using AArch32.
- Hyp mode is implemented only in Non-secure state, and only if EL2 is using AArch32.
- System, FIQ, IRQ, Supervisor, Abort, and Undefined modes are implemented:
  - **In Secure state** If either:
    - EL3 is using AArch32.
    - EL3 is using AArch64 and EL1 is using AArch32.
  - **In Non-secure state** If EL1 is using AArch32.
- User mode is implemented if EL0 is using AArch32.

Table G1-2 Mapping of AArch32 PE modes to Exception levels

Exception level	PE modes in the given Security state, and EL3 Execution state		
	Secure state, EL3 using AArch32	Secure state, EL3 using AArch64[a]	Non-secure state
EL3	Monitor, System, FIQ, IRQ, Supervisor, Abort, Undefined	-	-
EL2	-	-	Hyp
EL1	-	System, FIQ, IRQ, Supervisor, Abort, Undefined	System, FIQ, IRQ, Supervisor, Abort, Undefined
EL0	User	User	User

a. If ARMv8.4-SecEL2 is implemented and enabled in AArch64 State, this column can be applied to EL2.

Because AArch32 behavior is described in terms of the PE modes, and transitions between PE modes, the Exception levels are implicit in most of the description of operation in AArch32 state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G1.7.1 Limited use of Privilege level in Armv8 AArch32 state

As described in *The VMSAv8-32 translation regimes* on page G5-5924, a *translation regime* maps a virtual address (VA) to the corresponding physical address (PA). The VMSAv8-64 translation regimes are defined by the Exception levels that use them. However, because the mapping between PE modes and Exception levels in Secure state depends on whether EL3 is using AArch32 or is using AArch64, as shown in Table G1-2 on page G1-5682, the VMSAv8-32 translation regimes cannot be described simply in terms of either the Exception levels or the PE modes that use them.

To provide a consistent description of address translation as seen from AArch32 state, the VMSAv8-32 translation regimes are described in terms of the *Privilege levels* originally defined in the Armv7 descriptions of AArch32 state. Table G1-3 shows how the PE modes map to these Privilege levels:

Table G1-3 Mapping of PE modes to AArch32 Privilege levels

Privilege level	Secure state	Non-secure state
PL2	-	Hyp[a]
PL1	Monitor[b], System, FIQ, IRQ, Supervisor, Abort, Undefined	System, FIQ, IRQ, Supervisor, Abort, Undefined
PL0	User	User

a. Implemented only in Non-secure state, and only if EL2 is using AArch32 state.
b. Implemented only in Secure state, and only if EL3 is using AArch32 state.

Comparing Table G1-3 with Table G1-2 on page G1-5682 shows that:

In Non-secure state

Each privilege level maps to the corresponding Exception level. For example PL1 maps to EL1.

In Secure state

PL0 maps to EL0.

The mapping of PL1 depends on the Execution state being used by EL3, as follows:

EL3 using AArch64 Secure PL1 maps to Secure EL1. Monitor mode is not implemented.

EL3 using AArch32 Secure PL1 maps to Secure EL3. Monitor mode is implemented as one of the Secure PL1 modes.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G1.8 Virtualization

The support for virtualization described in this section applies only to an implementation that includes EL2. A PE is in *Hyp mode* when it is executing at EL2 in the AArch32 state. An exception return from Hyp mode to software running at EL1 or EL0 is performed using the `ERET` instruction.

EL2 provides a set of features that support virtualizing the Non-secure state of an Armv8-A implementation. The basic model of a virtualized system involves:

- A hypervisor, running in EL2, that is responsible for switching between *virtual machines*. A virtual machine is comprised of Non-secure EL1 and Non-secure EL0.
- A number of Guest operating systems, that each run in Non-secure EL1, on a virtual machine.
- For each Guest operating system, applications, that usually run in Non-secure EL0, on a virtual machine.

Note

In some systems, a Guest OS is unaware that it is running on a virtual machine, and is unaware of any other Guest OS. In other systems, a hypervisor makes the Guest OS aware of these facts. The Armv8-A architecture supports both of these models.

The hypervisor assigns a *virtual machine identifier* (VMID) to each virtual machine.

In AArch32 state, EL2 is implemented only in Non-secure state, to support Guest OS management. EL2 provides controls to:

- Provide virtual values for the contents of a small number of identification registers. A read of one of these registers by a Guest OS or the applications for a Guest OS returns the virtual value.
- *Trap* various operations, including memory management operations and accesses to many other registers. A trapped operation generates an exception that is taken to EL2.
- Route interrupts to the appropriate one of:
  - The current Guest OS.
  - A Guest OS that is not currently running.
  - The hypervisor.

In Non-secure state:

- The implementation provides an independent *translation regime* for memory accesses from EL2.
- For the PL1&0 translation regime, address translation occurs in two stages:
  - Stage 1 maps the *virtual address* (VA) to an *intermediate physical address* (IPA). This is managed at EL1, usually by a Guest OS. The Guest OS believes that the IPA is the *physical address* (PA).
  - Stage 2 maps the IPA to the PA. This is managed at EL2. The Guest OS might be completely unaware of this stage.

For more information on the translation regimes, see Chapter G5 *The AArch32 Virtual Memory System Architecture*.

G1.8.1 The effect of implementing EL2 on the Exception model

An implementation that includes EL2 implements the following exceptions:

- Hypervisor Call (HVC) exception.
- Traps to EL2. *EL2 configurable controls* on page G1-5787, describes these.
- All of the virtual interrupts:
  - Virtual SError.
  - Virtual IRQ.
  - Virtual FIQ.

HVC exceptions are always taken to EL2. All virtual interrupts are always taken to EL1, and can only be taken from Non-secure EL1 or EL0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Each of the virtual interrupts can be independently enabled using controls at EL2.

Each of the virtual interrupts has a corresponding physical interrupt. See *Virtual interrupts*.

When a virtual interrupt is enabled, its corresponding physical exception is taken to EL2, unless EL3 has configured that physical exception to be taken to EL3. For more information, see *Asynchronous exception behavior for exceptions taken from AArch32 state* on page G1-5730.

An implementation that includes EL2 also:

- Provides controls that can be used to route some synchronous exceptions, taken from Non-secure state, to EL2. For more information, see:
  - *Routing exceptions from Non-secure EL0 to EL2* on page G1-5718.
  - *Routing debug exceptions to EL2 using AArch32* on page G1-5720.
  - *Routing of aborts taken to AArch32 state* on page G1-5722
- Provides mechanisms to trap PE operations to EL2. See *EL2 configurable controls* on page G1-5787.

  When an operation is trapped to EL2, the hypervisor typically either:
  - Emulates the required operation. The application running in the Guest OS is unaware of the trap.
  - Returns an error to the Guest OS.

Virtual interrupts

The virtual interrupts have names that correspond to the physical interrupts, as shown in Table G1-4.

Table G1-4 The virtual interrupts

Physical interrupt	Corresponding virtual interrupt
External SError	Virtual SError
IRQ	Virtual IRQ
FIQ	Virtual FIQ

Software executing at EL2 can use virtual interrupts to signal physical interrupts to Non-secure EL1 and Non-secure EL0. Example G1-1 shows a usage model for virtual interrupts.

Example G1-1 Virtual interrupt usage model

A usage model is as follows:

1. Software executing at EL2 routes a physical interrupt to EL2.
2. When a physical interrupt of that type occurs, the exception handler executing in EL2 determines whether the interrupt can be handled in EL2 or requires routing to a Guest OS in EL1. If the interrupt requires routing to a Guest OS:
   - If the Guest OS is currently running, the hypervisor uses the appropriate virtual interrupt type to signal the physical interrupt to the Guest OS.
   - If the Guest OS is not currently running, the physical interrupt is marked as pending for the guest OS. When the hypervisor next switches to the virtual machine that is running that Guest OS, the hypervisor uses the appropriate virtual interrupt type to signal the physical interrupt to the Guest OS.

Non-secure EL1 and Non-secure EL0 modes cannot distinguish a virtual interrupt from the corresponding physical interrupt.

For more information, see *Virtual exceptions when an implementation includes EL2* on page G1-5730.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G1.9 AArch32 state PE modes, and general-purpose and Special-purpose registers

The following sections describe the AArch32 PE modes and the general-purpose registers and the PC:

- *AArch32 state PE mode descriptions*.
- *AArch32 general-purpose registers, the PC, and the Special-purpose registers* on page G1-5691.
- *Saved Program Status Registers (SPSRs)* on page G1-5693.
- *ELR_hyp* on page G1-5694.

Note

The PC is included in the scope of this section because, in AArch32 state, it is defined as being part of the same register file as the general-purpose registers. That is, the AArch32 register file R0-R15 comprises:

- The general-purpose registers R0-R14.
- The PC, that can be described as R15.

G1.9.1 AArch32 state PE mode descriptions

Table G1-5 shows the PE modes defined by the Arm architecture, for execution in AArch32 state. In this table:

- The *PE mode* column gives the name of each mode and the abbreviation used, for example, in the general-purpose register name suffixes used in *AArch32 general-purpose registers, the PC, and the Special-purpose registers* on page G1-5691.
- The *Encoding* column gives the corresponding PSTATE.M field.
- The *Exception level* column gives the Exception level at which the mode is implemented, including dependencies on the current Security state and on whether EL3 is using AArch32, see *Exception levels* on page G1-5673.

Table G1-5 AArch32 PE modes

PE mode		Encoding	Security state	Exception level	Implemented
User	usr	10000	Both	EL0	Always
FIQ	fiq	10001	Non-secure Secure	EL1 EL1 or EL3[a]	Always
IRQ	irq	10010	Non-secure Secure	EL1 EL1 or EL3[a]	Always
Supervisor	svc	10011	Non-secure Secure	EL1 EL1 or EL3[a]	Always
Monitor	mon	10110	Secure	EL3	If EL3 implemented and using AArch32
Abort	abt	10111	Non-secure Secure	EL1 EL1 or EL3[a]	Always
Hyp	hyp	11010	Non-secure	EL2	If EL2 implemented and using AArch32
Undefined	und	11011	Non-secure Secure	EL1 EL1 or EL3[a]	Always
System	sys	11111	Non-secure Secure	EL1 EL1 or EL3[a]	Always

a. EL3 if EL3 is using AArch32. EL1 if EL3 is using AArch64 and EL1 is using AArch32.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

ARMv8.4-SecEL2 is not supported if EL2 is using AArch32.

Mode changes can be made under software control, or can be caused by an external or internal exception.

Notes on the AArch32 PE modes

PE modes are defined only in AArch32 state. Because each mode is implemented as part of a particular Exception level that is using AArch32, the set of available modes depends on which Exception levels are implemented and using AArch32, as described in *Effect of the EL3 Execution state on the PE modes and Exception levels* on page G1-5688.

This section gives more information about each of the modes, when it is implemented.

User mode Software executing in User mode executes at EL0. Execution in User mode is sometimes described as unprivileged execution. Application programs normally execute in User mode, and any program executed in User mode:

- Makes only unprivileged accesses to system resources, meaning it cannot access protected system resources.
- Makes only unprivileged access to memory.
- Cannot change mode except by causing an exception, see *Handling exceptions that are taken to an Exception level using AArch32* on page G1-5703.

System mode System mode is implemented at EL1 or EL3, see *Effect of the EL3 Execution state on the PE modes and Exception levels* on page G1-5688.

System mode has the same registers available as User mode, and is not entered by any exception.

Supervisor mode

Supervisor mode is implemented at EL1 or EL3, see *Effect of the EL3 Execution state on the PE modes and Exception levels* on page G1-5688.

Supervisor mode is the default mode to which a Supervisor Call exception is taken. Executing an `SVC` (Supervisor Call) instruction generates a Supervisor Call exception.

In an implementation where the highest implemented Exception level is using AArch32, if that Exception level is EL3 or EL1, a PE enters Supervisor mode on Reset.

Abort mode Abort mode is implemented at EL1 or EL3, see *Effect of the EL3 Execution state on the PE modes and Exception levels* on page G1-5688.

Abort mode is the default mode to which a Data Abort exception or Prefetch Abort exception is taken.

Undefined mode

Undefined mode is implemented at EL1 or EL3, see *Effect of the EL3 Execution state on the PE modes and Exception levels* on page G1-5688.

Undefined mode is the default mode to which an instruction-related exception, including any attempt to execute an `UNDEFINED` instruction, is taken.

FIQ mode FIQ mode is implemented at EL1 or EL3, see *Effect of the EL3 Execution state on the PE modes and Exception levels* on page G1-5688.

FIQ mode is the default mode to which an FIQ interrupt is taken.

IRQ mode IRQ mode is implemented at EL1 or EL3, see *Effect of the EL3 Execution state on the PE modes and Exception levels* on page G1-5688.

IRQ mode is the default mode to which an IRQ interrupt is taken.

Hyp mode Hyp mode is the Non-secure EL2 mode.

Hyp mode is entered on taking an exception from Non-secure state that must be taken to EL2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In an implementation where the highest implemented Exception level is EL2 and EL2 uses AArch32 on reset, a PE enters Hyp mode on Reset.

The Hypervisor Call exception and Hyp Trap exception are implemented as part of EL2 and are always taken to Hyp mode.

Note

This means that Hypervisor Call and Hyp Trap exceptions cannot be taken from Secure state.

When the value of the Hypervisor Call enable bit, SCR.HCE, is 1, executing an `HVC` (Hypervisor Call) instruction in a Non-secure EL1 mode generates a Hypervisor Call exception.

For more information, see *Hyp mode* on page G1-5689.

Monitor mode

Monitor mode is the Secure EL3 mode. This means it is always in the Secure state, regardless of the value of the SCR.NS bit.

Monitor mode is the mode to which a Secure Monitor Call exception is taken. In a Non-secure EL1 mode, or a Secure EL3 mode, executing an `SMC` (Secure Monitor Call) instruction generates a Secure Monitor Call exception.

When EL3 is using AArch32, some exceptions that are taken to a different mode by default can be configured to be taken to EL3, see *PE mode for taking exceptions* on page G1-5713.

When EL3 is using AArch32, software executing in Monitor mode:

- Has access to both the Secure and Non-secure copies of System registers.
- Can perform an exception return to Secure state, or to Non-secure state.

This means that, when EL3 is using AArch32, Monitor mode provides the only recommended method of changing between the Secure and Non-secure Security states.

Secure and Non-secure modes

In an implementation that includes EL3, the names of most implemented modes can be qualified as Secure or Non-secure, to indicate whether the PE is also in Secure state or Non-secure state. For example:

- If a PE is in Supervisor mode and Secure state, it is in *Secure Supervisor mode*.
- If a PE is in User mode and Non-secure state, it is in *Non-secure User mode*.

Note

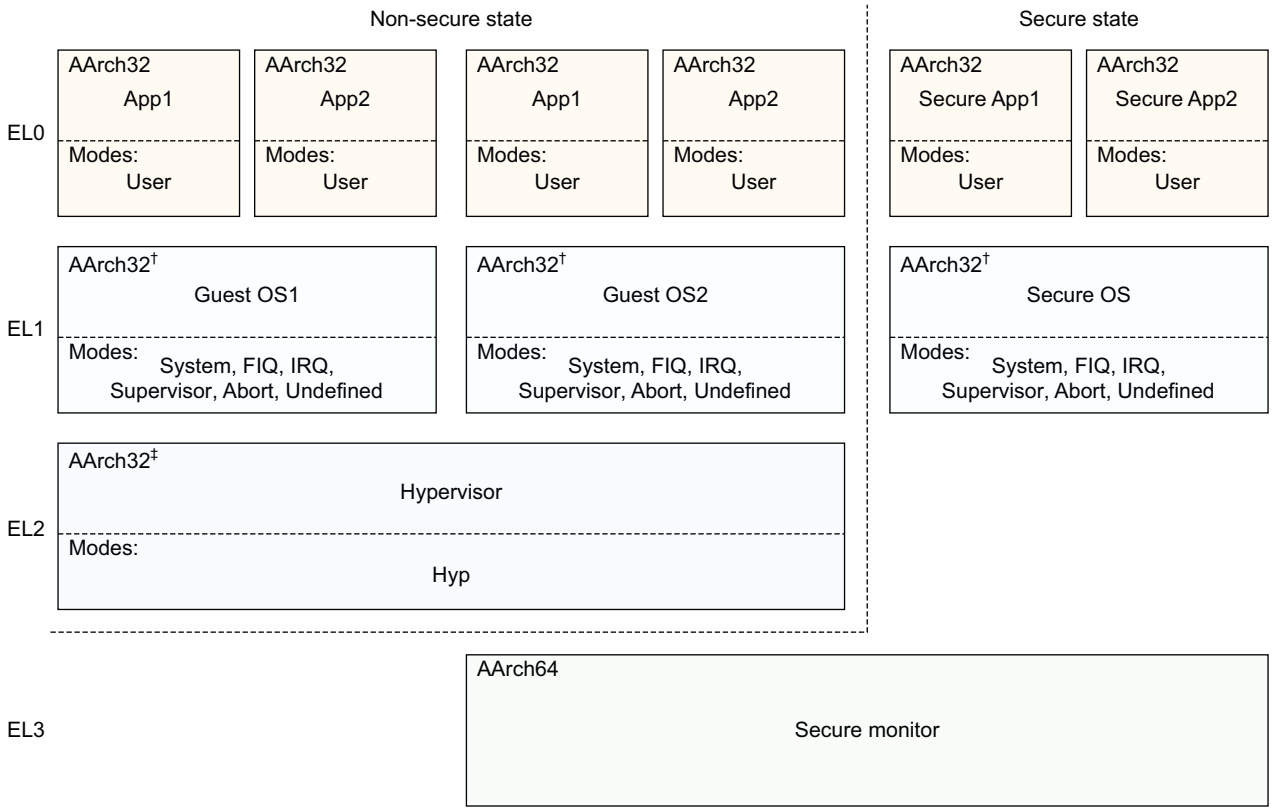
As indicated in the appropriate Mode descriptions:

- Monitor mode is a Secure mode, meaning it is always in the Secure state.
- Hyp mode is a Non-secure mode, meaning it is accessible only in Non-secure state.

Effect of the EL3 Execution state on the PE modes and Exception levels

Figure G1-1 on page G1-5680 shows the PE modes, Exception levels, and Security states, for an implementation that includes all of the Exception levels, when EL3 is using AArch32. Figure G1-2 on page G1-5689 shows how the implemented modes change when EL3 is using AArch64.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta



† When EL1 is using AArch64, System, FIQ, IRQ, Supervisor, Abort, and Undefined modes are not implemented
‡ When EL2 is using AArch64, Hyp mode is not implemented

Figure G1-2 Armv8 Exception levels, and PE modes, when EL3 is using AArch64

Comparing Figure G1-1 on page G1-5680 and Figure G1-2 shows how, in Secure state only, the implementation of System, FIQ, IRQ, Supervisor, Abort, and Undefined mode depends on the Execution state that EL3 is using. That is, these modes are implemented as follows:

Non-secure state

If Non-secure EL1 is using AArch32, then System, FIQ, IRQ, Supervisor, Abort, and Undefined modes are implemented as part of EL1. Otherwise, these modes are not implemented in Non-secure state.

Secure state The implementation of these modes depends on the Execution state that EL3 is using, as follows:

EL3 using AArch64 If Secure EL1 is using AArch32, then System, FIQ, IRQ, Supervisor, Abort, and Undefined modes are implemented as part of EL1. Otherwise, these modes are not implemented in Secure state.

EL3 using AArch32 In Secure state, System, FIQ, IRQ, Supervisor, Abort, and Undefined modes are implemented as part of EL3, see Figure G1-1 on page G1-5680.

Hyp mode

Hyp mode is the Non-secure EL2 mode. When EL2 is using AArch32, it provides the usual method of controlling the virtualization of Non-secure execution at EL1 and EL0.

Note

The alternative method of controlling this functionality is by accessing the EL2 controls from EL3 with the SCR_EL3.NS or SCR.NS bit set to 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This section summarizes how Hyp mode differs from the other modes, and references where this part of the manual describes the features of Hyp mode in more detail:

- Software executing in Hyp mode executes at EL2, see Figure G1-1 on page G1-5680.
- Hyp mode is accessible only in Non-secure state. In Secure state, an attempt by a `CPS` or an `MSR` instruction to change PSTATE.M to Hyp mode is an illegal change to PSTATE.M, as described in *Illegal changes to PSTATE.M* on page G1-5699.
- In Non-debug state, the only mechanisms for changing to Hyp mode are:
  - An exception taken from a Non-secure EL1 or EL0 mode.
  - When EL3 is using AArch32, an exception return from Secure Monitor mode.
  - When EL3 is using AArch64, an exception return from EL3.
- In Hyp mode, the only exception return is execution of an `ERET` instruction, see *ERET* on page F5-4369.
- In Hyp mode, the CPACR has no effect on the execution of;
  - System register access instructions.
  - Advanced SIMD and floating-point instructions.

  The HCPTR controls execution of these instructions in Hyp mode.
- If software running in Hyp mode executes an `SVC` instruction, the Supervisor Call exception generated by the instruction is taken to Hyp mode, see *SVC* on page F5-4847.
- An exception return with restored PSTATE specifying Hyp mode is an *illegal return event*, as described in *Illegal return events from AArch32 state* on page G1-5726, if any of the following applies:
  - EL3 is using AArch64 and the value of SCR_EL3.NS is 0.
  - EL3 is using AArch32 and the value of SCR.NS is 0.
  - The return is from a Non-secure EL1 mode.
- The instructions described in the following sections are UNDEFINED if executed in Hyp mode:
  - SRS. See *SRS, SRSDA, SRSDB, SRSIA, SRSIB* on page F5-4729.
  - RFE. See *RFE, RFEDA, RFEDB, RFEIA, RFEIB* on page F5-4624.
  - *LDM (exception return)* on page F5-4402.
  - *LDM (User registers)* on page F5-4404.
  - *STM (User registers)* on page F5-4769.
  - The `SUBS PC, LR` forms of the instructions described in *SUB, SUBS (immediate)* on page F5-4831.

    Note

    In T32 state, `ERET` is encoded as `SUBS PC, LR, #0`, and therefore this is a valid instruction.

  - The exception return form of the instructions described in *MOV, MOVS (register)* on page F5-4517.

  In addition, deprecated forms of the A32 `ADCS`, `ADDS`, `ANDS`, `BICS`, `EORS`, `MOVS`, `MVNS`, `ORRS`, `RSBS`, `RSCS`, `SBCS`, and `SUBS` instructions with the PC as the destination register are UNDEFINED if executed in Hyp mode. The instruction descriptions identify these UNDEFINED cases.
- The Load unprivileged and Store unprivileged instructions `LDRT`, `LDRSHT`, `LDRHT`, `LDRBT`, `STRT`, `STRHT`, and `STRBT`, are CONSTRAINED UNPREDICTABLE if executed in Hyp mode, see *Execution of Load/Store unprivileged instructions in Hyp mode* on page K1-7851.

In an implementation that includes EL3, from reset, the `HVC` instruction is UNDEFINED in Non-secure EL1 modes, meaning entry to Hyp mode is disabled by default. To permit entry to Hyp mode using the Hypervisor Call exception, Secure software must enable use of the `HVC` instruction:

- By setting the SCR_EL3.HCE bit to 1, if EL3 is using AArch64.
- By setting the SCR.HCE bit to 1, if EL3 is using AArch32.

If EL3 is implemented and using AArch32, and SCR.HCE bit is set to 0, the `HVC` instruction is UNPREDICTABLE in Hyp mode. The instruction is either UNDEFINED or executes as a `NOP`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If EL3 is implemented and using AArch64, and SCR_EL3.HCE bit is set to 0, the `HVC` instruction is UNDEFINED in Hyp mode.

If EL3 is not implemented and HCR_EL2 or HCR.HCD is set to 1, the `HVC` instruction is UNDEFINED in Hyp mode.

Pseudocode description of mode operations

The `BadMode()` function tests whether a 5-bit mode number corresponds to one of the permitted modes.

The `BadMode()` function is defined in Chapter J1 *Armv8 Pseudocode*.

G1.9.2 AArch32 general-purpose registers, the PC, and the Special-purpose registers

The general-purpose registers, and the PC, in AArch32 state on page E1-3951 describes the application level view of the general-purpose registers, and the PC. This view provides:

- The general-purpose registers R0-R14, of which:
  - The preferred name for R13 is SP (*stack pointer*).
  - The preferred name for R14 is LR (*link register*).
- The PC, that can be described as R15.

These registers are selected from a larger set of registers, that includes *banked* copies of some registers, with the current register selected by the execution mode. The implementation and banking of the general-purpose registers depends on whether or not the implementation includes EL2 and EL3, and whether those Exception levels are using AArch32. Figure G1-3 on page G1-5692 shows the full set of banked general-purpose registers, and the Special-purpose registers:

- The Program Status Registers CPSR and SPSR.
- ELR_hyp.

Note

The architecture uses system level register names, such as R0_usr, R8_usr, and R8_fiq, when it must identify a specific register. The application level names refer to the registers for the current mode, and usually are sufficient to identify a register.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Application level view	System level view								
	User	System	Hyp †	Supervisor	Abort	Undefined	Monitor ‡	IRQ	FIQ
R0	R0_usr								
R1	R1_usr								
R2	R2_usr								
R3	R3_usr								
R4	R4_usr								
R5	R5_usr								
R6	R6_usr								
R7	R7_usr								
R8	R8_usr								R8_fiq
R9	R9_usr								R9_fiq
R10	R10_usr								R10_fiq
R11	R11_usr								R11_fiq
R12	R12_usr								R12_fiq
SP	SP_usr		SP_hyp	SP_svc	SP_abt	SP_und	SP_mon	SP_irq	SP_fiq
LR	LR_usr			LR_svc	LR_abt	LR_und	LR_mon	LR_irq	LR_fiq
PC	PC								
APSR	CPSR								
			SPSR_hyp	SPSR_svc	SPSR_abt	SPSR_und	SPSR_mon	SPSR_irq	SPSR_fiq
			ELR_hyp						

‡ Part of EL3. Exists only in Secure state, and only when EL3 is using AArch32.
† Part of EL2. Exists only in Non-secure state, and only when EL2 is using AArch32.
Cells with no entry indicate that the User mode register is used.

Figure G1-3 AArch32 general-purpose registers, PC, and Special-purpose registers, showing banking

As described in *PE mode for taking exceptions* on page G1-5713, on taking an exception the PE changes mode, unless it is already in the mode to which it must take the exception. Each mode that the PE might enter in this way has:

- A banked copy of the stack pointer, for example SP_irq and SP_hyp.
- A register that holds a preferred return address for the exception. This is:
  - For the EL2 mode, Hyp mode, the Special-purpose register ELR_hyp.
  - For the other privileged modes to which exceptions can be taken, a banked copy of the link register, for example LR_und and LR_mon.
- A saved copy of PSTATE, made on exception entry, for example SPSR_irq and SPSR_hyp.

In addition, FIQ mode has banked copies of the general-purpose registers R8 to R12.

User mode and System mode share the same general-purpose registers.

User mode, System mode, and Hyp mode share the same LR.

For more information about the application level view of the SP, LR, and PC, and the alternative descriptions of them as R13, R14 and R15, see *The general-purpose registers, and the PC, in AArch32 state* on page E1-3951.

AArch32 Special-purpose registers

In AArch32 state, the Special-purpose registers are:

- The CPSR and its view as the APSR.
- The SPSR, including the banked copies SPSR_abt, SPSR_fiq, SPSR_hyp, SPSR_irq, SPSR_mon, SPSR_svc, and SPSR_und.
- The ELR_hyp.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Pseudocode description of general-purpose register and PC operations

The following pseudocode gives access to the general-purpose registers and the PC. These registers are an array, `shared/functions/registers/_R`, indexed by parameter `n`. This array is common to AArch32 and AArch64 operation and therefore contains 31 64-bit registers. `_PC` is the Program Counter, and its definition is common to AArch32 and AArch64 operation and therefore its size is 64-bit.

`LookUpRIndex()` looks up the index value, `n`, for the specified register number and PE mode, using `RBankSelect()` to evaluates the result.

`_R` accesses the specified general-purpose register in the current PE mode, using `Rmode[]` to access the register, accessing `_R` if necessary. `SP` accesses the stack pointer, `LR` accesses the link register, and `PC` accesses the Program Counter. Each function has a non-assignment form for register reads and an assignment form for register writes, other than `PC`, which has only a non-assignment form.

`BranchTo()` performs a branch to the specified address.

The `_R`, `_PC`, `LR`, `SP`, `LookUpRIndex()`, `RBankSelect()`, `Rmode[]`, and `BranchTo()` functions are defined in Chapter J1 *Armv8 Pseudocode*.

G1.9.3 Saved Program Status Registers (SPSRs)

The Saved Program Status Registers (SPSRs) are used to save PE state on taking exceptions. In AArch32 state, there is an SPSR for every mode that an exception can be taken to, as shown in Figure G1-3 on page G1-5692. For example, the SPSR for Monitor mode is called SPSR_mon.

Note

Exceptions cannot be taken to EL0.

When the PE takes an exception, PE state is saved from PSTATE in the SPSR for the mode the exception is taken to. For example, if the PE takes an exception to Monitor mode, PE state is saved in SPSR_mon. For more information on PSTATE, see *Process state, PSTATE* on page G1-5695.

Note

All PSTATE fields are saved, including those which have no direct read and write access.

Saving the PSTATE fields means the exception handler can:

- On return from the exception, restore the PE state to the values it had immediately before the exception was taken. When the PE returns from an exception, PE state is restored to the state stored in the SPSR of the mode the exception is returning from, if the exception return is made using one of:
  - ERET.
  - `LDM`.
  - The Exception return form of the instruction described in *MOV, MOVS (register)* on page F5-4517.
  - The Exception return form of the instruction described in *SUB, SUBS (immediate)* on page F5-4831.

  For example, on returning from Monitor mode, PE state is restored to the state stored in SPSR_mon. If the exception return is made using the RFE instruction, the PE restores the PE state from an SPSR valued read from memory.
- Examine the value that PSTATE had when the exception was taken, for example to determine the instruction set state and privilege level in which the instruction that caused an Undefined Instruction exception was executed.

The SPSRs are UNKNOWN on reset. Any operation in a Non-secure EL1 or EL0 mode makes SPSR_hyp unknown.

SPSR bits that are defined as RES0 on an exception taken from AArch32 state are ignored on any exception return to AArch32 state.

For more information on SPSR, see *SPSR, Saved Program Status Register* on page G8-6464.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Pseudocode description of SPSR operations

The following pseudocode gives access to the SPSRs.

The `SPSR`[] function accesses the current SPSR and is common to AArch32 and AArch64 operation.

The `SPSRWriteByInstr()` function is used by the MSR (register) and MSR (immediate) instructions to update the current SPSR.

The `SPSR`[] and `SPSRWriteByInstr()` functions are defined in Chapter J1 *Armv8 Pseudocode*.

G1.9.4 ELR_hyp

Hyp mode does not provide its own banked copy of LR. Instead, on taking an exception to Hyp mode, the preferred return address is stored in ELR_hyp, a 32-bit Special-purpose register implemented for this purpose.

ELR_hyp can be accessed explicitly only by executing:

- An `MRS` or `MSR` instruction that targets ELR_hyp, see:
  - *MRS (Banked register)* on page F5-4534.
  - *MSR (Banked register)* on page F5-4538.

The `ERET` instruction uses the value in ELR_hyp as the return address for the exception. For more information, see *ERET* on page F5-4369.

Software execution in any Non-secure EL1 or EL0 mode makes ELR_hyp UNKNOWN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G1.10 Process state, PSTATE

In the Armv8-A architecture, Process state or PSTATE is an abstraction of process state information. All of the instruction sets provide instructions that operate on elements of PSTATE.

PSTATE includes all of the following:

- Fields that are meaningful only in AArch32 state.
- Fields that are meaningful only in AArch64 state.
- Fields that are meaningful in both Execution states.

PSTATE is defined in pseudocode as the `PSTATE` structure, of type `ProcState`. `ProcState` is defined in Chapter J1 *Armv8 Pseudocode*.

The PSTATE fields that are meaningful in AArch32 state are:

The Condition flags

N Negative Condition flag.

Z Zero Condition flag.

C Carry Condition flag.

V Overflow Condition flag.

Process state, PSTATE on page E1-3953 gives more information about these.

The overflow or saturation flag

Q See *Process state, PSTATE* on page E1-3953.

The greater than or equal flags

GE[3:0] See *Process state, PSTATE* on page E1-3953.

The PE state controls

J, T Instruction set state. See *Process state, PSTATE* on page E1-3953. J is RES0. On a reset to AArch32 state, T is set to an IMPLEMENTATION DEFINED value. On taking an exception to:

- A PL1 mode using AArch32, T is set to SCTLR.TE.
- EL2 using AArch32, T is set to HSCTLR.TE.

IT[7:0] IT block state bits. See *Process state, PSTATE* on page E1-3953. On a reset or taking an exception to AArch32 state, these bits are set to 0.

E Endianness of data accesses. See *Process state, PSTATE* on page E1-3953. If an implementation provides both Big-endian and Little-endian support, then:

- On a reset to AArch32 state this bit is set to the IMPLEMENTATION DEFINED reset value of:
  - SCTLR.EE if the highest implemented Exception level is not EL2.
  - HSCTLR.EE if the highest implemented Exception level is EL2.
- On taking an exception to:
  - A PL1 mode using AArch32, this bit is set to SCTLR.EE.
  - EL2 using AArch32, this bit is set to HSCTLR.EE

IL Illegal Execution state bit. See *The Illegal Execution state exception* on page G1-5728. On a reset or taking an exception to AArch32 state, this bit is set to 0.

For information on how the J, T, IT[7:0], E, and IL fields can be accessed, see *Accessing the PE state controls and the Execution state bit* on page G1-5698.

The asynchronous exception mask bits

A SError interrupt mask bit.

I IRQ interrupt mask bit.

F FIQ interrupt mask bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For each bit, the values are:

0 Exception not masked.

1 Exception masked.

On a reset to AArch32 state, these bits are set to 1.

On taking an exception to AArch32 state, one or more of these bits are set to 1.

For more information, see both:

- *Asynchronous exception masking controls* on page G1-5733.
- *PE state on exception entry* on page G1-5716.

The mode bits

M[4:0] Current mode of the PE. Table G1-5 on page G1-5686 lists the permitted values of this field. All other values are reserved. *Illegal changes to PSTATE.M* on page G1-5699 describes the effect of setting M[4:0] to a reserved value.

M[4] is:

M[4], Execution state

The current Execution state:

0 AArch64 state.

1 AArch32 state.

Note

This is consistent with the use of M[4:0] in previous versions of the architecture.

On a reset to AArch32 state, M[4:0] is set to:

- `0b10011`, meaning Supervisor mode, if the highest implemented Exception level is not EL2.
- `0b11010`, meaning Hyp mode, if the highest implemented Exception level is EL2.

On taking an exception to AArch32 state, M[4:0] is set to the target mode for the exception type.

For more information about the PE modes, see:

- *AArch32 state PE mode descriptions* on page G1-5686.
- *PE state on exception entry* on page G1-5716.

Access control bits, from Armv8.1

PAN *Privileged Access Never* (PAN) state bit, see *About the PAN bit* on page G5-5971.

Timing control bits

DIT *Data Independent Timing* (DIT) bit. For more information, see *About the DIT bit* on page E1-3959.

This bit is implemented only when ARMv8.4-DIT is implemented.

On a reset to AArch32 state, this bit is set to 0.

Speculation control bits

SSBS *Speculative Store Bypass Safe* (SSBS) bit. For more information, see *Speculative Store Bypass Safe (SSBS)* on page E2-3990.

This bit is implemented only when ARMv8.0-SSBS is implemented.

On reset to AArch32 state, this bit is set to an IMPLEMENTATION DEFINED value.

G1.10.1 Accessing PSTATE fields

The PSTATE fields can be accessed as described in the following subsections:

- *The Current Program Status Register, CPSR* on page G1-5697.
- *Accessing the PE state controls and the Execution state bit* on page G1-5698.

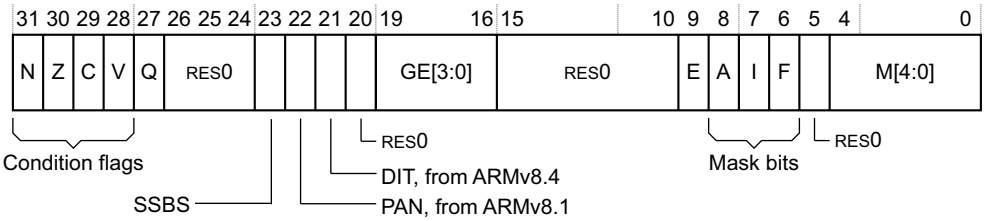
Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *The CPS instruction* on page G1-5698.
- *The SETEND instruction* on page G1-5699.
- *The SETPAN instruction* on page G1-5699.

The Current Program Status Register, CPSR

Some PSTATE fields can be accessed using the Special-purpose *Current Program Status Register* (CPSR). The CPSR can be directly read using the MRS instruction, and directly written using the MSR (register) and MSR (immediate) instructions.

The CPSR bit assignments are:



N, Z, C, V, bits [31:28]

The PSTATE Condition flags.

Q, bit [27] The PSTATE overflow or saturation flag.

Bits[26:23, 20, 15:10, 5]

Reserved, RES0.

SSBS, bit [23] Speculative Store Bypass Safe (SSBS) bit, see *Access permissions for instruction execution* on page G5-5972.

Bit[22] In Armv8.0, Reserved, RES0.

In Armv8.1, *Privileged Access Never* (PAN) state bit, see *About the PAN bit* on page G5-5971.

DIT, bit [21] Shows the value of CPSR.DIT immediately before the exception was taken.

GE[3:0], bits [19:16]

The PSTATE greater than or equal flags.

E, bit [9] The PSTATE endianness bit.

A, I, F, bits [8:6]

The PSTATE asynchronous exception mask bits.

M[4:0], bits [4:0]

The PSTATE mode bits.

The other PSTATE fields cannot be accessed by using the CPSR. For information on how to access them, see *Accessing the PE state controls and the Execution state bit* on page G1-5698.

The application level alias for the CPSR is the APSR. The APSR is a subset of the CPSR. See *The Application Program Status Register, APSR* on page E1-3955.

Writes to the CPSR have side-effects on various aspects of PE operation. All of these side-effects, except side-effects on memory accesses associated with fetching instructions, are synchronous to the CPSR write. This means that they are guaranteed:

- Not to be visible to earlier instructions in the execution stream.
- To be visible to later instructions in the execution stream.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The privilege level and address space of memory accesses associated with fetching instructions depend on the current Exception level and Security state. Writes to PSTATE.M can change one or both of the Exception level and Security state. The effect, on memory accesses associated with fetching instructions, of a change of Exception level or Security state is:

- Synchronous to the change of Exception level or Security state, if that change is caused by an exception entry or exception return.
- Guaranteed not to be visible to any memory access caused by fetching an earlier instruction in the execution stream.
- Guaranteed to be visible to any memory access caused by fetching any instruction after the next *Context synchronization event* in the execution stream.
- Might or might not affect memory accesses caused by fetching instructions between the mode change instruction and the point where the mode change is guaranteed to be visible.

See *Exception return to an Exception level using AArch32* on page G1-5725 for the definition of exception return instructions.

Accessing the PE state controls and the Execution state bit

The PE state controls are the PSTATE.{IL, IT[7:0], J, E, T} fields. Software can read or write these in an SPSR.

In the CPSR:

- The PE state controls, other than PSTATE.E, are RAZ when read by an MRS instruction.
- Writes to the PE state controls, other than PSTATE.E, by MSR (register) or MSR (immediate), are ignored in all modes.

Instructions other than MRS, MSR (register), or MSR (immediate) that access the PE state controls can read and write them in any mode.

Unlike the other PSTATE PE state controls, PSTATE.E can be read by an MRS instruction and might be written by MSR (register) or MSR (immediate). However, Arm deprecates PSTATE.E having a different value from the equivalent System register EE bit, see *Mixed-endian support* on page G4-5888.

Note

To determine the current endianness, software can use an `LDR` instruction to load a word from memory with a known value that differs if the endianness is reversed. For example, using an `LDR` instruction to load a word whose four bytes are `0x01`, `0x00`, `0x00`, and `0x00` in ascending order of memory address loads the destination register with:

- `0x00000001` if the current endianness is little-endian.
- `0x01000000` if the current endianness is big-endian.

The PSTATE.M[4] bit is the Execution state bit. When read by an MRS instruction in AArch32 state, this bit always reads as 1. When written by an MSR (register) instruction or MSR (immediate) instruction, writing a value other than 1 is an illegal change to the PSTATE.M field. See *Illegal changes to PSTATE.M* on page G1-5699.

The `CPS` instruction

The A32 and T32 instruction sets both include an instruction to manipulate PSTATE.{A, I, F} and PSTATE.M:

`CPSIE <iflags> {, #<mode>}`
: Sets the specified PSTATE. {A, I, F} exception masks to 0, enabling the exception, and optionally changes to the specified mode.

`CPSID <iflags> {, #<mode>}`
: Sets the specified PSTATE.{A, I, F} exception masks to 1, disabling the exception, and optionally changes to the specified mode.

`CPS #<mode>`
: Changes to the specified mode without affecting the PSTATE.{A, I, F} exception masks.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The CPS instruction is unconditional. For more information, see *CPS, CPSID, CPSIE* on page F5-4334.

The SETEND instruction

The A32 and T32 instruction sets both include an instruction to manipulate PSTATE.E:

SETEND BE Sets PSTATE.E to 1, for big-endian operation.

SETEND LE Sets PSTATE.E to 0, for little-endian operation.

The SETEND instruction is unconditional. For more information, see *SETEND* on page F5-4676. Arm deprecates use of the SETEND instruction.

The SETPAN instruction

ARMv8.1-PAN adds the SETPAN instruction to the A32 and T32 instruction sets, to manipulate PSTATE.PAN:

SETPAN #0 Sets PSTATE.PAN to 0, disabling Privileged access-never operation.

SETPAN #1 Sets PSTATE.PAN to 1, enabling Privileged access-never operation.

The SETPAN instruction is unconditional.

- *SETPAN* on page F5-4677.
- *About the PAN bit* on page G5-5971.

G1.10.2 The Saved Program Status Registers (SPSRs)

On taking an exception, PSTATE is preserved in the SPSR of the mode to which the exception is taken. The SPSRs are described in *Saved Program Status Registers (SPSRs)* on page G1-5693.

G1.10.3 Illegal changes to PSTATE.M

In AArch32 PE modes other than User mode, MSR and CPS instructions can explicitly change PSTATE.M. The following changes to PSTATE.M by MSR or CPS instructions are illegal:

- A change to an encoding that Table G1-5 on page G1-5686 does not show.
- A change to a mode that is not implemented.
- A change to a mode that is not accessible from the context the MRS or CPS instruction is executed in, as follows:
  - A change to a mode that would cause entry to a higher Exception level.
  - When executing in Non-secure state, a change to Monitor mode.
  - When executing in Secure EL1, a change to Monitor mode when EL3 is using AArch64.
  - A change to Hyp mode from any other mode.
  - A change from Hyp mode to any other mode.
  - When the value of HCR.TGE is 1, attempting to change from Monitor mode to a Non-secure PL1 mode, see *Trapping of general exceptions to Hyp mode* on page K1-7852.

On executing an instruction that attempts an illegal change to PSTATE.M:

- PSTATE.M is unchanged, and the current mode remains unchanged.
- PSTATE.IL is set to 1.
- All other PSTATE fields are written to as normal.

Note

For the PSTATE fields that MSR and CPS instructions update, see the instruction descriptions:

- *MSR (register)* on page F5-4544.
- *MSR (immediate)* on page F5-4542.
- *CPS, CPSID, CPSIE* on page F5-4334.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When the value of PSTATE.IL is 1, any attempt to execute any instruction results in an Illegal Execution state exception. See *The Illegal Execution state exception* on page G1-5728.

Note

- The PE ignores writes to PSTATE.M when executing at PL0.
- In Armv7, an instruction that attempts to make an illegal change to PSTATE.M is UNPREDICTABLE.

G1.10.4 Pseudocode description of PSTATE operations

The `CPSRWriteByInstr()` function is used by the MSR (register) and MSR (immediate) instructions to update PSTATE.

The `SetPSTATEFromPSR()` function updates PSTATE from a CPSR or SPSR.

Chapter J1 *Armv8 Pseudocode* defines these functions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G1.11 Instruction set states

The instruction set states are described in Chapter E2 *The AArch32 Application Level Memory Model* and application level operations on them are described there. This section supplies more information about how they interact with system level functionality, in the sections:

- *Exceptions and instruction set state*.
- *Unimplemented instruction sets*.

G1.11.1 Exceptions and instruction set state

If an exception is taken to an EL1 mode, the SCTLR.TE bit for the Security state the exception is taken to determines the instruction set state that handles the exception, and if necessary, the PE changes to this instruction set state on exception entry.

If the exception is taken to Hyp mode, the HSCTLR.TE bit determines the instruction set state that handles the exception, and if necessary, the PE changes to this instruction set state on exception entry.

On coming out of reset, if the highest implemented Exception level is using AArch32:

- If the highest implemented Exception level is EL2, the PE starts execution in Hyp mode, in the instruction set state determined by the reset value of HSCTLR.TE.
- Otherwise, the PE starts execution in Supervisor mode, in the instruction set state determined by the reset value of SCTLR.TE. If the implementation includes EL3, this execution is in Secure Supervisor mode.

For more information about exception entry, see *Overview of exception entry* on page G1-5710.

G1.11.2 Unimplemented instruction sets

The PSTATE.T bit defines the current instruction set state, see *Process state, PSTATE* on page E1-3953.

In the Armv8 architecture, there is no support for the hardware acceleration of Java bytecodes, and the Jazelle Instruction set state is obsolete. Every AArch32 implementation must support the Trivial Jazelle implementation described in *Trivial implementation of the Jazelle extension*.

Note

In previous versions of the Arm architecture, the PSTATE.{J, T} bits determined the Instruction set state. In Armv8, PSTATE.J is RES0.

Trivial implementation of the Jazelle extension

Armv8 requires that the implementation of AArch32 state includes the trivial Jazelle implementation.

In a trivial implementation of the Jazelle extension:

- At EL1, EL2, or EL3, if the Exception level is using AArch32:
  - The JMCR and JOSCR are RAZ/WI.
  - The JIDR is a RAZ read-only register.
- At EL0 when EL0 is using AArch32:
  - It is IMPLEMENTATION DEFINED whether the JMCR and JOSCR are RAZ/WI or UNDEFINED.
  - It is IMPLEMENTATION DEFINED whether JIDR is RAZ or UNDEFINED.
- The `BXJ` instruction behaves identically to the `BX` instruction in all circumstances.

  Note

  This is consistent with the JMCR.JE bit being RAZ, and means that the A32 and T32 instruction sets do not provide any mechanism for attempting to enter Jazelle state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Jazelle state, as defined in previous versions of the Arm architecture, is an unimplemented instruction set state.

These requirements ensure that operating systems that support an EJVM execute correctly.

A trivial implementation is not required to extend the PC to 32 bits, that is, it can implement PC[0] as RAZ/WI.

Note

This is because the only way that PC[0] is visible in A32 or T32 state is as a result of an exception occurring during Jazelle state execution, and Jazelle state execution cannot occur on a trivial implementation.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G1.12 Handling exceptions that are taken to an Exception level using AArch32

An exception causes the PE to suspend program execution to handle an event, such as an externally generated interrupt or an attempt to execute an undefined instruction. Exceptions can be generated by internal and external sources.

Normally, when an exception is taken the PE state is preserved immediately, before handling the exception. This means that, when the event has been handled, the original state can be restored and program execution resumed from the point where the exception was taken.

More than one exception might be generated at the same time, and a new exception can be generated while the PE is handling an exception.

The following sections describe exception handling:

- *Exception vectors and the exception base address*.
- *Exception prioritization for exceptions taken to AArch32 state* on page G1-5706.
- *Overview of exception entry* on page G1-5710.
- *PE mode for taking exceptions* on page G1-5713.
- *PE state on exception entry* on page G1-5716.
- *Routing exceptions from Non-secure EL0 to EL2* on page G1-5718.
- *Routing debug exceptions to EL2 using AArch32* on page G1-5720.

See also:

- *Routing of aborts taken to AArch32 state* on page G1-5722.
- *Exception return to an Exception level using AArch32* on page G1-5725.
- *Asynchronous exception behavior for exceptions taken from AArch32 state* on page G1-5730.
- *AArch32 state exception descriptions* on page G1-5738.

G1.12.1 Exception vectors and the exception base address

When an exception is taken, PE execution is forced to an address that corresponds to the type of exception. This address is called the *exception vector* for that exception. The vectors for the different types of exception form a *vector table*.

Note

There are significant differences in the sets of exception vectors for exceptions taken to an Exception level that is using AArch32 and for exceptions taken to an Exception level that is using AArch64. This part of this manual describes only how exceptions are taken to an Exception level that is using AArch32.

When an exception is taken to an Exception level that is using AArch64, then the exception is taken as described in Chapter D1 *The AArch64 System Level Programmers' Model* using the exception vectors described in *Exception vectors* on page D1-2341.

AArch32 state defines exception vector tables for exceptions taken to EL2 and EL3 when those Exception levels are using AArch32. Those vector tables are not used when the corresponding Exception levels are using AArch64.

A set of exception vectors for an Exception level that is using AArch32 comprises eight consecutive word-aligned memory addresses, starting at an *exception base address*. These eight vectors form an AArch32 *vector table*.

The number of possible exception base addresses, and therefore the number of vector tables, depends on the implemented Exception levels, as follows:

Implementation that does not include EL3

Any implementation that does not include EL3 must include the following AArch32 vector table if EL1 can use AArch32:

- An exception table for exceptions taken to EL1 modes other than System mode. This is the EL1 vector table, and is in the address space of the PL1&0 translation regime.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

Exceptions cannot be taken to System mode.

For this vector table:

— When SCTLR.V == 0, the VBAR holds the exception base address.

— When SCTLR.V == 1, the exception base address is `0xFFFF0000`.

Implementation that includes EL2

Any implementation that includes EL2 must include the following additional AArch32 vector table if EL2 can use AArch32:

- An exception table for exceptions taken to Hyp mode. This is the Hyp vector table, and is in the address space of the Non-secure PL2 translation regime.

  For this vector table, HVBAR holds the exception base address.

Implementation that includes EL3

Any implementation that includes EL3 must include the following AArch32 vector tables:

- If EL3 can use AArch32, a vector table for exceptions taken to Secure Monitor mode. This is the Monitor vector table, and is in the address space of the Secure PL1&0 translation regime.

  For this vector table, MVBAR holds the exception base address.
- If Secure EL1 can use AArch32, a vector table for exceptions taken to Secure privileged modes other than Monitor mode and System mode. This is the Secure vector table, and is in the address space of the Secure PL1&0 translation regime.

  — When the Secure instance of SCTLR.V == 0, the Secure instance of VBAR holds the exception base address.

  — When the Secure instance of SCTLR.V == 1, the exception base address is `0xFFFF0000`.
- If Non-secure EL1 can use AArch32, a vector table for exceptions taken to Non-secure PL1 modes. This is the Non-secure vector table, and is in the address space of the Non-secure PL1&0 translation regime.

  — When the Non-secure instance of SCTLR.V == 0, the Non-secure instance of VBAR holds the exception base address.

  — When the Non-secure instance of SCTLR.V == 1, the exception base address is `0xFFFF0000`.

The following subsections give more information:

- *The vector tables and exception offsets*.
- *Pseudocode determination of the exception base address* on page G1-5706.

The vector tables and exception offsets

Table G1-6 on page G1-5705 defines the AArch32 vector table entries. In this table:

- The *Hyp* column defines the vector table entries for exceptions taken to Hyp mode.
- The *Monitor* column defines the vector table entries for exceptions taken to Monitor mode.
- The *Secure* and *Non-secure* columns define the Secure and Non-secure vector table entries, that are used for exceptions taken to modes other than Monitor mode, Hyp mode, System mode, and User mode. Table G1-7 on page G1-5705 shows the mode to which each of these exceptions is taken. Each of these modes is described as the *default* mode for taking the corresponding exception.

  Note

  Exceptions cannot be taken to System mode or User mode.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For more information about determining the mode to which an exception is taken, see *PE mode for taking exceptions* on page G1-5713.

When EL2 is using AArch32, it provides a number of additional exceptions, some of which are not shown explicitly in the vector tables. For more information, see *Offsets of AArch32 exceptions provided by EL2* on page G1-5706.

Table G1-6 The AArch32 vector tables

Offset	Vector tables			
	Hyp[a]	Monitor[b]	Secure[c]	Non-secure[c]
0x00	Not used	Not used	Not used[d]	Not used
0x04	Undefined Instruction, from Hyp mode	Monitor Trap	Undefined Instruction	Undefined Instruction
0x08	Hypervisor Call, from Hyp mode	Secure Monitor Call	Supervisor Call	Supervisor Call
0x0C	Prefetch Abort, from Hyp mode	Prefetch Abort	Prefetch Abort	Prefetch Abort
0x10	Data Abort, from Hyp mode	Data Abort	Data Abort	Data Abort
0x14	Hyp Trap, or Hyp mode entry[e]	Not used	Not used	Not used
0x18	IRQ interrupt	IRQ interrupt	IRQ interrupt	IRQ interrupt
0x1C	FIQ interrupt	FIQ interrupt	FIQ interrupt	FIQ interrupt

a. Non-secure state only. Implemented only if the implementation includes EL2 and EL2 can use AArch32.
b. Secure state only. Implemented only if the implementation includes EL3 and EL3 can use AArch32.
c. If the implementation does not include EL3 then there is a single vector table for exceptions taken to EL1 when EL1 is using AArch32. That table holds the vectors shown in the Secure column of this table
d. In previous versions of the architecture, this entry has been used for the Reset vector, meaning the address at which execution starts on coming out of reset. In Armv8, the AArch32 Reset vector is IMPLEMENTATION DEFINED. An implementation might use this vector table entry to hold the Reset vector.
e. See *Use of offset 0x14 in the Hyp vector table* on page G1-5706.

Table G1-7 Modes for taking the exceptions shown in the Secure or Non-secure vector table

Exception	Mode taken to
Undefined Instruction	Undefined
Supervisor Call	Supervisor
Prefetch Abort	Abort
Data Abort	Abort
IRQ interrupt	IRQ
FIQ interrupt	FIQ

For more information about use of the vector tables, see *Overview of exception entry* on page G1-5710.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Offsets of AArch32 exceptions provided by EL2

EL2 provides the following exceptions. When EL2 is using AArch32, these exceptions are taken to Hyp mode, and the PE enters the handlers for these exceptions using the following vector table entries shown in Table G1-6 on page G1-5705:

Hypervisor Call
: If taken from Hyp mode, shown explicitly in the Hyp mode vector table. Otherwise, see *Use of offset 0x14 in the Hyp vector table*.

Hyp Trap Shown explicitly in the Hyp mode vector table.

Virtual Abort Entered through the Data Abort vector in the Non-secure vector table.

Virtual IRQ Entered through the IRQ vector in the Non-secure vector table.

Virtual FIQ Entered through the FIQ vector in the Non-secure vector table.

Note

Virtual exceptions when an implementation includes EL2 on page G1-5730 gives more information about the virtual exceptions.

Use of offset `0x14` in the Hyp vector table

The vector at offset `0x14` in the Hyp vector table is used for all exceptions that cause entry to Hyp mode from Non-secure EL0 and EL1, except for IRQ and FIQ exceptions.

Note

Virtual exceptions are never taken to Hyp mode.

Pseudocode determination of the exception base address

For an exception taken to a PL1 mode, the `ExcVectorBase()` function determines the exception base address.

The `ExcVectorBase()` function is defined in Chapter J1 *Armv8 Pseudocode*.

Note

The PL1 modes to which exceptions can be taken are Supervisor mode, Undefined mode, Abort mode, IRQ mode, and FIQ mode. In Non-secure state, and in Secure state when EL3 is using AArch64, these are EL1 modes. However, in Secure state when EL3 is using AArch32, these are EL3 modes. For more information see *Security state, Exception levels, and AArch32 execution privilege* on page G1-5682.

G1.12.2 Exception prioritization for exceptions taken to AArch32 state

The following sections describe the Armv8 requirements for the prioritization of synchronous exceptions, and the limits on when asynchronous exceptions can be taken:

- *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-5707.
- *Architectural requirements for taking asynchronous exceptions* on page G1-5709.

See also:

- *AArch32 state prioritization of synchronous aborts from a single stage of address translation* on page G5-6021, for information about:
  - The prioritization of aborts on a single memory access in a VMSA implementation.
  - The prioritization of exceptions generated during address translation.
- *Debug state entry and debug event prioritization* on page H2-6929 for information about the relative prioritization of exceptions and the debug events that cause entry to Debug state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Synchronous exception prioritization for exceptions taken to AArch32 state

In principle, any single instruction can generate a number of different synchronous exceptions, between the fetching of the instruction, its decode, and eventual execution. This section describes the prioritization of such exceptions when they are taken to an Exception level that is using AArch32.

Note

- An exception that is taken to an Exception level that is using AArch32 must have been taken from an Exception level that is using AArch32.
- The priority numbering in this list correlates with the equivalent AArch64 list in *Synchronous exception prioritization for exceptions taken to AArch64 state* on page D1-2355.

For an exception that is taken to an Exception level that is using AArch32, exceptions are prioritized as follows, where 1 is the highest priority.

1-5 These priority numbers are used by AArch64 exceptions or debug events.

6 PC alignment fault exceptions. A PC alignment fault exception can only be taken to an Exception level that is using AArch32 as a result of:

- The CONSTRAINED UNPREDICTABLE handling of a branch to an unaligned address, see *Branching to an unaligned PC* on page K1-7832.
- Exiting from Debug state to AArch32 specifying an unaligned PC value, see *Exiting Debug state* on page H2-6963.

A PC alignment fault exception that is taken to an Exception level that is using AArch32 is reported as a Prefetch Abort exception, see *Prefetch Abort exception reporting a PC alignment fault exception* on page G1-5746.

7 Prefetch Abort exceptions. See *Prefetch Abort exception* on page G1-5745 and *AArch32 state prioritization of synchronous aborts from a single stage of address translation* on page G5-6021.

8 Breakpoint exceptions or Address Matching Vector Catch exceptions. See:

- *Breakpoint exceptions* on page G2-5830.
- *Vector Catch exceptions* on page G2-5869.

Note

An Exception Trapping Vector Catch exception is generated on exception entry for an exception that has been prioritized as described in this section. This means that it does not have its own entry in this list.

9 Illegal Execution state exceptions. See *The Illegal Execution state exception* on page G1-5728.

10 Software Breakpoint Exceptions caused by the execution of a BKPT Exception generating instruction.

11 This priority number is used by AArch64 exceptions.

12 Exceptions taken from EL1 to EL2 because of one of the following configuration settings:

- HSTR.T*n*.
- HCR.TIDCP.

13 Undefined Instruction exceptions that occur as a result of one or more of the following:

- An attempt to execute an unallocated instruction encoding, including an encoding for an instruction that is not implemented in the PE implementation.
- An attempt to execute an instruction that is defined never to be accessible at the current Exception level regardless of any enables or traps.
- Debug state execution of an instruction encoding that is unallocated in Debug state.
- Non-debug state execution of an instruction encoding that is unallocated in Non-debug state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Execution of an `HVC` instruction, when `HVC` instructions are disabled by SCR.HCE or HCR.HCD.
- Execution of an `HLT` instruction when `HLT` instructions are disabled by EDSCR.HDE or when halting is prohibited.
- In Debug state:
  - — Execution of a `DCPS1` instruction in Non-secure EL0 when HCR.TGE is 1.
  - — Execution of a `DCPS2` instruction in EL1 or EL0 when SCR.NS is 0 or when EL2 is disabled or not implemented in the current Security state.
  - — Execution of a `DCPS3` instruction when EDSCR.SDD is 1 or when EL3 is not implemented.
  - — When the value of EDSCR.SDD is 1, execution in EL2, EL1, or EL0 of an instruction that is trapped to EL3.
- Execution of an instruction that is UNDEFINED as a result of any of:
  - — Being in an IT block when SCTLR.ITD is 1, or when HSCTLR.ITD is 1.
  - — Executing a `SETEND` instruction when SCTLR.SED is 1, or when HSCTLR.SED is 1.
  - — Executing a CP15DMB, CP15DSB, or CP15ISB barrier instruction when SCTLR.CP15BEN is 0, or when HSCTLR.CP15BEN is 0.

  See *Disabling or enabling PL0 and PL1 use of AArch32 deprecated functionality* on page G1-5781 and *Disabling or enabling EL2 use of AArch32 deprecated functionality* on page G1-5790.
- Execution of an instruction that is UNDEFINED because at least one of FPSCR.{Stride, Len} is nonzero, when programming these bits to nonzero values is supported. See *Floating-point exceptions and exception traps* on page G1-5776.

14 Exceptions taken to EL1, or taken to EL2 because the value of HCR.TGE is 1, that are generated because of configurable access to instructions, and that are not covered by any of priorities 6-13.

15 Exceptions taken from EL0 to EL2 because of one of the following configuration settings:
- HSTR.T*n*.
- HCR.TIDCP.

16 Exceptions taken to EL2 because of configuration settings in the HCPTR.

17 Exceptions taken to EL2 because of one of the following configuration settings:
- Any setting in HCR, other than the TIDCP bit.
- Any setting in CNTHCTL.
- Any setting in HDCR.
- If EL1 is using AArch64 state, any of the fine-grained traps in HAFGRTR_EL2, HDFGRTR_EL2, HDFGWTR_EL2, HFGITR_EL2, HFGRTR_EL2, HFGWTR_EL2.

18 Exceptions taken to EL2 because of configurable access to instructions, and that are not covered by any of priorities 6-17.

19 Exceptions caused by the `SMC` instruction being UNDEFINED because the value of SCR.SCD is 1.

20 Exceptions caused by the execution of an Exception generating instruction, `SVC`, `HVC`, or `SMC`.

21-22 These priority numbers are used by AArch64 exceptions.

23 Exceptions taken to EL3 from EL0, EL1 or EL2 because of configuration settings in the SDCR.

24 Exceptions taken to EL3 because of configurable access to instructions, and that are not covered by any of priorities 6-23.

25 This priority number is used by AArch64 exceptions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

26 Trapped floating-point exceptions, if supported. See *Floating-point exceptions and exception traps* on page G1-5776.

27-28 These priority numbers are used by AArch64 exceptions and debug events.

29 Data Abort exceptions other than a Data Abort exception generated by a synchronous External abort that was not generated by a translation table walk. That is, any Data Abort exception that is not covered by item 31. See *Data Abort exception* on page G1-5749 and *AArch32 state prioritization of synchronous aborts from a single stage of address translation* on page G5-6021. It is IMPLEMENTATION DEFINED whether synchronous External aborts are prioritized here or as item 31.

30 Watchpoint exceptions. See *Watchpoint exceptions* on page G2-5855.

31 Data Abort exception generated by a synchronous External abort that was not generated by a translation table walk, see *External aborts* on page G4-5914. It is IMPLEMENTATION DEFINED whether synchronous External aborts are prioritized here or as item 29.

For items 29-31, if an instruction results in more than one single-copy atomic memory access, the prioritization between synchronous exceptions generated on each of those different memory accesses is not defined by the architecture.

Note

Exceptions generated by a translation table walk are reported and prioritized as either a Prefetch Abort exception, priority 7 in this list, or a Data Abort exception, priority 29 in this list. See also *AArch32 state prioritization of synchronous aborts from a single stage of address translation* on page G5-6021.

Architectural requirements for taking asynchronous exceptions

The Arm architecture does not define when asynchronous exceptions are taken. The prioritization of asynchronous exceptions, including virtual asynchronous exceptions, is IMPLEMENTATION DEFINED.

An asynchronous exception that is pending before a *Context synchronization event* in the following list, is taken before the first instruction after the context synchronizing event, provided that the pending asynchronous event is not masked:

- Execution of an `ISB` instruction that does not fail its Condition code check.
- Exception entry.
- Exception return.
- Exit from Debug state.

Note

- If the first instruction after the context synchronizing event generates a synchronous exception, then the architecture does not define the order in which that synchronous exception and the asynchronous exception are taken.
- The ISR identifies any pending asynchronous exceptions.
- Interrupts are masked when the PE is in Debug state, and therefore this list of context synchronizing events does not include the `DCPS` and `DRPS` instructions.

In the absence of a specific requirement to take an asynchronous exception, the only requirement of the architecture is that an unmasked asynchronous exception is taken in finite time.

Note

The taking of an unmasked asynchronous exception in finite time must occur with all code sequences, including with a sequence that consists of unconditional loops.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If an unmasked interrupt was pending but is changed to not pending before it is taken, then the architecture permits the interrupt to be taken, but does not require this to happen. If the interrupt is taken, then it must be taken before the first *Context synchronization event* after the interrupt was changed to not pending.

PSTATE includes a mask bit for each type of asynchronous exception. Setting one of these bits to 1 can prevent the corresponding asynchronous exception from being taken, although when the PE is in Non-secure state other controls can modify the effect of these bits. For more information, see *Asynchronous exception behavior for exceptions taken from AArch32 state* on page G1-5730.

Taking an exception sets an exception-dependent subset of these mask bits.

Note

In some contexts, the PSTATE.{A, I, F} bits mask the taking of asynchronous exceptions. The way these are set on exception entry, described in *PSTATE.{A, I, F, M} values on exception entry* on page G1-5717, can prevent an exception handler being interrupted by an asynchronous exception.

G1.12.3 Overview of exception entry

There are some significant differences between the handling of exceptions taken to Hyp mode and exceptions taken to other modes. Because Hyp mode is the EL2 mode, this means that the following descriptions sometimes distinguish between the *EL2 mode* and the *non-EL2 modes*.

On taking an exception to an Exception level that is using AArch32:

1. The hardware determines the mode to which the exception must be taken, see *PE mode for taking exceptions* on page G1-5713.

2. A link value, indicating the *preferred return address* for the exception, is saved. This is a possible return address for the exception handler, and depends on:
   - The exception type.
   - Whether the exception is taken to the EL2 mode or to a non-EL2 mode.
   - For some exceptions taken to non-EL2 modes, the instruction set state when the exception was taken.

   Where the link value is saved depends on whether the exception is taken to the EL2 mode.

   For more information, see *Link values saved on exception entry* on page G1-5711.

3. The value of PSTATE is saved in the SPSR for the mode to which the exception must be taken. The value saved in SPSR.IT[7:0] is always correct for the preferred return address.

4. In an implementation that includes EL3, when EL3 is using AArch32:
   - If the exception is taken from Monitor mode, SCR.NS is cleared to 0.
   - Otherwise, taking the exception leaves SCR.NS unchanged.

   When EL3 is using AArch64, Monitor mode is not available.

5. PSTATE is updated with new context information for the exception handler. This includes:
   - Setting PSTATE.M to the PE mode to which the exception is taken.
   - Setting the appropriate PSTATE mask bits. This can disable the corresponding exceptions, preventing uncontrolled nesting of exception handlers.
   - Setting the instruction set state to the state required for exception entry.
   - Setting the endianness to the required value for exception entry.
   - Clearing the PSTATE.IT[7:0] bits to 0.

   For more information, see *PE state on exception entry* on page G1-5716.

6. The appropriate exception vector is loaded into the PC, see *Exception vectors and the exception base address* on page G1-5703.

7. Execution continues from the address held in the PC.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For an exception taken to a non-EL2 mode, on exception entry, the exception handler can use the SRS instruction to store the return state onto the stack of any mode at the same Exception level and in the same Security state, and can use the CPS instruction to change mode. For more information about the instructions, see *SRS, SRSDA, SRSDB, SRSIA, SRSIB* on page F5-4729 and *CPS, CPSID, CPSIE* on page F5-4334.

Later sections of this chapter describe each of the possible exceptions, and each of these descriptions includes a pseudocode description of the PE state changes on taking that exception. Table G1-8 gives an index to these descriptions:

Table G1-8 Pseudocode descriptions of exception entry for exceptions taken to AArch32 state

Exception	Description of exception entry
Reset	*Pseudocode descriptions of reset* on page G1-5763
Undefined Instruction	*Pseudocode description of taking the Undefined Instruction exception* on page G1-5740
Hyp Trap	*Pseudocode description of taking the Hyp Trap exception* on page G1-5742
Monitor Trap	*Pseudocode description of taking the Monitor Trap exception* on page G1-5741
Supervisor Call	*Pseudocode description of taking the Supervisor Call exception* on page G1-5743
Secure Monitor Call	*Pseudocode description of taking the Secure Monitor Call exception* on page G1-5744
Hypervisor Call	*Pseudocode description of taking the Hypervisor Call exception* on page G1-5745
Prefetch Abort	*Pseudocode description of taking the Prefetch Abort exception* on page G1-5749
Data Abort	*Pseudocode description of taking the Data Abort exception* on page G1-5752
Virtual Abort	*Pseudocode description of taking the Virtual SError interrupt exception* on page G1-5754
IRQ	*Pseudocode description of taking the physical IRQ exception* on page G1-5755
Virtual IRQ	*Pseudocode description of taking the Virtual IRQ exception* on page G1-5756
FIQ	*Pseudocode description of taking the FIQ exception* on page G1-5758
Virtual FIQ	*Pseudocode description of taking the Virtual FIQ exception* on page G1-5758

The following sections give more information about the PE state changes, for different architecture implementations. However, you must refer to the pseudocode for a full description of the state changes:

- *PE mode for taking exceptions* on page G1-5713.
- *PE state on exception entry* on page G1-5716.

Link values saved on exception entry

On exception entry, a link value for use on return from the exception, is saved. This link value is based on the *preferred return address for the exception*, as shown in Table G1-9:

Table G1-9 Exception return addresses for exceptions taken to AArch32 state

Exception	Preferred return address	Taken to a mode at
Undefined Instruction	Address of the UNDEFINED instruction	Non-EL2[a], or EL2[c]
Hyp Trap	Address of the trapped instruction	EL2 only[c]
Monitor Trap	Address of the trapped instruction	EL3 only
Supervisor Call	Address of the instruction after the SVC instruction	Non-EL2[a] or EL2[c]

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G1-9 Exception return addresses for exceptions taken to AArch32 state (continued)

Exception	Preferred return address	Taken to a mode at
Secure Monitor Call	Address of the instruction after the `SMC` instruction	EL3[b], and only in Secure state
Hypervisor Call	Address of the instruction after the `HVC` instruction	EL2 only[c]
Prefetch Abort	Address of aborted instruction fetch	Non-EL2[a] or EL2[c]
Data Abort	Address of instruction that generated the abort	Non-EL2[a] or EL2[c]
Virtual Abort	Address of next instruction to execute	EL1, and only in Non-secure state
IRQ or FIQ	Address of next instruction to execute	Non-EL2[a] or EL2[c]
Virtual IRQ or Virtual FIQ	Address of next instruction to execute	EL1, and only in Non-secure state

a. EL1 if the exception is taken to a Non-secure mode, or is taken to a Secure mode when EL3 is using AArch64. EL3 if the exception is taken to a Secure mode when EL3 is using AArch64.

b. A Secure Monitor Call exception is taken to EL3, and therefore is taken to AArch32 state only if EL3 is using AArch32, in which case it is taken to Monitor mode.

c. EL2 is implemented only in Non-secure state when using AArch32 state. Therefore, an exception can be taken to EL2 mode only if it is taken from Non-secure state when using AArch32 state.

Note

- Although Reset is described as an exception, it differs significantly from other exceptions. The architecture has no concept of a return from a Reset and therefore it is not listed in this section.
- For each exception, the preferred return address is not affected by the Exception level from which the exception was taken.

The link value saved, and where it is saved, depend on whether the exception is taken to a non-EL2 mode, or to an EL2 mode, as follows:

Exception taken to a non-EL2 mode

The link value is saved in the LR for the mode to which the exception is taken.

The saved link value is the preferred return address for the exception, plus an offset that depends on the instruction set state when the exception was taken, as Table G1-10 shows:

Table G1-10 Offsets applied to Link value for exceptions taken to non-EL2 modes

Exception	Offset, for PE state of:	
	A32	T32
Undefined Instruction	+4	+2
Monitor Trap	+4	+2
Supervisor Call	None	None
Secure Monitor Call	None	None
Prefetch Abort	+4	+4
Data Abort	+8	+8

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G1-10 Offsets applied to Link value for exceptions taken to non-EL2 modes (continued)

Exception	Offset, for PE state of:	
	A32	T32
Virtual Abort	+8	+8
IRQ or FIQ	+4	+4
Virtual IRQ or Virtual FIQ	+4	+4

Exception taken to an EL2 mode

The link value is saved in the ELR_hyp Special-purpose register.

The saved link value is the preferred return address for the exception, as shown in Table G1-9 on page G1-5711, with no offset.

G1.12.4 PE mode for taking exceptions

The following principles determine the Exception level to which an exception is taken, and if that Exception level is using AArch32, the PE mode to which the exception is taken:

- An exception cannot be taken to the EL0 mode.
- An exception is taken either:
  - To the Exception level at which the PE was executing when it took the exception.
  - To a higher Exception level.

  This means that, in Secure state:
  - When EL3 is using AArch32, an exception is always taken to an EL3 mode.
  - When EL3 is using AArch64, an exception that is taken to AArch32 state is taken to an EL1 mode.
- Configuration options and other features provided by EL2 and EL3 can determine the mode to which some exceptions are taken, as follows:

  In an implementation that does not include EL2 or EL3

  An exception is always taken to the default mode for that exception.

  In an implementation that includes EL3

  A Secure Monitor Call exception is always taken to EL3. This means:
  - If EL3 is using AArch32 the exception is taken to Secure Monitor mode.
  - If EL3 is using AArch64, then executing the instruction generates an exception that is taken to EL3, see *Execution of an SMC instruction from a privileged Exception level that is using AArch32* on page G1-5715.

  IRQ, FIQ, and External abort exceptions can be configured to be taken to EL3. Therefore, if EL3 is using AArch32 the exceptions are taken to Secure Monitor mode.

  When EL3 is using AArch32, a Monitor Trap exception is taken to Secure Monitor mode.

  Any exception taken from Secure state that is not taken to Secure Monitor mode is taken to Secure state in the default mode for that exception. As described in *Security state, Exception levels, and AArch32 execution privilege* on page G1-5682, this means it is taken to:
  - An EL3 mode other than Monitor mode if EL3 is using AArch32.
  - An EL1 mode if EL3 is using AArch64.

  If the implementation does not include EL2, any exception taken from Non-secure state that is not taken to Secure Monitor mode is taken to Non-secure state to the default mode for that exception. The default mode will be an EL1 mode.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In an implementation that includes EL2

An exception taken from Non-secure state that is not taken to Secure Monitor mode is taken to Non-secure state and:

- If the exception is taken from Hyp mode, then it is taken to Hyp mode.
- Otherwise, the exception is either taken to Hyp mode, as described in *Exceptions taken to Hyp mode*, or taken to the default mode for the exception.

Note

- Hyp mode is the EL2 mode. The other modes to which an exception can be taken in Non-secure state are EL1 modes.
- Hyp mode has no effect on the handling of exceptions taken from Secure state.

Table G1-7 on page G1-5705 shows the default mode to which each exception is taken.

Asynchronous exception routing controls on page G1-5732 describes the exception routing controls provided by EL2 and EL3.

Routing of aborts taken to AArch32 state on page G1-5722 gives more information about the modes to which memory aborts are taken.

The possible modes for taking each exception on page G1-5716 shows all modes to which each exception might be taken, in any implementation. That is, it applies to implementations:

- That include neither EL2 nor EL3.
- That include EL2 but not EL3.
- That do not include EL2 but include EL3.
- That include both EL2 and EL3.

Exceptions taken to Hyp mode

In an implementation that includes EL2 and EL3, when EL2 is using AArch32:

- Any exception taken from Hyp mode, that is not routed to EL3 by the controls described in *Asynchronous exception routing controls* on page G1-5732, is taken to Hyp mode.
- The following exceptions, if taken from Non-secure state, are taken to Hyp mode:
  - An abort that *Routing of aborts taken to AArch32 state* on page G1-5722 identifies as taken to Hyp mode.
  - A Hyp Trap exception, see *EL2 configurable controls* on page G1-5787.
  - A Hypervisor Call exception. This is generated by executing an `HVC` instruction in a Non-secure mode.
  - An SError interrupt exception, IRQ exception or FIQ exception that is not routed to EL3 but is explicitly routed to Hyp mode, as described in *Asynchronous exception routing controls* on page G1-5732.
  - A synchronous External abort, Alignment fault, Undefined Instruction exception, or Supervisor Call exception taken from the Non-secure EL0 mode and explicitly routed to Hyp mode, as described in *Routing exceptions from Non-secure EL0 to EL2* on page G1-5718.

    Note

    A synchronous External abort can be routed to Hyp mode only if it is not routed to EL3.

  - A debug exception that is explicitly routed to Hyp mode as described in *Routing debug exceptions to EL2 using AArch32* on page G1-5720.

Note

The virtual exceptions cannot be taken to Hyp mode. They are always taken to a Non-secure EL1 mode.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Security behavior in Exception levels using AArch32 when EL2 or EL3 are using AArch64

As described in *The Armv8-A security model* on page G1-5679, when EL3 is using AArch64, lower Exception levels, in either Security state, can be using AArch32. This means software executing in those Exception levels might try to access AArch32 security features that are not available. The following subsections describe the associated behaviors:

- *Execution of an SMC instruction from a privileged Exception level that is using AArch32*
- *Non-secure reads of the NSACR*
- *Secure EL1 operations when Secure EL1 is using AArch32 state*

Execution of an SMC instruction from a privileged Exception level that is using AArch32

When EL3 is using AArch64, an `SMC` instruction executed from Secure or Non-secure EL1 using AArch32, or from Non-secure EL2 using AArch32 when the value of HCR.TSC is 0, generates an exception that is taken to EL3. The exception syndrome is reported with an EC value of `0x13`, `SMC` instruction executed in AArch32 state, see *ISS encoding for an exception from SMC instruction execution in AArch32 state* on page D13-2985.

Non-secure reads of the NSACR

The NSACR is defined as being RO from Non-secure PE modes other than User mode. When EL3 is using AArch64, a read of the NSACR returns a fixed value of `0x00000C00` in the following cases:

- If the read is from a Non-secure EL1 mode when EL1 is using AArch32.
- If the read is from Hyp mode when EL2 is using AArch32.

Secure EL1 operations when Secure EL1 is using AArch32 state

When Secure EL1 is using AArch32 and if ARMv8.4-SecEL2 is implemented and enabled or EL3 is using AArch64:

- Any of the following operations performed in a Secure EL1 mode is trapped to Secure EL3:
  - A read or write of any of the SCR, NSACR, MVBAR, and SDCR.
  - Executing any of the ATS12NSO** instructions.
  - Executing an `SRS` instruction that would use SP_mon, see *SRS, SRSDA, SRSDB, SRSIA, SRSIB* on page F5-4729.
  - Executing an `MRS` (banked register) or `MSR` (banked register) instruction that would access SPSR_mon, SP_mon, or LR_mon, see *MRS (Banked register)* on page F5-4534 and *MSR (Banked register)* on page F5-4538.

  For more information about these traps, including the associated exception syndromes, see *Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32* on page D1-2391.

- Any attempt to move into Hypervisor mode, either by an exception return or by executing a `CPS` or `MSR` instruction, is treated as an illegal operation and is handled as described in *Illegal return events from AArch32 state* on page G1-5726.

- Any attempt to move into Monitor mode, either by an exception return or by executing a `CPS` or `MSR` instruction, is treated as an illegal operation and is handled as described in *Illegal return events from AArch32 state* on page G1-5726.

Note

This functionality supports a usage model where:

- EL3 uses AArch64.
- Secure software executed in Secure EL1 using AArch32 and Secure EL0 using AArch32.
- The Non-secure state uses AArch64.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The possible modes for taking each exception

Each of the exception descriptions in *AArch32 state exception descriptions* on page G1-5738 includes a subsection that describes the modes to which each exception can be taken. Those subsections are:

- *The PE mode to which the Undefined Instruction exception is taken* on page G1-5739.
- *The PE mode to which the Hyp Trap exception is taken* on page G1-5742.
- *The PE mode to which the Monitor Trap exception is taken* on page G1-5741.
- *The PE mode to which the Supervisor Call exception is taken* on page G1-5742.
- *The PE mode to which the Secure Monitor Call exception is taken* on page G1-5744.
- *The PE mode to which the Hypervisor Call exception is taken* on page G1-5745.
- *The PE mode to which the Prefetch Abort exception is taken* on page G1-5747.
- *The PE mode to which the Data Abort exception is taken* on page G1-5750.
- *The PE mode to which the Virtual SError interrupt exception is taken* on page G1-5754.
- *The PE mode to which the physical IRQ exception is taken* on page G1-5755.
- *The PE mode to which the Virtual IRQ exception is taken* on page G1-5756.
- *The PE mode to which the physical FIQ exception is taken* on page G1-5757.
- *The PE mode to which the Virtual FIQ exception is taken* on page G1-5758.

These descriptions also show the vector offset for the exception entry for each mode. These descriptions assume that all Exception levels are using AArch32, meaning:

- HCR, rather than HCR_EL2, controls the routing of exceptions to EL2.
- SCR, rather than SCR_EL3, controls the routing of exceptions to EL3.

For more information about:

- Vector offsets, see *Exception vectors and the exception base address* on page G1-5703.
- The routing of synchronous External aborts or SError, IRQ, and FIQ interrupt exceptions, and the virtual exceptions, see *Asynchronous exception routing controls* on page G1-5732.

UNPREDICTABLE cases when the value of HCR.TGE is 1

When the value of HCR.TGE is 1, exceptions that would otherwise be taken to EL1 are, instead, routed to EL2, see *Routing exceptions from Non-secure EL0 to EL2* on page G1-5718. Related to this, when the value of HCR.TGE is 1, execution in a Non-secure EL1 mode is UNPREDICTABLE. Armv8 does not constrain this UNPREDICTABLE behavior, but in Armv8 software that follows the Arm recommendations cannot get to this state. When following the Arm recommendations, any attempt to move to a Non-secure EL1 mode when the value of HCR.TGE is 1 is either:

- An illegal exception return, see *Illegal return events from AArch32 state* on page G1-5726.
- An illegal PE mode change, see *Illegal changes to PSTATE.M* on page G1-5699.

G1.12.5 PE state on exception entry

The description of each exception includes a pseudocode description of entry to that exception, as Table G1-8 on page G1-5711 shows. The following sections describe the PE state changes on entering an exception, for different implementations and operating states. However, you must always see the exception entry pseudocode for a full description of the state changes on exception entry:

- *Instruction set state on exception entry* on page G1-5717.
- *PSTATE.E value on exception entry* on page G1-5717.
- *PSTATE.{A, I, F, M} values on exception entry* on page G1-5717.

Note

The descriptions in these sections assume that EL2 and EL3, that control some aspects of the routing of exceptions taken from EL1 or EL0, are both using AArch32. If this is not the case:

- If EL2 is using AArch64:
  - Controls shown as provided by the HSCTLR are provided by the SCTLR_EL2.
  - Controls shown as provided by the HCR are provided by the HCR_EL2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If EL3 is using AArch64, controls shown as provided by the SCR are provided by the SCR_EL3.

Instruction set state on exception entry

Exception handlers can execute in either T32 state or A32 state. On exception entry, PSTATE.T is set to the required value, as determined by SCTLR.TE or HSCTLR.TE, depending on the mode the exception is taken to. Table G1-11 shows this:

Table G1-11 PSTATE.T bit value on exception entry

Mode to which exception is taken	HSCTLR.TE	SCTLR.TE	PSTATE.T	Exception handler state
Not Hyp mode	x	0	0	A32
		1	1	T32
Hyp mode	0	x	0	A32
	1	x	1	T32

When an implementation includes EL3 and EL3 is using AArch32, SCTLR is banked for Secure and Non-secure states, and therefore the TE bit value might be different for Secure and Non-secure states. For an exception taken to a PE mode other than Hyp mode, the SCTLR.TE bit for the Security state to which the exception is taken determines the instruction set state for the exception handler. This means the instruction set state in which an exception handler might execute depends on the Security state to which the exception is taken.

PSTATE.E value on exception entry

PSTATE.E controls the load and store endianness for data handling. Table G1-12 show the value to which this bit is set on exception entry:

Table G1-12 PSTATE.E value on exception entry

Exception mode	HSCTLR.EE	SCTLR.EE	Endianness for data loads and stores	PSTATE.E
Secure or Non-secure EL1	x	0	Little-endian	0
		1	Big-endian	1
Hyp	0	x	Little-endian	0
	1	x	Big-endian	1

For more information, see the bit description in *Saved Program Status Registers (SPSRs)* on page G1-5693.

PSTATE.{A, I, F, M} values on exception entry

On exception entry, PSTATE.M is set to the value for the mode to which the exception is taken, as described in *PE mode for taking exceptions* on page G1-5713.

Table G1-13 on page G1-5718 shows the cases where PSTATE.{A, I, F} bits are set to 1 on an exception entry, and how this depends on the mode and Security state to which an exception is taken. If the table entry for a particular mode and Security state does not define a value for a PSTATE.{A, I, F} bit then that bit is unchanged by the exception entry. In this table:

- The *PE mode exception is taken to* column is the mode to which the exception is taken.
- The *Non-secure* column applies to exceptions taken to Non-secure state in an implementation that includes EL3 but does not include EL2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The *Secure* column applies to:
  - Exceptions taken to Secure state.
  - Implementations that do not include the EL3.
  - Exceptions taken to Non-secure state in an implementation that includes EL2.

Table G1-13 PSTATE.{A, I, F} values on exception entry

PE mode exception is taken to	Security state	
	Non-secure	Secure
Hyp	If SCR.EA==0 then PSTATE.A is set to 1 If SCR.IRQ==0 then PSTATE.I is set to 1 If SCR.FIQ==0 then PSTATE.F is set to 1	-
Monitor	-	PSTATE.A is set to 1 PSTATE.I is set to 1 PSTATE.F is set to 1
FIQ	PSTATE.A is set to 1 PSTATE.I is set to 1 PSTATE.F is set to 1	PSTATE.A is set to 1 PSTATE.I is set to 1 PSTATE.F is set to 1
IRQ, Abort	PSTATE.A is set to 1 PSTATE.I is set to 1	PSTATE.A is set to 1 PSTATE.I is set to 1
Undefined, Supervisor	PSTATE.I is set to 1	PSTATE.I is set to 1

Asynchronous exception behavior for exceptions taken from AArch32 state on page G1-5730 describes how, in some situations, the PSTATE.{A, I, F} bits mask the taking of SError interrupts, IRQ interrupts, and FIQ interrupts.

G1.12.6 Routing exceptions from Non-secure EL0 to EL2

Note

The routing control described in this section permits a Non-secure state usage model where applications execute in User mode under a hypervisor, that executes in Hyp mode, without a Guest OS running at Non-secure EL1. This control applies when the PE is executing in Non-secure EL0 using AArch32 and EL2 is using AArch32 and the value of HCR.TGE is 1.

If the PE is in Non-secure User mode, any exception that would otherwise be taken to Non-secure EL1 is taken to EL2 if either:

- EL2 is using AArch32 and the value of HCR.TGE is 1.

  In this case the exception is taken to Hyp mode, instead of to the default Non-secure mode for handling the exception. For more information see *Exception reporting when HCR.TGE routes an exception to EL2 using AArch32* on page G1-5719.

- EL2 is using AArch64 and the value of HCR_EL2.TGE is 1.

  In this case the exception is taken to EL2 using AArch64, see *Exception entry* on page D1-2339.

Any exception that is routed to Secure Monitor mode or to EL3 using AArch64 is unaffected by the value of HCR.TGE or HCR_EL2.TGE.

When the value of HCR.TGE is 1, meaning TGE routing from Non-secure EL0 using AArch32 to EL2 using AArch32 applies:

- The SCTLR.M bit is treated as 0 for all purposes other than a direct read of the SCTLR register.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Each of the HCR.{FMO, IMO, AMO} bits is treated as 1 for all purposes other than a direct read of the HCR register
- Each of the HDCR.{TDE, TDA, TDRA, TDOSA} bits is treated as 1 for all purposes other than a direct read of the HDCR register.
- An exception return to Non-secure EL1 is treated as an illegal exception return, see *Illegal return events from AArch32 state* on page G1-5726.
- All virtual interrupts, including any IMPLEMENTATION DEFINED mechanisms for signaling virtual interrupts, are disabled.

Exception reporting when HCR.TGE routes an exception to EL2 using AArch32

The following sections give more information about the behavior of synchronous exceptions that are routed to Hyp mode because the value of HCR.TGE is 1:

- *Undefined Instruction exception, when the value of HCR.TGE is 1*.
- *Supervisor Call exception, when the value of HCR.TGE is 1*.
- *Abort exceptions, when the value of HCR.TGE is 1*.
- *Reporting of exceptions routed to EL2 using AArch32 because the value of HCR.TGE is 1* on page G1-5720.

Undefined Instruction exception, when the value of HCR.TGE is 1

When HCR.TGE is set to 1, if the PE is executing in Non-secure User mode and attempts to execute an UNDEFINED instruction, it takes the Hyp Trap exception, instead of an Undefined Instruction exception. On taking the Hyp Trap exception, the HSR reports an unknown reason for the exception, using the EC value `0x00`. For more information see *Use of the HSR* on page G5-6038.

Supervisor Call exception, when the value of HCR.TGE is 1

When HCR.TGE is set to 1, if the PE executes an `SVC` instruction in Non-secure User mode, the Supervisor Call exception generated by the instruction is taken to Hyp mode.

The HSR reports that entry to Hyp mode was because of a Supervisor Call exception, and:

- If the `SVC` is unconditional, takes for the `imm16` value in the HSR:
  - A zero-extended 8-bit immediate value for the T32 `SVC` instruction.

    Note

    The only T32 encoding for `SVC` is a 16-bit instruction encoding.

  - The bottom16 bits of the immediate value for the A32 `SVC` instruction.
- If the `SVC` is conditional, the `imm16` value in the HSR is UNKNOWN.

If the `SVC` is conditional, the PE takes the exception only if the instruction passes its Condition code check.

The HSR reports the exception as a Supervisor Call exception taken to Hyp mode, using the EC value `0x11`. For more information, see *Use of the HSR* on page G5-6038.

Note

The effect of setting HCR.TGE to 1 is to route the Supervisor Call exception to Hyp mode, not to trap the execution of the `SVC` instruction. This means that the preferred return address for the exception, when routed to Hyp mode in this way, is the instruction after the `SVC` instruction.

Abort exceptions, when the value of HCR.TGE is 1

When the value of HCR.TGE is 1, if the PE is executing in Non-secure User mode then any abort exception that is not routed to Secure Monitor mode or to EL3 using AArch64 generates an exception that is taken as a Hyp Trap exception. Where an attempt to execute an instruction causes an abort, on taking the Hyp Trap exception, the HSR indicates whether a Data Abort exception or a Prefetch Abort exception caused the Hyp Trap exception entry, and presents a valid syndrome in the HSR.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When SCR.EA is set to 1, External aborts and SError interrupts are routed to EL3, and this routing takes priority over the HCR.TGE routing. For more information, see *Routing of aborts taken to AArch32 state* on page G1-5722.

An SError interrupt that is routed to Hyp mode because the value of HCR.TGE is 1 is reported as a Data Abort exception routed to Hyp mode.

The HSR reports the exception either:

- As a Prefetch Abort exception routed to Hyp mode, using the EC value `0x20`.
- As a Data Abort exception routed to Hyp mode, using the EC value `0x24`.

For more information about the exception reporting, see *Use of the HSR* on page G5-6038.

Reporting of exceptions routed to EL2 using AArch32 because the value of HCR.TGE is 1

PL1 configurable controls on page G1-5779 describes controls that, when the value of HCR.TGE is 0, can generate exceptions that are taken from Non-secure EL0 to EL1. When EL2 is using AArch32 and the value of HCR.TGE is 1, the exceptions generated by these controls are routed to Hyp mode. Table G1-14 shows how these exceptions are then reported in the HSR.

Table G1-14 Syndrome reporting in HSR from HCR.TGE routing of traps, disables, and enables

Control provided by PL1	Control type[a]	Syndrome reporting in HSR
SCTLR.{nTWE, nTWI}	T	Uses EC value `0x00`, Exception for an unknown reason
SCTLR.{SED, ITD}	D	Uses EC value `0x00`, Exception for an unknown reason
SCTLR.CP15BEN	E	Uses EC value `0x00`, Exception for an unknown reason
CPACR.TRCDIS	T	Uses EC value `0x00`, Exception for an unknown reason
CPACR.{cp11, cp10}	E	Uses EC value `0x00`, Exception for an unknown reason
FPEXC.EN	E	Uses EC value `0x00`, Exception for an unknown reason
CPACR.ASEDIS	D	Uses EC value `0x00`, Exception for an unknown reason
DBGDSCRext.UDCCdis	T	Uses EC value `0x00`, Exception for an unknown reason
CNTKCTL.{PL0PTEN, PL0VTEN, PL0PCTEN, PL0VCTEN}	T	Uses EC value `0x00`, Exception for an unknown reason
PMUSERENR.{ER, CR, SW, EN}	T	Uses EC value `0x00`, Exception for an unknown reason

a. T indicates a trap control, E indicates an instruction enable, and D indicates an instruction disable. For the definition of these terms, see the list that begins with *Instruction enables and instruction disables* on page G1-5778.

G1.12.7 Routing debug exceptions to EL2 using AArch32

When the value of HDCR.TDE is 1, if the PE is executing in a Non-secure mode other than Hyp mode, any Debug exception is routed to Hyp mode. This means it generates a Hyp Trap exception. This applies to:

- Debug exceptions associated with an instruction fetch, that would otherwise generate a Prefetch Abort exception. These are the Breakpoint, Breakpoint Instruction, and Vector Catch exception, see Chapter G2 *AArch32 Self-hosted Debug*.
- Watchpoint exceptions associated with data accesses, that would otherwise generate a Data Abort exception. See *Watchpoint exceptions* on page G2-5855.

When the value of HDCR.TDE is 1, each of the HDCR.{TDRA, TDOSA, TDA} bits is treated as 1 for all purposes other than reading the HDCR register.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

- A Breakpoint or Watchpoint debug event that generates entry to Debug state cannot be trapped to Hyp mode. See *Breakpoint and Watchpoint debug events* on page H2-6928.
- When HDCR.TDE is set to 1, the Hyp Trap exception is generated instead of the Prefetch Abort exception or Data Abort exception that is otherwise generated by the Debug exception.
- Debug exceptions, other than Breakpoint Instruction exceptions, are never generated in Hyp mode.

When a Hyp Trap exception is generated because the value of HDCR.TDE is 1, The HSR reports the exception either:

- As a Prefetch Abort exception routed to Hyp mode, using the EC value `0x20`.
- As a Data Abort exception routed to Hyp mode, using the EC value `0x24`.

For more information see *Use of the HSR* on page G5-6038.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G1.13 Routing of aborts taken to AArch32 state

A memory abort is either a Data Abort exception or a Prefetch Abort exception. When executing in AArch32 state, depending on the cause of the abort, and possibly on configuration settings, an abort is taken either:

- To the Exception level of the PE mode from which the abort is taken. In this case the abort is taken to AArch32 state.
- To a higher Exception level. In this case the Exception level to which the abort is taken is either:
  - Using AArch32. In this case, this chapter describes how the abort is handled.
  - Using AArch64. In this case, Chapter D5 *The AArch64 Virtual Memory System Architecture* describes how the abort is handled.

For an abort taken to an Exception level that is using AArch32, the mode to which a memory abort is taken depends on the reason for the exception, the mode the PE is in when it takes the exception, and configuration settings, as follows:

Memory aborts taken to Monitor mode

If an implementation includes EL3, when the value of SCR.EA is 1, all External aborts are taken to EL3, and if EL3 is using AArch32 they are taken to Monitor mode. This applies to aborts taken from Secure modes and from Non-secure modes.

Memory aborts taken to Secure Abort mode

If an implementation includes EL3, when the PE is executing in Secure state, all memory aborts that are not routed to EL3 are taken to Secure Abort mode.

Note

The only memory aborts that can be routed to Monitor mode are External aborts.

Memory aborts taken to Hyp mode

If an implementation includes EL2, when the PE is executing in Non-secure state, the following aborts are taken to EL2. If EL2 is using AArch32 this means they are taken to Hyp mode:

- Alignment faults taken:
  - When the PE is in Hyp mode.
  - When the PE is in a Non-secure PL1 or EL0 mode and the exception is generated because the Non-secure PL1&0 stage 2 translation identifies the target of an unaligned access as any type of Device memory.
  - When the PE is in Non-secure User mode and HCR.TGE is set to 1. For more information see *Abort exceptions, when the value of HCR.TGE is 1* on page G1-5719.
- When the PE is using the Non-secure PL1&0 translation regime:
  - MMU faults from stage 2 translations, for which the stage 1 translation did not cause an MMU fault.
  - Any abort taken during the stage 2 translation of an address accessed in a stage 1 translation table walk that is not routed to Secure Monitor mode, see *Stage 2 fault on a stage 1 translation table walk* on page G5-6019.
- When the PE is using the Non-secure EL2 translation regime, MMU faults from stage 1 translations.

  Note

  The Non-secure EL2 translation regime has only one stage of translation.

- External aborts, if SCR.EA is set to 0 and any of the following applies:
  - The PE was executing in Hyp mode when it took the exception.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

  - The PE was executing in a Non-secure PL1 or EL0 mode when it took the exception, the abort is asynchronous, and HCR.AMO is set to 1. For more information see *Asynchronous exception routing controls* on page G1-5732.
  - The PE was executing in the Non-secure User mode when it took the exception, the abort is synchronous, and HCR.TGE is set to 1. For more information see *Abort exceptions, when the value of HCR.TGE is 1* on page G1-5719.
  - The The Reliability, Availability, and Serviceability Extension (RAS Extension) is implemented, the PE was executing in a Non-secure PL1 or EL0 mode when it took the exception, the abort is synchronous, and the value of HCR2.TEA is 1.
  - The abort occurred on a stage 2 translation table walk.
- Debug exceptions, if HDCR.TDE is set to 1. For more information, see *Routing debug exceptions to EL2 using AArch32* on page G1-5720.

Memory aborts taken to Non-secure Abort mode

In an implementation that does not include EL3, all memory aborts that are taken to an Exception level that is using AArch32 are taken to Abort mode.

Otherwise, when the PE is executing in Non-secure state, the following aborts are taken to Non-secure Abort mode:

- When the PE is in a Non-secure PL1 or EL0 mode, Alignment faults taken for any of the following reasons:
  - SCTLR.A is set to 1.
  - An instruction that does not support unaligned accesses is committed for execution, and the instruction accesses an unaligned address.
  - The PL1&0 stage 1 translation identifies the target of an unaligned access as any type of Device memory.

    Note

    In an implementation that does not include EL2, this case results in a CONSTRAINED UNPREDICTABLE memory access, see *Cases where unaligned accesses are CONSTRAINED UNPREDICTABLE* on page E2-4005 and *Loads and Stores to unaligned locations* on page K1-7832.

  If an implementation includes EL2 and the PE is in Non-secure User mode, these exceptions are taken to Abort mode only if the value of HCR.TGE is 0.
- When the PE is using the Non-secure PL1&0 translation regime, an MMU fault from a stage 1 translation.
- External aborts, if the PE was executing in a Non-secure PL1 or EL0 mode when it took the exception and both:
  - The value of SCR.EA is 0, meaning the abort is not taken to EL3.
  - The abort is not taken to EL2 for one of the reasons defined in Memory aborts taken to Hyp mode.
- Virtual Aborts, see *Virtual exceptions when an implementation includes EL2* on page G1-5730.
- When the value of HDCR.TDE is 0, Debug exceptions. For more information, see *Routing debug exceptions to EL2 using AArch32* on page G1-5720.

Note

If EL0 is using AArch32 and EL1 is using AArch64 then any of these memory aborts taken from User mode are taken to EL1 as described in Chapter D5 *The AArch64 Virtual Memory System Architecture*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Memory aborts with IMPLEMENTATION DEFINED behavior

In addition, a PE can generate an abort for an IMPLEMENTATION DEFINED reason associated with lockdown. In an implementation that includes EL2, whether such an abort is taken to Non-secure Abort mode or is taken to EL2 is IMPLEMENTATION DEFINED, and an implementation might include a mechanism to select whether the abort is routed to Non-secure Abort mode or to EL2.

When the PE is in a Non-secure mode other than Hyp mode, if multiple factors cause an Alignment fault, the abort is taken to Non-secure Abort mode if any of the factors require the abort to be taken to Abort mode. For example, if the SCTLR.A bit is set to 1, and the access is an unaligned access to an address that the stage 2 translation tables mark as Device-nGnRnE, then the abort is taken to Non-secure Abort mode.

For more information see *Handling exceptions that are taken to an Exception level using AArch32* on page G1-5703.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G1.14 Exception return to an Exception level using AArch32

In the Arm architecture, *exception return* to an Exception level that is using AArch32 requires the simultaneous restoration of the PC and PSTATE to values that are consistent with the desired state of execution on returning from the exception. Typically, exception return involves returning to one of:

- The instruction after the instruction boundary at which an asynchronous exception was taken.
- The instruction following an `SVC`, `SMC`, or `HMC` instruction, for an exception generated by one of those instructions.
- The instruction that caused the exception, after the reason for the exception has been removed.
- The subsequent instruction, if the instruction that caused the exception has been emulated in the exception handler.

The Arm architecture defines a *preferred return address* for each exception other than Reset, see *Link values saved on exception entry* on page G1-5711. The values of the SPSR.IT[7:0] bits generated on exception entry are always correct for this preferred return address, but might require adjustment by the exception handler if returning elsewhere.

In some cases, to calculate the appropriate preferred return address for a return to an Exception level that is using AArch32, a subtraction must be performed on the link value saved on taking the exception. The description of each exception includes any value that must be subtracted from the link value, and other information about the required exception return.

On an exception return, the PSTATE takes either:

- The value loaded by the `RFE` instruction.
- If the exception return is not performed by executing an `RFE` instruction, the value of the current SPSR at the time of the exception return.

If ARMv8.5-MemTag is implemented PSTATE.TCO is not updated on Exception return to AArch32 state.

Illegal return events from AArch32 state on page G1-5726 describes the behavior if the restored PE state would not be valid for the Exception level, PE mode, and Security state targeted by the exception return.

G1.14.1 Exception return instructions

The instructions that an exception handler can use to return from an exception depend on whether the exception was taken to an EL1 mode, or in an EL2 mode, see:

- *Return from an exception taken to a PE mode other than Hyp mode*.
- *Return from an exception taken to Hyp mode* on page G1-5726.

Return from an exception taken to a PE mode other than Hyp mode

For an exception taken to a PE mode other than Hyp mode, the Arm AArch32 architecture provides the following *exception return instructions*:

- From privileged modes other than System mode, the `ERET` instruction. After the exception return, execution resumes from the address held in the LR (R14) for the mode in which `ERET` is executed. See *ERET* on page F5-4369.
- Data-processing instructions with the S bit set and the PC as a destination, see *MOV, MOVS (register)* on page F5-4517 and *SUB, SUBS (immediate)* on page F5-4831.

Note

The A32 instruction set includes other instructions that can be used for an exception return, but Arm deprecates any use of those instructions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Typically:

— A return where no subtraction is required uses SUBS with an operand of 0, or the equivalent MOVS instruction.

— A return requiring subtraction uses SUBS with a nonzero operand.

- The RFE instruction, see *RFE, RFEDA, RFEDB, RFEIA, RFEIB* on page F5-4624. If a subtraction is required, typically it is performed before saving the LR value to memory. After the exception return, execution resumes from the address held in the memory location indicated by the base register specified by the RFE instruction.
- In A32 state, a form of the LDM instruction in which the PC is one of the registers loaded, see *LDM (exception return)* on page F5-4402. If a subtraction is required, typically it is performed before saving the LR value to memory.

Return from an exception taken to Hyp mode

For an exception taken to Hyp mode, the Arm architecture provides the ERET instruction, see *ERET* on page F5-4369. An exception handler executing in Hyp mode must return using the ERET instruction.

Hyp mode is implemented only as part of EL2.

G1.14.2 Alignment of exception returns

The T bit of the value transferred to the PSTATE by an exception return controls the target instruction set of that return. The behavior of the hardware for exception returns for different values of the T bit is as follows:

T == 0 The target instruction set state is A32 state. Bits[1:0] of the address transferred to the PC are ignored by the hardware.

T == 1 The target instruction set state is T32 state:
- Bit[0] of the address transferred to the PC is ignored by the hardware.
- Bit[1] of the address transferred to the PC is part of the instruction address.

Note

In previous versions of the Arm architecture, the PSTATE.{J, T} bits determined the Instruction set state. In Armv8, PSTATE.J is RES0.

Arm deprecates any dependence on the requirements that the hardware ignores bits of the address. Arm recommends that the address transferred to the PC for an exception return is correctly aligned for the target instruction set.

After an exception entry other than Reset, the LR value has the correct alignment for the instruction set indicated by the SPSR.T bit. This means that if exception return instructions are used with the LR and SPSR values produced by such an exception entry, the only precaution software needs to take to ensure correct alignment is that any subtraction is of a multiple of four if returning to A32 state, or a multiple of two if returning to T32 state.

G1.14.3 Illegal return events from AArch32 state

Throughout this section:

Return In AArch32 state, refers to any of:
- Execution of any exception return instruction.
- Execution of a DRPS instruction in Debug state.
- Exit from Debug state.

If an exception or debug return from an Exception level using AArch32 triggers an illegal exception return, then bit[1] of the PC is either:
- Zero.
- The value of bit[1] of the return address for the exception or debug return.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The choice between these two alternatives is made by the implementation, and might differ from instance to instance of an illegal exception return.

Note

This means software must support both alternatives.

Saved process state value
In AArch32 state, refers to any of:

- The value held in the SPSR for any exception return other than an exception return made by executing an RFE instruction.
- The value read from memory that is to be restored to PSTATE by the execution of an RFE instruction.
- The value held in the SPSR for the execution of a DRPS instruction in Debug state.
- The value held in the DSPSR for a Debug state exit.

Link address In AArch32 state, refers to any of:

- The address held in the link register for any exception return other than an exception return made by executing an ERET, LDM, or RFE instruction.
- The address held in ELR_hyp for any exception return made by executing an ERET instruction.
- The address read from memory that is to be restored to the PC by the execution of an LDM or RFE instruction.
- The address held in the DLR for Debug state exit.

Configured from reset
Indicates the state determined on powerup or reset by a configuration input signal, or by another IMPLEMENTATION DEFINED mechanism.

The Armv8 architecture has a generic mechanism for handling exception or debug returns to a mode or state that is illegal. In AArch32 state, this can occur as a result of any of the following situations:

- A return where the Exception level being returned to is higher than the current Exception level.
- A return where the mode being returned to is not implemented. For example:
  - A return to Hyp mode when EL2 is not implemented.
  - A return to Monitor mode, when EL3 is either not implemented or using AArch64 state.
- A return to EL2 when:
  - EL3 is implemented and using AArch64, and the values of SCR_EL3.{NS, EEL2} 0.
  - EL3 is implemented and using AArch32, and the value of the SCR.NS bit is 0.
- A return to Non-secure EL1 when:
  - EL2 is implemented and using AArch64, and the value of the HCR_EL2.TGE bit is 1.
  - EL2 is implemented and using AArch32, and the value of the HCR.TGE bit is 1.
- A return where the value of the saved process state M[4:0] field is not a valid AArch32 PE mode for the implementation. Table G1-5 on page G1-5686 shows the valid M[4:0] values for AArch32 PE modes.

In these cases:

- PSTATE.IL is set to 1, to indicate an illegal return.
- PSTATE.M is unchanged. This means the PE mode does not change.
- The SS bit is handled in the same way as any other exception or debug return, see *Software Step exceptions* on page D2-2472.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The following PSTATE bits are restored from the saved process state value:
  - The N, Z, C, V Condition flags.
  - The Q Overflow or saturation flag.
  - The GE Greater than or Equal flags.
  - The E Endianness mapping bit.
  - The A, I, F exception mask bits.
  - The DIT Data Independent Timing bit.
- The PSTATE.{IT, T} bits are each either:
  - Set to 0.
  - Copied from the saved process state in the SPSR for the PE mode in which the exception is handled.

  The choice between these two options is determined by an implementation, and might vary dynamically within an implementation. Correspondingly software must regard the value as being an UNKNOWN choice between the two values.
- The PC is restored from the link address, unless the illegal return is the execution of a DRPS instruction in Debug state.

When the value of the PSTATE.IL bit is 1, any attempt to execute any instruction results in an Illegal Execution state exception. See *The Illegal Execution state exception*.

All aspects of the illegal return, other than the effects described in this section, are the same as for a legal return.

G1.14.4 Legal returns that set PSTATE.IL to 1

In this section, return, saved process state value, and link address have the meaning that is defined in *Illegal return events from AArch32 state* on page G1-5726.

If the IL bit in the saved process state value is 1, then it is copied to PSTATE meaning that PSTATE.IL is set to 1. In this case, the PSTATE.{IT, T} bits are each either:

- Set to 0.
- Copied from the SPSR, or loaded from memory if the exception return was performed by executing an RFE instruction.

The choice between these two options is determined by an implementation, and might vary dynamically within the implementation. This means software must regard each value as being an UNKNOWN choice between the two permitted values.

Because the return sets the PSTATE.IL bit to 1, any attempt to execute any instruction results in an Illegal Execution state exception. See *The Illegal Execution state exception*.

G1.14.5 The Illegal Execution state exception

When the value of the PSTATE.IL bit is 1, any attempt to execute an instruction generates an Illegal Execution state exception. In AArch32 state, the PSTATE.IL bit can be set to 1 by one of the following:

- An illegal return, as described in *Illegal return events from AArch32 state* on page G1-5726.
- An illegal change to PSTATE.M, as described in *Illegal changes to PSTATE.M* on page G1-5699.
- A legal return that sets PSTATE.IL to 1, as described in *Legal returns that set PSTATE.IL to 1*.

An Illegal Execution state exception is taken in the same way as an Undefined Instruction exception in the current Exception level. If the current Exception level is EL2 using AArch32 state, the HSR provides additional syndrome information for the exception, see *Use of the HSR* on page G5-6038.

An Illegal Execution state exception has priority over any other Undefined Instruction exception that might arise from instruction execution.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

This section only describes the handling of an Illegal Execution state exception that is taken to an Exception level that is using AArch32 state. *The Illegal Execution state exception* on page D1-2353 describes the cases where an Illegal Execution state exception is taken to an Exception level that is using AArch64 state.

On taking any exception to an Exception level that is using AArch32 state:

1. The value of the PSTATE.IL bit is 1 and this is copied to the SPSR.IL bit for the PE mode to which the exception is taken.
2. The PSTATE.IL bit is cleared to 0.

Note

This means that it is not possible for software to observe the value of PSTATE.IL.

Pseudocode description of exception return

The `AArch32.ExceptionReturn()` function transfers the return address to the PC and restores PSTATE to its saved value.

This function uses the function `SetPSTATEFromPSR()`.

The `IllegalExceptionReturn()` function checks for an Illegal Execution state exception.

Chapter J1 *Armv8 Pseudocode* includes the definitions of these functions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G1.15 Asynchronous exception behavior for exceptions taken from AArch32 state

In an implementation that does not include EL2 or EL3, the asynchronous exceptions behave as follows when EL1 and EL0 are both using AArch32:

- An SError interrupt is taken to Abort mode.
- An IRQ exception is taken to IRQ mode.
- An FIQ exception is taken to FIQ mode.

These are the *default PE modes* for taking these exceptions.

Note

The SError interrupt replaces the Armv7 asynchronous abort. The new name better describes the nature of the exception.

However, the PSTATE.{A, I, F} bits *mask* the asynchronous exceptions, meaning that when the value of one of these PSTATE bits is 1, the corresponding exception is not taken.

If a masked asynchronous exception remains signaled, then the exception remains pending unless the value of the PSTATE bit is changed to 0.

EL2 and EL3 provide controls that affect:

- The routing of these exceptions, see *Asynchronous exception routing controls* on page G1-5732.
- Masking of these exceptions in Non-secure state, see *Asynchronous exception masking controls* on page G1-5733.

Similar register control bits are provided regardless of whether EL2 and EL3 are using AArch32 or AArch64:

- The EL2 controls are provided by the HCR when EL2 is using AArch32, and by the HCR_EL2 when EL2 is using AArch64.
- The EL3 controls are provided by the SCR when EL3 is using AArch32, and by the SCR_EL3 when EL3 is using AArch64.

Therefore, most references to the HCR or SCR in this section are to entries in Table K14-1 on page K14-8038, that disambiguates between AArch32 registers and AArch64 registers. However, the Execution states used by EL2 and EL3 do affect some aspects of the routing and masking of the asynchronous exceptions, see *Asynchronous exception routing and masking with higher Exception levels using AArch64* on page G1-5735.

G1.15.1 Virtual exceptions when an implementation includes EL2

When implemented, EL2 provides the following virtual exceptions, that correspond to the physical asynchronous exceptions:

- Virtual SError, that corresponds to a physical external SError interrupt.
- Virtual IRQ, that corresponds to a physical IRQ.
- Virtual FIQ, that corresponds to a physical FIQ.

When the value of HCR.TGE is 0 and the value of an HCR.{AMO, IMO, FMO} routing control bit is 1, the corresponding virtual interrupt is enabled and a virtual exception is generated either:

- By setting the corresponding virtual interrupt pending bit, HCR.{VA, VI, VF}, to 1.
- For a Virtual IRQ or Virtual FIQ, by an IMPLEMENTATION DEFINED mechanism. This might be a signal from an interrupt controller. See, for example, the *ARM Generic Interrupt Controller Architecture Specification*.

When the value of HCR_EL2.TGE is 1 all virtual interrupts are disabled.

When a virtual interrupt is disabled:

- It cannot be taken.
- It cannot be seen in the ISR.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In AArch32 state, a virtual exception is taken only from a Non-secure EL1 or EL0 mode. In any other mode, if the exception is generated it is not taken.

A virtual exception is taken in Non-secure state to the default mode for the corresponding physical exception. This means:

- A Virtual SError is taken to Non-secure Abort mode.
- A Virtual IRQ is taken to Non-secure IRQ mode.
- A Virtual FIQ is taken to Non-secure FIQ mode.

Table G1-15 summarizes the HCR bits that route asynchronous exceptions to EL2, and the bits that generate the virtual exceptions.

Table G1-15 HCR bits controlling asynchronous exceptions

Exception	Routing the physical exception to EL2	Generating the virtual exception
SError	HCR.AMO	HCR.VA
IRQ	HCR.IMO	HCR.VI
FIQ	HCR.FMO	HCR.VF

The HCR.{VA, VI, VF} bits generate a virtual exception only if set to 1 when the value of the corresponding HCR.{AMO, IMO, FMO} is 1.

Similarly, if the implementation also includes EL3, the HCR.{AMO, IMO, FMO} bits route the corresponding physical exception to Hyp mode only if the physical exception is not routed to Monitor mode by the SCR.{EA, IRQ, FIQ} bit. For more information, see *Asynchronous exception routing controls* on page G1-5732.

When the value of an HCR.{AMO, IMO, FMO} control bit is 1, the corresponding mask bit in PSTATE:

- Does not mask the physical exception.
- Masks the virtual exception when the PE is executing in a Non-secure EL1 or EL0 mode.

Taking a Virtual Abort exception clears HCR.VA to zero. Taking a Virtual IRQ exception or a Virtual FIQ exception does not affect the value of HCR.VI or HCR.VF.

Note

This means that the exception handler for a Virtual IRQ exception or a Virtual FIQ exception must cause software that is executing at EL2 or EL3 to update the HCR to clear the appropriate virtual exception bit to 0.

See *WFE wake-up events* on page G1-5766 and *Wait For Interrupt* on page G1-5767 for information about how virtual exceptions affect wake up from power-saving states.

Note

A hypervisor can use virtual exceptions to signal exceptions to the current Guest OS. The Guest OS takes a virtual exception exactly as it would take the corresponding physical exception, and is unaware of any distinction between virtual exception and the corresponding physical exception.

Effects of the HCR.{AMO, IMO, FMO} bits

As described in this section, the HCR.{AMO, IMO, FMO} bits are part of the mechanism for enabling the virtual exceptions. In addition, for exceptions generated in Non-secure state:

- As mentioned in this section, affect the routing of the exceptions. See *Asynchronous exception routing controls* on page G1-5732.
- Affect the masking of the exceptions. See *Asynchronous exception masking controls* on page G1-5733.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G1.15.2 Asynchronous exception routing controls

Note

This section describes the behavior when all Exception levels are using AArch32. For the differences when this is not the case see *Asynchronous exception routing and masking with higher Exception levels using AArch64* on page G1-5735.

In an implementation that includes EL3 the following bits in the SCR control the routing of asynchronous exceptions:

SCR.EA When the value of this bit is 1, any SError interrupt is taken to EL3.

Note

Although this section describes the asynchronous exception routing controls, SCR.EA also controls the routing of synchronous External aborts, see *Routing of aborts taken to AArch32 state* on page G1-5722.

SCR.FIQ When the value of this bit is 1, any FIQ exception is taken to EL3.

SCR.IRQ When the value of this bit is 1, any IRQ exception is taken to EL3.

When EL3 is using AArch32 and the value of one of the SCR.{EA, FIQ, IRQ} bits is 1, the exception is taken to Monitor mode.

Only Secure software can change the values of these bits.

In an implementation that includes EL2, the following bits in the HCR route asynchronous exceptions to EL2, for exceptions that are both:

- Taken from a Non-secure EL1 or EL0 mode.
- If the implementation also includes EL3, not configured, by the SCR.{EA, FIQ, IRQ} controls, to be taken to EL3.

HCR.AMO When the value of this bit is 1, an SError interrupt exception taken from a Non-secure EL1 or EL0 mode is taken to EL2, instead of to Non-secure Abort mode. If the implementation also includes EL3, this control applies only if the value of SCR.EA is 0. When the value of SCR.EA is 1, the value of the AMO bit is ignored.

HCR.FMO When the value of this bit is 1, an FIQ exception taken from a Non-secure EL1 or EL0 mode is taken to EL2, instead of to Non-secure FIQ mode. If the implementation also includes EL3, this control applies only if the value of SCR.FIQ is 0. When the value of SCR.FIQ is 1, the value of the FMO bit is ignored.

HCR.IMO When the value of this bit is 1, an IRQ exception taken from a Non-secure EL1 or EL0 mode is taken to EL2, instead of to Non-secure IRQ mode. If the implementation also includes EL3, this control applies only if the value of SCR.IRQ is 0. When the value of SCR.IRQ is 1, the value of the IMO bit is ignored.

When EL2 is using AArch32 and the value of one of the HCR.{AMO, FMO, IMO} bits is 1, the exception is taken to Hyp mode.

Only software executing in Hyp mode, or Secure software executing at EL3 with SCR.NS set to 1, can change the values of these bits. If EL3 is using AArch32, this requires the Secure software to be executing in Monitor mode.

The HCR.{AMO, FMO, IMO} bits also affect the masking of asynchronous exceptions in Non-secure state, as described in *Asynchronous exception masking controls* on page G1-5733.

The SCR.{EA, FIQ, IRQ} and HCR.{AMO, FMO, IMO} bits have no effect on the routing of Virtual Abort, Virtual FIQ, and Virtual IRQ exceptions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

When the PE is in Hyp mode:

- Physical asynchronous exceptions that are not routed to Monitor mode are taken to Hyp mode.
- Virtual exceptions are not signaled to the PE.

See also *Asynchronous exception behavior for exceptions taken from AArch32 state* on page G1-5730.

G1.15.3 Asynchronous exception masking controls

Note

This section describes the behavior when all Exception levels are using AArch32. For the differences when this is not the case see *Asynchronous exception routing and masking with higher Exception levels using AArch64* on page G1-5735.

The PSTATE.{A, I, F} bits can mask the taking of the corresponding exceptions from AArch32 state, as follows:

- PSTATE.A can mask SError interrupt exceptions.
- PSTATE.I can mask IRQ exceptions.
- PSTATE.F can mask FIQ exceptions.

In an implementation that does not include either of EL2 and EL3, setting one of these bits to 1 masks the corresponding exception, meaning the exception cannot be taken.

In an implementation that includes EL2, the HCR.{AMO, IMO, FMO} bits modify the masking of exceptions taken from Non-secure state.

Similarly, in an implementation that includes EL3, the SCR.{AW, FW} bits modify the masking of exceptions taken from Non-secure state by the PSTATE.{A, F} bits.

An implementation that includes only EL1 and EL0 does not provide any masking of the PSTATE.{A, I, F} bits. The following subsections describe the masking of these bits in other implementations:

- *Asynchronous exception masking in an implementation that includes EL2 but not EL3*.
- *Asynchronous exception masking in an implementation that includes EL3 but not EL2*.
- *Asynchronous exception masking in an implementation that includes both EL2 and EL3* on page G1-5734.
- *Summary of the asynchronous exception masking controls* on page G1-5734.

Asynchronous exception masking in an implementation that includes EL2 but not EL3

The HCR.{AMO, IMO, FMO} bits modify the effect of the PSTATE.{A, I, F} bits. When the value of an HCR.{AMO, IMO, FMO} mask override bit is 1, the value of the corresponding PSTATE.{A, I, F} bit is ignored when the exception is taken from a Non-secure mode other than Hyp mode.

Asynchronous exception masking in an implementation that includes EL3 but not EL2

The SCR.{AW, FW} bits modify the effect of the PSTATE.{A, F} bits. When the value of one of the SCR.{AW, FW} bits is 0, the corresponding PSTATE bit is ignored when both of the follow apply:

- The corresponding exception is taken from Non-secure state.
- The value of the corresponding SCR.{EA, FIQ} bit is 1, routing the exception to EL3. This means the exception is routed to Monitor mode if EL3 is using AArch32.

Note

Whenever the value of PSTATE.I is 1, IRQ exceptions are masked and cannot be taken.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Asynchronous exception masking in an implementation that includes both EL2 and EL3

When the value of an HCR.{AMO, IMO, FMO} mask override bit is 1, the value of the corresponding PSTATE.{A, I, F} bit is ignored when both of the following apply:

- The exception is taken from Non-secure state.
- Either:
  - The corresponding SCR.{EA, IRQ, FIQ} bit routes the exception to Monitor mode.
  - The exception is taken from a Non-secure mode other than Hyp mode.

In addition, when the value of an SCR.{AW, FW} bit is 0, the value of the corresponding PSTATE.{A, F} bit is ignored when all of the following apply:

- The exception is taken from Non-secure state.
- The corresponding SCR.{EA, FIQ} bit routes the exception to Monitor mode.
- The corresponding HCR.{AMO, FMO} mask override bit is set to 0.

Summary of the asynchronous exception masking controls

The tables in this section show the masking controls for each of the PSTATE.{A, I, F} bits. For an implementation that does not include all of the Exception levels:

If the implementation includes only EL1 and EL0

The PSTATE bits cannot be masked. The behavior is as shown in the *Secure* row of the tables.

If the implementation includes EL2 but not EL3

The behavior is as shown in the *Non-secure* table rows when the control bits in the SCR are both 0.

If the implementation includes EL3 but not EL2

The behavior is as shown in the table rows where the control bit in the HCR is 0.

Table G1-16 shows the controls of the masking of SError interrupt exceptions by PSTATE.A.

Table G1-16 Control of masking by PSTATE.A

Security state	HCR.AMO	SCR.EA	SCR.AW	Mode	PSTATE.A
Secure	x	x	x	x	Masks SError interrupt, when set to 1
Non-secure	0	0	x	x	Masks SError interrupt, when set to 1
		1	0	x	Ignored
			1	x	Masks SError interrupt, when set to 1
	1	x	x	Not Hyp	Ignored
		0	x	Hyp	Masks SError interrupt, when set to 1
		1	x	x	Ignored

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G1-17 shows the controls of the masking of IRQ exceptions by PSTATE.I:

Table G1-17 Control of masking by PSTATE.I

Security state	HCR.IMO	SCR.IRQ	Mode	PSTATE.I
Secure	x	x	x	Masks IRQs, when set to 1
Non-secure	0	x	x	Masks IRQs, when set to 1
	1	x	Not Hyp	Ignored
		0	Hyp	Masks IRQs, when set to 1
		1	x	Ignored

Table G1-18 shows the controls of the masking of FIQ exceptions by PSTATE.F:

Table G1-18 Control of masking by PSTATE.F

Security state	HCR.FMO	SCR.FIQ	SCR.FW	Mode	PSTATE.F
Secure	x	x	x	x	Masks FIQs, when set to 1
Non-secure	0	0	x	x	Masks FIQs, when set to 1
		1	0	x	Ignored
			1	x	Masks FIQs, when set to 1
	1	x	x	Not Hyp	Ignored
		0	x	Hyp	Masks FIQs, when set to 1
		1	x	x	Ignored

G1.15.4 Asynchronous exception routing and masking with higher Exception levels using AArch64

Asynchronous exception routing controls on page G1-5732 and *Asynchronous exception masking controls* on page G1-5733 give full descriptions of the routing and masking of the asynchronous exceptions when all Exception levels are using AArch32. However, when EL0 and EL1 are using AArch32:

- As already described, the SCR and HCR controls might be from Exception levels that are using AArch64.
- If EL3 is using AArch64, or EL2 is using AArch64, there are some changes to the asynchronous exception behaviors.

Therefore, the following sections summarize the asynchronous exception behaviors, taking account of the Execution state being used at EL2 and EL3:

- *Summary of physical interrupt routing*.
- *Summary of physical interrupt masking* on page G1-5736.

Summary of physical interrupt routing

The Table G1-19 on page G1-5736 shows the routing of physical FIQ, IRQ and SError interrupts when the highest Exception level is using AArch32. If the highest Exception level is using AArch64, see Table D1-10 on page D1-2364.

In this table:

SCR This is the *Effective value* of a field in SCR.

HCR This is the *Effective value* of a field in HCR.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

FIQ IRQ EA The *Effective value* of the field that handles the asynchronous exception type in SCR.

FMO IMO AMO The *Effective value* of the mask override field for the asynchronous exception type in HCR, if EL2 is using AArch32 or HCR_EL2 if EL2 is using AArch64.

FIQ IRQ Abt The exception is taken to the FIQ mode, the IRQ mode or the Abort mode according to the type of asynchronous exception.

Hyp The exception is taken to AArch32 Hyp mode.

Mon The exception is taken to AArch32 Monitor mode.

n/a This field does not exist, or the Exception level is not accessible in this configuration.

Table G1-19 Routing of physical asynchronous exceptions

Control bits				Target when taken from EL0	Target when taken from EL1	Target when taken from EL2	Target when taken from EL3
SCR		HCR					
NS	FIQ IRQ EA	TGE	FMO IMO AMO				
0	x	x	x	FIQ IRQ Abt	n/a	n/a	FIQ IRQ Abt
1	0	0	0	FIQ IRQ Abt	FIQ IRQ Abt	Hyp	FIQ IRQ Abt
			1	Hyp	Hyp	Hyp	FIQ IRQ Abt
		1	x	Hyp	n/a	Hyp	FIQ IRQ Abt
	1	0	x	Mon	Mon	Mon	Mon
		1	x	Mon	n/a	Mon	Mon

Summary of physical interrupt masking

Table G1-20 on page G1-5737 shows the masking of physical FIQ, IRQ and SError interrupts when the highest Exception level is using AArch32. When the highest Exception level is using AArch64, see Table D1-13 on page D1-2368.

In this table:

SCR This is the *Effective value* of a field in SCR.

HCR This is the *Effective value* of a field in HCR.

FIQ IRQ EA The *Effective value* of the field that handles the asynchronous exception type in SCR.

FMO IMO AMO The *Effective value* of the mask override field for the asynchronous exception type in HCR.

FW AW For FIQ interrupts, the SCR.FW field, and for SError interrupts, the SCR.AW field. For IRQ interrupts, there is no equivalent field, so the *Effective value* is 0 and rows where this cell is 1 should be ignored.

A When the interrupt is asserted, it is taken regardless of the value of the PSTATE mask bit.

B When the interrupt is asserted, it is subject to the corresponding PSTATE mask bit. If the value of the mask is 1, the interrupt is not taken. If the value of the mask is 0, the interrupt is taken.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

n/a This field does not exist, or the Exception level is not accessible in this configuration.

Table G1-20 Masking of physical asynchronous exceptions

Control bits					Effect of the interrupt mask when executing at:			
SCR			HCR					
NS	FW AW	FIQ IRQ EA	TGE	FMO IMO AMO	EL0	EL1	EL2	EL3
0	x	x	x	x	B	n/a	n/a	B
1	x	0	0	0	B	B	B	B
				1	A	A	B	B
			1	x	A	n/a	B	B
	0	1	0	x	A	A	A	B
			1	x	A	n/a	A	B
	1	1	0	0	B	B	B	B
				1	A	A	A	B
			1	x	A	n/a	A	B

G1.15.5 Taking an interrupt or other exception during a multiple-register load or store

In AArch32 state, an interrupt cannot be taken during a sequence of memory accesses caused by a single load or store instruction, except that when ARMv8.2-LSMAOC is implemented and the value of the applicable LSMAOE field is 0, an interrupt can be taken between two memory accesses made by a single AArch32 Load Multiple (`LDM`) or Store Multiple (`STM`) instruction.

The applicable LSMAOE field is the field in the SCTLR_EL1, SCTLR_EL2, HSCTLR, or SCTLR register that applies to the Exception level and Security state at which the `LDM` or `STM` instruction is executed.

When the value of the LSMAOE bit is 0 and an interrupt is taken between two memory accesses made by a single AArch32 `LDM` or `STM` instruction, then:

- For a load, any register being loaded by the instruction other than a register used in the generation of the address by the instruction or the PC, can contain an UNKNOWN value. Any register used in the generation of the address is restored to its initial value and the LR is set on the interrupt to a value consistent with returning to the instruction.
- For a store, any data location being stored to by the instruction can contain an UNKNOWN value.
- For either a load or store, if the instruction specifies writeback of the base address, then that register is restored to its initial value.

Armv8.2 deprecates software relying on interrupts not being taken during the sequence of memory accesses caused by a single load or store instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G1.16 AArch32 state exception descriptions

Handling exceptions that are taken to an Exception level using AArch32 on page G1-5703 gives general information about exception handling. This section describes each of the exceptions, in the following subsections:

- *Undefined Instruction exception*.
- *Monitor Trap exception* on page G1-5740.
- *Hyp Trap exception* on page G1-5741.
- *Supervisor Call (SVC) exception* on page G1-5742.
- *Secure Monitor Call (SMC) exception* on page G1-5743.
- *Hypervisor Call (HVC) exception* on page G1-5744.
- *Prefetch Abort exception* on page G1-5745.
- *Data Abort exception* on page G1-5749.
- *Virtual SError interrupt exception* on page G1-5753.
- *IRQ exception* on page G1-5754.
- *Virtual IRQ exception* on page G1-5756.
- *FIQ exception* on page G1-5756.
- *Virtual FIQ exception* on page G1-5758.

Additional pseudocode functions for exception handling on page G1-5758 gives additional pseudocode that is used in the pseudocode descriptions of a number of the exceptions.

G1.16.1 Undefined Instruction exception

An Undefined Instruction exception might be caused by:

- A System register access, floating-point, or Advanced SIMD instruction that is not accessible because of the settings in one or more of the CPACR, NSACR, HCPTR, and DBGDSCRext.
- A System register access, floating-point, or Advanced SIMD instruction that is not implemented.
- A System register access, floating-point, or Advanced SIMD instruction that causes an exception during execution. This includes:
  - Trapped floating-point exceptions that are taken to AArch32, if an implementation supports these traps. See *Floating-point exceptions and exception traps* on page E1-3963.
  - Execution of certain floating-point instructions when one or both of the FPSCR.{Stride, Len} fields in nonzero, in an implementation in which those fields are RW. The description of FPEXC specifies the instructions to which this applies.
- An instruction that is UNDEFINED.

Note

The Undefined Instruction exception is taken using offset `0x04` in the Hyp, Secure, or Non-secure vector table. In the Monitor vector table this offset is used for the Monitor Trap exception. See *Monitor Trap exception* on page G1-5740 and *The vector tables and exception offsets* on page G1-5704.

By default, an Undefined Instruction exception is taken to Undefined mode, but an Undefined Instruction exception can be taken to EL2, meaning it is taken to Hyp mode if EL2 is using AArch32, see *The PE mode to which the Undefined Instruction exception is taken* on page G1-5739.

The Undefined Instruction exception can provide:

- Signaling of an illegal instruction execution.
- *Lazy context switching* of System registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The preferred return address for an Undefined Instruction exception is the address of the instruction that generated the exception. For an exception taken to AArch32 state, this return is performed as follows:

- If returning from Secure or Non-secure Undefined mode, the exception return uses the SPSR and LR_und values generated by the exception entry, as follows:
  - If SPSR.T is 0, indicating that the exception occurred in A32 state, the return uses an exception return instruction with a subtraction of 4.
  - If SPSR.T is 1, indicating that the exception occurred in T32 state, the return uses an exception return instruction with a subtraction of 2.
- If returning from Hyp mode, the exception return is performed by an `ERET` instruction, using the SPSR and ELR_hyp values generated by the exception entry.

For more information, see *Exception return to an Exception level using AArch32* on page G1-5725.

Note

If handling the Undefined Instruction exception requires instruction emulation, followed by return to the next instruction after the instruction that caused the exception, the instruction emulator must use the instruction length to calculate the correct return address, and to calculate the updated values of the IT bits if necessary.

The PE mode to which the Undefined Instruction exception is taken

Figure G1-4 shows how the implementation, state, and configuration options determine the PE mode to which an Undefined Instruction exception is taken, when the exception is taken to an Exception level that is using AArch32.

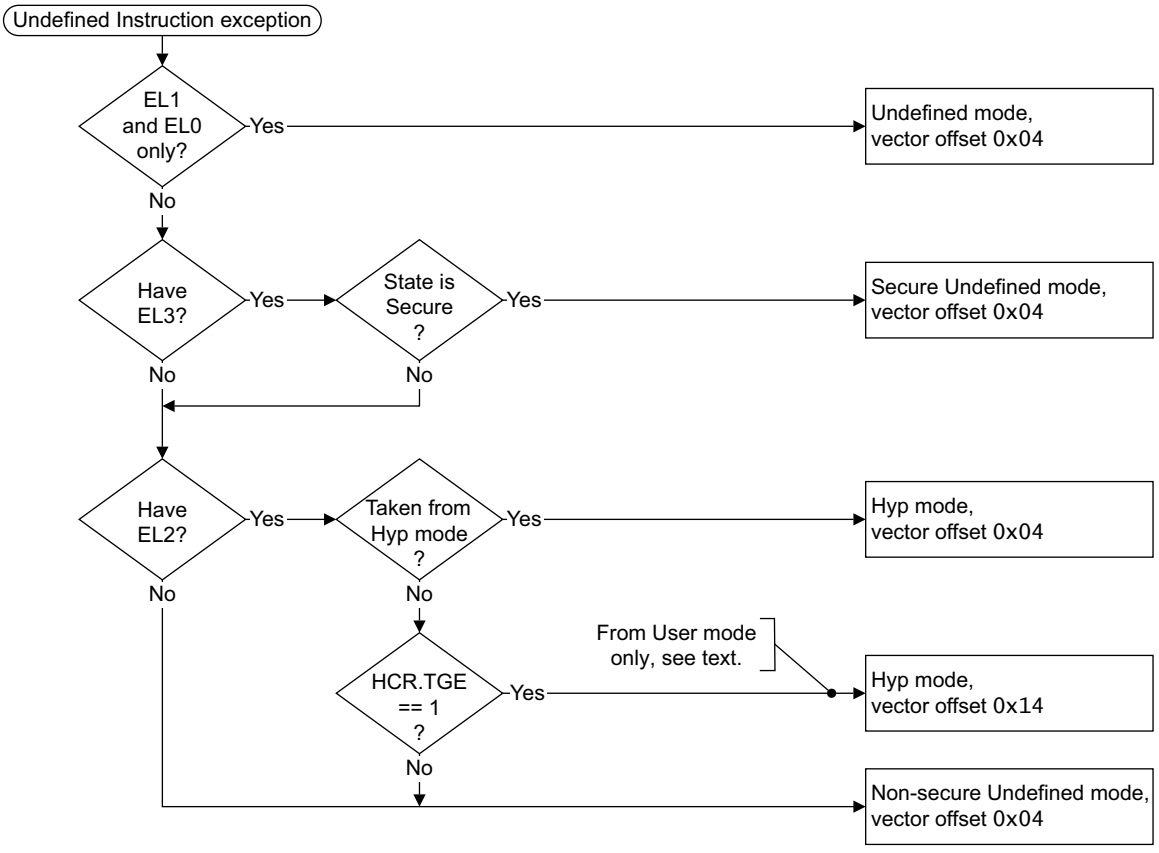


Figure G1-4 The PE mode an Undefined Instruction exception is taken to in AArch32 state

See also *UNPREDICTABLE cases when the value of HCR.TGE is 1* on page G1-5716.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Pseudocode description of taking the Undefined Instruction exception

The `AArch32.UndefinedFault()` pseudocode procedure determines whether the Undefined Instruction exception is taken to AArch32 state. If it is taken to AArch32 state, the `AArch32.TakeUndefInstrException()` pseudocode procedure describes how the PE takes the exception.

An Undefined Instruction exception is taken to an Exception level using AArch64 if either:

- It is generated in User mode when EL1 is using AArch64.
- It is generated in User mode when EL2 is enabled in the current Security state and is using AArch64 and the value of HCR_EL2.TGE is 1.

Conditional execution of undefined instructions

The conditional execution rules described in *Conditional execution* on page F2-4081 apply to all instructions. This includes undefined instructions and other instructions that would cause entry to the Undefined Instruction exception.

If such an instruction fails its condition check, the behavior depends on the potential cause of entry to the Undefined Instruction exception, as follows:

- If the potential cause is the execution of the instruction itself and depends on data values used by the instruction, the instruction executes as a `NOP` and does not cause an Undefined Instruction exception.
- In the following cases, it is IMPLEMENTATION DEFINED whether the instruction executes as a `NOP` or causes an Undefined Instruction exception:
  - The potential cause is the execution of an earlier System register access instruction, floating-point instruction, or Advanced SIMD instruction.
  - The potential cause is the execution of the instruction itself without dependence on the data values used by the instruction.

  An implementation must handle all such cases in the same way.

Note

Before Armv7, all implementations executed any instruction that failed its condition check as a `NOP`, even if it would otherwise have caused an Undefined Instruction exception. An Undefined Instruction handler written for these implementations might assume without checking that the undefined instruction passed its condition check. Such an Undefined Instruction handler is likely to need rewriting, to check the condition is passed, before it functions correctly on all AArch32 implementations.

Interaction of UNDEFINED instruction behavior with UNPREDICTABLE or CONSTRAINED UNPREDICTABLE instruction behavior

If this manual describes an instruction as both:

- UNPREDICTABLE and UNDEFINED then the instruction is UNPREDICTABLE.
- CONSTRAINED UNPREDICTABLE and UNDEFINED then the instruction is CONSTRAINED UNPREDICTABLE.

Note

An example of this is where both:

- An instruction, or instruction class, is made UNDEFINED by some general principle, or by a configuration field.
- A particular encoding of that instruction or instruction class is specified as CONSTRAINED UNPREDICTABLE.

G1.16.2 Monitor Trap exception

The Monitor Trap exception is implemented only as part of EL3, and can be generated only if EL3 is using AArch32.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

The Monitor Trap exception is taken using offset `0x04` in the Monitor vector table. In the other vector tables, this offset is used for the Undefined Instruction exception. See *Undefined Instruction exception* on page G1-5738 and *The vector tables and exception offsets* on page G1-5704.

A Monitor Trap exception is generated if the PE is running in a mode other than Monitor mode, and commits for execution a `WFI` or `WFE` instruction that would otherwise cause suspension of execution when:

- In the case of the `WFI` instruction, the value of the SCR.TWI bit is 1.
- In the case of the `WFE` instruction, the value of the SCR.TWE bit is 1.

Note

Since a `WFE` or `WFI` can complete at any time, even without a Wakeup event, the traps on `WFE` of `WFI` are not guaranteed to be taken, even if the `WFE` or `WFI` is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

The preferred return address for a Monitor Trap exception is the address of the instruction that generated the exception. The exception return uses the SPSR and LR_mon values generated by the exception entry, as follows:

- If SPSR.T is 0, indicating that the exception occurred in A32 state, the return uses an exception return instruction with a subtraction of 4.
- If SPSR.T is 1, indicating that the exception occurred in T32 state, the return uses an exception return instruction with a subtraction of 2.

For more information, see *Exception return to an Exception level using AArch32* on page G1-5725.

The PE mode to which the Monitor Trap exception is taken

When EL3 is using AArch32, a Monitor Trap exception is taken to Monitor mode, using a vector offset of `0x04` from the Monitor exception base address.

Pseudocode description of taking the Monitor Trap exception

The `AArch32.TakeMonitorTrapException()` pseudocode procedure describes how the PE takes the exception.

G1.16.3 Hyp Trap exception

The Hyp Trap exception provides the standard mechanism for trapping Guest OS functions to the hypervisor.

The Hyp Trap exception is implemented only as part of EL2 and can be generated only if EL2 is using AArch32.

A Hyp Trap exception is generated if the PE is running in a Non-secure mode other than Hyp mode, and commits for execution an instruction that is trapped to Hyp mode. Instruction traps are enabled by setting bits to 1 in the HCR, HCPTR, HDCR, or HSTR. For more information see *EL2 configurable controls* on page G1-5787.

Traps to Hyp mode never apply in Secure state, regardless of the value of the SCR.NS bit.

The preferred return address for a Hyp Trap exception is the address of the trapped instruction. The exception return is performed by an `ERET` instruction, using the SPSR and ELR_hyp values generated by the exception entry.

Note

The SPSR and ELR_hyp values generated on exception entry can be used, without modification, for an exception return to re-execute the trapped instruction. If the exception handler emulates the trapped instruction, and must return to the following instruction, the emulation of the instruction must include modifying ELR_hyp, and possibly updating SPSR_hyp.

When the PE enters the handler for a Hyp Trap exception, the HSR holds syndrome information for the exception. For more information see *Use of the HSR* on page G5-6038.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The PE mode to which the Hyp Trap exception is taken

A Hyp Trap exception is taken to Hyp mode, using a vector offset of `0x14` from the Hyp exception base address.

Pseudocode description of taking the Hyp Trap exception

The `AArch32.TakeHypTrapException()` pseudocode procedure describes how the PE takes the exception.

G1.16.4 Supervisor Call (SVC) exception

The Supervisor Call instruction, `SVC`, requests a supervisor function, typically to request an operating system function. When EL1 is using AArch32, executing an `SVC` instruction causes the PE to enter Supervisor mode. For more information, see *SVC* on page F5-4847.

Note

In an implementation that includes EL2, when EL2 is using AArch32:

- When an `SVC` instruction is executed in Hyp mode, the Supervisor Call exception is taken to Hyp mode. For more information see *SVC* on page F5-4847.
- When the HCR.TGE bit is set to 1, the Supervisor Call exception generated by execution of an `SVC` instruction in Non-secure User mode is routed to Hyp mode. For more information, see *Supervisor Call exception, when the value of HCR.TGE is 1* on page G1-5719.

By default, a Supervisor Call exception that is taken to AArch32 state is taken to Supervisor mode, but a Supervisor Call exception can be taken to EL2, meaning it is taken to Hyp mode if EL2 is using AArch32, see *The PE mode to which the Supervisor Call exception is taken*.

The preferred return address for a Supervisor Call exception is the address of the next instruction after the `SVC` instruction. For an exception taken to AArch32 state, this return is performed as follows:

- If returning from Secure or Non-secure Supervisor mode, the exception return uses the SPSR and LR_svc values generated by the exception entry, in an exception return instruction without subtraction.
- If returning from Hyp mode, the exception return is performed by an `ERET` instruction, using the SPSR and ELR_hyp values generated by the exception entry.

For more information, see *Exception return to an Exception level using AArch32* on page G1-5725.

The PE mode to which the Supervisor Call exception is taken

Figure G1-5 on page G1-5743 shows how the implementation, state, and configuration options determine the PE mode to which a Supervisor Call exception is taken, when the exception is taken to an Exception level that is using AArch32.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

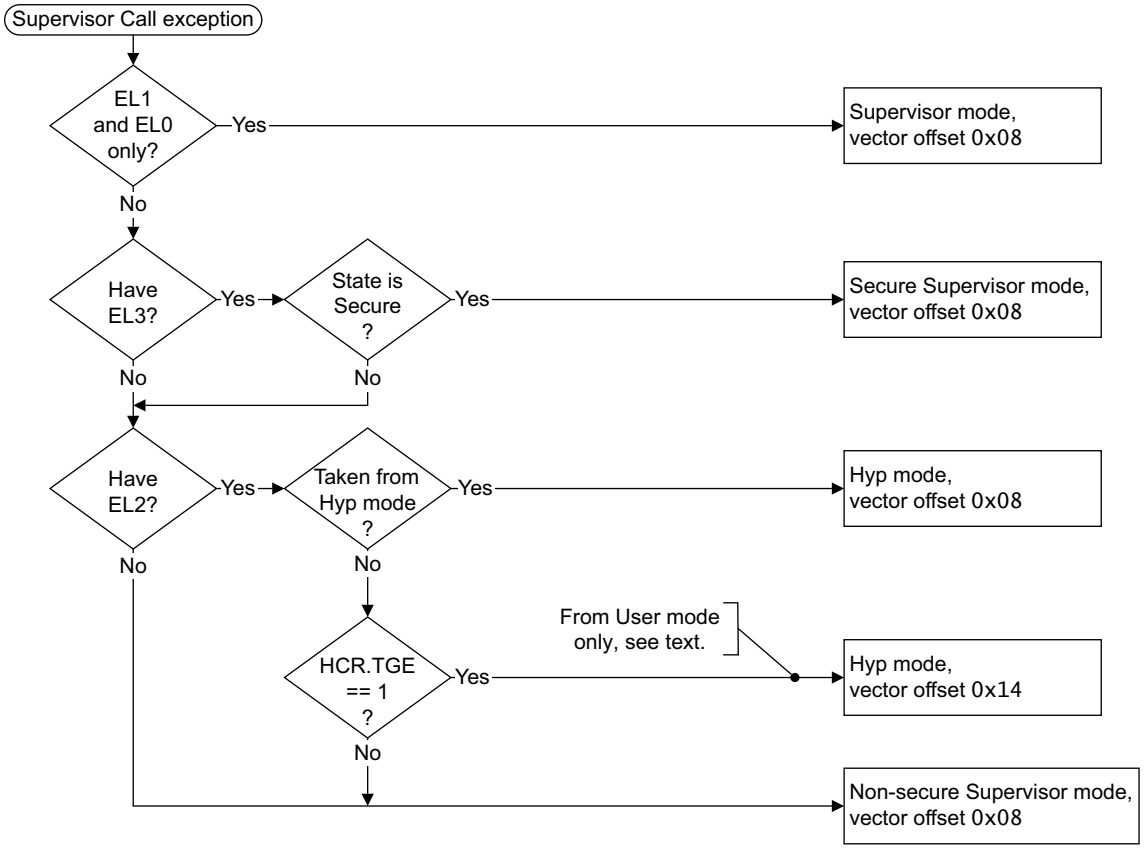


Figure G1-5 The PE mode the Supervisor Call exception is taken to in AArch32 state

See also *UNPREDICTABLE cases when the value of HCR.TGE is 1* on page G1-5716.

Pseudocode description of taking the Supervisor Call exception

The `AArch32.CallSupervisor()` pseudocode procedure determines whether the Supervisor Call exception is taken to AArch32 state. If it is taken to AArch32 state, the `AArch32.TakeSVCException()` pseudocode procedure describes how the PE takes the exception.

An Supervisor Call exception is taken to an Exception level using AArch64 if either:

- It is generated by executing an `SVC` instruction in User mode when EL1 is using AArch64.
- It is generated by executing an `SVC` instruction in Non-secure User mode when EL2 is using AArch64 and the value of HCR_EL2.TGE is 1.

G1.16.5 Secure Monitor Call (SMC) exception

The Secure Monitor Call exception is implemented only as part of EL3. When EL3 is using AArch32, the exception is taken to Monitor mode.

The Secure Monitor Call instruction, `SMC`, requests a Secure Monitor function. When EL3 is using AArch32, executing an `SMC` instruction causes the PE to enter Monitor mode. For more information, see *SMC* on page F5-4694.

Note

- In an implementation that includes EL2, execution of an `SMC` instruction in a Non-secure EL1 mode can be trapped to EL2. When EL2 is using AArch32, this means that when HCR.TSC 1, execution of an `SMC` instruction in a Non-secure EL1 mode generates a Hyp Trap Exception that is taken to Hyp mode. For more information see *Traps to Hyp mode of Non-secure EL1 execution of SMC instructions* on page G1-5794.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The Operation pseudocode in the description of the AArch32 `SMC` instruction, in *SMC* on page F5-4694, identifies cases where execution of the instruction generates an exception that is taken to EL3 using AArch64.

The preferred return address for a Secure Monitor Call exception is the address of the next instruction after the `SMC` instruction. For an exception taken to AArch32 state, this return is performed using the SPSR and LR_mon values generated by the exception entry, using an exception return instruction without a subtraction.

For more information, see *Exception return to an Exception level using AArch32* on page G1-5725.

Note

For an exception taken to AArch32 state, the exception handler can return to the `SMC` instruction itself by returning using a subtraction of 4, without any adjustment to the SPSR.IT[7:0] bits. If it does this, the return occurs, then asynchronous exceptions might occur and be handled, then the `SMC` instruction is re-executed and another Secure Monitor Call exception occurs.

This relies on:

- The `SMC` instruction being used correctly, either outside an IT block or as the last instruction in an IT block, so that the SPSR.IT[7:0] bits indicate unconditional execution.
- The Secure Monitor Call handler not changing the result of the original conditional execution test for the `SMC` instruction.

The PE mode to which the Secure Monitor Call exception is taken

The Secure Monitor Call exception is supported only as part of EL3. When EL3 is using AArch32, a Secure Monitor Call exception is taken to Monitor mode, using vector offset `0x08` from the Monitor exception base address.

Note

- An `SMC` instruction that is trapped to Hyp mode because HCR.TSC is set to 1 generates a Hyp Trap exception, see *The PE mode to which the Hyp Trap exception is taken* on page G1-5742.
- If EL3 is using AArch64 then *Security behavior in Exception levels using AArch32 when EL2 or EL3 are using AArch64* on page G1-5715 describes the effect of executing an `SMC` instruction at an Exception level that is using EL1.

Pseudocode description of taking the Secure Monitor Call exception

The `AArch32.TakeSMCException()` pseudocode procedure describes how the PE takes the exception when the exception is taken to an Exception level that is using AArch32.

G1.16.6 Hypervisor Call (HVC) exception

The Hypervisor Call exception is implemented only as part of EL2.

The Hypervisor Call instruction, `HVC`, requests a hypervisor function. When EL2 is using AArch32, executing an `HVC` instruction generates a Hypervisor Call exception that is taken to Hyp mode. For more information, see *HVC* on page F5-4375.

Note

- Execution of `HVC` instructions is disabled when the value of SCR.HCE is 0. Descriptions of `HVC` instruction execution elsewhere in this section assume the *Effective value* of SCR.HCE is 1.
- When EL2 is using AArch64 an `HVC` instruction executed in a Non-secure EL1 mode generates an exception that is taken to EL2 using AArch64. *Exception classes and the ESR_ELx syndrome registers* on page D1-2342 describes how this exception is reported in ESR_EL2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The preferred return address for a Hypervisor Call exception is the address of the next instruction after the HVC instruction. The exception return is performed by an ERET instruction, using the SPSR and ELR_hyp values generated by the exception entry.

For more information, see *Exception return to an Exception level using AArch32* on page G1-5725.

When EL2 is using AArch32, executing an HVC instruction transfers the immediate argument of the instruction to the HSR. The exception handler retrieves the argument from the HSR, and therefore does not have to access the original HVC instruction. For more information see *Use of the HSR* on page G5-6038.

The PE mode to which the Hypervisor Call exception is taken

The Hypervisor Call exception is supported only as part of EL2. When EL2 is using AArch32, a Hypervisor Call exception is taken to Hyp mode, using a vector offset that depends on the mode from which the exception is taken, as Figure G1-6 shows. This offset is from the Hyp exception base address.

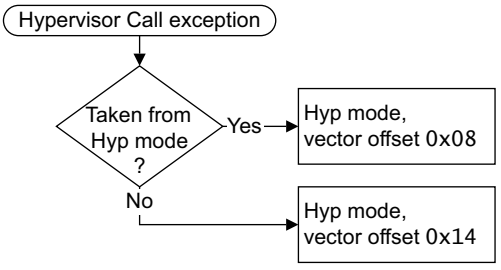


Figure G1-6 The PE mode the Hypervisor Call exception is taken to in AArch32 state

Pseudocode description of taking the Hypervisor Call exception

The `AArch32.CallHypervisor()` pseudocode procedure determines whether the valid execution of an HVC instruction in AArch32 state generates an exception that is taken to EL2 using AArch64, or generates a Hypervisor Call exception taken to Hyp mode. The `AArch32.TakeHVCException()` pseudocode procedure describes how the PE takes a Hypervisor Call exception.

G1.16.7 Prefetch Abort exception

A Prefetch Abort exception can be generated by:

- A synchronous memory abort on an instruction fetch.

  Note

  Asynchronous External aborts on instruction fetches are reported as SError interrupts using the Data Abort exception, see *Data Abort exception* on page G1-5749.

  A Prefetch Abort exception entry is synchronous to the instruction whose fetch aborted.

  For more information about memory aborts see *VMSAv8-32 memory aborts* on page G5-6012.

- A Breakpoint, Vector Catch or Breakpoint Instruction exception, see Chapter G2 *AArch32 Self-hosted Debug*.

Note

If an implementation fetches instructions speculatively, it must handle a synchronous abort on such an instruction fetch by:

- Generating a Prefetch Abort exception only if the instruction would be executed in a simple sequential execution of the program.
- Ignoring the abort if the instruction would not be executed in a simple sequential execution of the program.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

By default, when EL1 is using AArch32, a Prefetch Abort exception is taken to Abort mode, but a Prefetch Abort exception can be taken to:

- EL2, meaning it is taken to Hyp mode if EL2 is using AArch32.
- EL3, meaning it is taken to Monitor mode if EL3 is using AArch32.

For more information:

- About cases where the Prefetch Abort exception is taken to an Exception level that is using AArch32, see *The PE mode to which the Prefetch Abort exception is taken* on page G1-5747.
- About cases where the Prefetch Abort generates an exception that is taken to an Exception level that is using AArch64, see *Pseudocode description of taking the Prefetch Abort exception* on page G1-5749.

The preferred return address for a Prefetch Abort exception is the address of the aborted instruction. For an exception taken to AArch32 state this return is performed as follows:

- If returning from a mode other than Hyp mode, using the SPSR and LR values generated by the exception entry, using an exception return instruction with a subtraction of 4. This means using:
  - SPSR_abt and LR_abt if returning from Abort mode.
  - SPSR_mon and LR_mon if returning from Monitor mode.
- If returning from Hyp mode, using the SPSR_hyp and ELR_hyp values generated by the exception entry, using an `ERET` instruction.

For more information about the handling of Prefetch Abort exceptions in AArch32 state see *Exception return to an Exception level using AArch32* on page G1-5725.

Prefetch Abort exception reporting a PC alignment fault exception

A PC alignment fault exception that is taken to an Exception level that is using AArch32 is reported as a Prefetch Abort exception, and:

If the exception is taken to EL1 using AArch32 or EL3 using AArch32

- The IFSR indicates the cause of the exception:
  - If the value of TTBCR.EAE is 0, IFSR.FS takes the value `0b00001`.
  - If the value of TTBCR.EAE is 1, IFSR.STATUS takes the value `0b100001`.
- IFAR holds the value of the address that faulted, including the misaligned low order bit or bits.
- R14_abt holds the address that faulted, including the misaligned low order bit or bits, with the standard offset for a Prefetch Abort exception.

If the exception is taken to EL2 using AArch32

- HSR.EC takes the value `0b100010`.
- HSR.IL is UNKNOWN.
- HSR.ISS is RES0.
- HIFAR and ELR_hyp each hold the value of the address that faulted, including the misaligned low order bit or bits.

For a PC alignment fault exception taken to an Exception level that is using AArch32:

- If the exception occurred because of the CONSTRAINED UNPREDICTABLE behavior of a branch to an unaligned PC value, as described in *Branching to an unaligned PC* on page K1-7832, then bit[0] of the faulting address is forced to zero, and therefore the misalignment is because the value of bit[1] of this address is 1.
- If the exception occurred on an exit from Debug state, as described in *Exiting Debug state* on page H2-6963, then it is CONSTRAINED UNPREDICTABLE whether bit[0] of the faulting address is forced to zero.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The PE mode to which the Prefetch Abort exception is taken

Figure G1-7 on page G1-5748 shows how the implementation, state, and configuration options determine the PE mode to which a Prefetch Abort exception is taken, when the exception is taken to an Exception level that is using AArch32.

Note

In this figure, the Effective value of HCR2.TEA is 0 if the The Reliability, Availability, and Serviceability Extension (RAS Extension) is not implemented.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

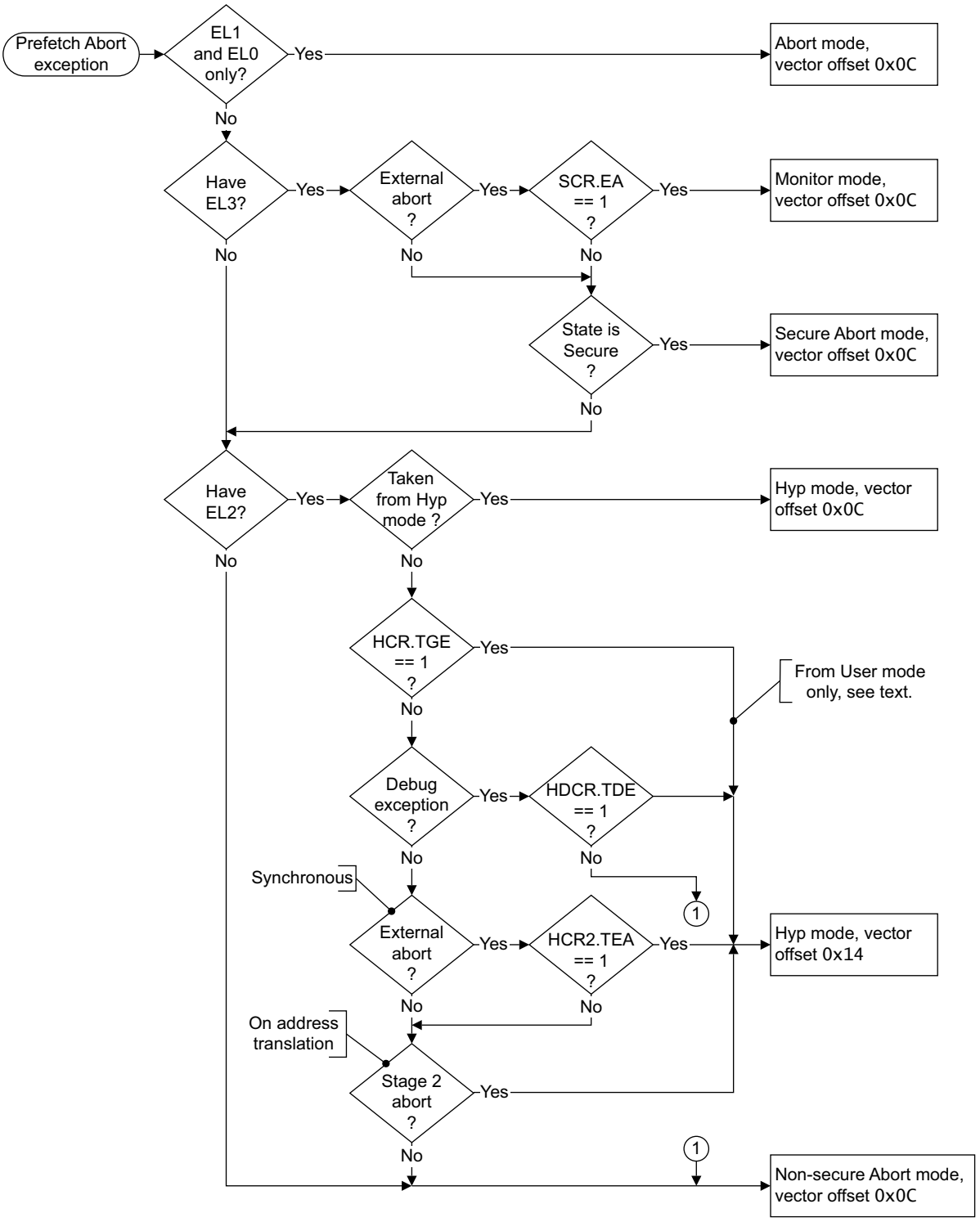


Figure G1-7 The PE mode the Prefetch Abort exception is taken to in AArch32 state

See also *UNPREDICTABLE cases when the value of HCR.TGE is 1* on page G1-5716.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Pseudocode description of taking the Prefetch Abort exception

The `AArch32.Abort()` pseudocode function determines whether the Prefetch Abort condition generates an exception that is taken to an Exception level that is using AArch64, or generates a Prefetch Abort exception that is taken in AArch32 state. When the exception is taken in AArch32 state, the `AArch32.TakePrefetchAbortException()` pseudocode procedure describes how the PE takes the exception.

The exception is taken to an Exception level using AArch64 if one of the following applies:

- The exception is generated in User mode when EL1 is using AArch64.
- The implementation includes EL2, EL2 is using AArch64, and one of the following applies:
  - The value of HCR_EL2.TGE is 1 and the exception is generated in Non-secure User mode.
  - The value of MDCR_EL2.TDE is 1 and the exception is generated by a Debug exception in a Non-secure EL1 or Non-secure EL0 mode.
  - The exception is generated by a stage 2 fault during a stage 1 translation table walk using the AArch32 Non-secure EL1&0 translation regime.
- The implementation includes EL3, EL3 is using AArch64, the value of SCR_EL3.EA is 1. and the exception is generated by an External abort in AArch32 state.

G1.16.8 Data Abort exception

In AArch32 state, a Data Abort exception can be generated by:

- A synchronous abort on a data read or write memory access. Exception entry is synchronous to the instruction that generated the memory access.
- An SError interrupt. The SError interrupt might be caused by an External abort on a memory access, which can be any of:
  - A data read or write access.
  - An instruction fetch.
  - In a VMSA memory system, a translation table access.

  Exception entry occurs asynchronously.

  As described in *Asynchronous exception masking controls* on page G1-5733, SError interrupts can be masked. When this happens, a generated SError interrupt is not taken until it is not masked.
- A watchpoint, see *Watchpoint exceptions* on page G2-5855.

By default, when EL1 is using AArch32 a Data Abort exception is taken to Abort mode, but a Data Abort exception can be taken to:

- EL2, meaning it is taken to Hyp mode if EL2 is using AArch32.
- EL3, meaning it is taken to Monitor mode if EL3 is using AArch32.

For more information:

- About cases where the Data Abort exception is taken to an Exception level that is using AArch32 see *The PE mode to which the Data Abort exception is taken* on page G1-5750.
- About memory aborts in AArch32 state see *VMSAv8-32 memory aborts* on page G5-6012.
- About cases where the Data Abort generates an exception that is taken to an Exception level that is using AArch64 see *Pseudocode description of taking the Data Abort exception* on page G1-5752.

The preferred return address for a Data Abort exception is the address of the instruction that generated the aborting memory access, or the address of the instruction following the instruction boundary at which an SError interrupt exception was taken. For an exception taken to AArch32 state, this return is performed as follows:

- If returning from a mode other than Hyp mode, using the SPSR and LR values generated by the exception entry, using an exception return instruction with a subtraction of 8. This means using:
  - SPSR_abt and LR_abt if returning from Abort mode.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

  — SPSR_mon and LR_mon if returning from Monitor mode.

- If returning from Hyp mode, using the SPSR_hyp and ELR_hyp values generated by the exception entry, using an ERET instruction.

For more information about the handling of Data Abort exceptions in AArch32 state see *Exception return to an Exception level using AArch32* on page G1-5725.

The PE mode to which the Data Abort exception is taken

Figure G1-8 on page G1-5751 shows the determination of the mode to which a Data Abort exception is taken when the exception is taken to an Exception level that is using AArch32.

Note

In this figure, the Effective value of HCR2.TEA is 0 if the The Reliability, Availability, and Serviceability Extension (RAS Extension) is not implemented.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

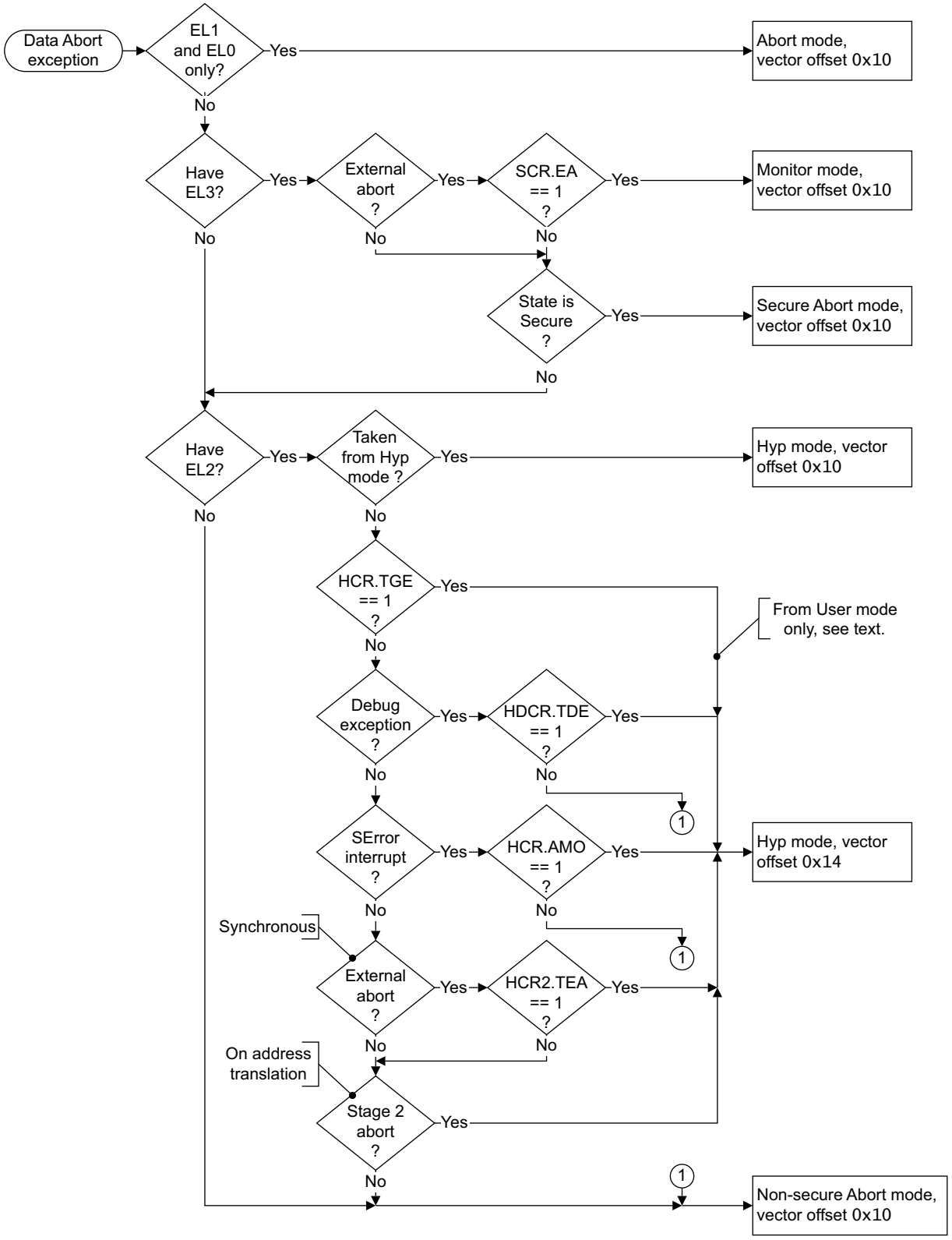


Figure G1-8 The PE mode the Data Abort exception is taken to in AArch32 state

See also *UNPREDICTABLE cases when the value of HCR.TGE is 1* on page G1-5716.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Pseudocode description of taking the Data Abort exception

The `AArch32.Abort()` pseudocode function determines whether the Data Abort condition generates an exception that is taken to an Exception level that is using AArch64, or generates a Data Abort exception that is taken in AArch32 state. When the exception is taken in AArch32 state, the `AArch32.TakeDataAbortException()` pseudocode procedure describes how the PE takes the exception.

The exception is taken to an Exception level using AArch64 if one of the following applies:

- The exception is generated in User mode when EL1 is using AArch64.
- The implementation includes EL2, EL2 is using AArch64, and one of the following applies:
  - The value of HCR_EL2.TGE is 1 and the exception is generated in Non-secure User mode.
  - The value of MDCR_EL2.TDE is 1 and the exception is generated by a Debug exception in a Non-secure EL1 or Non-secure EL0 mode.
  - The exception is generated by a stage 2 fault during a stage 1 translation table walk using the AArch32 Non-secure EL1&0 translation regime.
- The implementation includes EL3, EL3 is using AArch64, the value of SCR_EL3.EA is 1. and the exception is generated by an External abort in AArch32 state.

Effects of data-aborted instructions

An instruction that accesses data memory can modify memory by storing one or more values. If the execution of such an instruction generates a Data Abort exception, or causes Debug state entry because of a watchpoint set on the location, the value of each memory location that the instruction stores to is:

- Unchanged for any location for which one of the following applies:
  - An Alignment fault is generated.
  - An MMU fault is generated.
  - A Watchpoint is generated.
  - An External abort is generated, if that External abort is taken synchronously.
- UNKNOWN for any location for which no exception and no debug event is generated.

If the access to a memory location generates an External abort that is taken asynchronously, it is outside the scope of the architecture to define the effect of the store on that memory location, because this depends on the system-specific nature of the External abort. However, in general, Arm recommends that such locations are unchanged.

For External aborts and Watchpoints, where in principle faulting could be identified at byte or halfword granularity, the size of a location in this definition is the size for which a memory access is single-copy atomic.

In AArch32 state, instructions that access data memory can modify registers in the following ways:

- By loading values into one or more of the general-purpose registers. The registers loaded can include the PC.
- By loading values into one or more of the registers in the Advanced SIMD and floating-point register file.
- By specifying *base register writeback*, in which the base register used in the address calculation has a modified value written to it. All instructions that support base register writeback have CONSTRAINED UNPREDICTABLE results if base register writeback is specified with the PC as the base register. Only general-purpose registers can be modified reliably in this way.
- By a direct transfer to or from the Debug Communication Channel (DCC) register, using the LDC and STC instructions. For more information see Chapter H4 *The Debug Communication Channel and Instruction Transfer Register*.

  If the instruction that accesses the DCC registers is an LDC or STC instruction, UNKNOWN values are left in the Data Transfer Register and DCC flow-control flags.
- By modifying PSTATE.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the execution of such an instruction generates a synchronous Data Abort exception, the following rules determine the values left in these registers:

- On entry to the Data Abort exception handler:
  - The PC value is the Data Abort vector address, see *Exception vectors and the exception base address* on page G1-5703.
  - The LR_abt value is determined from the address of the aborted instruction.

  Neither value is affected by the results of any load specified by the instruction.
- The base register is restored to its original value if either:
  - The aborted instruction is a load and the list of registers to be loaded includes the base register.
  - The base register is being written back.
- If the instruction only loads one general-purpose register the value in that register is unchanged.
- If the instruction loads more than one general-purpose register, UNKNOWN values are left in destination registers other than the PC and the base register of the instruction.
- If the instruction affects any registers in the Advanced SIMD and floating-point register file, UNKNOWN values are left in the registers that are affected.
- PSTATE bits that are not defined as updated on exception entry retain their current value.
- If the instruction is a `STREX`, `STREXB`, `STREXH`, or `STREXD`, `<Rd>` is not updated.

After taking a Data Abort exception, the state of the Exclusives monitors is UNKNOWN. Therefore, Arm strongly recommends that the abort handler performs a `CLREX` instruction, or a dummy `STREX` instruction, to clear the Exclusives monitor state.

An External abort might signal a data corruption to the PE. For example a memory location might have been corrupted. The error that caused the External abort might have been propagated. The RAS Extension provides mechanisms for software to determine the extent of the corruption and contain propagation of the error. For more information, see the *ARM® Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for the ARMv8-A architecture profile*.

The Arm abort model

The abort model used by an Arm PE is described as a *Base Restored Abort Model*. This means that if a synchronous Data Abort exception is generated by executing an instruction that specifies base register writeback, the value in the base register is unchanged.

The abort model applies uniformly across all instructions.

G1.16.9 Virtual SError interrupt exception

The Virtual SError interrupt exception is implemented only as part of EL2 is enabled in the current Security state.

A Virtual SError interrupt exception is generated in AArch32 state if all of the following apply:

- The PE is in a mode other than Hyp mode.
- The value of PSTATE.A is 0.
- Either:
  - EL2 is using AArch32 and the values of the HCR.{TGE, AMO, VA} bits are {0, 1, 1}.
  - EL2 is using AArch64 and the values of the HCR_EL2.{TGE, AMO, VA} bits are {0, 1, 1}.

The preferred return address for a Virtual SError interrupt exception is the address of the instruction immediately after the instruction boundary where the exception was taken. For an exception taken to AArch32 state, this return is performed using the SPSR and LR_abt values generated by the exception entry, using an exception return instruction without subtraction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The PE mode to which the Virtual SError interrupt exception is taken

The Virtual SError interrupt exception is taken using a vector offset of `0x10` from the Non-secure exception base address.

The conditions for generating a Virtual SError interrupt exception in AArch32 state mean the exception is:

- Taken from a EL1 or EL0 mode.
- Taken to Abort mode if EL1 is using AArch32.
- Taken to EL1, when EL0 is using AArch32 and EL1 is using AArch64.

For more information see *Virtual exceptions when an implementation includes EL2* on page G1-5730.

Note

Because a Virtual SError interrupt exception taken to AArch32 state is always taken to Abort mode, on exception entry the preferred return address is always saved to LR_abt.

Pseudocode description of taking the Virtual SError interrupt exception

The `AArch32.TakeVirtualSErrorException()` pseudocode procedure describes how the PE takes the exception.

G1.16.10 IRQ exception

The IRQ exception is generated by IMPLEMENTATION DEFINED means. Typically this is by asserting an IRQ interrupt request input to the PE.

When an IRQ exception is taken, exception entry is precise to an instruction boundary.

As described in *Asynchronous exception masking controls* on page G1-5733, IRQ exceptions can be masked. When this happens, a generated IRQ exception is not taken until it is not masked.

By default, when EL1 is using AArch32, an IRQ exception is taken to IRQ mode, but an IRQ exception can be taken to:

- EL2, meaning it is taken to Hyp mode if EL2 is using AArch32.
- EL3, meaning it is taken to Monitor mode if EL3 is using AArch32.

For more information:

- About cases where the exception is taken to an Exception level using AArch32 see *The PE mode to which the physical IRQ exception is taken* on page G1-5755.
- About cases where the exception is taken to an Exception level using AArch64 see *Pseudocode description of taking the physical IRQ exception* on page G1-5755.

The preferred return address for an IRQ exception is the address of the instruction following the instruction boundary at which the exception was taken. For an exception taken to AArch32 state this return is performed as follows:

- If returning from a mode other than Hyp mode, using the SPSR and LR values generated by the exception entry, using an exception return instruction with a subtraction of 4. This means using:
  - SPSR_irq and LR_irq if returning from IRQ mode.
  - SPSR_mon and LR_mon if returning from Monitor mode.
- If returning from Hyp mode, using the SPSR_hyp and ELR_hyp values generated by the exception entry, using an `ERET` instruction.

For more information, see *Exception return to an Exception level using AArch32* on page G1-5725.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The PE mode to which the physical IRQ exception is taken

Figure G1-9 shows how the implementation, state, and configuration options determine the mode to which an IRQ exception is taken when the exception is taken to an Exception level that is using AArch32.

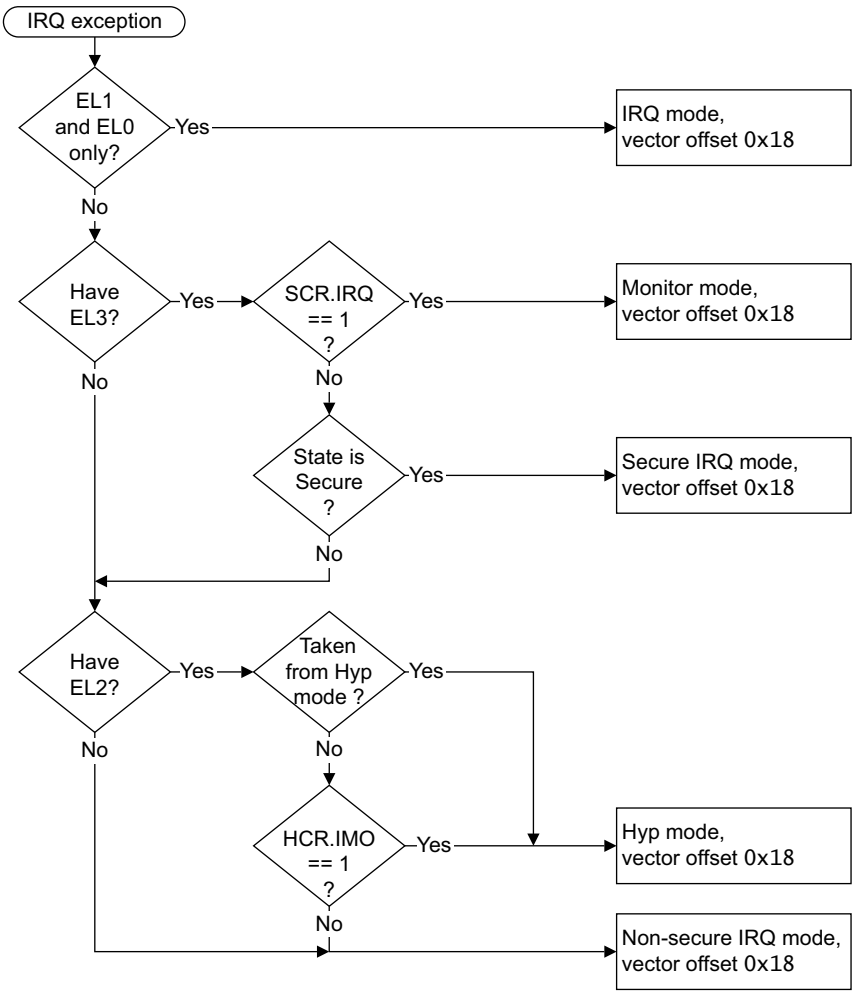


Figure G1-9 The PE mode the IRQ exception is taken to in AArch32 state

Pseudocode description of taking the physical IRQ exception

The `AArch32.TakePhysicalIRQException()` pseudocode procedure describes how the PE takes the exception. This procedure includes the case where the exception is taken to an Exception level that is using AArch64. This happens if one of the following applies:

- The exception is taken from User mode and EL1 is using AArch64. The Exception is taken to EL1 using AArch64.
- The exception is taken from User mode, EL2 is implemented in the current Security state and using AArch64, and the value of HCR_EL2.TGE is 1. The Exception is taken to EL2 using AArch64.
- The exception is taken from EL0 or EL1 mode, EL2 is implemented in the current Security state and using AArch64, and the value of HCR_EL2.IMO is 1. The Exception is taken to EL2 using AArch64.
- The exception is taken from a PE mode other than Monitor mode, EL3 is implemented and using AArch64, and the value of SCR_EL3.IRQ is 1. The Exception is taken to EL3 using AArch64.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G1.16.11 Virtual IRQ exception

The Virtual IRQ exception is implemented only as part of EL2, if EL2 is enabled in the current Security state.

A Virtual IRQ exception is generated in AArch32 state if all of the following apply:

- The PE is in a mode other than Hyp mode.
- The value of PSTATE.I is 0.
- Either:
  - EL2 is using AArch32 and the value of HCR.{TGE, IMO} is {0, 1}.
  - EL2 is using AArch64 and the value of HCR_EL2.{TGE, IMO} is {0, 1}.
- One of the following applies:
  - EL2 is using AArch32 and the value of HCR.VI is 1.
  - EL2 is using AArch64 and the value of HCR_EL2.VI is 1.
  - A Virtual IRQ exception is generated by an IMPLEMENTATION DEFINED mechanism.

The preferred return address for a Virtual IRQ exception is the address of the instruction immediately after the instruction boundary where the exception was taken. For an exception taken to AArch32 state this return is performed using the SPSR and LR_irq values generated by the exception entry, using an exception return instruction with a subtraction of 4.

The PE mode to which the Virtual IRQ exception is taken

The Virtual IRQ exception uses a vector offset of `0x18`.

The conditions for generating a Virtual IRQ exception in AArch32 state mean the exception is:

- Taken from an EL1 or EL0 mode.
- Taken to IRQ mode if EL1 is using AArch32.
- Taken to EL1 if EL0 is using AArch32 and EL1 is using AArch64.

For more information see *Virtual exceptions when an implementation includes EL2* on page G1-5730.

Pseudocode description of taking the Virtual IRQ exception

The `AArch32.TakeVirtualIRQException()` pseudocode procedure describes how the PE takes the exception.

G1.16.12 FIQ exception

The FIQ exception is generated by IMPLEMENTATION DEFINED means. Typically this is by asserting an FIQ interrupt request input to the PE.

When an FIQ exception is taken, exception entry is precise to an instruction boundary.

As described in *Asynchronous exception masking controls* on page G1-5733, FIQ exceptions can be masked. When this happens, a generated FIQ exception is not taken until it is not masked.

By default, an FIQ exception is taken to FIQ mode, but an FIQ exception can be taken to:

- EL2, meaning it is taken to Hyp mode if EL2 is using AArch32.
- EL3, meaning it is taken to Monitor mode if EL3 is using AArch32.

For more information:

- About cases where the exception is taken to an Exception level using AArch32 see *The PE mode to which the physical FIQ exception is taken* on page G1-5757.
- About cases where the exception is taken to an Exception level using AArch64 see *Pseudocode description of taking the FIQ exception* on page G1-5758.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The preferred return address for an FIQ exception is the address of the instruction following the instruction boundary at which the exception was taken. For an exception taken to AArch32 state this return is performed as follows:

- If returning from a mode other than Hyp mode, using the SPSR and LR values generated by the exception entry, using an exception return instruction with a subtraction of 4. This means using:
  - SPSR_fiq and LR_fiq if returning from FIQ mode.
  - SPSR_mon and LR_mon if returning from Monitor mode.
- If returning from Hyp mode, using the SPSR_hyp and ELR_hyp values generated by the exception entry, using an `ERET` instruction.

For more information see *Exception return to an Exception level using AArch32* on page G1-5725.

The PE mode to which the physical FIQ exception is taken

Figure G1-9 on page G1-5755 shows how the implementation, state, and configuration options determine the PE mode to which an FIQ exception is taken when the exception is taken to an Exception level that is using AArch32.

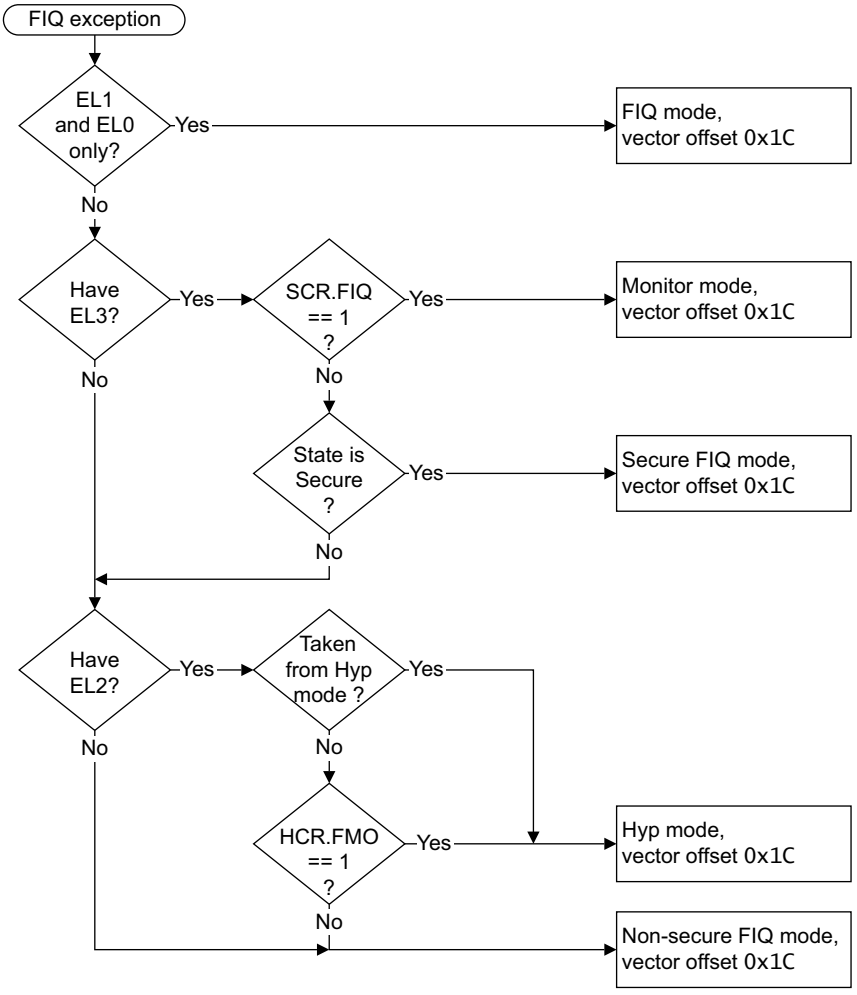


Figure G1-10 The PE mode the FIQ exception is taken to in AArch32 state

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Pseudocode description of taking the FIQ exception

The `AArch32.TakePhysicalFIQException()` pseudocode procedure describes how the PE takes the exception. This procedure includes the case where the exception is taken to an Exception level that is using AArch64. This happens if one of the following applies:

- The exception is taken from User mode and EL1 is using AArch64. The Exception is taken to EL1 using AArch64.
- The exception is taken from User mode, EL2 is implemented in the current Security state and using AArch64, and the value of HCR_EL2.TGE is 1. The Exception is taken to EL2 using AArch64.
- The exception is taken from an EL0 or EL1 mode, EL2 is implemented in the current Security state and using AArch64, and the value of HCR_EL2.FMO is 1. The Exception is taken to EL2 using AArch64.
- The exception is taken from a PE mode other than Monitor mode, EL3 is implemented and using AArch64, and the value of SCR_EL3.FIQ is 1. The Exception is taken to EL3 using AArch64.

G1.16.13 Virtual FIQ exception

The Virtual FIQ exception is implemented only as part of EL2, if EL2 is enabled in the current Security state.

A Virtual FIQ exception is generated in AArch32 state if all of the following apply:

- The PE is in a mode other than Hyp mode.
- The value of PSTATE.F is 0.
- Either:
  - EL2 is using AArch32 and the value of HCR.{TGE, FMO} is {0, 1}.
  - EL2 is using AArch64 and the value of HCR_EL2.{TGE, FMO} is {0, 1}.
- One of the following applies:
  - EL2 is using AArch32 and the value of HCR.VF is 1.
  - EL2 is using AArch64 and the value of HCR_EL2.VF is 1.
  - A Virtual FIQ exception is generated by an IMPLEMENTATION DEFINED mechanism.

The preferred return address for a Virtual FIQ exception is the address of the instruction immediately after the instruction boundary where the exception was taken. For an exception taken to AArch32 state this return is performed using the SPSR and LR_irq values generated by the exception entry, using an exception return instruction with a subtraction of 4.

The PE mode to which the Virtual FIQ exception is taken

The Virtual FIQ exception is taken using a vector offset of `0x1C`.

The conditions for generating a Virtual FIQ exception in AArch32 state mean the exception is:

- Taken from EL1 or EL0.
- Taken to FIQ mode if EL1 is using AArch32.
- Taken to EL1 if EL0 is using AArch32 and EL1 is using AArch64.

For more information see *Virtual exceptions when an implementation includes EL2* on page G1-5730.

Pseudocode description of taking the Virtual FIQ exception

The `AArch32.TakeVirtualFIQException()` pseudocode procedure describes how the PE takes the exception.

G1.16.14 Additional pseudocode functions for exception handling

The `AArch32.EnterMonitorMode()` pseudocode function changes the PE mode to Monitor mode, with the required state changes.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The `AArch32.EnterHypMode()` pseudocode function changes the PE mode to Hyp mode, with the required state changes.

The `AArch32.EnterMode()` pseudocode function changes the PE mode to a PL1 mode, with the required state changes. It is used for all exceptions that are not routed to Hyp mode or Monitor mode.

The `AArch32.EnterMonitorMode()`, `AArch32.EnterHypMode()`, and `AArch32.EnterMode()` functions are described in Chapter J1 *Armv8 Pseudocode*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G1.17 Reset into AArch32 state

Reset on page D1-2335 describes the Armv8 reset model, including the defined levels of reset. When reset is deasserted, the PE starts executing instructions in the highest implemented Exception level. If that Exception level is using AArch32, then it starts execution:

- In Secure state, if the implementation includes EL3.
- With interrupts disabled:
  - In Hyp mode, if the highest implemented Exception level is EL2.
  - In Supervisor mode, otherwise.

Note

- This section describes the architectural requirements for a reset into AArch32 state. It takes no account of whether Arm licenses any particular combination of Exception levels and Execution state. For more information about the licensed combinations, see *Support for Exception levels and Execution states* on page D1-2420.
- The Execution state in which the highest implemented Execution level starts executing instructions on coming out of reset might be determined by a configuration input signal.

Reset returns some PE state to architecturally-defined or IMPLEMENTATION DEFINED values, and makes other state UNKNOWN, as described in *PE state on reset into AArch32 state* on page G1-5761. For more information about behavior when resetting into an Exception level using AArch32, see:

- *Behavior of caches at reset* on page G4-5895.
- *Enabling stages of address translation* on page G5-5932.
- *TLB behavior at reset* on page G5-5992.
- *Reset and debug* on page H6-7041.

When reset is deasserted, if the PE resets into an Exception level that is using AArch32, it is IMPLEMENTATION DEFINED whether execution starts:

- From an IMPLEMENTATION DEFINED address.
- If reset is into EL3 or EL1, from the low or high reset vector address, as determined by the reset value of the SCTLR.V bit. This reset value can be determined by an IMPLEMENTATION DEFINED configuration input signal.

  Note

  This option might be implemented for compatibility with earlier versions of the architecture.

Software might be able to identify the reset address:

- If reset is into EL3, by reading the reset value of MVBAR. That is, after coming out of reset, by reading MVBAR before the boot software has updated it. It is IMPLEMENTATION DEFINED whether this discovery mechanism is supported.
- If reset is into EL2 or EL1, by reading RVBAR. RVBAR can only be implemented at the highest implemented Exception level, and only if that Exception level is not EL3.

If RVBAR is not implemented, and at all Exception levels other than the highest implemented Exception level, the encoding for RVBAR is UNDEFINED.

The Arm architecture does not define any way of returning to a previous Execution state from a reset.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G1.17.1 PE state on reset into AArch32 state

Note

See the *ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0, and version 4.0* for the reset requirements for GIC System registers.

Immediately after a reset, much of the PE state is UNKNOWN. However, some of the PE state is defined. If the PE resets to AArch32 state using either a Cold or a Warm reset, the PE state that is defined is as follows:

- The global exclusive monitor and local exclusive monitor for the PE are UNKNOWN.
- If reset is into EL3 using AArch32, then all fields of the SCR reset to zero.

  Note

  This means SCR.NS correctly indicates that the PE is in Secure state.

- If reset is into EL2 using AArch32, then reset is into Hyp mode and CPSR.M resets to `0b1010`, otherwise reset is into Supervisor mode and CPSR.M resets to `0b0011`,
- CPSR.IL resets to 0.
- The CPACR.{cp11, cp10} fields reset to zero, and if CPACR.ASEDIS is implemented as an RW field it resets to zero.

  Note

  When CPACR.TRCDIS is an RW field, its reset value is architecturally UNKNOWN.

- PSTATE is reset to the values defined by the `AArch32.TakeReset()` pseudocode function, see *Pseudocode descriptions of reset* on page G1-5763.
- The FPEXC.EN field resets to 0.
- In the SCTLR:
  - The {AFE, TRE, UWXN, WXN, I, SED, ITD, C, A, M} fields reset to 0.
  - The {nTWE, nTWI, CP15BEN} fields reset to 1.
  - The {TE, EE, V} fields reset to IMPLEMENTATION DEFINED values, see the register description for more information.

  When the reset is to EL3 using AArch32 then these reset values apply only to the Secure instance of the SCTLR, and the reset value of the Non-secure SCTLR is architecturally UNKNOWN.

- All field of the TTBCR reset to 0.

  When the reset is to EL3 using AArch32 then:
  - All fields of the Secure TTBCR reset to 0.
  - In the Non-secure TTBCR, the EAE field resets to 0, and the reset values of all other fields are architecturally UNKNOWN.

- The VBAR resets to an IMPLEMENTATION DEFINED value.

  When the reset is to EL3 using AArch32 then this reset value applies only to the Secure instance of the register, and the reset value of the Non-secure VBAR is architecturally UNKNOWN.

- All fields of the DBGDCCINT reset to 0.
- The DBGDSCRext.{MDBGen, UDCCdis} fields reset to 0.
- The DBGOSDLR.DLK field resets to 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In addition:

If the reset is into EL1 using AArch32

- In the RMR register, the RR field resets to 0 on any warm or cold reset, and the AA64 field resets to 0 on a Cold reset.

If the reset is into EL2 using AArch32

- In the HRMR, the RR field resets to 0 on any warm or Cold reset, and the AA64 field resets to 0 on a Cold reset.
- The HSCTLR.{I, C, M} fields all reset to 0, and the HSCTLR.EE field resets to an IMPLEMENTATION DEFINED value.

If the reset is into EL2 using AArch32 or into EL3 using AArch32

For a reset into EL3 using AArch32 these reset values apply only if the implementation includes EL2, see the register descriptions for more information.

- All fields of the HCPTR reset to zero.
- All fields of the HCR reset to zero.
- The HCR2.{ID, CD} fields reset to zero.
- All fields of the HSTR reset to zero.
- The VMPIDR resets to the value of the MPIDR, see the register description for more information.
- The VPIDR resets to the value of the MIDR, see the register description for more information.
- The VTTBR.VMID field resets to zero.
- In the HDCR:
  - The HPMN field resets to the IMPLEMENTATION DEFINED value of PMCR.N.
  - The reset value of the HPME field is architecturally UNKNOWN.
  - All other fields reset to 0.

If the reset is into EL3 using AArch32

- The MVBAR resets to an IMPLEMENTATION DEFINED value, see the register description for more information.
- If the NSACR.{NSTRCDIS, NSASEDIS} fields are RW fields then they reset to 0.
- In the RMR register, the RR field resets to 0 on any warm or Cold reset, and the AA64 field resets to 0 on a Cold reset.
- All fields of the SCR reset to zero.
- All fields of the SDER reset to 0.
- All fields of the SDCR reset to zero.

For either a warm or a Cold reset

- The EDPRSR.SR field resets to 1.
- The EDESR.{SS, RC, OSUC} fields reset to 0.

For a Cold reset only

- The EDSCR.{RXO, TXU, INTdis, TDA, MA, HDE, ERR, RXfull, TXfull} fields reset to 0.
- The EDECCR.{NSE, SE} fields reset to 0.
- The EDPRSR.{SPMAD, SDAD} fields reset to 0, and the EDPRSR.SPD field resets to 1.
- The DBGOSLSR.OSLK field resets to 1.
- The DBGPRCR.CORENPDRQ field resets to the value of EDPRCR.COREPURQ.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

An External Debug reset sets EDPRCR.COREPURQ to 0, see *External debug register resets* on page H8-7070. If an External Debug reset and a Cold reset coincide, both DBGPRCR.CORENPRDRQ and EDPRCR.COREPURQ are reset to 0.

- The debug CLAIM bits are reset to 0.

  Note

  These are the bits that are set to 1 by writing to DBGCLAIMSET.CLAIM, and reset to 0 by writing to DBGCLAIMCLR.CLAIM.

- Each bit of AMCNTENCLR0, AMCNTENCLR1, AMCNTENSET0, and AMCNTENSET1 is set to 0.
- Each of the implemented architected activity monitor counters AMEVCNTR0<n> and each of the implemented auxiliary activity monitor counters AMEVCNTR1<n> are set to 0.

For more information about resets in AArch32 System registers, see Chapter G8 *AArch32 System Register Descriptions*.

G1.17.2 Pseudocode descriptions of reset

The `AArch32.TakeReset()` pseudocode procedure describes how the PE behaves when reset is deasserted.

The `AArch32.ResetGeneralRegisters()` pseudocode function resets the general-purpose registers.

The `AArch32.ResetSIMDFPRegisters()` pseudocode function resets the SIMD and floating-point registers.

The `AArch32.ResetSpecialRegisters()` pseudocode function resets the Special-purpose registers, and the debug System registers DLR and DSPSR, that are used for handling Debug exceptions.

The `AArch32.ResetSystemRegisters()` pseudocode function resets all System registers in the (`coproc==0b111x`) encoding space to their reset state as defined in the register descriptions in Chapter G8 *AArch32 System Register Descriptions*.

Note

The `ResetSystemRegisters()` function only resets the System registers. It has no effect on memory-mapped registers.

The `ResetExternalDebugRegisters()` pseudocode function resets all external debug registers to their reset state as defined in the register descriptions in Chapter H9 *External Debug Register Descriptions*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G1.18 Mechanisms for entering a low-power state

The following sections describe the architectural mechanisms that a PE can use to request entry to a low-power state:

- *Wait For Event and Send Event*.
- *Wait For Interrupt* on page G1-5767.

G1.18.1 Wait For Event and Send Event

The Wait For Event (WFE) mechanism permits a PE to request entry to a low-power state, and, if the request succeeds, to remain in that state until an event is generated by a Send Event operation, or another WFE wake-up event occurs. Example G1-2 describes how a spinlock implementation might use this mechanism to save energy.

Example G1-2 Spinlock as an example of using Wait For Event and Send Event

---

A multiprocessor operating system requires locking mechanisms to protect data structures from being accessed simultaneously by multiple PEs. These mechanisms prevent the data structures becoming inconsistent or corrupted if different PEs try to make conflicting changes. If a lock is busy, because a data structure is being used by one PE, it might not be practical for another PE to do anything except wait for the lock to be released. For example, if a PE is handling an interrupt from a device it might need to add data received from the device to a queue. If another PE is removing data from the queue, it will have locked the memory area that holds the queue. The first PE cannot add the new data until the queue is in a consistent state and the lock has been released. It cannot return from the interrupt handler until the data has been added to the queue, so it must wait.

Typically, a spin-lock mechanism is used in these circumstances:

- A PE requiring access to the protected data attempts to obtain the lock using single-copy atomic synchronization primitives such as the Load-Exclusive and Store-Exclusive operations described in *Synchronization and semaphores* on page E2-4023.
- If the PE obtains the lock, it performs its memory operation and releases the lock.
- If the PE cannot obtain the lock, it reads the lock value repeatedly in a tight loop until the lock becomes available. At this point, it again attempts to obtain the lock.

A spin-lock mechanism is not ideal for all situations:

- In a low-power system, the tight read loop is undesirable because it uses energy to no effect.
- In a multithreaded implementation, the execution of spin-locks by waiting threads can significantly degrade overall performance.

Using the Wait For Event and Send Event mechanism can improve the energy efficiency of a spinlock. In this situation, a PE that fails to obtain a lock can execute a Wait For Event instruction, `WFE`, to request entry to a low-power state. When a PE releases a lock, it must execute a Send Event instruction, `SEV`, causing any waiting PEs to wake up. Then, these PEs can attempt to gain the lock again.

The execution of a `WFE` instruction can cause suspension of execution only if all of the following are true:

- The instruction does not cause any other exception.
- When the instruction is executed:
  - The Event Register is not set.
  - There is not a pending WFE wakeup event.

For more information about the trap to EL2, see *Traps to Hyp mode of Non-secure EL0 and EL1 execution of WFE and WFI instructions* on page G1-5797.

The architecture does not define the exact nature of the low power state entered as a result of executing a `WFE` instruction, but the execution of a `WFE` instruction must not cause a loss of memory coherency.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

Although a complex operating system can contain thousands of distinct locks, the event sent by this mechanism does not indicate which lock has been released. If the event relates to a different lock, or if another PE acquires the lock more quickly, the PE fails to acquire the lock and can reenter the low-power state waiting for the next event.

The Wait For Event system relies on hardware and software working together to achieve energy saving:

- The hardware provides the mechanism to enter the Wait For Event low-power state.
- The operating system software is responsible for issuing:
  - A Wait For Event instruction, to request entry to the low-power state, used in the example when waiting for a spin-lock.
  - A Send Event instruction, required in the example when releasing a spin-lock.

The mechanism depends on the interaction of:

- WFE wake-up events, see *WFE wake-up events* on page G1-5766.
- The Event Register, see *The Event Register*.
- The Send Event instructions, see *The Send Event instructions* on page G1-5766.
- The Wait For Event instruction, see *The Wait For Event instruction*.

The Event Register

The Event Register is a single bit register for each PE. When set, an event register indicates that an event has occurred, since the register was last cleared, that might require some action by the PE. Therefore, the PE must not suspend operation on issuing a `WFE` instruction.

The reset value of the Event Register is UNKNOWN.

The Event Register for a PE is set by:

- The execution of an `SEV` instruction on any PE in the multiprocessor system.
- The execution of an `SEVL` instruction by the PE.
- An exception return.
- An event from a Generic Timer event stream, see *Event streams* on page G6-6067.
- An event sent by some IMPLEMENTATION DEFINED mechanism.

As shown in this list, the Event Register might be set by IMPLEMENTATION DEFINED mechanisms.

The Event Register is cleared only by a Wait For Event instruction.

Software cannot read or write the value of the Event Register directly.

The Wait For Event instruction

The action of the Wait For Event instruction depends on the state of the Event Register:

- If the Event Register is set, the instruction clears the register and completes immediately. Normally, if this happens the software makes another attempt to claim the lock.
- If the Event Register is clear the PE can suspend execution, and hardware might enter a low-power state. The PE can remain suspended until a WFE wake-up event or a reset occurs. When a WFE wake-up event occurs, or earlier if the implementation chooses, the `WFE` instruction completes.

The execution in AArch32 state of a `WFE` instruction that would otherwise cause suspension of execution might be trapped, see:

- *Traps to Undefined mode of EL0 execution of WFE and WFI instructions* on page G1-5781.
- *Traps to Hyp mode of Non-secure EL0 and EL1 execution of WFE and WFI instructions* on page G1-5797.
- *Traps to Monitor mode of the execution of WFE and WFI instructions in modes other than Monitor mode* on page G1-5808.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The Wait For Event instruction, WFE, is available at all privilege levels, see *WFE* on page F5-4945.

Software using the Wait For Event mechanism must tolerate spurious wake-up events, including multiple wake ups.

WFE wake-up events

The following events are *WFE wake-up events*:

- The execution of an SEV instruction on any PE in the system.
- The execution of an SEVL instruction on the PE.
- A physical IRQ interrupt, unless masked by the PSTATE.I bit.
- A physical FIQ interrupt, unless masked by the PSTATE.F bit.
- A physical SError interrupt, unless masked by the PSTATE.A bit.
- In Non-secure state in any mode other than Hyp mode:
  - When HCR.IMO is set to 1, a virtual IRQ interrupt, unless masked by the PSTATE.I bit.
  - When HCR.FMO is set to 1, a virtual FIQ interrupt, unless masked by the PSTATE.F bit.
  - When HCR.AMO is set to 1, a virtual SError interrupt, unless masked by the PSTATE.A bit.
- An asynchronous External Debug Request debug event, if halting is allowed. For the definition of *halting is allowed*, see *Halting allowed and halting prohibited* on page H2-6927.

  See also *External Debug Request debug event* on page H3-6983.
- An event sent by the timer event stream, see *Event streams* on page D11-2845.
- An event sent by some IMPLEMENTATION DEFINED mechanism.
- An event caused by the clearing of the global monitor associated with the PE.

In addition to the possible masking of WFE wake-up events shown in this list, when invasive debug is enabled and EDSCR.HDE is set to 1, EDSCR.INTdis can mask interrupts, including masking them acting as WFE wake-up events. See the register description for more information.

As shown in the list of wake-up events, an implementation can include IMPLEMENTATION DEFINED hardware mechanisms to generate wake-up events.

Note

For more information about PSTATE masking, see *Asynchronous exception masking controls* on page G1-5733. If the configuration of the masking controls provided by EL2 and EL3 mean that a PSTATE mask bit cannot mask the corresponding exception, then the physical exception is a WFE wake-up event, regardless of the value of the PSTATE mask bit.

The Send Event instructions

The Send Event instructions are:

SEV, Send Event This causes an event to be signaled to all PEs in the multiprocessor system.

SEVL, Send Event Local

This must set the local Event Register. It might signal an event to other PEs, but is not required to do so.

The mechanism that signals an event to other PEs is IMPLEMENTATION DEFINED. The PE is not required to guarantee the ordering of this event with respect to the completion of memory accesses by instructions before the SEV instruction. Therefore, Arm recommends that software includes a DSB instruction before any SEV instruction.

Note

A DSB instruction ensures that no instruction, including any SEV instruction, that appears in program order after the DSB instruction, can execute until the DSB instruction has completed. For more information, see *Data Synchronization Barrier (DSB)* on page E2-3993.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The SEVL instruction appears to execute in program order relative to any subsequent WFE instruction executed on the same PE, without the need for any explicit insertion of barrier instructions.

Execution of the Send Event instruction sets the Event Register.

The Send Event instructions are available at all privilege levels.

Pseudocode description of the Wait For Event mechanism

This section defines pseudocode functions that describe the operation of the Wait For Event mechanism.

The ClearEventRegister() pseudocode procedure clears the Event Register of the current PE.

The IsEventRegisterSet() pseudocode function returns TRUE if the Event Register of the current PE is set and FALSE if it is clear.

The WaitForEvent() pseudocode procedure optionally suspends execution until a WFE wake-up event or reset occurs, or until some earlier time if the implementation chooses. It is IMPLEMENTATION DEFINED whether restarting execution after the period of suspension causes a ClearEventRegister() to occur.

The SendEvent() pseudocode procedure sets the Event Register of every PE in the system.

G1.18.2 Wait For Interrupt

AArch32 state supports Wait For Interrupt through an instruction, WFI, that is provided in the A32 and T32 instruction sets. For more information, see *WFI* on page F5-4947.

When a PE issues a WFI instruction, its execution can be suspended, and a low-power state can be entered.

The execution in AArch32 state of a WFI instruction that would otherwise cause suspension of execution might be trapped, see:

- *Traps to Undefined mode of EL0 execution of WFE and WFI instructions* on page G1-5781.
- *Traps to Hyp mode of Non-secure EL0 and EL1 execution of WFE and WFI instructions* on page G1-5797.
- *Traps to Monitor mode of the execution of WFE and WFI instructions in modes other than Monitor mode* on page G1-5808.

The execution of a WFI instruction can cause suspension of execution only if both:

- The instruction does not cause any other exception.
- When the instruction is executed, there is not a pending WFI wakeup event.

WFI wake-up events

The PE can remain suspended in its WFI state until it is reset, or one of the following *WFI wake-up events* occurs:

- A physical IRQ interrupt, regardless of the value of the PSTATE.I bit.
- A physical FIQ interrupt, regardless of the value of the PSTATE.F bit.
- A physical SError interrupt, regardless of the value of the PSTATE.A bit.
- In Non-secure state in any mode other than Hyp mode:
  - When HCR.IMO is set to 1, a virtual IRQ interrupt, regardless of the value of the PSTATE.I bit.
  - When HCR.FMO is set to 1, a virtual FIQ interrupt, regardless of the value of the PSTATE.F bit.
  - When HCR.AMO is set to 1, a virtual SError interrupt, regardless of the value of the PSTATE.A bit.
- An asynchronous External Debug Request debug event, if halting is allowed. For the definition of *halting is allowed*, see *Halting allowed and halting prohibited* on page H2-6927.

  See also *External Debug Request debug event* on page H3-6983.

An implementation can include other IMPLEMENTATION DEFINED hardware mechanisms to generate WFI wake-up events.

When a WFI wake-up event is detected, or earlier if the implementation chooses, the WFI instruction completes.

WFI wake-up events cannot be masked by the mask bits in the PSTATE.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The architecture does not define the exact nature of the low power state, but the execution of a `WFI` instruction must not cause a loss of memory coherency.

Note

- Because debug events are WFI wake-up events, Arm strongly recommends that Wait For Interrupt is used as part of an idle loop rather than waiting for a single specific interrupt event to occur and then moving forward. This ensures the intervention of debug while waiting does not significantly change the function of the program being debugged.
- In some previous implementations of Wait For Interrupt, the idle loop is followed by exit functions that must be executed before taking the interrupt. The operation of Wait For Interrupt remains consistent with this model, and therefore differs from the operation of Wait For Event.
- Some implementations of Wait For Interrupt drain down any pending memory activity before suspending execution. The Arm architecture does not require this operation, and software must not rely on Wait For Interrupt operating in this way.

Using WFI to indicate an idle state on bus interfaces

A common implementation practice is to complete any entry into powerdown routines with a `WFI` instruction. Typically, the `WFI` instruction:

1. Forces the completion of execution of any instructions that are in progress, and of all associated bus activity.
2. Suspends the execution of instructions by the PE.

The control logic required to do this tracks the activity of the bus interfaces used by the PE. This means it can signal to an external power controller when there is no ongoing bus activity.

However, memory-mapped and external debug interface accesses to debug registers must continue to be processed while the PE is in the WFI state. The indication of idle state to the system normally only applies to the non-debug functional interfaces used by the PE, not the debug interfaces.

If ARMv8.0-DoubleLock is implemented and the value of DBGOSDLR.DLK, the OS Double Lock status bit, is set to 1, this idle state must not be signaled to the PE unless the system can guarantee, also, that the debug interface is idle.

Note

When separate Core and Debug power domains are implemented, the debug interface referred to in this section is the interface between the Core and Debug power domains, since the signal to the power controller indicates that the Core power domain is idle. For more information about the power domains, see *Power domains and debug* on page H6-7029.

The exact nature of this interface is IMPLEMENTATION DEFINED, but the use of Wait For Interrupt as the only architecturally-defined mechanism that completely suspends execution makes it very suitable as the preferred powerdown entry mechanism.

Pseudocode description of Wait For Interrupt

The `WaitForInterrupt`() pseudocode function optionally suspends execution until a WFI wake-up event or reset occurs, or until some earlier time if the implementation chooses.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G1.19 The AArch32 System register interface

In ArmArmv8, most System registers are accessed using the instructions described in *System register access instructions* on page F1-4057. The System register interface provides access to those instructions, and:

- These registers are encoded using the parameters {`coproc`, `opc1`, `CRn`, `CRm`, `opc2`}, with permitted `coproc` values of `0b1110` and `0b1111`.
- Some of these encodings provide the AArch32 System instructions.
- To maintain compatibility with previous versions of the Arm architecture, the access controls for the AArch32 System registers include the access controls for AArch32 Advanced SIMD and floating-point functionality.

  Note

  See *Background to the System register interface* on page G1-5770 for more information.

The following sections give more information about the AArch32 System register interface:

- *System registers in the coproc == 0b111x encoding space*.
- *Access to System registers*.
- *Access controls for Advanced SIMD and floating-point functionality*.
- *Background to the System register interface* on page G1-5770.

G1.19.1 System registers in the `coproc == 0b111x` encoding space

In AArch32 state:

- The `coproc == 0b1110` encoding space is reserved for the configuration and control of:
  - Debug features, see *Debug registers* on page G8-6586.
  - Trace features, see the *Embedded Trace Macrocell Architecture Specification*.
  - Identification registers for the Trivial Jazelle implementation, see *Trivial implementation of the Jazelle extension* on page G1-5701.
- The `coproc == 0b1111` encoding space is reserved for the control and configuration of the PE, including architecture and feature identification. This means these encodings provide access to the System registers that control and return status information for PE operation.

For more information, see Chapter G8 *AArch32 System Register Descriptions*.

G1.19.2 Access to System registers

Most System registers are accessible only from EL1 or higher. For possible accesses from EL0 the register descriptions in Chapter G8 *AArch32 System Register Descriptions* indicate whether a register is accessible from EL0.

G1.19.3 Access controls for Advanced SIMD and floating-point functionality

In Armv8, the CPACR controls access to Advanced SIMD and floating-point functionality from software executing at PL1 or EL0 in AArch32 state:

- The {cp10, cp11} fields control access to all Advanced SIMD and floating-point functionality, and can:
  - Disable EL0 and PL1 access to this functionality.
  - Enable access to this functionality at PL1 only.
  - Enable access to this functionality at EL0 and PL1.
- The ASEDIS field controls access to Advanced SIMD instructions that are not also floating-point instructions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Initially on powerup or reset into AArch32 state, all access to all Advanced SIMD and floating-point functionality from PL1 and EL0 is disabled.

Note

The CPACR has no effect on accesses from Hyp mode.

If an implementation includes EL3, the NSACR determines whether Advanced SIMD and floating-point functionality can be accessed from Non-secure state:

- The {cp10, cp11} fields control Non-secure access to all Advanced SIMD and floating-point functionality.
- The NSASEDIS field controls Non-secure access to Advanced SIMD instructions that are not also floating-point instructions.

If an implementation includes EL2, the HCPTR provides additional controls on Non-secure accesses to Advanced SIMD and floating-point functionality. For accesses that are otherwise permitted by the CPACR and NSACR settings, setting HCPTR bits to 1:

- Traps otherwise-permitted accesses from EL1 or EL0 to EL2. When EL2 is using AArch32, these accesses are trapped to Hyp mode.
- Makes accesses from EL2 mode UNDEFINED. When EL2 is using AArch32, this makes accesses from Hyp mode UNDEFINED.

In the HCPTR:

- The {TCP10, TCP11} fields control access to all Advanced SIMD and floating-point functionality.
- The TASE field controls access to Advanced SIMD instructions that are not also floating-point instructions.
- The TCPAC field traps Non-secure EL1 accesses to the CPACR to Hyp mode.

For more information, see *General trapping to Hyp mode of Non-secure accesses to the SIMD and floating-point registers* on page G1-5798.

Note

Whenever a pair of fields control the access to the Advanced SIMD and floating-point functionality, the values of each field of the pair must be identical. In Armv8, if these settings are not identical the behavior of the Advanced SIMD and floating-point functionality is CONSTRAINED UNPREDICTABLE, see *Handling of System register control fields for Advanced SIMD and floating-point operation* on page K1-7836.

For more information about Advanced SIMD and floating-point support, see *Advanced SIMD and floating-point support* on page G1-5772.

G1.19.4 Background to the System register interface

Note

This section is not part of the Armv8 Architecture specification. It is included only to present the rationale of some aspects of the System register interface.

The interface to the System registers was originally defined as part of a generic *coprocessor* interface, that gave access to 15 coprocessors, CP0 - CP15. Of these, CP8 - CP15 were reserved for use by Arm, while CP0 - CP7 were available for IMPLEMENTATION DEFINED coprocessors.

The coprocessors were accessed using coprocessor instructions. These instructions remain part of the T32 and A32 instruction sets, see *System register access instructions* on page F1-4057.

In the Arm coprocessor model, a coprocessor included both:

- Primary and secondary coprocessor registers, that form part of the coprocessor interface.
- A number of internal registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When accessing a 32-bit internal coprocessor register, using an MCR or MRC instruction, the instruction specified:

- The target coprocessor, specified by the coproc parameter and taking a value between p0 for CP0 and p15 for CP15.
- The primary coprocessor register, specified by the CRn parameter and taking a value between c0 and c15.
- The secondary coprocessor register, specified by the CRm parameter and taking a value between c0 and c15.
- Up to two additional parameters, opc1 and opc2, taking values between 0 and 7.

Other instructions in the group described in *System register access instructions* on page F1-4057 take a subset of these parameters:

- In the Armv7 definitions, LDC and STC instructions take parameters {coproc, CRd}, where CRd is the primary coprocessor register.
- MCRR and MRRC instructions take parameters {coproc, opc1, CRm}, where CRm is the primary coprocessor register.

To maintain backwards compatibility, the arguments to an MCR or MRC instruction remain {coproc, opc1, CRn, CRm, opc2}. Correspondingly, the encoding of the AArch64 System registers is described using the parameters {op0, op1, CRn, CRm, op2}. However:

- The naming of these parameters no longer has any particular significance.
- While the coproc field is a 4-bit field, op0 is a 2-bit field.

Of the coprocessors reserved for use by Arm, in Armv7 and earlier versions of the architecture:

- CP15 provided access to the System registers relating to non-debug operation, and was originally called the System control coprocessor. In Armv8, these registers are described as being in the coproc == 0b1111 encoding space.
- CP14 provided access to additional System registers, including those relating to debug and trace. In Armv8, these registers are described as being in the coproc == 0b1110 encoding space.
- CP10 and CP11 were used for Advanced SIMD and floating-point control, and many coprocessor instruction encodings targeting CP10 and CP11 were used as floating-point instruction encodings:
  - Generally Armv8 does not relate these instructions to the coprocessor encoding space, but the naming of registers and register fields for Advanced SIMD and floating-point control reflects the historic coprocessor model.
  - Because the Advanced SIMD and floating-point functionality used both CP10 and CP11, some System register controls of this functionality have a pair of fields, for example NSACR.{cp10, cp11}. In these cases, both fields must be set to the same value. For more information, see *Access controls for Advanced SIMD and floating-point functionality* on page G1-5769.

In Armv8:

- The AArch32 System registers include registers that were described as Special registers in Armv7 and earlier versions of the architecture. This means that the Armv8 System registers include registers that are outside the earlier coprocessor model.
- The Armv7 AArch32 instruction encodings for LDC, STC, MCR, MRC, MCRR, and MRRC instructions with coproc field values other than {1010, 1011, 1110, 1111} are available for reuse. Armv8.2 re-uses some encodings in this way.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G1.20 Advanced SIMD and floating-point support

Advanced SIMD and floating-point instructions on page E1-3960 introduces:

- The scalar floating-point instructions in the A32 and T32 instruction sets.
- The Advanced SIMD integer and floating-point vector instructions in the A32 and T32 instruction sets.
- The SIMD and floating-point register file, that can be viewed as:
  - Singleword registers S0 - S31.
  - Doubleword registers D0 - D31.
  - Quadword registers Q0 - Q15.
- The *Floating-Point Status and Control Register* (FPSCR).

For more information about the System registers for the Advanced SIMD and floating-point operation, see *Advanced SIMD and floating-point System registers* on page G1-5774. Software can interrogate these registers to discover the implemented Advanced SIMD and floating-point support.

AArch32 implications of not including support for Advanced SIMD and floating-point summarizes the effects of not supporting these instructions, and the following subsections give more information about the Advanced SIMD and Floating-point support:

- *Enabling Advanced SIMD and floating-point support*.
- *Advanced SIMD and floating-point System registers* on page G1-5774.
- *Context switching when using Advanced SIMD and floating-point functionality* on page G1-5775.
- *Floating-point exceptions and exception traps* on page G1-5776.

G1.20.1 AArch32 implications of not including support for Advanced SIMD and floating-point

As stated in *Implementations not including Advanced SIMD and floating-point instructions* on page D1-2420, although Armv8-A generally requires the inclusion of the Advanced SIMD and floating-point instructions in all instruction sets, for implementations targeting specialized markets, Arm might produce or license Armv8-A implementations that do not provide any support for Advanced SIMD and floating-point instructions. In such an implementation, in AArch32 state:

- The CPACR.{ASEDIS, cp11, cp10} fields are RES0.
- The NSACR.{NSASEDIS, cp11, cp10} fields are RES0.
- The HCPTR.{TASE, TCP11, TCP10} fields are RES1.
- The FPEXC, FPSCR, FPSID, MVFR0, MVFR1, and MVFR2 registers are not implemented and their encodings are UNDEFINED.
- Attempted accesses to Advanced SIMD and floating-point functionality are UNDEFINED. This means:
  - All Advanced SIMD and floating-point instructions are UNDEFINED.
  - Attempts to access the Advanced SIMD and floating-point System registers are UNDEFINED.

G1.20.2 Enabling Advanced SIMD and floating-point support

Software must ensure that the required access to the Advanced SIMD and floating-point features is enabled. Most of those controls are described in *Configurable instruction enables and disables, and trap controls* on page G1-5778, and this section:

- Summarizes those controls.
- Provides additional information in the following subsections:
  - *FPEXC control of access to Advanced SIMD and floating-point functionality* on page G1-5774.
  - *EL0 access to Advanced SIMD and floating-point functionality* on page G1-5774.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

This section shows the controls when the controlling Exception levels are using AArch32. Similar controls are provided when the Exception levels are using AArch64, and then apply to lower Exception levels that are using AArch32.

The controls of access to Advanced SIMD and floating-point functionality are:

General {cp10, cp11} or {TCP10, TCP11} controls

This relates to the CPACR.{cp10, cp11}, NSACR.{cp10, cp11}, and HCPTR.{TCP10, TCP11} controls.

Note

Background to the System register interface on page G1-5770 explains the naming of these controls.

The {cp10, cp11} controls provide general control of the use of Advanced SIMD and floating-point functionality, as follows:

- CPACR.{cp10, cp11} control access from PE modes other than Hyp mode.

  These fields have no effect on accesses to Advanced SIMD and floating-point functionality from Hyp mode.
- In an implementation that includes EL3, NSACR.{cp10, cp11} control access from Non-secure state.
- In an implementation that includes EL2, if NSACR.{cp10, cp11} permit Non-secure accesses, or if EL3 is not implemented, HCPTR.{TCP10, TCP11} provide an additional control on those accesses.

In each case, the {cp10, cp11} controls must be programmed to the same value, otherwise operation is CONSTRAINED UNPREDICTABLE. The Armv8 CONSTRAINED UNPREDICTABLE behavior is that, for all purposes other than reading the value of the register field, behavior is as if the cp11 field has the same value as the cp10 field. For more information, see *Handling of System register control fields for Advanced SIMD and floating-point operation* on page K1-7836.

For more information about these controls, see:

- *Enabling PL0 and PL1 accesses to the SIMD and floating-point registers* on page G1-5783.
- *General trapping to Hyp mode of Non-secure accesses to the SIMD and floating-point registers* on page G1-5798.
- *Enabling Non-secure access to SIMD and floating-point functionality* on page G1-5810.

Control of accesses to the CPACR from Non-secure PL1 modes

As stated in *General {cp10, cp11} or {TCP10, TCP11} controls*, the CPACR controls access to Advanced SIMD and floating-point functionality from PE modes other than Hyp mode. Accesses to the CPACR from Non-secure PL1 modes can be trapped to EL2, see *Traps to Hyp mode of Non-secure EL1 accesses to the CPACR* on page G1-5799.

Additional controls of Advanced SIMD functionality

- If implemented as an RW field, CPACR.ASEDIS can make all Advanced SIMD instructions UNDEFINED in all modes other than Hyp mode.
- In an implementation that includes EL3, when CPACR.ASEDIS permits use of the Advanced SIMD instructions or if the CPACR.ASEDIS control is not implemented, NSACR.NSASEDIS can make all Advanced SIMD instructions UNDEFINED in Non-secure state.
- In an implementation that includes EL2, when the CPACR and NSACR settings permit Non-secure use of the Advanced SIMD instructions, if HCPTR.TASE is implemented as an RW field it can make these instructions UNDEFINED in Hyp mode, and trap to Hyp mode any use of these instructions in a Non-secure PL0 or PL1 mode.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For more information about these controls, see:

- *Disabling PL0 and PL1 execution of Advanced SIMD instructions* on page G1-5784.
- *Traps to Hyp mode of Non-secure accesses to Advanced SIMD functionality* on page G1-5799.
- *Disabling Non-secure access to Advanced SIMD functionality* on page G1-5811.

Pseudocode description of enabling SIMD and floating-point functionality on page G1-5811 provides links to the pseudocode descriptions of all of these controls.

FPEXC control of access to Advanced SIMD and floating-point functionality

In addition, FPEXC.EN is an enable bit for most Advanced SIMD and floating-point operations. When FPEXC.EN is 0, all Advanced SIMD and floating-point instructions are treated as UNDEFINED except for:

- A `VMSR` to the FPEXC or FPSID register.
- A `VMRS` from the FPEXC, FPSID, MVFR0, MVFR1, or MVFR2 register.

These instructions can be executed only at EL1 or higher.

Note

- When the FPSID is accessible, any write access to the FPSID is ignored.
- When FPEXC.EN is 0, these operations are treated as UNDEFINED:
  - A `VMSR` to the FPSCR.
  - A `VMRS` from the FPSCR.

See *Enabling access to the SIMD and floating-point registers* on page G1-5784 for more information about the scope of the FPEXC.EN control.

When executing at EL0, the PE behaves as if the value of FPEXC.EN is 1 if either:

- EL1 is using AArch64.
- EL2 is enabled in the current Security state and is using AArch64. and the value of HCR_EL2.TGE is 1.

Note

In Non-secure state, if the value of HCR_EL2.RW is 0 then it is permitted for the value of FPEXC32_EL2.EN to control whether Advanced SIMD and floating-point functionality is enabled. However, Arm deprecates using the value of FPEXC32_EL2.EN to determine behavior.

EL0 access to Advanced SIMD and floating-point functionality

When the access controls summarized in this section permit EL0 access to the Advanced SIMD and floating-point functionality, this applies only to the subset of functionality that is available at EL0. In particular:

- Only Advanced SIMD and Floating-point System register that is accessible is the FPSCR.
- The Advanced SIMD and floating-point instructions are available.

Execution at EL0 corresponds to the application level view of the Advanced SIMD and floating-point functionality, as described in *Advanced SIMD and floating-point System registers* on page E1-3962.

G1.20.3 Advanced SIMD and floating-point System registers

AArch32 state provides a common set of System registers for the Advanced SIMD and floating-point functionality. This section gives general information about this set of registers, and indicates where each register is described in detail. It contains the following subsections:

- *Register map of the Advanced SIMD and floating-point System registers* on page G1-5775.
- *Accessing the Advanced SIMD and floating-point System registers* on page G1-5775.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register map of the Advanced SIMD and floating-point System registers

Table G1-21 shows the register map of the Advanced SIMD and Floating-point registers. Each register is 32 bits wide.

Table G1-21 Floating-point registers

Name	Permitted access
FPEXC	RW
FPSCR	RW
FPSID	RW[a]
MVFR0	RO
MVFR1	RO
MVFR2	RO

a. When FPSID is accessible, VMSR accesses to FPSID are ignored.

In an implementation that includes EL3, the Advanced SIMD and Floating-point registers are common registers, see *Common System registers* on page G5-6055.

Accessing the Advanced SIMD and floating-point System registers

Software accesses the Advanced SIMD and floating-point System registers using the VMRS and VMSR instructions, see:

- *VMRS* on page F6-5346.
- *VMSR* on page F6-5349.

For example:

```
VMRS <Rt>, FPSID    ; Read  Floating-Point System ID Register
VMRS <Rt>, MVFR1    ; Read  Media and VFP Feature Register 1
VMSR FPSCR, <Rt>    ; Write Floating-Point System Control Register
```

Software can access the Advanced SIMD and floating-point System registers only if the access controls permit the access, see *Enabling Advanced SIMD and floating-point support* on page G1-5772.

Note

All hardware ID information can be accessed only from EL1 or higher. This means:

The FPSID is accessible only from EL1 or higher.

This is a change introduced from VFPv3. Previously, the FPSID register can be accessed in all modes.

The MVFR registers are accessible only from EL1 or higher.

Unprivileged software must issue a system call to determine what features are supported.

G1.20.4 Context switching when using Advanced SIMD and floating-point functionality

When the Advanced SIMD and floating-point functionality is used by only a subset of processes, the operating system might implement lazy context switching of the Advanced SIMD and floating-point register file and System registers.

In the simplest lazy context switch implementation, the primary context switch software uses the CPACR.{cp10, cp11} controls to disable access to the Advanced SIMD and floating-point functionality, see *Enabling Advanced SIMD and floating-point support* on page G1-5772. Subsequently, when a process or thread attempts to use an Advanced SIMD or floating-point instruction, it triggers an Undefined Instruction exception. The

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

operating system responds by saving and restoring the Advanced SIMD and floating-point register file and System registers. Typically, it then re-executes the Advanced SIMD or floating-point instruction that generated the Undefined Instruction exception.

G1.20.5 Floating-point exceptions and exception traps

Execution of a floating-point instruction can generate an exceptional condition, called a *floating-point exception*.

The Armv8-A architecture supports synchronous exception generation in the event of any or all of the following floating-point exceptions:

- Input Denormal.
- Inexact.
- Underflow.
- Overflow.
- Divide by Zero.
- Invalid Operation.

Note

Do not confuse floating-point exceptions with the AArch32 architectural exceptions summarized in *AArch32 state exception descriptions* on page G1-5738.

Whether an implementation includes synchronous exception generation for these floating-point exceptions is IMPLEMENTATION DEFINED:

- For an implementation that does provide this capability, FPSCR.{IDE, IXE, UFE, OFE, DZE, IOE} are the control bits that enable synchronous exception generation for each of the floating-point exceptions.
- For an implementation that does provide this capability, the FPSCR.{IDE, IXE, UFE, OFE, DZE, IOE} are RAZ/WI.

Note

- An Input Denormal floating-point exception is generated when a single-precision or double-precision floating-point value is flushed-to-zero because the value of FPSCR.FZ is 1. However, no Input Denormal exception is generated when a half-precision floating-point value is flushed-to-zero because the value of FPSCR.FZ16 is 1.
- The Armv8-A architecture does not support asynchronous reporting of floating-point exceptions.

Trapped exception handling never causes the corresponding cumulative exception bit of the FPSCR to be set to 1. If this behavior is desired, the trap handler routine must use a read, modify, write sequence on the FPSCR to set the cumulative exception bit.

When generating synchronous exceptions for one or more floating-point exceptions is enabled, the synchronous exceptions generated by the floating-point exception traps are taken to the lowest Exception level that can handle such an exception, while adhering to the rule that an exception can never be taken to a lower Exception level. This means that trapped floating-point exceptions taken:

- From EL0 are taken to EL1, except for the following cases when they are taken from EL0 to EL2:
  - EL2 is using AArch32 and the value of HCR.TGE is 1.
  - EL2 is using AArch64 and the value of HCR_EL2.TGE is 1
- From EL1 are taken to EL1.
- From EL2 are taken to EL2.
- From EL3 are taken to EL3.

If the exception is taken to an Exception level that is using AArch64, then it is reported in the ELR_ELx for the Exception level to which it is taken, as described in *Exception entry* on page D1-2339.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the exception is taken to an Exception level that is using AArch32, then it is taken as an Undefined Instruction exception, see *Undefined Instruction exception* on page G1-5738. The FPEXC identifies the floating-point exceptions that occurred since the corresponding status bits in that register were last set to 0.

See also *Floating-point exceptions and exception traps* on page E1-3963.

In an implementation that provides synchronous exception generation for floating-point exceptions:

- Synchronous exception generation applies to floating-point exceptions generated by scalar SIMD and floating-point instructions executed in AArch32 state.
- The registers that are presented to the exception handler are consistent with the state of the PE immediately before the instruction that caused the exception. An implementation is permitted not to restore the cumulative exception flags in the event of such an exception.

Armv8 does not support the trapping of floating-point exceptions from Advanced SIMD instructions executed in AArch32 state.

The `AArch32.FPTrappedException()` and `FPProcessException()` pseudocode functions describe the handling of trapped floating-point exceptions generated in AArch32 state.

The `AArch32.FPTrappedException()` and `FPProcessException()` functions are described in Chapter J1 *Armv8 Pseudocode*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G1.21 Configurable instruction enables and disables, and trap controls

This section describes the controls provided by AArch32 state for enabling, disabling, and trapping particular instructions. Each control is categorized as an *instruction enable*, an *instruction disable*, or a *trap control*:

Instruction enables and instruction disables

Enable or disable the use of one or more particular instructions at a particular Privilege level and Security state.

When an instruction is disabled as a result of an instruction enable or disable, it is UNDEFINED.

The exception generated by attempting to executed an UNDEFINED instruction is:

- Taken to EL1 if the UNDEFINED instruction was executed at EL0, unless the instruction was executed at Non-secure EL0 and is routed to EL2 by the control described in *Routing exceptions from Non-secure EL0 to EL2* on page G1-5718.

  When the exception is taken to EL1, it is taken to Undefined mode.
- Otherwise, taken to the Exception level at which the UNDEFINED instruction was executed:
  - If the instruction was executed in Hyp mode the exception is taken to Hyp mode.
  - Otherwise, the exception is taken to Undefined mode.

Trap controls

Control whether one or more instructions, when executed at a particular Privilege level, are *trapped*.

Note

AArch32 trap controls are described in terms of Privilege levels, rather than Exception levels, because the PL1 traps apply at and are controlled from:

EL1 In Non-secure state, and in Secure state when EL3 is using AArch64.

EL3 In Secure state when EL3 is using AArch32.

For more information see *Security state, Exception levels, and AArch32 execution privilege* on page G1-5682.

Trap controls are grouped as:

PL1, excluding Monitor mode

Trapped instructions generate Undefined Instruction exceptions that are taken to Undefined mode, unless the instruction was executed at Non-secure EL0 and is routed to EL2 by the control described in *Routing exceptions from Non-secure EL0 to EL2* on page G1-5718.

For more information about these traps, see *PL1 configurable controls* on page G1-5779.

Hyp mode (PL2)

These traps apply only to execution in Non-secure state. This section only describes the traps that apply when EL2 is using AArch32.

Trapped instructions generate:

- Hyp Trap exceptions, taken to Hyp mode, if trapped from a mode other than Hyp mode.
- Undefined Instruction exceptions taken to Hyp mode, if trapped from Hyp mode.

For more information about these traps, see *EL2 configurable controls* on page G1-5787.

See also *Routing exceptions from Non-secure EL0 to EL2* on page G1-5718.

Monitor mode (Secure PL1)

This section only describes the traps that apply when EL3 is using AArch32.

Trapped instructions generate Monitor Trap exceptions, that are taken to Monitor mode.

For more information about these traps, see *EL3 configurable controls* on page G1-5806.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

An exception generated as a result of an instruction enable or disable, or a trap control, is only taken if the instruction does not also generate a higher priority exception. *Exception prioritization for exceptions taken to AArch32 state* on page G1-5706 defines the prioritization of different exceptions on the same instruction.

Exceptions generated as a result of these controls are synchronous exceptions.

For exceptions taken to an Exception level that is using AArch32, only exceptions that are taken to Hyp mode are reported in a syndrome register, the HSR.

Note

- A particular control might have a mnemonic that suggests it is different type of control to the control type it is categorized as. For example, CPACR.TRCDIS is a trap control even though TRCDIS is a mnemonic for Trace Disable.
- An implementation might provide additional controls, in IMPLEMENTATION DEFINED registers, to provide control of trapping of IMPLEMENTATION DEFINED features.
- *Configurable instruction enables and disables, and trap controls* on page D1-2373 describes controls provided by AArch64 state for enabling, disabling, and trapping instructions. Generally, where an AArch64 control applies to execution at lower Exception levels, it traps the equivalent functionality when that lower Exception level is using AArch32. See the AArch64 trap controls for more information.

This section is organized as follows:

- *Register access instructions*.
- *PL1 configurable controls*.
- *EL2 configurable controls* on page G1-5787.
- *EL3 configurable controls* on page G1-5806.
- *Pseudocode description of configurable instruction enables, disables, and traps* on page G1-5811.

G1.21.1 Register access instructions

When an instruction is disabled or trapped, the exception is taken before execution of the instruction. This means that if the instruction is a register access instruction:

- No access is made before the exception is taken.
- Side-effects that are normally associated with the access do not occur before the exception is taken.

G1.21.2 PL1 configurable controls

In AArch32 state, each control is associated with a particular System register field that is accessible:

- When EL3 is using AArch64, or when an implementation does not include EL3, from EL1.
- When EL3 is using AArch32:
  - In Non-secure state, from EL1.
  - In Secure state, from EL3.

This means that the controls are described as PL1 controls, because PL1 is defined as being the Privilege level of software that is executing:

- At EL3, if the PE is executing in EL3 and EL3 is using AArch32.
- At EL1 under all other conditions.

Where there is an AArch64 control that is equivalent to an AArch32 PL1 control, the AArch64 control is an EL1 control.

Any exception that is generated because of an AArch32 PL1 control is taken to a PL1 mode.

Note

Any exception generated because of an AArch32 PL1 control is taken to AArch32 state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G1-22 shows the AArch32 System registers that contain these controls.

Table G1-22 System registers that contain instruction enables and disables, and trap controls

Register name	Register description
SCTLR	System Control Register
FPEXC	Floating-point Exception Control Register
CPACR	Architectural Feature Access Control Register
DBGDSCRext	Monitor System Debug Control Register
PMUSERENR	Performance Monitors User Enable Register
AMUSERENR	Activity Monitors User Enable Register

Table G1-23 summarizes these controls.

Table G1-23 Instruction enables and disables, and trap controls, for exceptions taken to Undefined mode

Control	Control type[a]	Description
SCTLR.{nTWE, nTWI}	T	*Traps to Undefined mode of EL0 execution of WFE and WFI instructions* on page G1-5781
SCTLR.{SED, ITD} SCTLR.CP15BEN	D E	*Disabling or enabling PL0 and PL1 use of AArch32 deprecated functionality* on page G1-5781
CPACR.TRCDIS	T	*Traps to Undefined mode of PL0 and PL1 System register accesses to trace registers* on page G1-5782
CPACR.{cp11, cp10} FPEXC.EN CPACR.ASEDIS	E E D	*Enabling use of Advanced SIMD and floating-point functionality* on page G1-5783
DBGDSCRext.UDCCdis	T	*Traps to Undefined mode of EL0 accesses to the Debug Communications Channel (DCC) registers* on page G1-5784
CNTKCTL.{PL0PTEN, PL0VTEN, PL0PCTEN, PL0VCTEN}	T	*Traps to Undefined mode of EL0 accesses to the Generic Timer registers* on page G1-5785
PMUSERENR.{ER, CR, SW, EN}	T	*Traps to Undefined mode of EL0 accesses to Performance Monitors registers* on page G1-5785
AMUSERENR.EN	T	*Traps to Undefined mode of EL0 accesses to Activity Monitors registers* on page G1-5786

a. See Table G1-24.

Table G1-24 Control types, for exceptions taken to Undefined mode

Abbreviation	Type	See
D	Disable	*Instruction enables and instruction disables* on page G1-5778
E	Enable	*Instruction enables and instruction disables* on page G1-5778
T	Trap	*Trap controls* on page G1-5778

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When generated in Non-secure User mode, exceptions generated by these controls can be routed to EL2, as described in *Routing exceptions from Non-secure EL0 to EL2* on page G1-5718.

Instructions that fail their Condition code check

See *Conditional execution of undefined instructions* on page G1-5740.

Trapping to PL1 of instructions that are UNPREDICTABLE

For an instruction that is UNPREDICTABLE or CONSTRAINED UNPREDICTABLE, when the instruction is disabled or trapped then it is CONSTRAINED UNPREDICTABLE whether execution of the instruction generates an Undefined Instruction exception.

Traps to Undefined mode of EL0 execution of `WFE` and `WFI` instructions

SCTLR.{nTWE, nTWI} trap EL0 execution of `WFE` and `WFI` instructions to Undefined mode:

SCTLR.nTWE

1	This control has no effect on the EL0 execution of `WFE` instructions.
0	Any attempt to execute a `WFE` instruction at EL0 is trapped to Undefined mode, if the instruction would otherwise have caused the PE to enter a low-power state.

SCTLR.nTWI

1	This control has no effect on the EL0 execution of `WFI` instructions.
0	Any attempt to execute a `WFI` instruction at EL0 is trapped to Undefined mode, if the instruction would otherwise have caused the PE to enter a low-power state.

The attempted execution of a conditional `WFE` or `WFI` instruction is only trapped if the instruction passes its Condition code check.

Note

Since a `WFE` or `WFI` can complete at any time, even without a Wakeup event, the traps on `WFE` of `WFI` are not guaranteed to be taken, even if the `WFE` or `WFI` is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

When generated in Non-secure User mode, exceptions generated by these controls can be routed to EL2, as described in *Routing exceptions from Non-secure EL0 to EL2* on page G1-5718.

For more information about these instructions, and when they can cause the PE to enter a low-power state, see:

- *Wait For Event and Send Event* on page G1-5764.
- *Wait For Interrupt* on page G1-5767.

Disabling or enabling PL0 and PL1 use of AArch32 deprecated functionality

Table G1-25 on page G1-5782 shows the deprecated AArch32 functionality that might have disable controls in the SCTLR:

- The SED control is always implemented.
- Whether each of the ITD or CP15BEN controls is implemented is IMPLEMENTATION DEFINED. If a control is not implemented, then the associated functionality cannot be disabled.

When an instruction is disabled by one of these controls, it is UNDEFINED at PL0 and PL1. This means an attempt to execute the instruction at PL0 or PL1 generates an Undefined Instruction exception that is taken to Undefined mode, unless both of the following apply, in which case the attempted execution generates an exception that is taken to EL2, as described in *Routing exceptions from Non-secure EL0 to EL2* on page G1-5718:

- The instruction is executed at Non-secure EL0 using AArch32.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Either:
  - EL2 is using AArch32 and the value of HCR.TGE is 1.
  - EL2 is using AArch64 and the value of HCR_EL2.TGE is 1.

Table G1-25 PL1 controls for disabling and enabling PL0 and PL1 use of AArch32 deprecated functionality

Deprecated AArch32 functionality	Instruction enable or disable in the SCTLR[a]	Disabled instructions
`SETEND` instructions	SED[b]	SETEND instructions
Some uses of `IT` instructions	ITD[c]	See the SCTLR.ITD description
Accesses to the CP15DMB, CP15DSB, and CP15ISB barrier instructions	CP15BEN[d]	`MCR` accesses to the CP15DMB, CP15DSB, and CP15ISB instructions

a. The controls that are implemented in SCTLR are also implemented in SCTLR_EL1, and apply when PL1 is using AArch64 and PL0 is using AArch32.
b. `SETEND` instruction disable. `SETEND` instructions are disabled when the value of this field is 1.
c. `IT` instruction disable. If this control is implemented, some uses of `IT` instructions are disabled when the value of this field is 1.
d. System register (`coproc==0b1111`) memory barrier enable. If this control is implemented, the specified register accesses are disabled when the value of CP15BEN is 0.

Note

The uses of the `IT` instruction, and use of the CP15DMB, CP15DSB, and CP15ISB barrier instructions, are deprecated for performance reasons.

Traps to Undefined mode of PL0 and PL1 System register accesses to trace registers

If implemented, the CPACR.TRCDIS control traps PL0 and PL1 System register accesses to the trace registers to Undefined mode, as follows:

1 PL0 and PL1 accesses to the System register interface to the PE Trace Unit are trapped to Undefined mode

0 This control has no effect on PL0 and PL1 accesses to the System register interface to the PE Trace Unit.

If the CPACR.TRCDIS control is not implemented, then the CPACR.TRCDIS field is RAZ/WI. This means the CPACR does not provide a trap to Undefined mode of PL1 and PL0 System register accesses to trace registers. See the register description for more information.

Note

- System register accesses to the PE Trace Unit use the (`coproc==0b1110`) encoding space.
- The ETMv4 architecture does not permit EL0 to access the trace registers. If the Armv8-A architecture is implemented with an ETMv4 implementation, EL0 accesses to the trace System registers are UNDEFINED.
- The Armv8-A architecture does not provide traps on trace register accesses through the optional memory-mapped external debug interface.

System register accesses to the trace System registers can have side-effects. When a System register access is trapped, no side-effects occur before the exception is taken, see *Register access instructions* on page G1-5779.

If EL3 is implemented and is using AArch32, and NSACR.NSTRCDIS is 1, CPACR.TRCDIS behaves as RAO/WI in Non-secure state. This behavior also applies if the CPACR.TRCDIS control is not implemented.

When generated in Non-secure User mode, an exception generated by this control can be routed to EL2, as described in *Routing exceptions from Non-secure EL0 to EL2* on page G1-5718.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Enabling use of Advanced SIMD and floating-point functionality

Table G1-26 summarizes the controls of Advanced SIMD and floating-point functionality.

Table G1-26 Controls of use of Advanced SIMD and floating-point functionality

Control	Type	Description, see
CPACR.{cp11, cp10}	E	*Enabling PL0 and PL1 accesses to the SIMD and floating-point registers*
FPEXC.EN	E	*Enabling access to the SIMD and floating-point registers* on page G1-5784
CPACR.ASEDIS	D	*Disabling PL0 and PL1 execution of Advanced SIMD instructions* on page G1-5784

If any of CPACR.{cp11, cp10}, FPEXC.EN, or for Advanced SIMD instructions, CPACR.ASEDIS, disable a floating-point or an Advanced SIMD instruction, the instruction is UNDEFINED. Support for the CPACR.ASEDIS control is optional, and if the control is not implemented behavior is as if the control permits the execution of Advanced SIMD instructions at PL1 and PL0.

When generated in Non-secure User mode, exceptions generated by these controls can be routed to EL2, as described in *Routing exceptions from Non-secure EL0 to EL2* on page G1-5718.

Enabling PL0 and PL1 accesses to the SIMD and floating-point registers

CPACR.{cp11, cp10} enable PL0 and PL1 accesses to the SIMD and floating-point registers.

When CPACR.cp10 is:

00 PL0 and PL1 accesses to Advanced SIMD and floating-point registers or instructions are UNDEFINED.

01 PL0 accesses to Advanced SIMD and floating-point registers or instructions are UNDEFINED.

10 Reserved. The effect of programming this field to this value is CONSTRAINED UNPREDICTABLE.

11 This control permits full access to the Advanced SIMD and floating-point functionality from PL0 and PL1.

The value of CPACR.cp11 is ignored. If CPACR.cp11 is programmed with a different value to CPACR.cp10 then CPACR.cp11 is UNKNOWN on a direct read of the CPACR.

Note

- Software must set CPACR.cp11 and CPACR.cp10 to the same value.

Table G1-27 shows the registers for which accesses are enabled.

Table G1-27 Register accesses enabled at PL0 and PL1 by CPACR.{cp11, cp10}

Enabled at	Registers
PL0 and PL1, or PL0 only[a]	FPSCR, FPEXC, FPSID, MVFR0, MVFR1, MVFR2, and any of the SIMD and floating-point registers Q0-Q15, including their views as D0-D31 registers or S0-S31 registers[b]

a. Depending on the value of CPACR.{cp11, cp10}. See the register description for details.

b. Permitted VMSR accesses to the FPSID are ignored, but for the purposes of the {cp10, cp11} controls the architecture defines a VMSR accesses to the FPSID from EL1 or higher is an access to a SIMD and floating-point register.

If EL3 is implemented and is using AArch32, and NSACR.{cp11, cp10} are both set to 0, the functionality described in this section is disabled in Non-secure state, and CPACR.{cp11, cp10} are RAZ/WI in Non-secure state. See *Enabling Non-secure access to SIMD and floating-point functionality* on page G1-5810.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For more information about SIMD and floating-point support, see *Advanced SIMD and floating-point support* on page G1-5772.

Enabling access to the SIMD and floating-point registers

FPEXC.EN enables accesses to the SIMD and floating-point registers at all Exception levels, but does not control the following:

- `VMSR` accesses to the FPEXC or FPSID.
- `VMRS` accesses from the FPEXC, FPSID, MVFR0, MVFR1, or MVFR2.

When FPEXC.EN is:

1 Accesses to the registers shown in Table G1-28 are enabled at all Exception levels.

0 All accesses to the registers shown in Table G1-28 are UNDEFINED.

Table G1-28 shows the registers for which accesses are enabled, and for an exception taken to Hyp mode, how the exception is reported in HSR.

Table G1-28 Register accesses enabled when FPEXC.EN is 1

Enabled at	Registers	Syndrome reporting in HSR[a]
All Exception levels	FPSCR, and any of the SIMD and floating-point registers Q0-Q15, including their views as D0-D31 registers or S0-S31 registers.	Exception for an unknown reason, using EC value `0x00`

a. Only for exceptions that are taken to Hyp mode.

For more information, see *Advanced SIMD and floating-point support* on page G1-5772.

Disabling PL0 and PL1 execution of Advanced SIMD instructions

If implemented as an RW field, CPACR.ASEDIS can disable PL0 and PL1 execution of Advanced SIMD instructions, as follows:

1 Advanced SIMD instructions are UNDEFINED at PL0 and PL1.

0 Advanced SIMD instruction execution is enabled at PL0 and PL1.

The instructions that CPACR.ASEDIS disables are those described in *Controls of Advanced SIMD operation that do not apply to floating-point operation* on page E1-3966.

When the control is not implemented, meaning the CPACR.ASEDIS field is RAZ/WI, behavior is as if the control permits execution of Advanced SIMD instructions at PL0 and PL1.

If EL3 is implemented and is using AArch32, and NSACR.NSASEDIS is 1, CPACR.ASEDIS is RAO/WI in Non-secure state. This also applies when the CPACR.ASEDIS control is not implemented.

Traps to Undefined mode of EL0 accesses to the Debug Communications Channel (DCC) registers

DBGDSCRext.UDCCdis traps EL0 accesses to the DCC registers to Undefined mode:

1 EL0 accesses to the DCC registers are trapped to Undefined mode

0 This control has no effect on EL0 accesses to the DCC registers.

Traps of EL0 accesses to the DBGDTRRXint and DBGDTRTXint are ignored in Debug state.

Table G1-29 shows the registers for which accesses are trapped.

Table G1-29 Register accesses trapped to Undefined mode when DBGDSCRext.UDCCdis is 1

Traps from	Registers
EL0	DBGDSCRint, DBGDTRRXint, DBGDTRTXint, DBGDIDR, DBGDSAR, DBGDRAR

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

All accesses to these registers are trapped, including LDC and STC accesses to DBGDTRTXint and DBGDTRRXint, and MRRC accesses to DBGDSAR and DBGDRAR.

When generated in Non-secure User mode, an exception generated by this control can be routed to EL2, as described in *Routing exceptions from Non-secure EL0 to EL2* on page G1-5718.

Traps to Undefined mode of EL0 accesses to the Generic Timer registers

CNTKCTL.{PL0PTEN, PL0VTEN, PL0PCTEN, PL0VCTEN} trap EL0 accesses to the Generic Timer registers to Undefined mode, as follows:

- CNTKCTL.PL0PTEN traps EL0 accesses to the physical timer registers.
- CNTKCTL.PL0VTEN traps EL0 accesses to the virtual timer registers.
- CNTKCTL.PL0PCTEN traps EL0 accesses to the frequency register and physical counter register.
- CNTKCTL.PL0VCTEN traps EL0 accesses to the frequency register and virtual counter register.

For all of these controls:

1 This control has no effect on EL0 accesses to the corresponding registers.

0 EL0 accesses to the corresponding registers are trapped to Undefined mode.

Accesses to the frequency register, CNTFRQ, are only trapped if CNTKCTL.PL0PCTEN and CNTKCTL.PL0VCTEN are both 0.

Table G1-30 shows the registers for which accesses are trapped.

Table G1-30 Register accesses trapped to Undefined mode by CNTKCTL trap controls

Traps from	Trap control	Registers
EL0	PL0PTEN	CNTP_CTL, CNTP_CVAL, CNTP_TVAL
	PL0VTEN	CNTV_CTL, CNTV_CVAL, CNTV_TVAL
	PL0PCTEN	CNTFRQ, CNTPCT
	PL0VCTEN	CNTFRQ, CNTVCT

When generated in Non-secure User mode, an exception generated by this control can be routed to EL2, as described in *Routing exceptions from Non-secure EL0 to EL2* on page G1-5718.

Traps to Undefined mode of EL0 accesses to Performance Monitors registers

PMUSERENR.{ER, CR, SW, EN} trap EL0 accesses to the Performance Monitors registers to Undefined mode. For each of these controls:

1 This control has no effect on EL0 accesses to the corresponding registers.

0 EL0 accesses to the corresponding registers are trapped to Undefined mode.

For those Performance Monitors registers that more than one PMUSERENR.{ER, CR, SW, EN} control applies to, accesses are only trapped if all controls that apply are set to 0.

The accesses that these trap controls trap might be reads, writes, or both.

Note

- The architecture does not provide traps on Performance Monitors register accesses through the memory-mapped external debug interface.
- If the Performance Monitors Extension is not implemented, the Performance Monitors registers, including PMUSERENR, are reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G1-31 shows the registers for which EL0 accesses are trapped. For each register, the table shows the type of access trapped.

Table G1-31 Register accesses trapped to Undefined mode when disabled from EL0

Traps from	Trap control	Registers	Access type
EL0	ER	PMXEVCNTR, PMEVCNTR<n>	R
		PMSELR	RW
	CR	PMCCNTR, accessed using an `MRC`	R
	CR	PMCCNTR, accessed using an `MRRC`	R
	SW	PMSWINC	W
	EN	PMCNTENSET, PMCNTENCLR, PMCR, PMOVSR, PMSWINC, PMSELR, PMCEID0, PMCEID1, PMCEID2, PMCEID3, PMCCNTR, PMXEVTYPER, PMXEVCNTR, PMOVSSET, PMEVCNTR<n>, PMEVTYPER<n>, PMCCFILTR	RW[a]

a. The EL0 access is trapped only if the corresponding EL1 accesses is permitted. For example, the PMSWINC register is WO at EL1, and therefore, when the value of EN is 0:
- Write accesses to the register from EL0 are trapped.
- Read accesses to the register from EL0 are UNDEFINED, because read accesses to the register from EL1 are UNDEFINED.

When generated in Non-secure User mode, an exception generated by this control can be routed to EL2, as described in *Routing exceptions from Non-secure EL0 to EL2* on page G1-5718.

Traps to Undefined mode of EL0 accesses to Activity Monitors registers

AMUSERENR.EN traps EL0 accesses to the Activity Monitors System registers other than AMUSERENR to Undefined mode:

1 This control has no effect on EL0 accesses to the corresponding registers.

0 EL0 accesses to the corresponding registers are trapped to Undefined mode.

Note

- The architecture does not provide traps on Activity Monitors register accesses through the memory-mapped external interface.
- If the Activity Monitors Extension is not implemented, the Activity Monitors registers, including AMUSERENR, are reserved.

Table G1-32 shows the registers for which EL0 accesses are trapped.

Table G1-32 Register accesses trapped to Undefined mode when disabled from EL0

Traps from	Registers
EL0	AMCFGR, AMCGCR, AMCNTENCLR0, AMCNTENCLR1, AMCNTENSET0, AMCNTENSET1, AMCR, AMEVTYPER0<n>, AMEVTYPER1<n>, AMEVCNTR0<n> or AMEVCNTR1<n>.

When generated in Non-secure User mode, an exception generated by this control can be routed to EL2, as described in *Routing exceptions from Non-secure EL0 to EL2* on page G1-5718.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G1.21.3 EL2 configurable controls

These controls are ignored in Secure state when using AArch32.

Table G1-33 shows the System registers that contain these controls.

Table G1-33 System registers that contain instruction enables and disables, and trap controls

Register name	Register description
FPEXC	Floating-point Exception Control Register
HCR	Hypervisor Configuration Register
HSTR	Hypervisor System Trap Register
HCPTR	Hyp Architectural Feature Trap Register
HDCR	Hyp Debug Control Register

Note

- FPEXC.EN is a control that is in a System register provided by PL1. However, some exceptions generated because the value of FPEXC.EN is 1 are taken to Hyp mode.
- For completeness, Table G1-34 includes the HCR.TGE routing control, that is described in *Routing exceptions from Non-secure EL0 to EL2* on page G1-5718.

Table G1-34 summarizes the controls.

Table G1-34 Instruction enables and disables, and trap controls, for exceptions taken to Hyp mode

Control	Control type[a]	Description
HSCTLR.{SED, ITD} HSCTLR.CP15BEN	D E	*Disabling or enabling EL2 use of AArch32 deprecated functionality* on page G1-5790
HCR.{TRVM, TVM}	T	*Traps to Hyp mode of Non-secure EL1 accesses to virtual memory control registers* on page G1-5790
HCR.HCD	D	*Disabling Non-secure state execution of HVC instructions* on page G1-5791
HCR.TGE	R	*Routing exceptions from Non-secure EL0 to EL2* on page G1-5718
HCR.TTLB	T	*Traps to Hyp mode of Non-secure EL1 execution of TLB maintenance instructions* on page G1-5791
HCR.{TSW, TPC, TPU}	T	*Traps to Hyp mode of Non-secure EL1 execution of cache maintenance instructions* on page G1-5792
HCR.TAC	T	*Traps to Hyp mode of Non-secure EL1 accesses to the Auxiliary Control Register* on page G1-5792
HCR.TIDCP	T	*Traps to Hyp mode of Non-secure EL0 and EL1 accesses to lockdown, DMA, and TCM operations* on page G1-5793
HCR.TSC	T	*Traps to Hyp mode of Non-secure EL1 execution of SMC instructions* on page G1-5794
HCR.{TID0, TID1, TID2, TID3}	T	*Traps to Hyp mode of Non-secure EL0 and EL1 accesses to the ID registers* on page G1-5794

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G1-34 Instruction enables and disables, and trap controls, for exceptions taken to Hyp mode (continued)

Control	Control type[a]	Description
HCR.{TWI, TWE}	T	*Traps to Hyp mode of Non-secure EL0 and EL1 execution of WFE and WFI instructions* on page G1-5797
HCPTR.TAM	T	*Traps to Hyp mode of Non-secure EL0 and EL1 accesses to Activity Monitors registers* on page G1-5797
HCPTR.{TCP11, TCP10}	T	*General trapping to Hyp mode of Non-secure accesses to the SIMD and floating-point registers* on page G1-5798
FPEXC.EN	T	*Enabling access to the SIMD and floating-point registers* on page G1-5798
HCPTR.TASE	T	*Traps to Hyp mode of Non-secure accesses to Advanced SIMD functionality* on page G1-5799
HCPTR.TCPAC	T	*Traps to Hyp mode of Non-secure EL1 accesses to the CPACR* on page G1-5799
HCPTR.TTA	T	*Traps to Hyp mode of Non-secure System register accesses to trace registers* on page G1-5800
HDCR.TTRF	T	*Traps to Hyp mode of Non-secure System register accesses to trace filter control registers* on page G1-5800
HSTR.{T0-T3, T5-T13, T15}	T	*General trapping to Hyp mode of Non-secure EL0 and EL1 accesses to System registers in the (coproc==0b1111) encoding space* on page G1-5801
HDCR.{TDRA, TDOSA, TDA}	T	*Traps to Hyp mode of Non-secure System register accesses to debug registers* on page G1-5802
CNTHCTL.{PL1PCEN, PL1PCTEN}	T	*Traps to Hyp mode of Non-secure EL0 and EL1 accesses to the Generic Timer registers* on page G1-5804
HDCR.{TPM, TPMCR}	T	*Traps to Hyp mode of Non-secure EL0 and EL1 accesses to Performance Monitors registers* on page G1-5805
HCR2.TERR	T	*Traps to Hyp mode of Non-secure EL1 accesses to the RAS error record registers* on page G1-5806

a. See Table G1-35.

Table G1-35 Control types, for exceptions taken to Hyp mode

Abbreviation	Type	See
D	Disable	*Instruction enables and instruction disables* on page G1-5778
E	Enable	*Instruction enables and instruction disables* on page G1-5778
R	Routing control	*Routing exceptions from Non-secure EL0 to EL2* on page G1-5718
T	Trap	*Trap controls* on page G1-5778

Also see the following:

- *Register access instructions* on page G1-5779.
- *Instructions that fail their Condition code check* on page G1-5789.
- *Trapping to EL2 of instructions that are UNPREDICTABLE* on page G1-5789.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Instructions that fail their Condition code check

For UNDEFINED instructions that fail their Condition code check, see *Conditional execution of undefined instructions* on page G1-5740.

For an instruction that has a Hyp trap set, that fails its Condition code check:

- Unless the trap description states otherwise, it is IMPLEMENTATION DEFINED whether the instruction:
  - Generates a Hyp Trap exception.
  - Executes as a `NOP`.

Any implementation must be consistent in its handling of instructions that fail their Condition code check. This means that:

- Whenever a Hyp trap is set on an instruction it must either:
  - Always generate a Hyp Trap exception.
  - Always treat the instruction as a `NOP`.
- The IMPLEMENTATION DEFINED part of the requirements of *Conditional execution of undefined instructions* on page G1-5740 must be consistent with the handling of Hyp traps on instructions that fail their Condition code check. Table G1-36 shows this:

Table G1-36 Consistent handling of instructions that fail their Condition code check

Behavior of conditional UNDEFINED instruction[a]	Hyp trap on instruction that fails its Condition code check[b]
Executes as a `NOP`	Executes as a `NOP`
Generates an Undefined Instruction exception	Generates a Hyp Trap exception

a. As defined in *Conditional execution of undefined instructions* on page G1-5740. In Non-secure EL0 and EL1 modes, this applies only if no Hyp trap is set for the instruction, otherwise see the behavior in the other column of the table.

b. For a trapped instruction executed in a Non-secure EL1 or EL0 mode.

Note

Hyp traps on `WFE` and `WFI` instructions generate Hyp Trap exceptions only if the instruction passes its Condition code check. See *Traps to Hyp mode of Non-secure EL0 and EL1 execution of WFE and WFI instructions* on page G1-5797.

Trapping to EL2 of instructions that are UNPREDICTABLE

For an instruction that is UNPREDICTABLE or CONSTRAINED UNPREDICTABLE, when the instruction is disabled or trapped then it is CONSTRAINED UNPREDICTABLE whether execution of the instruction generates a Hyp Trap exception.

Note

UNPREDICTABLE and CONSTRAINED UNPREDICTABLE behavior must not perform any function that cannot be performed at the current or lower Exception level using instructions that are not UNPREDICTABLE and are not CONSTRAINED UNPREDICTABLE. This means that disabling or trapping an instruction changes the set of instructions that might be executed in Non-secure state at EL1 or EL0. This indirectly affects the permitted behavior of UNPREDICTABLE and CONSTRAINED UNPREDICTABLE instructions.

If no instructions are trapped, the attempted execution of an UNPREDICTABLE instruction in a Non-secure EL1 or EL0 mode must not generate a Hyp Trap exception.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Disabling or enabling EL2 use of AArch32 deprecated functionality

Table G1-37 shows the deprecated AArch32 functionality that might have disable controls in the HSCTLR:

- The SED control is always implemented.
- Whether each of the ITD, CP15BEN controls is implemented is IMPLEMENTATION DEFINED. If a control is not implemented, then the associated functionality cannot be disabled.

These HSCTLR controls apply only to execution at EL2 using AArch32. When an instruction is disabled by one of these controls, it is UNDEFINED at EL2, meaning it is undefined in Hyp mode.

Table G1-37 EL2 controls for disabling and enabling EL2 use of AArch32 deprecated functionality

Deprecated AArch32 functionality	Instruction enable or disable in the HSCTLR	Disabled instructions
`SETEND` instructions	SED[a]	SETEND instructions
Some uses of `IT` instructions	ITD[b]	See the HSCTLR.IT description
Accesses to the System register (`coproc==0b1111`) DMB, DSB, and ISB barrier operations	CP15BEN[c]	`MCR` accesses to the CP15DMB, CP15DSB, and CP15ISB

a. `SETEND` instruction disable. `SETEND` instructions are disabled when the value of this field is 1.
b. `IT` instruction disable. If this control is implemented, some uses of `IT` instructions are disabled when the value of this field is 1.
c. System register (`coproc==0b1111`) memory barrier enable. If this control is implemented, the specified register accesses are disabled when the value of CP15BEN is 0.

Note

- These controls have no effect on instructions executed in any mode other than Hyp mode. The SCTLR provides similar controls that apply to execution in other modes.
- The uses of the `IT` instruction, and use of the CP15DMB, CP15DSB, and CP15ISB barrier instructions, are deprecated for performance reasons.

Traps to Hyp mode of Non-secure EL1 accesses to virtual memory control registers

HCR.{TRVM, TVM} trap Non-secure EL1 accesses to the virtual memory control registers to Hyp mode:

HCR.TRVM, for read accesses:

1 Non-secure EL1 reads of the virtual memory control registers are trapped to Hyp mode.

0 This control has no effect on Non-secure EL1 reads of the virtual memory control registers.

HCR.TVM, for write access:

1 Non-secure EL1 writes to the virtual memory control registers are trapped to Hyp mode.

0 This control has no effect on Non-secure EL1 writes to the virtual memory control registers.

Table G1-38 on page G1-5791 shows the registers for which:

- Reads are trapped to Hyp mode when HCR.TRVM is 1.
- Writes are trapped to Hyp mode when HCR.TVM is 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The table also shows how the exceptions are reported in HSR.

Table G1-38 Register read and write accesses trapped when HCR.{TRVM, TVM} are 1

Traps from	Registers	Syndrome reporting in HSR
Non-secure EL1	SCTLR, TTBR0, TTBR1, TTBCR, TTBCR2, DACR, DFSR, IFSR, DFAR, IFAR, ADFSR, AIFSR, PRRR, NMRR, MAIR0, MAIR1, AMAIR0, AMAIR1, CONTEXTIDR	Trapped `MCR` or `MRC` access (`coproc==0b1111`), using EC value `0x03` Trapped `MCRR` or `MRRC` access (`coproc==0b1111`), using EC value `0x04`

Note

These registers are not accessible at EL0.

Disabling Non-secure state execution of `HVC` instructions

HCR.HCD disables Non-secure state execution of `HVC` instructions:

1 `HVC` instructions are UNDEFINED at EL2 and Non-secure EL1. The Undefined Instruction exception is taken from the current Exception level to the current Exception level.

0 `HVC` instruction execution is enabled at EL2 and Non-secure EL1.

Note

`HVC` instructions are always UNDEFINED at EL0.

HCR.HCD is only implemented if EL3 is not implemented. Otherwise, it is RES0. See the HCR register description.

Table G1-39 shows how the exceptions are reported in HSR.

Table G1-39 Instruction that causes exceptions when HCR.HCD is 1

Attempted execution in	Disabled instruction	Syndrome reporting in HSR
Hyp mode	`HVC`	Exception for an unknown reason, using EC value `0x00`
Mode other than Hyp mode	`HVC`	Not applicable

Traps to Hyp mode of Non-secure EL1 execution of TLB maintenance instructions

In the Armv8-A architecture, the System instruction encoding space includes TLB maintenance instructions.

HCR.TTLB traps Non-secure EL1 execution of TLB maintenance instructions to Hyp mode:

1 Any attempt to execute a `TLBI` instruction at Non-secure EL1 is trapped to Hyp mode.

0 This control has no effect on the Non-secure EL1 execution of `TLBI` instructions.

Table G1-40 shows the instructions that are trapped, and how the exceptions are reported in HSR.

Table G1-40 Instructions trapped to Hyp mode when HCR.TTLB is 1

Traps from	Trapped instructions	Syndrome reporting in HSR
Non-secure EL1	TLBIALLIS, TLBIMVAIS, TLBIASIDIS, TLBIMVAAIS, TLBIMVALIS, TLBIMVAALIS, ITLBIALL, ITLBIMVA, ITLBIASID, DTLBIALL, DTLBIMVA, DTLBIASID, TLBIALL, TLBIMVA, TLBIASID, TLBIMVAA, TLBIMVAL, TLBIMVAAL.	Trapped `MCR` or `MRC` access (`coproc==0b1111`), using EC value `0x03`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

These instructions are always UNDEFINED at EL0.

For more information about these instructions, see *The scope of TLB maintenance instructions* on page G5-6004.

Traps to Hyp mode of Non-secure EL1 execution of cache maintenance instructions

HCR.{TSW, TPC, TPU} trap cache maintenance instructions to Hyp mode:

0 The control has no effect on the execution of cache maintenance instructions.

1 Any attempt to execute one of the cache maintenance instructions shown in Table G1-42 at Non-secure EL1 is trapped to Hyp mode.

Table G1-41 Controls for trapping cache maintenance instructions to Hyp mode

Trap control	Trapped instructions
HCR.TSW	Data or unified cache maintenance by set/way
HCR.TPC	Data or unified cache maintenance to point of coherency
HCR.TPU	Cache maintenance to point of unification

Table G1-42 shows the instructions that are trapped to Hyp mode, and how the exceptions are reported in HSR.

Table G1-42 Instructions trapped to Hyp mode when HCR.{TSW, TPC, TPU} are 1

Traps from	Trap control	Trapped instructions	Syndrome reporting in HSR
Non-secure EL1	TSW	DCISW, DCCSW, DCCISW	Trapped `MCR` or `MRC` access (`coproc==0b1111`), using EC value `0x3`
	TPC	DCIMVAC, DCCIMVAC, DCCMVAC	
	TPU	ICIMVAU, ICIALLU, ICIALLUIS, DCCMVAU	

Note

These instructions are always UNDEFINED at EL0.

For more information about these instructions, see *Cache maintenance system instructions* on page K14-8090.

Traps to Hyp mode of Non-secure EL1 accesses to the Auxiliary Control Register

HCR.TAC traps Non-secure EL1 accesses to the Auxiliary Control Registers to Hyp mode:

1 Non-secure EL1 accesses to the Auxiliary Control Registers are trapped to Hyp mode.

0 This control has no effect on Non-secure EL1 accesses to the Auxiliary Control Registers.

Table G1-43 shows the registers for which accesses are trapped, and how the exceptions are reported in HSR:

Table G1-43 Register accesses trapped to Hyp mode when HCR.TAC is 1

Traps from	Registers	Syndrome reporting in HSR
Non-secure EL1	ACTLR and, if implemented, ACTLR2.	Trapped `MCR` or `MRC` access (`coproc==0b1111`) access, using EC value `0x03`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

The ACTLR and ACTLR2 are not accessible at EL0.

Traps to Hyp mode of Non-secure EL0 and EL1 accesses to lockdown, DMA, and TCM operations

The lockdown, DMA, and TCM features of the Armv8-A architecture are IMPLEMENTATION DEFINED. The architecture reserves the encodings of a number of System registers for control of these features.

HCR.TIDCP traps the execution of System register access instructions that access these registers, as follows:

1 At Non-secure EL1, any attempt to execute an `MCR` or `MRC` instruction with a reserved register encoding shown in Table G1-44 is trapped to Hyp mode.

At Non-secure EL0, it is IMPLEMENTATION DEFINED whether attempts to execute `MCR` or `MRC` instructions with reserved register encodings are:

- Trapped to Hyp mode.
- UNDEFINED, and the PE takes the Undefined Instruction exception to Non-secure Undefined mode.

Any lockdown fault in the memory system caused by the use of these operations in Non-secure state generates a Data Abort exception that is taken to Hyp mode.

0 This control has no effect on Non-secure EL0 and EL1 System register access instructions with reserved register encodings shown in Table G1-44.

Note

This means that a Hyp Trap exception taken from Non-secure EL1 to Hyp mode, generated because of a configuration setting in HCR.TIDCP is a higher priority exception than an Undefined Instruction exception generated because either the System register encoding is unallocated or because the register is never accessible at EL1. As *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-5707 shows, this is an exception to the general exception prioritization rules, that prioritize most Undefined Instruction exceptions taken to Undefined mode above traps to EL2.

Table G1-44 shows the register encodings for which accesses are trapped to Hyp mode, and how the exceptions are reported in HSR.

Table G1-44 Encodings trapped to Hyp mode when HCR.TIDCP is 1

Traps from	Register encodings	Syndrome reporting in HSR
Non-secure EL0 and EL1	An access to any of the following encodings: • CRn==c9, opc1=={0-7}, CRm=={c0-c2, c5-c8}, opc2=={0-7}. • CRn==c10, opc1=={0-7}, CRm=={c0, c1, c4, c8}, opc2=={0-7}. • CRn==c11, opc1=={0-7}, CRm=={c0-c8, c15}, opc2=={0-7}.	Trapped `MCR` or `MRC` access (`coproc==0b1111`), using EC value `0x03`

An implementation can also include IMPLEMENTATION DEFINED registers that provide additional controls, to give finer-grained control of the trapping of IMPLEMENTATION DEFINED features.

Note

Arm expects the trapping of Non-secure User mode accesses to these functions to Hyp mode to be unusual, and used only when the hypervisor is virtualizing User mode operation. Arm strongly recommends that unless the hypervisor must virtualize User mode operation, a Non-secure User mode access to any of these functions generates an Undefined Instruction exception, as it would if the implementation did not include EL2. The PE then takes this exception to Non-secure Undefined mode.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Traps to Hyp mode of Non-secure EL1 execution of SMC instructions

HCR.TSC traps Non-secure EL1 execution of SMC instructions to Hyp mode:

1 Any attempt to execute an SMC instruction at Non-secure EL1 is trapped to Hyp mode, regardless of the value of SCR.SCD.

0 This control has no effect on Non-secure EL1 execution of SMC instructions.

Table G1-45 shows how the exceptions are reported in HSR:

Table G1-45 SMC Instruction trapped to Hyp mode when HCR.TSC is 1

Traps from	Trapped instruction	Syndrome reporting in HSR
Non-secure EL1	*SMC* on page F5-4694	Trapped SMC instruction execution in AArch32 state, using EC value 0x13

The Armv8-A architecture permits, but does not require, this trap to apply to conditional SMC instructions that fail their Condition code check, in the same way as with traps on other conditional instructions.

Note

- This trap is implemented only if the implementation includes EL3.
- SMC instructions are always UNDEFINED at EL0.
- HCR.TSC traps execution of the SMC instruction. It is not a routing control for the SMC exception. Hyp Trap and SMC exceptions have different preferred return addresses.

For more information about SMC instructions, see *SMC* on page F5-4694.

Traps to Hyp mode of Non-secure EL0 and EL1 accesses to the ID registers

Other than the MIDR, MPIDR, and PMCR.N, the ID registers are divided into groups, with a trap control in the HCR for each group.

Table G1-46 ID register groups

Trap control	Register group
HCR.TID0	*ID group 0, Primary device identification registers* on page G1-5795
HCR.TID1	*ID group 1, Implementation identification registers* on page G1-5796
HCR.TID2	*ID group 2, Cache identification registers* on page G1-5796
HCR.TID3	*ID group 3, Detailed feature identification registers* on page G1-5796

These controls trap register accesses from Non-secure EL0 or EL1 to Hyp mode, as follows:

HCR.TID0
- **0** This control has no effect on Non-secure EL1 reads of the ID group 0 registers.
- **1** Any attempt at Non-secure EL0 or EL1 to read any register in ID group 0 is trapped to Hyp mode.

HCR.TID1
- **0** This control has no effect on Non-secure EL1 reads of the ID group 1 registers.
- **1** Any attempt at Non-secure EL1 to read any register in ID group 1 is trapped to Hyp mode.

HCR.TID2
- **0** This control has no effect on Non-secure EL1 and EL0 accesses to the ID group 2 registers.
- **1** Any attempt at Non-secure EL0 or EL1 to read any register in ID group 2, and any attempt at Non-secure EL0 or EL1 to write to the CSSELR, is trapped to Hyp mode.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

HCR.TID3 **0** This control has no effect on Non-secure EL1 reads of the ID group 3 registers.

1 Any attempt at Non-secure EL1 to read any register in ID group 3 is trapped to Hyp mode.

For the MIDR and MPIDR, and for PMCR.N, the architecture provides read/write aliases. The original register becomes accessible only from Hyp mode and Secure state, and a Non-secure EL0 or EL1 read of the original register returns the value of the read/write alias. This substitution is invisible to the EL0 or EL1 software reading the register.

Table G1-47 ID register substitution

Register	Original	Alias, EL2 using AArch32
Main ID	MIDR	VPIDR
Multiprocessor Affinity	MPIDR	VMPIDR
Performance Monitors Control Register	PMCR.N	HDCR.HPMN

Reads of the MIDR, MPIDR, or PMCR.N from Hyp mode or Secure state are unchanged by the implementation of EL2, and access the physical registers.

Note

- If the optional Performance Monitors Extension is not implemented, HDCR.HPMN is RES0 and PMCR is reserved.
- HDCR.HPMN also affects whether a Performance Monitors counter can be accessed from Non-secure EL1 or EL0. See the register description of HDCR for more information.
- PMCR contains other fields that identify the implementation. For more information about trapping accesses to the PMCR, see *Traps to Hyp mode of Non-secure EL0 and EL1 accesses to Performance Monitors registers* on page G1-5805.

A reset into AArch32 state sets VPIDR to the MIDR value, VMPIDR to the MPIDR value, and HDCR.HPMN to the PMCR.N value.

ID group 0, Primary device identification registers

These registers identify some top-level implementation choices.

Table G1-48 shows the registers that are in ID group 0 for traps to Hyp mode, and how the exceptions are reported in HSR.

Table G1-48 ID group 0 registers

Traps from	Group 0 registers	Syndrome reporting in HSR
Non-secure EL1	FPSID	Trapped `VMRS` access, for ID group traps, using EC value `0x08`
Non-secure EL0 and EL1	JIDR	Trapped `MCR` or `MRC` access (`coproc==0b1110`), using EC value `0x05`

Note

The FPSID is not accessible at EL0.

If HCPTR.{TCP11, TCP10} traps accesses to SIMD and floating-point functionality, then for a read of FPSID, that trap has priority over this trap.

When the FPSID is accessible, a `VMSR FPSID, <Rt>` instruction is permitted but is ignored. The execution of this `VMSR` instruction is not trapped by the ID group 0 trap.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ID group 1, Implementation identification registers

These registers often provide coarse-grained identification mechanisms for implementation-specific features.

Table G1-49 shows the registers that are in ID group 1 for traps to Hyp mode, and how the exceptions are reported in HSR.

Table G1-49 ID group 1 registers

Traps from	Group 1 registers	Syndrome reporting in HSR
Non-secure EL1	TCMTR, TLBTR, REVIDR, AIDR	Trapped `MCR` or `MRC` access (`coproc==0b1111`), using EC value `0x03`

ID group 2, Cache identification registers

These registers describe and control the cache implementation.

Table G1-50 shows the registers that are in ID group 2 for traps to Hyp mode, and how the exceptions are reported in HSR.

Table G1-50 ID group 2 registers

Traps from	Group 2 registers	Syndrome reporting in HSR
Non-secure EL0 and EL1	CTR, CCSIDR, CLIDR, CSSELR, and, if implemented, CCSIDR2.	Trapped `MCR` or `MRC` access (`coproc==0b1111`), using EC value `0x03`

ID group 3, Detailed feature identification registers

These registers provide detailed information about the features of the implementation.

Note

These registers are called the CPUID registers. There is no requirement for this trap to apply to those registers that the CPUID Identification Scheme defines as reserved. See The CPUID identification scheme on page G4-4993.

Table G1-51 shows the registers that are in ID group 3 for traps to Hyp mode, and how the exceptions are reported in HSR.

Table G1-51 ID group 3 registers

Traps from	Group 3 registers	Syndrome reporting in HSR
Non-secure EL1	MVFR0, MVFR1, MVFR2.	Trapped `VMRS` access for ID group traps, using EC value `0x08`
	ID_PFR0, ID_PFR1, ID_DFR0, ID_AFR0. ID_MMFR0, ID_MMFR1, ID_MMFR2, ID_MMFR3. ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR3, ID_ISAR4, ID_ISAR5. ID_MMFR4, ID_ISAR6, ID_DFR1 are trapped, unless implemented as RAZ, when it is IMPLEMENTATION DEFINED whether HCR.TID3 traps accesses. It is IMPLEMENTATION DEFINED whether HCR.TID3 traps `MRC` accesses to registers with `coproc==0b1111` to encodings in the following range that are not already mentioned in this table `CRn` == c0, `opc1` == 0, `CRm` == {c2-c7}, `opc2` == {0-7}.	Trapped `MCR` or `MRC` access (`coproc==0b1111`), using EC value `0x03`

If HCPTR traps accesses to SIMD and floating-point functionality, then for reads of MVFR0, MVFR1, and MVFR2, that trap has priority over this trap.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Traps to Hyp mode of Non-secure EL0 and EL1 execution of WFE and WFI instructions

HCR.{TWE, TWI} trap Non-secure EL0 and EL1 execution of WFE and WFI instructions to Hyp mode:

HCR.TWE:

1 Any attempt to execute a WFE instruction at Non-secure EL0 or EL1 is trapped to Hyp mode, if the instruction would otherwise have caused the PE to enter a low-power state.

0 This control has no effect on Non-secure EL0 or EL1 execution of WFE instructions.

HCR.TWI:

1 Any attempt to execute a WFI instruction at Non-secure EL0 or EL1 is trapped to Hyp mode, if the instruction would otherwise have caused the PE to enter a low-power state.

0 This control has no effect on Non-secure EL0 or EL1 execution of WFI instructions.

Table G1-52 shows how the exceptions are reported in HSR.

Table G1-52 Instructions trapped to Hyp mode when HCR.{TWE, TWI} are 1

Traps from	Trapped instructions	Syndrome reporting in HSR
Non-secure EL0 and EL1	WFE	Trapped WFI or WFE instruction, using EC value 0x01
	WFI	

The attempted execution of a conditional WFE or WFI instruction is only trapped if the instruction passes its Condition code check.

Note

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

For more information about these instructions, and when they can cause the PE to enter a low-power state, see:

- *Wait For Event and Send Event* on page G1-5764.
- *Wait For Interrupt* on page G1-5767.

Traps to Hyp mode of Non-secure EL0 and EL1 accesses to Activity Monitors registers

If the Activity Monitors Extension is implemented, HCPTR.TAM traps Non-secure EL0 and EL1 accesses to the Activity Monitors registers to Hyp mode:

1 Non-secure EL0 and EL1 accesses to all Activity Monitors registers are trapped to Hyp mode.

0 This control has no effect on Non-secure EL0 and EL1 accesses to the Activity Monitors registers.

Note

- EL2 does not provide traps on Activity Monitor register accesses through the optional memory-mapped external interface.
- If the Activity Monitors Extension is not implemented, HCPTR.TAM is RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G1-53 shows the registers for which accesses are trapped, and how the exceptions are reported in HSR.

Table G1-53 Register accesses trapped to Hyp mode when HDCR.{TPM, TPMCR} are 1

Traps from	Trap control	Registers	Syndrome reporting in HSR
Non-secure EL0 and EL1	TPM	AMCFGR, AMCGCR, AMCNTENCLR0, AMCNTENCLR1, AMCNTENSET0, AMCNTENSET1, AMCR, AMEVTYPER0<n>, or AMEVTYPER1<n>.	Trapped `MCR` or `MRC` access (`coproc==0b1111`), using EC value `0x03`.
		AMEVCNTR0<n>or AMEVCNTR1<n>.	Trapped `MCRR` or `MRRC` access (`coproc==0b1111`), using EC value `0x04`.

General trapping to Hyp mode of Non-secure accesses to the SIMD and floating-point registers

HCPTR.{TCP11, TCP10} trap Non-secure accesses to the SIMD and floating-point registers to Hyp mode:

0b11 All Non-secure accesses to the SIMD and floating-point registers are trapped to Hyp mode. Trapped instructions generate:
- Hyp Trap exceptions, if the exception is taken from Non-secure EL0 or EL1.
- Undefined Instruction exceptions taken to Hyp mode, if the exception is taken from EL2.

0b00 This control has no effect on Non-secure accesses to the SIMD and floating-point registers.

Note

Software must set HCPTR.TCP11 and HCPTR.TCP10 to the same value.

Table G1-54 shows the registers for which accesses are trapped, and how the exceptions are reported in HSR.

Table G1-54 Register accesses trapped to Hyp mode when HCPTR.{TCP11, TCP10} are both 0b11

Traps from	Registers	Syndrome reporting in HSR
Non-secure state	FPSID, MVFR0, MVFR1, MVFR2, FPSCR, FPEXC, and any of the SIMD and floating-point registers Q0-Q15, including their views as D0-D31 registers or S0-S31 registers. See *Advanced SIMD and floating-point System registers* on page G1-5774.	Trapped access to SIMD and floating-point register, resulting from HCPTR, using EC value `0x07`[a]

a. `VMSR` accesses to the FPSID are ignored, but for the purposes of this trap the architecture defines a `VMSR` access to the FPSID from EL1 or higher as an access to a SIMD and floating-point register.

If EL3 is implemented and is using AArch32, and NSACR.{cp11, cp10} are both set to 0, then HCPTR.{TCP11, TCP10} behave as RAO/WI, regardless of their actual value.

For more information about SIMD and floating-point support, see *Advanced SIMD and floating-point support* on page G1-5772.

Enabling access to the SIMD and floating-point registers

FPEXC.EN is an instruction enable that enables access to the SIMD and floating-point registers from all Exception levels, but does not control the following:
- `VMSR` accesses to the FPEXC or FPSID.
- `VMRS` accesses from the FPEXC, FPSID, MVFR0, MVFR1, or MVFR2.

FPEXC.EN is a PL1 control that also applies at EL2. See *Enabling access to the SIMD and floating-point registers* on page G1-5784.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Traps to Hyp mode of Non-secure accesses to Advanced SIMD functionality

If implemented as an RW field, HCPTR.TASE can trap Non-secure execution of Advanced SIMD instructions to Hyp mode, as follows. This trap applies only when HCPTR.{TCP11, TCP10} are both `0`:

1 Any attempt to execute an Advanced SIMD instruction in Non-secure state is trapped to Hyp mode. Trapped instructions generate:
- Hyp Trap exceptions, if the exception is taken from Non-secure EL0 or EL1.
- Undefined Instruction exceptions taken to Hyp mode, if the exception is taken from EL2.

0 This control has no effect on Non-secure execution of Advanced SIMD instructions.

When the control is not implemented, meaning the HCPTR.TASE field is RAZ/WI, the HCPTR does not provide a trap to Hyp mode of the Non-secure execution of Advanced SIMD instructions, other than the HCPTR.{TCP11, TCP10} trap that applies to Non-secure execution of both Advanced SIMD and floating-point instructions.

Table G1-27 on page G1-5783 shows the instructions that are trapped, and how the exceptions are reported in HSR.

Table G1-55 Instructions trapped to Hyp mode when HCPTR.TASE is set to 1

Traps from	Instructions	Syndrome reporting in HSR
Non-secure state	All Advanced SIMD instructions that are not also floating-point instructions. For more information, see *Controls of Advanced SIMD operation that do not apply to floating-point operation* on page E1-3966.	Trapped access to SIMD and floating-point register, resulting from HCPTR, using EC value `0x07`

If EL3 is implemented and is using AArch32, and NSACR.NSASEDIS is 1, then HCPTR.TASE behaves as RAO/WI, regardless of its actual value. This behavior also applies when the HCPTR.TASE control is not implemented.

Traps to Hyp mode of Non-secure EL1 accesses to the CPACR

HCPTR.TCPAC traps Non-secure EL1 accesses to the CPACR to Hyp mode:

1 Non-secure EL1 accesses to the CPACR are trapped to Hyp mode.

0 This control has no effect on Non-secure EL1 accesses to the CPACR.

Table G1-56 shows how the exceptions are reported in HSR:

Table G1-56 Register accesses trapped to Hyp mode when HCPTR.TCPAC is 1

Traps from	Register	Syndrome reporting in HSR
Non-secure EL1	CPACR	Trapped `MCR` or `MRC` access to System register with `coproc==0b1111`, using EC value `0x03`

Note

- The CPACR is not accessible at EL0.
- In Armv7 and earlier versions of the Arm architecture, one use of the CPACR is to identify what coprocessor, or conceptual coprocessor, functionality is implemented. Legacy software might use this identification mechanism. A hypervisor can use this trap to emulate this mechanism. See *Background to the System register interface* on page G1-5770 for more information about this functionality.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Traps to Hyp mode of Non-secure System register accesses to trace registers

If implemented, the HCPTR.TTA control traps System register accesses to the trace registers from Non-secure state to Hyp mode, as follows:

1 Non-secure System register accesses to the trace registers are trapped to Hyp mode. Trapped instructions generate:
- Hyp Trap exceptions, if the exception is taken from Non-secure EL0 or EL1.
- Undefined Instruction exceptions taken to Hyp mode, if the exception is taken from EL2.

0 This control has no effect on Non-secure System register accesses to the trace registers.

If the HCPTR.TTA control is not implemented, then HCPTR.TTA is RAO/WI. See the register description for more information.

Note

- System register accesses to the trace registers use the System register (`coproc==0b1110`) encoding space.
- The ETMv4 architecture does not permit EL0 to access the trace registers. If the Armv8-A architecture is implemented with an ETMv4 implementation, EL0 accesses to the trace registers are UNDEFINED. A resulting Undefined Instruction exception is higher priority than an HCPTR.TTA Hyp Trap exception.
- EL2 does not provide traps on trace register accesses through the optional memory-mapped external debug interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, no side-effects occur before the exception is taken, see *Register access instructions* on page G1-5779.

Table G1-57 shows the registers for which accesses are trapped to Hyp mode when HCPTR.TTA is 1, and how the exceptions are reported in HSR.

Table G1-57 Register accesses trapped to Hyp mode when HCPTR.TTA is 1

Traps from	Registers	Syndrome reporting in HSR
Non-secure state	System register accesses to all implemented trace registers	For accesses using: • `MCR` or `MRC` instructions, trapped `MCR` or `MRC` access (`coproc==0b1110`), using EC value `0x05`. • `MCRR` or `MRRC` instructions, trapped `MCRR` or `MRRC` access (`coproc==0b1110`), using EC value `0x0C`.

If EL3 is implemented and is using AArch32, and NSACR.NSTRCDIS is 1, then HCPTR.TTA behaves as RAO/WI, regardless of its actual value. This behavior applies, also, when the HCPTR.TTA control is not implemented.

Traps to Hyp mode of Non-secure System register accesses to trace filter control registers

If implemented, the HDCR.TTRF control traps System register accesses to the trace filter control registers from Non-secure state to Hyp mode, as follows:

1 Non-secure System register accesses at EL1 to the trace filter control registers are trapped to Hyp mode. Trapped instructions generate Hyp Trap exceptions.

0 This control has no effect on Non-secure System register accesses to the trace registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G1-58 shows the registers for which accesses are trapped to Hyp mode when HDCR.TTRF is 1, and how the exceptions are reported in HSR.

Table G1-58 Register accesses trapped to Hyp mode when HDCR.TTRF is 1

Traps from	Registers	Syndrome reporting in HSR
Non-secure state	TRFCR	For accesses using `MCR` or `MRC` instructions, trapped `MCR` or `MRC` access (`coproc==0b1111`), using EC value `0x03`.

General trapping to Hyp mode of Non-secure EL0 and EL1 accesses to System registers in the (`coproc==0b1111`) encoding space

HSTR.{T0-T3, T5-T13, T15} trap Non-secure EL0 and EL1 accesses, using `MCR`, `MRC`, `MCRR`, or `MRRC` instructions, to the System registers in the (`coproc==0b1111`) encoding space, by:

- The value of the `CRn` argument to the instruction, for `MCR` and `MRC` instructions.
- The value of the `CRm` argument to the instruction, for `MCRR` and `MRRC` instructions.

This applies for the set of `CRn`, or `CRm`, values {c0-c3, c5-c13, c15}.

When an HSTR.Tn trap control is:

1 Non-secure EL1 accesses to the corresponding System registers in the (`coproc==0b1111`) encoding space are trapped to Hyp mode.

EL0 accesses to the corresponding System registers are trapped to Hyp mode if they would not be UNDEFINED if the bit was zero.

0 This control has no effect on Non-secure EL0 or EL1 accesses to System registers.

Note

This means that a Hyp Trap exception taken from EL1 to EL2, generated because of a configuration setting in HSTR.Tn, is a higher priority exception than an Undefined Instruction exception generated because either the System register encoding is unallocated or because a register is never accessible at Non-secure EL1. As *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-5707 shows, this is an exception to the general exception prioritization rules, that prioritize most Undefined Instruction exceptions taken to Undefined mode above traps to EL2. This prioritization includes any access from Non-secure EL1 to a register that is only accessible in Secure state. So, for example, an access to the SCR from Non-secure EL1:

- When the value of HSTR.T1 is 0, generates an Undefined Instruction exception.
- When the value of HSTR.T1 is 1, generates a Hyp Trap exception.

Table G1-59 shows the accesses that are trapped, and how the exceptions are reported in HSR.

Table G1-59 Accesses trapped to Hyp mode when an HSTR.Tn trap is enabled

Traps from	Trap control	Trapped accesses	Syndrome reporting in HSR
Non-secure EL0 and EL1[a]	Tn	`MCR` and `MRC` instructions, with `coproc` set to `0b1111` and `CRn` set to n	Trapped `MCR` or `MRC` access (`coproc==0b1111`), using EC value `0x03`
		`MCRR` and `MRRC` instructions, with `coproc` set to `0b1111` and `CRm` set to n	Trapped `MCRR` or `MRRC` access (`coproc==0b1111`), using EC value `0x04`

a. As described in this section, traps from EL1 apply whenever the value of HSTR.Tn is 1. Traps from EL0 apply only if the value of HSTR.Tn is 1 and the access would not be UNDEFINED if the value of HSTR.Tn was 0.

For example, when HSTR.T7 is 1, considering only accesses from Non-secure EL1:

- Any 32-bit access from a Non-secure PL1 mode using an `MRC` or `MCR` instruction with `coproc` set to `0b1111` and CRn set to c7, is trapped to Hyp mode.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Any 64-bit access from a Non-secure PL1 mode using an MRRC or MCRR instructions with coproc set to 0b1111 and CRm set to c7, is trapped to Hyp mode.

Note

- Bits[4,14] of the HSTR are reserved, RES0. Although the Generic Timer control registers are implemented in the coproc == 0b1111 encoding space with CRn==c14 for an MRC or MCR access, EL2 does not provide a trap on accesses to the Generic Timer System registers.
- An implementation might provide additional controls, in IMPLEMENTATION DEFINED registers, to provide finer-grained control of trapping of IMPLEMENTATION DEFINED features.

System registers in the (coproc==0b1111) encoding space with IMPLEMENTATION DEFINED access permission from EL0

For a System register in the (coproc==0b1111) encoding space, that is accessed using a CRn or CRm value that can be trapped by a HSTR.Tn control, if an access to the register from User mode is UNDEFINED when the value of the corresponding HSTR.Tn trap control is 0, then when that HSTR.Tn trap control is 1, it is IMPLEMENTATION DEFINED whether an access from Non-secure User mode generates:

- A Hyp Trap exception.
- An Undefined Instruction exception taken to Non-secure Undefined mode.

Note

Arm expects that trapping to Hyp mode of Non-secure User mode accesses to System register in the (coproc==0b1111) encoding space will be unusual, and used only when the hypervisor must virtualize User mode operation. Arm recommends that, whenever possible, Non-secure User mode accesses to System register in the (coproc==0b1111) encoding space behave as they would if the processor did not implement EL2, generating an Undefined Instruction exception taken to Non-secure Undefined mode if the architecture does not support the User mode access.

Traps to Hyp mode of Non-secure System register accesses to debug registers

HDCR.{TDRA, TDOSA, TDA} trap Non-secure System register accesses to debug registers to Hyp mode, as follows:

- HDCR.(TDRA, TDA} trap Non-secure EL0 and EL1 accesses.
- HDCR.TDOSA traps Non-secure EL1 accesses.

Note

EL2 does not provide traps of debug register accesses through the optional memory-mapped external debug interface.

System register accesses to the debug registers can have side-effects. When a System register access is trapped to Hyp mode, no side-effects occur before the exception is taken to Hyp mode. See *Register access instructions* on page G1-5779.

Table G1-60 shows the subsections that list the accesses trapped. The subsections describe how the traps are reported in HSR.

Table G1-60 Traps of Non-secure EL0 and EL1 accesses to debug registers

Trap control	Subsection
HDCR.TDRA	*Trapping Non-secure System register accesses to Debug ROM registers* on page G1-5803
HDCR.TDOSA	*Trapping Non-secure System register accesses to powerdown debug registers* on page G1-5803
HDCR.TDA	*Trapping general Non-secure System register accesses to debug registers* on page G1-5803

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

System register accesses to debug registers use the (`coproc==0b1110`) encoding space.

Trapping Non-secure System register accesses to Debug ROM registers

HDCR.TDRA traps Non-secure EL0 and EL1 System register accesses to the Debug ROM registers to Hyp mode:

1 Non-secure EL0 or EL1 System register accesses to the Debug ROM registers are trapped to Hyp mode.

0 This control has no effect on Non-secure EL0 and EL1 System register accesses to the Debug ROM registers.

Table G1-61 shows the register accesses that are trapped, and how the exceptions are reported in HSR:

Table G1-61 Register accesses trapped to Hyp mode when HDCR.TDRA is 1

Traps from	Registers	Syndrome reporting in HSR
Non-secure EL0 and EL1	DBGDRAR, DBGDSAR	For accesses using: • `MCR` or `MRC` instructions, trapped `MCR` or `MRC` access (`coproc==0b1110`), using EC value `0x05`. • `MRRC` instructions, trapped `MRRC` access (`coproc==0b1110`), using EC value `0x0C`.

If HDCR.TDE or HCR.TGE is 1, behavior is as if HDCR.TDRA is 1 other than for the purpose of a direct read.

Trapping Non-secure System register accesses to powerdown debug registers

HDCR.TDOSA traps Non-secure EL1 System register accesses to the powerdown debug registers to Hyp mode:

1 Non-secure EL1 System register accesses to the powerdown debug registers are trapped to Hyp mode.

0 This control has no effect on Non-secure EL1 System register accesses to the powerdown debug registers.

Table G1-62 shows the register accesses that are trapped, and how the exceptions are reported in HSR.

Table G1-62 Register accesses trapped to Hyp mode when HDCR.TDOSA is 1

Traps from	Registers	Syndrome reporting in HSR
Non-secure EL1	DBGOSLSR, DBGOSLAR, DBGOSDLR, DBGPRCR Any IMPLEMENTATION DEFINED integration registers. Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by HDCR.TDOSA.	Trapped `MCR` or `MRC` access (`coproc==0b1110`), using EC value `0x05`

Note

These registers are not accessible at EL0.

If HDCR.TDE or HCR.TGE is 1, behavior is as if HDCR.TDOSA is 1 other than for the purpose of a direct read.

Trapping general Non-secure System register accesses to debug registers

HDCR.TDA traps Non-secure EL0 and EL1 System register accesses to the debug registers that are not mentioned in either of the following:

- *Traps to Hyp mode of Non-secure System register accesses to debug registers* on page G1-5802.
- *Trapping Non-secure System register accesses to powerdown debug registers*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This means that HDCR.TDA traps to Hyp mode Non-secure EL0 and EL1 System register accesses to all debug registers except the following:

- Non-secure System register accesses to DBGDRAR or DBGDSAR. The HDCR.TDRA trap traps these accesses.
- Non-secure System register access to DBGOSLSR, DBGOSLAR, DBGOSDLR, or DBGPRCR. The HDCR.TDOSA trap traps these accesses.

HDCR.TDA does not trap accesses to DBGDTRTXint or DBGDTRRXint when the PE is in Debug state.

When HDCR.TDA is:

1 Non-secure EL0 or EL1 System register accesses to any of the registers shown in Table G1-63 are trapped to Hyp mode.

0 This control has no effect on Non-secure EL0 or EL1 System register accesses.

Table G1-63 shows how the exceptions are reported in HSR.

Table G1-63 Accesses trapped to Hyp mode when HDCR.TDA is 1

Traps from	Trapped accesses	Syndrome reporting in HSR
Non-secure EL0 and EL1	Accesses to the DBGDIDR, DBGDSCRint, DBGDCCINT, DBGDTRRXint, DBGDTRTXint, DBGWFAR, DBGVCR, DBGDSCRext, DBGDTRTXext, DBGDTRRXext, DBGBVR<n>, DBGBCR<n>, DBGBXVR<n>, DBGWCR<n>, DBGWVR<n>, DBGCLAIMSET, DBGCLAIMCLR, DBGAUTHSTATUS, DBGDEVID, DBGDEVID1, DBGDEVID2, and DBGOSECCR	For accesses using `MCR` or `MRC` instructions, trapped `MCR` or `MRC` access (`coproc==0b1110`), using EC value `0x05`
	`STC` accesses to DBGDTRRXint. `LDC` accesses to DBGDTRTXint.	Trapped `LDC` or `STC` access, using EC value `0x06`

If HDCR.TDE or HCR.TGE is 1, behavior is as if HDCR.TDA is 1 other than for the purpose of a direct read.

Traps to Hyp mode of Non-secure EL0 and EL1 accesses to the Generic Timer registers

CNTHCTL.{PL1PCEN, PL1PCTEN} trap Non-secure EL0 and EL1 accesses to the Generic Timer registers to Hyp mode, as follows:

- CNTHCTL.PL1PCEN traps Non-secure EL0 and EL1 accesses to the physical timer registers.
- CNTHCTL.PL1PCTEN traps Non-secure EL0 and EL1 accesses to the physical counter register.

For each of these controls:

1 This control has no effect on Non-secure EL0 and EL1 accesses to the registers shown in Table G1-64 on page G1-5805.

0 Non-secure EL0 and EL1 accesses are trapped to Hyp mode.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G1-64 shows the registers for which accesses are trapped, and how the exceptions are reported in HSR.

Table G1-64 Register accesses trapped to Hyp mode by CNTHCTL trap controls

Traps from	Trap control	Registers	Syndrome reporting in HSR
Non-secure EL0 and EL1	PL1PCEN	CNTP_CTL, CNTP_CVAL, CNTP_TVAL	For accesses using: • `MCR` or `MRC` instructions, trapped `MCR` or `MRC` access (`coproc==0b1111`), using EC value `0x03` • `MCRR` or `MRRC` instructions, trapped `MCRR` or `MRRC access` (`coproc==0b1111`), using EC value `0x04`
	PL1PCTEN	CNTPCT	Trapped `MCRR` or `MRRC` access (`coproc==0b1110`), using EC value `0x04`

Traps to Hyp mode of Non-secure EL0 and EL1 accesses to Performance Monitors registers

If the Performance Monitors Extension is implemented, HDCR.{TPM, TPMCR} trap Non-secure EL0 and EL1 accesses to the Performance Monitors registers to Hyp mode:

HDCR.TPM:

1 Non-secure EL0 and EL1 accesses to all Performance Monitors registers are trapped to Hyp mode.

0 This control has no effect on Non-secure EL0 and EL1 accesses to the Performance Monitors registers.

HDCR.TPMCR:

1 Non-secure EL0 and EL1 accesses to the Performance Monitors Control Register are trapped to Hyp mode.

Note

The conditions for this trap are identical to those for the trap controlled by HDCR.TPM

0 This control has no effect on Non-secure EL0 and EL1 accesses to the Performance Monitors Control Registers.

Note

- EL2 does not provide traps on Performance Monitor register accesses through the optional memory-mapped external debug interface.
- If the Performance Monitors Extension is not implemented, HDCR.{TPM, TPMCR} are RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G1-65 shows the registers for which accesses are trapped, and how the exceptions are reported in HSR.

Table G1-65 Register accesses trapped to Hyp mode when HDCR.{TPM, TPMCR} are 1

Traps from	Trap control	Registers	Syndrome reporting in HSR
Non-secure EL0 and EL1	TPM	PMCR, PMCNTENSET, PMCNTENCLR, PMOVSR, PMSWINC, PMSELR, PMCEID0, PMCEID1, PMCCNTR, PMXEVTYPER, PMXEVCNTR, PMUSERENR, PMINTENSET, PMINTENCLR, PMOVSSET, PMEVCNTR<n>, PMEVTYPER<n>, PMCCFILTR	For accesses using: • `MCR` or `MRC` instructions, trapped `MCR` or `MRC` access (`coproc==0b1111`), using EC value `0x03`. • `MCRR` or `MRRC` instructions, trapped `MCRR` or `MRRC` access (`coproc==0b1111`), using EC value `0x04`.
	TPMCR	PMCR	Trapped `MCR` or `MRC` access (`coproc==0b1111`), using EC value `0x03`

Note

HDCR.HPMN affects whether a counter can be accessed from Non-secure EL1 or EL0. See the register description of HDCR for more information.

Traps to Hyp mode of Non-secure EL1 accesses to the RAS error record registers

HCR2.TERR traps Non-secure EL1 accesses to the RAS ER* registers to Hyp mode. For more information on the RAS ER* registers, see the *ARM® Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for the ARMv8-A architecture profile*.

Table G1-66 Register accesses trapped to Hyp mode when HCR2.TERR is 1

Traps from	Trap control	Registers	Syndrome reporting in HSR
Non-secure EL0 and EL1	TERR	ERRIDR, ERRSELR, ERXADDR, ERXADDR2, ERXCTLR, ERXCTLR2, ERXFR, ERXFR2, ERXMISC0, ERXMISC1, ERXMISC2, ERXMISC3, ERXMISC4, ERXMISC5, ERXMISC6, ERXMISC7, ERXSTATUS.	For accesses using: • `MCR` or `MRC` instructions, trapped `MCR` or `MRC` access (`coproc==0b1111`), using EC value `0x03`. • `MCRR` or `MRRC` instructions, trapped `MCRR` or `MRRC` access (`coproc==0b1111`), using EC value `0x04`.

G1.21.4 EL3 configurable controls

Table G1-67 shows the System registers that contain these controls.

Table G1-67 System registers that contain instruction enables and disables, and trap controls

Register name	Register description
SCR	Secure Configuration Register
NSACR	Non-secure Access Control Register

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G1-68 summarizes the controls.

Table G1-68 EL3 Instruction enables and disables, and trap controls

Control	Type of control[a]	Trap
SCR.{TWE, TWI}	T	*Traps to Monitor mode of the execution of WFE and WFI instructions in modes other than Monitor mode* on page G1-5808
SCR.HCE	E	*Enabling EL2 and Non-secure EL1 execution of HVC instructions* on page G1-5809
SCR.SCD	D	*Disabling SMC instructions* on page G1-5809
NSACR.NSTRCDIS	D	*Disabling Non-secure System register access to the trace registers* on page G1-5810
SDCR.TTRF	T	*Traps to Monitor mode of System register accesses to the trace filter control registers* on page G1-5810
NSACR.{cp11, cp10}	E	*Enabling Non-secure access to SIMD and floating-point functionality* on page G1-5810
NSACR.NSASEDIS	D	*Disabling Non-secure access to Advanced SIMD functionality* on page G1-5811
SCR.TERR	T	*Traps to Monitor mode of accesses to RAS error record registers* on page G1-5809

a. See Table G1-69.

Table G1-69 Control types, for AArch32 EL3 controls

Abbreviation	Type	See
D	Disable	*Instruction enables and instruction disables* on page G1-5778
E	Enable	*Instruction enables and instruction disables* on page G1-5778
T	Trap	*Trap controls* on page G1-5778

Also see the following:

- *Register access instructions* on page G1-5779.
- *Instructions that fail their Condition code check*.
- *Trapping to EL3 of instructions that are UNPREDICTABLE* on page G1-5808.

Instructions that fail their Condition code check

For UNDEFINED instructions that fail their Condition code check, see *Conditional execution of undefined instructions* on page G1-5740.

For an instruction that has a Monitor trap set, that fails its Condition code check:

- Unless the trap description states otherwise, it is IMPLEMENTATION DEFINED whether the instruction:
  - Generates a Monitor Trap exception.
  - Executes as a `NOP`.

Any implementation must be consistent in its handling of instructions that fail their Condition code check. This means that:

- Whenever a Monitor trap is set on such an instruction it must either:
  - Always generate a Monitor trap exception.
  - Always treat the instruction as a `NOP`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The IMPLEMENTATION DEFINED part of the requirements of *Conditional execution of undefined instructions* on page G1-5740 must be consistent with the handling of Monitor traps on instructions that fail their Condition code check. Table G1-70 shows this:

Table G1-70 Consistent handling of instructions that fail their Condition code check

Behavior of conditional UNDEFINED instruction[a]	Monitor trap on instruction that fails its Condition code check[b]
Executes as a `NOP`	Executes as a `NOP`
Generates an Undefined Instruction exception	Generates a Monitor trap exception

a. As defined in *Conditional execution of undefined instructions* on page G1-5740. In Non-secure EL0 and EL1 modes, this applies only if no Monitor trap is set for the instruction, otherwise see the behavior in the other column of the table.

b. For a trapped instruction executed in a Non-secure EL1 or EL0 mode.

Note

When SCR{TWE, TWI} is set so that conditional `WFE` and `WFI` instructions are trapped to Monitor mode, the attempted execution of a conditional `WFE` or `WFI` instruction is only trapped if the instruction passes its Condition code check. See *Traps to Monitor mode of the execution of WFE and WFI instructions in modes other than Monitor mode*.

Trapping to EL3 of instructions that are UNPREDICTABLE

For an instruction that is UNPREDICTABLE, when the instruction is disabled or trapped then it is CONSTRAINED UNPREDICTABLE whether execution of the instruction generates a Monitor Trap exception.

Note

UNPREDICTABLE and CONSTRAINED UNPREDICTABLE behavior must not perform any function that cannot be performed at the current or lower Exception level using instructions that are not UNPREDICTABLE and are not CONSTRAINED UNPREDICTABLE. This means that disabling or trapping an instruction changes the set of instructions that might be executed in modes other than Monitor mode. This affects, indirectly, the permitted behavior of UNPREDICTABLE and CONSTRAINED UNPREDICTABLE instructions.

If no instructions are trapped, the attempted execution of an UNPREDICTABLE instruction in a mode other than Monitor mode must not generate a Monitor Trap exception.

Traps to Monitor mode of the execution of `WFE` and `WFI` instructions in modes other than Monitor mode

SCR{TWE, TWI} trap `WFE` and `WFI` instructions to Monitor mode:

SCR.TWE
- **1** Any attempt to execute a `WFE` instruction in any mode other than Monitor mode is trapped to Monitor mode, if the instruction would otherwise have caused the PE to enter a low-power state.
- **0** This control has no effect on the execution of `WFE` instructions.

SCR.TWI
- **1** Any attempt to execute a `WFI` instruction in any mode other than Monitor mode is trapped to Monitor mode, if the instruction would otherwise have caused the PE to enter a low-power state.
- **0** This control has no effect on the execution of `WFI` instructions.

For PL0 and PL1, these traps apply to `WFE` and `WFI` instruction execution in both Security states.

The attempted execution of a conditional `WFE` or `WFI` instruction is only trapped if the instruction passes its Condition code check.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

For more information about these instructions, and when they can cause the PE to enter a low-power state, see:

- *Wait For Event and Send Event* on page G1-5764.
- *Wait For Interrupt* on page G1-5767.

Traps to Monitor mode of accesses to RAS error record registers

SCR.TERR traps accesses to the RAS ER* registers from modes other than Monitor mode to Monitor mode.

Table G1-71 Register accesses trapped to EL3 when SCR.TERR is 1

Traps from	Registers	Syndrome reporting in ESR_EL3
AArch32 state	ERRIDR, ERRSELR, ERXADDR, ERXADDR2, ERXCTLR, ERXCTLR2, ERXFR, ERXFR2, ERXMISC0, ERXMISC1, ERXMISC2, ERXMISC3, ERXMISC4, ERXMISC5, ERXMISC6, ERXMISC7, ERXSTATUS.	For accesses using: • MCR or MRC instructions, trapped MCR or MRC access (coproc==0b1111), using EC value 0x03 • MCRR or MRRC instructions, trapped MCRR or MRRC access, (coproc==0b1111) using EC value 0x04

This trap control applies to accesses from both Security states.

Enabling EL2 and Non-secure EL1 execution of HVC instructions

SCR.HCE enables EL2 and Non-secure EL1 execution of HVC instructions:

1 HVC instruction execution is enabled at EL2 and Non-secure EL1.

0 HVC instructions are:

- UNDEFINED at Non-secure EL1. The Undefined Instruction exception is taken to Undefined mode.
- CONSTRAINED UNPREDICTABLE at EL2. The behavior must be one of the following:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.

Note

- If EL2 is not implemented, SCR.HCE is RES0 and HVC is UNDEFINED.
- HVC instructions are always UNDEFINED at EL0 and in Secure state.

Disabling SMC instructions

SCR.SCD disables SMC instructions:

1

In Non-secure state

SMC instructions are UNDEFINED. The Undefined Instruction exception is taken from the current Exception level to the current Exception level.

In Secure state

Behavior is one of the following:

- The instruction is UNDEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The instruction executes as a NOP.

0 `SMC` instructions are enabled.

Note

- `SMC` instructions are always UNDEFINED at EL0.
- When the value of HCR.TSC is 1, any attempted execution of an `SMC` instruction at Non-secure EL1 is trapped to EL2, regardless of the value of SCR.SCD, see *Traps to Hyp mode of Non-secure EL1 execution of SMC instructions* on page G1-5794. As *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-5707 shows, this is an exception to the general exception prioritization rules, that prioritize most Undefined Instruction exceptions taken to Undefined mode above traps to a higher Exception level.

Disabling Non-secure System register access to the trace registers

NSACR.NSTRCDIS disables Non-secure System register accesses to the trace registers, from all Privilege levels:

1 Non-secure state accesses are disabled. Secure state accesses are enabled. If the PE is in Non-secure state:

- CPACR.TRCDIS behaves as RAO/WI, regardless of its actual value. See *Traps to Undefined mode of PL0 and PL1 System register accesses to trace registers* on page G1-5782.

  This behavior applies even if the CPACR.TRCDIS control is not implemented. See the referenced section for more information.
- HCPTR.TTA behaves as RAO/WI, regardless of its actual value. See *Traps to Hyp mode of Non-secure System register accesses to trace registers* on page G1-5800.

0 There is no effect on accesses to CPACR.TRCDIS and HCPTR.TTA.

Note

- System register accesses to the trace registers use the (`coproc==0b1111`) encoding space.
- NSACR.NSTRCDIS might be implemented as RAZ/WI. See the NSACR register description for more information.
- The ETMv4 architecture does not permit EL0 to access the trace registers. If the Armv8-A architecture is implemented with an ETMv4 implementation, EL0 accesses to the trace registers are UNDEFINED.
- EL3 does not provide Non-secure access controls on trace register accesses through the optional memory-mapped external debug interface.

Traps to Monitor mode of System register accesses to the trace filter control registers

SDCR.TTRF traps any System register accesses to trace filter control registers to Monitor mode:

1 Any attempt to access a trace filter control register in any mode other than Monitor mode is trapped to Monitor mode.

0 This control has no effect.

Enabling Non-secure access to SIMD and floating-point functionality

NSACR.{cp11, cp10} enable Non-secure access to the SIMD and floating-point registers, from all Privilege levels:

0b11 All accesses, from both Security states, are enabled.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

0b00 Non-secure state accesses are disabled. Secure state accesses are enabled. If the PE is in Non-secure state:

- CPACR.{cp11, cp10} behave as RAZ/WI. See *Enabling PL0 and PL1 accesses to the SIMD and floating-point registers* on page G1-5783.
- HCPTR.{TCP11, TCP10} behave as RAO/WI. See *General trapping to Hyp mode of Non-secure accesses to the SIMD and floating-point registers* on page G1-5798.

Note

Software must set NSACR.cp11 and NSACR.cp10 to the same value.

For more information about SIMD and floating-point support, see *Advanced SIMD and floating-point support* on page G1-5772.

Disabling Non-secure access to Advanced SIMD functionality

NSACR.NSASEDIS disables Non-secure accesses to the Advanced SIMD functionality, from all Privilege levels:

1 Non-secure state accesses are disabled. Secure accesses are enabled. If the PE is in Non-secure state:

- CPACR.ASEDIS behaves as RAO/WI. See *Disabling PL0 and PL1 execution of Advanced SIMD instructions* on page G1-5784.
- HCPTR.TASE behaves as RAO/WI. See *Traps to Hyp mode of Non-secure accesses to Advanced SIMD functionality* on page G1-5799.

These behaviors apply even if one or both of the CPACR.ASEDIS and HCPTR.TASE controls is not implemented. See the referenced sections for more information.

0 There is no effect on CPACR.ASEDIS and HCPTR.TASE.

G1.21.5 Pseudocode description of configurable instruction enables, disables, and traps

The pseudocode function `AArch32.CheckITEnabled()` checks whether the T32 IT instruction is enabled.

The pseudocode function `AArch32.CheckSETENDEnabled()` checks whether the `SETEND` instruction is disabled.

The pseudocode function for `AArch32.CheckForSMCUndefOrTrap()` checks for traps on an `SMC` instruction.

The `AArch32.CheckForWFxTrap()` pseudocode function checks for traps on `WFE` and `WFI` instructions:

Pseudocode description of enabling SIMD and floating-point functionality

The `AArch32.CheckAdvSIMDOrFPEnabled()` and `AArch32.CheckFPAdvSIMDTrap()` pseudocode functions take appropriate action if an SIMD or floating-point instruction is used when the SIMD and floating-point functionality is not enabled or is trapped.

The `CheckAdvSIMDOrVFPEnabled()`, `CheckAdvSIMDEnabled()`, and `CheckVFPEnabled()` wrapper functions support the `AArch32.CheckAdvSIMDOrFPEnabled()` and `AArch32.CheckFPAdvSIMDTrap()` functions.

The `AArch32.CheckAdvSIMDOrFPEnabled()`, `AArch32.CheckFPAdvSIMDTrap()`, `CheckAdvSIMDOrVFPEnabled()`, `CheckAdvSIMDEnabled()`, and `CheckVFPEnabled()` functions are described in Chapter J1 *Armv8 Pseudocode*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter G2
AArch32 Self-hosted Debug

When the PE is using self-hosted debug, it generates *debug exceptions*. This chapter describes the AArch32 self-hosted debug exception model. It is organized as follows:

Introductory information

- *About self-hosted debug* on page G2-5814.
- *The debug exception enable controls* on page G2-5818.

The debug Exception model

- *Routing debug exceptions* on page G2-5819.
- *Enabling debug exceptions from the current Privilege level and Security state* on page G2-5821.
- *The effect of powerdown on debug exceptions* on page G2-5823.
- *Summary of permitted routing and enabling of debug exceptions* on page G2-5824.
- *Pseudocode description of debug exceptions* on page G2-5826.

The debug exceptions

- *Breakpoint Instruction exceptions* on page G2-5827.
- *Breakpoint exceptions* on page G2-5830.
- *Watchpoint exceptions* on page G2-5855.
- *Vector Catch exceptions* on page G2-5869.

Synchronization requirements

The behavior of self-hosted debug after changes to System registers, or after changes to the authentication interface, but before a *Context synchronization event* guarantees the effects of the changes:

- *Synchronization and debug exceptions* on page G2-5876.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G2.1 About self-hosted debug

Self-hosted debug supports debugging through the generation and handling of *debug exceptions*, that are taken using the exception model described in:

- Chapter D1 *The AArch64 System Level Programmers' Model*, if the exception is taken to AArch64 state.
- Chapter G1 *The AArch32 System Level Programmers' Model*, if the exception is taken to AArch32 state.

This section introduces some terms used in describing self-hosted debug, and then introduces the debug exceptions. See:

- *Definition of a debugger in the context of self-hosted debug*.
- *Context ID and Process ID*.

G2.1.1 Definition of a debugger in the context of self-hosted debug

Within this chapter, *debugger* means that part of an operating system, or higher level of system software, that handles debug exceptions and programs the debug System registers. An operating system with rich application environments might provide debug services that support a debugger user interface executing at EL0. From the architectural perspective, the debug services are the debugger.

G2.1.2 Context ID and Process ID

In AArch32 state, the CONTEXTIDR identifies the current *Context ID*, that is used by:

- The debug logic, for breakpoint and watchpoint matching.
- Implemented trace logic, to identify the current process.

When using the Long-descriptor translation table format, the CONTEXTIDR has a single field, PROCID, that is defined as the *Process Identifier* (Process ID). Therefore, in AArch64 state, the Context ID and Process ID are identical when using this translation table format.

When using the Short-descriptor translation table format:

- CONTEXTIDR[31:0] defines the Context ID, that is used for breakpoint and watchpoint matching.
- CONTEXTIDR[31:8] defines the Process ID.
- CONTEXTIDR[7:0] define the ASID. See *Global and process-specific translation table entries* on page G5-5991. This means that, when using the Short-descriptor translation table format, the ASID is always bits[7:0] of the Context ID.

G2.1.3 About debug exceptions

Debug exceptions occur during normal program flow if a debugger has programmed the PE to generate them. For example, a software developer might use a debugger contained in an operating system to debug an application. To do this, the debugger might enable one or more debug exceptions. The debug exceptions that can be generated in an AArch32 stage 1 translation regime are:

- *Breakpoint Instruction exceptions* on page G2-5815.
- *Breakpoint exceptions* on page G2-5815, generated by hardware breakpoints.
- *Watchpoint exceptions* on page G2-5816, generated by hardware watchpoints.
- *Vector Catch exceptions* on page G2-5816.

Note

In addition, *Software Step exceptions* can be generated in stage 1 of an AArch32 translation regime. However, these are always taken to AArch64 state. *Software Step exceptions* on page D2-2426 describes this.

The PE can only generate a particular debug exception when both:

1. Debug exceptions are enabled from the current Exception level and Security state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

See *Enabling debug exceptions from the current Privilege level and Security state* on page G2-5821. Breakpoint Instruction exceptions are always enabled from the current Exception level and Security state.

2. A debugger has enabled that particular debug exception.

   All of the debug exceptions except for Breakpoint Instruction exceptions have an enable control contained in the DBGDSCRext. See *The debug exception enable controls* on page G2-5818.

Note

If *halting is allowed* and EDSCR.HDE is 1, hardware breakpoints and watchpoints cause entry to Debug state instead of causing debug exceptions. In Debug state, the PE is halted.

For the definition of halting is allowed, see *Halting allowed and halting prohibited* on page H2-6927.

When a debug exception is taken to an Exception level that is using AArch32:

- If the debug exception is a Watchpoint exception, it is taken as a Data Abort exception.
- Otherwise, it is taken as a Prefetch Abort exception.

The following list summarizes each of the debug exceptions:

Breakpoint Instruction exceptions

Breakpoint instructions generate these. Breakpoint instructions are instructions that software developers can use to cause exceptions at particular points in the program flow.

The breakpoint instruction in the A32 and T32 instruction sets is `BKPT #<immediate>`. Whenever one of these is committed for execution, the PE takes a Breakpoint Instruction exception.

PE behavior

Breakpoint Instruction exceptions cannot be masked. The PE takes Breakpoint Instruction exceptions regardless of both of the following:

- The current Privilege level and AArch32 mode.
- The current Security state.

For more information, see *Breakpoint Instruction exceptions* on page G2-5827.

Breakpoint exceptions

The Armv8-A architecture provides 2-16 hardware breakpoints. These can be programmed to generate Breakpoint exceptions based on particular instruction addresses, or based on particular PE contexts, or both.

For example, a software developer might program a hardware breakpoint to generate a Breakpoint exception whenever the instruction with address `0x1000` is committed for execution.

The Armv8-A architecture supports the following types of hardware breakpoint for use in stage 1 of an AArch32 translation regime:

- Address:
  - Address Match.
  - Address Mismatch.

  Comparisons are made with the virtual address of each instruction in the program flow.
- Context:
  - Context ID Match. Matches with the Context ID value held in the CONTEXTIDR.
  - VMID Match. Matches with the VMID value held in the VTTBR.
  - Context ID and VMID Match. Matches with both the Context ID and the VMID value.

An Address breakpoint can link to a Context breakpoint, so that the Address breakpoint only generates a Breakpoint exception if the PE is in a particular context when the address match or mismatch occurs.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A breakpoint generates a Breakpoint exception whenever an instruction that causes a match is committed for execution.

PE behavior

If halting is allowed and EDSCR.HDE is 1, hardware breakpoints cause entry to Debug state. That is, they halt the PE. See Chapter H2 *Debug State*.

Otherwise:

- If debug exceptions are enabled, hardware breakpoints cause Breakpoint exceptions.
- If debug exceptions are disabled, hardware breakpoints are ignored.

For more information, see *Breakpoint exceptions* on page G2-5830.

Watchpoint exceptions

The Armv8-A architecture provides 2-16 hardware watchpoints. These can be programmed to generate Watchpoint exceptions based on accesses to particular data addresses, or based on accesses to any address in a data address range.

For example, a software developer might program a hardware watchpoint to generate a Watchpoint exception on an access to any address in the data address range `0x1000 - 0x101F`.

A hardware watchpoint can link to a hardware breakpoint if the hardware breakpoint is a *Linked Context* type. In this case, the watchpoint only generates a Watchpoint exception if the PE is in a particular context when the data address match occurs.

The smallest data address size that a watchpoint can be programmed to match on is a byte. A single watchpoint can be programmed to match on one or more bytes.

A watchpoint generates a Watchpoint exception whenever an instruction that initiates an access that causes a match is committed for execution.

PE behavior

If halting is allowed and EDSCR.HDE is 1, hardware watchpoints cause entry to Debug state. That is, they halt the PE. See Chapter H2 *Debug State*.

Otherwise:

- If debug exceptions are enabled, hardware watchpoints cause Watchpoint exceptions.
- If debug exceptions are disabled, hardware watchpoints are ignored.

For more information, see *Watchpoint exceptions* on page G2-5855.

Vector Catch exceptions

These are used to trap exceptions. The Armv8-A architecture provides two forms of vector catch, *address-matching* and *exception-trapping*. Only one form can be implemented.

Whichever form is implemented, a debugger must enable Vector Catch exceptions for one or more exception vectors by programming the DBGVCR. Generation of Vector Catch exceptions is then as follows:

- For the address-matching form, a Vector Catch exception is generated whenever the virtual address of an instruction matches a vector that Vector Catch exceptions are enabled for.
- For the Exception-trapping form, a Vector Catch exception is generated as part of exception entry for exception types that correspond to vectors that Vector Catch exceptions are enabled for.

PE behavior

If debug exceptions are:

- Enabled, Vector Catch exceptions can be generated.
- Disabled, vector catch is ignored.

For more information, see *Vector Catch exceptions* on page G2-5869.

Table G2-1 on page G2-5817 summarizes PE behavior and shows the location of the pseudocode for each of the debug exceptions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G2-1 PE behavior and pseudocode for each of the debug exceptions

Debug exception	PE behavior if debug exceptions are:		Pseudocode
	Enabled	Disabled	
Breakpoint Instruction exception	Takes Prefetch Abort exception	Takes Prefetch Abort exception	`AArch32.SoftwareBreakpoint()`
Breakpoint exception	Takes Prefetch Abort exception[a]	Ignored	See *Pseudocode description of Breakpoint exceptions taken from AArch32 state* on page G2-5854
Watchpoint exception	Takes Data Abort exception[a]	Ignored	See *Pseudocode description of Watchpoint exceptions taken from AArch32 state* on page G2-5867
Vector Catch exception	Takes Prefetch Abort exception	Ignored	See *Pseudocode description of Vector Catch exceptions* on page G2-5875

a. If halting is allowed and EDSCR.HDE is 1, hardware breakpoints and watchpoints cause the PE to enter Debug state instead of causing debug exceptions. See Chapter H2 *Debug State*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G2.2 The debug exception enable controls

The enable controls for each debug exception are as follows:

Breakpoint Instruction exceptions

None. Breakpoint Instruction exceptions are always enabled.

Breakpoint exceptions

DBGDSCRext.MDBGen, plus an enable control for each breakpoint, DBGBCR<n>.E.

Watchpoint exceptions

DBGDSCRext.MDBGen, plus an enable control for each watchpoint, DBGWCR<n>.E.

Vector Catch exceptions

DBGDSCRext.MDBGen.

In addition, for all debug exceptions other than Breakpoint Instruction exceptions, software must configure the controls that enable debug exceptions from the current Exception level and Security state. See *Enabling debug exceptions from the current Privilege level and Security state* on page G2-5821.

The PE cannot take a debug exception if debug exceptions are disabled from either the current Exception level or the current Security state.

Breakpoint Instruction exceptions are always enabled from the current Exception level and Security state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G2.3 Routing debug exceptions

Debug exceptions are usually routed to Abort mode. However, if EL2 is implemented, the routing of debug exceptions depends on the *Effective values* of HDCR.TDE and HCR.TGE:

If the *Effective value* of {HDCR.TDE, HCR.TGE} is not {0, 0}

Debug exceptions taken from Non-secure state are routed to Hyp mode.

If EL2 is using AArch64 and ARMv8.4-SecEL2 is implemented, debug exceptions taken from Secure EL0 and Secure EL1 may be routed to Secure EL2. For more information, see *Routing debug exceptions* on page D2-2429.

Otherwise

In Non-secure state debug exceptions behave as follows:

- Debug exceptions taken from Non-secure EL1 and Non-secure EL0 are routed to Non-secure Abort mode.
- Breakpoint Instruction exceptions taken from Hyp mode are routed to Hyp mode.
- All other debug exceptions are disabled from Hyp mode.

Note

If EL2 is not implemented, the *Effective value* of HCR.TGE is 0 and the *Effective value* of HDCR.TDE is 0.

Table G2-2, Table G2-3, and Table G2-4 on page G2-5820 show the routing of debug exceptions taken from an Exception level that is using AArch32 to an Exception level that is using AArch32. In these tables:

TDE Means the logical OR of HDCR.TDE and HCR.TGE.

(Hyp mode) Means:

- All debug exceptions other than Breakpoint Instruction exceptions are disabled from this Privilege level.
- Breakpoint Instruction exceptions taken from this Privilege level are taken to Hyp mode.

Table G2-2 Routing when both EL3 and EL2 are implemented

TDE	Target AArch32 mode when executing in:			
	Non-secure:			Secure state
	PL0	PL1	PL2	
0	Non-secure Abort mode	Non-secure Abort mode	(Hyp mode)	Secure Abort mode
1	Hyp mode	Hyp mode	(Hyp mode)	Secure Abort mode

Table G2-3 Routing when EL3 is implemented and EL2 is not implemented

Target AArch32 mode when executing in:	
Non-secure state	Secure state
Non-secure Abort mode	Secure Abort mode

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G2-4 Routing when EL3 is not implemented and EL2 is implemented

TDE	Target AArch32 mode when executing in Non-secure:		
	PL0	**PL1**	**PL2**
0	Non-secure Abort mode	Non-secure Abort mode	(Hyp mode)
1	Hyp mode	Hyp mode	(Hyp mode)

G2.3.1 Pseudocode description of routing debug exceptions

`DebugTarget()` returns the current debug target Exception level. `DebugTargetFrom()` returns the debug target Exception level for the specified Security state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G2.4 Enabling debug exceptions from the current Privilege level and Security state

A debug exception can only be taken if all of the following are true:

- The OS Lock is unlocked.
- `DoubleLockStatus()` == FALSE.
- The debug exception is enabled from the current Privilege level.
- The debug exception is enabled from the current Security state.

Table G2-5 shows when debug exceptions are enabled from the current Privilege level.

Table G2-5 Whether debug exceptions are enabled from the current Privilege level

Current Privilege level	Breakpoint Instruction exceptions	All other debug exceptions
PL2	Enabled	Disabled
PL1	Enabled	Enabled
PL0		

Table G2-6 shows when debug exceptions are enabled from the current Security state.

Table G2-6 Whether debug exceptions are enabled from the current Security state

Current Security state	Breakpoint Instruction exceptions	All other debug exceptions
Non-secure	Enabled	Enabled from PL1 and PL0 only.
Secure	Enabled	Depends on SDCR.SPD and SDER.SUIDEN. See *Disabling debug exceptions from Secure state*.

G2.4.1 Disabling debug exceptions from Secure state

If EL3 is implemented, software executing at EL3 can enable or disable all debug exceptions taken from Secure PL1 other than Breakpoint Instruction exceptions, by using one of:

- The *Secure Privileged Debug* field, SDCR.SPD, if EL3 is using AArch32.
- The *AArch32 Secure Privileged Debug* field, MDCR_EL3.SPD32, if EL3 is using AArch64.

If debug exceptions are disabled from Secure PL1, software executing at Secure PL1 can set the *Secure User Invasive Debug Enable* bit, SDER.SUIDEN, to 1 to enable all debug exceptions taken from Secure PL0 other than Breakpoint Instruction exceptions.

Note

Breakpoint Instruction exceptions are always enabled.

The Armv8-A architecture does not support disabling debug in Non-secure state.

Note

If the boot software that is executed when reset is deasserted programs SUIDEN and SPD so that all debug exceptions are disabled from Secure state, software operating at EL3 never has to switch any of the debug registers between the Security states.

G2.4.2 Pseudocode description of enabling debug exceptions

`AArch64.GenerateDebugExceptions()` determines whether debug exceptions are enabled from the current Exception level and Security state. `AArch64.GenerateDebugExceptionsFrom()` determines whether debug exceptions are enabled from the specified Exception level and Security state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G2.5 The effect of powerdown on debug exceptions

Debug OS Save and Restore sequences on page H6-7035 describes the *powerdown save routine* and the *restore routine*.

When executing either routine, software must use the OS Lock to disable generation of all of the following:

- Breakpoint exceptions.
- Watchpoint exceptions.
- Vector Catch exceptions.

This is because the generation of these exceptions depends on the state of the debug registers, and the state of the debug registers might be lost over these routines.

Debug exceptions other than Breakpoint Instruction exceptions are enabled only if both the OS Lock is unlocked and `DoubleLockStatus()` == FALSE.

Breakpoint Instruction exceptions are enabled regardless of the state of the OS Lock and the OS Double Lock.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G2.6 Summary of permitted routing and enabling of debug exceptions

Behavior is as follows:

Breakpoint Instruction exceptions

These are always enabled, regardless of the current Privilege level and Security state. Table G2-7 shows the routing of these. In the table, n/a means not applicable.

Table G2-7 Routing of Breakpoint Instruction exceptions

Current Security state	HDCR.TDE[a]:	Target when enabled from:		
		PL0	PL1	PL2
Secure	X	Secure Abort mode[b]	Secure Abort mode[b]	n/a
Non-secure	0	Non-secure Abort mode	Non-secure Abort mode	Hyp mode
	1	Hyp mode	Hyp mode	Hyp mode

a. If EL2 is not implemented, behavior is as if the value of this bit is 0. Otherwise, if the value of HCR.TGE is 1, HDCR.TDE is treated as being 1 other than for a direct read of HDCR.

b. If EL3 is implemented and is using AArch32, Secure Abort mode is at EL3. Otherwise, Secure Abort mode is at EL1.

All other debug exceptions

The enabling and permitted routing is controlled by all of the following:

- SDCR.SPD.
- SDER.SUIDEN.
- HDCR.TDE.
- The IMPLEMENTATION DEFINED authentication interface.

Table G2-8 shows the valid combinations of the values of SDCR.SPD, SDER.SUIDEN, HDCR.TDE, and, in the *Auth* column, the input from the IMPLEMENTATION DEFINED authentication interface described by the pseudocode function `AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()`. For each combination, the table shows where debug exceptions are enabled from and where they are taken to.

In the table, n/a means not applicable and a dash, -, means that debug exceptions are disabled from that Exception level.

Table G2-8 Breakpoint, Watchpoint, and Vector Catch exceptions

Debug state	Lock[a]	Current Security state	SPD[b]	Auth[c]	SUIDEN	TDE[d]	Target AArch32 mode when enabled from:		
							PL0	PL1	PL2
Yes	X	X	`0bXX`	X	X	X	-	-	-
No	TRUE	X	`0bXX`	X	X	X	-	-	-
No	FALSE	Secure	`0b00`	FALSE	0	X	-	-	n/a
No	FALSE	Secure	`0b00`	FALSE	1	X	Secure Abort mode[e]	-	n/a
No	FALSE	Secure	`0b00`	TRUE	X	X	Secure Abort mode[e]	Secure Abort mode[e]	n/a
No	FALSE	Secure	`0b10`	X	0	X	-	-	n/a

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G2-8 Breakpoint, Watchpoint, and Vector Catch exceptions (continued)

Debug state	Lock[a]	Current Security state	SPD[b]	Auth[c]	SUIDEN	TDE[d]	Target AArch32 mode when enabled from:		
							PL0	PL1	PL2
No	FALSE	Secure	`0b10`	X	1	X	Secure Abort mode[e]	-	n/a
No	FALSE	Secure	`0b11`	X	X	X	Secure Abort mode[e]	Secure Abort mode[e]	n/a
No	FALSE	Non-secure	`0bXX`	X	X	0	Non-secure Abort mode	Non-secure Abort mode	-
No	FALSE	Non-secure	`0bXX`	X	X	1	Hyp mode	Hyp mode	-

a. The value of (OSLSR_EL1.OSLK == ’1’ || `DoubleLockStatus()`).

b. If EL3 is not implemented, behavior is as if this is `0b11`.

c. See the text that introduces this table for an explanation of the *Auth* on page G2-5824 column. An entry of TRUE indicates that the authentication mechanism permits the debug exceptions to be taken to their default target PE mode.

d. If HCR.TGE is 1, this bit is treated as being 1 other than for a direct read of HDCR. If EL2 is not implemented, behavior is as if TDE is 0.

e. If EL3 is implemented and is using AArch32, Secure Abort mode is at EL3. Otherwise, Secure Abort mode is at EL1

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G2.7 Pseudocode description of debug exceptions

`AArch32.DebugFault()` returns a `FaultRecord()` that indicates that a memory access has generated a debug exception.

The `AArch32.Abort()` function processes `FaultRecord()`, as described in *Abort exceptions* on page G4-5919, and generates:

- Data Abort exceptions for watchpoints.
- Prefetch Abort exceptions for all other debug exceptions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G2.8 Breakpoint Instruction exceptions

This section describes Breakpoint Instruction exceptions in an AArch32 translation regime.

Note

When the PE is executing in EL0 using AArch32 and EL1 is using AArch64, it is using the AArch64 EL1&0 translation regime. A T32 or A32 `BKPT` instruction executed at EL0 can generate a Breakpoint Instruction exception that is taken to an Exception level that is using AArch64. For more information about the handling of these exceptions, see *Breakpoint Instruction exceptions* on page D2-2437.

It contains the following subsections:

- *About Breakpoint Instruction exceptions*.
- *Breakpoint instruction in the A32 and T32 instruction sets*.
- *BKPT instructions as the first instruction in an IT block* on page G2-5828.
- *Exception syndrome information and preferred return address for a BKPT instruction* on page G2-5828.
- *Pseudocode description of Breakpoint Instruction exceptions* on page G2-5829.

G2.8.1 About Breakpoint Instruction exceptions

A *breakpoint* is an event that results from the execution of an instruction, based on either:

- The instruction address, the PE context, or both. This type of breakpoint is called a *hardware breakpoint*.
- The instruction itself. That is, the instruction is a *breakpoint instruction*. These can be included in the program that the PE executes. This type of breakpoint is called a *software breakpoint*.

Breakpoint Instruction exceptions, that this section describes, are software breakpoints. *Breakpoint exceptions* on page G2-5830 describes hardware breakpoints.

There is no enable control for Breakpoint Instruction exceptions. They are always enabled, and cannot be masked.

A Breakpoint Instruction exception is generated whenever a breakpoint instruction is committed for execution, regardless of all of the following:

- The current Exception level.
- The current Security state.
- Whether the *debug target Exception level*, EL_D, is using AArch64 or AArch32.

Note

- EL_D is the Exception level that debug exceptions are targeting. See *Enabling debug exceptions from the current Privilege level and Security state* on page G2-5821.
- Debuggers using breakpoint instructions must be aware of the Armv8 rules for concurrent modification and execution of instructions. See *Concurrent modification and execution of instructions* on page B2-120.

G2.8.2 Breakpoint instruction in the A32 and T32 instruction sets

The breakpoint instruction, in both instruction sets, is:

- `BKPT #<immediate>`

For details of the instruction encoding, see *BKPT* on page F5-4307.

About whether the BKPT instruction is conditional

In the T32 instruction set, `BKPT` instructions are always unconditional.

In the A32 instruction set:

- If the Condition code field is AL, the `BKPT` instruction is unconditional.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If the Condition code field is anything other than AL, behavior is CONSTRAINED UNPREDICTABLE, and is one of the following:
  - The instruction is UNDEFINED.
  - The instruction is treated as a `NOP` instruction.
  - The instruction is executed unconditionally.
  - The instruction is executed conditionally.

G2.8.3 BKPT instructions as the first instruction in an IT block

If the first instruction in an IT block is a T32 `BKPT` instruction, then in an implementation that supports the ITD control, if ITD field that applies to the current Exception level is:

0 The `BKPT` instruction generates a Breakpoint Instruction exception.

1 The combination of `IT` instruction and `BKPT` instruction is UNDEFINED. Either the `IT` instruction or the `BKPT` instruction generates an Undefined Instruction exception.

In such an implementation, to ensure consistent behavior when making the first instruction in one or more IT blocks a `BKPT` instruction, the debugger must replace the `IT` instruction.

An implementation that does not support the ITD control behaves as if the value of the ITD field is 0.

The ITD control fields are:

HSCTLR.ITD Applies to execution at EL2 when EL2 is using AArch32.

SCTLR.ITD Applies to execution at EL0 or EL1 when EL1 is using AArch32.

SCTLR_EL1.ITD
Applies to execution at EL0 using AArch32 when EL1 is using AArch64.

Note

T32 `BKPT` instructions are always unconditional, even when they are inside an IT block. See:

- *Disabling or enabling PL0 and PL1 use of AArch32 deprecated functionality* on page G1-5781.
- *Disabling or enabling EL2 use of AArch32 deprecated functionality* on page G1-5790.

G2.8.4 Exception syndrome information and preferred return address for a BKPT instruction

See the following:

- *Exception syndrome information for a Breakpoint Instruction exception*.
- *Preferred return address for a Breakpoint Instruction exception* on page G2-5829.

Note

Usually, the term *exception syndrome* is used only for exceptions taken to Hyp mode, or to AArch64 state. The referenced section uses the term more generally, to include exception information reported in the IFSR.

Exception syndrome information for a Breakpoint Instruction exception

The PE takes a Breakpoint Instruction exception as either:

- A Prefetch Abort exception if it is taken to PL1. In this case, it is taken to Abort mode.
- A Hyp Trap exception, if it is taken to PL2 because either HCR.TGE or HDCR.TDE is 1. In this case, it is taken to Hyp mode.

If the exception is taken to:

PL1 Abort mode

The PE sets all of the following:

- DBGDSCRext.MOE to `0b0011`, to indicate a Breakpoint Instruction exception.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- IFSR.FS to the code for a debug, `0b00010`.
- The IFAR with an UNKNOWN value.

PL2 Hyp mode

The PE does all of the following:

- Records information about the exception in the *Hypervisor Syndrome Register*, HSR. See Table G2-9.
- Sets DBGDSCRext.MOE to `0b0011`, to indicate a Breakpoint Instruction exception.
- Sets the HIFAR to an unknown value.

Table G2-9 Information recorded in the HSR

HSR field	Information recorded
Exception Class, EC	The PE sets this to the code for a Prefetch Abort exception routed to Hyp mode, `0x20`.
Instruction Length, IL	The PE sets this to: • 0 for a T32 `BKPT` instruction. • 1 for an A32 `BKPT` instruction.
Instruction Specific Syndrome, ISS	**ISS[24:10]** RES0. **ISS[9]** *External Abort type* (EA). The PE sets this to 0. **ISS[8:6]** RES0. **ISS[5:0]** *Instruction Fault Status Code* (IFSC). The PE sets this to the code for a debug exception, `0b100010`.

Note

For information about how debug exceptions can be routed to PL2, see *Routing debug exceptions* on page G2-5819.

Preferred return address for a Breakpoint Instruction exception

The preferred return address is the address of the breakpoint instruction, not the next instruction. This is different to the behavior of other exception-generating instructions, like `SVC`.

G2.8.5 Pseudocode description of Breakpoint Instruction exceptions

`AArch32.SoftwareBreakpoint()` generates a Prefetch Abort exception that is taken from AArch32 state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G2.9 Breakpoint exceptions

This section describes Breakpoint exceptions in stage 1 of an AArch32 translation regime.

The PE is using an AArch32 translation regime when it is executing either:

- At EL1 or higher in an Exception level that is using AArch32.
- At EL0 using AArch32 when EL1 is using AArch32.

This section contains the following subsections:

- *About Breakpoint exceptions*.
- *Breakpoint types and linking of breakpoints* on page G2-5831.
- *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page G2-5840.
- *Breakpoint instruction address comparisons* on page G2-5842.
- *Breakpoint context comparisons* on page G2-5847.
- *Using breakpoints* on page G2-5848.
- *Exception syndrome information and preferred return address for a Breakpoint exception* on page G2-5853.
- *Pseudocode description of Breakpoint exceptions taken from AArch32 state* on page G2-5854.

G2.9.1 About Breakpoint exceptions

A *breakpoint* is an event that results from the execution of an instruction, based on either:

- The instruction address, the PE context, or both. This type of breakpoint is called a *hardware breakpoint.*
- The instruction itself. That is, the instruction is a *breakpoint instruction*. These can be included in the program that the PE executes. This type of breakpoint is called a *software breakpoint*.

Breakpoint exceptions are generated by *Breakpoint debug events*. Breakpoint debug events are generated by hardware breakpoints. Software breakpoints are described in *Breakpoint Instruction exceptions* on page G2-5827.

An implementation can include between 2-16 hardware breakpoints. DBGDIDR.BRPs shows how many are implemented.

To use an implemented hardware breakpoint, a debugger programs the following registers for the breakpoint:

- The *Breakpoint Control Register*, DBGBCR<n>. This contains controls for the breakpoint, for example an enable control.
- The *Breakpoint Value Register*, DBGBVR<n>. This holds a value used for breakpoint matching, that is one of:
  - An instruction virtual address.
  - A Context ID.
- If EL2 is implemented, the *Breakpoint Extended Value Register*, DBGBXVR<n>, that holds a VMID value used for breakpoint matching.

These registers are numbered, so that:

- DBGBCR1, DBGBVR1, and DBGBXVR1 are for breakpoint number one.
- DBGBCR2, DBGBVR2, and DBGBXVR2 are for breakpoint number two.
- …
- …
- DBGBCR<n>, DBGBVR<n>, and DBGBXVR<n> are for breakpoint number <n>.

A debugger can link a breakpoint that is programmed with an address and a breakpoint that is programmed with anything other than an address together, so that a Breakpoint debug event is only generated if both breakpoints match.

For each instruction in the program flow, all of the breakpoints are tested. When a breakpoint is tested, it generates a Breakpoint debug event if all of the following are true:

- The breakpoint is enabled. That is, the breakpoint enable control for it, DBGBCR<n>.E, is 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The conditions specified in the DBGBCR<n> are met.
- The comparisons with the values held in one or both of the DBGBVR<n> and DBGBXVR<n>, as applicable, are successful.
- If the breakpoint is linked to another breakpoint, the comparisons made by that other breakpoint are also successful.
- The instruction is committed for execution.

If all of these conditions are met, the breakpoint generates the Breakpoint debug event regardless of the following:

- Whether the instruction passes its Condition code check.
- The instruction type.

If halting is allowed and EDSCR.HDE is 1, Breakpoint debug events cause entry to Debug state.

Otherwise, if debug exceptions are

- Enabled, Breakpoint debug events generate Breakpoint exceptions
- Disabled, Breakpoint debug events are ignored.

Note

The remainder of this Breakpoint exceptions section, including all subsections, describes breakpoints as generating Breakpoint exceptions. However, the behavior described also applies if breakpoints are causing entry to Debug state.

The debug exception enable controls on page G2-5818 describes the enable controls for Breakpoint debug events.

G2.9.2 Breakpoint types and linking of breakpoints

Each implemented breakpoint is one of the following:

- A *context-aware* breakpoint. This is a breakpoint that can be programmed to generate a Breakpoint exception on any one of the following:
  - An instruction address match.
  - An instruction address mismatch.
  - A Context ID match, with the value held in the CONTEXTIDR.
  - A VMID match, with the value held in the VTTBR.
  - Both a Context ID match and a VMID match.
- A breakpoint that is not context-aware. These can only be programmed to generate a Breakpoint exception on an instruction address match or an instruction address mismatch.

DBGDIDR.CTX_CMPs shows how many of the implemented breakpoints are context-aware breakpoints. At least one implemented breakpoint must be context-aware. The context-aware breakpoints are the highest numbered breakpoints.

Any breakpoint that is programmed to generate a Breakpoint exception on an instruction address match or mismatch is categorized as an *Address breakpoint*. Breakpoints that are programmed to match on anything else are categorized as *Context breakpoints*.

When a debugger programs a breakpoint to be an Address or a Context breakpoint, it must also program that breakpoint so that it is either:

- Used in isolation. In this case, the breakpoint is called an *Unlinked breakpoint*.
- Enabled for linking to another breakpoint. In this case, the breakpoint is called a *Linked breakpoint*.

By linking an Address breakpoint and a Context breakpoint together, the debugger can create a breakpoint pair that only generates a Breakpoint exception if the PE is in a particular context when an instruction address match or mismatch occurs. For example, a debugger might:

1. Program breakpoint number one to be a *Linked Address Match breakpoint*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

2. Program breakpoint number five to be a *Linked Context ID Match breakpoint*.

3. Link these two breakpoints together. A Breakpoint exception is only generated if both the instruction address matches and the Context ID matches.

The *Breakpoint Type* field for a breakpoint, DBGBCR<n>.BT, controls the breakpoint type and whether the breakpoint is enabled for linking. If BT[0] is 1, the breakpoint is enabled for linking.

Address breakpoints can be programmed to generate Breakpoint exceptions on addresses that are halfword-aligned but not word-aligned. This makes it possible to breakpoint on T32 instructions. See *Specifying the halfword-aligned address that an Address breakpoint matches on* on page G2-5842.

Rules for linking breakpoints

The rules for breakpoint linking are as follows:

- Only Linked breakpoint types can be linked.
- Any type of Linked Address breakpoint can link to any type of Linked Context breakpoint. The *Linked Breakpoint Number* field, DBGBCR<n>.LBN, for the Linked Address breakpoint specifies the particular Linked Context breakpoint that the Linked Address breakpoint links to, and:
  - DBGBCR<n>.{SSC, HMC, PMC} for the Linked Address breakpoint define the execution conditions that the breakpoint pair generates Breakpoint exceptions for. See *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page G2-5840.
  - DBGBCR<n>.{SSC, HMC, PMC} for the Linked Context breakpoint are ignored.
- Linked Context breakpoint types can only be linked to. The LBN field for Context breakpoints is therefore ignored.
- Linked Address breakpoints cannot link to watchpoints. The LBN field can therefore only specify another breakpoint.
- If a Linked Address breakpoint links to a breakpoint that is not context-aware, the behavior of the Linked Address breakpoint is CONSTRAINED UNPREDICTABLE. See *Other usage constraints for Address breakpoints* on page G2-5852.
- If a Linked Address breakpoint links to an Unlinked Context breakpoint, the Linked Address breakpoint never generates any Breakpoint exceptions.
- Multiple Linked Address breakpoints can link to a single Linked Context breakpoint.

  Note

  Multiple Linked watchpoints can also link to a single Linked Context breakpoint. *Watchpoint exceptions* on page G2-5855 describes watchpoints.

These rules mean that a single Linked Context breakpoint might be linked to by all, or any combination of, the following:

- Multiple Linked Address Match breakpoints.
- Multiple Linked Address Mismatch breakpoints.
- Multiple Linked watchpoints.

It is also possible that a Linked Context breakpoint might have no breakpoints or watchpoints linked to it.

Figure G2-1 on page G2-5833 shows an example of permitted breakpoint and watchpoint linking.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

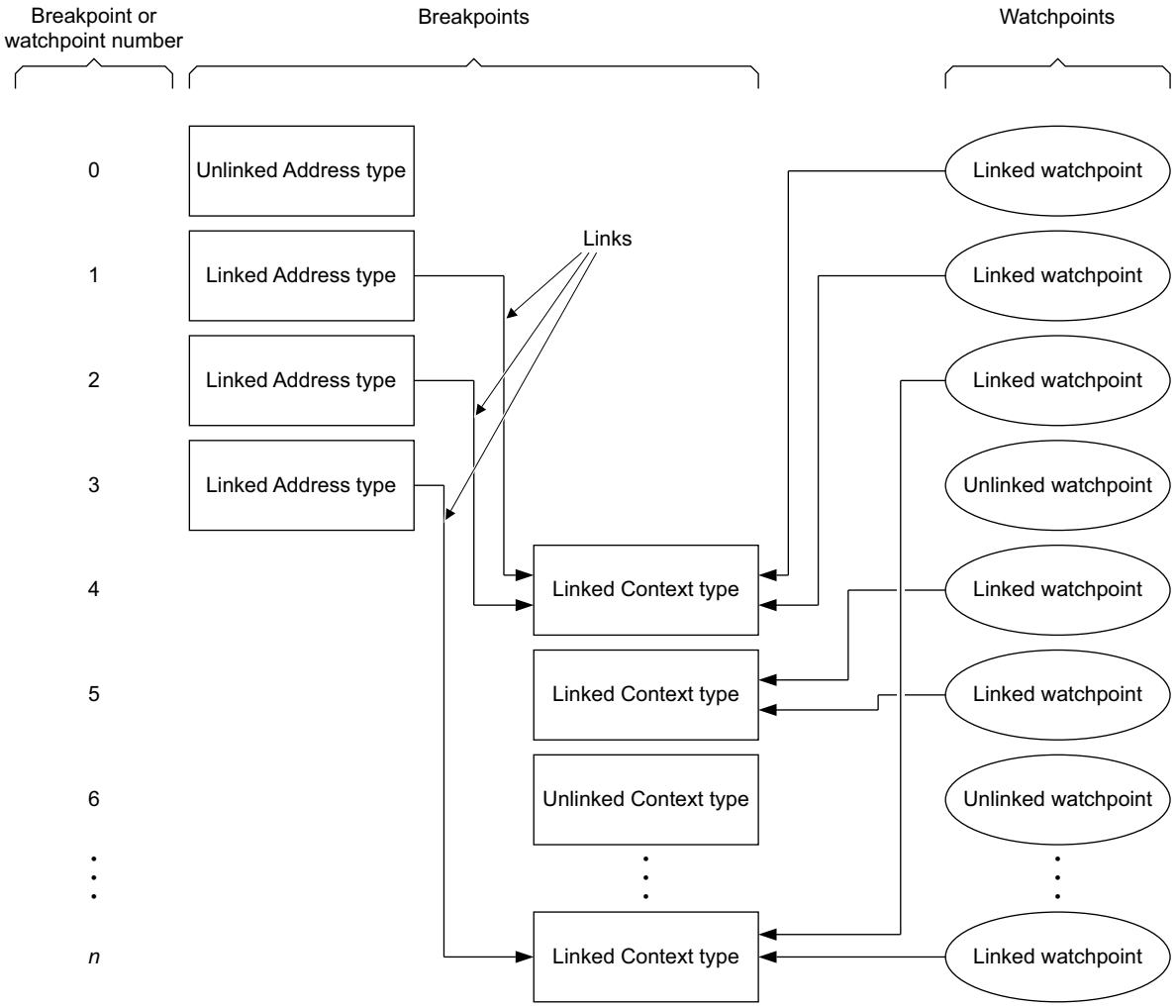


Figure G2-1 The role of linking in Breakpoint and Watchpoint exception generation

In Figure G2-1, each Linked Address breakpoint can only generate a Breakpoint exception if the comparisons made by both it, and the Linked Context breakpoint that it links to, are successful. Similarly, each Linked watchpoint can only generate a Watchpoint exception if the comparisons made by both it, and the Linked Context breakpoint that it links to, are successful.

Breakpoint types defined by DBGBCRn.BT

The following list provides more detail about each breakpoint type:

`0b0000`, Unlinked Address Match breakpoint

Generation of a Breakpoint exception depends on both:

- DBGBCR<n>.{SSC, HMC, PMC}. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page G2-5840.
- A successful address match, as described in *Breakpoint instruction address comparisons* on page G2-5842.

DBGBCR<n>.LBN for this breakpoint is ignored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b0001`, Linked Address Match breakpoint

Generation of a Breakpoint exception depends on all of the following:

- DBGBCR<n>.{SSC, HMC, PMC} for this breakpoint. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page G2-5840.
- A successful address match defined by this breakpoint, as described in *Breakpoint instruction address comparisons* on page G2-5842.
- A successful context match defined by the Linked Context breakpoint that this breakpoint links to.

DBGBCR<n>.LBN for this breakpoint selects the Linked Context breakpoint that this breakpoint links to.

`0b0010`, Unlinked Context ID Match breakpoint

BT == `0b0010` is a reserved value if the breakpoint is not a context-aware breakpoint.

For context-aware breakpoints, generation of a Breakpoint exception depends on both:

- DBGBCR<n>.{SSC, HMC, PMC}. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page G2-5840.
- A successful Context ID match, as described in *Breakpoint context comparisons* on page G2-5847.

The value of DBGBVR<n>.ContextID is compared with the current Context ID.

CONTEXTIDR_EL2 holds the current Context ID when all of:

- The implementation includes ARMv8.1-VHE.
- EL2 is implemented and enabled in the current Security state.
- EL2 using AArch64 and the value of HCR_EL2.E2H is 1.
- The PE is executing at EL0 and HCR_EL2.TGE is 1, or the PE is executing at EL2.

Otherwise, CONTEXTIDR holds the current Context ID.

DBGBCR<n>.{LBN, BAS} for this breakpoint are ignored

`0b0011`, Linked Context ID Match breakpoint

BT == `0b0011` is a reserved value if the breakpoint is not a context-aware breakpoint.

For context-aware breakpoints, either:

- This breakpoint does not generate any Breakpoint exceptions, if no Linked breakpoints or Linked watchpoints link to it.
- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address breakpoint that links to this breakpoint, see *Breakpoint instruction address comparisons* on page G2-5842.
  - A successful Context ID match defined by this breakpoint, as described in *Breakpoint context comparisons* on page G2-5847.
- Generation of a Watchpoint exception depends on both:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see *Watchpoint data address comparisons* on page G2-5859.
  - A successful Context ID match defined by this breakpoint, as described in *Breakpoint context comparisons* on page G2-5847.

The value of DBGBVR<n>.ContextID is compared with the current Context ID.

CONTEXTIDR_EL2 holds the current Context ID when all of:

- The implementation includes ARMv8.1-VHE.
- EL2 is implemented and enabled in the current Security state.
- EL2 using AArch64 and the value of HCR_EL2.E2H is 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The PE is executing at EL0 and HCR_EL2.TGE is 1, or the PE is executing at EL2.

Otherwise, CONTEXTIDR holds the current Context ID.

DBGBCR<n>.{LBN, SSC, HMC, BAS PMC} for this breakpoint are ignored.

`0b0100`, Unlinked Address Mismatch breakpoint

Generation of a Breakpoint exception depends on both:

- DBGBCR<n>.{SSC, HMC, PMC}. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page G2-5840.
- A successful address mismatch, as described in *Breakpoint instruction address comparisons* on page G2-5842.

DBGBCR<n>.LBN for this breakpoint is ignored.

`0b0101`, Linked Address Mismatch breakpoint

Generation of a Breakpoint exception depends on all of the following:

- DBGBCR<n>.{SSC, HMC, PMC}. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page G2-5840.
- A successful address mismatch defined by this breakpoint, as described in *Breakpoint instruction address comparisons* on page G2-5842.
- A successful context match defined by the Linked Context breakpoint that this breakpoint links to.

DBGBCR<n>.LBN for this breakpoint selects the Linked Context breakpoint that this breakpoint links to.

`0b0110`, Unlinked CONTEXTIDR_EL1 Match breakpoint

BT == `0b0110` is a reserved value if either:

- The breakpoint is not a context-aware breakpoint.
- The implementation does not include ARMv8.1-VHE.

For context-aware breakpoints, generation of a Breakpoint exception depends on both:

- DBGBCR<n>.{SSC, HMC, PMC}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions.
- A successful Context ID match defined by this breakpoint, as described in *Breakpoint context comparisons* on page G2-5847.

The Context ID check is made against the value in CONTEXTIDR, or CONTEXTIDR_EL1. The value of DBGBVR<n>.ContextID is compared with the Context ID value held in CONTEXTIDR or CONTEXTIDR_EL1.

DBGBCR<n>.{LBN, BAS} for this breakpoint are ignored.

`0b0111`, Linked CONTEXTIDR_EL1 Match breakpoint

BT == `0b0111` is a reserved value if either:

- The breakpoint is not a context-aware breakpoint.
- The implementation does not include ARMv8.1-VHE.

For context-aware breakpoints, one of the following applies:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint, then the breakpoint does not generate any Breakpoint exceptions.
- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address match breakpoint that links to this breakpoint, see *Breakpoint instruction address comparisons* on page G2-5842.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

  - A successful Context ID match defined by this breakpoint, as described in *Breakpoint context comparisons* on page G2-5847.
- Generation of a Watchpoint exception depends on both:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see *Watchpoint data address comparisons* on page G2-5859.
  - A successful Context ID match defined by this breakpoint, as described in *Breakpoint context comparisons* on page G2-5847.

The Context ID check is made against the value in CONTEXTIDR, or CONTEXTIDR_EL1. The value of DBGBVR<n>.ContextID is compared with the Context ID value held in CONTEXTIDR or CONTEXTIDR_EL1.

DBGBCR<n>.{LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

`0b1000`, Unlinked VMID Match breakpoint

BT == `0b1000` is a reserved value if either:

- The breakpoint is not a context-aware breakpoint.
- EL2 is not implemented.

For context-aware breakpoints, generation of a Breakpoint exception depends on both:

- DBGBCR<n>.{SSC, HMC, PMC}. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page G2-5840.
- A successful VMID match, as described in *Breakpoint context comparisons* on page G2-5847.

DBGBCR<n>.{LBN, BAS} for this breakpoint are ignored.

`0b1001`, Linked VMID Match breakpoint

BT == `0b1000` is a reserved value if either:

- The breakpoint is not a context-matching breakpoint.
- EL2 is not implemented.

For context-aware breakpoints, either:

- This breakpoint does not generate any Breakpoint exceptions, if no Linked breakpoints or Linked watchpoints link to it.
- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address Match breakpoint that links to this breakpoint. See *Breakpoint instruction address comparisons* on page G2-5842.
  - A successful VMID match defined by this breakpoint.
- Generation of a Watchpoint exception depends on both:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see *Watchpoint data address comparisons* on page G2-5859.
  - A successful VMID match defined by this breakpoint, as described in *Breakpoint context comparisons* on page G2-5847.

DBGBCR<n>.{LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

`0b1010`, Unlinked Context ID and VMID Match breakpoint

BT == `0b1010` is a reserved value if either:

- The breakpoint is not a context-matching breakpoint.
- EL2 is not implemented.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For context-matching breakpoints, generation of a Breakpoint exception depends on all of the following:

- DBGBCR<n>.{SSC, HMC, PMC}. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page G2-5840.
- A successful Context ID match, as described in *Breakpoint context comparisons* on page G2-5847.
- A successful VMID match.

The value of DBGBVR<n>.ContextID is compared with CONTEXTIDR.

Breakpoint context comparisons on page G2-5847 describes the requirements for a successful Context ID match and a successful VMID match.

DBGBCR<n>.{LBN, BAS} for this breakpoint are ignored.

`0b1011`, Linked Context ID and VMID Match breakpoint

BT == `0b1011` is a reserved value if either:

- The breakpoint is not a context-matching breakpoint.
- EL2 is not implemented.

For context-matching breakpoints, either:

- This breakpoint does not generate any Breakpoint exceptions, if no Linked breakpoints or Linked watchpoints link to it.
- Generation of a Breakpoint exception depends on all of the following:
  - A successful instruction address match, defined by a Linked Address breakpoint that links to this breakpoint, see *Breakpoint instruction address comparisons* on page G2-5842.
  - A successful Context ID match defined by this breakpoint, as described in *Breakpoint context comparisons* on page G2-5847.
  - A successful VMID match defined by this breakpoint.
- Generation of a Watchpoint exception depends on all of the following:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see *Watchpoint data address comparisons* on page G2-5859.
  - A successful Context ID match defined by this breakpoint, as described in *Breakpoint context comparisons* on page G2-5847.
  - A successful VMID match defined by this breakpoint.

The value of DBGBVR<n>.ContextID is compared with CONTEXTIDR.

Breakpoint context comparisons on page G2-5847 describes the requirements for a successful Context ID match and a successful VMID match by this breakpoint.

DBGBCR<n>.{LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

`0b1100`, Unlinked CONTEXTIDR_EL2 Match breakpoint

BT == `0b1100` is a reserved value if:

- The breakpoint is not a context-aware breakpoint.
- ARMv8.1-VHE is not implemented and ARMv8.2-Debug is not implemented, which means the implementation does not include CONTEXTIDR_EL2.
- EL2 is not implemented.

For context-aware breakpoints, generation of a Breakpoint exception depends on both:

- DBGBCR<n>.{SSC, HMC, PMC}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions.
- A successful CONTEXTIDR_EL2 match.The value of DBGBVR<n>.ContextID2 is compared with the Context ID value held in CONTEXTIDR_EL2, as described in *Breakpoint context comparisons* on page G2-5847.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The check against CONTEXTIDR_EL2 means this breakpoint can be generated only if EL2 is implemented and enabled in the current Security state and EL2 is using AArch64.

Note

The operation of this breakpoint does not depend on the value of HCR_EL2.E2H.

DBGBCR<n>.{LBN, BAS} for this breakpoint are ignored.

`0b1101`, Linked CONTEXTIDR_EL2 Match

BT == `0b1101` is a reserved value if:

- The breakpoint is not a context-aware breakpoint.
- ARMv8.1-VHE is not implemented and ARMv8.2-Debug is not implemented, which means the implementation does not include CONTEXTIDR_EL2.
- EL2 is not implemented.

For context-aware breakpoints, either:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint, then the breakpoint does not generate any Breakpoint exceptions.
- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address match breakpoint that links to this breakpoint, see *Breakpoint instruction address comparisons* on page G2-5842.
  - A successful CONTEXTIDR_EL2 match, as described in *Breakpoint context comparisons* on page G2-5847.
- Generation of a Watchpoint exception depends on both:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see *Watchpoint data address comparisons* on page G2-5859.
  - A successful CONTEXTIDR_EL2 match. The value of DBGBVR<n>.ContextID2 is compared with the Context ID value held in CONTEXTIDR_EL2, as described in *Breakpoint context comparisons* on page G2-5847.

The check against the CONTEXTIDR_EL2 means the breakpoint or watchpoint can be generated only if EL2 is implemented and enabled in the current Security state and EL2 is using AArch64.

Note

The operation of this breakpoint does not depend on the value of HCR_EL2.E2H.

DBGBCR<n>.{LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

`0b1110`, Unlinked Full Context ID Match breakpoint

BT == `0b1110` is a reserved value if:

- The breakpoint is not a context-aware breakpoint.
- ARMv8.1-VHE is not implemented and ARMv8.2-Debug is not implemented, which means the implementation does not include CONTEXTIDR_EL2.
- EL2 is not implemented.

For context-aware breakpoints, generation of a Breakpoint exception depends on both:

- DBGBCR<n>.{SSC, HMC, PMC}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions.
- A successful Context ID match, as described in *Breakpoint context comparisons* on page G2-5847.

The Context ID check is made by checking both:

- The value of DBGBVR<n>.ContextID against the value in CONTEXTIDR, or CONTEXTIDR_EL1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The value of DBGBXVR<n>.ContextID2 against the value in CONTEXTIDR_EL2.

Both comparisons must match for the check to succeed.

The check against the CONTEXTIDR_EL2 means this breakpoint can be generated only if EL2 is implemented and enabled in the current Security state and EL2 is using AArch64.

Note

The operation of this breakpoint does not depend on the value of HCR_EL2.E2H.

DBGBCR<n>.{LBN, BAS} for this breakpoint are ignored.

`0b1111`, Linked Full Context ID Match breakpoint

BT == `0b1111` is a reserved value if:

- The breakpoint is not a context-aware breakpoint.
- ARMv8.1-VHE is not implemented and ARMv8.2-Debug is not implemented, which means the implementation does not include CONTEXTIDR_EL2.
- EL2 is not implemented.

For context-aware breakpoints, one of the following applies:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint, then the breakpoint does not generate any Breakpoint exceptions.
- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address match breakpoint that links to this breakpoint, see *Breakpoint instruction address comparisons* on page G2-5842.
  - A successful Context ID match, as described in *Breakpoint context comparisons* on page G2-5847.
- Generation of a Watchpoint exception depends on both:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see *Watchpoint data address comparisons* on page G2-5859.
  - A successful Context ID match, as described in *Breakpoint context comparisons* on page G2-5847.

The Context ID check is made by checking both:

- The value of DBGBVR<n>.ContextID against the value in CONTEXTIDR, or CONTEXTIDR_EL1.
- The value of DBGBXVR<n>.ContextID2 against the value in CONTEXTIDR_EL2.

Both comparisons must match for the check to succeed.

The check against the CONTEXTIDR_EL2 means the breakpoint or watchpoint can be generated only if EL2 is implemented and enabled in the current Security state and EL2 is using AArch64.

Note

The operation of this breakpoint does not depend on the value of HCR_EL2.E2H.

DBGBCR<n>.{LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

Note

See *Reserved DBGBCR<n>.BT values* on page G2-5850 for the behavior of breakpoints programmed with reserved BT values.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G2.9.3 Execution conditions for which a breakpoint generates Breakpoint exceptions

Each breakpoint can be programmed so that it only generates Breakpoint exceptions for certain execution conditions. For example, a breakpoint might be programmed to generate Breakpoint exceptions only when the PE is executing at PL0 in Secure state.

DBGBCR<n>.{SSC, HMC, PMC} define the execution conditions the breakpoint generates Breakpoint exceptions for, as follows:

Security State Control, SSC

Controls whether the breakpoint generates Breakpoint exceptions only in Secure state, only in Non-secure state, or in both Security states.

Note

This is determined by the Security state of the PE, not from the NS attribute returned by the translation of the virtual address on which the breakpoint is set.

Higher Mode Control, HMC, and Privileged Mode Control, PMC

HMC and PMC together control which AArch32 modes the breakpoint generates Breakpoint exceptions in.

Table G2-10 shows the valid combinations of the values of HMC, SSC, and PMC, and for each combination shows which Privilege levels breakpoints generate Breakpoint exceptions in.

In the table:

Y Means that a breakpoint programmed with the values of HMC, SSC and PMC shown in that row can generate Breakpoint exceptions in AArch32 modes at that Privilege level.

- Means that a breakpoint programmed with the values of HMC, SSC and PMC shown in that row cannot generate Breakpoint exceptions in AArch32 modes at that Privilege level.

Res Means that the combination of HMC, SSC, and PMC is reserved. See *Reserved DBGBCR<n>.{SSC, HMC, PMC} values* on page G2-5851.

Table G2-10 Summary of breakpoint HMC, SSC, and PMC encodings

HMC	SSC	PMC	Security state the breakpoint is programmed to match in	PL2[a]	PL1	PL0
0	00	00	Both	-	Y[b]	Y
0	00	01		-	Y	-
0	00	10		-	-	Y
0	00	11		-	Y	Y
0	01	00	Non-Secure	-	Y[b]	Y
0	01	01		-	Y	-
0	01	10		-	-	Y
0	01	11		-	Y	Y

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G2-10 Summary of breakpoint HMC, SSC, and PMC encodings (continued)

HMC	SSC	PMC	Security state the breakpoint is programmed to match in	PL2[a]	PL1	PL0
0	10	00	Secure	-	Y[b]	Y
0	10	01		-	Y	-
0	10	10		-	-	Y
0	10	11		-	Y	Y
0	11	01	Secure	Y	Y	-
0	11	11		Y	Y	Y
1	00	01	Both	Y	Y	-
1	00	11		Y	Y	Y
1	01	00	Non-secure	Y		-
1	01	01		Y	Y	-
1	01	11		Y	Y	Y
1	10	01	Secure	Y	Y	-
1	10	11		Y	Y	Y
1	11	00	Both	Y	-	-
1	11	01		Y	Y	-
1	11	11		Y	Y	Y

a. Debug exceptions are not generated at PL2 using AArch32. This means that these combinations of HMC, SSC, and PMC are only relevant if breakpoints cause entry to Debug state. Self-hosted debuggers must avoid combinations of HMC, SSC, and PMC that generate Breakpoint exceptions at PL2 using AArch32.

b. Only in User, System and Supervisor modes.

All combinations of HMC, SSC, and PMC that this table does not show are reserved. See *Reserved HMC, SSC, and PMC combinations* on page G2-5851.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G2.9.4 Breakpoint instruction address comparisons

Address comparisons are made for each instruction in the program flow. The following subsections describe the criteria for a successful address comparison, for:

- *Address Match breakpoints*.
- *Address Mismatch breakpoints*.

Address Match breakpoints

An address match comparison is successful if both:

- Bits [31:2] of the current instruction virtual address are equal to DBGBVR<n>[31:2].
- The word or halfword selected by DBGBCR<n>.BAS matches. That is, either:
  - — DBGBCR<n>.BAS is programmed with `0b0011` or `0b1111`, and the instruction is at a word-aligned address.
  - — DBGBCR<n>.BAS is programmed with `0b1100`, and the instruction is not at a word-aligned address.

  See *Specifying the halfword-aligned address that an Address breakpoint matches on*.

Note

DBGBVR<n>[1:0] are RES0 and are ignored.

Address Mismatch breakpoints

An address mismatch comparison is successful if either:

- Bits [31:2] of the current instruction virtual address are not equal to DBGBVR<n>[31:2].
- The word or halfword selected by DBGBCR<n>.BAS does not match. That is, either:
  - — DBGBCR<n>.BAS is programmed with `0b0011` or `0b1111`, and the instruction is not at a word-aligned address.
  - — DBGBCR<n>.BAS is programmed with `0b1100`, and the instruction is at a word-aligned address.

  See *Specifying the halfword-aligned address that an Address breakpoint matches on*.

Note

- DBGBVR<n>[1:0] are RES0 and are ignored.
- Address Mismatch breakpoints can be used to single-step through code. See *Using an Address Mismatch breakpoint to single-step an instruction* on page G2-5848.

Specifying the halfword-aligned address that an Address breakpoint matches on

For an Address breakpoint, a debugger can use the *Byte Address Selection* field, DBGBCR<n>.BAS, so that the address comparison is successful on one of:

- The whole word starting at address DBGBVR<n>[31:2]:00.
- The halfword starting at address DBGBVR<n>[31:2]:00.
- The halfword starting at address ((DBGBVR<n>[31:2]:00) + 2).

Note

The address programmed into the DBGBVR<n> must be word-aligned.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

DBGBCR<n>.BAS can be used in both Address Match breakpoints and Address Mismatch breakpoints, as follows:

- For an Address Match breakpoint, DBGBCR<n>.BAS selects which halfword-aligned address the breakpoint must generate a Breakpoint exception for. This means that an address comparison is successful only if both of the following match:
  - The instruction address held in bits [31:2] of the DBGBVR<n>.
  - The halfword defined by the BAS field.

  That is, a successful address comparison = DBGBVR<n>[31:2] match AND BAS match.

- For an Address Mismatch breakpoint, DBGBCR<n>.BAS selects which halfword-aligned address the breakpoint must not generate a Breakpoint exception for. This means that an address comparison is successful if either or both of the following do not match:
  - The instruction address held in bits [31:2] of the DBGBVR<n>.
  - The halfword defined by the BAS field.

  That is, a successful address comparison = NOT (DBGBVR<n>[31:2] match AND BAS match).

The following subsections show the supported BAS values:

- *Using the BAS field in Address Match breakpoints*.
- *Using the BAS field in Address Mismatch breakpoints* on page G2-5845.

For Context breakpoints, DBGBCR<n>.BAS is RES1 and is ignored.

Using the BAS field in Address Match breakpoints

The supported BAS values are:

`0b0000` This value is reserved. Behavior is a CONSTRAINED UNPREDICTABLE choice of:
- The breakpoint is disabled.
- The breakpoint behaves as if BAS is `0b0011`, `0b1100`, or `0b1111`.

`0b0011` The breakpoint generates a Breakpoint exception if an instruction with an address described as follows is committed for execution:
- Bits [31:2] of the address equals DBGBVR<n>[31:2].
- Bits [1:0] of the address are `0b00`.

This means that breakpoints programmed with this BAS value generate Breakpoint exceptions for all of the following:
- 32-bit T32 instructions at word-aligned addresses.
- 16-bit T32 instructions at word-aligned addresses.
- A32 instructions. These are always at word-aligned addresses.

However, Arm recommends that a debugger uses this BAS value only for T32 instructions.

It is CONSTRAINED UNPREDICTABLE whether a breakpoint programmed with this BAS value generates a Breakpoint exception on the second halfword of a 32-bit T32 instruction starting at the halfword-aligned address ((DBGBVR<n>[31:2]:00) - 2).

`0b1100` The breakpoint generates a Breakpoint exception if an instruction with an address described as follows is committed for execution:
- Bits [31:2] of the address equals DBGBVR<n>[31:2].
- Bits [1:0] of the address are `0b10`.

This means that breakpoints programmed with this BAS value generate Breakpoint exceptions for both of the following:
- 32-bit T32 instructions at addresses that are halfword-aligned but not word-aligned.
- 16-bit T32 instructions at addresses that are halfword-aligned but not word-aligned.

It is CONSTRAINED UNPREDICTABLE whether a breakpoint programmed with this BAS value generates a Breakpoint exception on the second halfword of a 32-bit T32 or A32 instruction starting at a word-aligned address.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1111` The breakpoint generates a Breakpoint exception if an instruction with an address described as follows is committed for execution:

- Bits [31:2] of the address equals DBGBVR<n>[31:2].
- Bits [1:0] of the address are `0b00`.

This means that breakpoints programmed with this BAS value generate Breakpoint exceptions for all of the following:

- 32-bit T32 instructions at word-aligned addresses.
- 16-bit T32 instructions at word-aligned addresses.
- A32 instructions. These are always at word-aligned addresses.

However, Arm recommends that a debugger uses this BAS value only for A32 instructions.

It is CONSTRAINED UNPREDICTABLE whether a breakpoint programmed with this BAS value generates a Breakpoint exception on the second halfword of a 32-bit T32 instruction starting at the halfword-aligned address ((DBGBVR<n>[31:2]:00) - 2).

It is CONSTRAINED UNPREDICTABLE whether a breakpoint programmed with this BAS value generates a Breakpoint exception on a 32-bit T32 instruction or a 16-bit T32 instruction at the halfword-aligned address ((DBGBVR<n>[31:2]:00) + 2).

All other BAS values are reserved. For these reserved other values, DBGBCR<n>.BAS[3,1] ignore writes and read the same values as DBGBCR<n>[2,0] respectively. This means that the smallest instruction size a debugger can program breakpoints to match on is a halfword.

Figure G2-2 on page G2-5845 shows a summary of when breakpoints programmed with particular BAS values generate Breakpoint exceptions.

The figure contains four parts:

- A column showing the row number, on the left.
- An instruction set and instruction size table.
- A location of instruction figure.
- A BAS field values table, on the right.

To use the figure, read across the rows. For example:

- Row 2 shows that a breakpoint with a BAS value of `0b1100` generates Breakpoint exceptions for 16-bit T32 instructions starting at the halfword-aligned address ((DBGBVR<n>[31:2]:00) + 2).
- Row 6 shows that a breakpoint with a BAS value of either `0b0011` or `0b1111` generates Breakpoint exceptions for A32 instructions. A32 instructions are always at word-aligned addresses.

In the figure:

Yes Means that the breakpoint generates a Breakpoint exception.

No Means that the breakpoint does not generate a Breakpoint exception.

UNP Means that it is CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception. See *Other usage constraints for Address breakpoints* on page G2-5852.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

	Instruction set	Size	Location of instruction[a]								BAS[3:0]		
			-2	-1	0	+1	+2	+3	+4	+5	0b0011	0b1100	0b1111
Row 1	T32	16-bit			■	■					Yes	No	Yes
Row 2		16-bit					■	■			No	Yes	UNP
Row 3	T32	32-bit	■	■	■	■					UNP	No	UNP
Row 4		32-bit			■	■	■	■			Yes	UNP	Yes
Row 5		32-bit					■	■	■	■	No	Yes	UNP
Row 6	A32	32-bit			■	■	■	■			Yes	UNP	Yes

a. 0 means the word-aligned address held in the DBGBVRn. The other locations are as follows:
- -2 means ((DBGBVRn[31:2]:00) - 2).
- -1 means ((DBGBVRn[31:2]:00) - 1).
- ...
- ...
- +5 means ((DBGBVRn[31:2]:00) + 5).

The solid areas show the location of the instruction.

Figure G2-2 Summary of BAS field meanings for Address Match breakpoints

Using the BAS field in Address Mismatch breakpoints

An Address Mismatch breakpoint generates Breakpoint exceptions for all instructions committed for execution, except the instruction whose address the breakpoint is programmed to match.

The supported BAS values are:

`0b0000` The breakpoint ignores the address held in the DBGBVR<n> and generates Breakpoint exceptions for all instruction addresses.

`0b0011` The breakpoint does not generate a Breakpoint exception if an instruction with an address described as follows is committed for execution:
- Bits [31:2] of the address equals DBGBVR<n>[31:2].
- Bits [1:0] of the address are `0b00`.

This means that breakpoints programmed with this BAS value do not generate Breakpoint exceptions for any of the following:
- 32-bit T32 instructions at word-aligned addresses.
- 16-bit T32 instructions at word-aligned addresses.
- A32 instructions. These are always at word-aligned addresses.

However, Arm recommends that a debugger uses this BAS value only for T32 instructions.

It is CONSTRAINED UNPREDICTABLE whether a breakpoint programmed with this BAS value does not generate a Breakpoint exception on the second halfword of a 32-bit T32 instruction starting at the halfword-aligned address ((DBGBVR<n>[31:2]:00) - 2).

`0b1100` The breakpoint does not generate a Breakpoint exception if an instruction with an address described as follows is committed for execution:
- Bits [31:2] equals DBGBVR<n>[31:2].
- Bits [1:0] of the address are `0b10`.

This means that breakpoints programmed with this BAS value do not generate Breakpoint exceptions for either of the following:
- 32-bit T32 instructions at addresses that are halfword-aligned but not word-aligned.
- 16-bit T32 instructions at addresses that are halfword-aligned but not word-aligned.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

It is CONSTRAINED UNPREDICTABLE whether a breakpoint programmed with this BAS value does not generate a Breakpoint exception on the second halfword of a 32-bit T32 or A32 instruction at a word-aligned address.

`0b1111` The breakpoint does not generate a Breakpoint exception if an instruction with an address described as follows is committed for execution:

- Bits [31:2] of the address equals DBGBVR<n>[31:2].
- Bits [1:0] of the address are `0b00`.

This means that breakpoints programmed with this BAS value do not generate Breakpoint exceptions for any of the following:

- 32-bit T32 instructions at word-aligned addresses.
- 16-bit T32 instructions at word-aligned addresses.
- A32 instructions. These are always at word-aligned addresses.

However, Arm recommends that a debugger uses this BAS value only for A32 instructions.

It is CONSTRAINED UNPREDICTABLE whether a breakpoint programmed with this BAS value does not generate a Breakpoint exception on the second halfword of a 32-bit T32 instruction starting at the halfword-aligned address ((DBGBVR<n>[31:2]:00) - 2).

It is CONSTRAINED UNPREDICTABLE whether a breakpoint programmed with this BAS value does not generate a Breakpoint exception on a 32-bit T32 instruction or a 16-bit T32 instruction at the halfword-aligned address ((DBGBVR<n>[31:2]:00) + 2).

All other BAS values are reserved. For these reserved other values, DBGBCR<n>.BAS[3,1] ignore writes and read the same values as DBGBCR<n>[2,0] respectively. This means that the smallest instruction size that a breakpoint can never generate a Breakpoint exception for is a halfword.

Figure G2-3 on page G2-5847 shows a summary of when breakpoints programmed with particular BAS values generate Breakpoint exceptions.

The figure contains four parts:

- A column showing the row number, on the left.
- An instruction set and instruction size table.
- A location of instruction figure.
- A BAS field values table, on the right.

To use the figure, read across the rows. For example:

- Row 1 shows that a breakpoint with a BAS value of `0b1100` generates Breakpoint exceptions for 16-bit T32 instructions starting at the word-aligned address held in the DBGBVR<n>.
- Row 5 shows that a breakpoint with a BAS value of `0b0011` generates Breakpoint exceptions for 32-bit T32 instructions starting at the halfword-aligned address immediately after the word aligned address held in the DBGBVR<n>.

In the figure:

Yes Means that the breakpoint does generate a Breakpoint exception.

No Means that the breakpoint does not generate a Breakpoint exception.

UNP Means that is it CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception. See *Other usage constraints for Address breakpoints* on page G2-5852.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

	Instruction set	Size	Location of instruction[a]								BAS[3:0]			
			-2	-1	0	+1	+2	+3	+4	+5	0b0000	0b0011	0b1100	0b1111
Row 1	T32	16-bit			■	■					Yes	No	Yes	No
Row 2		16-bit					■	■			Yes	Yes	No	UNP
Row 3	T32	32-bit	■	■	■	■					Yes	UNP	Yes	UNP
Row 4		32-bit			■	■	■	■			Yes	No	UNP	No
Row 5		32-bit					■	■	■	■	Yes	Yes	No	UNP
Row 6	A32	32-bit			■	■	■	■			Yes	No	UNP	No

a. 0 means the word-aligned address held in the DBGBVRn. The other locations are as follows:
- -2 means ((DBGBVRn[31:2]:00) - 2).
- -1 means ((DBGBVRn[31:2]:00) - 1).
- ...
- ...
- +5 means ((DBGBVRn[31:2]:00) + 5).

The solid areas show the location of the instruction.

Figure G2-3 Summary of BAS field meanings for Address Mismatch breakpoints

G2.9.5 Breakpoint context comparisons

The breakpoint type defined by DBGBCR<n>.BT determines what context comparison is required, if any. Table G2-11 shows the BT values that require a comparison, and the match required for the comparison to be successful.

Table G2-11 Breakpoint Context ID and VMID comparison tests

DBGBCR<n>.BT	Test required for successful context comparison
0b001x	• When ARMv8.1-VHE is implemented, EL2 is using AArch64, the *Effective value* of HCR_EL2.E2H is 1, and either the PE is executing at EL0 with HCR_EL2.TGE set to 1, or the PE is executing at EL2, CONTEXTIDR_EL2 must match the DBGBVR<n>. ContextID value. • Otherwise, CONTEXTIDR must match the DBGBVR<n>.ContextID value.
0b011x	CONTEXTIDR, or CONTEXTIDR_EL1, must match the DBGBVR<n>.ContextID value.
0b100x	VTTBR.VMID must match the DBGBXVR<n>.VMID value.
0b101x	CONTEXTIDR, or CONTEXTIDR_EL1, must match the DBGBVR<n>.ContextID value, and VTTBR.VMID must match the DBGBXVR<n>.VMID value.
0b110x	CONTEXTIDR_EL2 must match the DBGBXVR<n>.ContextID2 value.
0b111x	Both: • CONTEXTIDR, or CONTEXTIDR_EL1, must match the DBGBVR<n>.ContextID value. • CONTEXTIDR_EL2 must match the DBGBXVR<n>.ContextID2 value.

No context comparison is required for other valid DBGBCR<n>.BT values.

Context breakpoints do not generate Breakpoint exceptions when any of:

- The comparison uses the value of CONTEXTIDR, or CONTEXTIDR_EL1, and any of:
  - The PE is executing at EL3 using AArch64.
  - The PE is executing at EL2.
  - ARMv8.1-VHE is implemented, EL2 is using AArch64, EL2 is implemented and enabled in the current Security state, and HCR_EL2.{E2H, TGE} == {1, 1}.
- The comparison uses the value of CONTEXTIDR_EL2 and any of:
  - Neither ARMv8.1-VHE is implemented, nor ARMv8.2-Debug is implemented.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

  - EL2 is either not implemented or not enabled in the current Security state.
  - EL2 is using AArch32.

- The comparison uses the current VMID value and any of:
  - EL2 is not implemented.
  - EL2 is either not implemented or not enabled in the current Security state.
  - The PE is executing at EL2.
  - ARMv8.1-VHE is implemented, EL2 is using AArch64, EL2 is implemented and enabled in the current Security state, and HCR_EL2.{E2H, TGE} == {1, 1}.

Note

- For all Context breakpoints, DBGBCR<n>.BAS is RES1 and is ignored.
- For Linked Context breakpoints, DBGBCR<n>.{LBN, SSC, HMC, PMC} are RES0 and are ignored.

G2.9.6 Using breakpoints

This section contains the following:

- *Using an Address Mismatch breakpoint to single-step an instruction*.
- *ITD control effects on address breakpoints on the first instruction in an IT block* on page G2-5849.
- *Breakpoint usage constraints* on page G2-5850.

Using an Address Mismatch breakpoint to single-step an instruction

In execution conditions that an Address Mismatch breakpoint matches, defined by DBGBCR<n>.{LBN, SSC, PMC}, the breakpoint generates Breakpoint exceptions for all instructions committed for execution, except the instruction whose address the breakpoint is programmed with. Figure G2-4 shows an example of Address Mismatch breakpoint operation, for an Address Mismatch breakpoint programmed with address `0x1014`.

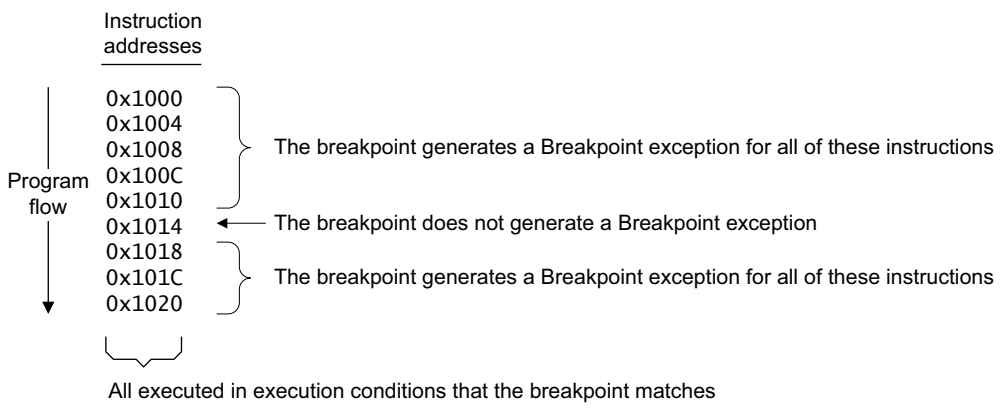


Figure G2-4 Operation of an Address Mismatch breakpoint

This means that an Address Mismatch breakpoint can be used to single-step an instruction.

In the example shown in Figure G2-4:

- If the target of a branch is an instruction other than the instruction at address `0x1014`, the breakpoint generates a Breakpoint exception when the instruction is committed for execution.
- If the target of a branch is the instruction at address `0x1014`, the PE executes the instruction at `0x1014` and the breakpoint does not generate a Breakpoint exception until the instruction at address `0x1018` is committed for execution. The instruction at address `0x1014` is therefore single-stepped.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

However, if the instruction at `0x1014` generates a synchronous exception, or if the PE takes an asynchronous exception while the instruction is being stepped, the breakpoint is evaluated again after taking the exception. This means that behavior is as follows:

— If the exception handler executes in execution conditions that the breakpoint matches, the breakpoint generates a Breakpoint exception for the exception vector, because the exception vector is not address `0x1014`. This means that software execution steps into the exception.

— If the exception handler executes in execution conditions that the breakpoint does not match, the breakpoint does not generate any Breakpoint exceptions after the PE has taken the exception, until the exception handler completes and executes an exception return instruction. The effect is to step over the exception. Whether the instruction is stepped again depends on whether the target of the exception return instruction is the instruction at `0x1014` or the instruction at `0x1018`.

If the instruction at `0x1014` is single-stepped and branches to itself, it is CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception after the PE has executed the branch.

This means that an instruction is only single-stepped if it is the target of a branch instruction and its address matches the address the breakpoint is programmed for. In the example shown in Figure G2-4 on page G2-5848, this is `0x1014`.

Usually this branch instruction is an exception return instruction that changes PE mode, branching from a PE mode in which the breakpoint does not generate a Breakpoint exception. A branch instruction that does not change PE mode would itself generate a Breakpoint exception. However, it might be a branch-to-self instruction as described above.

Because Address Mismatch breakpoints can single-step instructions, the behavior of an address mismatch Breakpoint exception is similar to the behavior of an AArch64 Software Step exception.

Note

- The example shown in Figure G2-4 on page G2-5848 assumes an A32 instruction. The same behavior applies for both 32-bit and 16-bit T32 instructions.
- Software Step exceptions are the highest priority synchronous exception. Breakpoint exceptions are lower priority. See *Synchronous exception prioritization for exceptions taken to AArch64 state* on page D1-2355.

ITD control effects on address breakpoints on the first instruction in an IT block

In an implementation that supports the ITD control, if the value of the ITD field that applies to the current Exception level is 1, all of the following are true:

- An `IT` instruction can only be used to apply to one 16-bit T32 instruction.
- Only certain combinations of an `IT` instruction and second single 16-bit T32 instruction are permitted.
- For a permitted combination, it is IMPLEMENTATION DEFINED whether the implementation treats the combination as:
  — A pair of 16-bit instructions.
  — One 32-bit instruction.

If the implementation treats the combination as one 32-bit instruction, then as described in *Other usage constraints for Address breakpoints* on page G2-5852, an Address breakpoint might not generate a Breakpoint exception for an address match only on the second halfword of the instruction.

For this reason, if the ITD bit associated with the current Exception level is 1, Arm recommends that a debugger that wants to program a breakpoint to match on the second T32 instruction programs it to match on the `IT` instruction instead.

However, if returning from an exception whose preferred return address is the address of the second T32 instruction, then because the debugger is aware that the implementation has treated the combination as a pair of 16-bit instructions, the debugger is permitted to program the breakpoint to match on the second T32 instruction.

The ITD control fields are:

HSCTLR.ITD Applies to execution at EL2 when EL2 is using AArch32.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

SCTLR.ITD Applies to execution at EL0 or EL1 when EL1 is using AArch32.

SCTLR_EL1.ITD
Applies to execution at EL0 using AArch32 when EL1 is using AArch64.

An implementation that does not support the ITD control behaves as if the value of the ITD field is 0, and therefore the information in this section does not apply to such an implementation.

Note

Programming the breakpoint to match on the second T32 instruction might be necessary when using an Address Mismatch breakpoint for single stepping.

Breakpoint usage constraints

See the following sections:

- *Reserved DBGBCR<n>.BT values*.
- *Reserved DBGBCR<n>.{SSC, HMC, PMC} values* on page G2-5851.
- *Reserved DBGBCR<n>.BAS values* on page G2-5851.
- *Reserved DBGBCR<n>.LBN values* on page G2-5852.
- *Other usage constraints for Address breakpoints* on page G2-5852.
- *Other usage constraints for Context breakpoints* on page G2-5852.

Reserved DBGBCR<n>.BT values

Table G2-12 shows when particular DBGBCR<n>.BT values are reserved.

Table G2-12 Reserved BT values

BT value	Breakpoint type	Reserved
`0b001x`	Context ID Match	If the breakpoint is not context-aware
`0b010x`	Address Mismatch	If EDSCR.HDE is 1 and halting is allowed
`0b011x`	CONTEXTIDR_EL1 Match	If ARMv8.1-VHE is not implemented, or the breakpoint is not context-aware
`0b100x`	VMID Match	If EL2 is not implemented, or the breakpoint is not context-aware
`0b101x`	Context ID and VMID Match	
`0b110x`	CONTEXTIDR_EL2 Match	If ARMv8.1-VHE is not implemented and ARMv8.2-Debug is not implemented, or if the breakpoint is not context-aware **Note** For these BT values, breakpoints are not generated if EL2 is using AArch32.
`0b111x`	Full Context ID Match	

If a breakpoint is programmed with one of these reserved BT values:

- The breakpoint must behave as if it is either:
  - Disabled.
  - Programmed with a BT value that is not reserved, other than for a direct or external read of DBGBCR<n>.
- For a direct or external read of DBGBCR<n>, if the reserved BT value:
  - Has no function for any execution conditions, the value read back is UNKNOWN.
  - Has a function for execution conditions other than the current execution conditions, the value read back is the value written. This permits software to save and restore the BT value so that the breakpoint functions for the other execution conditions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The behavior of breakpoints with reserved BT values might change in future revisions of the architecture. For this reason, software must not rely on the behavior described here.

Reserved DBGBCR<n>.{SSC, HMC, PMC} values

Table G2-13 shows when particular combinations of DBGBCR<n>.{SSC, HMC, PMC} are reserved in stage 1 of an AArch32 translation regime.

Table G2-13 Reserved HMC, SSC, and PMC combinations

HMC, SSC, and PMC combination	Reserved
All combinations with SSC set to `0b01` or `0b10`, except for the combination with HMC set to 1, SSC set to `0b01` and PMC set to `0b00`	When EL3 is not implemented and EL2 is implemented.
Any combination where HMC or SSC is nonzero	When both of EL2 and EL3 are not implemented
The combination with HMC set to 1, SSC set to `0b11`, and PMC set to `0b00`	When EL2 is not implemented
The combinations with SSC set to `0b11` and PMC set to `0b01` or `0b11`	When Secure EL2 is not implemented
The combination with HMC set to 1, SSC set to `0b01` and PMC set to `0b00`	When Secure EL2 is not implemented
Combinations not included in Table G2-10 on page G2-5840	Always

For all breakpoints except Linked Context breakpoints, if a breakpoint is programmed with one of these reserved combinations:

- If the reserved combination has a function for other execution conditions:
  - The breakpoint must behave as if it is disabled.
  - A direct or external read of DBGBCR<n>.{SSC, HMC, PMC} returns the values written. This means that software can save and restore the combination so that the breakpoint can function for the other execution conditions.
- If the reserved combination does not have a function for other execution conditions:
  - It must behave either as if it is programmed with a combination that is not reserved or as if it is disabled.
  - A direct or external read of DBGBCR<n>.{SSC, HMC, PMC} returns UNKNOWN values.

If the breakpoint is a Linked Context breakpoint, then:

- The values of HMC, SSC, and PMC are ignored.
- A direct or external read of DBGBCR<n>.{SSC, HMC, PMC} returns UNKNOWN values

The behavior of breakpoints with reserved combinations of HMC, SSC, and PMC might change in future revisions of the architecture. For this reason, software must not rely on the behavior described here.

Reserved DBGBCR<n>.BAS values

For all Context breakpoints

DBGBCR<n>.BAS is RES1 and is ignored.

For all Address breakpoints

The supported values of the BAS field for the Address Match and Address Mismatch breakpoints are shown in *Specifying the halfword-aligned address that an Address breakpoint matches on* on page G2-5842.

If a breakpoint is programmed with a reserved BAS value:

- The breakpoint must behave as if it is either:
  - Disabled.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— Programmed with a BAS value that is not reserved, other than for a direct or external read of DBGBCR<n>.

- A direct or external read of DBGBCR<n>.BAS returns an UNKNOWN value.

Software must not rely on these properties as the behavior of reserved values might change in a future revision of the architecture.

Reserved DBGBCR<n>.LBN values

For all Context breakpoints

DBGBCR<n>.LBN reads UNKNOWN and its value is ignored.

For Linked Address breakpoints

A Linked Address breakpoint must link to a context-aware breakpoint. For a Linked Address breakpoint, any DBGBCR<n>.LBN value that is not for a context-aware breakpoint is reserved.

If a Linked Address breakpoint links to a breakpoint that is not implemented, or that is not context-aware, then reads of DBGBCR<n>.LBN return an unknown value and the behavior is CONSTRAINED UNPREDICTABLE. The Linked Address breakpoint behaves as if it is either:

- Disabled.
- Linked to an UNKNOWN context-aware breakpoint.

If a Linked Address breakpoint that links to a breakpoint that is implemented and that is context-aware, but that is either not enabled or not programmed as a Linked Context breakpoint, it behaves as if it is disabled.

For Unlinked Address breakpoints

DBGBCR<n>.LBN reads UNKNOWN and its value is ignored.

Other usage constraints for Address breakpoints

For all Address breakpoints

- DBGBVR<n>[1:0] are RES0 and are ignored.
- The DBGBXVR<n> is ignored.

For Address Match breakpoints

- For 32-bit instructions, if a breakpoint matches on the address of the second halfword but not the address of the first halfword, it is CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception.
- If DBGBCR<n>.BAS is `0b1111`, it is CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception for a T32 instruction starting at address ((DBGBVR<n>[31:2]:00) + 2). For T32 instructions, Arm recommends that the debugger programs the BAS field with either `0b0011` or `0b1100`.

For Address Mismatch breakpoints

The constraints are the same as those described in *For Address Match breakpoints*, except that if two Address Mismatch breakpoints are programmed to match in the same Exception level and Security state, it is CONSTRAINED UNPREDICTABLE whether or not the instruction is stepped or a Breakpoint debug even is generated.

Other usage constraints for Context breakpoints

For all Context breakpoints

Any bits of DBGBVR<n> and DBGBXVR<n> that are not used to specify Context ID or VMID are RES0 and are ignored.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

This means that for Context ID Match breakpoints, the DBGBXVR<n> is RES0 and is ignored, and for VMID Match breakpoints, the DBGBVR<n> is RES0 and is ignored.

For Linked Context breakpoints

If no Linked Address breakpoints or Linked Watchpoints link to a Linked Context breakpoint, the Linked Context breakpoint does not generate any Breakpoint exceptions.

G2.9.7 Exception syndrome information and preferred return address for a Breakpoint exception

See the following:

- *Exception syndrome information for a Breakpoint exception*.
- *Preferred return address for a Breakpoint exception* on page G2-5854.

Note

Usually, the term *exception syndrome* is used only for exceptions taken to Hyp mode, or to AArch64 state. The referenced section uses the term more generally, to include exception information reported in the IFSR.

Exception syndrome information for a Breakpoint exception

The PE takes a Breakpoint exception as either:

- A Prefetch Abort exception if it is taken to PL1. In this case, it is taken to Abort mode.
- A Hyp trap exception, if it is taken to PL2 because HCR.TGE or HDCR.TDE is 1. In this case, it is taken to Hyp mode.

If the exception is taken to:

Abort mode

The PE sets all of the following:

- DBGDSCRext.MOE to `0b0001`, to indicate a Breakpoint exception.
- IFSR.FS to the code for a debug exception, `0b00010`.
- The IFAR with an UNKNOWN value.

Hyp mode

The PE does all of the following:

- Records information about the exception in the *Hypervisor Syndrome Register*, HSR. See Table G2-14.
- Sets DBGDSCRext.MOE to `0b0001`, to indicate a Breakpoint exception.
- Sets the HIFAR to an unknown value.

Table G2-14 Information recorded in the HSR

HSR field	Information recorded
Exception Class, EC	The PE sets this to the code for a Prefetch Abort exception routed to Hyp mode, `0x20`.
Instruction Length, IL	The PE sets this to 1.
Instruction Specific Syndrome, ISS	**ISS[24:10]** RES0. **ISS[9]** *External Abort type* (EA). The PE sets this to 0. **ISS[8:6]** RES0. **ISS[5:0]** *Instruction Fault Status Code* (IFSC). The PE sets this to the code for a debug exception, `0b100010`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

For information about how debug exceptions can be routed to PL2, see *Routing debug exceptions* on page G2-5819.

Preferred return address for a Breakpoint exception

The preferred return address of a Breakpoint exception is the address of the instruction that was not executed because the PE took the Breakpoint exception instead.

This means that the preferred return address is the address of the instruction that caused the exception.

G2.9.8 Pseudocode description of Breakpoint exceptions taken from AArch32 state

`AArch32.BreakpointValueMatch()` returns a pair of results:

- A result for Address Match and Context breakpoints.
- A result for Address Mismatch breakpoints.

`AArch32.StateMatch()` tests the values in DBGBCR<n>.{SSC, HMC, PMC} and, if the breakpoint links to a Linked Context breakpoint, also tests the Linked Context breakpoint.

`AArch32.BreakpointMatch()` tests a committed instruction against all breakpoints.

`AArch32.CheckBreakpoint()` generates a `FaultRecord`. A Breakpoint exception is taken if all of the following are true:

- DBGDSCRext.MDBGen is 1.
- Debug exceptions are enabled from the current Exception level and Security state. See *Enabling debug exceptions from the current Privilege level and Security state* on page G2-5821.
- All of the conditions required for Breakpoint exception generation are met. See *About Breakpoint exceptions* on page G2-5830.

Note

`AArch32.CheckBreakpoint()` might halt the PE and cause it to enter Debug state. External debug uses Debug state.

The `AArch32.Abort()` function processes the `FaultRecord` object returned by `AArch32.CheckBreakpoint()`, as described in *Abort exceptions* on page G4-5919. When a Breakpoint exception is taken to AArch32 state, the `AArch32.Abort()` function generates a Prefetch Abort exception.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G2.10 Watchpoint exceptions

This section describes Watchpoint exceptions in stage 1 of an AArch32 translation regime.

The PE is using an AArch32 translation regime when it is executing either:

- At EL1 or higher in an Exception level that is using AArch32.
- At EL0 using AArch32 when EL1 is using AArch32.

This section contains the following subsections:

- *About Watchpoint exceptions*.
- *Watchpoint types and linking of watchpoints* on page G2-5856.
- *Execution conditions for which a watchpoint generates Watchpoint exceptions* on page G2-5857.
- *Watchpoint data address comparisons* on page G2-5859.
- *Determining the memory location that caused a Watchpoint exception* on page G2-5862.
- *Watchpoint behavior on other instructions* on page G2-5863.
- *Usage constraints* on page G2-5864.
- *Exception syndrome information and preferred return address* on page G2-5866.
- *Pseudocode description of Watchpoint exceptions taken from AArch32 state* on page G2-5867.

G2.10.1 About Watchpoint exceptions

A *watchpoint* is an event that results from the execution of an instruction, based on a data address. Watchpoints are also known as *data breakpoints*.

A watchpoint operates as follows:

1. A debugger programs the watchpoint with a data address, or a data address range.
2. The watchpoint generates a *Watchpoint debug event* on an access to the address, or any address in the address range.

A watchpoint never generates a Watchpoint debug event on an instruction fetch.

An implementation can include between 2-16 watchpoints. In an implementation, DBGDIDR.WRPs shows how many are implemented.

To use an implemented watchpoint, a debugger programs the following registers for the watchpoint:

- The *Watchpoint Control Register*, DBGWCR<n>. This holds control information for the watchpoint, for example an enable control.
- The *Watchpoint Value Register*, DBGWVR<n>. This holds the data virtual address used for watchpoint matching.

The registers are numbered, so that:

- DBGWCR1 and DBGWVR1 are for watchpoint number one.
- DBGWCR2 and DBGWVR2 are for watchpoint number two.
- …
- …
- DBGWCRn and DBGWVRn are for watchpoint number n.

A watchpoint can:

- Be programmed to generate Watchpoint debug events on read accesses only, on write accesses only, or on both types of access.
- Link to a *Linked Context breakpoint*, so that a Watchpoint debug event is only generated if the PE is in a particular context when the address match occurs.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A single watchpoint can be programmed to match on one or more address bytes. A watchpoint generates a Watchpoint debug event on an access to any byte that it is watching. The number of bytes a watchpoint is watching is either:

- One to eight bytes, provided that these bytes are contiguous and that they are all in the same naturally-aligned doubleword. A debugger uses the *Byte Address Select* field, DBGWCR<n>.BAS, to select the bytes. See *Programming a watchpoint with eight bytes or fewer* on page G2-5860.
- Eight bytes to 2GB, provided that both of the following are true:
  - The number of bytes is a power-of-two.
  - The range starts at an address that is aligned to the range size.

  A debugger uses the *MASK* field, DBGWCR<n>.MASK, to program a watchpoint with eight bytes to 2GB. See *Programming a watchpoint with eight or more bytes* on page G2-5861.

A debugger must use either the BAS field or the MASK field. If it uses both, whether the watchpoint generates Watchpoint exceptions is CONSTRAINED UNPREDICTABLE. See *Programming dependencies of the BAS and MASK fields* on page G2-5865.

For each memory access, all of the watchpoints are tested. When a watchpoint is tested, it generates a Watchpoint debug event if all of the following are true:

- The watchpoint is enabled. That is, the watchpoint enable control for it, DBGWCR<n>.E, is 1.
- The conditions specified in the DBGWCR<n> are met.
- The comparison with the address held in the DBGWVR<n> is successful.
- If the watchpoint links to a Linked Context breakpoint, the comparison or comparisons made by the Linked Context breakpoint are successful. See on page G2-5833 shows this. See also *Breakpoint context comparisons* on page G2-5847.
- The instruction that initiates the memory access is committed for execution.
- The instruction that initiates the memory access passes its Condition code check.

If halting is allowed and EDSCR.HDE is 1, Watchpoint debug events cause entry to Debug state.

Otherwise, if debug exceptions are:
- Enabled, Watchpoint debug events generate Watchpoint exceptions.
- Disabled, Watchpoint debug events are ignored.

Note

The remainder of this Watchpoint Exceptions section, including all subsections, describes watchpoints as generating Watchpoint exceptions. However, the behavior described also applies if watchpoints are causing entry to Debug state.

The debug exception enable controls on page G2-5818 describes the enable controls for Watchpoint debug events.

G2.10.2 Watchpoint types and linking of watchpoints

When a debugger programs a watchpoint, it must program that watchpoint so that it is either:
- Used in isolation. In this case, the watchpoint is called an *Unlinked watchpoint*.
- Enabled for linking to a Linked Context breakpoint. In this case, the watchpoint is called a *Linked watchpoint*.

When a Linked watchpoint links to a Linked Context breakpoint, the Linked watchpoint only generates a Watchpoint exception if the PE is in a particular context when the data address match occurs. For example, a debugger might:

1. Program watchpoint number one with a data address.
2. Program breakpoint number five to be a *Linked VMID Match breakpoint*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

3. Link the watchpoint and the breakpoint together. A Watchpoint exception is only generated if both the data address matches and the VMID matches.

The *Watchpoint Type* field for a watchpoint, DBGWCR<n>.WT, controls whether the watchpoint is enabled for linking. If DBGWCR<n>.WT is 1, the watchpoint is enabled for linking.

Rules for linking watchpoints

The rules for watchpoint linking are as follows:

- Only Linked watchpoints can be linked.
- A Linked watchpoint can link to any type of Linked Context breakpoint. The *Linked Breakpoint Number* field, DBGWCR<n>.LBN, for the Linked watchpoint specifies the particular Linked Context breakpoint that the Linked watchpoint links to, and:
  - DBGWCR<n>.WT.{SSC, HMC, PAC} for the Linked watchpoint define the execution conditions that the watchpoint generates Watchpoint exceptions for. See *Execution conditions for which a watchpoint generates Watchpoint exceptions*.
  - DBGBCR<n>.{SSC, HMC, PMC} for the Linked Context breakpoint are ignored.
- A Linked watchpoint cannot link to another watchpoint. The LBN field can therefore only specify a breakpoint.
- If a Linked watchpoint links to a breakpoint that is not context-aware, the behavior of the Linked watchpoint is CONSTRAINED UNPREDICTABLE. See *Usage constraints* on page G2-5864.
- If a Linked watchpoint links to an Unlinked Context breakpoint, the Linked watchpoint never generates any Watchpoint exceptions.
- Multiple Linked watchpoints can link to a single Linked Context breakpoint.

  Note

  Multiple Address breakpoints can also link to a single Linked Context breakpoint. *Breakpoint exceptions* on page G2-5830 describes breakpoints.

Figure G2-1 on page G2-5833 shows an example of permitted watchpoint linking.

G2.10.3 Execution conditions for which a watchpoint generates Watchpoint exceptions

Each watchpoint can be programmed so that it only generates Watchpoint exceptions for certain execution conditions. For example, a watchpoint might be programmed to generate Watchpoint exceptions only when the PE is executing at EL2.

DBGWCR<n>.{SSC, HMC, PAC} define the execution conditions a watchpoint generates Watchpoint exceptions for, as follows:

Security State Control, SSC

Controls whether the watchpoint generates Watchpoint exceptions only in Secure state, only in Non-secure state, or in both Security states.

Note

This is determined by the Security state of the PE, not from the NS attribute returned by the translation of the virtual address on which the watchpoint is set.

Higher Mode Control, HMC, and Privileged Access Control, PAC

HMC and PAC together control which Privilege level the watchpoint generates Watchpoint exceptions in.

The PAC control relates to the privilege of the memory access, not to the Exception level or Privilege level at which the access was made.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

This means that, if the PE executes a Load unprivileged or Store unprivileged instruction at PL1, the resulting data access triggers a watchpoint only if both:

- PAC is programmed to a value that generates watchpoints on PL0 accesses.
- All other conditions for generating the watchpoint are met.

Example A32/T32 Load unprivileged and Store unprivileged instructions are `LDRT` and `STRT`.

Table G2-15 shows the valid combinations of HMC, SSC, and PAC, and for each combination shows which Privilege levels watchpoints generate Watchpoint exceptions in.

In the table:

Y or - Means that a watchpoint programmed with the values of HMC, SSC, and PAC shown in that row:

Y Can generate Watchpoint exceptions at that Privilege level.

- Cannot generate Watchpoint exceptions at that Privilege level.

Res Means that the combination of HMC, SSC, and PAC is reserved. See *Reserved DBGWCR<n>.{SSC, HMC, PAC} values* on page G2-5864.

Table G2-15 Summary of watchpoint HMC, SSC, and PAC encodings

HMC	SSC	PAC	Security state the watchpoint is programmed to match in	PL2[a]	PL1	PL0	Implementation	
							No EL3	No EL2 and no EL3
0	00	01	Both	-	Y	-	-	-
0	00	10		-	-	Y	-	-
0	00	11		-	Y	Y	-	-
0	01	01	Non-secure	-	Y	-	Res	Res
0	01	10		-	-	Y	Res	Res
0	01	11		-	Y	Y	Res	Res
0	10	01	Secure	-	Y	-	Res	Res
0	10	10		-	-	Y	Res	Res
0	10	11		-	Y	Y	Res	Res
0	11	01	Secure	Y	Y	-	-	Res
0	11	11		Y	Y	Y	-	Res
1	00	01	Both	Y	Y	-	-	Res
1	00	11		Y	Y	Y	-	Res
1	01	00	Non-secure	Y	-	-		
1	01	01		Y	Y	-	Res	Res
1	01	11		Y	Y	Y	Res	Res
1	10	01	Secure	-	Y	-	Res	Res
1	10	11		-	Y	Y	Res	Res

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G2-15 Summary of watchpoint HMC, SSC, and PAC encodings (continued)

HMC	SSC	PAC	Security state the watchpoint is programmed to match in	PL2[a]	PL1	PL0	Implementation	
							No EL3	No EL2 and no EL3
1	11	00	Both	Y	-	-	-	Res if no EL2[b]
1	11	01		Y	Y	-		
1	11	11		Y	Y	Y		

a. Debug exceptions are not generated at PL2 using AArch32. This means that these combinations of HMC, SSC, and PAC are only relevant if watchpoints cause entry to Debug state. Self-hosted debuggers must avoid combinations of HMC, SSC, and PAC that generate Watchpoint exceptions at PL2 using AArch32.

b. This encoding is only reserved when EL2 is not implemented, regardless of whether EL3 is implemented.

All combinations of HMC, SSC, and PAC that this table does not show are reserved. See *Reserved DBGWCR<n>.{SSC, HMC, PAC} values* on page G2-5864.

G2.10.4 Watchpoint data address comparisons

An address comparison is successful if bits [31:2] of the current data virtual address are equal to DBGWVR<n>[31:2], taking into account all of the following:

- The size of the access. See *Size of the data access*.
- The bytes selected by DBGWVR<n>.BAS. See *Programming a watchpoint with eight bytes or fewer* on page G2-5860.
- Any address ranges indicated by DBGWVR<n>.MASK. See *Programming a watchpoint with eight or more bytes* on page G2-5861.

Note

DBGWVR<n>[1:0] are RES0 and are ignored.

Size of the data access

Because watchpoints can be programmed to generate Watchpoint exceptions on individual bytes, the size of each access must be taken into account. See Example G2-1.

Example G2-1

1. A debugger programs a watchpoint to generate Watchpoint exceptions only when the byte at address `0x1009` is accessed.
2. The PE accesses the unaligned doubleword starting at address `0x1003`.

In this scenario, the watchpoint must generate a Watchpoint exception.

The size of data accesses initiated by `DCIMVAC` instructions is an IMPLEMENTATION DEFINED size that is both:

- From the inclusive range between:
  - The size that CTR.DminLine defines.
  - 2KB.
- A power-of-two.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The lowest address accessed by a DCIMVAC instruction is the address supplied to the instruction, rounded down to the nearest multiple of the access size initiated by that instruction.

The highest address accessed is (size - 1) bytes above the lowest address accessed.

See also, *Watchpoint behavior on accesses by DCIMVAC instructions* on page G2-5864.

Programming a watchpoint with eight bytes or fewer

The Byte Address Select field, DBGWCR<n>.BAS, selects which bytes in the doubleword starting at the address contained in the DBGWVR<n> the watchpoint generates Watchpoint exceptions for.

If the address programmed into the DBGWVR<n> is:

- Doubleword-aligned:
  - All eight bits of DBGWCR<n>.BAS are used, and the descriptions given in Table G2-16 apply.
- Word-aligned but not doubleword-aligned:
  - Only DBGWCR<n>.BAS[3:0] are used, and the descriptions given in Table G2-17 apply. In this case, DBGWCR<n>.BAS[7:4] are RES0.

Table G2-16 Supported BAS values when the DBGWVRn address alignment is doubleword

BAS value	Description
0b00000000	Watchpoint never generates a Watchpoint exception
BAS[0] == 1	Generates a Watchpoint exception if byte at address DBGWVR<n>[31:3]:000 is accessed
BAS[1] == 1	Generates a Watchpoint exception if byte at address DBGWVR<n>[31:3]:001 is accessed
BAS[2] == 1	Generates a Watchpoint exception if byte at address DBGWVR<n>[31:3]:010 is accessed
BAS[3] == 1	Generates a Watchpoint exception if byte at address DBGWVR<n>[31:3]:011 is accessed
BAS[4] == 1	Generates a Watchpoint exception if byte at address DBGWVR<n>[31:3]:100 is accessed
BAS[5] == 1	Generates a Watchpoint exception if byte at address DBGWVR<n>[31:3]:101 is accessed
BAS[6] == 1	Generates a Watchpoint exception if byte at address DBGWVR<n>[31:3]:110 is accessed
BAS[7] == 1	Generates a Watchpoint exception if byte at address DBGWVR<n>[31:3]:111 is accessed

Table G2-17 Supported BAS values when the DBGWVRn address alignment is word

BAS value[a]	Description
0b00000000	Watchpoint never generates a Watchpoint exception
BAS[0] == 1	Generates a Watchpoint exception if byte at address DBGWVR<n>[31:2]:00 is accessed
BAS[1] == 1	Generates a Watchpoint exception if byte at address DBGWVR<n>[31:2]:01 is accessed
BAS[2] == 1	Generates a Watchpoint exception if byte at address DBGWVR<n>[31:2]:10 is accessed
BAS[3] == 1	Generates a Watchpoint exception if byte at address DBGWVR<n>[31:2]:11 is accessed

a. DBGWCR<n>.BAS[7:4] are RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the BAS field is programmed with more than one byte, the bytes that it is programmed with must be contiguous. For watchpoint behavior when its BAS field is programmed with non-contiguous bytes, see *Other usage constraints* on page G2-5866.

When programming the BAS field with anything other than `0b11111111`, a debugger must also program DBGWCR<n>.MASK to be `0b00000`. See *Programming dependencies of the BAS and MASK fields* on page G2-5865.

A watchpoint generates a Watchpoint exception whenever a watched byte is accessed, even if:

- The access size is smaller or larger than the address region being watched.
- The access is misaligned, and the base address of the access is not in the doubleword or word of memory addressed by the DBGWVR<n>[31:3]. See Example G2-1 on page G2-5859.

The following are some example configurations of the BAS field:

- To program a watchpoint to generate a Watchpoint exception on the byte at address `0x1003`, program:
  - DBGWVR<n> with `0x1000`.
  - DBGWCR<n>_EL1.BAS to be `0b00001000`.
- To program a watchpoint to generate a Watchpoint exception on the bytes at addresses `0x2003`, `0x2004` and `0x2005`, program:
  - DBGWVR<n> with `0x2000`.
  - DBGWCR<n>_EL1.BAS to be `0b00111000`.
- If the address programmed into the DBGWVR<n> is doubleword-aligned:
  - To generate a Watchpoint exception when any byte in the word starting at the doubleword-aligned address is accessed, program DBGWCR<n>.BAS to be `0b00001111`.
  - To generate a Watchpoint exception when any byte in the word starting at address DBGWVR<n>[31:3]:100 is accessed, program DBGWCR<n>.BAS to be `0b11110000`.

Note

Arm deprecates programming a DBGWVR<n> with an address that is not doubleword-aligned.

Programming a watchpoint with eight or more bytes

A debugger can use the *MASK* field, DBGWCR<n>.MASK, to program a single watchpoint with a data address range. The data address range must meet all of the following criteria:

- It is a size that is both:
  - A power-of-two.
  - A minimum of eight bytes.
  - A maximum of 2GB.
- It starts at an address that is aligned to the size.

The MASK field specifies the number of least significant data address bits that must be masked. Up to 31 least significant bits can be masked:

MASK	`0b00000`	No bits are masked.
	`0b00001`	Reserved.
	`0b00010`	Reserved.
	`0b00011`	Three least significant bits are masked.
	`0b00100`	Four least significant bits are masked.
	`0b00101`	Five least significant bits are masked.
	...	…
	`0b11111`	31 least significant bits are masked.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If *n* least significant address bits are masked, the watchpoint generates a Watchpoint exception on all of the following:

- Address DBGWVR<n>[31:*n*]:000…
- Address DBGWVR<n>[31:*n*]:111…
- Any address between these two addresses.

For example, if the four least significant address bits are masked, Watchpoint exceptions are generated for all addresses between DBGWVR<n>[31:4]:0000 and DBGWVR<n>[31:4]:1111, including these addresses.

Note

- The most significant bit cannot be masked. This means that the full address cannot be masked.
- For watchpoint behavior when its MASK field is programmed with a reserved value, see *Reserved DBGWCR<n>.MASK values* on page G2-5866.

When masking address bits, a debugger must both:

- Program DBGWCR<n>.BAS to be `0b11111111`. See *Programming dependencies of the BAS and MASK fields* on page G2-5865.
- In the DBGWVR<n>, set the masked address bits to 0. For watchpoint behavior when any of the masked address bits are not 0, see *Other usage constraints* on page G2-5866.

G2.10.5 Determining the memory location that caused a Watchpoint exception

On a Watchpoint exception, the PE records an address in a *Fault Address Register* that the debugger can use to determine the memory location that triggered the watchpoint.

The Fault Address Register (FAR) used is either:

- DFAR, if the exception is taken to PL1.
- HDFAR, if the exception is taken to PL2.

In cases where one instruction triggers multiple watchpoints, only one address is recorded.

On entering Debug state on a Watchpoint debug event, the PE records the address in the EDWAR.

Note

If Debug state was entered from AArch32 state, then EDWAR[63:32] is UNKNOWN and must be ignored by the debugger.

For more information, see the subsections that follow. These are:

- *Address recorded for Watchpoint exceptions generated by instructions other than data cache maintenance instructions*.
- *Address recorded for Watchpoint exceptions generated by data cache maintenance instructions* on page G2-5863.

Address recorded for Watchpoint exceptions generated by instructions other than data cache maintenance instructions

The address recorded must be both:

- From the inclusive range between:
  - The lowest address accessed by the memory access that triggered the watchpoint.
  - The highest *watchpointed address* accessed by the memory access. A watchpointed address is an address that the watchpoint is watching.
- Within a naturally-aligned block of memory that is all of the following:
  - A power-of-two size.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— No larger than 2KB.
— No larger than the block size used by the A64 DC ZVA instruction.

Note

There are no architectural means to discover the A64 DC ZVA instruction block size from AArch32 state.

— Contains a watchpointed address accessed by the memory access.

The size of the block is IMPLEMENTATION DEFINED. There is no architectural means of discovering the size.

Example G2-2 Address recorded for a watchpoint programmed on `0x8019`

A debugger programs a watchpoint to generate a Watchpoint exception on any access to the byte `0x8019`.

An A32 load multiple instruction then loads nine registers starting from address `0x8004` upwards. This triggers the watchpoint.

If the `DC ZVA` block size is:

- 32 bytes, the address that the PE records must be between `0x8004` and `0x8019` inclusive.
- 16 bytes, the address that the PE records must be between `0x8010` and `0x8019` inclusive.

Address recorded for Watchpoint exceptions generated by data cache maintenance instructions

The address recorded is the address passed to the instruction. This means that the address recorded might be higher than the address of the location that triggered the watchpoint.

G2.10.6 Watchpoint behavior on other instructions

Under normal operating conditions, the following do not generate Watchpoint exceptions:

- Instruction cache maintenance instructions.
- Address translation instructions.
- TLB maintenance instructions.
- Preload instructions.
- All data cache maintenance instructions except `DCIMVAC`.

However, the debug architecture allows for IMPLEMENTATION DEFINED controls, such as those in ACTLR registers, to enable watchpoints on an implementation defined subset of these instructions. Whether a watchpoint treats the instruction as a load or a store, and the access size of instruction cache maintenance, address translation, and TLB maintenance instructions are implementation defined.

The access size of the IMPLEMENTATION DEFINED instruction cache maintenance, address translation, and TLB maintenance instructions that generate Watchpoint exceptions are IMPLEMENTATION DEFINED.

See also:

- *Watchpoint behavior on accesses by Store-Exclusive instructions*.
- *Watchpoint behavior on accesses by DCIMVAC instructions* on page G2-5864.

Watchpoint behavior on accesses by Store-Exclusive instructions

If a watchpoint matches on a data access caused by a Store-Exclusive instruction, then:

- If the store fails because an Exclusives monitor does not permit it, it is IMPLEMENTATION DEFINED whether the watchpoint generates a Watchpoint exception.
- Otherwise, the watchpoint generates a Watchpoint exception.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Watchpoint behavior on accesses by DCIMVAC instructions

It is IMPLEMENTATION DEFINED whether DCIMVAC operations can generate Watchpoint exceptions. If they can, they are treated as data stores. This means that for a watchpoint to match on an access caused by a DCIMVAC instruction, the debugger must program DBGWCR<n>.LSC to be one of the following:

10 Match on data stores only.

11 Match on data stores and data loads.

Note

For the size of data accesses performed by DCIMVAC instructions, see *Watchpoint data address comparisons* on page G2-5859. The size of all data accesses must be considered because watchpoints can be programmed to match on individual bytes.

G2.10.7 Usage constraints

See the following:

- *Reserved DBGWCR<n>.{SSC, HMC, PAC} values*.
- *Reserved DBGWCR<n>.LBN values* on page G2-5865.
- *Programming dependencies of the BAS and MASK fields* on page G2-5865.
- *Reserved DBGWCR<n>.BAS values* on page G2-5865.
- *Reserved DBGWCR<n>.MASK values* on page G2-5866.
- *Other usage constraints* on page G2-5866.

Reserved DBGWCR<n>.{SSC, HMC, PAC} values

Table G2-18 shows when particular combinations of DBGWCR<n>.{SSC, HMC, PAC} are reserved.

Table G2-18 Reserved SSC, HMC, and PAC combinations

HMC, SSC, and PAC combination	Reserved
All combinations with SSC set to 0b01 or 0b10, except for the combination with HMC set to 1, SSC set to 0b01 and PAC set to 0b00	When EL3 is not implemented and EL2 is implemented.
Any combination where HMC or SSC is nonzero	When both of EL2 and EL3 are not implemented
The combination with HMC set to 1, SSC set to 0b11, and PAC set to 0b00	When EL2 is not implemented
The combinations with SSC set to 0b11 and PAC set to 0b01 or 0b11	When Secure EL2 is not implemented
The combination with HMC set to 1, SSC set to 0b01 and PAC set to 0b00	When Secure EL2 is not implemented
Combinations not included in Table G2-15 on page G2-5858.	Always

If a watchpoint is programmed with one of these reserved combinations:

- The watchpoint must behave as if it is either:
  - Disabled.
  - Programmed with a combination that is not reserved, other than for a direct or external read of DBGWCR<n>.
- For a direct or external read of DBGWCR<n>, if the reserved combination:
  - Has no function for any execution conditions, the value read back for each of SSC, HMC, and PMC is UNKNOWN.
  - Has a function for execution conditions other than the current execution conditions, the value read back is the value written. This permits software to save and restore the combination so that the watchpoint functions for the other execution conditions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The behavior of watchpoints with reserved combinations of SSC, HMC, and PAC might change in future revisions of the architecture. For this reason, software must not rely on the behavior described here.

Reserved DBGWCR<n>.LBN values

For Linked watchpoints

A Linked watchpoint must link to a context-aware breakpoint. For a Linked watchpoint, any DBGWCR<n>.LBN value that is not for a context-aware breakpoint is reserved.

If a Linked watchpoint links to a breakpoint that is not implemented, or that is not context-aware, then reads of DBGWCR<n>.LBN return an UNKNOWN value and the behavior is CONSTRAINED UNPREDICTABLE. The Linked watchpoint behaves as if it is either:

- Disabled.
- Linked to an UNKNOWN context-aware breakpoint.

If a Linked watchpoint links to a breakpoint that is implemented and is context-aware, but that is either not enabled or not programmed as a Linked Context breakpoint, it behaves as if it is disabled.

For Unlinked watchpoints

For Unlinked watchpoints, DBGWCR<n>.LBN reads UNKNOWN and its value is ignored.

Programming dependencies of the BAS and MASK fields

When programming a watchpoint, a debugger must use either:

- The MASK field, to program the watchpoint with an address range that can be eight bytes to 2GB.
- The BAS field, to select which bytes in the doubleword or word starting at the address contained in the DBGWVR<n> the watchpoint must generate Watchpoint exceptions for.

If the debugger uses the:

- MASK field, it must program BAS to be `0b11111111`, so that all bytes in the doubleword or word are selected.
- BAS field, it must program MASK to be `0b00000`, so that the MASK field does not indicate any address ranges.

If an enabled watchpoint has a MASK field that is non-zero and a BAS field that is not set to 0b11111111, then for each byte in the address range, it is CONSTRAINED UNPREDICTABLE whether or not a Watchpoint exception is generated.

Reserved DBGWCR<n>.BAS values

The BAS field must be programmed with a value `Zeros(8-n-m):Ones(n):Zeros(m)`, where:

- `n` is a non-zero positive integer less-than-or-equal-to 8.
- `m` is a positive integer less-than 8.
- `n+m` is less-than-or-equal-to 8.

All other values are reserved.

Note

If `x` is zero, then `Zeros(x)` is an empty bitstring.

If DBGWVR<n>[2] is 1, DBGWCR<n>.BAS[7:4] are RES0 and are ignored.

If a watchpoint is programmed with a reserved BAS value:

- It is CONSTRAINED UNPREDICTABLE whether the watchpoint generates a Watchpoint exception for each byte in the doubleword or word of memory addressed by the DBGWVR<n>.
- A direct or external read of DBGWCR<n>.BAS returns an UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Software must not rely on these properties as the behavior of reserved values might change in a future revision of the architecture.

Reserved DBGWCR<n>.MASK values

If a watchpoint is programmed with a reserved MASK value:

- The watchpoint must behave as if it is either:
  - Disabled.
  - Programmed with an UNKNOWN value that is not reserved, that might be `0b00000`, other than for a direct or external read of DBGWCR<n>.
- A direct or external read of DBGWCR<n>.MASK returns an UNKNOWN value.

Other usage constraints

For all watchpoints:

- DBGWVR<n>[1:0] are RES0 and are ignored.
- If DBGWCR<n>.MASK is nonzero, and any masked bits of DBGWVR<n> are not 0, it is CONSTRAINED UNPREDICTABLE whether the watchpoint generates a Watchpoint exception when the unmasked bits match.
- A watchpoint never generates any Watchpoint exceptions if DBGWCR<n>.LSC is `0b00`.

G2.10.8 Exception syndrome information and preferred return address

See the following:

- *Exception syndrome information*.
- *Preferred return address* on page G2-5867.

Exception syndrome information

The PE takes a Watchpoint exception as either:

- A Data Abort exception, if it is taken to PL1. In this case, it is taken to Abort mode.
- A Hyp trap exception, if it is taken to PL2 because HCR.TGE or HDCR.TDE is 1. In this case, it is taken to Hyp mode.

If the exception is taken to:

Abort mode

The PE sets all of the following:

- DBGDSCRext.MOE to `0b1010`, to indicate a Watchpoint exception.
- DFSR.CM to indicate whether a cache maintenance instruction caused the exception.
- DFSR.WnR to indicate whether the exception was generated on a read instruction or a write instruction.
- DFAR to an address that the debugger can use to determine the memory location that triggered the watchpoint. See *Determining the memory location that caused a Watchpoint exception* on page G2-5862.

In addition, if using the:

- Short-descriptor format, the PE sets DFSR.FS to the code for a debug exception, `0b00010`, and DFSR.Domain to an UNKNOWN value.
- Long-descriptor format, the PE sets DFSR.STATUS to the code for a debug exception, `0b100010`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Hyp mode

The PE does all of the following:

- Records information about the exception in the *Hypervisor Syndrome Register*, HSR. See Table G2-19.
- Sets DBGDSCRext.MOE to `0b1001`, to indicate a Watchpoint exception.
- Sets the HDFAR to an address that the debugger can use to determine the memory location that triggered the watchpoint. See *Determining the memory location that caused a Watchpoint exception* on page G2-5862.

Table G2-19 Information recorded in the HSR

HSR field	Information recorded
Exception Class, EC	The PE sets this to the code for a Data Abort exception routed to Hyp mode, `0x24`.
Instruction Length, IL	The PE sets this to 1.
Instruction Specific Syndrome, ISS	**ISV[24]** *Instruction Syndrome Valid* (ISV). The PE sets this to 0. **ISS[23:10]** RES0. **ISS[9]** *External Abort type* (EA). The PE sets this to 0. **ISS[8]** *Cache Maintenance* (CM). The PE sets this to indicate whether a cache maintenance instruction caused the exception. **ISS[7]** RES0. **ISS[6]** *Write not Read* (WnR). The PE sets this to indicate whether the exception was generated on a read instruction or a write instruction. **ISS[5:0]** *Data Fault Status Code* (DFSC). The PE sets this to the code for a debug exception, `0b100010`.

Note

For information about how debug exceptions can be routed to PL2, see *Routing debug exceptions* on page G2-5819.

Preferred return address

The preferred return address of a Watchpoint exception is the address of the instruction that was not executed because the PE took the Watchpoint exception instead.

This means that the preferred return address is the address of the instruction that caused the exception.

G2.10.9 Pseudocode description of Watchpoint exceptions taken from AArch32 state

`AArch32.WatchpointByteMatch()` tests an individual byte accessed by an operation.

`AArch32.StateMatch()` tests the values in DBGWCR<n>.{HMC, SSC, PAC}, and if the watchpoint is Linked, also tests the Linked Context breakpoint that the watchpoint links to.

`AArch32.WatchpointMatch()` tests the value in DBGWVR<n>.

`AArch32.CheckWatchpoint()` generates a `FaultRecord`. A Watchpoint exception is taken if all of the following are true:

- DBGDSCRext.MDBGen is 1.
- Debug exceptions are enabled from the current Exception level and Security state. See *Enabling debug exceptions from the current Privilege level and Security state* on page G2-5821.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- All of the conditions required for Watchpoint exception generation are met. See *About Watchpoint exceptions* on page G2-5855.

Note

`AArch32.CheckWatchpoint` might halt the PE and cause it to enter Debug state. External debug uses Debug state.

The `AArch32.Abort()` function processes the `FaultRecord` object returned by `AArch32.CheckWatchpoint()`, as described in *Abort exceptions* on page G4-5919. If a Watchpoint exception is taken to AArch32 state, the `AArch32.Abort()` function generates a Data Abort exception.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G2.11 Vector Catch exceptions

Arm deprecates the use of vector catch.

This section describes Vector Catch exceptions in stage 1 of an AArch32 translation regime.

The PE is using an AArch32 translation regime when it is executing either:

- At EL1 or higher in an Exception level that is using AArch32.
- At EL0 using AArch32 when EL1 is using AArch32.

Note

Vector Catch exceptions cannot be generated when the PE is using an AArch64 translation regime.

This section contains the following subsections:

- *About Vector Catch exceptions*.
- *Exception vectors that Vector Catch exceptions can be enabled for* on page G2-5871.
- *Generation of Vector Catch exceptions* on page G2-5872.
- *Usage constraints* on page G2-5874.
- *Exception syndrome information and preferred return address for a Vector Catch exception* on page G2-5874.
- *Pseudocode description of Vector Catch exceptions* on page G2-5875.

G2.11.1 About Vector Catch exceptions

Whenever the PE takes an exception, execution is forced to an address that is the *exception vector* for that exception. Vector catch permits a debugger to trap exceptions based on the exception vector, or based on the exception type associated with the exception vector, as follows:

- If the *address-matching* form of vector catch is implemented, the debugger can trap exceptions based on the exception vector.
- If the *exception-trapping* form of vector catch is implemented, the debugger can trap exceptions based on the exception type associated with the exception vector.

The Armv8-A architecture supports only these two forms of vector catch. Only one form can be implemented, and which is implemented is IMPLEMENTATION DEFINED. The DBGDEVID indicates which form is implemented.

Regardless of the form of vector catch implemented, a debugger enables Vector Catch exceptions for exception vectors or types by programming the DBGVCR. This register contains *vector catch enable bits*. Each of these bits corresponds to a different vector. When a debugger sets a vector catch enable bit to 1, Vector Catch exceptions are enabled for the corresponding exception vector or type.

Note

EL2 using AArch64 or EL3 using AArch64 can enable Vector Catch exceptions for vectors by programming the DBGVCR32_EL2. The DBGVCR32_EL2 is architecturally mapped to the DBGVCR.

When Vector Catch exceptions are enabled for an exception vector, this is called an *enabled vector catch*. The set of exception vectors that Vector Catch exceptions are enabled for is called the *enabled vector catch set*.

If the form of vector catch implemented is the:

Address-matching form:

The PE compares the virtual address of each instruction in the program flow with a subset of the enabled vector catch set.

If an address match occurs, a Vector Catch exception is generated when the instruction that caused the match is committed for execution.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Exception-trapping form

Whenever the PE takes an exception, if the vector the exception is taken to is included in a subset of the enabled vector catch set, a Vector Catch exception is generated.

The Vector Catch exception is generated as part of entry to the exception, and must be taken before the PE either executes any instructions or takes any further exceptions.

The addresses that comprise the subset depend on whether EL3 is implemented and, for the:

- Address-matching form, the current Security state.
- Exception-trapping form, the Security state that the exception is handled in.

See *Generation of Vector Catch exceptions* on page G2-5872.

Table G2-20 summarizes the differences between the address-matching and exception-trapping forms.

Table G2-20 Differences in behavior of the address-matching and exception-trapping forms of vector catch

Address-matching	Exception-trapping
An enabled vector catch generates a Vector Catch exception when an instruction that is fetched from the vector is committed for execution. This means that spurious Vector Catch exceptions might occur, where the Vector Catch exception does not result from an exception entry, but is instead caused by a branch to the vector. A branch to the vector might occur, for example, on a return from a nested exception or when simulating an exception entry.	An enabled vector catch generates a Vector Catch exception immediately after the PE takes the exception that is associated with the vector. This means that Vector Catch exceptions always result from exception entry, and not from branches to exception vectors.
A Vector Catch exception is generated as a result of an instruction fetch. This means that the Vector Catch exception has a priority relative to the other synchronous exceptions that result from an instruction fetch. *Synchronous exception prioritization for exceptions taken to AArch64 state* on page D1-2355 describes this prioritization.	A Vector Catch exception is generated as a result of an exception entry. This means that the Vector Catch exception is part of the exception that caused the Vector Catch exception. Therefore, the Vector Catch exception has no priority associated with it. For this reason, Vector Catch exceptions are outside the scope of the prioritization that *Synchronous exception prioritization for exceptions taken to AArch64 state* on page D1-2355 describes.
A Vector Catch exception can be preempted by another exception. If this happens, the Vector Catch exception is generated again when the exception handler branches back to the vector.	Vector Catch exceptions must be taken before other exceptions.
A Vector Catch exception can be generated as a result of an instruction fetch executed in any AArch32 mode except Hyp mode, including User mode.	Because a Vector Catch exception is generated as the result of an exception entry, the Vector Catch exception is only generated when the PE is in the AArch32 exception handling mode.
If HCR.TGE is 1, Vector Catch exceptions can be generated for User mode instruction fetches from Non-secure PL1 vectors.	If HCR.TGE is 1, Vector Catch exceptions are never generated in Non-secure state, because: • Exceptions are routed away from Non-secure PL1 vectors, to PL2. • The architecture does not provide vector catch enable bits for the Hyp exception vectors.

Depending on the implementation, some vector catch enable bits in the DBGVCR might be RES0. For example, if EL3 is not implemented or is implemented but is using AArch64, Monitor mode is not implemented, and so the enable bits for exception vectors for exceptions taken to Monitor mode are RES0. See *Exception vectors that Vector Catch exceptions can be enabled for* on page G2-5871 for the vector catch enable bits that exist for different implementations.

The debug exception enable controls on page G2-5818 describes the enable controls for Vector Catch exceptions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G2.11.2 Exception vectors that Vector Catch exceptions can be enabled for

When the PE takes an exception, the exception vector is contained in a *vector table* at the Privilege level the exception is taken to.

Depending on the Security state and AArch32 mode the exception is taken to, when the exception is taken, the vector table used is the table that contains one of:

- *Local exception vectors*.
- *Non-secure Local exception vectors*.
- *Secure Local exception vectors*.
- *Hyp exception vectors*.
- *Monitor exception vectors*.

Table G2-21 shows which vector tables are implemented for different implementations. In the table:

- A dash, -, means that the Exception level is not implemented.
- 64 means that the Exception level is using AArch64.
- 32 means that the Exception level is using AArch32.

Table G2-21 Vector tables implemented for different implementations

Implementation				Vector table or tables implemented
EL0	EL1	EL2	EL3	
32	32	-	-	Local exception vectors.
		64	-	Non-secure Local exception vectors.
		32	-	Non-secure Local exception vectors. Hyp exception vectors.
		-	64	Secure Local exception vectors. Non-secure Local exception vectors.
		-	32	Secure Local exception vectors. Non-secure Local exception vectors. Monitor exception vectors.
		64	64	Secure Local exception vectors. Non-secure Local exception vectors.
		32	64	Secure Local exception vectors. Non-secure Local exception vectors. Hyp exception vectors.
		32	32	Secure Local exception vectors. Non-secure Local exception vectors. Hyp exception vectors. Monitor exception vectors.

For example, in an AArch32-only implementation that includes EL0, EL1, and EL3, when the PE takes an exception to Monitor mode, it uses the vector table containing Monitor exception vectors.

The tables that follow show the vectors that Vector Catch exceptions can be enabled for, and their corresponding vector catch enable bits in the DBGVCR:

- Table G2-22 on page G2-5872 shows the Local exception vectors, Secure Local exception vectors, and Non-secure Local exception vectors that Vector Catch exceptions can be enabled for.
- Table G2-23 shows the Monitor exception vectors that Vector Catch exceptions can be enabled for.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The Armv8-A architecture does not provide vector catch enable bits for the Hyp exception vectors.

Table G2-22 Local exception vectors, Secure Local exception vectors, and Non-secure Local exception vectors that Vector Catch exceptions can be enabled for

Vector catch enable bit		Exception type	Local exception vectors	
Local or Secure Local exception vectors	Non-secure Local exception vectors		Normal. SCTLR.V is 0.[a]	High. SCTLR.V is 1.
SF	NSF	FIQ interrupt	VBAR + `0x0000001C`	`0xFFFF001C`
SI	NSI	IRQ interrupt	VBAR + `0x00000018`	`0xFFFF0018`
SD	NSD	Data Abort	VBAR + `0x00000010`	`0xFFFF0010`
SP	NSP	Prefetch Abort	VBAR + `0x0000000C`	`0xFFFF000C`
SS	NSS	Supervisor Call	VBAR + `0x00000008`	`0xFFFF0008`
SU	NSU	Undefined Instruction	VBAR + `0x00000004`	`0xFFFF0004`

a. If EL3 is implemented and is using AArch32, VBAR is banked. The Secure Local exception vectors use $VBAR_S$ and the Non-secure Local Exception vectors use $VBAR_{NS}$.

Table G2-23 Monitor exception vectors that Vector Catch exceptions can be enabled for

Vector catch enable bit	Exception type	Monitor exception vectors
MF	FIQ interrupt	MVBAR + `0x0000001C`
MI	IRQ interrupt	MVBAR + `0x00000018`
MD	Data Abort	MVBAR + `0x00000010`
MP	Prefetch Abort	MVBAR + `0x0000000C`
MS	Secure Monitor Call	MVBAR + `0x00000008`

Note

There is no vector catch enable bit for Monitor trap exceptions.

G2.11.3 Generation of Vector Catch exceptions

How Vector Catch exceptions are generated depends on which form is implemented:

- *Address-matching form*.
- *Exception-trapping form* on page G2-5873.

Address-matching form

The PE compares the virtual address of each instruction in the program flow is with some or all of the addresses in the enabled vector catch set, as follows:

- If EL3 is not implemented, the enabled vector catch set contains only Local exception vectors. The PE compares the virtual address of each instruction in the program flow, including those executed at EL0, with all addresses in the enabled vector catch set.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If EL3 is implemented, the enabled vector catch set might contain one or more of the following:
  - Monitor exception vectors, if EL3 is using AArch32.
  - Secure Local exception vectors.
  - Non-secure Local exception vectors.

  In this case, Table G2-24 shows which addresses, in the enabled vector catch set, the virtual address of each instruction in the program flow is compared with.

Table G2-24 Comparisons made if the implementation includes EL3

EL3 is using	For exceptions taken to:	
	Secure PL1 modes	Non-secure PL1 modes
AArch64	Secure Local exception vectors	Non-secure Local exception vectors
AArch32	Secure Local exception vectors and Monitor exception vectors	

For example, for exceptions taken to a Secure PL1 mode when EL3 is using AArch64, the virtual address of each instruction in the program flow is compared with each Secure Local exception vector in the enabled vector catch set.

For each instruction in the program flow, the PE tests for any possible Vector Catch exceptions before executing the instruction. If a match occurs, a Vector Catch exception is generated when the instruction is committed for execution, regardless of all of the following:

- Whether the instruction passes its Condition code check.
- Whether the instruction is executed as part of exception entry.
- If EL2 is implemented, what HCR.{IMO, FMO, AMO} are set to.
- If EL3 is implemented, what SCR.{IRQ, FIQ, EA} are set to.

Exception-trapping form

When the PE takes an exception, it tests whether the exception is by branching to an exception vector in a subset of the enabled vector catch set, as follows:

- If EL3 is not implemented, the enabled vector catch set contains only Local exception vectors. The PE tests whether the exception is by branching to any address in the enabled vector catch set.
- If EL3 is implemented, the enabled vector catch set might contain one or more of the following:
  - Monitor exception vectors, if EL3 is using AArch32.
  - Secure Local exception vectors.
  - Non-secure Local exception vectors.

  In this case, the PE tests whether the exception is by branching to a vector in one of the subsets that Table G2-25 shows. In the table, n/a means not applicable.

Table G2-25 Subsets that the PE tests within if EL3 is implemented

EL3 is using	For exceptions taken to:		
	Monitor mode	Other Secure PL1 modes	Non-secure PL1 modes
AArch64	n/a	Secure Local exception vectors	Non-secure Local exception vectors
AArch32	Monitor exception vectors		

For example, for an exception taken to a Secure PL1 mode when EL3 is using AArch64, the PE tests whether the exception is by branching to any of the Secure Local exception vectors in the enabled vector address set.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the exception is by branching to a vector in the subset, a Vector Catch exception is generated as part of exception entry. That is, a Vector Catch exception is generated instead of the exception handler executing its first instruction.

G2.11.4 Usage constraints

See the following subsections:

- *Usage constraints that apply to both forms of vector catch*.
- *Usage constraints that apply only to the address-matching form*.

Usage constraints that apply to both forms of vector catch

For Vector Catch exceptions enabled for either the Prefetch Abort exception vector or the Data Abort exception vector, if one of these exception types is taken to the Exception level that debug exceptions are targeting, behavior is CONSTRAINED UNPREDICTABLE. Either:

- Vector catch is ignored, therefore a Vector Catch exception is not generated.
- Vector catch generates a Prefetch Abort debug exception. For Vector Catch exceptions enabled for the Prefetch Abort exception vector, the PE might enter a recursive loop of Prefetch Abort exceptions causing Vector Catch exceptions and Vector Catch exceptions causing Prefetch Abort exceptions.

Note

The Exception level that debug exceptions are targeting is called the *debug target Exception level*, EL_D. *Routing debug exceptions* on page G2-5819 describes how EL_D is derived.

Usage constraints that apply only to the address-matching form

Exception vectors are at word-aligned addresses, and:

- It is CONSTRAINED UNPREDICTABLE whether an enabled vector catch generates a Vector Catch exception for a 32-bit T32 instruction starting at the halfword-aligned address immediately prior to the vector address.
- T32 instructions that start at the halfword-aligned address immediately after the exception vector do not generate Vector Catch exceptions.

For the address-matching form, Vector Catch exceptions have the same priority as Breakpoint exceptions. If a single instruction causes both a Vector Catch exception and a Breakpoint exception, it is CONSTRAINED UNPREDICTABLE which of these debug exceptions the PE takes.

G2.11.5 Exception syndrome information and preferred return address for a Vector Catch exception

See the following:

- *Exception syndrome information for a Vector Catch exception*.
- *Preferred return address for a Vector Catch exception* on page G2-5875.

Note

Usually, the term *exception syndrome* is used only for exceptions taken to Hyp mode, or to AArch64 state. The referenced section uses the term more generally, to include exception information reported in the IFSR.

Exception syndrome information for a Vector Catch exception

The PE takes a Vector Catch exception as either:

- A Prefetch Abort exception if it is taken to PL1. In this case, it is taken to Abort mode.
- A Hyp trap exception, if it is taken to PL2 because HCR.TGE or HDCR.TDE is 1. In this case, it is taken to Hyp mode.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the exception is taken to:

PL1 Abort mode

The PE sets all of the following:

- IFSR.FS to the code for a debug exception, `0b00010`.
- DBGDSCRext.MOE to `0b0101`, to indicate a Vector Catch exception.
- The IFAR with an UNKNOWN value.

PL2 Hyp mode

The PE does all of the following:

- Records information about the exception in the *Hypervisor Syndrome Register*, HSR. See Table G2-26.
- Sets DBGDSCRext.MOE to `0b0101`, to indicate a Vector Catch exception.
- Sets the HIFAR to an unknown value.

Table G2-26 Information recorded in the HSR

HSR field	Information recorded
Exception Class, EC	The PE sets this to the code for a Prefetch Abort exception routed to Hyp mode, `0x20`.
Instruction Length, IL	The PE sets this to 1.
Instruction Specific Syndrome, ISS	**ISS[24:10]** RES0. **ISS[9]** *External Abort type* (EA). The PE sets this to 0. **ISS[8:6]** RES0. **ISS[5:0]** *Instruction Fault Status Code* (IFSC). The PE sets this to the code for a debug exception, `0b100010`.

Note

For information about how debug exceptions can be routed to PL2, see *Routing debug exceptions* on page G2-5819.

Preferred return address for a Vector Catch exception

The preferred return address of a Vector Catch exceptions is the address of the instruction that was not executed because the PE took the Vector Catch exception instead.

This means that the preferred return address is the exception vector. This is true regardless of whether the address-matching form or the exception trapping form is implemented.

G2.11.6 Pseudocode description of Vector Catch exceptions

The `AArch32.VCRMatch()` pseudocode function checks whether the instruction at `address` generates a Vector Catch exception. It therefore shows the address-matching form of vector catch.

The `AArch32.CheckVectorCatch()` pseudocode function uses `AArch32.VCRMatch()` to test whether the instruction generates a Vector Catch exception, and if `AArch32.VCRMatch()` returns TRUE it generates that event.

The `AArch32.Abort()` function processes the `FaultRecord` object returned by `AArch32.CheckVectorCatch()`, as described in *Abort exceptions* on page G4-5919. If there is a Vector Catch exception, the `AArch32.Abort()` function generates a Prefetch Abort exception.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G2.12 Synchronization and debug exceptions

The behavior of debug depends on all of the following:

- The state of the external debug authentication interface.
- Indirect reads of:
  - External debug registers.
  - System registers, including system debug registers.
  - Special-purpose registers.

If a change is made to any of these, the effect of that change on debug exception generation cannot be relied on until after a *Context synchronization event* has occurred.

For any instructions executed between the time when the change is made and the time when the next *Context synchronization event* occurs, it is CONSTRAINED UNPREDICTABLE whether debug uses the state of the PE before the change, or the state of the PE after the change.

Example G2-3

1. Software changes DBGDSCRext.MDBGen from 0 to 1.
2. An instruction is executed, that would cause a Breakpoint exception if self-hosted debug uses the state of the PE after the change.
3. A *Context synchronization event* occurs.

In this case, it is CONSTRAINED UNPREDICTABLE whether the instruction generates a Breakpoint exception.

Example G2-4

1. Software unlocks the OS Lock.
2. The PE executes some instructions.
3. A *Context synchronization event* occurs.

During the time when the PE is executing some instructions, step 2, it is CONSTRAINED UNPREDICTABLE whether debug exceptions other than Breakpoint Instruction exceptions can be generated.

Note

Some register updates are self-synchronizing. Others require an explicit *Context synchronization event*. For more information, see:

- *Synchronization of changes to AArch32 System registers* on page G8-6100.
- *Accessing PSTATE fields* on page G1-5696.
- *Synchronization of changes to the external debug registers* on page H8-7050.

G2.12.1 State and mode changes without explicit context synchronization events

Most changes to the Exception level, and most changes to the Security state if EL3 is implemented, happen as a result of operations that are an explicit *Context synchronization event*. This is because taking an exception and returning from an exception are both explicit *Context synchronization event*s, and the Privilege level and Security state can only change as a result of taking or returning from an exception.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

However, some Security state and AArch32 mode changes can happen because of operations that are not an explicit *Context synchronization event*. These are:

- AArch32 mode changes caused by `MSR` and `CPS` instructions. A mode change might be to a mode at a lower Privilege level.
- If EL3 is using AArch32, a Security state change caused by a direct write to the SCR in a privileged mode other than Monitor mode, to set SCR.NS to 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter G3
AArch32 Self-hosted Trace

This chapter describes the AArch32 self-hosted trace:

Introductory information:

- *About self-hosted trace* on page G3-5880.
- *Trace Sinks* on page G3-5880.
- *Register controls to enable self-hosted trace* on page G3-5880.

Prohibited regions in trace:

- *Controls to prohibit trace at Exception levels* on page G3-5881.
- *Self-hosted trace and address translation* on page G3-5881.

Timestamps and Synchronization:

- *Self-hosted trace timestamps* on page G3-5882.
- *Synchronization in self-hosted trace* on page G3-5883.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G3.1 About self-hosted trace

A PE Trace Unit generates trace data to describe the program flow of the PE.

The PE Trace Unit may be an implementation of a standard Arm Embedded Trace Macrocell (ETM), or another type of Arm Trace Architecture, or an IMPLEMENTATION DEFINED trace function.

If an Armv8.4-compliant PE implements an ETM Architecture PE Trace Unit, ARMv8.4-Trace extension must be implemented.

If an Armv8.4-compliant PE implements a Trace Unit that is not an ETM Architecture PE Trace Unit, Arm recommends that ARMv8.4-Trace extension is implemented, but this is not mandatory.

Self-hosted trace happens when the agent controlling the trace collection is part of the same software stack as the software being traced. The agent controls prohibited regions. The information collected by the agent is sent to a trace sink.

If the self-hosted trace extensions are implemented, the PE Trace Unit must implement the system register interface. The PE Trace Unit and the PE must have the same view of the debug authentication interface. If ARMv8.4-Trace is implemented, `ExternalNoninvasiveDebugEnabled`() is always TRUE.

G3.1.1 Trace Sinks

The PE Trace Unit sends the trace data to a trace sink. A system might include multiple trace sinks, and allow software to configure which trace sink or sinks are used.

An example of an internal trace sink is an Embedded Trace Router (ETR), which allows software to define a buffer in memory. Trace data is written to this buffer.

Arm recommends that a system that includes ARMv8.4-Trace incorporates an ETR, and follows the system architecture described by the *CoreSight Base System Architecture (CS-BSA)*.

The self-hosted trace extensions do not describe the programmers' model trace sinks.

G3.1.2 Register controls to enable self-hosted trace

For EL1 using AArch64, see Chapter D3 *AArch64 Self-hosted Trace*.

If ARMv8.4-Trace is implemented, and external self-hosted trace is not implemented, self-hosted trace is always enabled.

If ARMv8.4-Trace is implemented, and external self-hosted trace is implemented, self-hosted trace is also enabled if one of the following is true:

- EDSCR.TFO == 0.
- EDSCR.TFO == 1, EL3 is implemented, SDCR.STE == 1 and `ExternalSecureNoninvasiveDebugEnabled`() == FALSE.
- EDSCR.TFO ==1, EL3 is not implemented, the PE executes in Secure state and `ExternalSecureNoninvasiveDebugEnabled`() = FALSE.

The pseudocode function `SelfHostedTraceEnabled`() shows these rules.

If ARMv8.4-Trace is not implemented, `SelfHostedTraceEnabled`() returns FALSE.

While `SelfHostedTraceEnabled`() == FALSE, `ExternalSecureNoninvasiveDebugEnabled`() and `ExternalNoninvasiveDebugEnabled`() control whether external tracing is prohibited or allowed in each Security state.

The self-hosted trace extensions do not provide any mechanism to control software access to the PE Trace Unit external debug interface.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G3.2 Prohibited regions in self-hosted trace

Trace is not generated in prohibited regions. The pseudocode function `TraceAllowed`() indicates whether tracing is allowed in the current Security state and Exception Level.

The IMPLEMENTATION DEFINED debug authentication interface can allow an external agent to disable the self-hosted trace extension.

If `SelfHostedTraceEnabled`() == TRUE, tracing is prohibited in Secure state when SDCR.STE == 0. If ARMv8.4-Trace is implemented but not enabled, tracing is prohibited in Secure state when `ExternalSecureNoninvasiveDebugEnabled`() == FALSE.

G3.2.1 Controls to prohibit trace at Exception levels

If `SelfHostedTraceEnabled`() == TRUE, TRFCR, TRFCR_EL1, TRFCR_EL2 and HTRFCR control whether trace is prohibited at an Exception level. While `SelfHostedTraceEnabled`() == FALSE, these registers are ignored.

If `SelfHostedTraceEnabled`() == TRUE, tracing is prohibited at EL0 if one of the following is true:

- The Effective value of HCR_EL2.TGE == 0 and TRFCR_EL1.E0TRE == 0.
- The Effective value of HCR.TGE == 0 and TRFCR.E0TRE == 0.
- The Effective value of HCR_EL2.TGE == 1 and TRFCR_EL2.E0HTRE == 0.

If `SelfHostedTraceEnabled`() == TRUE, tracing is prohibited at EL1 if TRFCR.E1TRE == 0.

If `SelfHostedTraceEnabled`() == TRUE, tracing is prohibited at EL2 if HTRFCR.E2TRE == 0.

If `SelfHostedTraceEnabled`() == TRUE, tracing is prohibited at EL3 if one of the following is true:

- EL3 is in AArch64 state.
- EL3 is in AArch32 state and TRFCR.E1TRE == 0.

The pseudocode `TraceAllowed`() shows the preceding rules.

If `SelfHostedTraceEnabled`() == TRUE, Table G3-1 shows when export of PMU events Attributable to an Exception level is prohibited.

Table G3-1 Export of PMU events prohibited

HCR_EL2.TGE	Tracing prohibited in	Export of PMU events Attributable to this Exception level prohibited
1	EL0, EL2, EL3	EL0
x	EL0, EL1, EL2, EL3	EL0
x	EL1, EL2, EL3	EL1
x	EL2, EL3	EL2
x	EL3	EL3

G3.2.2 Self-hosted trace and address translation

A hypervisor can use HTRFCR.CX to control visibility of VTTBR.VMID.

If `SelfHostedTraceEnabled`() == TRUE, and HTRFCR.CX == 0, or if EL2 is not implemented:

- The value of VTTBR.VMID is not traced.
- Comparisons with VTTBR.VMID do not match and results of comparison are not exposed through the comparators.

The PE Trace Unit may either prohibit trace for these values, or may record a VTTBR.VMID value of zero in the trace.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G3.3 Self-hosted trace timestamps

For EL1 using AArch64, see Chapter D3 *AArch64 Self-hosted Trace*.

The trace timestamp is a value that represents the passage of time in real-time. It is calculated from a counter which increments all the time, when the PE is generating trace and when the PE is in a prohibited region.

While `SelfHostedTraceEnabled`() == FALSE, the external trace provides the trace timestamp. If the external trace is a standard CoreSight system, the relationship between CoreSight time and the Generic Timer counter is IMPLEMENTATION DEFINED.

When `SelfHostedTraceEnabled`() == TRUE, the trace time stamp is one of the following:

- The physical counter value CNTPCT_EL0 or CNTPCT.
- A virtual counter, which is calculated from the physical counter CNTPCT_EL0 minus an offset CNTVOFF_EL2, if EL2 is implemented and using AArch64.
- A virtual counter, which is calculated from the physical counter CNTPCT minus an offset CNTVOFF, if EL2 is implemented and using AArch32.
- If EL2 is not implemented, the value of the offset is zero.

The fields TRFCR_EL2.TS, TRFCR.TS and HTRFCR.TS control which counter is used for self-hosted trace.

The timestamp used for trace is shown in Table G3-2.

Table G3-2 Timestamp used for trace.

`SelfHostedTraceEnabled`()	TRFCR_EL2.TS or HTRFCR.TS	TRFCR_EL1.TS	Timestamp traced
FALSE	xx	xx	CoreSight time
TRUE	0b00	0b01	CNTPCT - CNTVOFF
	0b00	0b11	CNTPCT
	0b01	xx	CNTPCT - CNTVOFF or CNTPCT_EL0 - CNTVOFF_EL2
	0b11	xx	CNTPCT or CNTPCT_EL0

Note

The value of HCR_EL2.E2H does not affect the counter used for the trace timestamp.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G3.4 Synchronization in self-hosted trace

The PE Trace Unit is an indirect observer of the trace control registers.

While `SelfHostedTraceEnabled`() == TRUE, indirect reads of the trace filter control fields, TRFCR.{E1TRE, E0TRE} and HTRFCR.{E2TRE, E0HTRE} are treated as indirect reads made by the instruction being traced, and are subject to the standard requirements for synchronization of System register accesses.

The `TSB CSYNC` operation is used to ensure that a trace operation, due to a PE Trace Unit generating trace for an instruction has completed. The `TSB CSYNC` operation may be reordered with respect to other instructions, so must be combined with at least one context synchronization event to ensure the operations are executed in the required order. This means that a direct write to TRFCR or HTRFCR is guaranteed to be observed by the PE Trace Unit only after a subsequent Context synchronization event. For more information, see *Trace Synchronization Barrier (TSB CSYNC)* on page E2-3995.

While `SelfHostedTraceEnabled`() == FALSE, the PE Trace Unit might impose stronger synchronization requirements.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter G4
The AArch32 System Level Memory Model

This chapter provides a system level view of the general features of the memory system. It contains the following sections:

- *About the memory system architecture* on page G4-5886.
- *Address space* on page G4-5887.
- *Mixed-endian support* on page G4-5888.
- *AArch32 cache and branch predictor support* on page G4-5889.
- *System register support for IMPLEMENTATION DEFINED memory features* on page G4-5913.
- *External aborts* on page G4-5914.
- *Memory barrier instructions* on page G4-5916.
- *Pseudocode description of general memory System instructions* on page G4-5917.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G4.1 About the memory system architecture

The Arm architecture supports different implementation choices for the memory system microarchitecture and memory hierarchy, depending on the requirements of the system being implemented. In this respect, the memory system architecture describes a design space in which an implementation is made. The architecture does not prescribe a particular form for the memory systems. Key concepts are abstracted in a way that permits implementation choices to be made while enabling the development of common software routines that do not have to be specific to a particular microarchitectural form of the memory system. For more information about the concept of a hierarchical memory system, see *Memory hierarchy* on page E2-3999.

G4.1.1 Form of the memory system architecture

The Armv8 A-profile architecture includes a *Virtual Memory System Architecture* (VMSA). Chapter G5 *The AArch32 Virtual Memory System Architecture* describes the AArch32 view of the VMSA.

G4.1.2 Memory attributes

Memory types and attributes on page E2-4010 describes the memory attributes, including how different memory types have different attributes. Each location in memory has a set of memory attributes, and the translation tables define the virtual memory locations, and the attributes for each location.

Table G4-1 shows the memory attributes that are visible at the system level.

Table G4-1 Memory attribute summary

Memory type	Shareability	Cacheability
Device[a]	Outer Shareable	Non-cacheable.
Normal	One of: • Non-shareable. • Inner Shareable. • Outer Shareable.	One of[b]: • Non-cacheable. • Write-Through Cacheable. • Write-Back Cacheable.

a. Takes additional attributes, see *Device memory* on page E2-4014.

b. See also *Cacheability, cache allocation hints, and cache transient hints* on page G4-5892.

For more information on Cacheability and Shareability, see *The Cacheability and Shareability memory attributes* on page E2-4000, *Non-shareable Normal memory* on page E2-4012, and *Caches and memory hierarchy* on page E2-3999.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G4.2 Address space

The Armv8 architecture is designed to support a wide range of applications with different memory requirements. It supports a range of *physical address* (PA) sizes, and provides associated control and identification mechanisms. For more information, see *About VMSAv8-32* on page G5-5922.

G4.2.1 Address space overflow or underflow

This subsection describes address space overflow or underflow:

Instruction address space overflow

When a PE performs a normal, sequential execution of instructions, it calculates:

```
(address_of_current_instruction) + (size_of_executed_instruction)
```

This calculation is performed after each instruction to determine which instruction to execute next.

If the address calculation performed after executing an A32 or T32 instruction overflows `0xFFFF FFFF`, the program counter becomes UNKNOWN.

If the PE executes an instruction for which the instruction address, size, and alignment mean that it contains the bytes `0xFFFFFFFF` and `0x00000000`, the bytes that apparently from `0x00000000` onwards come from an UNKNOWN address.

Data address space overflow and underflow

If the PE executes a load or store instruction for which the computed address, total access size, and alignment mean that it accesses bytes `0xFFFFFFFF` and `0x00000000`, then the bytes that apparently come from `0x00000000` onwards come from UNKNOWN addresses.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G4.3 Mixed-endian support

Table G4-2 shows the endianness of explicit data accesses and translation table walks.

Table G4-2 Endianness support

Exception level	Explicit data accesses	Stage 1 translation table walks	Stage 2 translation table walks
EL0	PSTATE.E	SCTLR(S/NS).EE	HSCTLR.EE
EL1	PSTATE.E	SCTLR(S/NS).EE	HSCTLR.EE
EL2	PSTATE.E	HSCTLR.EE	N/A
EL3	PSTATE.E	SCTLR(S).EE	N/A

AArch32 state provides the following options for endianness support:

- All Exception levels support mixed-endianness:
  - SCTLR(S/NS).EE, HSCTLR.EE, and PSTATE.E are RW.
- Only EL0 supports mixed-endianness and EL1, EL2, and EL3 support only little-endianness:
  - SCTLR(S/NS).EE and HSCTLR.EE are RES0. PSTATE.E is RW when in EL0 and RES0 when in EL1, EL2, or EL3. SPSR.E is also RES0 when not returning to EL0.
- Only EL0 supports mixed-endianness and EL1, EL2, and EL3 support only big-endianness:
  - SCTLR(S/NS).EE and HSCTLR.EE are RES1. PSTATE.E is RW when in EL0 and RES1 when in EL1, EL2, or EL3. SPSR.E is also RES1 when not returning to EL0.
- All Exception levels support only little-endianness:
  - Each of SCTLR(S/NS).EE, HSCTLR.EE, PSTATE.E, and SPSR.E is RES0.
- All Exception levels support only big-endianness:
  - Each of SCTLR(S/NS).EE, HSCTLR.EE, PSTATE.E, and SPSR.E is RES1.

If mixed endian support is implemented for an Exception level using AArch32, endianness is controlled by PSTATE.E. For exception returns to AArch32 state, PSTATE.E is copied from SPSR_ELx.E. If the target Exception level supports only little-endian accesses, SPSR_ELx.E is RES0. If the target Exception level supports only big-endian accesses, SPSR_ELx.E is RES1.

Note

- When using AArch32, Arm deprecates PSTATE.E having a different value from the equivalent System register EE bit when in EL1, EL2 or EL3. The use of the `SETEND` instruction is also deprecated.
- If the higher Exception levels are using AArch64, the corresponding registers are:
  - SCTLR_EL1 for SCTLR(NS).
  - SCTLR_EL2 for HSCTLR.
  - SCTLR_EL3 for SCTLR(S).

The `BigEndian()` function determines whether the current Exception level and Execution state is using big-endian data.

For more information about endianness in the Arm architecture, see *Endian support* on page E2-4006.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G4.4 AArch32 cache and branch predictor support

The following sections describe the support for caches and branch predictors in AArch32 state:

- *General behavior of the caches*.
- *Cache identification* on page G4-5890.
- *Cacheability, cache allocation hints, and cache transient hints* on page G4-5892.
- *Enabling and disabling the caching of memory accesses in AArch32 state* on page G4-5893.
- *Behavior of caches at reset* on page G4-5895.
- *About cache maintenance in AArch32 state* on page G4-5895.
- *AArch32 cache and branch predictor maintenance instructions* on page G4-5899.
- *Execution and data prediction restriction System instructions* on page G4-5910.
- *Cache lockdown* on page G4-5911.
- *System level caches* on page G4-5912.

See also Chapter G5 *The AArch32 Virtual Memory System Architecture*, and in particular *Caches in VMSAv8-32* on page G5-6009.

Note

- Branch predictors typically use a form of cache to hold branch target data. Therefore, they are included in this section.
- In the instruction mnemonics, MVA is a synonym for VA.

G4.4.1 General behavior of the caches

When a memory location is marked with a Normal Cacheable memory attribute, determining whether a copy of the memory location is held in a cache still depends on many aspects of the implementation. The following non-exhaustive list of factors might be involved:

- The size, line length, and associativity of the cache.
- The cache allocation algorithm.
- Activity by other elements of the system that can access the memory.
- Speculative instruction fetching algorithms.
- Speculative data fetching algorithms.
- Interrupt behaviors.

Given this range of factors, and the large variety of cache systems that might be implemented, the architecture cannot guarantee whether:

- A memory location present in the cache remains in the cache.
- A memory location not present in the cache is brought into the cache.

Instead, the following principles apply to the behavior of caches:

- The architecture has a concept of an entry locked down in the cache. How lockdown is achieved is IMPLEMENTATION DEFINED, and lockdown might not be supported by:
  - A particular implementation.
  - Some memory attributes.
- An unlocked entry in a cache might not remain in that cache. The architecture does not guarantee that an unlocked cache entry remains in the cache or remains incoherent with the rest of memory. Software must not assume that an unlocked item that remains in the cache remains dirty.
- A locked entry in a cache is guaranteed to remain in that cache. The architecture does not guarantee that a locked cache entry remains incoherent with the rest of memory, that is, it might not remain dirty.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

For more information, see *The interaction of cache lockdown with cache maintenance instructions* on page G4-5911.

- Any memory location that has a Normal Cacheable attribute at either the current Exception level or at a higher Exception level can be allocated to a cache at any time.
- It is guaranteed that no memory location that does not have a Normal Cacheable attribute is allocated into the cache.
- It is guaranteed that no memory location is allocated to the cache if it has a Normal Non-cacheable attribute or any type of Device memory attribute in both:
  - The translation regime at the current Exception level.
  - The translation regime at any higher Exception level.
- For data accesses, any memory location with a Normal Inner Shareable or Normal Outer Shareable attribute is guaranteed to be coherent with all masters in its Shareability domain.
- Any memory location is not guaranteed to remain incoherent with the rest of memory.
- The eviction of a cache entry from a cache level can overwrite memory that has been written by another observer only if the entry contains a memory location that has been written to by an observer in the Shareability domain of that memory location. The maximum size of the memory that can be overwritten is called the *Cache Write-back Granule*. In some implementations the CTR identifies the Cache Write-back Granule.
- The allocation of a memory location into a cache cannot cause the most recent value of that memory location to become invisible to an observer, if it was previously visible to that observer.

Note

The Cacheability attribute of an address is determined by the applicable translation table entry for that address, as modified by any applicable System register Cacheability controls, such as the SCTLR.{I, C} controls.

For the purpose of these principles, a cache entry covers at least 16 bytes and no more than 2KB of contiguous address space, aligned to the size of the cache entry.

G4.4.2 Cache identification

The Armv8 cache identification consists of a set of registers that describe the implemented caches that are affected by cache maintenance instructions executed on the PE. This includes cache maintenance instructions that:

- Affect the entire cache, for example ICIALLUIS.
- Operate by VA, for example ICIMVAU.
- Operate by set/way, for example DCISW.

The cache identification registers are:

- A single Cache Type Register, CTR, that defines:
  - The minimum line length of any of the instruction caches affected by the instruction cache maintenance instructions.
  - The minimum line length of any of the data or unified caches, affected by the data cache maintenance instructions.
  - The cache indexing and tagging policy of the Level 1 instruction cache.

Note

It is IMPLEMENTATION DEFINED whether caches beyond the PoC will be reported by this mechanism, and because of the possible existence of system caches some caches before the PoC might not be reported. For more information about system caches see *System level caches* on page G4-5912.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- A single Cache Level ID Register, CLIDR, that defines:
  - The type of cache that is implemented and can be maintained using the architected cache maintenance instructions that operate by set/way or operate on the entire cache at each cache level, up to the maximum of seven levels.
  - The *Level of Unification Inner Shareable* (LoUIS), *Level of Coherence* (LoC) and the *Level of Unification* (LoU) for the caches. See *Terms used in describing the cache maintenance instructions* on page G4-5896 for a definition of these terms.
  - An optional ICB field to indicate the boundary between the caches use for caching Inner Cacheable memory regions and those used only for caching Outer Cacheable regions.
- A single Cache Size Selection Register, CSSELR, that selects the cache level and cache type of the current Cache Size Identification Register.
- For each implemented cache that is identifiable by this mechanism, across all the levels of caching, a Cache Size Identification Register, that defines:
  - Whether the cache supports Write-Through, Write-Back, Read-Allocate and Write-Allocate.
  - The number of sets, associativity, and line length of the cache. See *Terms used in describing the cache maintenance instructions* on page G4-5896 for a definition of these terms.

  Note

  From Armv8.3, it is possible to have multiple Cache Size Identification Registers. For more details, see *Possible formats of the Cache Size Identification Registers, CCSIDR and CCSIDR2*.

To determine the cache topology associated with a PE:

1. Read the Cache Type Register to find the indexing and tagging policy used for the Level 1 instruction cache. This register also provides the size of the smallest cache lines used for the instruction caches, and for the data and unified caches. These values are used in cache maintenance instructions.
2. Read the Cache Level ID Register to find what caches are implemented. The register includes seven Cache type fields, for cache levels 1 to 7. Scanning these fields, starting from Level 1, identifies the instruction, data or unified caches implemented at each level. This scan ends when it reaches a level at which no caches are defined. The Cache Level ID Register also specifies the Level of Unification (LoU) and the Level of Coherence (LoC) for the cache implementation.
3. For each cache identified at stage 2:
   - Write to the Cache Size Selection Register to select the required cache. A cache is identified by its level, and whether it is:
     - An instruction cache.
     - A data or unified cache.
   - Read the Cache Size Identification Register to find details of the cache.

Possible formats of the Cache Size Identification Registers, CCSIDR and CCSIDR2

From Armv8.3, two different formats are available for defining the number of sets and associativity of the currently selected cache. For a definition of these terms, see *Terms used in describing the cache maintenance instructions* on page G4-5896.

When ARMv8.3-CCIDX is implemented:

- There are two Cache Size Identification Registers, CCSIDR and CCSIDR2.
- The length of the CCSIDR.Assoc field is 21 bits. This limits the associativity of the currently selected cache to 2^{21}.
- The length of the CCSIDR2.NumSets field is 24 bits. This limits the number of sets in the currently selected cache to 2^{24}.

This is the 64-bit format of the Cache Size Identification Register.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When ARMv8.3-CCIDX is not implemented:

- There is a single Cache Size Identification Register, CCSIDR.
- The length of the CCSIDR.Assoc field is 10 bits. This limits the associativity of the currently selected cache to 2^{10}.
- The length of the CCSIDR.NumSets field is 15 bits. This limits the number of sets in the currently selected cache to 2^{15}.

This is the 32-bit format of the Cache Size Identification Register.

When one of these formats is implemented, it is implemented across all the levels of caching.

G4.4.3 Cacheability, cache allocation hints, and cache transient hints

Cacheability only applies to Normal memory, and is defined independently for Inner and Outer cache locations. All types of Device memory are always treated as Non-cacheable.

As described in *Memory types and attributes* on page E2-4010, the memory attributes include a cacheability attribute that is one of:

- Non-cacheable.
- Write-Through cacheable.
- Write-Back cacheable.

In Armv8, Cacheability attributes other than Non-cacheable can be complemented by a *cache allocation hint*. This is an indication to the memory system of whether allocating a value to a cache is likely to improve performance. In addition, it is IMPLEMENTATION DEFINED whether a *cache transient hint* is supported, see *Transient cacheability hint*.

The cache allocation hints are assigned independently for read and write accesses, and therefore when the Transient hint is supported the following cache allocation hints can be used:

For read accesses: Read-Allocate, Transient Read-Allocate, or No Read-Allocate.

For write accesses: Write-Allocate, Transient Write-Allocate, or No Write-Allocate.

Note

- A Cacheable location with both No Read-Allocate and No Write-Allocate hints is not the same as a Non-cacheable location. A Non-cacheable location has coherency guarantees for all observers within the system that do not apply for a location that is Cacheable, No Read-Allocate, No Write-Allocate.
- Implementations can use the cache allocation hints to limit cache pollution to a part of a cache, such as to a subset of ways.
- For VMSAv8-32 translation table walks using the Long-descriptor translation table format, the appropriate TCR.{IRGN*n*, ORGN*n*} fields define the memory attributes of the translation tables, including the cacheability. However, this assignment supports only a subset of the cacheability attributes described in this section.

The architecture does not require an implementation to make any use of cache allocation hints. This means an implementation might not make any distinction between memory locations with attributes that differ only in their cache allocation hint.

Transient cacheability hint

In Armv8, it is IMPLEMENTATION DEFINED whether a Transient hint is supported for the VMSAv8-32 translation scheme when using the Long-descriptor translation table format. In an implementation that supports the Transient hint, the Transient hint is a qualifier of the cache allocation hints, and indicates that the benefit of caching is for a relatively short period. It indicates that it might be better to restrict allocation of transient entries, to avoid possibly casting-out other, less transient, entries.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

The architecture does not specify what is meant by a *relatively short period*.

When using the Short-descriptor translation table format, VMSAv8-32 cannot support the Transient hint.

The description of the MAIR0, MAIR1, HMAIR0, and HMAIR1 registers includes the assignment of the Transient attribute in an implementation that supports this option. In this assignment:

- The Transient hint is defined independently for Inner Cacheable and Outer Cacheable memory regions.
- A single Transient hint applies to both read and write accesses to a memory region.

G4.4.4 Enabling and disabling the caching of memory accesses in AArch32 state

In Armv8, Cacheability control fields can force all memory locations with the Normal memory type to be treated as Non-cacheable, regardless of their assigned Cacheability attribute. Independent controls are provided for each stage of address translation, with separate controls for:

- Data accesses. These controls also apply to accesses to the translation tables.
- Instruction accesses.

Note

These Cacheability controls replace the cache enable controls provided in previous versions of the Arm architecture.

In AArch32 state, the Cacheability control fields and their effects are as follows:

For the Non-secure PL1&0 translation regime

The Non-secure instance of SCTLR holds the EL1 controls that affect cacheability:

- When the value of SCTLR.C is 0:
  - All stage 1 translations for data accesses to Normal memory are Non-cacheable.
  - All accesses to the PL1&0 stage 1 translation tables are Non-cacheable.
- When the value of SCTLR.I is 0:
  - All stage 1 translations for instruction accesses to Normal memory are Non-cacheable.
- When the value of HCR2.CD is 1:
  - All stage 2 translations for data accesses to Normal memory are Non-cacheable.
  - All accesses to the PL1&0 stage 2 translation tables are Non-cacheable.
- When the value of HCR2.ID is 1:
  - All stage 2 translations for instruction accesses to Normal memory are Non-cacheable.
- When the value of HCR.DC is 1, all Non-secure stage 1 translations and all accesses to the Non-secure EL1&0 stage 1 translation tables, are treated as accesses to Normal Non-shareable Inner Write-Back Cacheable Read-Allocate Write-Allocate, Outer Write-Back Cacheable Read-Allocate Write-Allocate memory, regardless of the value of SCTLR.C. This applies to translations for both data and instruction accesses.

In addition, when the value of SCTLR.M is 0, indicating that the stage 1 translations are disabled for the translation regime, then if EL2 is using AArch32 and the value of HCR.DC is 0 or if EL2 is using AArch64 and the value of HCR_EL2.DC is 0, then:

- If the value of SCTLR.I is 0, instruction accesses to Normal memory from stage 1 of the translation regime are Outer Shareable, Inner Non-cacheable, Outer Non-cacheable.
- If the value of SCTLR.I is 1, instruction accesses to Normal memory from stage 1 of the translation regime are Outer Shareable, Inner Write-Through cacheable, Outer Write-Through cacheable.

Note

- In Non-secure state, the stage 1 and stage 2 cacheability attributes are combined as described in *Combining the Cacheability attribute* on page G5-5989.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The Non-secure SCTLR.{C, I} and HCR.DC fields have no effect on the Secure PL1&0 and EL2 translation regimes.
- The HCR2.{ID, CD} fields affect only stage 2 of the Non-secure PL1&0 translation regime.
- In Non-secure state, the PL1&0 translation regime can be described as the Non-secure EL1&0 translation regime. This is consistent with the equivalent AArch64 descriptions.

For the Secure PL1&0 translation regime

The Secure instance of SCTLR holds the controls that determine cacheability:

- When the value of SCTLR.C is 0:
  - All data accesses to Normal memory using the Secure PL1&0 translation regime are Non-cacheable.
  - All accesses to the Secure PL1&0 translation tables are Non-cacheable.
- When the value of SCTLR.I is 0:
  - All instruction accesses to Normal memory using the Secure PL1&0 translation regime are Non-cacheable.

In addition, when the value of SCTLR.M is 0, indicating that stage 1 translations are disabled, then:

- If the value of SCTLR.I is 0, instruction accesses to Normal memory from stage 1 of the translation regime are Outer Shareable, Inner Non-cacheable, Outer Non-cacheable.
- If the value of SCTLR.I is 1, instruction accesses to Normal memory from stage 1 of the translation regime are Outer Shareable, Inner Write-Through cacheable, Outer Write-Through cacheable.

Note

The Secure SCTLR.{I, C, M} fields have no effect on the Non-secure PL1&0 and EL2 translation regimes.

For the EL2 translation regime

- When the value of HSCTLR.C is 0:
  - All data accesses to Normal memory using the EL2 translation regime are Non-cacheable.
  - All accesses to the EL2 translation tables are Non-cacheable.
- When the value of HSCTLR.I is 0:
  - All instruction accesses to Normal memory using the EL2 translation regime are Non-cacheable.

In addition, when the value of HSCTLR.M is 0, indicating that stage 1 translations are disabled, then:

- If the value of HSCTLR.I is 0, instruction accesses to Normal memory from stage 1 of the translation regime are Outer Shareable, Inner Non-cacheable, Outer Non-cacheable.
- If the value of HSCTLR.I is 1, instruction accesses to Normal memory from stage 1 of the translation regime are Outer Shareable, Inner Write-Through cacheable, Outer Write-Through cacheable.

Note

The HSCTLR.{I. C, M} fields have no effect on the PL1&0 and EL3 translation regimes.

The effect of the SCTLR.C or HSCTLR.C and HCR2.CD bits is reflected in the result of the address translation instructions in the PAR.

Note

- The requirements in this section mean the architecturally required effects of SCTLR.I and HSCTLR.I are limited to their effects on caching instruction accesses in unified caches.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- This specification can give rise to different cacheability attributes between instruction and data accesses to the same location. Where this occurs, the measures for mismatch memory attributes described in *Mismatched memory attributes* on page E2-4020 must be followed to manage the corresponding loss of coherency.

G4.4.5 Behavior of caches at reset

In Armv8:

- All caches reset to IMPLEMENTATION DEFINED states that might be UNKNOWN.
- The Cacheability control fields described in *Enabling and disabling the caching of memory accesses in AArch32 state* on page G4-5893 reset to values that force all memory locations to be treated as Non-cacheable.

  Note

  This applies only to the controls that apply to the Translation regime that is used by the Exception level, PE mode, and Security state entered on reset.

- An implementation can require the use of a specific cache initialization routine to invalidate its storage array before caching is enabled. The exact form of any required initialization routine is IMPLEMENTATION DEFINED, and the routine must be documented clearly as part of the documentation of the device.
- If an implementation permits cache hits when the Cacheability control fields force all memory locations to be treated as Non-cacheable then the cache initialization routine must:
  - Provide a mechanism to ensure the correct initialization of the caches.
  - Be documented clearly as part of the documentation of the device.

  In particular, if an implementation permits cache hits when the Cacheability controls force all memory locations to be treated as Non-cacheable, and the cache contents are not invalidated at reset, the initialization routine must avoid any possibility of running from an uninitialized cache. It is acceptable for an initialization routine to require a fixed instruction sequence to be placed in a restricted range of memory.
- Arm recommends that whenever an invalidation routine is required, it is based on the Armv8 cache maintenance instructions.

Similar rules apply to:

- Branch predictor behavior, see *Behavior of the branch predictors at reset* on page G4-5903.
- TLB behavior, see *TLB behavior at reset* on page G5-5992.

G4.4.6 About cache maintenance in AArch32 state

The following sections give general information about cache maintenance in Armv8:

- *Terms used in describing the cache maintenance instructions* on page G4-5896.
- *The Armv8 abstraction of the cache hierarchy* on page G4-5898.

The following sections describe the AArch32 state cache maintenance instructions:

- *AArch32 instruction cache maintenance instructions (IC*)* on page G4-5900.
- *AArch32 data cache maintenance instructions (DC*)* on page G4-5901.

Note

Some descriptions of the cache maintenance instructions refer to the Cacheability of the address on which the instruction operates. The Cacheability of an address is determined by the applicable translation table entry for that address, as modified by any applicable System register Cacheability controls, such as the SCTLR.{I, C} controls.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Terms used in describing the cache maintenance instructions

Cache maintenance instructions are defined to act on particular memory locations. Instructions can be defined:

- By the virtual address of the memory location to be maintained, referred to as operating *by VA*.
- By a mechanism that describes the location in the hardware of the cache, referred to as operating *by set/way*.

In addition, for instruction caches and branch predictors, there are instructions that invalidate all entries.

The following subsections define the terms used in the descriptions of the cache maintenance instructions:

- *Terminology for cache maintenance instructions operating by set/way*.
- *Terminology for Clean, Invalidate, and Clean and Invalidate instructions*.

Note

There is no terminology specific to cache maintenance instructions that operate by VA. When all applicable stages of translation are disabled, the VA used is identical to the PA. For more information about memory system behavior when address translation is disabled, see *The effects of disabling address translation stages on VMSAv8-32 behavior* on page G5-5930.

Terminology for cache maintenance instructions operating by set/way

Cache maintenance instruction that operate by set/way refer to the particular structures in a cache. Three parameters describe the location in a cache hierarchy that an instruction works on. These parameters are:

Level The cache level of the hierarchy. The number of levels of cache is IMPLEMENTATION DEFINED. The cache levels that can be managed using the architected cache maintenance instructions that operate by set/way can be determined from the CLIDR.

In the Arm architecture, the lower numbered cache levels are those closest to the PE. See *Memory hierarchy* on page E2-3999.

Set Each level of a cache is split up into a number of *sets*. Each set is a set of locations in a cache level to which an address can be assigned. Usually, the set number is an IMPLEMENTATION DEFINED function of an address.

In the Arm architecture, sets are numbered from 0.

Way The associativity of a cache is the number of locations in a set to which a specific address can be assigned. The *way* number specifies one of these locations.

In the Arm architecture, ways are numbered from 0.

Note

Because the allocation of a memory address to a cache location is entirely IMPLEMENTATION DEFINED, Arm expects that most portable software will use only the cache maintenance instructions by set/way as single steps in a routine to perform maintenance on the entire cache.

Terminology for Clean, Invalidate, and Clean and Invalidate instructions

Caches introduce coherency problems in two possible directions:

1. An update to a memory location by a PE that accesses a cache might not be visible to other observers that can access memory. This can occur because new updates are still in the cache and are not visible yet to the other observers that do not access that cache.

2. Updates to memory locations by other observers that can access memory might not be visible to a PE that accesses a cache. This can occur when the cache contains an old, or *stale*, copy of the memory location that has been updated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The *Clean* and *Invalidate* instructions address these two issues. The definitions of these instructions are:

Clean

A cache clean instruction ensures that updates made by an observer that controls the cache are made visible to other observers that can access memory at the point to which the instruction is performed. Once the Clean has completed, the new memory values are guaranteed to be visible to the point to which the instruction is performed, for example to the Point of Unification.

The cleaning of a cache entry from a cache can overwrite memory that has been written by another observer only if the entry contains a location that has been written to by an observer in the Shareability domain of that memory location.

Invalidate

A cache invalidate instruction ensures that updates made visible by observers that access memory at the point to which the invalidate is defined, are made visible to an observer that controls the cache. This might result in the loss of updates to the locations affected by the invalidate instruction that have been written by observers that access the cache, if those updates have not been cleaned from the cache since they were made.

If the address of an entry on which the invalidate instruction operates is Normal, Non-cacheable or any type of Device memory then an invalidate instruction also ensures that this address is not present in the cache.

Note

Entries for addresses that are Normal Cacheable can be allocated to the cache at any time, and so the cache invalidate instruction cannot ensure that the address is not present in a cache.

Clean and Invalidate

A cache *clean and invalidate* instruction behaves as the execution of a clean instruction followed immediately by an invalidate instruction. Both instructions are performed to the same location.

The points to which a cache maintenance instruction can be defined differ depending on whether the instruction operates by VA or by set/way:

- For instructions operating by set/way, the point is defined to be to the next level of caching. For the All operations, the point is defined as the Point of Unification for each location held in the cache.
- For instruction operating by VA, two conceptual points are defined:

  Point of Coherency (PoC)

  The point at which all agents that can access memory are guaranteed to see the same copy of a memory location for accesses of any memory type or cacheability attribute. In many cases this is effectively the main system memory, although the architecture does not prohibit the implementation of caches beyond the PoC that have no effect on the coherency between memory system agents.

  Note

  The presence of system caches can affect the determination of the point of coherency as described in *System level caches* on page G4-5912.

  Point of Unification (PoU)

  The PoU for a PE is the point by which the instruction and data caches and the translation table walks of that PE are guaranteed to see the same copy of a memory location. In many cases, the Point of Unification is the point in a uniprocessor memory system by which the instruction and data caches and the translation table walks have merged.

  The PoU for an Inner Shareable Shareability domain is the point by which the instruction and data caches and the translation table walks of all the PEs in that Inner Shareable Shareability domain are guaranteed to see the same copy of a memory location. Defining this point permits self-modifying software to ensure future instruction fetches are associated with the modified version of the software by using the standard correctness policy of:

  1. Clean data cache entry by address.
  2. Invalidate instruction cache entry by address.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following fields in the CLIDR relate to these conceptual points:

LoC, Level of Coherence

This field defines the last level of cache that must be cleaned or invalidated when cleaning or invalidating to the Point of Coherency. The LoC value is a cache level, so, for example, if LoC contains the value 3:

- A clean to the Point of Coherency operation requires the level 1, level 2 and level 3 caches to be cleaned.
- Level 4 cache is the first level that does not have to be maintained.

If the LoC field value is `0x0`, this means that no levels of cache need to cleaned or invalidated when cleaning or invalidating to the Point of Coherency.

If the LoC field value is a nonzero value that corresponds to a level that is not implemented, this indicates that all implemented caches are before the Point of Coherency.

LoUU, Level of Unification, uniprocessor

This field defines the last level of cache that must be cleaned or invalidated when cleaning or invalidating to the Point of Unification for the PE. As with LoC, the LoUU value is a cache level.

If the LoUU field value is `0x0`, this means that no levels of cache need to cleaned or invalidated when cleaning or invalidating to the Point of Unification.

If the LoUU field value is a nonzero value that corresponds to a level that is not implemented, this indicates that all implemented caches are before the Point of Unification.

LoUIS, Level of Unification, Inner Shareable

In any implementation:

- This field defines the last level of cache that must be cleaned or invalidated when cleaning or invalidating to the Point of Unification for the Inner Shareable Shareability domain. As with LoC, the LoUIS value is a cache level.
- If the LoUIS field value is `0x0`, this means that no levels of cache need to cleaned or invalidated when cleaning or invalidating to the Point of Unification for the Inner Shareable Shareability domain.
- If the LoUIS field value is a nonzero value that corresponds to a level that is not implemented, this indicates that all implemented caches are before the Point of Unification.

For more information, see the CLIDR description.

The Armv8 abstraction of the cache hierarchy

The following subsections describe the Armv8 abstraction of the cache hierarchy:

- *Cache maintenance instructions that operate by VA*.
- *Cache maintenance instructions that operate by set/way* on page G4-5899.

Cache maintenance instructions that operate by VA

The VA-based cache maintenance instructions are described as operating by VA. Each of these instructions is always qualified as being either:

- Performed to the Point of Coherency.
- Performed to the Point of Unification.

See *Terms used in describing the cache maintenance instructions* on page G4-5896 for definitions of Point of Coherency and Point of Unification, and more information about possible meanings of VA.

AArch32 cache and branch predictor maintenance instructions on page G4-5899 lists the VA-based maintenance instructions.

The CTR holds minimum line length values for:

- The instruction caches.
- The data and unified caches.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

These values support efficient invalidation of a range of addresses, because this value is the most efficient address stride to use to apply a sequence of VA-based maintenance instructions to a range of VAs.

For the Invalidate data or unified cache line by VA instruction, the Cache Write-back Granule field of the CTR defines the maximum granule that a single invalidate instruction can invalidate. This meaning of the Cache Write-back Granule is in addition to its defining the maximum size that can be written back.

Cache maintenance instructions that operate by set/way

AArch32 cache and branch predictor maintenance instructions lists the set/way-based maintenance instructions. Some encodings of these instructions include a required field that specifies the cache level for the instruction:

- A clean instruction cleans from the level of cache specified through to at least the next level of cache, moving further from the PE.
- An invalidate instruction invalidates only at the level specified.

G4.4.7 AArch32 cache and branch predictor maintenance instructions

The instruction and data cache maintenance instructions have the same functionality in AArch32 state and in AArch64 state. Table G4-3 shows the AArch32 System instructions. Instructions that take an argument include Rt in the instruction description.

AArch32 state also provides branch predictor maintenance instructions.

Note

- In Table G4-3 the Point of Unification is the Point of Unification of the PE executing the cache maintenance instruction.
- In AArch32 state, all of the maintenance instructions are available from EL1 or higher.
- In AArch64 state, branch predictors are always invisible to software, and therefore AArch64 state does not provide any branch predictor maintenance instructions.

Table G4-3 AArch32 System instructions for cache maintenance

Register	Instruction
Instruction cache maintenance instructions	
ICIALLUIS	Invalidate all to Point of Unification, Inner Shareable
ICIALLU	Invalidate all to Point of Unification
ICIMVAU, Rt	Invalidate by virtual address to Point of Unification
Data cache maintenance instructions	
DCIMVAC, Rt	Invalidate by virtual address to Point of Coherency
DCISW, Rt	Invalidate by set/way
DCCMVAC, Rt	Clean by virtual address to Point of Coherency
DCCSW, Rt	Clean by set/way
DCCMVAU, Rt	Clean by virtual address to Point of Unification
DCCIMVAC, Rt	Clean and invalidate by virtual address to Point of Coherency
DCCISW, Rt	Clean and invalidate by set/way

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G4-3 AArch32 System instructions for cache maintenance (continued)

Register	Instruction
Branch prediction maintenance instructions	
BPIMVA, Rt	Invalidate the virtual address from the branch predictors
BPIALLIS, Rt	Invalidate all entries from branch predictors, Inner Shareable
BPIALL, Rt	Invalidate all entries from branch predictors

A DSB or DMB instruction intended to ensure the completion of cache or branch predictor maintenance instructions must have an access type of both loads and stores.

In an implementation where the branch predictors are architecturally invisible, the BPIMVA, BPIALLIS, and BPIALL instructions can execute as NOPs.

The following subsections give more information about these instructions:

- *AArch32 instruction cache maintenance instructions (IC*)*.
- *AArch32 data cache maintenance instructions (DC*)* on page G4-5901.
- *Branch predictors* on page G4-5901.
- *General requirements for the scope of cache and branch predictor maintenance instructions* on page G4-5903.
- *Effects of instructions that operate by VA to the Point of Coherency* on page G4-5903.
- *Effects of instructions that operate by VA but not to the Point of Coherency* on page G4-5904.
- *Effects of All and set/way maintenance instructions* on page G4-5904.
- *Effects of virtualization and security on the AArch32 cache maintenance instructions* on page G4-5905.
- *Boundary conditions for cache maintenance instructions* on page G4-5907.
- *Ordering of cache and branch predictor maintenance instructions* on page G4-5907.
- *Performing cache maintenance instructions* on page G4-5908.

AArch32 instruction cache maintenance instructions (IC*)

Where an address argument for these instructions is required, it takes the form of a 32-bit register that holds the virtual address argument. No alignment restrictions apply for this address.

Any cache maintenance instruction operating by VA includes as part of any required VA to PA translation:

- For an instruction executed at EL1, the current system ASID.
- The current Security state.
- Whether the instruction was performed from Hyp mode, or at EL1.
- For an instruction executed at EL1, the VMID.

That VA to PA translation might fault. However for an instruction cache maintenance instruction that operates by VA:

- It is IMPLEMENTATION DEFINED whether the operation can generate a Data Abort exception for a Translation fault or an Access flag fault.
- The operation cannot generate a Data Abort exception for a Domain fault or a Permission fault, except for the Permission fault case on a Stage 2 fault on a stage 1 translation table walk.

For more information about the possible faults on an instruction that operates by VA, see *Types of MMU faults* on page G5-6013.

An instruction cache maintenance instruction can complete at any time after it is executed, but is only guaranteed to be complete, and its effects visible to other observers, following a DSB instruction executed by the PE that executed the cache maintenance instruction. See also the completion requirements for cache and branch predictor maintenance instructions in *Completion and endpoint ordering* on page E2-3987.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

See also *Ordering of cache and branch predictor maintenance instructions* on page G4-5907.

AArch32 data cache maintenance instructions (DC*)

Data cache maintenance instructions that take a set/way/level argument take a 32-bit register.

If a data cache maintenance by set/way instruction specifies a set, way, or level argument that is larger than the value supported by the implementation then the instruction is CONSTRAINED UNPREDICTABLE, see *Out of range values of the Set/Way/Index fields in cache maintenance instructions* on page K1-7843 or the instruction description.

DCISW instructions executed at EL1 perform a clean and invalidate, meaning it performs the same maintenance as a DCCISW instruction, if all of the following apply:

- EL2 is implemented and enabled in the current Security state.
- Either:
  - EL2 is using AArch32 and the value of HCR.SWIO is 1.
  - EL2 is using AArch64 and the value of HCR_EL2.SWIO is 1.

Where an address argument for these instructions is required, it takes the form of a 32-bit register that holds the virtual address argument. No alignment restrictions apply for this address.

Any cache maintenance instruction operating by VA includes as part of any required VA to PA translation:

- For an instruction executed at EL1, the current system ASID.
- The current Security state.
- Whether the instruction was performed from Hyp mode, or from EL1.
- For an instruction executed from EL1, the VMID.

That VA to PA translation might fault. However a data or unified cache maintenance instruction that operates by VA cannot generate a Data Abort exception for a Domain fault, and cannot generate a Data Abort exception for a Permission fault, except for the Permission fault case on a Stage 2 fault on a stage 1 translation table walk.

For more information about the possible faults on an instruction that operates by VA, see *Types of MMU faults* on page G5-6013.

DCIMVAC and DCISW instructions executed at EL1 perform a clean and invalidate, meaning they perform the same maintenance as a DCCIMVAC or DCCISW instruction respectively, if all of the following apply:

- EL2 is implemented and enabled in the current Security state.
- PL1&0 stage two address translation is enabled, meaning either:
  - EL2 is using AArch32 and the value of HCR.VM is 1.
  - EL2 is using AArch64 and the value of HCR_EL2.VM is 1.

If a memory fault that sets FAR for the translation regime applicable for the cache maintenance instruction is generated from a data cache maintenance instruction, the FAR holds the address specified in the register argument of the instruction.

See also *Ordering of cache and branch predictor maintenance instructions* on page G4-5907.

Branch predictors

In AArch32 state it is IMPLEMENTATION DEFINED whether branch prediction is architecturally visible. This means that under some circumstances software must perform branch predictor maintenance to avoid incorrect execution caused by out-of-date entries in the branch predictor. For example, to ensure correct operation it might be necessary to invalidate branch predictor entries on a change to instruction memory, or a change of instruction address mapping. For more information, see *Specific requirements for branch predictor maintenance instructions* on page G4-5902.

In an implementation where the branch predictors are architecturally invisible, the branch predictor maintenance instructions can execute as `NOP`s.

An invalidate all operation on the branch predictor ensures that any location held in the branch predictor has no functional effect on execution. An invalidate branch predictor by VA instruction operates on the address of the branch instruction, but can affect other branch predictor entries.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

The architecture does not make visible the range of addresses in a branch predictor to which the invalidate operation applies. This means the address used in the invalidate by VA operation must be the address of the branch to be invalidated.

If branch prediction is architecturally visible, an instruction cache invalidate all operation also invalidates all branch predictors.

See also *Ordering of cache and branch predictor maintenance instructions* on page G4-5907.

Specific requirements for branch predictor maintenance instructions

If, for a given translation regime and a given ASID and VMID as appropriate, the instructions at any virtual address change, then branch predictor maintenance instructions must be performed to invalidate entries in the branch predictor, to ensure that the change is visible to subsequent execution. This maintenance is required when writing new values to instruction locations. It can also be required as a result of any of the following situations that change the translation of a virtual address to a physical address, if, as a result of the change to the translation, the instructions at the virtual addresses change:

- For any translation regime other than the Non-secure PL1&0 translation regime, enabling or disabling stage 1 translations.
- For the Non-secure PL1&0 translation regime:
  - When stage 2 translations are enabled, enabling or disabling stage 1 translations unless accompanied by a change of VMID.
  - When stage 2 translations are disabled, enabling or disabling stage 1 translations.
  - Enabling or disabling stage 2 translations.
- Writing new mappings to the translation tables.
- Any change to the TTBR0, TTBR1, or TTBCR registers, unless:
  - For a change to the Secure PL1&0 translation regime, the change is accompanied by a change to the ASID.
  - For a change to the stage 1 translations of the Non-secure PL1&0 translation regime, the change is accompanied by a change to the ASID or a change to the VMID.
- Any change to the VTTBR or VTCR registers, unless accompanied by a change to the VMID.

Note

Invalidation is not required if the changes to the translations are such that the instructions associated with the non-faulting translations of a virtual address, for a given translation regime and a given ASID and VMID, as appropriate, remain unchanged throughout the sequence of changes to the translations. Examples of translation changes to which this applies are:

- Changing a valid translation to a translation that generates an MMU fault.
- Changing a translation that generates an MMU fault to a valid translation.

Failure to invalidate entries might give CONSTRAINED UNPREDICTABLE results, caused by the execution of old branches. For more information, see *Ordering of cache and branch predictor maintenance instructions* on page G4-5907.

Note

- In Armv8, there is no requirement to use the branch predictor maintenance operations to invalidate the branch predictor after:
  - Changing the ContextID or VMID.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— A cache maintenance instruction that is identified as also flushing the branch predictors, see *AArch32 cache and branch predictor maintenance instructions* on page G4-5899.

Cache maintenance system instructions on page K14-8090 shows the branch predictor maintenance operations in a VMSA implementation.

Behavior of the branch predictors at reset

In AArch32 state:

- If branch predictors are not architecturally invisible:
  - The branch predictors reset to an IMPLEMENTATION DEFINED state that might be UNKNOWN.
  - The branch predictors are disabled at reset.
- An implementation can require the use of a specific branch predictor initialization routine to invalidate the branch predictor storage array before it is enabled. The exact form of any required initialization routine is IMPLEMENTATION DEFINED, but the routine must be documented clearly as part of the documentation of the device.
- Arm recommends that whenever an invalidation routine is required, it is based on the AArch32 branch predictor maintenance operations.

Similar rules apply:

- To cache behavior, see *Behavior of caches at reset* on page G4-5895.
- To TLB behavior, see *TLB behavior at reset* on page G5-5992.

General requirements for the scope of cache and branch predictor maintenance instructions

The Armv8 specification of the cache maintenance and branch predictor instructions describes what each instruction is guaranteed to do in a system. It does not limit other behaviors that might occur, provided they are consistent with the requirements described in *General behavior of the caches* on page G4-5889, *Behavior of caches at reset* on page G4-5895, and *Preloading caches* on page E2-4002.

This means that as a side-effect of a cache maintenance instruction:

- Any location in the cache might be cleaned.
- Any unlocked location in the cache might be cleaned and invalidated.

As a side-effect of a branch predictor maintenance instruction, any entry in the branch predictor might be invalidated.

Note

Arm recommends that, for best performance, such side-effects are kept to a minimum. Arm strongly recommends that the side-effects of operations performed in Non-secure state do not have a significant performance impact on execution in Secure state.

Effects of instructions that operate by VA to the Point of Coherency

For Normal memory that is not Inner Non-cacheable, Outer Non-cacheable, these instructions must affect the caches of other PEs in the Shareability domain described by the Shareability attributes of the VA supplied with the instruction.

For Device memory and Normal memory that is Inner Non-cacheable, Outer Non-cacheable, these instructions must affect the caches of all PEs in the Outer Shareable Shareability domain of the PE on which the instruction is operating.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In all cases, for any affected PE, these instructions affect all data and unified caches to the Point of Coherency.

Table G4-4 PEs affected by cache maintenance instructions to the Point of Coherency

Shareability	PEs affected	Effective to
Non-shareable	The PE performing the operation	The Point of Coherency of the entire system
Inner Shareable	All PEs in the same Inner Shareable Shareability domain as the PE performing the operation	The Point of Coherency of the entire system
Outer Shareable	All PEs in the same Outer Shareable Shareability domain as the PE performing the operation	The Point of Coherency of the entire system

Effects of instructions that operate by VA but not to the Point of Coherency

The following instruction operate by VA but not to the Point of Coherency:

- Clean data or unified cache line by MVA to the Point of Unification, DCCMVAU.
- Invalidate instruction cache line by MVA to Point of Unification, ICIMVAU.
- Invalidate by MVA from branch predictors, BPIMVA.

For these instructions, Table G4-5 shows how, for a VA in a Normal or Device memory location, the Shareability attribute of the VA determines the minimum set of PEs affected, and the point to which the instruction must be effective.

Table G4-5 PEs affected by cache maintenance instructions to the Point of Unification

Shareability	PEs affected	Effective to
Non-shareable	The PE executing the instruction	The Point of Unification of instruction cache fills, data cache fills and write-backs, and translation table walks, on the PE executing the instruction
Inner Shareable or Outer Shareable	All PEs in the same Inner Shareable Shareability domain as the PE executing the instruction	The Point of Unification of instruction cache fills, data cache fills and write-backs, and translation table walks, of all PEs in the same Inner Shareable Shareability domain as the PE executing the instruction

Note

The set of PEs guaranteed to be affected is never greater than the PEs in the Inner Shareable Shareability domain containing the PE executing the instruction.

Effects of All and set/way maintenance instructions

The ICIALLU, BPIALL and DC* set/way instructions apply only to the caches and branch predictors of the PE that performs the instruction. If the branch predictors are architecturally-visible, ICIALLU also performs a BPIALL operation.

The ICIALLUIS and BPIALLIS instructions can affect the caches and branch predictors of all PEs in the same Inner Shareable Shareability domain as the PE that performs the instruction. If the branch predictors are architecturally-visible, ICIALLUIS also performs a BPIALLIS operation. These instructions have an effect to the Point of Unification of instruction cache fills, data cache fills, and write-backs, and translation table walks, of all PEs in the same Inner Shareable Shareability domain.

Note

The possible presence of system caches, as described in *System level caches* on page G4-5912, means architecture does not guarantee that all levels of cache can be maintained using set/way instructions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Effects of virtualization and security on the AArch32 cache maintenance instructions

Each Security state has its own physical address space, and therefore cache entries are associated with physical address space. In addition, cache maintenance and branch predictor instructions performed in Non-secure state have to take account of:

- Whether the instruction was performed at EL1 or at EL2.
- For instructions that operate by VA, the current VMID.

Table G4-6 shows the effects of virtualization and security on these maintenance instructions.

Table G4-6 Effects of virtualization and security on the AArch32 cache maintenance instructions

Cache maintenance instructions	Security state	Specified entry
Data or unified cache maintenance instructions		
Invalidate, Clean, or Clean and Invalidate by VA: DCIMVAC, DCCMVAC, DCCMVAU, DCCIMVAC	Either	All lines that hold the PA that, in the current translation regime, are mapped to by the combination of all of: • The specified VA. • For an instruction executed at EL1, the current ASID if the location is mapped to by a non-global page. • For an instruction executed at EL1, the current VMID[a].
Invalidate, Clean, or Clean and Invalidate by set/way: DCISW, DCCSW, DCCISW	Non- secure	Line specified by set/way provided that the entry comes from the Non-secure PA space.
	Secure	Line specified by set/way regardless of the PA space that the entry has come from.
Instruction cache maintenance instructions		
Invalidate by VA: ICIMVAU	Either	All lines corresponding to the specified VA[b] in the current translation regime and: • For an instruction executed at EL1 or EL0, the current ASID. • For an instruction executed at EL1 or EL0, the current VMID[a].
Invalidate All: ICIALLU, ICIALLUIS		• Can invalidate any unlocked entry in the instruction cache. • Are required to invalidate any entries relevant to the software component that executed it. The Non-secure and Secure descriptions give more information: **Non-secure** An instruction executed at EL1 must operate on all instruction cache lines that contain entries associated with the current virtual machine, meaning any entry with the current VMID.[a] An instruction executed at EL2 must operate on all instruction cache lines that contain entries that can be accessed from Non-secure state. **Secure** The instruction must invalidate all instruction cache lines.
Branch predictor instructions[c]		
Invalidate by VA: BPIMVA	Either	All lines that, in the current translation regime, are mapped to by the combination of: all of: • The specified VA. • For an instruction executed at EL1 or EL0, the current ASID if the location is mapped to by a non-global page. • For an instruction executed at EL1 or EL0, the current VMID[a].
Invalidate all: BPIALL, BPIALLIS	• Can invalidate any unlocked entry in the branch predictor. • Are required to invalidate any entries relevant to the software component that executed it. The Non-secure and Secure descriptions give more information.	

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

a. Dependencies on the VMID apply even when either EL2 is using AArch32 and the value of HCR.VM is 0 or EL2 is using AArch64 when enabled for the current Security state, and the value of HCR_EL2.VM is 0. If the PE resets into an Exception level that is using AArch32, VTTBR.VMID resets to zero, meaning there is a valid VMID from reset. However, if the PE resets into an Exception level that is using AArch64, VTTBR_EL2.VMID resets to a value that is architecturally UNKNOWN, and the VTTBR_EL2.VMID field must be set to a known value, that might be zero, as part of the PE initialization sequence.

b. The type of instruction cache used affects the interpretation of the specified entries in this table such that:
 For a PIPT instruction cache, the cache maintenance applies to all entries whose physical address corresponds to the specified address.
 For a VIPT instruction cache, the cache maintenance applies to entries whose virtual index and physical tag corresponds to the specified address.

For information of types of instruction cache, see *Instruction caches* on page G5-6009.

c. In an implementation where the branch predictors are architecturally invisible, these instructions can execute as `NOP`s.

For locked entries and entries that might be locked, the behavior of cache maintenance instructions described in *The interaction of cache lockdown with cache maintenance instructions* on page G4-5911 applies.

With an implementation that generates aborts if entries are locked or might be locked in the cache, when the use of lockdown aborts is enabled, these aborts can occur on any cache maintenance instructions.

In an implementation that includes EL2:

- The architecture does not require cache cleaning when switching between virtual machines. Cache invalidation by set/way must not present an opportunity for one virtual machine to corrupt state associated with a second virtual machine. To ensure this requirement is met, EL1 invalidate by set/way instructions executed in at EL1 when HCR_EL2.VM or HCR.VM is 1 and EL2 is enabled can, instead, perform a clean and invalidate by set/way.

- The AArch32 Data cache invalidate instructions DCIMVAC and DCISW perform a cache clean as well as a cache invalidate, meaning DCIMVAC performs the same invalidation as a DCCIMVAC instruction, and DCISW performs the same invalidation as a DCCISW instruction, if both of the following apply:
  — EL2 is using AArch32, the value of HCR.VM is 1, and the instruction is executed at Non-secure EL1.
  — EL2 is using AArch64, the value of HCR_EL2.VM is 1, EL2 is enabled, and the instruction is executed at EL1.

- The AArch32 Data cache invalidate by set/way instruction DCISW performs a cache clean as well as a cache invalidate, meaning it performs the same invalidation as a DCCISW instruction, if either of the following apply:
  — EL2 is using AArch32, the value of HCR.SWIO is 1, and the instruction is executed at Non-secure ELl.
  — EL2 is using AArch64, the value of HCR_EL2.SWIO is 1, EL2 is enabled, and the instruction is executed at EL1.

- TLB and instruction cache invalidate instructions are broadcast across the Inner Shareable domain when either:
  — EL2 is using AArch32, the value of HCR.FB is 1, and execution is at Non-secure EL1.
  — EL2 is using AArch64, the value of HCR_EL2.FWB is 1, EL2 is enabled, and the instruction is executed at EL1.

  When EL1 is using AArch32, this applies to the TLBIMVA, TLBIASID, TLBIMVAA, TLBIMVAL, TLBIMVAAL, and ICIALLU instructions. This means the instruction performs the invalidation that would be performed by the corresponding Inner Shareable instruction, for example ICIALLU performs the invalidation that would be performed by ICIALLUIS, and BPIALL performs the invalidation that would be performed by BPIALLIS.

For more information about the cache maintenance instructions, see *About cache maintenance in AArch32 state* on page G4-5895, *AArch32 cache and branch predictor maintenance instructions* on page G4-5899, and Chapter G5 *The AArch32 Virtual Memory System Architecture*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Boundary conditions for cache maintenance instructions

Cache maintenance instructions operate on the caches regardless of whether the System register Cacheability controls force all memory accesses to be Non-cacheable.

For VA-based cache maintenance instructions, the instructions operate on the caches regardless of the memory type and cacheability attributes marked for the memory address in the VMSA translation table entries. This means that the effects of the cache maintenance instructions can apply regardless of:

- Whether the address accessed:
  - Is Normal memory or Device memory.
  - Has the Cacheable attribute or the Non-cacheable attribute.
- Any applicable domain control of the address accessed.
- The access permissions for the address accessed, other than the effect of the stage two write permission on data or unified cache invalidation instructions.

Ordering of cache and branch predictor maintenance instructions

The following rules describe the effect of the memory order model on the cache and branch predictor maintenance instructions:

- All cache and branch predictor maintenance instructions that do not specify an address execute, relative to each other, in program order.

  All cache and branch predictor instructions that specify an address:
  - Execute in program order relative to all cache and branch predictor operations that do not specify an address.
  - Execute in program order relative to all cache and branch predictor operations that specify the same address.
  - Can execute in any order relative to cache and branch predictor operations that specify a different address.
- Where a cache maintenance or branch predictor instruction appears in program order before a change to the translation tables, the architecture guarantees that the cache or branch predictor maintenance instruction uses the translations that were visible before the change to the translation tables.
- Where a change of the translation tables appears in program order before a cache maintenance or branch predictor instruction, software must execute the sequence outlined in *Ordering and completion of TLB maintenance instructions* on page G5-5998 before performing the cache or branch predictor maintenance instruction, to ensure that the maintenance operation uses the new translations.
- A `DMB` instruction causes the effect of all data or unified cache maintenance instructions appearing in program order before the `DMB` to be visible to all explicit load and store operations appearing in program order after the `DMB`.

  Also, a `DMB` instruction ensures that the effects of any data or unified cache maintenance instruction appearing in program order before the `DMB` are observable by any observer in the same required Shareability domain before any data or unified cache maintenance or explicit memory operations appearing in program order after the `DMB` are observed by the same observer. Completion of the `DMB` does not guarantee the visibility of all data to other observers. For example, all data might not be visible to a translation table walk, or to instruction fetches.
- A `DSB` is required to guarantee the completion of all cache maintenance instruction that appear in program order before the `DSB` instruction.
- A *Context synchronization event* is required to guarantee the effects of any branch predictor maintenance operation. This means a *Context synchronization event* causes the effect of all completed branch predictor maintenance operations appearing in program order before the *Context synchronization event* to be visible to all instructions after the *Context synchronization event*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This means that, if a branch instruction appears after an invalidate branch predictor operation and before any *Context synchronization event*, it is CONSTRAINED UNPREDICTABLE whether the branch instruction is affected by the invalidate. Software must avoid this ordering of instructions, because it might cause CONSTRAINED UNPREDICTABLE behavior.

- Any data or unified cache maintenance instruction by VA must be executed in program order relative to any explicit load or store on the same PE to an address covered by the VA of the cache instruction if that load or store is to Normal Cacheable memory. The order of memory accesses that result from the cache maintenance instruction, relative to any other memory accesses to Normal Cacheable memory, are subject to the memory ordering rules. For more information, see *Definition of the Armv8 memory model* on page E2-3980.

  Any data or unified cache maintenance instruction by VA can be executed in any order relative to any explicit load or store on the same PE to an address covered by the VA of the cache maintenance instruction if that load or store is not to Normal Cacheable memory.

- There is no restriction on the ordering of data or unified cache maintenance instruction by VA relative to any explicit load or store on the same PE where the address of the explicit load or store is not covered by the VA of the cache instruction. Where the ordering must be restricted, a `DMB` instruction must be inserted to enforce ordering.

- There is no restriction on the ordering of a data or unified cache maintenance instruction by set/way relative to any explicit load or store on the same PE. Where the ordering must be restricted, a `DMB` instruction must be inserted to enforce ordering.

- Software must execute a *Context synchronization event* after the completion of an instruction cache maintenance instruction, to guarantee that the effect of the maintenance instruction is visible to any instruction fetch.

A `DSB` or `DMB` instruction intended to ensure the completion of cache maintenance instructions or branch predictor instructions must have an access type of both loads and stores.

See also the completion requirements for cache and branch predictor maintenance instructions in *Completion and endpoint ordering* on page E2-3987.

The scope of instruction cache maintenance depends on the type of the instruction cache. For more information see *Instruction caches* on page G5-6009.

Example G4-1 Cache cleaning operations for self-modifying code

The sequence of cache cleaning operations for a line of self-modifying code on a uniprocessor system is:

```
; Coherency example for data and instruction accesses within the same Inner Shareable domain.
; Enter this code with <Rt> containing a new 32-bit instruction,
; to be held in Cacheable space at a location pointed to by Rn. Use STRH in the first line
; instead of STR for a 16-bit instruction.
        STR Rt, [Rn]
        DCCMVAU Rn          ; Clean data cache by MVA to point of unification (PoU)
        DSB                 ; Ensure visibility of the data cleaned from cache
        ICIMVAU Rn          ; Invalidate instruction cache by MVA to PoU
        BPIMVA  Rn          ; Invalidate branch predictor by MVA to PoU
        DSB                 ; Ensure completion of the invalidations
        ISB                 ; Synchronize the fetched instruction stream
```

Performing cache maintenance instructions

To ensure all cache lines in a block of address space are maintained through all levels of cache Arm strongly recommends that software:

- For data or unified cache maintenance, uses the CTR.DMinLine value to determine the loop increment size for a loop of data cache maintenance by VA instructions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- For instruction cache maintenance, uses the CTR.IMinLine value to determine the loop increment size for a loop of instruction cache maintenance by VA instructions.

Example code for cache maintenance instructions

The cache maintenance instructions by set/way can be used to clean or invalidate, or both, the entirety of one or more levels of cache attached to a PE. However, unless all PEs attached to the caches regard all memory locations as Non-cacheable, it is not possible to prevent locations being allocated into the cache during such a sequence of the cache maintenance instructions.

Note

Because the set/way instructions operate only locally, there is no guarantee of the atomicity of cache maintenance between different PEs, even if those different PEs are each executing the same cache maintenance instructions at the same time. Because any cacheable line can be allocated into the cache at any time, it is possible for a cache line to migrate from an entry in the cache of one PE to the cache of a different PE in a way that means the cache line is not affected by set/way based cache maintenance. Therefore, Arm strongly discourages the use of set/way instructions to manage coherency in coherent systems. The expected use of the cache maintenance instructions that operate by set/way is limited to the cache maintenance associated with the powerdown and powerup of caches, if this is required by the implementation.

The limitations of cache maintenance by set/way mean maintenance by set/way does not happen on multiple PEs, and cannot be made to happen atomically for each address on each PE. Therefore in multiprocessor or multithreaded systems, the use of cache maintenance by set/way to clean, or clean and invalidate, the entire cache for coherency management with very large buffers or with buffers with unknown address can fail to provide the expected coherency results because of speculation by other PEs, or possibly by other threads. The only way that these instructions can be used in this way is to first ensure that all PEs that might cause speculative accesses to caches that need to be maintained are not capable of generating speculative accesses. This can be achieved by ensuring that those PEs have no memory locations with a Normal Cacheable attribute. Such an approach can have very large system performance effects, and Arm advises implementers to use hardware coherency mechanisms in systems where this will be an issue.

System level caches on page G4-5912 refers to other limitations of cache maintenance by set/way.

The following example code for cleaning a data or unified cache to the Point of Coherency illustrates a generic mechanism for cleaning the entire data or unified cache to the Point of Coherency. It assumes the current Cache Size Identification Register is in 32-bit format. For more information, see *Possible formats of the Cache Size Identification Registers, CCSIDR and CCSIDR2* on page G4-5891.

```
    MRC p15, 1, R0, c0, c0, 1   ; Read CLIDR into R0
    ANDS R3, R0, #0x07000000
    MOV R3, R3, LSR #23         ; Cache level value (naturally aligned)
    BEQ Finished
    MOV R10, #0
Loop1
    ADD R2, R10, R10, LSR #1    ; Work out 3 x cache level
    MOV R1, R0, LSR R2          ; bottom 3 bits are the Cache type for this level
    AND R1, R1, #7              ; get those 3 bits alone
    CMP R1, #2
    BLT Skip                    ; no cache or only instruction cache at this level
    MCR p15, 2, R10, c0, c0, 0  ; write CSSELR from R10
    ISB                         ; ISB to sync the change to the CCSIDR
    MRC p15, 1, R1, c0, c0, 0   ; read current CCSIDR to R1
    AND R2, R1, #7              ; extract the line length field
    ADD R2, R2, #4              ; add 4 for the line length offset (log2 16 bytes)
    MOV R4, #0x3FF
    ANDS R4, R4, R1, LSR #3     ; R4 is the max number on the way size (right aligned)
    CLZ R5, R4                  ; R5 is the bit position of the way size increment
    MOV R9, R4                  ; R9 working copy of the max way size (right aligned)
Loop2
    MOV R7, #0x00007FFF
    ANDS R7, R7, R1, LSR #13    ; R7 is the max number of the index size (right aligned)
Loop3
    ORR R11, R10, R9, LSL R5    ; factor in the way number and cache number into R11
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    ORR R11, R11, R7, LSL R2    ; factor in the index number
    MCR p15, 0, R11, c7, c10, 2 ; DCCSW, clean by set/way
    SUBS R7, R7, #1             ; decrement the index
    BGE Loop3
    SUBS R9, R9, #1             ; decrement the way number
    BGE Loop2
Skip
    ADD R10, R10, #2            ; increment the cache number
    CMP R3, R10
    DSB                         ; ensure completion of previous cache maintenance instruction
    BGT Loop1
Finished
```

Similar approaches can be used for all cache maintenance instructions.

G4.4.8 Execution and data prediction restriction System instructions

When ARMv8.0-PredInv is implemented, the System instructions in Table G4-7 prevent predictions based on information gathered from earlier execution within a particular execution context from affecting the later Speculative execution within that context, to the extent that the speculation execution is observable through side-channels.

The prediction restriction System instructions being used by a particular execution context apply to:

- All control flow prediction resources that predict execution addresses.
- Data value prediction.
- Cache allocation prediction.

Table G4-7 Prediction restriction System instructions

Register	Instruction
CFPRCTX	Control Flow Prediction Restriction by Context
CPPRCTX	Cache Prefetch Prediction Restriction by Context
DVPRCTX	Data Value Prediction Restriction by Context

For these System instructions, the execution context is defined by:

- The Security state.
- The Exception level.
- When executing at EL1, the VMID.
- When executing at EL0 when using the PL1&0 translation regime, the ASID and VMID.

Note

- The data value prediction applies to all prediction resources that use some form of training to speculate data values as part of an execution.
- The cache allocation applies to all instruction and data caches, and TLB prefetching hardware used by the executing PE that applies to the supplied context.

The context information is passed as a register argument, and is restricted so that:

- Execution of the System instruction at EL0 only applies to the current hardware defined context.
- Execution of the System instruction at EL1only applies to the current VMID and Security state, and does not apply to EL2 or EL3.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Execution of the System instruction at EL2 can only apply to the current Security state, and does not apply to EL3.

If the System instruction is specified to apply to Exception levels that are not implemented, or which are higher than the Exception level that the System instruction is executed at, then the System instruction is treated as a `NOP`.

When the System instruction is complete and synchronized, no predictions of the restricted type for the affected context are influenced by the execution of the program before the System instruction in a manner that can be observed by the use of any side channels.

Note

Prediction restriction System instructions do not require the invalidation of prediction structures so long as the behavior described for completion is met by an implementation.

These System instructions are guaranteed to be complete following a DSB that covers both read and write behavior on the same PE that executed the original instruction. A subsequent Context synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

In AArch32 state, EL0 access to the System instructions is controlled by SCTLR.EnRCTX.

G4.4.9 Cache lockdown

The concept of an entry locked in a cache is allowed, but not architecturally defined. How lockdown is achieved is IMPLEMENTATION DEFINED and might not be supported by:

- An implementation.
- Some memory attributes.

An unlocked entry in a cache might not remain in that cache. The architecture does not guarantee that an unlocked cache entry remains in the cache or remains incoherent with the rest of memory. Software must not assume that an unlocked item that remains in the cache remains dirty.

A locked entry in a cache is guaranteed to remain in that cache. The architecture does not guarantee that a locked cache entry remains incoherent with the rest of memory, that is, it might not remain dirty.

The interaction of cache lockdown with cache maintenance instructions

The interaction of cache lockdown and cache maintenance instructions is IMPLEMENTATION DEFINED. However, an architecturally-defined cache maintenance instruction on a locked cache line must comply with the following general rules:

- The effect of the following instructions on locked cache entries is IMPLEMENTATION DEFINED:
  - Cache clean by set/way, DCCSW.
  - Cache invalidate by set/way, DCISW.
  - Cache clean and invalidate by set/way, DCISW.
  - Instruction cache invalidate all, ICIALLU and ICIALLUIS.

  However, one of the following approaches must be adopted in all these cases:

  1. If the instruction specified an invalidation, a locked entry is not invalidated from the cache.
  2. If the instruction specified a clean it is IMPLEMENTATION DEFINED whether locked entries are cleaned.
  3. If an entry is locked down, or could be locked down, an IMPLEMENTATION DEFINED Data Abort exception is generated, using the Fault status code defined for this purpose. See *Data Abort exception on page G1-5749*.

  This permits a usage model for cache invalidate routines to operate on a large range of addresses by performing the required operation on the entire cache, without having to consider whether any cache entries are locked.

The effect of the following instructions is IMPLEMENTATION DEFINED:

- Cache clean by virtual address, DCCMVAC and DCCMVAU.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Cache invalidate by virtual address, DCIMVAC.
- Cache clean and invalidate by virtual address, DCCIMVAC.

However, one of the following approaches must be adopted in all these cases:

1. If the instruction specified an invalidation, a locked entry is invalidated from the cache. For the clean and invalidate instructions, the entry must be cleaned before it is invalidated.

2. If the instruction specified an invalidation, a locked entry is not invalidated from the cache. If the instruction specified a clean it is IMPLEMENTATION DEFINED whether locked entries are cleaned.

3. If an entry is locked down, or could be locked down, an IMPLEMENTATION DEFINED Data Abort exception is generated, using the Fault status code defined for this purpose. See DFSR or HSR.

In an implementation that includes EL2, if HCR.TIDCP is set to 1, any exception relating to lockdown of an entry associated with Non-secure memory is routed to EL2.

Note

An implementation that uses an abort mechanism for entries that can be locked down but are not actually locked down must:

- Document the IMPLEMENTATION DEFINED instruction sequences that perform the required operations on entries that are not locked down.

- Implement one of the other permitted alternatives for the locked entries.

Arm recommends that, when possible, such IMPLEMENTATION DEFINED instruction sequences use architecturally-defined instructions. This minimizes the number of customized instructions required.

In addition, an implementation that uses an abort to handle cache maintenance instructions for entries that might be locked must provide a mechanism that ensures that no entries are locked in the cache.

The reset setting of the cache must be that no cache entries are locked.

Additional cache functions for the implementation of lockdown

An implementation can add additional cache maintenance functions for the handling of lockdown in the IMPLEMENTATION DEFINED space.

G4.4.10 System level caches

The Arm Architecture defines a *system cache* as a cache that is not described in the PE Cache Identification registers, CCSIDR, CCSIDR2, and CLIDR, and for which the set/way cache maintenance instructions do not apply.

Conceptually, three classes of system cache can be envisaged:

1. System caches which lie before the point of coherency and cannot be managed by cache maintenance instructions. Such systems fundamentally undermine the concept of cache maintenance instructions operating to the point of coherency, as they imply the use of non-architecture mechanisms to manage coherency. The use of such systems in the Arm architecture is explicitly prohibited.

2. System caches which lie before the point of coherency and can be managed by cache maintenance by address instructions that apply to the point of coherency, but cannot be managed by cache maintenance by set/way instructions. Where maintenance of the entire system cache must be performed, as is the case for power management, it must be performed using non-architectural mechanisms.

3. System caches which lie beyond the point of coherency and so are invisible to software. The management of such caches is outside the scope of architecture.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G4.5 System register support for IMPLEMENTATION DEFINED memory features

The VMSAv8-32 defines the following registers for describing IMPLEMENTATION DEFINED features of the memory system:

- The TCM Type Register,TCMTR must be implemented on any implementation where EL1 or above supports AArch32. The format of this register is IMPLEMENTATION DEFINED.
- The System register encoding space with {`coproc`==0b1111, `CRn`==`c9`, `CRm`=={`c0-c2`, `c5-c7`}} is IMPLEMENTATION DEFINED for all values of `opc2` and `opc1`. This space is reserved for branch predictor, cache and TCM functionality, for example maintenance, override behaviors and lockdown.
- In a VMSAv8-32 implementation, part of the System register encoding space with {`coproc`==`0b1111`, `CRn`==`c10`} is IMPLEMENTATION DEFINED and reserved for TLB functionality, see *TLB lockdown* on page G5-5993.
- The System register encoding space with {`coproc`==0b1111, `CRn`==`c11`, `CRm`=={`c0-c8`, `c15`}} is IMPLEMENTATION DEFINED for all values of `opc2` and `opc1`. This space is reserved for DMA operations to and from the TCMs.

In addition, the System register encoding space with {`coproc`==0b1111, `CRn`==`c15`}is reserved for IMPLEMENTATION DEFINED registers, and can provide additional registers for the memory system. For more information, see *VMSAv8-32 organization of registers in the (coproc==0b1111) encoding space* on page G7-6078.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G4.6 External aborts

The Arm architecture defines External aborts as errors that occur in the memory system, other than those that are detected by the MMU or Debug hardware. An External abort might signal a data corruption to the PE. For example, a memory location might have been corrupted, and this corruption is detected by hardware using a parity or error correction code (ECC). The error might have been propagated. The RAS Extension provides mechanisms for software to determine the extent of the corruption and contain propagation of the error. For more information, see the *ARM® Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for the ARMv8-A architecture profile*.

An External abort is one of:

- Synchronous.
- Precise asynchronous.
- Imprecise asynchronous.

For more information, see *Exception terminology* on page G1-5674.

The RAS Extension provides an expanded taxonomy for describing aborts. When the RAS Extension is not implemented, the Arm architecture does not provide any method to distinguish between precise asynchronous and imprecise asynchronous External aborts.

VMSAv8-32 permits External aborts on data accesses, translation table walks, and instruction fetches to be either synchronous or asynchronous. The reported fault code identifies whether the External abort is synchronous or asynchronous.

It is IMPLEMENTATION DEFINED which External aborts, if any, are supported. Asynchronous External aborts generate SError interrupt exceptions.

In AArch32 state:

- SError interrupts are taken as asynchronous Data Abort exceptions.
- Synchronous External aborts:
  - On data accesses are taken as synchronous Data Abort exceptions.
  - On instruction fetches, or prefetches, are taken as synchronous Prefetch Abort exceptions.

See also:

- *External abort on a translation table walk* on page G5-6020.
- *Handling exceptions that are taken to an Exception level using AArch32* on page G1-5703.

Normally, External aborts are rare. An imprecise asynchronous External abort is likely to be fatal to the process that is running. Arm recommends that implementations make External aborts precise wherever possible.

The following subsections give more information about possible External aborts:

- *Provision for classification of External aborts*.
- *Parity or ECC error reporting, RAS Extension not implemented* on page G4-5915.

The section *Exception reporting in a VMSAv8-32 implementation* on page G5-6024 describes the reporting of External aborts.

G4.6.1 Provision for classification of External aborts

For an External abort taken to a privileged mode other than Hyp mode, an implementation can use the DFSR.ExT or IFSR.ExT bits to provide more information about the External abort:

- DFSR.ExT provides an IMPLEMENTATION DEFINED classification of External aborts on data accesses.
- IFSR.ExT provides an IMPLEMENTATION DEFINED classification of External aborts on instruction accesses.

For an External abort taken to Hyp mode, the HSR.EA bit, provides an IMPLEMENTATION DEFINED classification of External aborts.

For all aborts other than External aborts these bits return a value of 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the RAS Extension is implemented:

- The HSR.AET field provides information about the state of the PE following an SError interrupt exception taken to Hyp mode.
- The DFSR.AET field provides information about the state of the PE following an asynchronous Data Abort exception.
- The implementation might define error record registers.

For more information on the RAS Extension, see *ARM® Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for the ARMv8-A architecture profile*.

G4.6.2 Parity or ECC error reporting, RAS Extension not implemented

The Arm architecture supports the reporting of both synchronous and asynchronous parity or ECC errors from the cache systems. It is IMPLEMENTATION DEFINED what parity or ECC errors in the cache systems, if any, result in synchronous or asynchronous parity or ECC errors.

A fault code is defined for reporting parity or ECC errors, see *Exception reporting in a VMSAv8-32 implementation* on page G5-6024. However when parity or ECC error reporting is implemented it is IMPLEMENTATION DEFINED whether a parity or ECC error is reported using the assigned fault code, or using another appropriate encoding.

For all purposes other than the Fault status encoding, parity or ECC errors are treated as External aborts.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G4.7 Memory barrier instructions

Memory barriers on page E2-3991 describes the memory barrier instructions. This section describes the system level controls of those instructions.

G4.7.1 EL2 control of the Shareability of data barrier instructions executed at EL0 or EL1

In an implementation that includes EL2 and supports Shareability limitations on the data barrier instructions, the HCR.BSU field can modify the required Shareability of an instruction that is executed at EL0 or EL1 in Non-secure state. Table G4-8 shows the encoding of this field:

Table G4-8 EL2 control of Shareability of barrier instructions executed at EL0 or EL1

HCR.BSU	Minimum Shareability of barrier instructions
00	No effect, Shareability is as specified by the instruction
01	Inner Shareable
10	Outer Shareable
11	Full system

For an instruction executed at EL0 or EL1 in Non-secure state, Table G4-9 shows how the HCR.BSU is combined with the Shareability specified by the argument of the `DMB` or `DSB` instruction to give the scope of the instruction:

Table G4-9 Effect of the HCR_EL2.BSU on barrier instructions executed at Non-secure EL1 or EL1

Shareability specified by the `DMB` or `DSB` argument	HCR.BSU	Resultant Shareability
Full system	Any	Full system
Outer Shareable	`00`, `01`, or `10`	Outer Shareable
	`11`, Full system	Full system
Inner Shareable	`00` or `01`	Inner Shareable
	`10`, Outer Shareable	Outer Shareable
	`11`, Full system	Full system
Non-shareable	`00`, No effect	Non-shareable
	`01`, Inner Shareable	Inner Shareable
	`10`, Outer Shareable	Outer Shareable
	`11`, Full system	Full system

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G4.8 Pseudocode description of general memory System instructions

This section lists the pseudocode describing general memory operations:

- *Memory data type definitions*.
- *Basic memory access*.
- *Aligned memory access*.
- *Unaligned memory access* on page G4-5918.
- *Exclusives monitors operations* on page G4-5918.
- *Access permission checking* on page G4-5919.
- *Abort exceptions* on page G4-5919.
- *Memory barriers* on page G4-5919.

G4.8.1 Memory data type definitions

This section lists the memory data types.

The memory data types are:

- Address descriptor, defined by the `AddressDescriptor` type.
- Full address, defined by the `FullAddress` type.
- Memory attributes, defined by the `MemoryAttributes` type.
- Memory type, defined by the `MemType` enumeration.
- Device memory type, defined by the `DeviceType` enumeration.
- Normal memory attributes, defined by the `MemAttrHints` type.
- Cacheability attributes, defined by the `MemAttr_NC`, `MemAttr_WT`, and `MemAttr_WB` constants.
- Allocation hints, defined by the `MemHint_No`, `MemHint_WA`, `MemHint_RA`, and `MemHint_RWA` constants.
- Access permissions, defined by the `Permissions` type.

G4.8.2 Basic memory access

The two forms of the `_Mem[]` accessor, non-assignment (memory read) and assignment (memory write), are the operations that perform single-copy atomic, aligned, little-endian memory accesses of `size` bytes to or from the underlying physical memory array of bytes.

The functions address the array using `desc.paddress`, that supplies:

- The physical address.
- An NS bit that selects between the Secure and Non-secure parts of the array.

The attributes in `desc.memattrs` are used by the memory system to determine caching and ordering behaviors as described in *Memory types and attributes* on page E2-4010, *Definition of the Armv8 memory model* on page E2-3980, and *Atomicity in the Arm architecture* on page E2-3976.

An additional parameter to the `_Mem[]` accessor defines the access type, for example normal, exclusive, or ordered, and whether the access is made as part of a translation table walk.

The actual implemented array of memory might be smaller than the maximum address size that can be accessed from AArch32 state. In this case the scheme for aliasing is IMPLEMENTATION DEFINED, or some parts of the address space might give rise to External aborts or SErrors (System Errors).

`PAMax()` returns the IMPLEMENTATION DEFINED size of the physical address.

Note

A stage of address translation using VMSAv8-32 cannot generate an output address of more than 40 bits.

G4.8.3 Aligned memory access

The `AArch32.MemSingle[]` functions make atomic, little-endian accesses of `size` bytes.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G4.8.4 Unaligned memory access

See *Unaligned data access* on page E2-4004 for details of the SCTLR.A and HSCTLR.A controls on the generation of alignment faults. The HSCTLR control applies to Normal memory accesses from Hyp mode, and the SCTLR control applies to Normal memory accesses from all other modes.

The `Mem_with_type[]` functions make an access of the required type. If that access is naturally aligned, each form of the function performs an atomic access by making a single call to `AArch32.MemSingle[]`. If that access is not aligned but passes the `AArch32.CheckAlignment()` checks, each form of the function synthesizes the required access from multiple calls to `AArch32.MemSingle[]`. It also reverses the byte order if the access is big-endian.

G4.8.5 Exclusives monitors operations

The `AArch32.SetExclusiveMonitors()` function sets the Exclusives monitors for a Load-Exclusive instruction, for a block of bytes. The size of the blocks is determined by `size`, at the VA `address`. The `ExclusiveMonitorsPass()` function checks whether a Store-Exclusive instruction still has possession of the Exclusives monitors and therefore completes successfully.

The `AArch32.ExclusiveMonitorsPass()` function checks whether a Store-Exclusive instruction still has possession of the Exclusives monitors, by checking whether the Exclusives monitors are set to include the location of the memory block specified by `size`, at the virtual address defined by `address`. The atomic write that follows after the Exclusives monitors have been set must be to the same physical address. It is permitted, but not required, for this function to return FALSE if the virtual address is not the same as that used in the previous call to `AArch32.SetExclusiveMonitors()`.

The `ExclusiveMonitorsStatus()` function returns 0 if the previous atomic write was to the same physical memory locations selected by `ExclusiveMonitorsPass()` and therefore succeeded. Otherwise the function returns 1, indicating that the address translation delivered a different physical address.

The `MarkExclusiveGlobal()` procedure takes as arguments a `FullAddress.paddress`, the PE identifier `processorid` and the size of the transfer. The procedure records that the PE `processorid` has requested exclusive access covering at least `size` bytes from address `paddress`. The size of the location marked as exclusive is IMPLEMENTATION DEFINED, up to a limit of 2KB and no smaller than two words, and aligned in the address space to the size of the location. It is CONSTRAINED UNPREDICTABLE whether this causes any previous request for exclusive access to any other address by the same PE to be cleared.

The `MarkExclusiveLocal()` procedure takes as arguments a FullAddress `paddress`, the PE identifier `processorid` and the size of the transfer. The procedure records in a local record that PE `processorid` has requested exclusive access to an address covering at least `size` bytes from address paddress. The size of the location marked as exclusive is IMPLEMENTATION DEFINED, and can at its largest cover the whole of memory but is no smaller than two words, and is aligned in the address space to the size of the location. It is IMPLEMENTATION DEFINED whether this procedure also performs a `MarkExclusiveGlobal()` using the same parameters.

The `IsExclusiveGlobal()` function takes as arguments a FullAddress `paddress`, the PE identifier `processorid` and the size of the transfer. The function returns TRUE if the PE `processorid` has marked in a global record an address range as exclusive access requested that covers at least `size` bytes from address `paddress`. It is IMPLEMENTATION DEFINED whether it returns TRUE or FALSE if a global record has marked a different address as exclusive access requested. If no address is marked in a global record as exclusive access, `IsExclusiveGlobal()` returns FALSE.

The `IsExclusiveLocal()` function takes as arguments a FullAddress `paddress`, the PE identifier `processorid` and the size of the transfer. The function returns TRUE if the PE `processorid` has marked an address range as exclusive access requested that covers at least the `size` bytes from address `paddress`. It is IMPLEMENTATION DEFINED whether this function returns TRUE or FALSE if the address marked as exclusive access requested does not cover all of `size` bytes from address `paddress`. If no address is marked as exclusive access requested, then this function returns FALSE. It is IMPLEMENTATION DEFINED whether this result is ANDed with the result of `IsExclusiveGlobal()` with the same parameters.

The `ClearExclusiveByAddress()` procedure takes as arguments a `FullAddress` `paddress`, the PE identifier `processorid` and the size of the transfer. The procedure clears the global records of all PEs, other than `processorid`, for which an address region including any of `size` bytes starting from `paddress` has had a request for an exclusive access. It is

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

IMPLEMENTATION DEFINED whether the equivalent global record of the PE `processorid` is also cleared if any of `size` bytes starting from `paddress` has had a request for an exclusive access, or if any other address has had a request for an exclusive access.

The `ClearExclusiveLocal()` procedure takes as arguments the PE identifier `processorid`. The procedure clears the local record of PE `processorid` for which an address has had a request for an exclusive access. It is IMPLEMENTATION DEFINED whether this operation also clears the global record of PE `processorid` that an address has had a request for an exclusive access.

G4.8.6 Access permission checking

The function `AArch32.CheckPermission()` is used by the architecture to perform access permission checking based on attributes derived from the translation tables or location descriptors.

The interpretation of access permission is shown in *Memory access control* on page G5-5968.

G4.8.7 Abort exceptions

The function `AArch32.Abort()` generates a Data Abort exception or a Prefetch Abort exception by calling the `AArch32.TakeDataAbortException()` or `AArch32.TakePrefetchAbortException()` function.

The `FaultRecord` type describes a fault. Functions that check for faults return a record of this type appropriate to the type of fault. *Pseudocode description of VMSAv8-32 memory system operations* on page G5-6050 provides a number of wrappers to generate a FaultRecord.

The function `AArch32.NoFault()` returns a null record that indicates no fault. The `IsFault()` function tests whether a FaultRecord contains a fault.

G4.8.8 Memory barriers

The definition for the memory barrier functions is given by the enumerations `MBReqDomain` and `MBReqTypes`.

These enumerations define the required Shareability domains and required access types used as arguments for `DMB` and `DSB` instructions.

The procedures `DataMemoryBarrier()`, `DataSynchronizationBarrier()`, and `InstructionSynchronizationBarrier()` perform the memory barriers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter G5
The AArch32 Virtual Memory System Architecture

This chapter describes the Armv8-A AArch32 Virtual Memory System Architecture (VMSA), that is backwards-compatible with VMSAv7. It includes the following sections:

- *About VMSAv8-32* on page G5-5922.
- *The effects of disabling address translation stages on VMSAv8-32 behavior* on page G5-5930.
- *Translation tables* on page G5-5934.
- *The VMSAv8-32 Short-descriptor translation table format* on page G5-5939.
- *The VMSAv8-32 Long-descriptor translation table format* on page G5-5948.
- *Memory access control* on page G5-5968.
- *Memory region attributes* on page G5-5979.
- *Translation Lookaside Buffers (TLBs)* on page G5-5991.
- *TLB maintenance requirements* on page G5-5995.
- *Caches in VMSAv8-32* on page G5-6009.
- *VMSAv8-32 memory aborts* on page G5-6012.
- *Exception reporting in a VMSAv8-32 implementation* on page G5-6024.
- *Address translation instructions* on page G5-6043.
- *Pseudocode description of VMSAv8-32 memory system operations* on page G5-6050.
- *About the System registers for VMSAv8-32* on page G5-6052.
- *Functional grouping of VMSAv8-32 System registers* on page G5-6058.

Note

This chapter must be read with Chapter G4 *The AArch32 System Level Memory Model*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G5.1 About VMSAv8-32

This chapter describes the Armv8 VMSA for AArch32 state, VMSAv8-32. This is generally equivalent to VMSAv7 for an implementation that includes all of the Security Extensions, the Multiprocessing Extensions, the Large Physical Address Extension, and the Virtualization Extensions.

This chapter describes the control of the VMSA by Exception levels that are using AArch32. *Security state, Exception levels, and AArch32 execution privilege* on page G1-5682 summarizes how the AArch32 PE modes map onto the Exception levels.

ARMv8.4-SecEL2, if implemented, is not available in AArch32 state and EL2 only executes in Non-secure state.

ARMv8.4-S2FWB, if implemented, is not available in AArch32 state. If EL2 is executing in AArch64 state 2 stage translations might be affected. For more informations see Chapter D5 *The AArch64 Virtual Memory System Architecture*.

Chapter D5 *The AArch64 Virtual Memory System Architecture* describes the control of the VMSA by Exception levels that are using AArch64.

The main function of the VMSA is to perform address translation, and access permissions and memory attribute determination and checking, for memory accesses made by the PE. Address translation, and permissions and attribute determination and checking, is performed by a *stage* of address translation.

In VMSAv8-32, the *Memory Management Unit* (MMU) provides a number of stages of address translation. This chapter describes only the stages that are visible from Exception levels that are using AArch32, which are as follows:

For operation in Secure state

A single stage of address translation, for use when executing at PL1 or EL0. This is the *Secure PL1&0 stage 1* address translation stage.

For operation in Non-secure state

- A single stage of address translation for use when executing at EL2. This is the *Non-secure EL2 stage 1* address translation stage.
- Two stages of address translation for use when executing at PL1 or EL0. These are:
  - The *Non-secure PL1&0 stage 1* address translation stage.
  - The *Non-secure PL1&0 stage 2* address translation stage.

The System registers provide independent control of each supported stage of address translation, including a control to disable that stage of translation.

However, if the PE is executing at EL0 using AArch32 when EL1 is using AArch64 then it is using the VMSAv8-64 EL1&0 translation regime, described in Chapter D5 *The AArch64 Virtual Memory System Architecture*.

These features mean the VMSAv8-32 can support a hierarchy of software supervision, for example an Operating System and a hypervisor.

Each stage of address translation uses address translations and associated memory properties held in memory mapped tables called *translation tables*.

For information about how the MMU features differ if an implementation does not include all of the Exception levels, see *About address translation for VMSAv8-32* on page G5-5925.

The translation tables define the following properties:

Access to the Secure or Non-secure address map

The translation table entries determine whether an access from Secure state accesses the Secure or the Non-secure address map. Any access from Non-secure state accesses the Non-secure address map.

Memory access permission control

This controls whether a program is permitted to access a memory region. For instruction and data access, the possible settings are:

- No access.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Read-only.
- Write-only. This is possible only in a translation regime with two stages of translation.
- Read/write.

For instruction accesses, additional controls determine whether instructions can be fetched and executed from the memory region.

If a PE attempts an access that is not permitted, a memory fault is signaled to the PE.

Memory region attributes

These describe the properties of a memory region. The top-level attribute, the Memory type, is one of Normal, or a type of Device memory, as follows:

- Both translation table formats support the following Device memory types:
  - Device-nGnRnE
  - Device-nGnRE
- The Long-descriptor translation table format supports, in addition, the following Device memory types:
  - Device-nGRE
  - Device-GRE

Note

Armv8 added the Device-nGRE and Device-GRE memory types. Also, in versions of the Arm architecture before Armv8:

- Device-nGnRnE memory is described as Strongly-ordered memory.
- Device-nGnRE memory is described as Device memory.

Normal memory regions can have additional attributes.

For more information, see *Memory types and attributes* on page E2-4010.

Address translation mappings

An address translation maps an *input address* to an *output address*.

A stage 1 translation takes the address of an explicit data access or instruction fetch, a *virtual address* (VA), as the input address, and translates it to a different output address:

- If only one stage of translation is provided, this output address is the *physical address* (PA).
- If two stages of address translation are provided, the output address of the stage 1 translation is an *intermediate physical address* (IPA).

Note

In the Armv8-32 architecture, a software agent, such as an Operating System, that uses or defines stage 1 memory translations, might be unaware of the distinction between IPA and PA.

A stage 2 translation translates the IPA to a PA.

The possible Security states and privilege levels of memory accesses define a set of *translation regimes*, where a translation regime maps an input VA to the corresponding PA, using one or two stages of translation. See *The VMSAv8-32 translation regimes* on page G5-5924.

System registers control VMSAv8-32, including defining the location of the translation tables, and enabling and configuring the MMU, including enabling and disabling the different address translation stages. Also, they report any faults that occur on a memory access. For more information, see *Functional grouping of VMSAv8-32 System registers* on page G5-6058.

The following sections give an overview of VMSAv8-32, and of the implementation options for VMSAv8-32:

- *The VMSAv8-32 translation regimes* on page G5-5924.
- *Address types used in a VMSAv8-32 description* on page G5-5924.
- *Address spaces in VMSAv8-32* on page G5-5925.
- *About address translation for VMSAv8-32* on page G5-5925.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The remainder of the chapter fully describes the VMSA, including the different implementation options, as summarized in *Organization of the remainder of this chapter* on page G5-5929.

G5.1.1 The VMSAv8-32 translation regimes

As introduced in *Address translation mappings* on page G5-5923, a translation regime maps an input VA to the corresponding PA, using one or two stages of translation. Figure G5-1 shows the VMSAv8-32 translation regimes, and their associated translation stages and the Exception levels from which they are controlled.

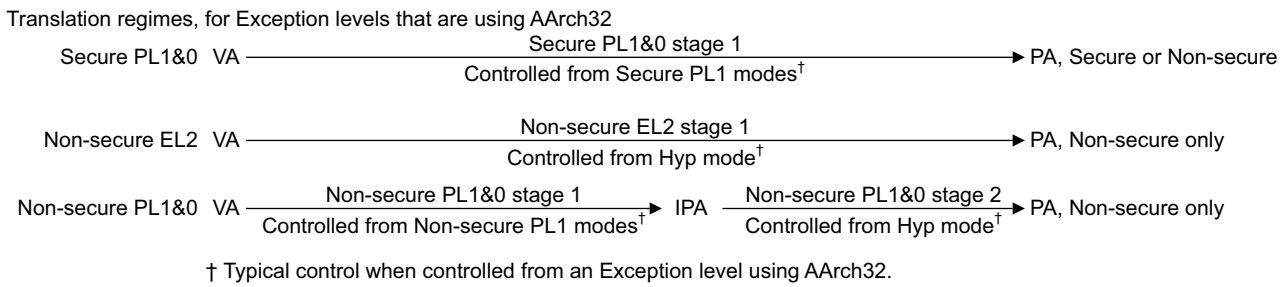


Figure G5-1 VMSAv8-32 translation regimes, and associated control

Note

Conceptually, a translation regime that has only a stage 1 address translation is equivalent to a regime with a fixed, flat stage 2 mapping from IPA to PA.

Limited use of Privilege level in Armv8 AArch32 state on page G1-5683 describes the mapping between the PE modes and the Privilege levels (PLs).

Alternative descriptions of the PL1&0 translation regime

The PL1&0 is described in terms of Privilege level because of the way the AArch32 PE modes map onto the Exception levels, as described in *Limited use of Privilege level in Armv8 AArch32 state* on page G1-5683. The description of this translation regime in terms of the Exception levels using depends on the current state of the PE, as follows:

- In Non-secure state, PL1 always maps to EL1, and therefore the Non-secure PL1&0 translation regime could be described as the Non-secure EL1&0 translation regime.
- In Secure state:
  - When EL3 is using AArch32, PL1 maps to EL3, and therefore under these conditions the Secure PL1&0 translation regime could be described as the Secure EL3&0 translation regime,
  - When EL3 is using AArch64, Secure PL1 maps to Secure EL1, and therefore under these conditions the Secure PL1&0 translation regime could be described as the Secure EL1&0 translation regime,

However, these descriptions all refer to the same translation regime, with the same System registers associated with its stage 1 translations. Therefore, the regime is generally referred to as the PL1&0 translation regime.

Note

As Figure G5-1 shows, stage 2 translation is supported only in Non-secure state.

G5.1.2 Address types used in a VMSAv8-32 description

A description of VMSAv8-32 refers to the following address types.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

These descriptions relate to a VMSAv8-32 description and therefore sometimes differ from the generic definitions given in the Glossary.

Virtual address (VA)

An address used in an instruction, as a data or instruction address, is a Virtual Address (VA).

An address held in the PC, LR, or SP, is a VA.

The VA map runs from zero to the size of the VA space. For AArch32 state, the maximum VA space is 4GB, giving a maximum VA range of `0x00000000-0xFFFFFFFF`.

Intermediate physical address (IPA)

In a translation regime that provides two stages of address translation, the IPA is the address after the stage 1 translation, and is the input address for the stage 2 translation.

In a translation regime that provides only one stage of address translation, the IPA is identical to the PA.

A VMSAv8-32 implementation provides only one stage of address translation:

- If the implementation does not include EL2.
- When executing in Secure state.
- When executing in Hyp mode.

Physical address (PA)

The address of a location in the Secure or Non-secure memory map. That is, an output address from the PE to the memory system.

G5.1.3 Address spaces in VMSAv8-32

For execution in AArch32 state, the Armv8 architecture supports:

- A VA space of up to 32 bits. The actual width is IMPLEMENTATION DEFINED.
- An IPA space of up to 40 bits. The translation tables and associated System registers define the width of the implemented address space.

Note

AArch32 defines two translation table formats. The *Long-descriptor* format gives access to the full 40-bit IPA or PA space at a granularity of 4KB. The *Short-descriptor* format:

- Gives access to a 32-bit PA space at 4KB granularity.
- Gives access to a 40-bit PA space, but only at 16MB granularity, by the use of Supersections.

If an implementation includes EL3, the address maps are defined independently for Secure and Non-secure operation, providing two independent 40-bit address spaces, where:

- A VA accessed from Non-secure state can only be translated to the Non-secure address map.
- A VA accessed from Secure state can be translated to either the Secure or the Non-secure address map.

G5.1.4 About address translation for VMSAv8-32

Address translation is the process of mapping one address type to another, for example, mapping VAs to IPAs, or mapping VAs to PAs. A *translation table* defines the mapping from one address type to another, and a *Translation table base register* (TTBR) indicates the start of a translation table. Each implemented stage of address translation shown in Figure G5-1 on page G5-5924 requires its own translation tables.

For PL1&0 stage 1 translations, the mapping can be split between two tables, one controlling the lower part of the VA space, and the other controlling the upper part of the VA space. This can be used, for example, so that:

- One table defines the mapping for operating system and I/O addresses, that do not change on a context switch.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- A second table defines the mapping for application-specific addresses, and therefore might require updating on a context switch.

The VMSAv8-32 implementation options determine the supported address translation stages. The following descriptions apply when all implemented Exception levels are using AArch32:

VMSAv8-32 without EL2 or EL3

Supports only a single PL1&0 stage 1 address translation. Translation of this stage of address translation can be split between two sets of translation tables, with base addresses defined by TTBR0 and TTBR1, and controlled by TTBCR.

VMSAv8-32 with EL3 but without EL2

Supports only the Secure PL1&0 stage 1 address translation and the Non-secure PL1&0 stage 1 address translation. In each Security state, this stage of translation can be split between two sets of translation tables, with base addresses defined by the Secure and Non-secure copies of TTBR0 and TTBR1, and controlled by the Secure and Non-secure copies of TTBCR.

VMSAv8-32 with EL2 but without EL3

The implementation supports the following stages of address translation:

Non-secure EL2 stage 1 address translation

The HTTBR defines the base address of the translation table for this stage of address translation, controlled by HTCR.

Non-secure PL1&0 stage 1 address translation

Translation of this stage of address translation can be split between two sets of translation tables, with base addresses defined by the Non-secure copies of TTBR0 and TTBR1 and controlled by the Non-secure instance of TTBCR.

Non-secure PL1&0 stage 2 address translation

The VTTBR defines the base address of the translation table for this stage of address translation, controlled by VTCR.

VMSAv8-32 with EL2 and EL3

The implementation supports all of the stages of address translation, as follows:

Secure PL1&0 stage 1 address translation

Translation of this stage of address translation can be split between two sets of translation tables, with base addresses defined by the Secure copies of TTBR0 and TTBR1, and controlled by the Secure instance of TTBCR.

Non-secure EL2 stage 1 address translation

The HTTBR defines the base address of the translation table for this stage of address translation, controlled by HTCR.

Non-secure PL1&0 stage 1 address translation

Translation of this stage of address translation can be split between two sets of translation tables, with base addresses defined by the Non-secure copies of TTBR0 and TTBR1 and controlled by the Non-secure instance of TTBCR.

Non-secure PL1&0 stage 2 address translation

The VTTBR defines the base address of the translation table for this stage of address translation, controlled by VTCR.

Figure G5-2 on page G5-5927 shows the translation regimes and stages in a VMSAv8-32 implementation that includes all of the Exception levels, and indicates the PE mode that, typically, defines each set of translation tables, if that stage of address translation is controlled by a Privilege level that is using AArch32:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

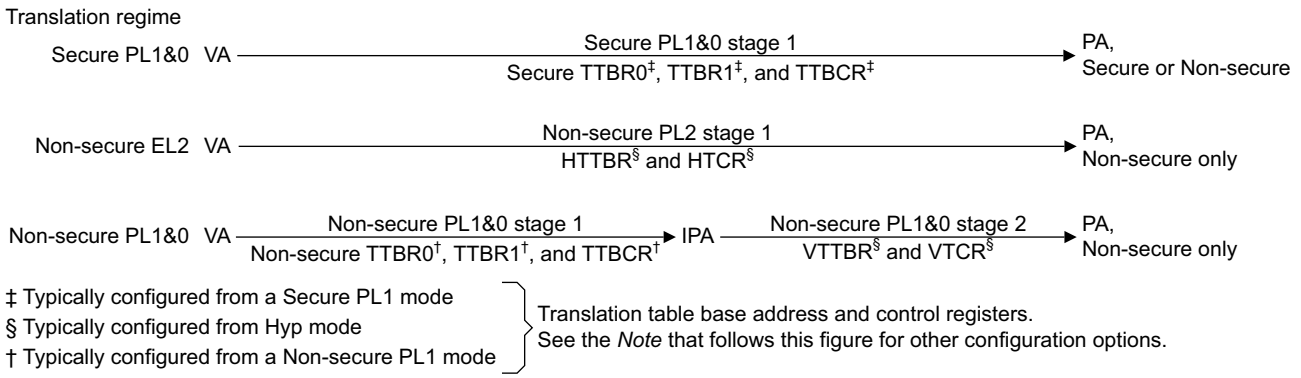


Figure G5-2 VMSAv8-32 address translation summary

Note

The term *Typically configured* is used in Figure G5-2 to indicate the expected software usage. However, stages of address translation used in AArch32 state can also be configured:

- From an Exception level higher than the Exception level of the configuring PE mode shown in Figure G5-2, regardless of whether that Exception level is using AArch32 or is using AArch64, except that a Non-secure Exception level can never configure a stage of address translation that is used in Secure state.
- From an Exception level that is using AArch64 and is higher than the level at which the translation stage is being used. For example, if Non-secure EL0 is the only Non-secure Exception level that is using AArch32, then the Non-secure PL1&0 stage of address translation is configured from Non-secure EL1, that is using AArch64.

In general:

- The translation from VA to PA can require multiple *stages* of address translation, as Figure G5-2 shows.
- A single stage of address translation takes an *input address* and translates it to an *output address*.

A full translation table lookup is called a *translation table walk*. It is performed automatically by hardware, and can have a significant cost in execution time. To support fine granularity of the VA to PA mapping, a single input address to output address translation can require multiple accesses to the translation tables, with each access giving finer granularity. Each access is described as a *level* of address lookup. The final level of the lookup defines:

- The required output address.
- The *attributes* and *access permissions* of the addressed memory.

Translation Lookaside Buffers (TLBs) reduce the average cost of a memory access by caching the results of translation table walks. TLBs behave as caches of the translation table information, and VMSAv8-32 provides TLB maintenance instructions for the management of TLB contents.

Note

The Arm architecture permits TLBs to hold any translation table entry that does not directly cause a Translation fault, an Address size fault, or an Access flag fault.

To reduce the software overhead of TLB maintenance, for the PL1&0 translation regimes VMSAv8-32 distinguishes between *Global pages* and *Process-specific pages*. The *Address Space Identifier* (ASID) identifies pages associated with a specific process and provides a mechanism for changing process-specific tables without having to maintain the TLB structures.

If an implementation includes EL2, the *virtual machine identifier* (VMID) identifies the current virtual machine, with its own independent ASID space. The TLB entries include this VMID information, meaning TLBs do not require explicit invalidation when changing from one virtual machine to another, if the virtual machines have different VMIDs. For stage 2 translations, all translations are associated with the current VMID. There is no mechanism to associate a particular stage 2 translation with multiple virtual machines.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Atomicity of register changes on changing virtual machine

From the viewpoint of software executing at Non-secure PL1 or EL0, when there is a switch from one virtual machine to another, the registers that control or affect address translation must be changed atomically. This applies to the registers for the Non-secure PL1&0 translation regime. This means that all of the following registers must change atomically:

- The registers associated with the stage 1 translations:
  - MAIR0, MAIR1, AMAIR0, and AMAIR1.
  - TTBR0, TTBR1, TTBCR, TTBCR2, and CONTEXTIDR.
  - SCTLR.
- The registers associated with the stage 2 translations:
  - VTTBR and VTCR.
  - HSCTLR.

Note

Only some fields of SCTLR affect the stage 1 translation, and only some fields of HSCTLR affect the stage 2 translation. However, in each case, changing these fields requires a write to the register, and that write must be atomic with the other register updates.

These registers apply to execution using the Non-secure PL1&0 translation regime. However, when updated as part of a switch of virtual machines they are updated by software executing at EL2. This means the registers are out of context when they are updated, and no synchronization precautions are required.

Use of out-of-context translation regimes

The architecture requires that:

- When executing at EL3 or EL2, the PE must not use the registers associated with the Non-secure PL1&0 translation regime for speculative memory accesses.
- When executing at EL3 the PE must not use the registers associated with the EL2 translation regime for speculative memory accesses.
- When executing at EL3, EL2, or Non-secure EL1, the PE must not use the registers associated with the Secure PL1&0 translation regime for speculative memory accesses.

If SPE is not in use for a lower Exception level when entering an Exception level on completion of a `DSB` instruction, then no new memory accesses using any translation table entries from a translation regime of an Exception level lower than the Exception level that has been entered will be observed by any observers to the extent that those accesses are required to be observed, as determined by the Shareability and Cacheability of those translation table entries.

If SPE is in use for a lower Exception level when entering an Exception level on completion of a `PSB CSYNC` and a subsequent `DSB` instruction, then no new memory accesses using any translation table entries from a translation regime of an Exception level lower than the Exception level that has been entered will be observed by any observers, to the extent that those accesses are required to be observed, as determined by the Shareability and Cacheability of those translation table entries.

Note

- This does not require that speculative memory accesses cannot be performed using those entries if it is impossible to tell that those memory accesses have been observed by the observers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- This requirement does not imply that, on taking an exception to a higher Exception level, any translation table walks started before the exception was taken will be completed by the time the higher Exception level is entered, and therefore memory accesses required for such a translation table walk might, in effect, be performed speculatively. However, the execution of a `DSB` on entry to the higher Exception level ensures that these accesses are complete.

G5.1.5 Organization of the remainder of this chapter

The remainder of this chapter is organized as follows.

The next part of the chapter describes address translation and the associated memory properties held in the translation table entries, in the following sections:

- *The effects of disabling address translation stages on VMSAv8-32 behavior* on page G5-5930.
- *Translation tables* on page G5-5934.
- *Secure and Non-secure address spaces* on page G5-5937.
- *The VMSAv8-32 Short-descriptor translation table format* on page G5-5939.
- *The VMSAv8-32 Long-descriptor translation table format* on page G5-5948.
- *Memory access control* on page G5-5968.
- *Memory region attributes* on page G5-5979.
- *Translation Lookaside Buffers (TLBs)* on page G5-5991.
- *TLB maintenance requirements* on page G5-5995.

Caches in VMSAv8-32 on page G5-6009 describes VMSAv8-32-specific cache requirements.

The following sections then describe aborts on VMSAv8-32 memory accesses, and how these and other faults are reported:

- *VMSAv8-32 memory aborts* on page G5-6012.
- *Exception reporting in a VMSAv8-32 implementation* on page G5-6024.

Address translation instructions on page G5-6043 then describes these operations, and how they relate to address translation.

A number of sections then describe the System registers for VMSAv8-32. The following sections give general information about the System registers, and the organization of the registers in the primary encoding spaces, (`coproc==0b1110`) and (`coproc==0b1111`) for these registers:

- *About the System registers for VMSAv8-32* on page G5-6052.
- *Functional grouping of VMSAv8-32 System registers* on page G5-6058.

Note

The System registers in the (`coproc==0b1110`) encoding space provide the following functionality:

- Self-hosted debug. These registers are described in *Debug registers* on page G8-6586.
- The System register interface to a PE Trace Unit These registers are not described in this manual.
- Jazelle registers. These registers are summarized in *Legacy feature registers and system instructions* on page K14-8092.

Therefore, there is no summary of these registers by functional groups.

Pseudocode description of VMSAv8-32 memory system operations on page G5-6050 then summarizes the pseudocode functions that describe many features of VMSAv8-32 operation.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G5.2 The effects of disabling address translation stages on VMSAv8-32 behavior

About VMSAv8-32 on page G5-5922 defines the translation regimes and the associated stages of address translation, each of which has its own System registers for control and configuration. VMSAv8-32 includes an enable bit for each stage of address translation, as follows:

- SCTLR.M, in the Secure instance of the register, controls Secure PL1&0 stage 1 address translation.
- SCTLR.M, in the Non-secure instance of the register, controls Non-secure PL1&0 stage 1 address translation.
- HCR.VM controls Non-secure PL1&0 stage 2 address translation.
- HSCTLR.M controls Non-secure EL2 stage 1 address translation.

Note

- The descriptions throughout this chapter describe address translation as seen by Exception levels that are using AArch32. However, for the Non-secure PL1&0 translation regime, the stage 2 translation:
  - Is controlled by the HCR if EL2 is using AArch32.
  - Is controlled by the HCR_EL2 if EL2 is using AArch64.

  For this reason, links to the HCR link to a table that disambiguates between the AArch32 HCR and the AArch64 HCR_EL2.
- If EL2 is using AArch64, then the equivalent of the Non-secure EL2 translation regime is described in Chapter D5 *The AArch64 Virtual Memory System Architecture*, not in this chapter.

The following sections describe the effect on VMSAv8-32 behavior of disabling each stage of translation:

- *VMSAv8-32 behavior when stage 1 address translation is disabled*.
- *VMSAv8-32 behavior when stage 2 address translation is disabled* on page G5-5932.
- *Behavior of instruction fetches when all associated address translations are disabled* on page G5-5932.

Enabling stages of address translation on page G5-5932 gives more information about each stage of address translation, in particular after a reset on an implementation that includes EL3.

G5.2.1 VMSAv8-32 behavior when stage 1 address translation is disabled

When stage 1 address translation is disabled, memory accesses that would otherwise be translated by that stage of address translation are treated as follows:

Non-secure PL1 and EL0 accesses when EL2 is implemented and HCR.DC is set to 1

In an implementation that includes EL2, for an access from a Non-secure PL1 or EL0 mode when HCR.DC is set to 1, the stage 1 translation assigns the Normal Non-shareable, Inner Write-Back Read-Allocate Write-Allocate, Outer Write-Back Read-Allocate Write-Allocate memory attributes.

See also *Effect of the HCR.DC field* on page G5-5931.

All other accesses

For all other accesses, when a stage 1 address translation is disabled, the assigned attributes depend on whether the access is a data access or an instruction access, as follows:

Data access

The stage 1 translation assigns the Device-nGnRnE memory type.

Instruction access

The stage 1 translation assigns Normal memory attribute, with the Cacheability and Shareability attributes determined by the value of:

- The Secure instance of SCTLR.I for the Secure PL1&0 translation regime.
- The Non-secure instance of SCTLR.I for the Non-secure PL1&0 translation regime.
- HSCTLR.I for the Non-secure EL2 translation regime.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In these cases, the meaning of the I field is as follows:

When I is set to 0

The stage 1 translation assigns the attributes Outer Shareable, Non-cacheable.

When I is set to 1

The stage 1 translation assigns the attributes Inner Write-Through Read-Allocate No Write-Allocate, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.

Note

On some implementations, if the SCTLR.TRE field is set to 0 then this behavior can be changed by the remap settings in the memory remap registers. The details of TEX remap when SCTLR.TRE is set to 0 are IMPLEMENTATION DEFINED, see *SCTLR.TRE, SCTLR.M, and the effect of the TEX remap registers* on page G5-5984.

For this stage of translation, no memory access permission checks are performed, and therefore no MMU faults relating to this stage of translation can be generated.

Note

Alignment checking is performed, and therefore Alignment faults can occur.

For every access, when stage 1 translation is disabled, the output address of the stage 1 translation is equal to the input address. This is called a flat address mapping. If the implementation supports output addresses of more than 32 bits then the output address bits above bit[31] are zero. For example, for a VA to PA translation on an implementation that supports 40-bit PAs, PA[39:32] is `0x00`.

For a Non-secure PL1 or EL0 access, if the PL1&0 stage 2 address translation is enabled, the stage 1 memory attribute assignments and output address can be modified by the stage 2 translation.

See also *Behavior of instruction fetches when all associated address translations are disabled* on page G5-5932.

Effect of the HCR.DC field

The HCR.DC field determines the default memory attributes assigned for the first stage of the Non-secure PL1&0 translation regime when that stage of translation is disabled.

When executing in a Non-secure PL1 or EL0 mode with HCR.DC set to 1:

- For all purposes other than reading the value of the SCTLR, the PE behaves as if the value of the SCTLR.M field is 0. This means Non-secure PL1&0 stage 1 address translation is disabled.
- For all purposes other than reading the value of the HCR, the PE behaves as if the value of the HCR.VM field is 1. This means Non-secure PL1&0 stage 2 address translation is enabled.

The effect of HCR.DC might be held in TLB entries associated with a particular VMID. Therefore, if software executing at EL2 changes the HCR.DC value without also changing the current VMID, it must also invalidate all TLB entries associated with the current VMID. Otherwise, the behavior of Non-secure software executing at EL1 or EL0 is CONSTRAINED UNPREDICTABLE, see *CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values* on page K1-7835.

Effect of disabling translation on maintenance and address translation instructions

Cache maintenance instructions act on the target cache whether address translation is enabled or not, and regardless of the values of the memory attributes. However, if a stage of translation is disabled, they use the flat address mapping for that stage, and all mappings are considered global.

TLB invalidate operations act on the target TLB whether address translation is enabled or not.

When the Non-secure PL1&0 stage 1 address translation is disabled, any ATS1C** or ATS12NSO** address translation instruction that accesses the Non-secure state translation reflects the effect of the HCR.DC field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G5.2.2 VMSAv8-32 behavior when stage 2 address translation is disabled

When stage 2 address translation is disabled:

- The IPA output from the stage 1 translation maps flat to the PA
- The memory attributes and permissions from the stage 1 translation apply to the PA.

If the stage 1 address translation and the stage 2 address translation are both disabled, see *Behavior of instruction fetches when all associated address translations are disabled*.

G5.2.3 Behavior of instruction fetches when all associated address translations are disabled

The information in this section applies to memory accesses:

- From Secure PL1 and EL0 modes, when the Secure PL1&0 stage 1 address translation is disabled
- From Hyp mode, when the Non-secure EL2 stage 1 address translation is disabled
- From Non-secure PL1 and EL0 modes, when all of the following apply:
  - The Non-secure PL1&0 stage 1 address translation is disabled.
  - The Non-secure PL1&0 stage 2 address translation is disabled.
  - HCR.DC is set to 0.

In these cases, when execution is in AArch32 state a memory location might be accessed as a result of an instruction fetch if either:

- The memory location is in the same 4KB block of memory, aligned to 4KB, as an instruction which a simple sequential execution of the program either requires to be fetched now or has required to be fetched since the last reset, or is in the 4KB block immediately following such a block.
- The memory location is the target of a direct branch that a simple sequential execution of the program would have taken since the most recent of:
  - The last reset.
  - If the branch predictor is architecturally invisible, the last synchronization of instruction cache maintenance targeting the address of the branch instruction.
  - If the branch predictor is not architecturally invisible, the last synchronization of branch predictor maintenance targeting the address of the branch instruction.

These accesses can be caused by speculative instruction fetches, regardless of whether the prefetched instruction is committed for execution.

Note

To ensure architectural compliance, software must ensure that both of the following apply:

- Instructions that will be executed when address translation is disabled are located in 4KB blocks of the address space that contain only memory that is tolerant to speculative accesses.
- Each 4KB block of the address space that immediately follows a 4KB block that holds instructions that will be executed when address translation is disabled also contains only memory that is tolerant to speculative accesses.

G5.2.4 Enabling stages of address translation

On powerup or reset, only the SCTLR.M field for the Exception level and Security state entered on reset is reset to 0, disabling address translation for the initial state of the PE. All other SCTLR.M and HSCTLR.M fields that are implemented are UNKNOWN after the reset.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This means, on powerup or reset:

- On an implementation that includes EL3, where EL3 is using AArch32:
  - The PL1&0 stage 1 address translation enable bit, SCTLR.M, is banked, meaning there are separate enables for operation in Secure and Non-secure state.
  - If EL3 is using AArch32, only the Secure instance of the SCTLR.M field resets to 0, disabling the Secure state PL1&0 stage 1 address translation. The reset value of the Non-secure instance of SCTLR.M is UNKNOWN.
- On an implementation that includes EL2, where EL2 is using AArch32, the HSCTLR.M field, that controls the Non-secure EL2 stage 1 address translation:
  - If the implementation does not include EL3, resets to 0.
  - Otherwise, is UNKNOWN.
- On an implementation that does not include either EL2 or EL3, there is a single stage of translation. This is controlled by SCTLR.M, that resets to 0.

Note

If, for the software that enables or disables a stage of address translation, the input address of a stage 1 translation differs from the output address of that stage 1 translation, and the software is running in translation regime that is affected by that stage of translation, then the requirement to synchronize changes to the System registers means it is uncertain where in the instruction stream the change of the translation takes place. For this reason, Arm strongly recommends that the input address and the output address are identical in this situation.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G5.3 Translation tables

VMSAv8-32 defines two alternative translation table formats:

Short-descriptor format

It uses 32-bit descriptor entries in the translation tables, and provides:

- Up to two levels of address lookup.
- 32-bit input addresses.
- Output addresses of up to 40 bits.
- Support for PAs of more than 32 bits by use of supersections, with 16MB granularity.
- Support for No access, Client, and Manager domains.

Long-descriptor format

It uses 64-bit descriptor entries in the translation tables, and provides:

- Up to three levels of address lookup.
- Input addresses of up to 40 bits, when used for stage 2 translations.
- Output addresses of up to 40 bits.
- 4KB assignment granularity across the entire PA range.
- No support for domains, all memory regions are treated as in a Client domain.
- Fixed 4KB table size, unless truncated by the size of the input address space.

Note

— Translation with a 40-bit input address range requires two concatenated 4KB top-level tables, aligned to 8KB.

— The VMSAv8-64 Long-descriptor translation table format is generally similar to this format, but supports input and output addresses of up to 48 bits, and has an assignment granularity and table size defined by its *translation granule*. This can be 4KB, 16KB, or 64KB. See *The VMSAv8-64 translation table format* on page D5-2575.

In all implementations, of the possible address translations shown in Figure G5-2 on page G5-5927, for stages of address translation that are using AArch32:

- In a particular Security state, the translation tables for the PL1&0 stage 1 translations can use either translation table format, and the TTBCR.EAE field indicates the current translation table format.
- The translation tables for the Non-secure EL2 stage 1 translations, and for the Non-secure PL1&0 stage 2 translations, must use the Long-descriptor translation table format.

Many aspects of performing a translation table walk depend on the current translation table format. Therefore, the following sections describe the two formats, including how the MMU performs a translation table walk for each format:

- *The VMSAv8-32 Short-descriptor translation table format* on page G5-5939.
- *The VMSAv8-32 Long-descriptor translation table format* on page G5-5948.

The following subsections describe aspects of the translation tables and translation table walks, for memory accesses from AArch32 state, that are independent of the translation table format:

- *Translation table walks for memory accesses using VMSAv8-32 translation regimes* on page G5-5935.
- *Information returned by a translation table lookup* on page G5-5935.
- *Determining the translation table base address in the VMSAv8-32 translation regimes* on page G5-5936.
- *Control of translation table walks on a TLB miss* on page G5-5937.
- *Access to the Secure or Non-secure PA map* on page G5-5937.

See also *TLB maintenance requirements* on page G5-5995.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G5.3.1 Translation table walks for memory accesses using VMSAv8-32 translation regimes

A translation table walk occurs as the result of a TLB miss, and starts with a read of the appropriate starting-level translation table. The result of that read determines whether additional translation table reads are required, for this stage of translation, as described in either:

- *Translation table walks, when using the VMSAv8-32 Short-descriptor translation table format* on page G5-5945.
- *Translation table walks, when using the VMSAv8-32 Long-descriptor translation table format* on page G5-5963.

Note

When using the Short-descriptor translation table format, the starting level for a translation table walk is always a level 1 lookup. However, with the Long-descriptor translation table format, the starting-level can be either a level 1or a level 2 lookup.

For the PL1&0 stage 1 translations, SCTLR.EE determines the endianness of the translation table lookups. SCTLR is banked, and therefore the endianness is determined independently for each Security state.

HSCTLR.EE defines the endianness for the Non-secure EL2 stage 1 and Non-secure PL1&0 stage 2 translations.

Note

Dynamically changing translation table endianness

Because any change to SCTLR.EE or HSCTLR.EE requires synchronization before it is visible to subsequent operations, Arm strongly recommends that:

- SCTLR.EE is changed only when either:
  - Executing in a mode that does not use the translation tables affected by SCTLR.EE.
  - Executing with SCTLR.M set to 0.
- HSCTLR.EE is changed only when either:
  - Executing in a mode that does not use the translation tables affected by HSCTLR.EE.
  - Executing with HSCTLR.M set to 0.

The PA of the base of the starting-level translation table is determined from the appropriate TTBR, see *Determining the translation table base address in the VMSAv8-32 translation regimes* on page G5-5936.

For more information, see *Ordering and completion of TLB maintenance instructions* on page G5-5998.

Translation table walks must access data or unified caches, or data and unified caches, of other agents participating in the coherency protocol, according to the Shareability attributes described in the TTBR. These Shareability attributes must be consistent with the Shareability attributes for the translation tables themselves.

G5.3.2 Information returned by a translation table lookup

When an associated stage of address translation is enabled, a memory access requires one or more translation table lookups. If the required translation table descriptor is not held in a TLB, a translation table walk is performed to obtain the descriptor. A lookup, whether from the TLB or as the result of a translation table walk, returns both:

- An output address that corresponds to the input address for the lookup.
- A set of properties that correspond to that output address.

The returned properties are classified as providing *address map control*, *access controls*, or *region attributes*. This classification determines how the descriptions of the properties are grouped. The classification is based on the following model:

Address map control

Memory accesses from Secure state can access either the Secure or the Non-secure address map, as summarized in *Access to the Secure or Non-secure PA map* on page G5-5937.

Memory accesses from Non-secure state can only access the Non-secure address map.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Access controls

Determine whether the PE, in its current state, can access the output address that corresponds to the given input address. If not, an MMU fault is generated and there is no memory access.

Memory access control on page G5-5968 describes the properties in this group.

Attributes Are valid only for an output address that the PE, in its current state, can access. The attributes define aspects of the required behavior of accesses to the target memory region.

Memory region attributes on page G5-5979 describes the properties in this group.

G5.3.3 Determining the translation table base address in the VMSAv8-32 translation regimes

On a TLB miss, the VMSA must perform a translation table walk, and therefore must find the base address of the translation table to use for its lookup. A TTBR holds this address. As Figure G5-2 on page G5-5927 shows:

- For a Non-secure EL2 stage 1 translation, the HTTBR holds the required base address. The HTCR is the control register for these translations.
- For a Non-secure PL1&0 stage 2 translation, the VTTBR holds the required base address. The VTCR is the control register for these translations.
- For a PL1&0 stage 1 translation, either TTBR0 or TTBR1 holds the required base address. The TTBCR is the control register for these translations.

  The Non-secure copies of TTBR0, TTBR1, and TTBCR, relate to the Non-secure PL1&0 stage 1 translation. The Secure copies of TTBR0, TTBR1, and TTBCR, relate to the Secure PL1&0 stage 1 translation.

For the PL1&0 translation table walks:

- TTBR0 can be configured to describe the translation of VAs in the entire address map, or to describe only the translation of VAs in the lower part of the address map.
- If TTBR0 is configured to describe the translation of VAs in the lower part of the address map, TTBR1 is configured to describe the translation of VAs in the upper part of the address map.

The contents of the appropriate instance of the TTBCR determine whether the address map is separated into two parts, and where the separation occurs. The details of the separation depend on the current translation table format, see:

- *Selecting between TTBR0 and TTBR1, VMSAv8-32 Short-descriptor translation table format* on page G5-5944.
- *Selecting between TTBR0 and TTBR1, VMSAv8-32 Long-descriptor translation table format* on page G5-5957.

Example G5-1 shows a typical use of the two sets of translation tables:

Example G5-1 Example use of TTBR0 and TTBR1

An example of using the two TTBRs for PL1&0 stage 1 address translations is:

TTBR0 Used for process-specific addresses.

Each process maintains a separate level 1 translation table. On a context switch:

- TTBR0 is updated to point to the level 1 translation table for the new context.
- TTBCR is updated if this change changes the size of the translation table.
- The CONTEXTIDR is updated.

TTBCR can be programmed so that all translations use TTBR0 in a manner compatible with architecture versions before Armv6.

TTBR1 Used for operating system and I/O addresses, that do not change on a context switch.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G5.3.4 Control of translation table walks on a TLB miss

Two fields in the TCR for the translation stage required by a memory access control whether a translation table walk is performed on a TLB miss. These two fields are the:

- PD0 and PD1 fields, on a PE using the Short-descriptor translation table format.
- EPD0 and EPD1 fields, on a PE using the Long-descriptor translation table format.

Note

For the VMSAv8-32 translation regimes, the different field names are because the fields are in different positions in TTBCR, depending on the translation table format.

The effect of these fields is:

{E}PDx == 0 If a TLB miss occurs based on TTBRx, a translation table walk is performed. The current Security state determines whether the memory access is Secure or Non-secure.

{E}PDx == 1 If a TLB miss occurs based on TTBRx, a level 1 Translation fault is returned, and no translation table walk is performed.

G5.3.5 Access to the Secure or Non-secure PA map

As stated in *Address spaces in VMSAv8-32* on page G5-5925, a PE can access independent Secure and Non-secure address maps. When the PL1 Exception level is using AArch32, these are defined by the translation tables identified by the Secure TTBR0 and TTBR1. In both translation table formats in the Secure translation tables, the NS field in a descriptor indicates whether the descriptor refers to the Secure or the Non-secure address map:

NS == 0 Access the Secure PA space.

NS == 1 Access the Non-secure PA space.

Note

In the Non-secure translation tables, the corresponding field is SBZ. Non-secure accesses always access the Non-secure PA space, regardless of the value of this field.

The Long-descriptor translation table format extends this control, adding an NSTable field to the Secure translation tables, as described in *Hierarchical control of Secure or Non-secure memory accesses, Long-descriptor format* on page G5-5956. In the Non-secure translation tables, the corresponding field is SBZ, and Non-secure accesses ignore the value of this field.

The following sections describe the address map controls in the two implementations:

- *Control of Secure or Non-secure memory access, VMSAv8-32 Short-descriptor format* on page G5-5944.
- *Control of Secure or Non-secure memory access, VMSAv8-32 Long-descriptor format* on page G5-5956.

The following subsection gives more information.

Secure and Non-secure address spaces

EL3 provides two PA spaces, a Secure PA space and a Non-secure PA space.

As described in *Access to the Secure or Non-secure PA map*, for the PL1&0 stage 1 translations when controlled from an Exception level using AArch32, the registers that control the stage of translation, TTBR0, TTBR1, TTBCR, and TTBCR2 are banked to provide independent Secure and Non-secure instances of the registers, and the Security state of the PE when it performs a memory access whether the Secure or Non-secure instances are used. This means that for stage 1 of the PL1&0 translation regime there are independent Secure and Non-secure translation tables, and translation table walks are made to the PA space corresponding to the Security state of the translation tables used.

For a translation table walk caused by a memory access from Non-secure state, all memory accesses are to the Non-secure address space.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For a translation table walk caused by a memory access from Secure state:

- When address translation is using the Long-descriptor translation table format:
  - The initial lookup performed must access the Secure address space.
  - If a table descriptor read from the Secure address space has the NSTable field set to 0, then the next level of lookup is from the Secure address space.
  - If a table descriptor read from the Secure address space has the NSTable field set to 1, then the next level of lookup, and any subsequent level of lookup, is from the Non-secure address space.

  For more information, see *Control of Secure or Non-secure memory access, VMSAv8-32 Long-descriptor format* on page G5-5956.

- Otherwise, all memory accesses are to the Secure address space.

Note

- When executing in Non-secure state, additional translations are supported. For memory accesses from AArch32 state, these are:
  - Non-secure EL2 stage 1 translation.
  - Non-secure PL1&0 stage 2 translation.

  These translations can access only the Non-secure address space.

- A system implementation can alias parts of the Secure PA space to the Non-secure PA space in an implementation-specific way. As with any other aliasing of physical memory, the use of aliases in this way can require the use of cache maintenance instructions to ensure that changes to memory made using one alias of the physical memory are visible to accesses to the other alias of the physical memory.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G5.4 The VMSAv8-32 Short-descriptor translation table format

The Short-descriptor translation table format supports a memory map based on memory sections or pages:

Supersections Consist of 16MB blocks of memory. Support for Supersections is optional, except that an implementation that supports more than 32 bits of PA must also support Supersections to provide access to the entire PA space.

Sections Consist of 1MB blocks of memory.

Large pages Consist of 64KB blocks of memory.

Small pages Consist of 4KB blocks of memory.

Supersections, Sections, and Large pages map large regions of memory using only a single TLB entry.

Note

- Whether a VMSAv8-32 implementation of the Short-descriptor format translation tables supports supersections is IMPLEMENTATION DEFINED.
- The EL2 translation regime cannot use the Short-descriptor translation table format.

When using the Short-descriptor translation table format, two levels of translation tables are held in memory:

Level 1 table

Holds *level 1 descriptors* that contain the base address and

- Translation properties for a Section and Supersection.
- Translation properties and pointers to a level 2 table for a Large page or a Small page.

Level 2 tables

Hold *level 2 descriptors* that contain the base address and translation properties for a Small page or a Large page. With the Short-descriptor format, level 2 tables can be referred to as *translation tables*.

A level 2 table requires 1KB of memory.

In the translation tables, in general, a descriptor is one of:

- An invalid or fault entry.
- A translation table entry, that points to a next-level translation table.
- A page or section entry, that defines the memory properties for the access.
- A reserved format.

Bits[1:0] of the descriptor give the primary indication of the descriptor type.

Figure G5-3 on page G5-5940 gives a general view of address translation when using the Short-descriptor translation table format.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

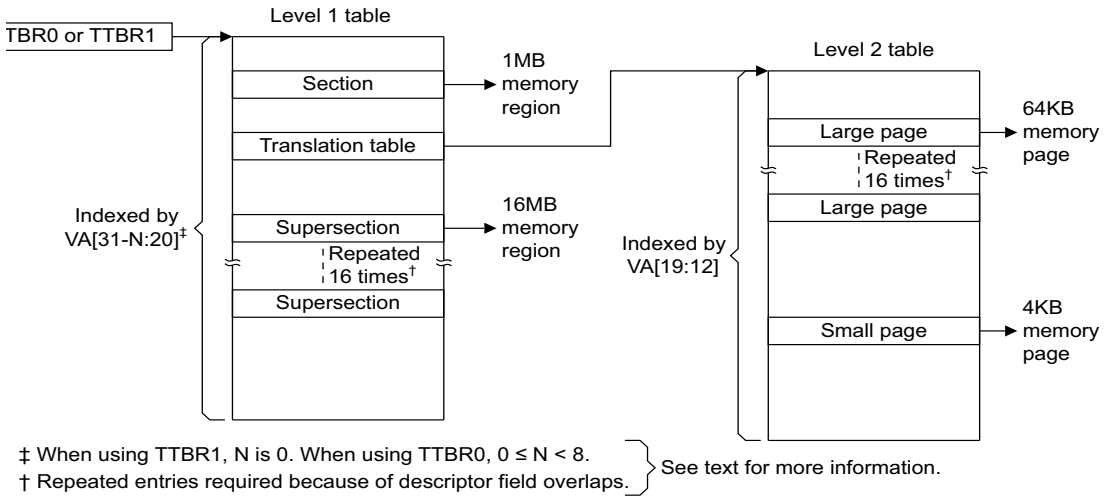


Figure G5-3 General view of address translation using VMSAv8-32 Short-descriptor format translation tables

Additional requirements for Short-descriptor format translation tables on page G5-5943 describes why, when using the Short-descriptor format, Supersection and Large page entries must be repeated 16 times, as shown in Figure G5-3.

VMSAv8-32 Short-descriptor translation table format descriptors, *Memory attributes in the VMSAv8-32 Short-descriptor translation table format descriptors* on page G5-5943, and *Control of Secure or Non-secure memory access, VMSAv8-32 Short-descriptor format* on page G5-5944 describe the format of the descriptors in the Short-descriptor format translation tables.

The following sections then describe the use of this translation table format:

- *Selecting between TTBR0 and TTBR1, VMSAv8-32 Short-descriptor translation table format* on page G5-5944.
- *Translation table walks, when using the VMSAv8-32 Short-descriptor translation table format* on page G5-5945.

G5.4.1 VMSAv8-32 Short-descriptor translation table format descriptors

The following sections describe the formats of the entries in the Short-descriptor translation tables:

- *Short-descriptor translation table level 1 descriptor formats*.
- *Short-descriptor translation table level 2 descriptor formats* on page G5-5942.

For more information about level 2 translation tables, see *Additional requirements for Short-descriptor format translation tables* on page G5-5943.

Note

Previous versions of the *Arm Architecture Reference Manual*, and some other documentation, describes the AP[2] bit in the translation table entries as the APX bit.

Information returned by a translation table lookup on page G5-5935 describes the classification of the non-address fields in the descriptors as address map control, access control, or attribute fields.

Short-descriptor translation table level 1 descriptor formats

Each entry in the level 1 table describes the mapping of the associated 1MB VA range.

Figure G5-4 on page G5-5941 shows the possible level 1 descriptor formats.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

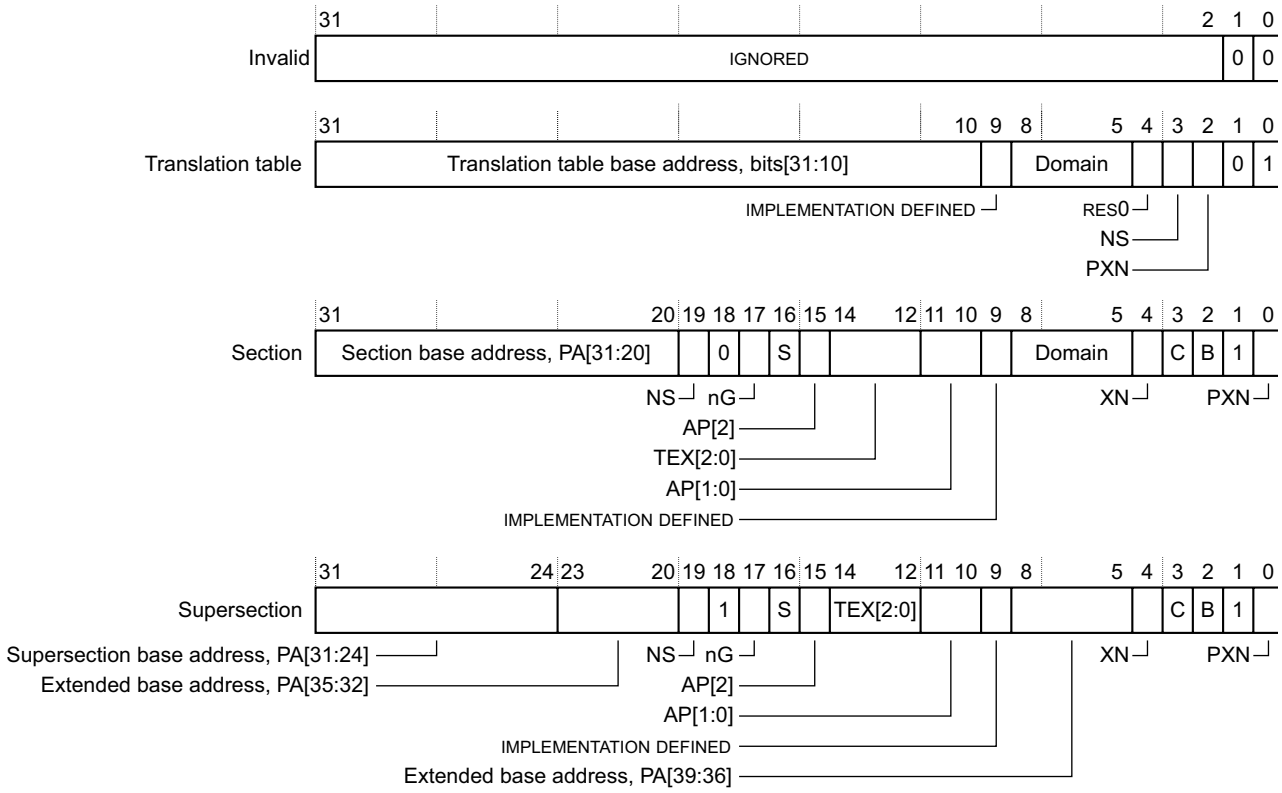


Figure G5-4 VMSAv8-32 Short-descriptor level 1 descriptor formats

Descriptor bits[1:0] identify the descriptor type. The encoding of these bits is:

`0b00`**, Invalid entry**

The associated VA is unmapped, and any attempt to access it generates a Translation fault.

Bits[31:2] of the descriptor are IGNORED, see *IGNORED* on page Glossary-8115. This means software can use these bits for its own purposes.

`0b01`**, Translation table**

The descriptor gives the address of a level 2 translation table, that specifies the mapping of the associated 1MByte VA range.

`0b10`**, Section or Supersection**

The descriptor gives the base address of the Section or Supersection. Bit[18] determines whether the entry describes a Section or a Supersection.

This encoding also defines the PXN field as 0.

`0b11`**, Section or Supersection, if the implementation supports the PXN attribute**

This encoding is identical to `0b10`, except that it defines the PXN field as 1.

Note

A VMSAv8-32 implementation can use the Short-descriptor translation table format for the PL1&0 stage 1 translations, by setting TTBCR.EAE to 0.

The address information in the level 1 descriptors is:

Translation table Bits[31:10] of the descriptor are bits[31:10] of the address of a translation table.

Section Bits[31:20] of the descriptor are bits[31:20] of the address of the Section.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Supersection Bits[31:24] of the descriptor are bits[31:24] of the address of the Supersection.
Optionally, bits[8:5, 23:20] of the descriptor are bits[39:32] of the extended Supersection address.

For the Non-secure PL1&0 translation tables, the address in the descriptor is the IPA of the translation table, Section, or Supersection. Otherwise, the address is the PA of the translation table, Section, or Supersection.

For descriptions of the other fields in the descriptors, see *Memory attributes in the VMSAv8-32 Short-descriptor translation table format descriptors* on page G5-5943.

Short-descriptor translation table level 2 descriptor formats

Figure G5-5 shows the possible formats of a level 2 descriptor.

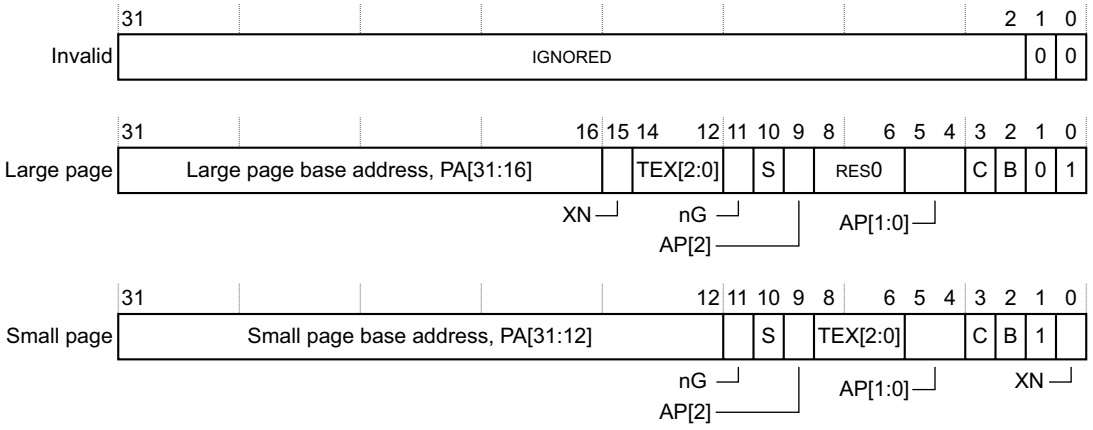


Figure G5-5 Short-descriptor level 2 descriptor formats

Descriptor bits[1:0] identify the descriptor type. The encoding of these bits is:

`0b00`**, Invalid entry**

The associated VA is unmapped, and attempting to access it generates a Translation fault.

Bits[31:2] of the descriptor are IGNORED, see *IGNORED* on page Glossary-8115. This means software can use these bits for its own purposes.

`0b01`**, Large page**

The descriptor gives the base address and properties of the Large page.

`0b1x`**, Small page**

The descriptor gives the base address and properties of the Small page.

In this descriptor format, bit[0] of the descriptor is the XN field.

The address information in the level 2 descriptors is:

Large page Bits[31:16] of the descriptor are bits[31:16] of the address of the Large page.

Small page Bits[31:12] of the descriptor are bits[31:12] of the address of the Small page.

For the Non-secure PL1&0 translation tables, the address in the descriptor is the IPA of the translation table, Section, or Supersection. Otherwise, the address is the PA of the translation table, Section, or Supersection.

For descriptions of the other fields in the descriptors, see *Memory attributes in the VMSAv8-32 Short-descriptor translation table format descriptors* on page G5-5943.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Additional requirements for Short-descriptor format translation tables

When using Supersection or Large page descriptors in the Short-descriptor translation table format, the input address field that defines the Supersection or Large page descriptor address overlaps the table address field. In each case, the size of the overlap is 4 bits. The following diagrams show these overlaps:

- Figure K7-14 on page K7-7939 for the level 1 translation table entry for a Supersection.
- Figure K7-16 on page K7-7941 for the level 2 translation table entry for a Large page.

Considering the case of using Large page descriptors in a level 2 translation table, this overlap means that for any specific Large page, the bottom four bits of the level 2 translation table entry might take any value from `0b0000` to `0b1111`. Therefore, each of these 16 index values must point to a separate copy of the same descriptor.

This means that each Large page or Supersection descriptor must:

- Occur first on a sixteen-word boundary.
- Be repeated in 16 consecutive memory locations.

G5.4.2 Memory attributes in the VMSAv8-32 Short-descriptor translation table format descriptors

This section describes the descriptor fields other than the descriptor type field and the address field:

TEX[2:0], C, B

Memory region attribute fields, see *Memory region attributes* on page G5-5979.

These fields are not present in a descriptor for a translation table.

XN bit

The Execute-never field, see *Access permissions for instruction execution* on page G5-5972.

This bit is not present in a descriptor for a translation table.

PXN bit

The Privileged execute-never field, see *Access permissions for instruction execution* on page G5-5972.

When this field is set to 1 in the descriptor for a translation table, it indicates that all memory pages described in the corresponding translation table are Privileged execute-never.

NS bit

Non-secure bit. Specifies whether the translated PA is in the Secure or Non-secure address map, see *Control of Secure or Non-secure memory access, VMSAv8-32 Short-descriptor format* on page G5-5944.

This bit is not present in level 2 descriptors. The value of the NS bit in a level 1 descriptor for a translation table applies to all entries in the corresponding level 2 translation table.

Domain

Domain field, see *Domains, Short-descriptor format only* on page G5-5976.

This field is not present in a Supersection entry. Memory described by Supersections is in domain 0.

This bit is not present in level 2 descriptors. The value of the Domain field in the level 1 descriptor for a translation table applies to all entries in the corresponding level 2 translation table.

An IMPLEMENTATION DEFINED bit

This bit is not present in level 2 descriptors.

AP[2], AP[1:0]

Access Permissions bits, see *Memory access control* on page G5-5968.

AP[0] can be configured as the *Access flag*, see *The Access flag* on page G5-5976.

These bits are not present in a descriptor for a translation table.

S bit

Shareable bit. Used in determining the Shareability of the addressed region, see *Memory region attributes* on page G5-5979.

Note

The naming of this bit as the *Shareable* bit is carried forward from early versions of the Arm architecture. This name is no longer an adequate description of the interpretation of the bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This bit is not present in a descriptor for a translation table.

nG bit The not global bit. If a lookup using this descriptor is cached in a TLB, determines whether the TLB entry applies to all ASID values, or only to the current ASID value. See *Global and process-specific translation table entries* on page G5-5991.

This bit is not present in a descriptor for a translation table.

Bit[18], when bits[1:0] indicate a *Section or Supersection* descriptor

0 Descriptor is for a Section.

1 Descriptor is for a Supersection.

G5.4.3 Control of Secure or Non-secure memory access, VMSAv8-32 Short-descriptor format

Access to the Secure or Non-secure PA map on page G5-5937 describes how the NS bit in the translation table entries:

- For accesses from Secure state, determines whether the access is to Secure or Non-secure memory.
- Is ignored by accesses from Non-secure state.

In the Short-descriptor translation table format, the NS bit is defined only in the level 1 translation tables. This means that, in a level 1 descriptor for a translation table, the NS bit defines the PA map, Secure or Non-secure, for all of the Large pages and Small pages of memory described by that table.

The NS bit of a level 1 descriptor for a translation table has no effect on the PA map in which that translation table is held. As stated in *Secure and Non-secure address spaces* on page G5-5937, the PA of that translation table is in:

- The Secure address map if the translation table walk is in Secure state.
- The Non-secure address map if the translation table walk is in Non-secure state.

This means the granularity of the Secure and Non-secure memory maps is 1MB. However, in these memory maps, table entries can define physical memory regions with a granularity of 4KB.

G5.4.4 Selecting between TTBR0 and TTBR1, VMSAv8-32 Short-descriptor translation table format

As described in *Determining the translation table base address in the VMSAv8-32 translation regimes* on page G5-5936, two sets of translation tables can be defined for each of the PL1&0 stage 1 translations, and TTBR0 and TTBR1 hold the base addresses for the two sets of tables. When using the Short-descriptor translation table format, the value of TTBCR.N indicates the number of most significant bits of the input VA that determine whether TTBR0 or TTBR1 holds the required translation table base address, as follows:

- If N == 0 then use TTBR0. Setting TTBCR.N to zero disables use of a second set of translation tables.
- If N > 0 then:
  - If bits[31:32-N] of the input VA are all zero, then use TTBR0.
  - Otherwise use TTBR1.

Table G5-1 shows how the value of N determines the lowest address translated using TTBR1, and the size of the level 1 translation table addressed by TTBR0.

Table G5-1 Effect of TTBCR.N on address translation, Short-descriptor format

TTBCR.N	First address translated with TTBR1	TTBR0 table	
		Size	Index range
`0b000`	TTBR1 not used	16KB	VA[31:20]
`0b001`	`0x80000000`	8KB	VA[30:20]
`0b010`	`0x40000000`	4KB	VA[29:20]
`0b011`	`0x20000000`	2KB	VA[28:20]
`0b100`	`0x10000000`	1KB	VA[27:20]

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G5-1 Effect of TTBCR.N on address translation, Short-descriptor format (continued)

TTBCR.N	First address translated with TTBR1	TTBR0 table	
		Size	Index range
0b101	0x08000000	512 bytes	VA[26:20]
0b110	0x04000000	256 bytes	VA[25:20]
0b111	0x02000000	128 bytes	VA[24:20]

Whenever TTBCR.N is nonzero, the size of the translation table addressed by TTBR1 is 16KB.

Figure G5-6 shows how the value of TTBCR.N controls the boundary between VAs that are translated using TTBR0, and VAs that are translated using TTBR1.

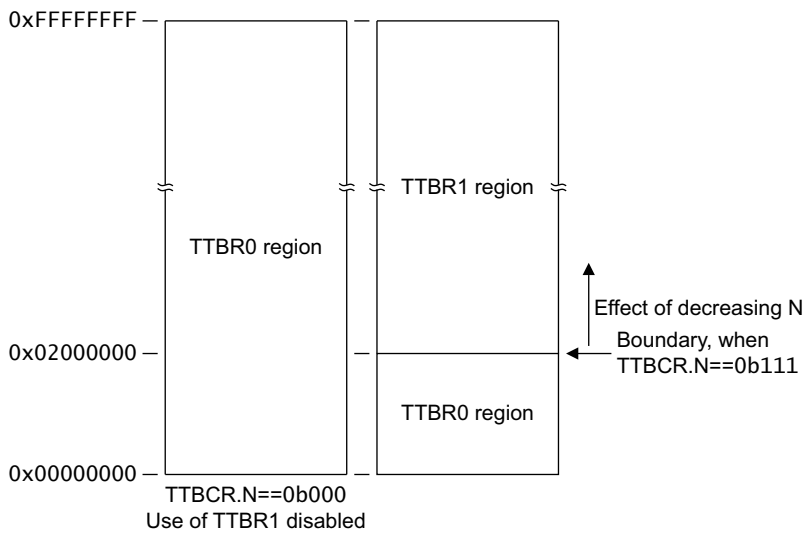


Figure G5-6 How TTBCR.N controls the boundary between the TTBRs, Short-descriptor format

In the selected TTBR, bits RGN, S, and IRGN[1:0] define the memory region attributes for the translation table walk.

Translation table walks, when using the VMSAv8-32 Short-descriptor translation table format describes the translation.

G5.4.5 Translation table walks, when using the VMSAv8-32 Short-descriptor translation table format

When using the Short-descriptor translation table format, and a memory access requires a translation table walk:

- A section-mapped access only requires a read of the level 1 translation table.
- A page-mapped access also requires a read of the level 2 translation table.

Reading a level 1 translation table on page G5-5946 describes how either TTBR1 or TTBR0 is used, with the accessed VA, to determine the address of the level 1 descriptor.

Reading a level 1 translation table on page G5-5946 shows the output address as A[39:0]:

- For a Non-secure PL1&0 stage 1 translation, this is the IPA of the required descriptor. A Non-secure PL1&0 stage 2 translation of this address is performed to obtain the PA of the descriptor.
- Otherwise, this address is the PA of the required descriptor.

The full translation flow for Sections, Supersections, Small pages and Large pages on page G5-5946 then shows the complete translation flow for each valid memory access.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Reading a level 1 translation table

When performing a fetch based on TTBR0:

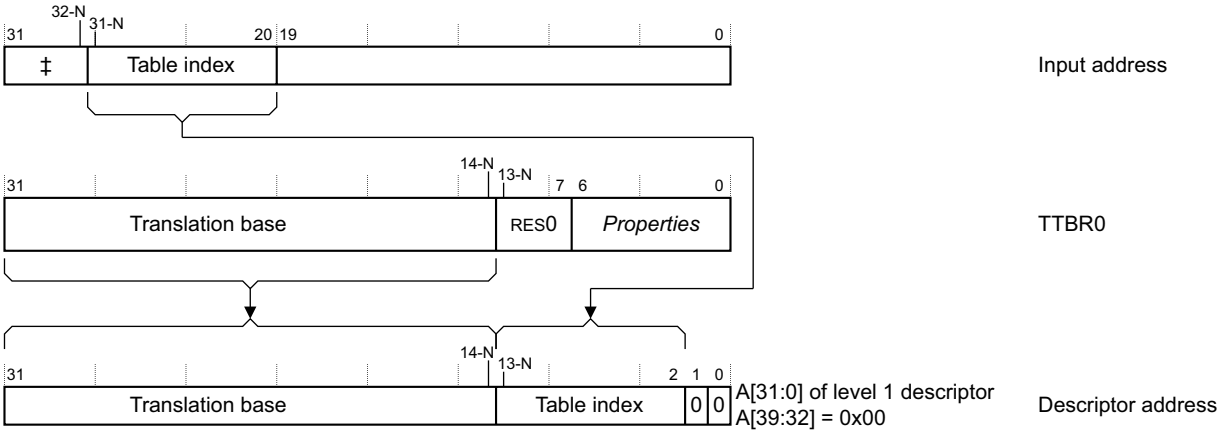
- The address bits taken from TTBR0 vary between bits[31:14] and bits[31:7].
- The address bits taken from the VA, that is the input address for the translation, vary between bits[31:20] and bits[24:20].

The width of the TTBR0 and VA fields depend on the value of TTBCR.N, as Figure G5-7 shows.

When performing a fetch based on TTBR1, Bits TTBR1[31:14] are concatenated with bits[31:20] of the VA. This makes the fetch equivalent to that shown in Figure G5-7, with N==0.

Note

See *The address and Properties fields shown in the translation flows* on page K7-7942 for more information about the *Properties* label used in this and other figures.



‡ This field is absent if N is 0
N is the value of TTBCR.N
For details of the *Properties* field, see the register description

Figure G5-7 Accessing level 1 translation table based on TTBR0, Short-descriptor format

Regardless of which register is used as the base for the fetch, the resulting output address selects a four-byte translation table entry that is one of:

- A level 1 descriptor for a Section or Supersection.
- A descriptor for a *translation table*, that points to a level 2 translation table. In this case:
  - A second fetch is performed to retrieve a level 2 descriptor.
  - The descriptor also contains some attributes for the access, see Figure G5-4 on page G5-5941.
- A faulting entry.

The full translation flow for Sections, Supersections, Small pages and Large pages

In a translation table walk, only the initial lookup uses the translation table base address from the appropriate TTBR. Subsequent lookups use a combination of address information from:

- The table descriptor read in the previous lookup.
- The input address.

Address translation examples using the VMSAv8-32 Short descriptor translation table format on page K7-7938 shows the full translation flow for each of the memory section and page options. As described in *VMSAv8-32 Short-descriptor translation table format descriptors* on page G5-5940, these options are:

Supersection A 16MB memory region, see *Translation flow for a Supersection* on page K7-7938.

Section A 1MB memory region, see *Translation flow for a Section* on page K7-7939.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Large page A 64KB memory region, described by the combination of:

- A level 1 translation table entry that indicates the address of a level 2 translation table.
- A level 2 descriptor that indicates a Large page.

See *Translation flow for a Large page* on page K7-7940.

Small page A 4KB memory region, described by the combination of:

- A level 1 translation table entry that indicates the address of a level 2 translation table.
- A level 2 descriptor that indicates a Small page.

See *Translation flow for a Small page* on page K7-7941.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G5.5 The VMSAv8-32 Long-descriptor translation table format

The VMSAv8-32 Long-descriptor translation table format supports the assignment of memory attributes to memory *Pages,* at a granularity of 4KB, across the complete input address range. It also supports the assignment of memory attributes to *blocks* of memory, where a block can be 2MB or 1GB.

Note

- Although the VMSAv8-32 Long-descriptor format is limited to three levels of address lookup, its design and naming conventions support extension to additional levels, to support a larger input address range.
- Similarly, while the VMSAv8-32 implementation limits the output address range to 40 bits, its design supports extension to a larger output address range.

Figure G5-2 on page G5-5927 shows the different address translation stages. The Long-descriptor translation table format:

- Is used for:
  - The Non-secure EL2 stage 1 translation.
  - The Non-secure PL1&0 stage 2 translation.
- Can be used for the Secure and Non-secure PL1&0 translations.

When used for a stage 1 translation, the translation tables support an input address of up to 32 bits, corresponding to the VA address range of the PE.

When used for a stage 2 translation, the translation tables support an input address range of up to 40 bits, to support the translation from IPA to PA. If the input address for the stage 2 translation is a 32-bit address, then this address is zero-extended to 40 bits.

Note

When the Short-descriptor translation table format is used for the Non-secure stage 1 translations, this generates 32-bit IPAs. These are zero-extended to 40 bits to provide the input address for the stage 2 translation.

Overview of VMSAv8-32 address translation using Long-descriptor translation tables summarizes address translation from AArch32 state when using the Long-descriptor format translation tables.

The following sections then describe the format of the descriptors in the Long-descriptor format translation tables:

- *VMSAv8-32 Long-descriptor translation table format descriptors* on page G5-5949.
- *Attribute fields in VMSAv8-32 Long-descriptor translation table format descriptors* on page G5-5952.
- *Control of Secure or Non-secure memory access, VMSAv8-32 Long-descriptor format* on page G5-5956.

The following sections then describe this translation table format:

- *Selecting between TTBR0 and TTBR1, VMSAv8-32 Long-descriptor translation table format* on page G5-5957.
- *VMSAv8-32 Long-descriptor translation table format address lookup levels* on page G5-5959.
- *Translation table walks, when using the VMSAv8-32 Long-descriptor translation table format* on page G5-5963.
- *The algorithm for finding the translation table entries, VMSAv8-32 Long-descriptor format* on page G5-5966.

G5.5.1 Overview of VMSAv8-32 address translation using Long-descriptor translation tables

Figure G5-8 on page G5-5949 gives a general view of VMSAv8-32 stage 1 address translation when using the Long-descriptor translation table format.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

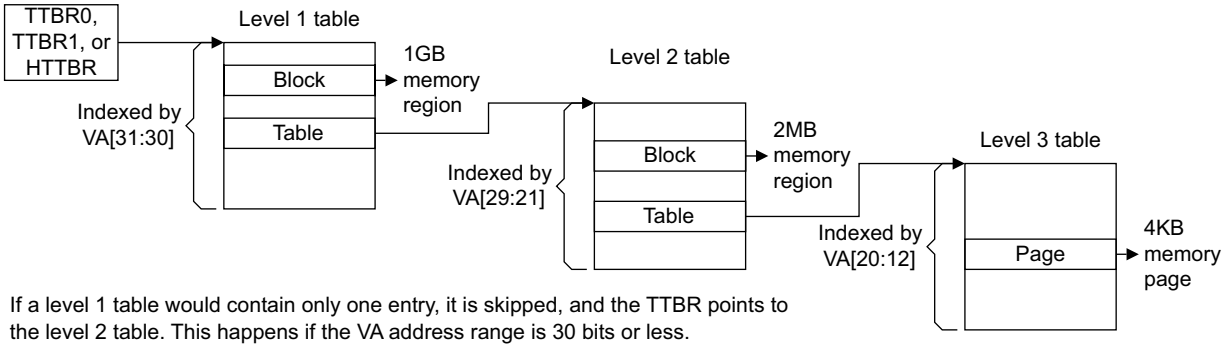


Figure G5-8 General view of VMSAv8-32 stage 1 address translation using Long-descriptor format

Figure G5-9 gives a general view of VMSAv8-32 stage 2 address translation. Stage 2 translation always uses the Long-descriptor translation table format.

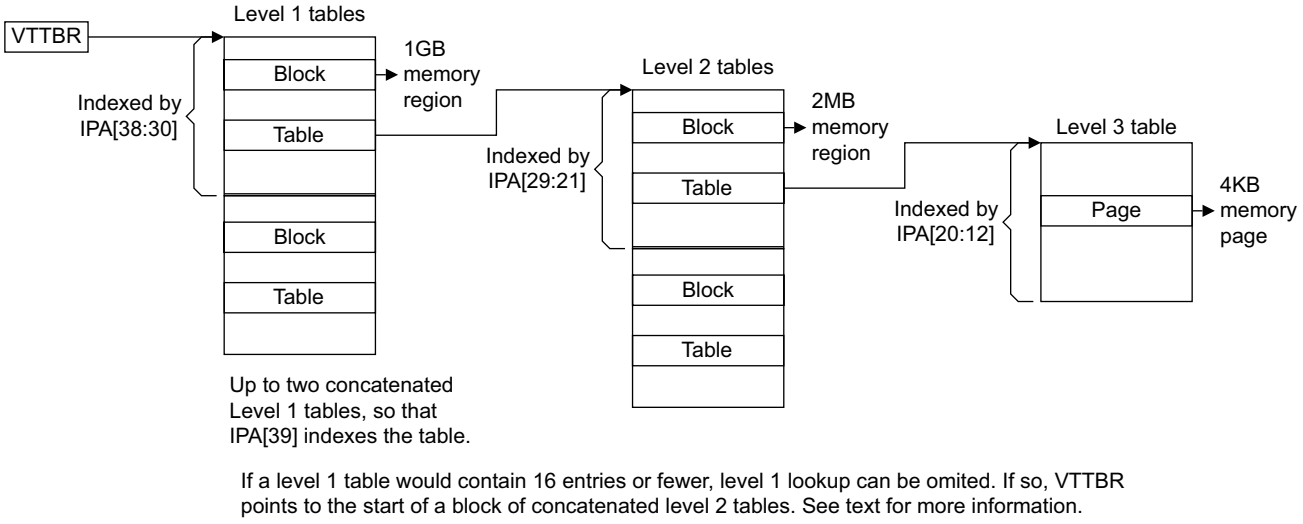


Figure G5-9 General view of VMSAv8-32 stage 2 address translation, Long-descriptor translation table format

Use of concatenated translation tables for the initial stage 2 lookup on page G5-5960 describes how using concatenated level 2 tables means lookup can start at level 2, as referred to in Figure G5-9.

G5.5.2 VMSAv8-32 Long-descriptor translation table format descriptors

As described in *VMSAv8-32 Long-descriptor translation table format address lookup levels* on page G5-5959, the Long-descriptor translation table format provides up to three levels of address lookup. A translation table walk starts either at level 1 or level 2 of the address lookup.

In general, a descriptor is one of:

- An invalid or fault entry.
- A table entry, that points to the next-level translation table.
- A block entry, that defines the memory properties for the access.
- A reserved format.

Bit[1] of the descriptor indicates the descriptor type, and bit[0] indicates whether the descriptor is valid.

The following sections describe the Long-descriptor translation table descriptor formats:

- *VMSAv8-32 Long-descriptor level 1 and level 2 descriptor formats* on page G5-5950.
- *VMSAv8-32 Long-descriptor translation table level 3 descriptor formats* on page G5-5951.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

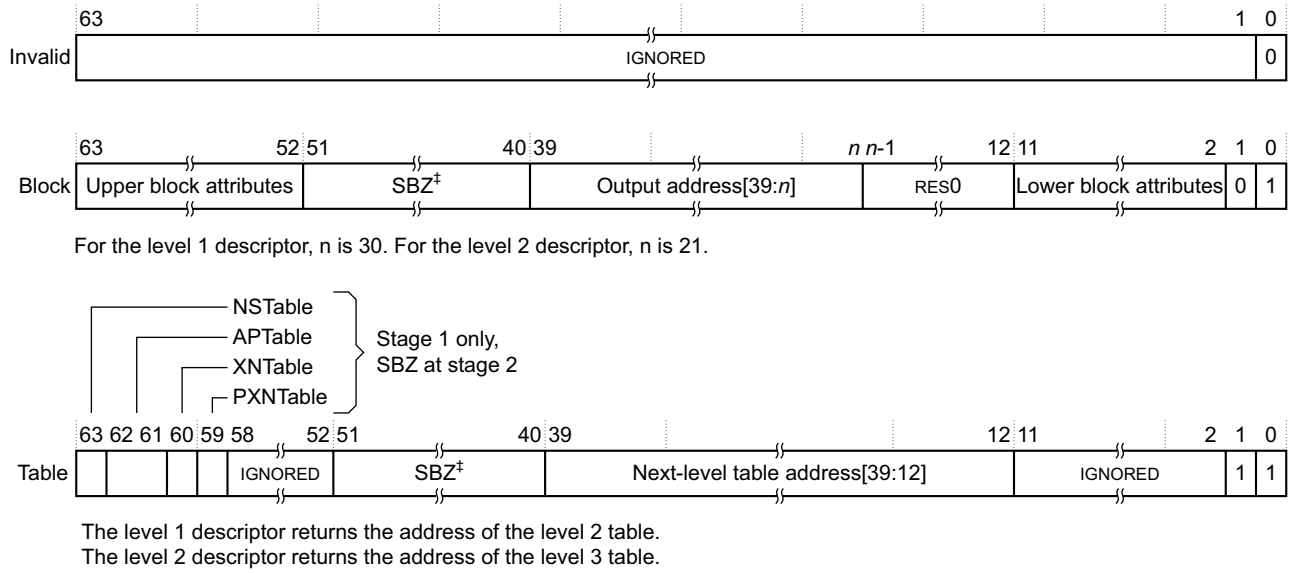
Information returned by a translation table lookup on page G5-5935 describes the classification of the non-address fields in the descriptors between *address map control*, *access controls*, and *region attributes*.

VMSAv8-32 Long-descriptor level 1 and level 2 descriptor formats

In the Long-descriptor translation tables, the formats of the level 1 and level 2 descriptors differ only in the size of the block of memory addressed by the block descriptor. A block entry:

- In a level 1 table describes the mapping of the associated 1GB input address range.
- In a level 2 table describes the mapping of the associated 2MB input address range.

Figure G5-10 shows the Long-descriptor level 1 and level 2 descriptor formats:



‡ See the descriptions of the address fields for more information about bits[47:40] of the Block and Table descriptors.

Figure G5-10 VMSAv8-32 Long-descriptor level 1and level 2 descriptor formats

Descriptor encodings, Long-descriptor level 1 and level 2 formats

Descriptor bit[0] identifies whether the descriptor is valid, and is 1 for a valid descriptor. If a lookup returns an invalid descriptor, the associated input address is unmapped, and any attempt to access it generates a Translation fault.

Descriptor bit[1] identifies the descriptor type, and is encoded as:

0, Block The descriptor gives the base address of a block of memory, and the attributes for that memory region.

1, Table The descriptor gives the address of the next level of translation table, and for a stage 1 translation, some attributes for that translation.

The other fields in the valid descriptors are:

Block descriptor

Gives the base address and attributes of a block of memory:

- For a level 1 Block descriptor, bits[39:30] are bits[39:30] of the output address that specifies a 1GB block of memory.
- For a level 2 Block descriptor, bits[39:21] are bits[39:21] of the output address that specifies a 2MB block of memory.

In both cases, if bits[47:40] of the descriptor are not zero then a translation that uses the descriptor will generate an Address size fault, see *Address size fault* on page G5-6014.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bits[63:52, 11:2] provide attributes for the target memory block, see *Attribute fields in VMSAv8-32 Long-descriptor translation table format descriptors* on page G5-5952. The position and contents of these bits is identical in the level 2 block descriptor and in the level 3 page descriptor.

Table descriptor

Bits[39:m] are bits[39:m] of the address of the required next-level table. Bits[m-1:0] of the table address are zero:

- For a level 1 Table descriptor, this is the address of a level 2 table.
- For a level 2 Table descriptor, this is the address of a level 3 table.

In both cases, if bits[47:40] of the descriptor are not zero then a translation that uses the descriptor will generate an Address size fault, see *Address size fault* on page G5-6014.

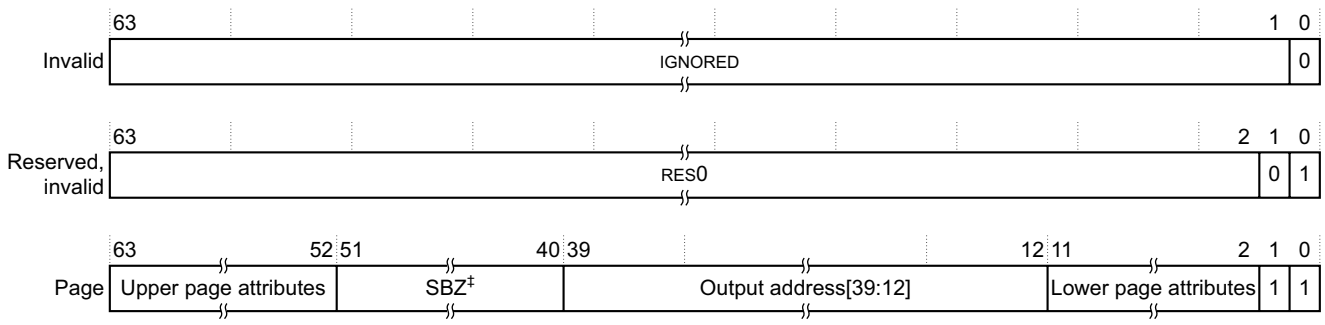
For a stage 1 translation only, bits[63:59] provide attributes for the next-level lookup, see *Attribute fields in VMSAv8-32 Long-descriptor translation table format descriptors* on page G5-5952.

If the translation table defines the Non-secure PL1&0 stage 1 translations, then the output address in the descriptor is the IPA of the target block or table. Otherwise, it is the PA of the target block or table.

VMSAv8-32 Long-descriptor translation table level 3 descriptor formats

Each entry in a level 3 table describes the mapping of the associated 4KB input address range.

Figure G5-11 shows the Long-descriptor level 3 descriptor formats.



‡ See the description of the address field for more information about bits[47:40] of the Page descriptor.

Figure G5-11 VMSAv8-32 Long-descriptor level 3 descriptor formats

Descriptor bit[0] identifies whether the descriptor is valid, and is 1 for a valid descriptor. If a lookup returns an invalid descriptor, the associated input address is unmapped, and any attempt to access it generates a Translation fault.

Descriptor bit[1] identifies the descriptor type, and is encoded as:

0, Reserved, invalid

Behaves identically to encodings with bit[0] set to 0.

This encoding must not be used in level 3 translation tables.

1, Page Gives the address and attributes of a 4KB page of memory.

At this level, the only valid format is the Page descriptor. The other fields in the Page descriptor are:

Page descriptor

Bits[39:12] are bits[39:12] of the output address for a page of memory.

If bits[47:40] of the descriptor are not zero, then a translation that uses the descriptor will generate an Address size fault, see *Address size fault* on page G5-6014.

Bits[63:52, 11:2] provide attributes for the target memory page, see *Attribute fields in VMSAv8-32 Long-descriptor translation table format descriptors* on page G5-5952. The position and contents of these bits are identical in the level 1 block descriptor and in the level 2 block descriptor.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the translation table defines the Non-secure PL1&0 stage 1 translations, then the output address in the descriptor is the IPA of the target page. Otherwise, it is the PA of the target page.

G5.5.3 Attribute fields in VMSAv8-32 Long-descriptor translation table format descriptors

The memory attributes in the VMSAv8-32 Long-descriptor translation tables are based on those in the Short-descriptor translation table format, with some extensions. *Memory region attributes* on page G5-5979 describes these attributes. In the Long-descriptor translation table format:

- Table entries for stage 1 translations define attributes for the next level of lookup, see *Next-level attributes in VMSAv8-32 Long-descriptor stage 1 Table descriptors*

  The hierarchical attributes in the translation tables, APTable, XNTable, and PXNTable, permit subtrees of the translation tables to be used by different agents. Not all operating systems use this functionality, and so ARMv8.2-AA32HPD adds a facility to disable these bits.

  This ability to disable hierarchical attribute bits has no effect on the NSTable bit.

- Block and Page entries define memory attributes for the target block or page of memory. Stage 1 and stage 2 translations have some differences in these attributes, see:
  — *Attribute fields in VMSAv8-32 Long-descriptor stage 1 Block and Page descriptors* on page G5-5953.
  — *Attribute fields in VMSAv8-32 Long-descriptor stage 2 Block and Page descriptors* on page G5-5955.

Next-level attributes in VMSAv8-32 Long-descriptor stage 1 Table descriptors

In a Table descriptor for a stage 1 translation, bits[63:59] of the descriptor define the following attributes for the next-level translation table access:

NSTable, bit[63] For memory accesses from Secure state, specifies the Security state for subsequent levels of lookup, see *Hierarchical control of Secure or Non-secure memory accesses, Long-descriptor format* on page G5-5956.

For memory accesses from Non-secure state, this bit is ignored.

APTable, bits[62:61] Access permissions limit for subsequent levels of lookup, see *Hierarchical control of access permissions, Long-descriptor format* on page G5-5970.

APTable[0] is reserved, SBZ, in the Non-secure EL2 stage 1 translation tables.

From Armv8.2, when ARMv8.2-AA32HPD is implemented, this field can be disabled.

When the value of TTBCR2.HPD0 or TTBCR2.HPD1 is 1, and the value of TTBCR.T2E is also 1:

- The value of the corresponding APTable field is IGNORED by hardware, allowing the field to be used by software.
- The behavior of the system is as if the value of the corresponding APTable field is 0, that is to say, the APTable field has an *Effective value* of 0.

XNTable, bit[60] XN limit for subsequent levels of lookup, see *Hierarchical control of instruction fetching, Long-descriptor format* on page G5-5974.

From Armv8.2, when ARMv8.2-AA32HPD is implemented, this field can be disabled.

When the value of TTBCR2.HPD0 or TTBCR2.HPD1 is 1, and the value of TTBCR.T2E is also 1:

- The value of the corresponding XNTable field is IGNORED by hardware, allowing the field to be used by software.
- The behavior of the system is as if the value of the corresponding XNTable field is 0, that is to say, the XNTable field has an *Effective value* of 0.

PXNTable, bit[59] PXN limit for subsequent levels of lookup, see *Hierarchical control of instruction fetching, Long-descriptor format* on page G5-5974.

This bit is RES0 in the Non-secure EL2 stage 1 translation tables.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

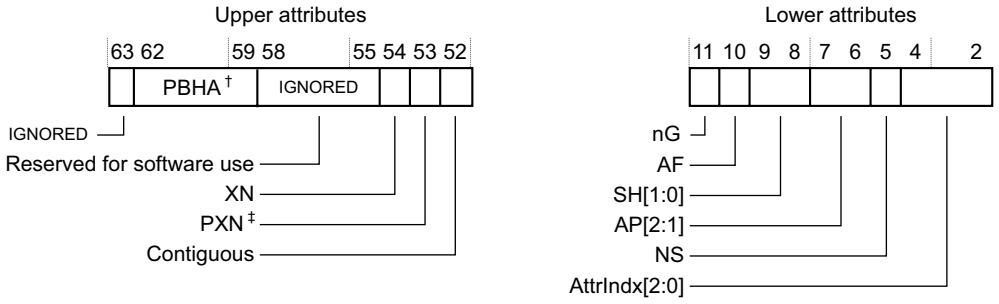
From Armv8.2, when ARMv8.2-AA32HPD is implemented, this field can be disabled.

When the value of TTBCR2.HPD0 or TTBCR2.HPD1 is 1 and the value of TTBCR.T2E is also 1:

- The value of the corresponding PXNTable field is ignored by hardware, allowing the field to be used by software.
- The behavior of the system is as if the value of the corresponding PXNTable field is 0, that is to say, the PXNTable field has an *Effective value* of 0.

Attribute fields in VMSAv8-32 Long-descriptor stage 1 Block and Page descriptors

In Block and Page descriptors, the memory attributes are split into an upper block and a lower block as shown for a stage 1 translation:



† IGNORED if ARMv8.2-TTPBHA is not implemented.

‡ RES0 for a translation regime that cannot apply to execution at EL0.

For a stage 1 descriptor, the attributes are:

PBHA, bits[62:59]

Page-based hardware attributes bits.

These bits are IGNORED when ARMv8.2-TTPBHA is not implemented.

When ARMv8.2-TTPBHA is implemented, the HTCR and the TTBCR2 registers both contain a control bit for each PBHA bit in the translation tables that they control. When the value of that control bit is 1, and the value of the corresponding Hierarchical permission disables bit is 1, hardware can use that PBHA bit for IMPLEMENTATION DEFINED purposes.

The control bits for this feature are:

For a Non-secure EL2 translation regime:

HTCR.HWUnn

Controls whether Block or Page descriptor bit[nn] can be used by hardware.

These controls apply only when the value of HTCR.HPD is 1.

For a PL1&0 translation regime:

TTBCR2.HWU1nn

For the translation tables indicated by TTBR1, controls whether Block or Page descriptor bit[nn] can be used by hardware.

These controls apply only when the value of TTBCR2.HPD1 is 1 and the value of TTBCR.T2E is 1.

TTBCR2.HWU0nn

For the translation tables indicated by TTBR0, controls whether Block or Page descriptor bit[nn] can be used by hardware.

These controls apply only when the value of TTBCR2.HPD0 is 1 and the value of TTBCR.T2E is 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Implementation of ARMv8.2-TTPBHA requires the implementation of ARMv8.2-AA32HPD, which provides the Hierarchical permission disables bits. If ARMv8.2-AA32HPD is implemented but ARMv8.2-TTPBHA is not implemented, then the control bits are RAZ/WI but other aspects of ARMv8.2-AA32HPD functionality are implemented. If neither feature is implemented, then:

- The control bits are RAZ/WI.
- The ARMv8.2-AA32HPD identification registers indicate that the functionality is not supported, see *ARMv8.2-AA32HPD* on page A2-75.
- The TTBCR2 register encoding is treated as unallocated.

XN, bit[54] The Execute-never field, see *Access permissions for instruction execution* on page G5-5972.

PXN, bit[53] The Privileged execute-never field, see *Access permissions for instruction execution* on page G5-5972.

This bit is RES0 in the Non-secure EL2 stage 1 translation tables.

Contiguous, bit[52]

Indicates that 16 adjacent translation table entries point to contiguous memory regions, see *Contiguous bit* on page G5-5986.

nG, bit[11] The not global bit. Determines how the translation is marked in the TLB, see *Global and process-specific translation table entries* on page G5-5991.

This bit is RES0 in the Non-secure EL2 stage 1 translation tables.

AF, bit[10] The Access flag, see *The Access flag* on page G5-5976.

SH, bits[9:8] Shareability field, see *Memory region attributes* on page G5-5979.

AP[2:1], bits[7:6]

Access Permissions bits, see *Memory access control* on page G5-5968.

Note

For consistency with the Short-descriptor translation table formats, the Long-descriptor format defines AP[2:1] as the Access Permissions bits, and does not define an AP[0] bit.

AP[1] is RES1 in the Non-secure EL2 stage 1 translation tables.

NS, bit[5] Non-secure bit. For memory accesses from Secure state, specifies whether the output address is in Secure or Non-secure memory, see *Control of Secure or Non-secure memory access, VMSAv8-32 Long-descriptor format* on page G5-5956.

For memory accesses from Non-secure state, this bit is RES0 and is ignored by the PE.

AttrIndx[2:0], bits[4:2]

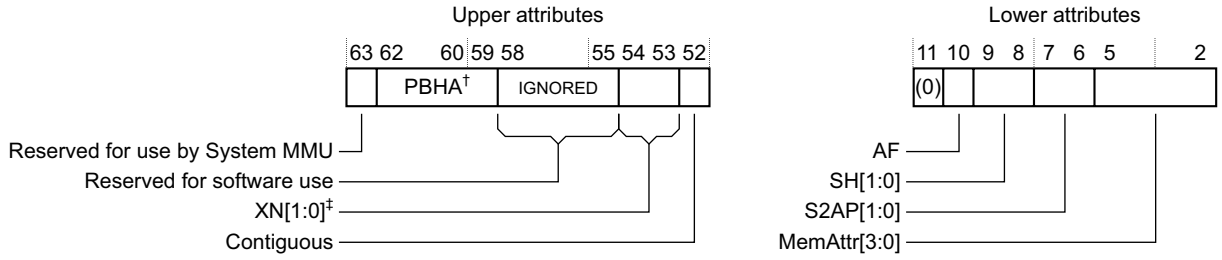
Stage 1 memory attributes index field, for the indicated Memory Attribute Indirection Register, see *VMSAv8-32 Long-descriptor format memory region attributes* on page G5-5985.

The definition of IGNORED means the architecture guarantees that the PE makes no use of the field, see *IGNORED* on page Glossary-8115. For more information about these fields, see *Other fields in the Long-descriptor translation table format descriptors* on page G5-5986.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Attribute fields in VMSAv8-32 Long-descriptor stage 2 Block and Page descriptors

In Block and Page descriptors, the memory attributes are split into an upper block and a lower block as shown for a stage 2 translation:



‡ Bit[53] is RES0 if ARMv8.2-TTS2UXN is not implemented.

† Bits [62:60] are IGNORED and reserved for use by System MMU if ARMv8.2-TTPBHA is not implemented.
Bits [59] is IGNORED if ARMv8.2-TTPBHA is not implemented.

For a stage 2 descriptor, the attributes are:

PBHA[3:1], bits[62:60]

Page-based hardware attributes bits.

These bits are IGNORED and reserved for System MMU use when ARMv8.2-TTPBHA is not implemented.

When ARMv8.2-TTPBHA is implemented, VTCR_EL2 has a control bit for each PBHA bit in the EL1&0 stage 2 translation tables. When the value of that control bit is 1, hardware can use the corresponding PBHA bit for IMPLEMENTATION DEFINED purposes. When the value of the control bit is 0, the corresponding PBHA bit is IGNORED and reserved for System MMU use.

PBHA[0], bit[59]

Page-based hardware attributes bit.

This bit is IGNORED when ARMv8.2-TTPBHA is not implemented.

When ARMv8.2-TTPBHA is implemented, VTCR_EL2 has a control bit for this bit in the EL1&0 stage 2 translation tables. When the value of that control bit is 1, hardware can use this bit for IMPLEMENTATION DEFINED purposes. When the value of the control bit is 0, this bit is IGNORED.

XN[1:0], bits[54:53]

The stage 2 Execute-never field, see *Access permissions for instruction execution* on page G5-5972.

If ARMv8.2-TTS2UXN is not implemented, bit[53] is RES0.

Contiguous, bit[52]

Indicates that 16 adjacent translation table entries point to contiguous memory regions, see *Contiguous bit* on page G5-5986.

AF, bit[10] The Access flag, see *The Access flag* on page G5-5976.

SH, bits[9:8] Shareability field, see *EL2 control of Non-secure memory region attributes* on page G5-5987.

S2AP, bits[7:6]

Stage 2 Access Permissions bits, see *Hyp mode control of Non-secure access permissions* on page G5-5977.

Note

In the original VMSAv7-32 Long-descriptor attribute definition, this field was called HAP[2:1], for consistency with the AP[2:1] field in the stage 1 descriptors and despite there being no HAP[0] bit. Armv8 renames the field for greater clarity.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MemAttr, bits[5:2]

Stage 2 memory attributes, see *EL2 control of Non-secure memory region attributes* on page G5-5987.

The definition of IGNORED means the architecture guarantees that the PE makes no use of the field, see *IGNORED* on page Glossary-8115. For more information about these fields, see *Other fields in the Long-descriptor translation table format descriptors* on page G5-5986.

G5.5.4 Control of Secure or Non-secure memory access, VMSAv8-32 Long-descriptor format

Access to the Secure or Non-secure PA map on page G5-5937 describes how the NS bit in the translation table entries:

- For accesses from Secure state, determines whether the access is to Secure or Non-secure memory.
- Is ignored by accesses from Non-secure state.

In the Long-descriptor format:

- The NS bit relates only to the memory block or page at the output address defined by the descriptor.
- The descriptors also include an NSTable bit, see *Hierarchical control of Secure or Non-secure memory accesses, Long-descriptor format*.

The NS and NSTable bits are valid only for memory accesses from Secure state. Memory accesses from Non-secure state ignore the values of these bits.

Hierarchical control of Secure or Non-secure memory accesses, Long-descriptor format

For Long-descriptor format table descriptors for stage 1 translations, the descriptor includes an NSTable bit, that indicates whether the table identified in the descriptor is in Secure or Non-secure memory. For accesses from Secure state, the meaning of the NSTable bit is:

NSTable == 0 The defined table address is in the Secure PA map. In the descriptors in that translation table, NS bits and NSTable bits have their defined meanings.

NSTable == 1 The defined table address is in the Non-secure PA map. Because this table is fetched from the Non-secure address map, the NS and NSTable bits in the descriptors in this table must be ignored. This means that, for this table:

- The value of the NS bit in any block or page descriptor is ignored. The block or page address refers to Non-secure memory.
- The value of the NSTable bit in any table descriptor is ignored, and the table address refers to Non-secure memory. When this table is accessed, the NS bit in any block or page descriptor is ignored, and all descriptors in the table refer to Non-secure memory.

In addition, an entry fetched in Secure state is treated as non-global if it is read from Non-secure memory. That is, these entries must be treated as if nG==1, regardless of the value of the nG bit. For more information about the nG bit, see *Global and process-specific translation table entries* on page G5-5991.

The effect of NSTable applies to later entries in the translation table walk, and so its effects can be held in one or more TLB entries. Therefore, a change to NSTable requires coarse-grained invalidation of the TLB to ensure that the effect of the change is visible to subsequent memory transactions.

Note

- When using the Long-descriptor format, table descriptors are defined only for the level 1 and level 2 of lookup.
- Stage 2 translations are performed only for operations in Non-secure state, that can access only the Non-secure address map. Therefore, the stage 2 descriptors do not include NS or NSTable bits.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G5.5.5 Selecting between TTBR0 and TTBR1, VMSAv8-32 Long-descriptor translation table format

As described in *Determining the translation table base address in the VMSAv8-32 translation regimes* on page G5-5936, two sets of translation tables can be defined for each of the PL1&0 stage 1 translations, and TTBR0 and TTBR1 hold the base addresses for the two sets of tables. The Long-descriptor translation table format provides more flexibility in defining the boundary between using TTBR0 and using TTBR1. When a PL1&0 stage 1 address translation is enabled, TTBR0 is always used. If TTBR1 is also used then:

- TTBR1 is used for the top part of the input address range.
- TTBR0 is used for the bottom part of the input address range.

The TTBCR.T0SZ and TTBCR.T1SZ size fields control the use of TTBR0 and TTBR1, as Table G5-2 shows.

Table G5-2 Use of TTBR0 and TTBR1, Long-descriptor format

TTBCR		Input address range using:	
T0SZ	**T1SZ**	**TTBR0**	**TTBR1**
`0b000`	`0b000`	All addresses	Not used
M[a]	`0b000`	Zero to $(2^{(32-M)}-1)$	2^{32-M} to maximum input address
`0b000`	*N*[a]	Zero to $(2^{32}-2^{(32-N)}-1)$	$2^{32}-2^{(32-N)}$ to maximum input address
M[a]	*N*[a]	Zero to $(2^{(32-M)}-1)$	$2^{32}-2^{(32-N)}$ to maximum input address

a. *M*, *N* must be greater than 0.The maximum possible value for each of T0SZ and T1SZ is 7.

For stage 1 translations, the input address is always a VA, and the maximum possible VA is $(2^{32}-1)$.

When address translation is using the Long-descriptor translation table format:

- Figure G5-12 shows how, when TTBCR.T1SZ is zero, the value of TTBCR.T0SZ controls the boundary between VAs that are translated using TTBR0, and VAs that are translated using TTBR1.

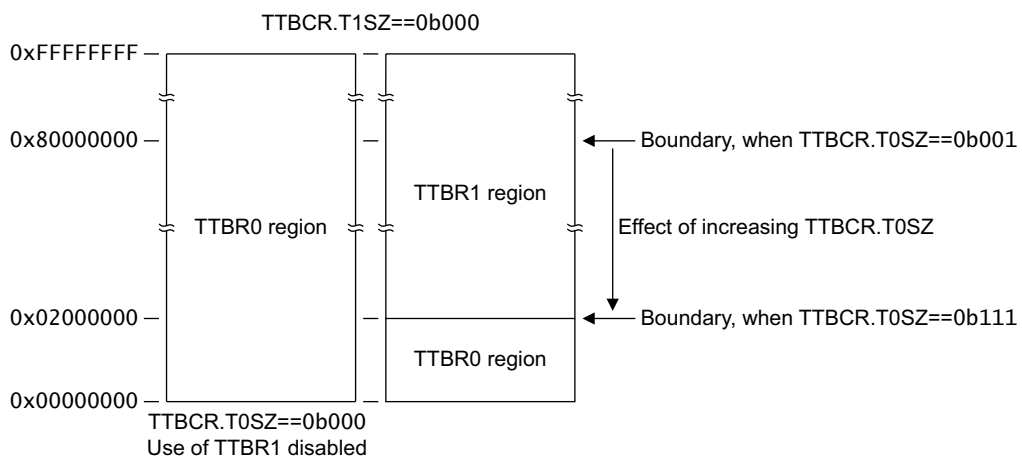


Figure G5-12 Control of TTBR boundary, when TTBCR.T1SZ is zero

- Figure G5-13 on page G5-5958 shows how, when TTBCR.T1SZ is nonzero, the values of TTBCR.T0SZ and TTBCR.T1SZ control the boundaries between VAs that are translated using TTBR0, and VAs that are translated using TTBR1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

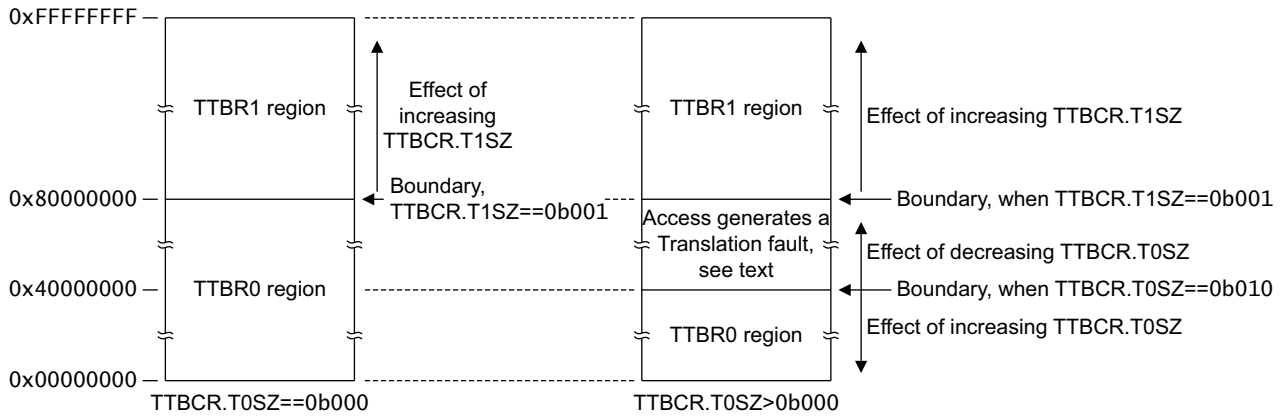


Figure G5-13 Control of TTBR boundaries, when TTBCR.T1SZ is nonzero

When T0SZ and T1SZ are both nonzero:

— If both fields are set to `0b001`, the boundary between the two regions is `0x80000000`. This is identical to having T0SZ set to `0b000` and T1SZ set to `0b001`.

— Otherwise, the TTBR0 and TTBR1 regions are non-contiguous. In this case, any attempt to access an address that is in that gap between the TTBR0 and TTBR1 regions generates a Translation fault.

Note

The handling of the Contiguous bit can mean that the boundary between the translation regions defined by the TCR_EL1.TnSZ values and the region for which an access generates a Translation fault is wider than shown in Figure G5-13. That is, if the descriptor for an access to the region shown as generating a fault has the Contiguous bit set to 1, the access might not generate a fault. *Possible errors in programming the translation table registers* describes this possibility.

When using the Long-descriptor translation table format:

- The TTBCR contains fields that define memory region attributes for the translation table walk, for each TTBR. These are the SH0, ORGN0, IRGN0, SH1, ORGN1, and IRGN1 bits.
- TTBR0 and TTBR1 each contain an ASID field, and the TTBCR.A1 field selects which ASID to use.

For this translation table format, *VMSAv8-32 Long-descriptor translation table format address lookup levels* on page G5-5959 summarizes the lookup levels, and *Translation table walks, when using the VMSAv8-32 Long-descriptor translation table format* on page G5-5963 describes the possible translations.

Possible errors in programming the translation table registers

In all the descriptions in this subsection, the *size of the input address* supported for a PL1&0 stage 1 translation refers to the size specified by a TTBCR.TxSZ field.

Note

For a PL1&0 stage 1 translation, the input address range can be split so that the lower addresses are translated by TTBR0 and the higher addresses are translated by TTBR1. In this case, each of input address sizes specified by TTBCR.{T0SZ, T1SZ} is smaller than the total address size supported by the stage of translation.

The following are possible errors in the programming of TTBR0, TTBR1, and TTBCR. For the translation of a particular address at a particular stage of translation, either:

- The block size being used to translate the address is larger than the size of the input address supported at a stage of translation used in performing the required translation. This can occur only for the PL1&0 stage 1 translations, and only when either TTBCR.T0SZ or TTBCR.T1SZ is zero, meaning there is no gap between

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

the address range translated by TTBR0 and the range translated by TTBR1. In this case, this programming error occurs if a block translated from the region that has TxSZ set to zero straddles the boundary between the two address ranges. Example G5-2 shows an example of this mis-programming.

- The address range translated by a set of blocks marked as contiguous, by use of the contiguous bit, is larger than the size of the input address supported at a stage of translation used in performing the required translation.

Example G5-2 Error in programming the translation table registers

---

If TTBCR.T0SZ is programmed to 0 and TTBCR.T1SZ is programmed to 7, this means:

- TTBR0 translates addresses in the range `0x00000000-0xFDFFFFFF`.
- TTBR1 translates addresses in the range `0xFE000000-0xFFFFFFFF`.

The translation table indicated by TTBR0 might be programmed with a block entry for a 1GB region starting at `0xC0000000`. This covers the address range `0xC0000000-0xFFFFFFFF`, that overlaps the TTBR1 address range. This means this block size is larger than the input address size supported for translations using TTBR0, and therefore this is a programming error.

To understand why this must be a programming error, consider a memory access to address `0xFFFF0000`. According to the TTBCR.{T0SZ, T1SZ} values, this must be translated using TTBR1. However, the access matches a TLB entry for the translation, using TTBR0, of the block at `0xC0000000`. Hardware is not required to detect that the access to `0xFFFF0000` is being translated incorrectly.

In these cases, an implementation might use one of the following approaches:

- Treat such a block as causing a Translation fault, even though the block is valid, and the address accessed within that block is within the size of the input address supported at a stage of translation.

  The block might be a block within a contiguous set of blocks.

- Treat such a block as not causing a Translation fault, even though the address accessed within that block is outside the size of the input address supported at a stage of translation, provided that both of the following apply:
  - The block is valid.
  - At least one address within the block, or contiguous set of blocks, is within the size of the input address supported at a stage of translation.

  The block might be a block within a contiguous set of blocks.

Additional constraints apply to programming the VTCR, see *Determining the required initial lookup level for stage 2 translations* on page G5-5965.

G5.5.6 VMSAv8-32 Long-descriptor translation table format address lookup levels

As stated at the start of this section, because the Long-descriptor translation table format is used for the Non-secure PL1&0 stage 2 translations, the format must support input addresses of up to 40 bits.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G5-3 summarizes the properties of the different levels of address lookup when using this format.

Table G5-3 Properties of the three levels of address lookup with VMSAv8-32 Long-descriptor translation tables

Level	Input address		Output address[a]		Number of entries
	Size	Address range[b]	Size	Address range	
First	Up to 512GB	Up to Address[38:0]	1GB	Address[39:30]	Up to 512
Second	Up to 1GB	Up to Address[29:0]	2MB	Address[39:21]	Up to 512
Third	2MB	Address[20:0]	4KB	Address[39:12]	512

a. Output address when an entry addresses a block of memory or a memory page. If an entry addresses the next level of address lookup it specifies Address[39:12] for the next-level translation table.

b. Input address range for the translation table. See *Use of concatenated level 1 translation tables* on page G5-5961 for details of support for additional bits of address at a given level, including possible support of a 40-bit input address range for stage 2 translations at level 1. For stage 1 translations at level 1 the input address range is limited to the VA size of [31:0].

For level 1 and level 2 tables, reducing the input address range reduces the number of addresses in the table and therefore reduces the table size. The appropriate Translation Table Control Register specifies the input address range.

Stage 1 translations require an input address range of up to 32 bits, corresponding to VA[31:0]. For these translations:

- For a memory access from a mode other than Hyp mode, the Secure or Non-secure TTBR0 or TTBR1 holds the translation table base address, and the Secure or Non-secure TTBCR is the control register.
- For a memory access from Hyp mode, HTTBR holds the translation table base address, and HTCR is the control register.

Note

For translations controlled by TTBR0 and TTBR1, if neither TTBR has an input address range larger than 1GB, then translation starts at level 2. Together, TTBR0 and TTBR1 can still cover the 32-bit VA input address range.

Stage 2 translations require an input address range of up to 40 bits, corresponding to IPA[39:0], and the supported input address size is configurable in the range 25-40 bits. Table G5-3 indicates a requirement for the translation mechanism to support a 39-bit input address range, Address[38:0]. *Use of concatenated translation tables for the initial stage 2 lookup* describes how a 40-bit IPA address range is supported. For stage 2 translations:

- VTTBR holds the translation table base address, and VTCR is the control register.
- If a supplied input address is larger than the configured input address size, a Translation fault is generated.

Use of concatenated translation tables for the initial stage 2 lookup

If a stage 2 translation would require 16 entries or fewer in its top-level translation table, that stage of translation can, instead, be configured so that:

- It requires the corresponding number of concatenated translation tables at the next translation level, aligned to the size of the block of concatenated translation tables.
- The stage 2 translation starts at that next translation level.

Note

Stage 2 translations always use the Long-descriptor translation table format.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This use of concatenated translation tables is:

- Required when the stage 2 translation supports a 40-bit input address range, see *Use of concatenated level 1 translation tables*.
- Supported for a stage 2 translation with an input address range of 31-34 bits, see *Use of concatenated level 2 translation tables*.

The use of concatenated translation tables requires the software that is defining the translation to:

- Define the concatenated translation tables with the required overall alignment.
- Program VTTBR to hold the address of the first of the concatenated translation tables.
- Program VTCR to indicate the required input address range and initial lookup level.

Note

The use of concatenated translation tables avoids the overhead of an additional level of translation.

Use of concatenated level 1 translation tables

The Long-descriptor format translation tables provide 9 bits of address resolution at each level of lookup. However, a 40-bit input address range with a translation granularity of 4KB requires a total of 28 bits of address resolution. Therefore, a stage 2 translation that supports a 40-bit input address range requires two concatenated level 1 translation tables, together aligned to 8KB, where:

- The table at the address with PA[12:0]==`0b0_0000_0000_0000` defines the translations for input addresses with bit[39]==0.
- The table at the address with PA[12:0]==`0b1_0000_0000_0000` defines the translations for input addresses with bit[39]==1.
- The 8KB alignment requirement means that both tables have the same value for PA[39:13].

Use of concatenated level 2 translation tables

A stage 2 translation with an input address range of 31-34 bits can start the translation either:

- With a level 1 lookup, accessing a level 1 translation table with 2-16 entries.
- With a level 2 lookup, accessing a set of concatenated level 2 translation tables.

Table G5-4 shows these options, for each of the input address ranges that can use this scheme.

Note

Because these are stage 2 translations, the input address range is an IPA range.

Table G5-4 Possible uses of concatenated translation tables for level 2 lookup

Input address range		Lookup starts at level 1	Lookup starts at level 2	
IPA range	**Size**	**Required level 1 entries**	**Number of concatenated tables**	**Required alignment[a]**
IPA[30:0]	2^{31} bytes	2	2	8KB
IPA[31:0]	2^{32} bytes	4	4	16KB
IPA[32:0]	2^{33} bytes	8	8	32KB
IPA[33:0]	2^{34} bytes	16	16	64KB

a. Required alignment of the set of concatenated level 2 tables.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

See also *Determining the required initial lookup level for stage 2 translations* on page G5-5965.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G5.5.7 Translation table walks, when using the VMSAv8-32 Long-descriptor translation table format

Figure G5-2 on page G5-5927 shows the possible address translations. If a stage of translation is controlled from an Exception level that is using AArch32, the input and output address constraints and the registers that control the translation are as follows:

Stage 1 translations

For all stage 1 translations:

- The input address range is up to 32 bits, as determined by either:
  - TTBCR.T0SZ or TTBCR.T1SZ, for a PL1&0 stage 1 translation.
  - HTCR.T0SZ, for an EL2 stage 1 translation.
- The output address range is 40 bits.

The stage 1 translations are:

Non-secure PL1&0 stage 1 translation

The stage 1 translation for memory accesses from Non-secure modes other than Hyp mode. This translates a VA to an IPA. For this translation, when Non-secure EL1 is using AArch32:

- Non-secure TTBR0 or TTBR1 holds the translation table base address.
- Non-secure TTBCR determines which TTBR is used.

Non-secure EL2 stage 1 translation

The stage 1 translation for memory accesses from Hyp mode, translates a VA to a PA. For this translation, when EL2 is using AArch32, HTTBR holds the translation table base address.

Secure PL1&0 stage 1 translation

The stage 1 translation for memory accesses from Secure modes, translates a VA to a PA. For this translation, when the Secure PL1 modes are using AArch32:

- Secure TTBR0 or TTBR1 holds the translation table base address.
- Secure TTBCR determines which TTBR is used.

Stage 2 translation

Non-secure PL1&0 stage 2 translation

The stage 2 translation for memory accesses from Non-secure modes other than Hyp mode, and translates an IPA to a PA. For this translation, when EL2 is using AArch32:

- The input address range is 40 bits, and VTCR.T0SZ determines the input address size.
- The output address range depends on the implemented memory system, and is up to 40 bits.
- VTTBR holds the translation table base address.
- VTCR specifies the required input address range, and whether the initial lookup is at level 1 or at level 2.

The descriptions of the VMSAv8-32 translation stages state that the maximum output address size is 40 bits. However, the register and Long-descriptor format descriptor fields that hold these addresses are 48 bits wide. If bits[47:40] of an output address are not all zero, then the address generates an Address size fault.

The Long-descriptor translation table format provides up to three levels of address lookup, as described in *VMSAv8-32 Long-descriptor translation table format address lookup levels* on page G5-5959, and the initial lookup, in which the MMU reads the translation table base address, is at either level 1 or level 2. The following determines the level of the initial lookup:

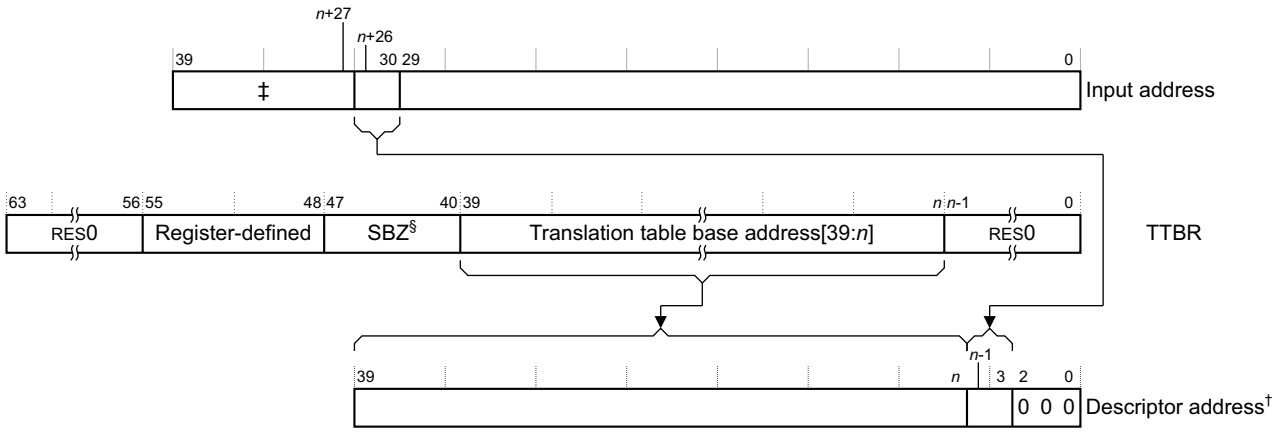
- For a stage 1 translation, the required input address range. For more information, see *Determining the required initial lookup level for stage 1 translations* on page G5-5965.
- For a stage 2 translation, the level specified by the VTCR.SL0 field. For more information, see *Determining the required initial lookup level for stage 2 translations* on page G5-5965.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

For a stage 2 translation, the size of the required input address range constrains the VTCR.SL0 value.

Figure G5-14 shows how the descriptor address for the initial lookup for a translation using the Long-descriptor translation table format is determined from the input address and the TTBR value. This figure shows the lookup for a translation that starts with a level 1 lookup, that translates bits[39:30] of the input address, zero extended if necessary.



See text for more information about the translation table base register used, and the value of *n*.
‡ This field is absent if *n* is 13.
† For a Non-secure PL1&0 stage 1 translation, the IPA of the descriptor. Otherwise, the PA of the descriptor.
§ See the lookup description for more information about bits[40:47] of the TTBR

Figure G5-14 VMSAv8-32 Long-descriptor initial lookup, starting at level 1

If bits[47:40] of the TTBR are not zero then the initial lookup will generate an Address size fault, see *Address size fault* on page G5-6014.

For a translation that starts with a level 1 lookup, as shown in Figure G5-14:

For a stage 1 translation

n is in the range 4-5 and:

- For a memory access from Hyp mode:
  - HTTBR is the TTBR.
  - *n*=5-(HTCR.T0SZ).
- For other accesses:
  - The Secure or Non-secure instance of TTBR0 or TTBR1 is the TTBR.
  - *n*=(5-TTBCR.T*x*SZ), where *x* is 0 when using TTBR0, and 1 when using TTBR1.

For a stage 2 translation

n is in the range 4-13 and:

- VTTBR is the TTBR.
- *n*=5-(VTCR.T0SZ).

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For a translation that starts with a level 2 lookup, the descriptor address is obtained in the same way, except that bits[(n+17):21] of the input address provide bits[(n-1):3] of the descriptor address, where:

For a stage 1 translation

n is in the range 7-12. As *Determining the required initial lookup level for stage 1 translations* shows, for a stage 1 translation to start with a level 2 lookup, the corresponding T0SZ or T1SZ field must be 2 or more. This means:

- For a memory access from Hyp mode, n=14-HTCR.T0SZ.
- For other memory accesses, n=14-(TTBCR.TxSZ), where x is 0 when using TTBR0, and 1 when using TTBR1.

For a stage 2 translation

n is in the range 7-16. For a stage 2 translation to start with a level 2 lookup, VTCR.SL0 is `0b00`, and n=14-(VTCR.T0SZ).

The following sections describe how the level of the initial lookup is determined:

- *Determining the required initial lookup level for stage 1 translations*.
- *Determining the required initial lookup level for stage 2 translations*.

Address translation examples using the VMSAv8-32 Long descriptor translation table format on page K7-7943 shows examples of full translation flows, to an entry for a 4KB memory page, for lookups starting at level 1 and lookups starting at level 2.

Determining the required initial lookup level for stage 1 translations

For a stage 1 translation, the required input address range, indicated by a T0SZ or T1SZ field in a translation table control register, determines the initial lookup level. The size of this input address region is $2^{(32-T\mathit{x}SZ)}$ bytes, and if this size is:

- Less than or equal to 2^{30} bytes, the required start is at level 2, and translation requires two levels of table to map to 4KB pages. This corresponds to a TxSZ value of 2 or more.
- More than 2^{30} bytes, the required start is at level 1, and translation requires three levels of table to map to 4KB pages. This corresponds to a TxSZ value that is less than 2.

For the PL1&0 stage 1 translations, the TTBCR:

- Splits the 32-bit VA input address range between TTBR0 and TTBR1, see *Selecting between TTBR0 and TTBR1, VMSAv8-32 Long-descriptor translation table format* on page G5-5957.
- Holds the input address range sizes for TTBR0 and TTBR1, in the TTBCR.T0SZ and TTBCR.T1SZ fields.

For the EL2 stage 1 translations, HTCR.T0SZ indicates the size of the required input address range. For example, if this field is `0b000`, it indicates a 32-bit VA input address range, and translation lookup must start at level 1.

Determining the required initial lookup level for stage 2 translations

For a PL1&0 stage 2 translation, the output address range from the PL1&0 stage 1 translations determines the required input address range for the stage 2 translation.

VTCR.SL0 indicates the starting level for the lookup. The permitted SL0 values are:

`0b00` Stage 2 translation lookup must start at level 2.

`0b01` Stage 2 translation lookup must start at level 1.

In addition, VTCR.T0SZ must indicate the required input address range. The size of the input address region is $2^{(32-T0SZ)}$ bytes.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

VTCR.T0SZ holds a four-bit signed integer value, meaning it supports values from -8 to 7. This is different from the other translation control registers, where T*n*SZ holds a three-bit unsigned integer, supporting values from 0 to 7.

The programming of VTCR must follow the constraints shown in Table G5-5, otherwise any attempt to perform a translation table walk that uses the stage 2 address translation generates a stage 2 level 1 Translation Fault. The table also shows how the VTCR.SL0 and VTCR.T0SZ values determine the VTTBR.BADDR field width.

Note

If VTCR.SL0 is programmed to a reserved value then the constraints shown in Table G5-5 are not met, and a translation table walk that uses stage 2 translation generates a stage 2 level 1 Translation fault.

Table G5-5 Input address range constraints on programming VTCR

VTCR.SL0	VTCR.T0SZ	Input address range, R	Initial lookup level	BADDR[39:x] width[a]
0b00	2 to 7	$R \le 2^{30}$ bytes	Level 2	[39:12] to [39:7]
0b00	-2 to 1	$2^{30} < R \le 2^{34}$ bytes	Level 2	[39:16] to [39:13]
0b01	-2 to 1		Level 1	[39:7] to [39:4]
0b01	-8 to -3	$2^{34} < R$	Level 1	[39:13] to [39:8]

a. The first range corresponds to the first T0SZ value, the second range to the second T0SZ value.

In addition, VTCR.S must be programmed to the value of T0SZ[3], otherwise behavior is CONSTRAINED UNPREDICTABLE with the resulting behavior being that VTCR.T0SZ is treated as an UNKNOWN value.

Note

VTCR.T0SZ being treated as an UNKNOWN value results in a stage 2 level 1 Translation Fault if that UNKNOWN value is not consistent with the programmed value of VTCR.SL0.

CONSTRAINED UNPREDICTABLE behaviors associated with the VTCR on page K1-7850 describes these CONSTRAINED UNPREDICTABLE behaviors.

Where necessary, the initial lookup level provides multiple concatenated translation tables, as described in *Use of concatenated level 2 translation tables* on page G5-5961. This section also gives more information about the alternatives, shown in Table G5-5, when R is in the range 2^{31}-2^{34}.

G5.5.8 The algorithm for finding the translation table entries, VMSAv8-32 Long-descriptor format

This section gives the algorithm for finding the translation table entry that corresponds to a given IA, for each required level of lookup. The algorithm encodes the descriptions of address translation given earlier in this section. The VMSAv8-32 Long-descriptor format uses a 4KB translation granule.

The description uses the following terms:

BaseAddr The base address for the level of lookup, as defined by:

- For the initial lookup level, the TTBR.BADDR base address field in the appropriate TTBR, see the description of *TnSZ* on page G5-5967.
- Otherwise, the translation table address returned by the previous level of lookup.

IA The supplied IA for this stage of translation.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T*n*SZ The translation table size for this stage of translation:

For PL1&0 stage 1 Either:

- TTBCR.T0SZ if the translation is using TTBR0.
- TTBCR.T1SZ if the translation is using TTBR1.

For PL1&0 stage 2 VTCR.T0SZ. The translation uses VTTBR.

For EL2 stage 1 HTCR.T0SZ. The translation uses HTTBR.

SL0 VTCR.SL0. Applies to the Non-secure PL1&0 stage 2 translation only.

Table G5-6 shows the translation table descriptor address, for each level of lookup. The table shows only architecturally-valid programming of the TCR. See also *Possible errors in programming the translation table registers* on page G5-5958.

Table G5-6 Translation table entry addresses, VMSAv8-32 using Long-descriptor format

Lookup level	Entry address and conditions		General conditions
	Stage 1 translation	Stage 2 translation	
One	BaseAddr[39:*x*]:IA[*y*:30]:`0b000` if[a] 0 ≤ T*n*SZ ≤ 1 then *x* = (5 - T*n*SZ)	BaseAddr[39:*x*]:IA[*y*:30]:`0b000` if SL0[b] == 1 then if[a] -8 ≤ T0SZ ≤ 1 then *x* = (5 - T0SZ)	*y* = (*x* + 26)
Two	BaseAddr[39:*x*]:IA[*y*:21]:`0b000` if[a] 2 ≤ T*n*SZ ≤ 7 then *x* = (14 - T*n*SZ) else[c] *x* =12	BaseAddr[39:*x*]:IA[*y*:21]:`0b000` if SL0 == 0 then if[a] -2 ≤ T0SZ ≤ 7 then *x* = (14 - T0SZ) elsif[c] SL0[b] == 1 then *x* = 12	*y* = (*x* + 17)
Three	BaseAddr[39:12]:IA[20:12]:`0b000`	BaseAddr[39:12]:IA[20:12]:`0b000`	-

a. This line indicates the range of permitted values for T*n*SZ, for a lookup that starts at this level, see *Use of concatenated translation tables for the initial stage 2 lookup* on page G5-5960.

b. SL0 == 0 if the initial lookup is level 2, SL0 == 1 if the initial lookup is level 1.

c. This is the case where this level of lookup is not the initial level of lookup.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G5.6 Memory access control

In addition to an output address, a translation table entry that refers to page or region of memory includes fields that define properties of the target memory region. *Information returned by a translation table lookup* on page G5-5935 describes the classification of those fields as address map control, access control, and memory attribute fields. The access control fields, described in this section, determine whether the PE, in its current state, is permitted to perform the required access to the output address given in the translation table descriptor. If a translation stage does not permit the access, then an MMU fault is generated for that translation stage, and no memory access is performed.

The following sections describe the memory access controls:

- *About access permissions*.
- *About the PAN bit* on page G5-5971.
- *Access permissions for instruction execution* on page G5-5972.
- *Domains, Short-descriptor format only* on page G5-5976.
- *The Access flag* on page G5-5976.
- *Hyp mode control of Non-secure access permissions* on page G5-5977.

G5.6.1 About access permissions

The translation table descriptors include fields that define access permissions for data accesses and for instruction fetches. This section introduces those fields. In addition:

- System register controls can prevent execution from writable locations, see *Preventing execution from writable locations* on page G5-5975.
- In Armv8.1, the PSTATE.PAN can affect the access permissions for privileged data accesses, see *About the PAN bit* on page G5-5971.

Note

This section gives a general description of memory access permissions. Software executing at PL1 in Non-secure state can see only the access permissions defined by the Non-secure PL1&0 stage 1 translations. However, software executing at EL2 can modify these permissions, as described in *Hyp mode control of Non-secure access permissions* on page G5-5977. This modification is invisible to Non-secure software executing at EL1 or EL0.

Access permission bits in a translation table descriptor control access to the corresponding memory region. The details of this control depend on the translation table format, as follows:

Short-descriptor format

This format supports two options for defining the access permissions:

- Three bits, AP[2:0], define the access permissions.
- Two bits, AP[2:1], define the access permissions, and AP[0] can be used as an Access flag.

SCTLR.AFE selects the access permissions option. Setting this bit to 1, to enable the Access flag, also selects use of AP[2:1] to define access permissions.

Arm deprecates any use of the AP[2:0] scheme for defining access permissions.

Long-descriptor format

AP[2:1] to control the access permissions, and the descriptors provide an AF bit for use as an Access flag. This means VMSAv8-32 behaves as if the value of SCTLR.AFE is 1, regardless of the value that software has written to this bit.

Note

When use of the Long-descriptor format is enabled, SCTLR.AFE is UNK/SBOP.

The Access flag on page G5-5976 describes the Access flag, for both translation table formats.

The XN and PXN bits provide additional access controls for instruction fetches, see *Access permissions for instruction execution* on page G5-5972.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

An attempt to perform a memory access that the translation table access permission bits do not permit generates a Permission fault, for the corresponding stage of translation. However, when using the Short-descriptor translation table format, it generates the fault only if the access is to memory in the Client domain, see *Domains, Short-descriptor format only* on page G5-5976.

Note

For the Non-secure PL1&0 translation regime, memory accesses are subject to two stages of translation. Each stage of translation has its own, independent, fault checking. Fault handling is different for the two stages, see *Exception reporting in a VMSAv8-32 implementation* on page G5-6024.

The following sections describe the two access permissions models:

- *AP[2:1] access permissions model*.
- *AP[2:0] access permissions control, Short-descriptor format only* on page G5-5970. This section includes some information on access permission control in earlier versions of the Arm VMSA.

AP[2:1] access permissions model

Note

Arm recommends that this model is always used, even where the AP[2:0] model is permitted. Some documentation describes the AP[2:1] model as the *simplified access permissions* model.

This access permissions model is used if the translation is either:

- Using the Long-descriptor translation table format.
- Using Short-descriptor translation table format, and the SCTLR.AFE bit is set to 1.

In this model:

- One bit, AP[2], selects between read-only and read/write access.
- A second bit, AP[1], selects between Application level (EL0) and System level (PL1) control.
  For the Non-secure EL2 stage 1 translations, AP[1] is SBO.

This provides four access combinations:

- Read-only at all privilege levels.
- Read/write at all privilege levels.
- Read-only at PL1, no access by software executing at EL0.
- Read/write at PL1, no access by software executing at EL0.

Table G5-7 shows this access control model.

Table G5-7 VMSAv8-32 AP[2:1] access permissions model

AP[2], disable write access	AP[1], enable unprivileged access	Access
0	0[a]	Read/write, only at PL1
0	1	Read/write, at any privilege level
1	0[a]	Read-only, only at PL1
1	1	Read-only, at any privilege level

a. Not valid for Non-secure EL2 stage 1 translation tables. AP[1] is SBO in these tables.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Hierarchical control of access permissions, Long-descriptor format

The Long-descriptor translation table format introduces a mechanism that entries at one level of translation table lookup can use to set limits on the permitted entries at subsequent levels of lookup. This applies to the access permissions, and also to the restrictions on instruction fetching described in *Hierarchical control of instruction fetching, Long-descriptor format* on page G5-5974.

The restrictions apply only to subsequent levels of lookup at the same stage of translation. The APTable[1:0] field restricts the access permissions, as Table G5-8 shows.

However, in an implementation that includes ARMv8.2-AA32HPD, when hierarchical control of data access permissions is disabled for a translation regime, the information in this subsection does not apply. See *Attribute fields in VMSAv8-32 Long-descriptor translation table format descriptors* on page G5-5952.

As stated in the table footnote, for the Non-secure EL2 stage 1 translation tables, APTable[0] is reserved, SBZ.

Table G5-8 Effect of APTable[1:0] on subsequent levels of lookup

APTable[1:0]	Effect
00	No effect on permissions in subsequent levels of lookup.
01 [a]	Access at EL0 not permitted, regardless of permissions in subsequent levels of lookup.
10	Write access not permitted, at any Exception level, regardless of permissions in subsequent levels of lookup.
11 [a]	Regardless of permissions in subsequent levels of lookup: • Write access not permitted, at any Exception level. • Read access not permitted at EL0.

a. Not valid for the Non-secure EL2 stage 1 translation tables. In those tables, APTable[0] is SBZ.

Note

The APTable[1:0] settings are combined with the translation table access permissions in the translation tables descriptors accessed in subsequent levels of lookup. They do not restrict or change the values entered in those descriptors.

The Long-descriptor format provides APTable[1:0] control only for the stage 1 translations. The corresponding bits are SBZ in the stage 2 translation table descriptors.

The effect of APTable applies to later entries in the translation table walk, and so its effects can be held in one or more TLB entries. Therefore, a change to APTable requires coarse-grained invalidation of the TLB to ensure that the effect of the change is visible to subsequent memory transactions.

AP[2:0] access permissions control, Short-descriptor format only

This access permissions model applies when using the Short-descriptor translation tables format, and the SCTLR.AFE bit is set to 0. Arm deprecates any use of this access permissions model.

When SCTLR.AFE is set to 0, ensuring that the AP[0] bit is always set to 1 effectively changes the access model to the simpler model described in *AP[2:1] access permissions model* on page G5-5969.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G5-9 shows the full AP[2:0] access permissions model:

Table G5-9 VMSAv8-32 MMU access permissions

AP[2]	AP[1:0]	PL1 access	Unprivileged access	Description
0	00	No access	No access	All accesses generate Permission faults
	01	Read/write	No access	Access only at PL1
	10	Read/write	Read-only	Writes at EL0 generate Permission faults
	11	Read/write	Read/write	Full access
1	00	-	-	Reserved
	01	Read-only	No access	Read-only, only at PL1
	10	Read-only	Read-only	Read-only at any Exception level, deprecated[a]
	11	Read-only	Read-only	Read-only at any Exception level[b]

a. From VMSAv7, Arm strongly recommends use of the `0b11` encoding for Read-only at any Exception level.

b. This mapping was introduced in VMSAv7, and is reserved in earlier versions of the VMSA.

Note

- VMSAv8-32 supports the full set of access permissions shown in Table G5-9 only when SCTLR.AFE is set to 0. When SCTLR.AFE is set to 1, the only supported access permissions are those described in *AP[2:1] access permissions model* on page G5-5969.
- Some old documentation describes the AP[2] bit in the translation table entries as the APX bit.

G5.6.2 About the PAN bit

When the value of PSTATE.PAN is 1, any privileged data access from PL1 or EL2 to a virtual memory address that is accessible at EL0 generates a Permission fault.

When the value of PSTATE.PAN is 0, the translation system is the same as in Armv8.0.

A corresponding PAN bit is added to CPSR and SPSR for exception returns, and DSPSR for entry to and exit from Debug state.

A new SPAN bit is added to SCTLR that controls whether the PAN state bit is set on taking an exception to EL1 from either Secure or Non-secure state, or to EL3 from Secure state when EL3 is using AArch32.

CPSR.PAN bit can be written using an `MSR` instruction at PL1 or higher. Data writes to CPSR.PAN using an `MSR` instruction at EL0 are ignored. The value that is returned for an `MRS` instruction of CPSR from EL0 is UNKNOWN. In keeping with all other writes to the CPSR, other than for instruction fetches, the effect of the PAN state bit does not need to be explicitly synchronized.

The PAN state bit has no effect on:

- Data Cache instructions.
- Address translation instructions, other than `ATS1CPRP` and `ATS1CPWP` when ARMv8.2-ATS1E1 is implemented.
- Unprivileged instructions, `LDRBT`, `LDRHT`, `LDRT`, `LDRSBT`, `LDRSHT`, `STRBT`, `STRHT`, `STRT`, `STRSBT`, and `STRSHT`, unless HCR_EL2.{E2H, TGE} == {1, 0}.
- Instruction accesses.
- Manager domains.

The PAN bit has no effect when the first stage of translation is disabled for the current translation regime or when the first stage of translation for the current translation regime does not describe the permissions for access at EL0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If access is disabled, then the access will give rise to a stage 1 Permission fault.

On an exception taken from AArch32:

- CPSR.PAN is copied to SPSR_ELx.PAN, when the target Exception level is AArch64.
- CPSR.PAN is copied to SPSR.PAN, when the target Exception level is AArch32.

On an exception return from AArch32 to AArch32, SPSR.PAN is copied to CPSR.PAN.

On entry to Debug state, CPSR.PAN is copied to DSPSR.PAN.

On exit from Debug state, DSPSR.PAN is copied to CPSR.PAN.

The CPSR.PAN bit is not an Execution state bit.

Note

- In Non-debug state, in AArch32 state, software can use the `SETPAN #imm` instruction to modify PSTATE.PAN.
- In Debug state, in AArch32 state, a debugger can use the `ERET` instruction to perform a DRPS operation to modify PSTATE.PAN.

G5.6.3 Access permissions for instruction execution

Execute-never controls provide an additional level of control on memory accesses permitted by the access permissions settings. These controls are:

XN, Execute-never

Descriptor bit[54], defined as XN for:

- Stage 1 of any translation regime.
- Stage 2 translations when ARMv8.2-TTS2UXN is not implemented.

Note

XN[1:0], Execute-never, stage 2 only describes the stage 2 control when ARMv8.2-TTS2UXN is implemented.

This field applies to execution at any Exception level to which the stage of translation applies. A value of 0 indicates that this control permits execution.

PXN, Privileged execute-never, stage 1 only

Descriptor bit[53], used only for stage 1 of any translation regime for which the stage 1 translation can support two VA ranges:

- For stage 1 of a translation regime for which the stage 1 translation supports only a single VA range the stage 1 descriptors define a PXN field that is RES0, meaning it is ignored by hardware.

This field applies only to execution at an Exception level higher than EL0. A value of 0 indicates that this control permits execution.

XN[1:0], Execute-never, stage 2 only

Descriptor bits[54:53], defined as XN[1:0] for:

- Stage 2 translations when ARMv8.2-TTS2UXN is implemented.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G5-10 shows the operation of this control.

Table G5-10 XN[1:0] stage 2 access permissions model

XN[1]	XN[0]	Access
0	0	The stage 2 control permits execution at EL1 and EL0 if read access is permitted
0	1	The stage 2 control does not permit execution at EL1, but permits execution at EL0 if read access is permitted
1	0	The stage 2 control does not permit execution at EL1 or at EL0
1	1	The stage 2 control permits execution at EL1 if read access is permitted, but does not permit execution at EL0

Note

For stage 2 translations when ARMv8.2-TTS2UXN is not implemented, descriptor bit[53] is RES0, meaning it is ignored by hardware, and bit[54] is the XN control, see *XN, Execute-never* on page G5-5972.

Executing an instruction at ELx in a particular Security state generates a Permission fault unless all of the following are true for the instruction address:

- Any stage 1 execute-never control that applies to execution at ELx in the current Security state permits execution.
- If the translation regime that applies to ELx in the current Security state has two stages of translations, the stage 2 execute-never control that applies to execution at ELx permits execution.
- Read access is permitted.

However, if a stage 1 translation is using the Short-descriptor translation table format and the address is in a Managers domain the stage 1 access permissions are not checked, and therefore the access cannot cause a stage 1 Permission fault, see *Domains, Short-descriptor format only* on page G5-5976.

See also *Hyp mode control of Non-secure access permissions* on page G5-5977.

In addition, System register controls can enforce execute-never restrictions, regardless of the settings in the translation table XN and PXN fields, see:

- *Restriction on Secure instruction fetch* on page G5-5975.
- *Preventing execution from writable locations* on page G5-5975.

The execute-never controls apply also to speculative instruction fetching. This means a speculative instruction fetch from a memory region that is execute-never at the current level of privilege is prohibited.

The execute-never controls means that, when the stage of address translation is enabled, the PE can fetch, or speculatively fetch, an instruction from a memory location only if all of the following apply:

- If using the Short-descriptor translation table format, the translation table descriptor for the location does not indicate that it is in a No access domain.
- If using the Long-descriptor translation table format, or using the Short descriptor format and the descriptor indicates that the location is in a Client domain, in the descriptor for the location the following apply:
  - The stage 1 execute-never control for the Exception level at which the instruction is executed permits execution.
  - For a translation regime with two stages of address translation, the stage 2 execute-never control that applies to the Exception level at which the instruction is executed permits execution.
  - The access permissions permit a read access from the current PE mode.
- No other Prefetch Abort condition exists.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

- The PXN control applies to the PE privilege when it attempts to execute the instruction. In an implementation that fetches instructions speculatively, this might not be the privilege when the instruction was prefetched. Therefore, the architecture does not require the PXN control to prevent instruction fetching.
- Although the XN control applies to speculative fetching, on a speculative instruction fetch from an XN location, no Permission fault is generated unless the PE attempts to execute the instruction that would have been fetched from that location. This means that, if a speculative fetch from an XN location is attempted, but there is no attempt to execute the corresponding instruction, a Permission fault is not generated.
- The software that defines a translation table must mark any region of memory that is read-sensitive as XN, to avoid the possibility of a speculative fetch accessing the memory region. This means it must mark any memory region that corresponds to a read-sensitive peripheral as XN. Hardware does not prevent speculative accesses to a region of any Device memory type unless that region is also marked as execute-never for all Exception levels from which it can be accessed.
- When using the Short-descriptor translation table format, the XN attribute is not checked for domains marked as Manager. Therefore, the system must not include read-sensitive memory in domains marked as Manager, because the XN field does not prevent speculative fetches from a Manager domain.

When no stage of address translation for the translation regime is enabled, memory regions cannot have XN or PXN attributes assigned. *Behavior of instruction fetches when all associated address translations are disabled* on page G5-5932 describes how disabling all MMUs affects instruction fetching.

Hierarchical control of instruction fetching, Long-descriptor format

The Long-descriptor translation table format introduces a mechanism that means entries at one level of translation tables lookup can set limits on the permitted entries at subsequent levels of lookup. This applies to the restrictions on instruction fetching, and also to the access permissions described in *Hierarchical control of access permissions, Long-descriptor format* on page G5-5970.

Note

Similar hierarchical controls apply to data accesses, see *Hierarchical control of access permissions, Long-descriptor format* on page G5-5970.

However, in an implementation that includes ARMv8.2-AA32HPD, when hierarchical control of instruction fetching is disabled for a translation regime, the information in this subsection does not apply. See *Attribute fields in VMSAv8-32 Long-descriptor translation table format descriptors* on page G5-5952.

The restrictions apply only to subsequent levels of lookup at the same stage of translation, and:

- XNTable restricts the XN control:
  - When XNTable is set to 1, the XN field is treated as 1 in all subsequent levels of lookup, regardless of the actual value of the field.
  - When XNTable is set to 0 it has no effect.
- PXNTable restricts the PXN control:
  - When PXNTable is set to 1, the PXN field is treated as 1 in all subsequent levels of lookup, regardless of the actual value of the field.
  - When PXNTable is set to 0 it has no effect.

Note

The XNTable and PXNTable settings are combined with the XN and PXN fields in the translation table descriptors accessed at subsequent levels of lookup. They do not restrict or change the values entered in those descriptors.

The XNTable and PXNTable controls are provided only in the Long-descriptor translation table format, and only for stage 1 translations. The corresponding bits are SBZ in the stage 2 translation table descriptors.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The effect of XNTable or PXNTable applies to later entries in the translation table walk, and so its effects can be held in one or more TLB entries. Therefore, a change to XNTable or PXNTable requires coarse-grained invalidation of the TLB to ensure that the effect of the change is visible to subsequent memory transactions.

Preventing execution from writable locations

Armv8 provides control bits that, when the corresponding stage 1 address translation is enabled, force writable memory to be treated as XN or PXN, regardless of the value of the XN or PXN field. When the translation stages are controlled by an Exception level that is using AArch32:

- For PL1&0 stage 1 translations, when SCTLR.WXN is set to 1, all regions that are writable at stage 1 of the address translation are treated as XN.
- For PL1&0 stage 1 translations, when SCTLR.UWXN is set to 1, an instruction fetch is treated as accessing a PXN region if it accesses a region that software executing at EL0 can write to.
- For Non-secure EL2 stage 1 translations, when HSCTLR.WXN is set to 1, all regions that are writable at stage 1 of the address translation are treated as XN.

Note

- The SCTLR.WXN controls are intended to be used in systems with very high security requirements.
- Setting a WXN or UWXN bit to 1 changes the interpretation of the translation table entry, overriding a zero value of an XN or PXN field. It does not cause any change to the translation table entry.

For any given virtual machine, Arm expects WXN and UWXN to remain static in normal operation. In particular, it is IMPLEMENTATION DEFINED whether TLB entries associated with a particular VMID reflect the effect of the values of these fields. A generic sequence to ensure synchronization of a change to these fields, when that change is made without a corresponding change of VMID, is:

```
Change the WXN or UWXN bit
ISB                     ; This ensures synchronization of the change
Invalidate entire TLB of associated entries
DSB                     ; This completes the TLB Invalidation
ISB                     ; This ensures instruction synchronization
```

As with all Permission fault checking, if the stage 1 translation is using the Short-descriptor translation table format, the permission checks are performed only for Client domains. For more information, see *About access permissions* on page G5-5968.

For more information about address translation, see *About address translation for VMSAv8-32* on page G5-5925.

Restriction on Secure instruction fetch

EL3 provides a Secure instruction fetch bit, SCR.SIF. When this bit is to 1, any attempt in Secure state to execute an instruction fetched from Non-secure physical memory causes a Permission fault. As with all Permission fault checking, when using the Short-descriptor format translation tables the check applies only to Client domains, see *About access permissions* on page G5-5968.

Arm expects SCR.SIF to be static during normal operation. In particular, whether the TLB holds the effect of the SIF bit is IMPLEMENTATION DEFINED. The generic sequence to ensure visibility of a change to the SIF bit is:

```
Change the SCR.SIF bit
ISB                     ; This ensures synchronization of the change
Invalidate entire TLB
DSB                     ; This completes the TLB Invalidation
ISB                     ; This ensures instruction synchronization
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G5.6.4 Domains, Short-descriptor format only

A domain is a collection of memory regions. The Short-descriptor translation table format supports 16 domains, and requires the software that defines a translation table to assign each VMSAv8-32 memory region to a domain. When using the Short-descriptor format:

- Level 1 translation table entries for translation tables and Sections include a domain field.
- Translation table entries for Supersections do not include a domain field. The Short-descriptor format defines Supersections as being in domain 0.
- Level 2 translation table entries inherit a domain setting from the parent level 1 translation table descriptor.
- Each TLB entry includes a domain field.

The domain field specifies which of the 16 domains the entry is in, and a two-bit field in the DACR defines the permitted access for each domain. The possible settings for each domain are:

No access Any access using the translation table descriptor generates a Domain fault.

Clients On an access using the translation table descriptor, the access permission attributes are checked. Therefore, the access might generate a Permission fault.

Managers On an access using the translation table descriptor, the access permission attributes are not checked. Therefore, the access cannot generate a Permission fault.

See *The MMU fault-checking sequence* on page G5-6016 for more information about how, when using the Short-descriptor translation table format, the Domain attribute affects the checking of the other attributes in the translation table descriptor.

Note

A single program might:

- Be a Client of some domains.
- Be a Manager of some other domains.
- Have no access to the remaining domains.

The Long-descriptor translation table format does not support domains. When a stage of translation is using this format, all memory is treated as being in a Client domain, and the settings in the DACR are ignored.

G5.6.5 The Access flag

The Access flag indicates when a page or section of memory is accessed for the first time since the Access flag in the corresponding translation table descriptor was set to 0:

- If address translation is using the Short-descriptor translation table format, it must set SCTLR.AFE to 1 to enable use of the Access flag. Setting this bit to 1 redefines the AP[0] bit in the translation table descriptors as an Access flag, and limits the access permissions information in the translation table descriptors to AP[2:1], as described in *AP[2:1] access permissions model* on page G5-5969.
- The Long-descriptor format always supports an Access flag bit in the translation table descriptors, and address translation using this format behaves as if SCTLR.AFE is set to 1, regardless of the value of that bit.

In Armv8.0, the Access flag is managed by software as described in *Software management of the Access flag* on page G5-5977.

Note

Previous version of the Arm architecture optionally supported hardware management of the Access flag. Armv8.0 obsoletes this option. However, ARMv8.1-TTHM provides a new mechanism for hardware management of the Access flag, that is supported only for the VMSAv8-64 translation regimes.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Software management of the Access flag

Armv8.0 requires that software manages the Access flag. This means an Access flag fault is generated whenever an attempt is made to read into the TLB a translation table descriptor entry for which the value of the Access flag is 0.

Note

When using the Short-descriptor translation table format, Access flag faults are generated only if SCTLR.AFE is set to 1, to enable use of a translation table descriptor bit as an Access flag.

The Access flag mechanism expects that, when an Access flag fault occurs, software resets the Access flag to 1 in the translation table entry that caused the fault. This prevents the fault occurring the next time that memory location is accessed. Entries with the Access flag set to 0 are never held in the TLB, meaning software does not have to flush the entry from the TLB after setting the flag.

Note

If a system incorporates components that can autonomously update translation table entries that are shared with the Arm PE, then the software must be aware of the possibility that such components can update the access flag autonomously.

In such a system, system software should perform any changes of translation table entries with an Access flag of 0, other than changes to the Access flag value, by using an Load-Exclusive/Store-Exclusive loop, to allow for the possibility of simultaneous updates.

G5.6.6 Hyp mode control of Non-secure access permissions

When EL2 is using AArch32, Non-secure software executing in Hyp mode controls two sets of translation tables, both of which use the Long-descriptor translation table format:

- The translation tables that control the Non-secure EL2 stage 1 translations. These map VAs to PAs, for memory accesses made when executing in Non-secure state in Hyp mode, and are indicated and controlled by the HTTBR and HTCR.

  These translations have similar access controls to other Non-secure stage 1 translations using the Long-descriptor translation table format, as described in:

  — *AP[2:1] access permissions model* on page G5-5969.

  — *Access permissions for instruction execution* on page G5-5972.

  The differences from the Non-secure stage 1 translations are that:

  — The APTable[0], PXNTable, and PXN bits are reserved, SBZ.

  — AP[1] is reserved, SBO.

- The translation tables that control the Non-secure PL1&0 stage 2 translations. These map the IPAs from the stage 1 translation onto PAs, for memory accesses made when executing in Non-secure state at PL1 or EL0, and are indicated and controlled by the VTTBR and VTCR.

  The descriptors in the virtualization translation tables define stage 2 access permissions, that are combined with the permissions defined in the stage 1 translation. This section describes this combining of access permissions.

Note

The level 2 access permissions mean a hypervisor can define additional access restrictions to those defined by a Guest OS in the stage 1 translation tables. For a particular access, the actual access permission is the more restrictive of the permissions defined by:

- The Guest OS, in the stage 1 translation tables.
- The hypervisor, in the stage 2 translation tables.

The stage 2 access controls defined from Hyp mode:

- Affect only the Non-secure stage 1 access permissions settings.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Take no account of whether the accesses are from a Non-secure PL1 mode or a Non-secure EL0 mode.
- Permit software executing in Hyp mode to assign a write-only attribute to a memory region.

The S2AP field in the stage 2 descriptors define the stage 2 access permissions, as Table G5-11 shows:

Table G5-11 Stage 2 control of access permissions

S2AP	Access permission
00	No access permitted
01	Read-only. Writes to the region are not permitted, regardless of the stage 1 permissions.
10	Write-only. Reads from the region are not permitted, regardless of the stage 1 permissions.
11	Read/write. The stage 1 permissions determine the access permissions for the region.

For more information about the S2AP field, see *Attribute fields in VMSAv8-32 Long-descriptor stage 2 Block and Page descriptors* on page G5-5955.

If the stage 2 permissions cause a Permission fault, this is a stage 2 MMU fault. Stage 2 MMU faults are taken to Hyp mode, and reported in the HSR using an EC code of `0x20` or `0x24`. For more information, see *Use of the HSR* on page G5-6038.

Note

In the HSR, the combination of the EC code and the DFSC or IFSC value in the ISS indicate that the fault is a stage 2 MMU fault.

The stage 2 permissions include an XN attribute. If this identifies the region as execute-never, execution from the region is not permitted, regardless of the value of the XN or UXN attribute in the stage 1 translation. If a Permission fault is generated because the stage 2 XN field identifies the region as execute-never, this is reported as a stage 2 MMU fault.

Note

The stage 2 XN attribute:

- Is a single bit if ARMv8.2-TTS2UXN is not implemented, see *XN, Execute-never* on page G5-5972.
- Is a 2-bit field if ARMv8.2-TTS2UXN is implemented, see *XN[1:0], Execute-never, stage 2 only* on page G5-5972.

AArch32 state prioritization of synchronous aborts from a single stage of address translation on page G5-6021 describes the abort prioritization if both stages of a translation generate a fault.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G5.7 Memory region attributes

In addition to an output address, a translation table entry that refers to a page or region of memory includes fields that define properties of that target memory region. *Information returned by a translation table lookup* on page G5-5935 describes the classification of those fields as address map control, access control, and memory attribute fields. The memory region attribute fields control the memory type, Cacheability, and Shareability of the region.

The following sections describe the assignment of memory region attributes for stage 1 translations:

- *Overview of memory region attributes for stage 1 translations*.
- *Short-descriptor format memory region attributes, without TEX remap* on page G5-5980.
- *Short-descriptor format memory region attributes, with TEX remap* on page G5-5982.
- *VMSAv8-32 Long-descriptor format memory region attributes* on page G5-5985.

For an implementation that is operating in Secure state, or in Hyp mode, these assignments define the memory attributes of the accessed region.

For an implementation that is operating in a Non-secure PL1 or EL0 mode, the Non-secure PL1&0 stage 2 translation can modify the memory attributes assigned by the stage 1 translation. *EL2 control of Non-secure memory region attributes* on page G5-5987 describes these stage 2 assignments.

G5.7.1 Overview of memory region attributes for stage 1 translations

The description of the memory region attributes in a translation descriptor divides into:

Memory type and attributes

These are described either:

- Directly, by bits in the translation table descriptor.
- Indirectly, by registers referenced by bits in the table descriptor. This is described as *remapping* the memory type and attribute description.

The Short-descriptor translation table format can use either of these approaches, selected by the SCTLR.TRE bit:

TRE == 0 Remap disabled. The TEX[2:0], C, and B bits in the translation table descriptor define the memory region attributes. *Short-descriptor format memory region attributes, without TEX remap* on page G5-5980 describes this encoding.

Note

With the Short-descriptor format, remapping is called *TEX remap*, and the SCTLR.TRE bit is the *TEX remap enabled* bit.

The description of the TRE == 0 encoding includes information about the encoding in previous versions of the architecture.

TRE == 1 Remap enabled. The TEX[0], C, and B bits in the translation table descriptor are index bits to the remap registers, that define the memory region attributes:

- The Primary Region Remap Register, PRRR.
- The Normal Memory Remap Register, NMRR.

Short-descriptor format memory region attributes, with TEX remap on page G5-5982 describes this encoding scheme.

This scheme reassigns translation table descriptor bits TEX[2:1] for use as bits managed by the operating system.

The Long-descriptor translation table format always uses remapping. This means that when the value of TTBCR.EAE is 1, enabling use of the Long-descriptor translation table format, SCTLR.TRE is RES1.

VMSAv8-32 Long-descriptor format memory region attributes on page G5-5985 describes this encoding.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Shareability In the Short-descriptor translation table format, the S bit in the translation table descriptor is used in determining the Shareability of the region. How the S bit is interpreted depends on whether TEX remap is enabled, see:

- *Shareability and the S bit, without TEX remap* on page G5-5981.
- *Determining the Shareability, with TEX remap* on page G5-5983.

In the Long-descriptor translation table format, the SH[1:0] field in the translation table descriptor encodes the Shareability of the region, see *Shareability, Long-descriptor format* on page G5-5985.

Note

Shareability is one of Non-shareable, Inner Shareable, and Outer Shareable. However, when using the Short-descriptor translation table format without TEX remap, VMSAv8-32 does not support any distinction between Inner Shareable and Outer Shareable memory, and a memory region is either Non-shareable or Outer Shareable.

G5.7.2 Short-descriptor format memory region attributes, without TEX remap

When using the Short-descriptor translation table formats, TEX remap is disabled when the value of SCTLR.TRE is 0.

Note

- The Short-descriptor format scheme without TEX remap is the scheme used in VMSAv6.
- The B (Bufferable), C (Cacheable), and TEX (Type extension) bit names are inherited from earlier versions of the architecture. These names no longer adequately describe the function of the B, C, and TEX bits.

Table G5-12 shows the C, B, and TEX[2:0] encodings when TEX remap is disabled. In the *Page Shareability* column, an entry of *S bit* indicates that the S bit in the translation table descriptor determines the Shareability, see *Shareability and the S bit, without TEX remap* on page G5-5981.

Table G5-12 TEX, C, and B encodings when TRE == 0

TEX[2:0]	C	B	Description	Memory type	Page Shareability
000	0	0	Device-nGnRnE	Device-nGnRnE	Outer Shareable
		1	Device-nGnRE[a]	Device-nGnRE	Outer Shareable[a]
	1	0	Outer and Inner Write-Through, Read-Allocate No Write-Allocate	Normal	S bit
		1	Outer and Inner Write-Back, Read-Allocate No Write-Allocate	Normal	S bit
001	0	0	Outer and Inner Non-cacheable	Normal	Outer Shareable[b]
		1	Reserved	-	-
	1	0	IMPLEMENTATION DEFINED	IMPLEMENTATION DEFINED	IMPLEMENTATION DEFINED
		1	Outer and Inner Write-Back, Read-Allocate Write-Allocate	Normal	S bit
010	0	0	Device-nGnRE[a]	Device-nGnRE	Outer Shareable[a]
		1	Reserved	-	-
	1	x	Reserved	-	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G5-12 TEX, C, and B encodings when TRE == 0 (continued)

TEX[2:0]	C	B	Description		Memory type	Page Shareability
011	x	x	Reserved		-	-
1BB	A	A	Cacheable memory:	AA = Inner attribute[c] BB = Outer attribute	Normal	S bit

a. In Armv8, all Device memory types are Outer Shareable. For the Device-nGnRE memory type this is a change from previous versions of the architecture. This is why Device-nGnRE memory has two entries in this table. For compatibility with Armv7 software should use the {TEX, C, B} values {000, 0, 1}.

b. In Armv8, Normal Inner Non-cacheable, Outer Non-cacheable memory is always Outer Shareable. This is a change from previous versions of the architecture, where the S bit determined the Shareability. For compatibility with Armv7 software should set S to 1.

c. For more information, see *Cacheability attributes, without TEX remap*.

See *Memory types and attributes* on page E2-4010 for an explanation of Normal memory, and the types of Device memory, and of the Shareability attribute.

Cacheability attributes, without TEX remap

When the value of TEX[2] is 0, the same Cacheability attribute applies to Inner Cacheable and Outer Cacheable memory regions, and the {TEX[1:0], C, B} values identify this attribute, as Table G5-12 on page G5-5980 shows.

When the value of TEX[2] is 1, the memory described by the translation table entry is cacheable, and the rest of the encoding defines the Inner Cacheability and Outer Cacheability attributes:

TEX[1:0] Define the Outer Cacheability attribute.

C, B Define the Inner Cacheability attribute.

The translation table entries use the same encoding for the Outer and Inner Cacheability attributes, as Table G5-13 shows.

Table G5-13 Inner and Outer Cacheability attribute encoding

Encoding	Cacheability attribute
00	Non-cacheable
01	Write-Back, Read-Allocate Write-Allocate
10	Write-Through, Read Allocate No Write-Allocate
11	Write-Back, Read Allocate No Write-Allocate

Shareability and the S bit, without TEX remap

The Short-descriptor format translation table entries include an S bit. This bit:

- Is ignored if the entry refers to any type of Device memory, or to Normal memory that is Inner Non-cacheable, Outer Non-cacheable.
- For Normal memory that is not Inner Non-cacheable, Outer Non-cacheable, determines whether the memory region is Outer Shareable or Non-shareable:

  S == 0 Normal memory region is Non-shareable.

  S == 1 Normal memory region is Outer Shareable.

Without TEX remapping there is no distinction between Inner Shareable and Outer Shareable memory, meaning the S bit determines whether the region is Non-shareable or Outer Shareable.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G5.7.3 Short-descriptor format memory region attributes, with TEX remap

When using the Short-descriptor translation table formats, TEX remap is enabled when the value of SCTLR.TRE is 1. In this configuration:

- The software that defines the translation tables must program the PRRR and NMRR to define seven possible memory region attributes.
- The TEX[0], C, and B bits of the translation table descriptors define the memory region attributes, by indexing PRRR and NMRR.
- Hardware makes no use of TEX[2:1], see *The OS managed translation table bits* on page G5-5984.

When TEX remap is enabled:

- For seven of the eight possible combinations of the TEX[0], C and B bits, fields in the PRRR and NMRR define the region attributes, as described in this section.
- The meaning of the eighth combination for the TEX[0], C and B bits is IMPLEMENTATION DEFINED.
- If the TEX[0], C and B bits determine that the region is a Device memory type, or is Normal Inner Non-cacheable, Outer Non-cacheable, then the region is Outer Shareable. Otherwise, the Shareability is determined by the combination of:
  - The S bit from the translation table descriptor.
  - The value of the PRRR.NS0 or PRRR.NS1 bit.
  - The value of the appropriate PRRR.NOS*n* bit, as shown in Table G5-14.

  For more information, see *Determining the Shareability, with TEX remap* on page G5-5983.

For each of the possible encodings of the TEX[0], C, and B bits in a translation table entry, Table G5-14 shows which fields of the PRRR and NMRR registers describe the memory region attributes.

Table G5-14 TEX, C, and B encodings when TRE == 1

Encoding			Memory type[a]	Cache attributes[a, b]:		Outer Shareable attribute[a, c]
TEX[0]	C	B		Inner cacheability	Outer cacheability	
0	0	0	PRRR.TR0	NMRR.IR0	NMRR.OR0	NOT(PRRR.NOS0)
		1	PRRR.TR1	NMRR.IR1	NMRR.OR1	NOT(PRRR.NOS1)
	1	0	PRRR.TR2	NMRR.IR2	NMRR.OR2	NOT(PRRR.NOS2)
		1	PRRR.TR3	NMRR.IR3	NMRR.OR3	NOT(PRRR.NOS3)
1	0	0	PRRR.TR4	NMRR.IR4	NMRR.OR4	NOT(PRRR.NOS4)
		1	PRRR.TR5	NMRR.IR5	NMRR.OR5	NOT(PRRR.NOS5)
	1	0	IMPLEMENTATION DEFINED			
		1	PRRR.TR7	NMRR.IR7	NMRR.OR7	NOT(PRRR.NOS7)

a. For details of the *Memory type* and *Outer Shareable* encodings see the description of the PRRR. For details of the *Cache attributes* encodings the description of the NMRR.

b. Applies only if the memory type for the region is mapped as Normal memory.

c. Applies only if both of the following apply:
The memory type for the region is mapped as Normal memory that is not Inner Non-cacheable and Outer Non-cacheable.
The region is not Non-shareable.
See *Determining the Shareability, with TEX remap* on page G5-5983.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

As Table G5-14 on page G5-5982 shows, when TEX remap is enabled, for a given set of {TEX[0], C, B} bits from a translation table descriptor:

1. The primary mapping, to memory type, is given by the PRRR.TR*n* field as shown in the Memory type column.
2. For any region that the PRRR.TR*n* maps as Normal memory, NMRR.IR*n* determines the Inner cacheability attribute, and NMRR.OR*n* determines the Outer cacheability attribute.
3. For a region that PRRR.TR*n* maps as Normal memory, if NMRR.{IR*n*, OR*n*} do not map the region as Inner Non-cacheable, Outer Non-cacheable, PRRR.{NS0, NS1} and PRRR.NOS*n* are used to determine the Shareability of the region, see *Determining the Shareability, with TEX remap*.

The TEX remap registers and the SCTLR.TRE bit are banked between the Secure and Non-secure Security states. For more information, see *The effect of EL3 on TEX remap* on page G5-5985.

The TEX remap registers must be static during normal operation. In particular, when the remap registers are changed:

- It is IMPLEMENTATION DEFINED when the changes take effect.
- It is CONSTRAINED UNPREDICTABLE whether the TLB caches the effect of the TEX remap on translation tables, see *CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values* on page K1-7835.

The software sequence to ensure the synchronization of changes to the TEX remap registers is:

1. Execute a `DSB` instruction. This ensures any memory accesses using the old mapping have completed.
2. Write the TEX remap registers or SCTLR.TRE bit.
3. Execute an `ISB` instruction. This ensures synchronization of the register updates.
4. Invalidate the entire TLB.
5. Execute a `DSB` instruction. This ensures completion of the entire TLB operation.
6. Clean and invalidate all caches. This removes any cached information associated with the old mapping.
7. Execute a `DSB` instruction. This ensures completion of the cache maintenance.
8. Execute an `ISB` instruction. This ensures instruction synchronization.

This extends the standard rules for the synchronization of changes to System registers described in *Synchronization of changes to AArch32 System registers* on page G8-6100, and provides implementation freedom as to whether or not the effect of the TEX remap is cached.

Determining the Shareability, with TEX remap

The memory type of a region, as indicated in the Memory type column of Table G5-14 on page G5-5982, provides the first level of control of the Shareability of the region:

- If the memory is any type of Device memory, then the region is Outer Shareable, and any Shareability attributes in the translation table descriptor and PRRR for that region are ignored.

  This applies also to a Normal memory region that the NMRR attributes identify as Inner Non-cacheable and Outer Non-cacheable,

- If using the Short descriptor translation table format then the Shareability of the region is determined using the value of the S bit in the translation table descriptor to index one of the PRRR.{NS1. NS0} bits, as described in this section.

Table G5-15 shows how the translation table S bit indexes into the PRRR:

Table G5-15 Determining the Shareability attribute, with TEX remap

Memory type	Remapping when S == 0	Remapping when S == 1
Device or Normal Inner Non-cacheable, Outer Non-cacheable	Outer Shareable	Outer Shareable
Normal, not Inner Non-cacheable, Outer Non-cacheable	PRRR.NS0	PRRR.NS1

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For a Normal memory region that is not Inner Non-cacheable, Outer Non-cacheable, the appropriate bit of the PRRR indicates whether the region is Non-shareable, as follows:

PRRR.NS*n*==0 Non-shareable.

PRRR.{NOS7:NOS0} are ignored.

PRRR.NS*n*==1 The appropriate PRRR.NOS*m* field, as shown in Table G5-14 on page G5-5982, indicates whether the region is Inner Shareable or Outer Shareable:

PRRR.NOS*m*==0 Region is Outer Shareable.

PRRR.NOS*m*==1 Region is Inner Shareable.

Note

This means that TEX remapping can map a translation table entry with S == 0 as shareable memory.

SCTLR.TRE, SCTLR.M, and the effect of the TEX remap registers

When TEX remap is disabled, because the value of the SCTLR.TRE bit is 0:

- The effect of the PRRR and NMRR registers can be IMPLEMENTATION DEFINED.
- The interpretation of the fields of the PRRR and NMRR registers can differ from the description given earlier in this section. One implication of this is that the implementation can provide an IMPLEMENTATION DEFINED mechanism to interpret the PRRR.{NOS7:NOS0} fields.

VMSAv8-32 requires that the effect of these registers is limited to remapping the attributes of memory locations. These registers must not change whether any cache hardware or stages of address translation are enabled. The mechanism by which the TEX remap registers have an effect when the value of the SCTLR.TRE bit is 0 is IMPLEMENTATION DEFINED. The AArch32 architecture requires that from reset, if the IMPLEMENTATION DEFINED mechanism has not been invoked:

- If the PL1&0 stage 1 address translation is enabled and is using the Short-descriptor format translation tables, the architecturally-defined behavior of the TEX[2:0], C, and B bits must apply, without reference to the TEX remap functionality. In other words, memory attribute assignment must comply with the scheme described in *Short-descriptor format memory region attributes, without TEX remap* on page G5-5980.
- If the PL1&0 stage 1 address translation is disabled, then the architecturally-defined behavior of VMSAv8-32 with address translation disabled must apply, without reference to the TEX remap functionality. See *The effects of disabling address translation stages on VMSAv8-32 behavior* on page G5-5930.

Possible mechanisms for enabling the IMPLEMENTATION DEFINED effect of the TEX remap registers when the value of SCTLR.TRE is 0 include:

- A control bit in the ACTLR, or in an IMPLEMENTATION DEFINED System register.
- Changing the behavior when the PRRR and NMRR registers are changed from their IMPLEMENTATION DEFINED reset values.

In addition, if the stage of address translation is disabled and the value of the SCTLR.TRE bit is 1, the architecturally-defined behavior of the VMSAv8-32 with the stage of address translation disabled must apply without reference to the TEX remap functionality.

In an implementation that includes EL3, the IMPLEMENTATION DEFINED effect of these registers must only take effect in the Security state of the registers. See also *The effect of EL3 on TEX remap* on page G5-5985.

The OS managed translation table bits

When TEX remap is enabled, the TEX[2:1] bits in the translation table descriptors are available as two bits that can be managed by the operating system. In VMSAv8-32, as long as the SCTLR.TRE bit is set to 1, the values of the TEX[2:1] bits are IGNORED by the PE. Software can write any value to these bits in the translation tables.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The effect of EL3 on TEX remap

In an implementation that includes EL3, when EL3 is using AArch32, the TEX remap registers are banked between the Secure and Non-secure Security states. When EL3 is using AArch32, write accesses to the Secure register for the current security state apply to all PL1&0 stage 1 translation table lookups in that state. The SCTLR.TRE bit is banked in the Secure and Non-secure copies of the register, and the appropriate version of this bit determines whether TEX remap is applied to translation table lookups in the current security state.

Write accesses to the Secure copies of the TEX remap registers are disabled when the **CP15SDISABLE** input is asserted HIGH, meaning the `MCR` operations to access these registers are UNDEFINED. For more information, see *The CP15SDISABLE and CP15SDISABLE2 input signals* on page G5-6056.

G5.7.4 VMSAv8-32 Long-descriptor format memory region attributes

When a PE is using the VMSAv8-32 Long-descriptor translation table format, the AttrIndx[2:0] field in a block or page translation table descriptor for a stage 1 translation indicates the 8-bit field, in the appropriate MAIR register, that specifies the attributes for the corresponding memory region, as follows:

- AttrIndx[2] indicates the MAIR register to be used:

  AttrIndx[2] == 0 Use MAIR0.

  AttrIndx[2] == 1 Use MAIR1.

- AttrIndx[2:0] indicates the required Attr field, Attr*n*, where *n* = AttrIndx[2:0].

Each AttrIndx field defines, for the corresponding memory region:

- The memory type, Normal or a type of Device memory.
- For Normal memory:
  - The Inner cacheability and Outer cacheability attributes, each of which is one of Non-cacheable, Write-Through Cacheable, or Write-Back Cacheable.
  - For Write-Through Cacheable and Write-Back Cacheable regions, the Read-Allocate and Write-Allocate policy hints, each of which is *Allocate* or *No allocate*.

For more information about the AttrIndx[2:0] descriptor field, see *Attribute fields in VMSAv8-32 Long-descriptor stage 1 Block and Page descriptors* on page G5-5953.

Shareability, Long-descriptor format

When a PE is using the Long-descriptor translation table format, the SH[1:0] field in a block or page translation table descriptor specifies the Shareability attributes of the corresponding memory region, if the MAIR entry for that region identifies it as Normal memory that is not both Inner Non-cacheable and Outer Non-cacheable. Table G5-16 shows the encoding of this field.

Table G5-16 SH[1:0] field encoding for Normal memory, Long-descriptor format

SH[1:0]	Normal memory
00	Non-shareable
01	Reserved, CONSTRAINED UNPREDICTABLE, see *Reserved values in System and memory-mapped registers and translation table entries* on page K1-7853 for the permitted behavior.
10	Outer Shareable
11	Inner Shareable

See *Combining the Shareability attribute* on page G5-5990 for constraints on the Shareability attributes of a Normal memory region that is Inner Non-cacheable, Outer Non-cacheable.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For any type of Device memory, and for Normal Inner Non-cacheable, Outer Non-cacheable memory, the value of the SH[1:0] field of the translation table descriptor is ignored.

Other fields in the Long-descriptor translation table format descriptors

The following subsections describe the other fields in the translation table block and page descriptors when a PE is using the Long-descriptor translation table format:

- *Contiguous bit*
- *IGNORED fields*.
- *Field reserved for software use* on page G5-5987

Contiguous bit

The Long-descriptor translation table format descriptors contain a Contiguous bit. Setting this bit to 1 indicates that 16 adjacent translation table entries point to a *contiguous output address range*. These 16 entries must be aligned in the translation table so that the top five bits of their input addresses, that index their position in the translation table, are the same. For example, to use this bit for a block of 16 entries in the level 3 translation table, bits[20:16] of the input addresses for the 16 entries must be the same.

The contiguous output address range must be aligned to size of 16 translation table entries at the same translation table level.

Use of this bit means that the TLB can cache a single entry to cover the 16 translation table entries.

This bit acts as a hint. The architecture does not require a PE to cache TLB entries in this way. To avoid TLB coherency issues, any TLB maintenance by address must not assume any optimization of the TLB tables that might result from use of this bit.

Note

The use of the contiguous bit is similar to the approach used, in the Short-descriptor translation table format, for optimized caching of Large Pages and Supersections in the TLB. However, an important difference in the contiguous bit capability is that TLB maintenance must be performed based on the size of the underlying translation table entries, to avoid TLB coherency issues. That is, any use of the contiguous bit has no effect on the minimum size of entry that must be invalidated from the TLB.

IGNORED fields

In the VMSAv8-32 translation table long-descriptor format, the following fields are defined as IGNORED, meaning the architecture guarantees that a PE makes no use of these fields:

- In the stage 1 and stage 2 Table descriptors, bits[58:52] and bits[11:2].
- In the stage 1 and stage 2 Block and Page descriptors, bit[63] and bits[58:55].
- In the stage 1 and stage 2 Block and Page descriptors in an implementation that does not include ARMv8.2-TTPBHA, bits[62:59].

Of these fields:

- In the stage 1 and stage 2 block and page descriptors, bits[58:55] are reserved for software use, see *Field reserved for software use* on page G5-5987.
- In the stage 2 block and page descriptors:
  - Bit[63] is reserved for use by a System MMU.
  - In an implementation that does not include ARMv8.2-TTPBHA, bits[62:59] are reserved for use by a System MMU.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Field reserved for software use

The architecture reserves a 4-bit IGNORED field in the Block and translation table descriptors, bits[58:55], for software use. In considering migration from using the Short-descriptor format to the Long-descriptor format, this field is an extension of the Short-descriptor field described in *The OS managed translation table bits* on page G5-5984.

Note

The definition of IGNORED means there is no need to invalidate the TLB if these bits are changed.

G5.7.5 EL2 control of Non-secure memory region attributes

Software executing at EL2 controls two sets of translation tables, both of which use the Long-descriptor translation table format. These are:

- The translation tables that control Non-secure EL2 stage 1 translations. These map VAs to PAs, and when EL2 is using AArch32 they are indicated and controlled by the HTTBR and HTCR.

  These translations have exactly the same memory region attribute controls as any other stage 1 translations, as described in *VMSAv8-32 Long-descriptor format memory region attributes* on page G5-5985.

- The translation tables that control Non-secure PL1&0 stage 2 translations. These map the IPAs from the stage 1 translation onto PAs, and are indicated and when EL2 is using AArch32 they are controlled by the VTTBR and VTCR.

  The descriptors in the virtualization translation tables define level 2 memory region attributes, that are combined with the attributes defined in the stage 1 translation. This section describes this combining of attributes.

VMSAv8-32 Long-descriptor translation table format descriptors on page G5-5949 describes the format of the entries in these tables.

Note

In a virtualization implementation, a hypervisor might usefully:

- Reduce the permitted Cacheability of a region.
- Increase the required Shareability of a region.

The combining of attributes from stage 1 and stage 2 translations supports both of these options.

In the stage 2 translation table descriptors for memory regions and pages, the MemAttr[3:0] and SH[1:0] fields describe the stage 2 memory region attributes:

- The definition of the stage 2 SH[1:0] field is identical to the same field for a stage 1 translation, see *Shareability, Long-descriptor format* on page G5-5985.

- MemAttr[3:2] give a top-level definition of the memory type, and of the cacheability of a Normal memory region, as Table G5-17 shows:

Table G5-17 Long-descriptor MemAttr[3:2] encoding, stage 2 translation

MemAttr[3:2]	Memory type	Cacheability
00	Device, of type determined by MemAttr[1:0]	Not applicable
01	Normal, Inner cacheability determined by MemAttr[1:0]	Outer Non-cacheable
10		Outer Write-Through Cacheable
11		Outer Write-Back Cacheable

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The encoding of MemAttr[1:0] depends on the Memory type indicated by MemAttr[3:2]:

— When MemAttr[3:2]==`0b00`, indicating a type of Device memory, Table G5-18 shows the encoding of MemAttr[1:0]:

Table G5-18 MemAttr[1:0] encoding for the types of Device memory

MemAttr[1:0]	Meaning when MemAttr[3:2] == `0b00`
00	Region is Device-nGnRnE memory
01	Region is Device-nGnRE memory
10	Region is Device-nGRE memory
11	Region is Device-GRE memory

— When MemAttr[3:2]!=`0b00`, indicating Normal memory, Table G5-19 shows the encoding of MemAttr[1:0]:

Table G5-19 MemAttr[1:0] encoding for Normal memory

MemAttr[1:0]	Meaning when MemAttr[3:2] != `0b00`
00	Reserved, CONSTRAINED UNPREDICTABLE, See *Reserved values in System and memory-mapped registers and translation table entries* on page K1-7853 for the permitted behavior.
01	Inner Non-cacheable
10	Inner Write-Through Cacheable
11	Inner Write-Back Cacheable

Note

The stage 2 translation does not assign any allocation hints.

The following sections describe how the memory type attributes assigned at stage 2 of the translation are combined with those assigned at stage 1:

- *Combining the memory type attribute* on page G5-5989.
- *Combining the Cacheability attribute* on page G5-5989.
- *Combining the Shareability attribute* on page G5-5990.

Note

- The following stage 2 translation table attribute settings leave the stage 1 settings unchanged:
  — MemAttr[3:2] == `0b11`, Normal memory, Outer Write-Back Cacheable.
  — MemAttr[1:0] == `0b11`, Inner Write-Back Cacheable.
- In addition to the attribute combinations described in this section, *Access permissions for instruction execution* on page G5-5972 describes how the stage 1 and stage 2 execute-never permission fields are combined, so that a region is execute-never if it is defined as execute-never in at least one stage of translation.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Combining the memory type attribute

Table G5-20 shows how the stage 1 and stage 2 memory type assignments are combined:

Table G5-20 Combining the stage 1 and stage 2 memory type assignments

Assignment in stage 1	Assignment in stage 2	Resultant type
Device-nGnRnE	Any	Device-nGnRnE
Device-nGnRE	Device-nGnRnE	Device-nGnRnE
	Not Device-nGnRnE	Device-nGnRE
Device-nGRE	Device-nGnRnE	Device-nGnRnE
	Device-nGnRE	Device-nGnRE
	Not (Device-nGnRnE or Device-nGnRE)	Device-nGRE
Device-GRE	Device-nGnRnE	Device-nGnRnE
	Device-nGnRE	Device-nGnRE
	Device-nGRE	Device-nGRE
	Device-GRE or Normal	Device-GRE
Normal	Any type of Device	Device type assigned at stage 2
	Normal	Normal

See *Combining the Shareability attribute* on page G5-5990 for information about the Shareability of:

- A region for which the resultant type is any Device type.
- A region with a resultant type of Normal for which the resultant cacheability, described in *Combining the Cacheability attribute*, is Inner Non-cacheable, Outer Non-cacheable.

The combining of the memory type attribute means a translation table walk for a stage 1 translation can be made to a type of Device memory. If this occurs, then:

- If the value of HCR.PTW is 0, then the translation table walk occurs as if it is to Normal Non-cacheable memory. This means it can be done speculatively.
- If the value of HCR.PTW is 1, then the memory access generates a stage 2 Permission fault.

Combining the Cacheability attribute

For a Normal memory region, Table G5-21 shows how the stage 1 and stage 2 Cacheability assignments are combined. This combination applies, independently, for the Inner Cacheability and Outer Cacheability attributes:

Table G5-21 Combining the stage 1 and stage 2 cacheability assignments

Assignment in stage 1	Assignment in stage 2	Resultant cacheability
Non-cacheable	Any	Non-cacheable
Any	Non-cacheable	Non-cacheable

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G5-21 Combining the stage 1 and stage 2 cacheability assignments (continued)

Assignment in stage 1	Assignment in stage 2	Resultant cacheability
Write-Through Cacheable	Write-Through or Write-Back Cacheable	Write-Through Cacheable
Write-Through or Write-Back Cacheable	Write-Through Cacheable	Write-Through Cacheable
Write-Back Cacheable	Write-Back Cacheable	Write-Back Cacheable

Note

Only Normal memory has a Cacheability attribute.

Combining the Shareability attribute

In the following cases, a memory region is treated as Outer Shareable, regardless of any shareability assignments at either stage of translation:

- The resultant memory type attribute, described in *Combining the memory type attribute* on page G5-5989, is any type of Device memory.
- The resultant memory type attribute is Normal memory, and the resultant Cacheability, described in *Combining the Cacheability attribute* on page G5-5989, is Inner Non-cacheable Outer Non-cacheable.

For a memory region with a resultant memory type attribute of Normal that is not Inner Non-cacheable Outer Non-cacheable, Table G5-22 shows how the stage 1 and stage 2 shareability assignments are combined:

Table G5-22 Combining the stage 1 and stage 2 Shareability assignments

Assignment in stage 1	Assignment in stage 2	Resultant Shareability
Outer Shareable	Any	Outer Shareable
Inner Shareable	Outer Shareable	Outer Shareable
Inner Shareable	Inner Shareable	Inner Shareable
Inner Shareable	Non-shareable	Inner Shareable
Non-shareable	Outer Shareable	Outer Shareable
Non-shareable	Inner Shareable	Inner Shareable
Non-shareable	Non-shareable	Non-shareable

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G5.8 Translation Lookaside Buffers (TLBs)

Translation Lookaside Buffers (TLBs) are an implementation technique that caches translations or translation table entries. TLBs avoid the requirement to perform a translation table walk in memory for every memory access. The Arm architecture does not specify the exact form of the TLB structures for any design. In a similar way to the requirements for caches, the architecture only defines certain principles for TLBs:

- The architecture has a concept of an entry locked down in the TLB. The method by which lockdown is achieved is IMPLEMENTATION DEFINED, and an implementation might not support lockdown.
- The architecture does not guarantee that an unlocked TLB entry remains in the TLB.
- The architecture guarantees that a locked TLB entry remains in the TLB. However, a locked TLB entry might be updated by subsequent updates to the translation tables. Therefore, when a change is made to the translation tables, the architecture does not guarantee that a locked TLB entry remains incoherent with an entry in the translation table.
- The architecture guarantees that a translation table entry that generates a Translation fault, an Address size fault, or an Access flag fault is not held in the TLB. However a translation table entry that generates a Domain fault or a Permission fault might be held in the TLB.
- When address translation is enabled, any translation table entry that does not generate a Translation fault, an Address size fault, or an Access flag fault and is not from a translation regime for an Exception level that is lower than the current Exception level can be allocated to a TLB at any time. The only translation table entries guaranteed not to be held in the TLB are those that generate a Translation fault, an Address size fault, or an Access flag fault.

  Note

  An TLB can hold translation table entries that do not generate a Translation fault but point to subsequent tables in the translation table walk. This can be referred to as *intermediate caching* of TLB entries.

- Software can rely on the fact that between disabling and re-enabling a stage of address translation, entries in the TLB relating to that stage of translation have not been corrupted to give incorrect translations.

The following sections give more information about TLB implementation:

- *Global and process-specific translation table entries*.
- *TLB matching* on page G5-5992.
- *TLB behavior at reset* on page G5-5992.
- *TLB lockdown* on page G5-5993.
- *TLB conflict aborts* on page G5-5993.

See also *TLB maintenance requirements* on page G5-5995.

Note

In addition to the functions described in this section, the TLB might cache information from control registers that are described as being "permitted to be cached in a TLB", even when any or all of the stages of translation are disabled. This caching of information gives rise to the maintenance requirements described in *General TLB maintenance requirements* on page G5-5995

G5.8.1 Global and process-specific translation table entries

For VMSAv8-32, system software can divide a virtual memory map used by memory accesses at PL1 and EL0 into global and non-global regions, indicated by the nG bit in the translation table descriptors:

nG == 0 The translation is global, meaning the region is available for all processes.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

nG == 1 The translation is non-global, or process-specific, meaning it relates to the current ASID, as defined by:

- TTBR0.ASID or TTBR1.ASID, if using the Long-descriptor translation table format. In this case, TTBCR.A1 selects which ASID is current.
- CONTEXTIDR.ASID, if using the Short-descriptor translation table format.

Each non-global region has an associated ASID. These identifiers mean different translation table mappings can co-exist in a caching structure such as a TLB. This means that software can create a new mapping of a non-global memory region without removing previous mappings.

For a symmetric multiprocessor cluster where a single operating system is running on the set of PEs, the architecture requires all ASID values to be assigned uniquely within any single Inner Shareable domain. In other words, each ASID value must have the same meaning to all PEs in the system.

In AArch32 state, the translation regime used for accesses made at EL2 never supports ASIDs, and all pages are treated as global.

When a PE is using the Long-descriptor translation table format, and is in Secure state, a translation must be treated as non-global, regardless of the value of the nG bit, if NSTable is set to 1 at any level of the translation table walk.

For more information see *Control of Secure or Non-secure memory access, VMSAv8-32 Long-descriptor format* on page G5-5956.

G5.8.2 TLB matching

A TLB is a hardware caching structure for translation table information. Like other hardware caching structures, it is mostly invisible to software. However, there are some situations where it can become visible. These are associated with coherency problems caused by an update to the translation table that has not been reflected in the TLB. Use of the TLB maintenance instructions described in *TLB maintenance requirements* on page G5-5995 can prevent any TLB incoherency becoming a problem.

A particular case where the presence of the TLB can become visible is if the translation table entries that are in use under a particular ASID and VMID are changed without suitable invalidation of the TLB. This can occur only if the architecturally-required break-before-make sequence described in *Using break-before-make when updating translation table entries* on page G5-5996 is not used. If the break-before make sequence is not used, the TLB can hold two mappings for the same address, and this:

- Might generate an exception that is reported using the TLB Conflict fault code, see *TLB conflict aborts* on page G5-5993.
- Might lead to CONSTRAINED UNPREDICTABLE behavior. In this case, behavior will be consistent with one of the mappings held in the TLB, or with some amalgamation of the values held in the TLB, but cannot give access to regions of memory with permissions or attributes that could not be assigned by valid translation table entries in the translation regime being used for the access. See *CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values* on page K1-7835.

G5.8.3 TLB behavior at reset

The Arm architecture does not require a reset to invalidate the TLBs, and recognizes that an implementation might require caches, including TLBs, to maintain context over a system reset. Possible reasons for doing so include power management and debug requirements.

Therefore, for Armv8:

- All TLBs reset to an IMPLEMENTATION DEFINED state that might be UNKNOWN.
- All TLBs are disabled from reset. All stages of address translation that are used from the PE state entered on coming out of reset are disabled from reset, and the contents of the TLBs have no effect on address translation. For more information see *Enabling stages of address translation* on page G5-5932.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- An implementation can require the use of a specific TLB invalidation routine, to invalidate the TLB arrays before they are enabled after a reset. The exact form of this routine is IMPLEMENTATION DEFINED, but if an invalidation routine is required it must be documented clearly as part of the documentation of the device.

  Arm recommends that if an invalidation routine is required for this purpose, and the PE resets into AArch32 state, the routine is based on the AArch32 TLB maintenance instructions described in *The scope of TLB maintenance instructions* on page G5-6004.

Similar rules apply:

- To cache behavior, see *Behavior of caches at reset* on page G4-5895.
- To branch predictor behavior, see *Behavior of the branch predictors at reset* on page G4-5903.

G5.8.4 TLB lockdown

The Arm architecture recognizes that any TLB lockdown scheme is heavily dependent on the microarchitecture, making it inappropriate to define a common mechanism across all implementations. This means that:

- The architecture does not require TLB lockdown support.
- If TLB lockdown support is implemented, the lockdown mechanism is IMPLEMENTATION DEFINED. However, key properties of the interaction of lockdown with the architecture must be documented as part of the implementation documentation.

This means that:

- The TLB Type Register, TLBTR, does not define the lockdown scheme in use.
- In AArch32 state, a region of the {`coproc`==`0b1111`, `CRn`==`c10`} encodings is reserved for IMPLEMENTATION DEFINED TLB functions, such as TLB lockdown functions. The reserved encodings are those with:
  - `<CRm>` == {c0, c1, c4, c8}.
  - All values of `<opc2>` and `<opc1>`.

An implementation might use some of the {`coproc`==`0b1111`, `CRn`==`c10`} encodings that are reserved for IMPLEMENTATION DEFINED TLB functions to implement additional TLB control functions. These functions might include:

- Unlock all locked TLB entries.
- Preload into a specific level of TLB. This is beyond the scope of the `PLI` and `PLD` hint instructions.

The inclusion of EL2 in an implementation does not affect the TLB lockdown requirements. However, in an implementation that includes EL2, exceptions generated as a result of TLB lockdown when executing in a Non-secure PL1 mode or in Non-secure User mode can be routed to either:

- Non-secure Abort mode, using the Non-secure Data Abort exception vector.
- Hyp mode, using the Hyp Trap exception vector.

For more information, see *Traps to Hyp mode of Non-secure EL0 and EL1 accesses to lockdown, DMA, and TCM operations* on page G1-5793.

G5.8.5 TLB conflict aborts

If an address matches multiple entries in the TLB, it is IMPLEMENTATION DEFINED whether a TLB conflict abort is generated.

An implementation can generate TLB conflict aborts on either or both instruction fetches and data accesses. A TLB conflict abort is classified as an MMU fault, see *Types of MMU faults* on page G5-6013. This means:

- A TLB conflict abort on an instruction fetch is reported as a Prefetch Abort exception,
- A TLB conflict abort on a data access is reported as a Data Abort exception,

Fault status codes for TLB conflict aborts are defined for both the Short-descriptor and Long-descriptor translation table formats, see:

- *PL1 fault reporting with the Short-descriptor translation table format* on page G5-6029
- *PL1 fault reporting with the Long-descriptor translation table format* on page G5-6031.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

On a TLB conflict abort, the fault address register returns the address that generated the fault. That is, it returns the address that was being looked up in the TLB.

It is IMPLEMENTATION DEFINED whether a TLB conflict abort is a stage 1 abort or a stage 2 abort.

Note

- An address can hit multiple entries in the TLB if the TLB has been invalidated inappropriately, for example if TLB invalidation required by this manual has not been performed.
- A stage 2 abort cannot be generated if the Non-secure PL1&0 stage 2 address translation is disabled.

The priority of the TLB conflict abort is IMPLEMENTATION DEFINED, because it depends on the form of any TLB that can generate the abort. However, the TLB conflict abort must have higher priority than any abort that depends on a value held in the TLB.

If an address matches multiple entries in the TLB and no TLB conflict abort not generated, the resulting behavior is CONSTRAINED UNPREDICTABLE, see *CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values* on page K1-7835. The CONSTRAINED UNPREDICTABLE behavior must not permit access to regions of memory with permissions or attributes that mean they cannot be accessed in the current Security state at the current Privilege level.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G5.9 TLB maintenance requirements

Translation Lookaside Buffers (TLBs) on page G5-5991 describes the Arm architectural provision for TLBs. Although the Arm architecture does not specify the form of any TLB structures, it does define the mechanisms by which TLBs can be maintained. The following sections describe the VMSAv8-32 TLB maintenance instructions:

- *General TLB maintenance requirements*.
- *Maintenance requirements on changing System register values* on page G5-6000.
- *Atomicity of register changes on changing virtual machine* on page G5-6001.
- *Synchronization of changes of ASID and TTBR* on page G5-6002.
- *The scope of TLB maintenance instructions* on page G5-6004.

G5.9.1 General TLB maintenance requirements

TLB maintenance instructions provide a mechanism to invalidate entries from a TLB. As *Translation Lookaside Buffers (TLBs)* on page G5-5991 describes, when address translation is enabled translation table entries can be allocated to a TLB at any time. This means that software must perform TLB maintenance between updating translation table entries that apply in a particular context and accessing memory locations whose translation is determined by those entries in that context.

Note

This requirement applies to any translation table entry at any level of the translation tables, including an entry that points to further levels of the tables, provided that the entry in that level of the tables does not cause a Translation fault, an Address size fault, or an Access flag fault.

In addition to any TLB maintenance requirement, when changing the cacheability attributes of an area of memory, software must ensure that any cached copies of affected locations are removed from the caches. For more information see *Cache maintenance requirement created by changing translation table attributes* on page G5-6011.

Because a TLB never holds any translation table entry that generates a Translation fault, an Address size fault, or an Access flag fault, a change from a translation table entry that causes a Translation, Address size, or Access flag fault to one that does not fault, does not require any TLB or branch predictor invalidation. However, a *Context synchronization event* is required to ensure that instruction fetches are affected by a completed change to translation table entries that, before the change, generated a Translation, Address size, or Access flag fault.

Special considerations apply to translation table updates that change the memory type, cacheability, or output address of an entry, see *Using break-before-make when updating translation table entries* on page G5-5996.

In addition, software must perform TLB maintenance after updating the System registers if the update means that the TLB might hold information that applies to a current translation context, but is no longer valid for that context. *Maintenance requirements on changing System register values* on page G5-6000 gives more information about this maintenance requirement.

Each of the translation regimes defined in Figure G5-1 on page G5-5924 is a different context, and:

- For the Non-secure PL1&0 regime, a change in the VMID or ASID value changes the context.
- For the Secure PL1&0 regime, a change in the ASID value changes the context.

For operation in Non-secure PL1 or EL0 modes, a change of HCR.VM, unless made at the same time as a change of VMID, requires the invalidation of all TLB entries for the Non-secure PL1&0 translation regime that apply to the current VMID. Otherwise, there is no guarantee that the effect of the change of HCR.VM is visible to software executing in the Non-secure PL1 and EL0 modes.

Any TLB maintenance instruction can affect any other TLB entries that are not locked down.

AArch32 state defines {`coproc==0b1111`, `CRn==c8`} System instructions for TLB maintenance instructions, and supports the following operations:

- Invalidate all unlocked entries in the TLB.
- Invalidate a single TLB entry, by VA, or VA and ASID for a non-global entry.
- Invalidate all TLB entries that match a specified ASID.
- Invalidate all TLB entries that match a specified VA, regardless of the ASID.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Operations that apply across multiprocessors in the same Inner Shareable domain.

  Note

  An address-based TLB maintenance instruction that applies to the Inner Shareable domain does so regardless of the Shareability attributes of the address supplied as an argument to the instruction.

A TLB maintenance instruction that specifies a VA that would generate any MMU fault, including a VA that is not in the range of VAs that can be translated, does not generate an abort.

EL2 provides additional TLB maintenance instructions for use in AArch32 state at EL2, and has some implications for the effect of the other TLB maintenance instructions, see *The scope of TLB maintenance instructions* on page G5-6004.

In an implementation that includes EL3, the TLB maintenance instructions take account of the current Security state, as part of the address translation required for the TLB maintenance instruction.

Some TLB maintenance instructions are defined as operating only on instruction TLBs, or only on data TLBs. Armv8 AArch32 state includes these instructions for backwards compatibility. However, more recent TLB maintenance instructions do not support this distinction. From the introduction of Armv7, Arm deprecates any use of Instruction TLB maintenance instructions, or of Data TLB maintenance instructions, and developers must not rely on this distinction being maintained in future revisions of the Arm architecture.

The Arm architecture does not dictate the form in which the TLB stores translation table entries. However, for TLB invalidate instructions, the minimum size of the table entry that is invalidated from the TLB must be at least the size that appears in the translation table entry.

The scope of TLB maintenance instructions on page G5-6004 describes the TLB maintenance instructions. The following subsections give more information about the general requirements for TLB maintenance:

- *Using break-before-make when updating translation table entries*.
- *The interaction of TLB lockdown with TLB maintenance instructions* on page G5-5997.
- *Ordering and completion of TLB maintenance instructions* on page G5-5998.
- *Use of ASIDs and VMIDs to reduce TLB maintenance requirements* on page G5-5999.

Using break-before-make when updating translation table entries

To avoid possibly creating multiple TLB entries for the same address, and to avoid the effects of TLB caching possibly breaking coherency, single-copy atomicity properties, ordering guarantees or uniprocessor semantics, or possibly failing to clear the Exclusives monitors, the architecture requires the use of a break-before-make sequence when changing translation table entries whenever multiple threads of execution can use the same translation tables and the change to the translation table entries involves any of:

- A change of the memory type.
- A change of the cacheability attributes.
- A change of the output address (OA), if the OA of at least one of the old translation table entries and the new translation table entry is writable.
- A change to the size of block used by the translation system. This applies both:
  - When changing from a smaller size to a larger size, for example by replacing a table mapping with a block mapping in a stage 2 translation table.
  - When changing from a larger size to a smaller size, for example by replacing a block mapping with a table mapping in a stage 2 translation table.
- A change of the output address (OA), if the contents of memory at the new OA do not match the contents of memory at the previous OA.
- Creating a global entry when there might be non-global entries in a TLB that overlap with that global entry.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A break-before-make sequence on changing from an old translation table entry to a new translation table entry requires the following steps:

1. Replace the old translation table entry with an invalid entry, and execute a `DSB` instruction.
2. Invalidate the translation table entry with a broadcast TLB invalidation instruction, and execute a `DSB` instruction to ensure the completion of that invalidation.
3. Write the new translation table entry, and execute a `DSB` instruction to ensure that the new entry is visible.

This sequence ensures that at no time are both the old and new entries simultaneously visible to different threads of execution, and therefore the problems described at the start of this subsection cannot arise.

The interaction of TLB lockdown with TLB maintenance instructions

The precise interaction of TLB lockdown with the TLB maintenance instructions is IMPLEMENTATION DEFINED. However, the architecturally-defined TLB maintenance instructions must comply with these rules:

- The effect on locked entry of a TLB invalidate all unlocked entries instruction or a TLB invalidate by VA all ASID instruction that would invalidate that entry if the entry was not locked must be one of the following, and it is IMPLEMENTATION DEFINED which behavior applies:
  - — The instructions have no effect on entries that are locked down.
  - — The instructions generate an IMPLEMENTATION DEFINED Data Abort exception if an entry is locked down, or might be locked down. For an invalidate instruction performed in AArch32 state, the `{coproc==0b1111, CRn==c5}` fault status register definitions include a Fault status code for cache and TLB lockdown faults, see Table G5-26 on page G5-6029 for the codes used with the Short-descriptor translation table formats, or Table G5-27 on page G5-6031 for the codes used with the Long-descriptor translation table formats.

    In an implementation that includes EL2, if EL2 is using AArch32 and the value of HCR.TIDCP is 1, any such exceptions taken from a Non-secure PL1 mode are routed to Hyp mode, see *Traps to Hyp mode of Non-secure EL0 and EL1 accesses to lockdown, DMA, and TCM operations* on page G1-5793.

  This permits a usage model for TLB invalidate routines, where the routine invalidates a large range of addresses, without considering whether any entries are locked in the TLB.

- The effect on a locked TLB entry of a TLB invalidate by VA instruction or a TLB invalidate by ASID match instruction that would invalidate that entry if the entry was not locked must be one of the following, and it is IMPLEMENTATION DEFINED which behavior applies:
  - — A locked entry is invalidated in the TLB.
  - — The instruction has no effect on a locked entry in the TLB. In the case of the Invalidate single entry by VA, this means the PE treats the instruction as a NOP.
  - — The instruction generates an IMPLEMENTATION DEFINED Data Abort exception if it operates on an entry that is locked down, or might be locked down. For an invalidate instruction performed in AArch32 state, the `{coproc==0b1111, CRn==c5}` fault status register definitions include a Fault status code for cache and TLB lockdown faults, see Table G5-26 on page G5-6029 and Table G5-27 on page G5-6031.

Note

Any implementation that uses an abort mechanism for entries that can be locked down but are not actually locked down must:

- Document the IMPLEMENTATION DEFINED instruction sequences that perform the required invalidation on entries that are not locked down.
- Implement one of the other specified alternatives for the locked entries.

Arm recommends that, when possible, such IMPLEMENTATION DEFINED instruction sequences use the architecturally-defined maintenance instructions. This minimizes the number of customized maintenance operations required.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In addition, an implementation that uses an abort mechanism for handling TLB maintenance instructions on entries that can be locked down but are not actually locked down must also must provide a mechanism that ensures that no TLB entries are locked.

Similar rules apply to cache lockdown, see *The interaction of cache lockdown with cache maintenance instructions* on page G4-5911.

The architecture does not guarantee that any unlocked entry in the TLB remains in the TLB. This means that, as a side-effect of a TLB maintenance instruction, any unlocked entry in the TLB might be invalidated.

Ordering and completion of TLB maintenance instructions

The following rules describe the relations between the memory order model and the TLB maintenance instructions:

- A TLB invalidate instruction is complete for a PE when all memory accesses generated by that PE using the TLB entries that have been invalidated are complete.

  A TLB invalidate instruction is complete when it is complete for all PEs.

  After the TLB invalidate instruction is complete, no new memory accesses using the invalidated TLB entries will be architecturally performed by any observer that is affected by the TLB invalidation.

  Note

  This requirement does not mean that speculative memory accesses cannot be performed using those entries if it is impossible for software running on any observer to tell that those memory accesses have been performed.

- A TLB maintenance instruction is only guaranteed to be complete after the execution of a `DSB` instruction.
- An `ISB` instruction, or a return from an exception, causes the effect of all completed TLB maintenance instructions that appear in program order before the `ISB` or return from exception to be visible to all subsequent instructions, including the instruction fetches for those instructions.
- An exception causes all completed TLB maintenance instructions, that appear in the instruction stream before the point where the exception is taken, to be visible to all subsequent instructions, including the instruction fetches for those instructions.
- All TLB maintenance instructions are executed in program order relative to each other.
- The execution of a Data or Unified TLB maintenance instruction is only guaranteed to be visible to a subsequent explicit load or store instruction after both:
  - The execution of a `DSB` instruction to ensure the completion of the TLB maintenance instruction.
  - Execution of a subsequent *Context synchronization event*.
- The execution of an Instruction or Unified TLB maintenance instruction is only guaranteed to be visible to a subsequent instruction fetch after both:
  - The execution of a `DSB` instruction to ensure the completion of the TLB maintenance instruction.
  - Execution of a subsequent *Context synchronization event*.

In all cases in this section where a `DMB` or `DSB` is referred to, it refers to a `DMB` or `DSB` whose required access type is both loads and stores. A `DSB NSH` is sufficient to ensure completion of TLB maintenance instructions that apply to a single PE. A `DSB ISH` is sufficient to ensure completion of TLB maintenance instructions that apply to PEs in the same Inner Shareable domain.

The following rules apply when writing translation table entries. They ensure that the updated entries are visible to subsequent accesses and cache maintenance instructions.

For TLB maintenance, the translation table walk is treated as a separate observer. This means:

- A write to the translation tables is only guaranteed to be seen by a translation table walk caused by an explicit load or store after the execution of both a `DSB` and an `ISB`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

However, the architecture guarantees that any writes to the translation tables are not seen by any explicit memory access that occurs in program order before the write to the translation tables.

- A write to the translation tables is only guaranteed to be seen by a translation table walk caused by the instruction fetch of an instruction that follows the write to the translation tables after both a `DSB` and an `ISB`.

Therefore, in a uniprocessor system, an example instruction sequence for writing a translation table entry, covering changes to the instruction or data mappings is:

```
STR rx, [Translation table entry]          ; write new entry to the translation table
DSB            ; ensures visibility of the new entry
Invalidate TLB entry by VA (and ASID if non-global) [page address]
Invalidate BTC
DSB            ; ensure completion of the Invalidate TLB instruction
ISB            ; ensure table changes visible to instruction fetch
```

Use of ASIDs and VMIDs to reduce TLB maintenance requirements

To reduce the need for TLB maintenance on context switches, the lookups from some translation regimes can be associated with an ASID, or with an ASID and a VMID.

Note

The use of ASIDs and VMIDs in VMSAv8-32 is generally similar to their use in VMSAv8-64, see *Use of ASIDs and VMIDs to reduce TLB maintenance requirements* on page D5-2662.

For more information about the use of ASIDs in VMSAv8-32 see *Global and process-specific translation table entries* on page G5-5991.

Common not private translations in VMSAv8-32

In an implementation that includes ARMv8.2-TTCNP, multiple PEs in the same Inner Shareable domain can use the same translation table entries for a given stage of address translation in a particular translation regime. This sharing is enabled by the TTBR.CnP field for the stage of address translation.

When the value of a TTBR.CnP field is 1, translation table entries pointed to by that TTBR are shared with all other PEs in the Inner Shareable domain for which the following conditions are met:

- The corresponding TTBR.CnP field has the value 1.
- That TTBR is using the Long-descriptor translation table format.
- If an ASID applies to the stage of translation corresponding to that TTBR then the current ASID value must be the same for all of the PEs that are sharing entries.
- If a VMID applies to the stage of translation corresponding to that TTBR then the current VMID value must be the same for all of the PEs that are sharing entries.

Note

In an implementation that includes EL3, the Secure instances of TTBR0 and TTBR1 relate to the Secure PL1&0 translation regime, and the Non-secure instances of TTBR0 and TTBR1 relate to the Non-secure PL1&0 translation regime.

For a translation regime with both stage 1 and stage 2 translations, where a TLB combines information from stage 1 and stage 2 translation table entries into a single entry, this entry can be shared between different PEs only if the value of the TTBR.CnP bit is 1 for both stage 1 and stage 2 of the translation table walk.

The TTBR.CnP bit can be cached in a TLB.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For a given TTBR, if the value of TTBR.CnP is 1 on multiple PEs in the same Inner Shareable domain, and those PEs meet the other conditions for sharing translation table entries as defined in this section, but those TTBRs do not point to the same translation table entries, then the system is misconfigured, and performing an address translation using that TTBR:

- Might generate multiple hits in the TLB, and as a result generate an exception that is reported using the TLB conflict fault code, see *TLB conflict aborts* on page G5-5993.
- Otherwise, has a CONSTRAINED UNPREDICTABLE result, as described in *CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values* on page K1-7835.

G5.9.2 Maintenance requirements on changing System register values

The TLB contents can be influenced by control bits in a number of System registers. This means the TLB entries associated with a translation regime affected by these control bits must be invalidated after any changes to these bits, unless the changes are accompanied by a change to the VMID or ASID, if appropriate depending on the translation regime, that defines the context to which the bits apply. The general form of the required invalidation sequence is as follows:

```
; Change control bits in System registers
ISB        ; Synchronize changes to the control bits
; Perform TLB invalidation of all entries that might be affected by the changed control bits
```

The System register changes that this applies to are:

- Any change to the NMRR, PRRR, MAIR0,MAIR1, HMAIR0 or HMAIR1 registers.
- Any change to the SCTLR.AFE bit, see *Changing the Access flag enable*.
- Any change to any of the SCTLR.{TRE, WXN, UWXN} bits.
- Any change to the Translation table base 0 address in TTBR0.
- Any change to the Translation table base 1 address in TTBR1.
- Any change to HTTBR.BADDR.
- Any change to VTTBR.BADDR.
- Changing TTBCR.EAE, see *Changing the current Translation table format* on page G5-6001.
- In an implementation that includes EL3, any change to the SCR.SIF bit.
- In an implementation that includes EL2:
  - Any change to the HCR.VM bit.
  - Any change to HCR.PTW bit, see *Changing HCR.PTW* on page G5-6001.
- When using the Short-descriptor translation table format:
  - Any change to the RGN, IRGN, S, or NOS fields in TTBR0 or TTBR1.
  - Any change to the N, EAE, PD0 or PD1 fields in TTBCR
- When using the Long-descriptor translation table format:
  - Any change to the EAE, T*n*SZ, ORGN*n*, IRGN*n*, SH*n*, or EPD*n* fields in the TTBCR, where *n* is 0 or 1.
  - Any change to the TTBCR2.
  - Any change to the T0SZ, ORGN0, IRGN0, or SH0 fields in the HTCR.
  - Any change to the T0SZ, ORGN0, IRGN0, or SH0 fields in the VTCR.

Changing the Access flag enable

In a PE that is using the Short-descriptor translation table format, it is CONSTRAINED UNPREDICTABLE whether the TLB caches the effect of the SCTLR.AFE bit on translation tables. This means that, after changing the SCTLR.AFE bit software must invalidate the TLB before it relies on the effect of the new value of the SCTLR.AFE bit, otherwise behavior is CONSTRAINED UNPREDICTABLE, see *CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values* on page K1-7835.

Note

There is no enable bit for use of the Access flag when using the Long-descriptor translation table format.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Changing HCR.PTW

When EL2 is using AArch32 and the value of the Protected table walk bit, HCR.PTW, is 1, a stage 1 translation table access in the Non-secure PL1&0 translation regime, to an address that is mapped to any type of Device memory by its stage 2 translation, generates a stage 2 Permission fault. A TLB associated with a particular VMID might hold entries that depend on the effect of HCR.PTW. Therefore, if the value of HCR.PTW is changed without a change to the VMID value, all TLB entries associated with the current VMID must be invalidated before executing software in a Non-secure PL1 or EL0 mode. If this is not done, behavior is CONSTRAINED UNPREDICTABLE, see *CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values* on page K1-7835.

Changing the current Translation table format

The effect of changing TTBCR.EAE when executing in the translation regime affected by TTBCR.EAE with any stage of address translation for that translation regime enabled is CONSTRAINED UNPREDICTABLE. This means that, when TTBCR.EAE is changed for a given context, the TLB must be invalidated before resuming execution in that context, otherwise the effect is CONSTRAINED UNPREDICTABLE, see *CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values* on page K1-7835.

G5.9.3 Atomicity of register changes on changing virtual machine

From the viewpoint of software executing in a Non-secure PL1 or EL0 mode, when there is a switch from one virtual machine to another, the registers that control or affect address translation must be changed atomically. This applies to the registers for:

- Non-secure PL1&0 stage 1 address translations. This means that all of the following registers must change atomically:
  - PRRR and NMRR, if using the Short-descriptor translation table format.
  - MAIR0 and MAIR1, if using the Long-descriptor translation table format.
  - TTBR0, TTBR1, TTBCR, TTBCR2, DACR, and CONTEXTIDR.
  - The SCTLR.
- Non-secure PL1&0 stage 2 address translations. When EL2 is using AArch32, this means that all of the following registers and register fields must change atomically:
  - VTTBR and VTCR.
  - HMAIR0 and HMAIR1.
  - The HSCTLR.

Note

Only some bits of SCTLR affect the stage 1 translation, and only some bits of HSCTLR affect the stage 2 translation. However, in each case, changing these bits requires a write to the register, and that write must be atomic with the other register updates.

These registers apply to execution in Non-secure PL1&0 modes. However, when updated as part of a switch of virtual machines they are updated by software executing in Hyp mode. This means the registers are *out of context* when they are updated, and no synchronization precautions are required.

Note

By contrast, a translation table change associated with a change of ASID, made by software executing at PL1, can require changes to registers that are *in context*. *Synchronization of changes of ASID and TTBR* on page G5-6002 describes appropriate precautions for such a change.

Software executing in Hyp mode, or in Secure state, must not use the registers associated with the Non-secure PL1&0 translation regime for speculative memory accesses.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G5.9.4 Synchronization of changes of ASID and TTBR

A common virtual memory management requirement is to change the ASID and TTBR together to associate the new ASID with different translation tables, without any change to the current translation regime. When using the Short-descriptor translation table format, different registers hold the ASID and the translation table base address, meaning these two values cannot be updated atomically. Since a PE can perform a speculative memory access at any time, this lack of atomicity is a problem that software must address. Such a change is complicated by:

- The depth of speculative fetch being IMPLEMENTATION DEFINED.
- The use of branch prediction.

When using the Short-descriptor translation table format, the virtual memory management operations must ensure the synchronization of changes of the ContextID and the translation table registers. For example, some or all of the TLBs, branch predictors, and other caching of ASID and translation information might become corrupt with invalid translations. Synchronization is required to avoid either:

- The old ASID being associated with translation table walks from the new translation tables.
- The new ASID being associated with translation table walks from the old translation tables.

There are a number of possible solutions to this problem, and the most appropriate approach depends on the system. Example G5-3, Example G5-4, and Example G5-5 on page G5-6003 describe three possible approaches.

Note

Another instance of the synchronization problem occurs if a branch is encountered between changing the ASID and performing the synchronization. In this case the value in the branch predictor might be associated with the incorrect ASID. Software can address this possibility using any of these approaches, but instead software might be written in a way that avoids such branches.

Example G5-3 Using a reserved ASID to synchronize ASID and TTBR changes

In this approach, a particular ASID value is reserved for use by the operating system, and is used only for the synchronization of the ASID and TTBR. This example uses the value of 0 for this purpose, but any value could be used.

This approach can be used only when the size of the mapping for any given VA is the same in the old and new translation tables.

The maintenance software uses the following sequence, that must be executed from memory marked as global:

```
Change ASID to 0
ISB
Change TTBR
ISB
Change ASID to new value
```

This approach ensures that any non-global pages fetched at a time when it is uncertain whether the old or new translation tables are being accessed are associated with the unused ASID value of 0. Since the ASID value of 0 is not used for any normal operations these entries cannot cause corruption of execution.

Example G5-4 Using translation tables containing only global mappings when changing the ASID

A second approach involves switching the translation tables to a set of translation tables that only contain global mappings while switching the ASID.

The maintenance software uses the following sequence, that must be executed from memory marked as global:

```
Change TTBR to the global-only mappings
ISB
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
Change ASID to new value
ISB
Change TTBR to new value
```

This approach ensures that no non-global pages can be fetched at a time when it is uncertain whether the old or new ASID value will be used.

This approach works without the need for TLB invalidations in systems that have caching of intermediate levels of translation tables, as described in *General TLB maintenance requirements* on page G5-5995, provided that the translation tables containing only global mappings have only level 1 translation table entries of the following kinds:

- Entries that are global.
- Pointers to level 2 tables that hold only global entries, and that are the same level 2 tables that are used for accessing global entries by both:
  - The set of translation tables that were used under the old ASID value.
  - The set of translation tables that will be used with the new ASID value.
- Invalid level 1 entries.

In addition, all sets of translation tables in this example should have the same Shareability and Cacheability attributes, as held in the TTBR0.{ORGN, IRGN} or TTBR1.{ORGN, IRGN} fields.

If these rules are not followed, then the implementation might cache level 1 translation table entries that require explicit invalidation.

Example G5-5 Disabling non-global mappings when changing the ASID

In systems where only the translation tables indexed by TTBR0 hold non-global mappings, maintenance software can use the TTBCR.PD0 field to disable use of TTBR0 during the change of ASID. This means the system does not require a set of global-only mappings.

The maintenance software uses the following sequence, that must be executed from a memory region with a translation that is accessed using the base address in the TTBR1 register, and is marked as global:

```
Set TTBCR.PD0 = 1
ISB
Change ASID to new value
Change TTBR to new value
ISB
Set TTBCR.PD0 = 0
```

This approach ensures that no non-global pages can be fetched at a time when it is uncertain whether the old or new ASID value will be used.

When using the Long-descriptor translation table format, TTBCR.A1 holds the number, 0 or 1, of the TTBR that holds the current ASID. This means the current TTBR can also hold the current ASID, and the current translation table base address and ASID can be updated atomically when:

- TTBR0 is the only TTBR being used. TTBCR.A1 must be set to 0.
- TTBR0 points to the only translation tables that hold non-global entries, and TTBCR.A1 is set to 0.
- TTBR1 points to the only translation tables that hold non-global entries, and TTBCR.A1 is set to 1.

In these cases, software can update the current translation table base address and ASID atomically, by updating the appropriate TTBR, and does not require a specific routine to ensure synchronization of the change of ASID and base address.

However, in all other cases using the Long-descriptor format, the synchronization requirements are identical to those when using the Short-descriptor formats, and the examples in this section indicate how synchronization might be achieved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

When using the Long-descriptor translation table format, CONTEXTIDR.ASID has no significance for address translation, and is only an extension of the Context ID value.

G5.9.5 The scope of TLB maintenance instructions

TLB maintenance instructions provide a mechanism for invalidating entries from TLB caching structures, to ensure that changes to the translation tables are reflected correctly in the TLB caching structures. To support TLB maintenance in multiprocessor systems, there are maintenance operations that apply to the TLBs of all PEs in the same Inner Shareable domain.

The architecture permits the caching of any translation table entry that has been returned from memory without a fault and that does not, itself, cause a Translation Fault, an Address size fault, or an Access Flag fault. This means the TLB:

- Cannot hold an entry that, when used for a translation table lookup, causes a Translation fault, an Address size fault, or an Access Flag fault.
- Can hold an entry for a translation table lookup for a translation that causes a Translation Fault, an Address size fault, or an Access Flag fault at a subsequent level of translation table lookup. For example, it can hold an entry for the level 1 lookup of a translation that causes a Translation fault, an Address size fault, or an Access Flag fault at level 2 or level 3 of lookup.

This means that entries cached in the TLB can include:

- Translation table entries that point to a subsequent table to be used in the current stage of translation.
- In an implementation that includes EL2:
  - Stage 2 translation table entries that are used as part of a stage 1 translation table walk.
  - Stage 2 translation table entries for translating the output address of a stage 1 translation.

Such entries might be held in intermediate TLB caching structures that are distinct from the data caches, in that they are not required to be invalidated as the result of writes of the data. The architecture makes no restriction on the form of these intermediate TLB caching structures.

The architecture does not intend to restrict the form of TLB caching structures used for holding translation table entries. In particular for translation regimes that involve two stages of translation, it recognizes that such caching structures might contain:

- At any level of the translation table walk, entries containing information from stage 1 translation table entries.
- In an implementation that includes EL2:
  - At any level of the translation table walk, entries containing information from stage 2 translation table entries.
  - At any level of the translation table walk, entries combining information from both stage 1 and stage 2 translation table entries.

Note

For the purpose of TLB maintenance, the term *TLB entry* denotes any structure, including temporary working registers in translation table walk hardware, that holds a translation table entry.

For the TLB maintenance instructions:

- If a TLB maintenance instruction is required to apply to stage 1 entries then it must apply to any cached entry in the caching structures that includes any stage 1 information that would be used to translate the address being invalidated, including any entry that combines information from both stage 1 and stage 2 translation table entries.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

— Where stage 1 information has been cached in multiple TLB entries, as could occur from splintering a page when caching in the TLB, then the invalidation must apply to each cached entry containing stage 1 information from the page that is used to translate the address being invalidated, regardless of whether or not that cached entry would be used to translate the address being invalidated.

— As stated in *Global and process-specific translation table entries* on page G5-5991, translation table entries from levels of translation other than the final level are treated as being non-global. Arm expects that, in at least some implementations, cached copies of levels of the translation table walk other than the last level are tagged with their ASID, regardless of whether the final level is global. This means that TLB invalidations that involve the ASID require the ASID to match such entries to perform the required invalidation.

- If a TLB maintenance instruction is required to apply to stage 2 entries only, then:
  - — It is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.
  - — It must apply to caching structures that contain information only from stage 2 translation table entries.
- If a TLB maintenance instruction is required to apply to both stage 1 and stage 2 entries, then it must apply to any entry in the caching structures that includes information from either a stage 1 translation table entry or a stage 2 translation table entry, including any entry that combines information from both stage 1 and stage 2 translation table entries.

Table G5-23 on page G5-6006 summarizes the required effect of the AArch32 TLB maintenance instructions, that operate only on TLBs on the PE that executes the instruction. Additional TLB maintenance instructions that:

- Apply across all PEs in the same Inner Shareable domain. Each instruction shown in the table has an Inner Shareable equivalent, identified by an `IS` suffix. For example, the Inner Shareable equivalent of TLBIALL is TLBIALLIS. See also *EL2 forced broadcasting of TLB maintenance instructions* on page G5-6007.
- Can apply to separate Instruction or Data TLBs. These instructions are indicated by a footnote to the table. Arm deprecates any use of these instructions.

Note

- The architecture permits a TLB invalidation instruction to affect any unlocked entry in the TLB. Table G5-23 on page G5-6006 defines only the entries that each instruction must invalidate.
- All TLB instructions, including those that operate on a VA match, operate as described regardless of the value of SCTLR.M.

When interpreting the table:

Related operations Each instruction description applies also to any equivalent instruction that either:
- Applies to all PEs in the same Inner Shareable domain.
- Applies only to a data TLB, or only to an instruction TLB.

So, for example, the TLBIALL instruction description applies also to TLBIALLIS, ITLBIALL, and DTLBIALL.

TLB maintenance system instructions on page K14-8091 lists all of the TLB maintenance instructions.

Matches the VA Means the VA argument for the instruction must match the VA value in the TLB entry.

Matches the ASID Means the ASID argument for the instruction must match the ASID in use when the TLB entry was assigned.

Matches the current VMID

Means the current VMID must match the VMID in use when the TLB entry was assigned.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The dependency on the VMID applies even when the value of HCR.VM is 0, including situations where there is no use of virtualization. However, VTTBR.VMID resets to zero, meaning there is a valid VMID from reset.

Execution at EL2 Descriptions of operations at EL2 apply only to implementations that include EL2.

For the definitions of the translation regimes referred to in the table see *About VMSAv8-32* on page G5-5922.

Table G5-23 Effect of the TLB maintenance instructions

Instruction	Executed from		Effect, must invalidate any entry that matches all stated conditions[a]
	State	Mode	
TLBIALL[b]	Secure	PL1	All entries for the Secure PL1&0 translation regime. That is, all entries that were allocated in Secure state.
	Non-secure	PL1	All entries for stage 1 of the Non-secure PL1&0 translation regime that match the current VMID.
		Hyp	All entries for stage 1 or stage 2 of the Non-secure PL1&0 translation regime that match the current VMID.
TLBIMVA[b]	Secure	PL1	Any entry for the Secure PL1&0 translation regime that both: • Matches the VA argument. • Matches the ASID argument, or is global.
	Non-secure	PL1 or Hyp	Any entry for stage 1 of the Non-secure PL1&0 translation regime to which all of the following apply. The entry: • Matches the VA argument. • Matches the ASID argument, or is global. • Matches the current VMID.
TLBIASID[b]	Secure	PL1	Any entry for the Secure PL1&0 translation regime that matches the specified ASID and either: • Is from a level of lookup above the final level. • Is a non-global entry from the final level of lookup.
	Non-secure	PL1 or Hyp	Any entry for stage 1 of the Non-secure PL1&0 translation regime that both: • Matches the specified ASID and either: — Is from a level of lookup above the final level. — Is a non-global entry from the final level of lookup. • Matches the current VMID.
TLBIMVAA	Secure	PL1	Any entry for the Secure PL1&0 translation regime that matches the VA argument.
	Non-secure	PL1 or Hyp	Any entry for stage 1 of the Non-secure PL1&0 translation regime that both: • Matches the VA argument. • Matches the current VMID.
TLBIALLNSNH[c]	Secure	Monitor	All entries for stage 1 or stage 2 of the Non-secure PL1&0 translation regime, regardless of the associated VMID.
	Non-secure	Hyp	
TLBIALLH[c]	Secure	Monitor	All entries for the Non-secure EL2 translation regime. That is, any entry that was allocated in Non-secure state from Hyp mode.
	Non-secure	Hyp	

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G5-23 Effect of the TLB maintenance instructions (continued)

Instruction	Executed from State	Executed from Mode	Effect, must invalidate any entry that matches all stated conditions[a]
TLBIMVAL	Secure	PL1	Any entry for stage 1 of the Secure PL1&0 translation regime that is from the last level of the translation table walk and both: • Matches the VA argument. • Matches the ASID argument, or is global.
	Non-secure	PL1 or Hyp	Any entry for stage 1 of the Non-secure PL1&0 translation regime that is from the last level of the translation table walk and to which all of the following apply. The entry: • Matches the VA argument. • Matches the ASID argument, or is global. • Matches the current VMID.
TLBIMVAAL	Secure	PL1	Any entry for stage 1 of the Secure PL1&0 translation regime that is from the last level of the translation table walk and matches the VA argument.
	Non-secure	PL1 or Hyp	Any entry for stage 1 of the Non-secure PL1&0 translation regime that is from the last level of the translation table walk and both: • Matches the VA argument. • Matches the current VMID.
TLBIMVAH[c]	Secure	Monitor	Any entry for the Non-secure EL2 translation regime that matches the VA argument.
	Non-secure	Hyp	
TLBIMVALH[c]	Secure	Monitor	Any entry for the Non-secure EL2 translation regime that is from the last level of the translation table walk and matches the VA argument.
	Non-secure	Hyp	
TLBIIPAS2[c, d]	Secure	Monitor[e]	Any entry for stage 2 of the PL1&0 translation regime that both: • Matches the IPA argument. • Matches the current VMID.
	Non-secure	Hyp	
TLBIIPAS2L[c, d]	Secure	Monitor[e]	Any entry for stage 2 of the PL1&0 translation regime that is from the last level of translation and both: • Matches the IPA argument. • Matches the current VMID.
	Non-secure	Hyp	

a. When a TLB maintenance instruction is executed at Secure EL1 in AArch32 state when EL3 is using AArch64, it only affects TLB entries related to the Secure EL1 translation regime.

b. The architecture defines variants of these instructions that apply only to instruction TLBs, and only to data TLBs. Arm deprecates any use of these variants. For more information, see the referenced description of the operation.

c. Available only in an implementation that includes EL2. See also *EL2 forced broadcasting of TLB maintenance instructions*.

d. This instruction is CONSTRAINED UNPREDICTABLE if executed in any AArch32 Secure privileged mode, see *Hyp mode TLB maintenance instructions* on page K1-7852.

e. This instruction executes as a NOP when SCR.NS == 0.

EL2 forced broadcasting of TLB maintenance instructions

In an implementation that includes EL2, when the value of HCR.FB is 1, the TLB maintenance instructions that are not broadcast across the Inner Shareable domain are forced to operate across the Inner Shareable domain when executed in a Non-secure PL1 mode. For example, when the value of HCR.FB is 1, a TLBIMVA instruction executed in a Non-secure PL1 mode performs the same invalidation as the invalidation performed by a TLBIMVAIS instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TLB maintenance with different translation granule sizes

If a TLB maintenance instruction specifying a VA affecting the EL2 translation regime is broadcast from a PE using AArch32 to a PE using AArch64 using a translation granule size that is different from the AArch32 translation granule size for that same translation regime, the TLB maintenance instruction is not required to perform any invalidation on the recipient PE.

If a TLB maintenance instruction specifying a VA affecting the PL1 translation regime is broadcast from a PE using AArch32 using one translation granule size for that translation regime for a particular ASID, VMID (if applicable), and Security state, to a PE using AArch64 where EL1 for the same ASID, VMID (if applicable), and Security state, is using a translation granule size that is different from the AArch32 translation granule size, the TLB maintenance instruction is not required to perform any invalidation on the recipient PE.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G5.10 Caches in VMSAv8-32

The Arm architecture describes the required behavior of an implementation of the architecture. As far as possible it does not restrict the implemented microarchitecture, or the implementation techniques that might achieve the required behavior.

Maintaining this level of abstraction is difficult when describing the relationship between memory address translation and caches, especially regarding the indexing and tagging policy of caches. This section:

- Summarizes the architectural requirements for the interaction between caches and memory translation.
- Gives some information about the likely implementation impact of the required behavior.

The following sections give this information:

- *Data and unified caches*.
- *Instruction caches*.

In addition *Cache maintenance requirement created by changing translation table attributes* on page G5-6011 describes the cache maintenance required after updating the translation tables to change the attributes of an area of memory.

For more information about cache maintenance see:

- *AArch32 cache and branch predictor support* on page G4-5889. This section describes the Arm cache maintenance instructions.
- *Cache maintenance system instructions* on page K14-8090. This section summarizes the System register encodings used for these operations when executing in AArch32 state.

G5.10.1 Data and unified caches

For data and unified caches, the use of memory address translation is entirely transparent to any data access other than as described in *Mismatched memory attributes* on page E2-4020.

This means that the behavior of accesses from the same observer to different VAs, that are translated to the same PA with the same memory attributes, is fully coherent. This means these accesses behave as follows, regardless of which VA is accessed:

- Two writes to the same PA occur in program order.
- A read of a PA returns the value of the last successful write to that PA.
- A write to a PA that occurs, in program order, after a read of that PA, has no effect on the value returned by that read.

The memory system behaves in this way without any requirement to use barrier or cache maintenance instructions.

In addition, if cache maintenance is performed on a memory location, the effect of that cache maintenance is visible to all aliases of that physical memory location.

These properties are consistent with implementing all caches that can handle data accesses as *Physically-indexed, physically-tagged* (PIPT) caches.

G5.10.2 Instruction caches

In the Arm architecture, an instruction cache is a cache that is accessed only as a result of an instruction fetch. Therefore, an instruction cache is never written to by any load or store instruction executed by the PE.

The Arm architecture permits different behaviors for instruction caches. These are identified by descriptions of the associated expected implementation. The following subsections describe the behavior associated with these cache types, including any occasions where explicit cache maintenance is required to make the use of memory address translation transparent to the instruction cache:

- *PIPT (Physically-indexed, physically-tagged) instruction caches* on page G5-6010.
- *VPIPT (VMID-aware PIPT) instruction caches* on page G5-6010.
- *VIPT (Virtually-indexed, physically-tagged) instruction caches* on page G5-6010.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *The IVIPT architecture Extension* on page G5-6011.

In AArch32 state, the CTR.L1Ip field identifies the form of the instruction caches.

Note

For software to be portable between implementations that might use any of PIPT instruction caches, VPIPT instruction caches, or VIPT instruction caches, software must invalidate the instruction cache whenever any condition occurs that would require instruction cache maintenance for at least one of the instruction cache types.

PIPT (*Physically-indexed, physically-tagged*) instruction caches

For a PIPT instruction cache:

- The use of memory address translation is entirely transparent to all instruction fetches other than as described in *Mismatched memory attributes* on page E2-4020.
- If cache maintenance is performed on a memory location, the effect of that cache maintenance is visible to all aliases of that physical memory location.

An implementation that provides PIPT instruction caches implements the IVIPT Extension, see *The IVIPT architecture Extension* on page G5-6011.

VPIPT (*VMID-aware PIPT*) instruction caches

An Armv8.2 implementation can implement VPIPT instruction caches. If it does so then it is described as implementing ARMv8.2-VPIPT.

The CTR.L1Ip field identifies the implemented cache type, meaning it identifies whether ARMv8.2-VPIPT is implemented.

For a VPIPT instruction cache:

- Instruction fetches from Non-secure EL1 and Non-secure EL0 are only permitted to hit in the cache if the instruction fetch is made using the VMID that was used when the entry in the instruction cache was fetched.
- An instruction cache maintenance instruction executed at Non-secure EL0 or at Non-secure EL1 is required to have an effect on entries in the instruction cache only if those entries were fetched using the VMID that is current when the cache maintenance instruction is executed.

All other requirements for the use of cache maintenance instructions are the same as for *PIPT (Physically-indexed, physically-tagged) instruction caches*.

An implementation that provides VPIPT instruction caches implements the IVIPT Extension, see *The IVIPT architecture Extension* on page G5-6011.

VIPT (*Virtually-indexed, physically-tagged*) instruction caches

For a VIPT instruction cache:

- The use of memory address translation is transparent to all instruction fetches other than for the effect of memory address translation on instruction cache invalidate by address operations or as described in *Mismatched memory attributes* on page E2-4020.

  Note

  Cache invalidation is the only cache maintenance instruction that can be performed on an instruction cache.

- If instruction cache invalidation by address is performed on a memory location, the effect of that invalidation is visible only to the VA supplied with the operation. The effect of the invalidation might not be visible to any other VA aliases of that physical memory location.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The only architecturally-guaranteed way to invalidate all aliases of a PA from a VIPT instruction cache is to invalidate the entire instruction cache.

An implementation that provides VIPT instruction caches implements the IVIPT Extension, see *The IVIPT architecture Extension*.

The IVIPT architecture Extension

In Armv8, any permitted instruction cache implementation can be described as implementing the *IVIPT Extension* to the Arm architecture.

The formal definition of the Arm IVIPT Extension is that it reduces the instruction cache maintenance requirement to the following condition:

- Instruction cache maintenance is required only after writing new data to a PA that holds an instruction.

Note

Previous versions of the Arm architecture have permitted an instruction cache option that does not implement the Arm IVIPT Extension.

G5.10.3 Cache maintenance requirement created by changing translation table attributes

Any change to the translation tables to change the attributes of an area of memory can require maintenance of the translation tables, as described in *General TLB maintenance requirements* on page G5-5995. If the change affects the cacheability attributes of the area of memory, including any change between Write-Through and Write-Back attributes, software must ensure that any cached copies of affected locations are removed from the caches, typically by cleaning and invalidating the locations from the levels of cache that might hold copies of the locations affected by the attribute change. Any of the following changes to the inner cacheability or outer cacheability attribute creates this maintenance requirement:

- Write-Back to Write-Through.
- Write-Back to Non-cacheable.
- Write-Through to Non-cacheable.
- Write-Through to Write-Back.

The cache clean and invalidate avoids any possible coherency errors caused by mismatched memory attributes.

Similarly, to avoid possible coherency errors caused by mismatched memory attributes, the following sequence must be followed when changing the Shareability attributes of a cacheable memory location:

1. Make the memory location Non-cacheable, Outer Shareable.
2. Clean and invalidate the location from them cache.
3. Change the Shareability attributes to the required new values.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G5.11 VMSAv8-32 memory aborts

In a VMSAv8-32 implementation, the following mechanisms cause a PE to take an exception on a failed memory access:

Debug exception An exception caused by the debug configuration, see Chapter G2 *AArch32 Self-hosted Debug*.

Alignment fault An Alignment fault is generated if the address used for a memory access does not have the required alignment for the operation. For more information see *Unaligned data access* on page E2-4004 and *Alignment faults* on page G5-6020.

MMU fault An MMU fault is a fault generated by the fault checking sequence for the current translation regime. See *Types of MMU faults* on page G5-6013.

External abort Any memory system fault other than a Debug exception, an Alignment fault, or an MMU fault.

Collectively, these mechanisms are called *aborts*. Chapter G2 *AArch32 Self-hosted Debug* and Chapter H3 *Halting Debug Events* describe Debug exceptions, and the remainder of this section describes Alignment faults, MMU faults, and External aborts.

An access that causes an abort is said to be aborted, and uses the *Fault Address Registers* (FARs) and *Fault Status Registers* (FSRs) or *Exception Syndrome Registers* (ESRs) to record context information.

The exception generated on a synchronous memory abort:

- On an instruction fetch is called the Prefetch Abort exception.
- On a data access is called the Data Abort exception.

Note

The Prefetch Abort exception applies to any synchronous memory abort on an instruction fetch. It is not restricted to speculative instruction fetches.

The Exception level and PE mode that a VMSAv8-32 memory abort is taken to depends on the translation regime and stage that generate the abort. The fault context is dependent on whether:

- The abort is reported as a Prefetch Abort or as a Data Abort.
- The exception is taken from the same or a lower Exception level.

Note

A memory access from AArch32 state may be subject to one or more VMSAv8-64 translation stages. For example, a Non-secure EL0 access when EL1 is using AArch64 is subject to both stages of the VMSAv8-64 Non-secure EL1&0 translation regime. A memory abort generated on a VMSAv8-64 translation stage is handled as described in *VMSAv8-64 memory aborts* on page D5-2652.

For more information, see *Routing of aborts taken to AArch32 state* on page G1-5722.

External aborts can be reported synchronously or asynchronously. Asynchronous External aborts are reported using the SError interrupt. For more information, see *External aborts* on page G4-5914.

In AArch32 state, asynchronous memory aborts are a type of External abort, and are treated as a type of Data Abort exception.

The following sections describe the abort mechanisms:

- *Types of MMU faults* on page G5-6013.
- *VMSAv8-32 MMU fault terminology* on page G5-6015.
- *The MMU fault-checking sequence* on page G5-6016.
- *Alignment faults* on page G5-6020.
- *External abort on a translation table walk* on page G5-6020.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *AArch32 state prioritization of synchronous aborts from a single stage of address translation* on page G5-6021.

An access that causes an abort is said to be aborted. On an abort, System registers are used to record context information. For more information see *Exception reporting in a VMSAv8-32 implementation* on page G5-6024.

G5.11.1 Types of MMU faults

This section describes the faults that might be detected during one of the fault-checking sequences described in *The MMU fault-checking sequence* on page G5-6016. Unless indicated otherwise, information in this section applies to the fault checking sequences for both the Short-descriptor translation table format and the Long-descriptor translation table format.

MMU faults are always synchronous.

When an MMU fault generates an abort for a region of memory, no memory access is made if that region is or could be marked as any type of Device memory.

The MMU faults that might be detected during a fault checking sequence are:

- Permission fault.
- Translation fault.
- Address size fault.
- Access flag fault.
- Domain fault, short-descriptor translation tables only.
- TLB conflict abort.

See also *External abort on a translation table walk* on page G5-6020.

Note

- Although the TLB conflict abort is classified as an MMU fault, it is described in the section *Translation Lookaside Buffers (TLBs)* on page G5-5991.
- In VMSAv8-64 an External abort on a translation table walk is classified as an MMU fault. However, in VMSAv8-32, for consistency with earlier versions of the architecture these aborts are not classified as MMU faults.

Permission fault

A Permission fault can be generated at any level of lookup, and the reported fault code identifies the lookup level. See *About access permissions* on page G5-5968 for information about conditions that cause a Permission fault.

Note

When using the Short-descriptor translation table format, the translation table descriptors are checked for Permission faults only for accesses to memory regions in Client domains.

A TLB might hold a translation table entry that cause a Permission fault. Therefore, if the handling of a Permission fault results in an update to the associated translation tables, the software that updates the translation tables must invalidate the appropriate TLB entry, to prevent the stale information in the TLB being used on a subsequent memory access. For more information, see the translation table entry update examples in *Ordering and completion of TLB maintenance instructions* on page G5-5998.

In an implementation that includes EL2, this maintenance requirement applies to Permission faults in both stage 1 and stage 2 translations.

Cache or branch predictor maintenance operations cannot cause a Permission fault, except that a stage 1 translation table walk performed as part of a cache or branch predictor maintenance operation can generate a stage 2 Permission fault as described in Stage 2 fault on a stage 1 translation table walk.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Translation fault

A Translation fault can be generated at any level of lookup, and the reported fault code identifies the lookup level. A Translation fault is generated if bits[1:0] of a translation table descriptor identify the descriptor as either a Fault encoding or a reserved encoding. For more information see:

- *VMSAv8-32 Short-descriptor translation table format descriptors* on page G5-5940.
- *VMSAv8-32 Long-descriptor translation table format descriptors* on page G5-5949.

In addition, a Translation fault is generated if the input address for a translation either does not map onto an address range of a TTBR, or the TTBR range that it maps onto is disabled. In these cases the fault is reported as a level 1 Translation fault on the translation stage at which the mapping to a region described by a TTBR failed.

The architecture guarantees that any translation table entry that causes a Translation fault is not cached, meaning the TLB never holds such an entry. Therefore, when a Translation fault occurs, the fault handler does not have to perform any TLB maintenance instructions to remove the faulting entry.

A data or unified cache maintenance by VA instruction can generate a Translation fault. However:

- If the Point of Coherency is before any level of cache, it is IMPLEMENTATION DEFINED whether a data or unified cache maintenance by VA to the Point of Coherency instruction can generate a Translation fault.
- If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether a data or unified cache clean by VA to the Point of Unification instruction can generate a Translation fault.

It is IMPLEMENTATION DEFINED whether an instruction cache invalidate by VA operation can generate a Translation fault.

It is IMPLEMENTATION DEFINED whether a branch predictor maintenance operation can generate a Translation fault.

Address size fault

An Address size fault can be generated at any level of lookup, and the reported fault code identifies the lookup level.

An Address size fault is generated if the translation table entries or the TTBR for the stage of translation have nonzero address bits above the most significant bit of the maximum output address size. Because VMSAv8-32 supports a maximum PA and IPA size of 40 bits, this means any case where a translation table entry or the TTBR holds an address for which A[47:40] is nonzero generates an Address size fault.

A data or unified cache maintenance by VA instruction can generate an Address size fault. However:

- If the Point of Coherency is before any level of cache, it is IMPLEMENTATION DEFINED whether a data or unified cache maintenance by VA instruction can generate an Address size fault.
- If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether a data or unified cache clean by VA to the Point of Unification instruction can generate an Address size fault.

It is IMPLEMENTATION DEFINED whether an instruction cache invalidate by VA operation can generate an Address size fault.

It is IMPLEMENTATION DEFINED whether a branch predictor maintenance operation can generate an Address size fault.

The architecture guarantees that any translation table entry that causes an Address size fault is not cached, meaning the TLB never holds such an entry. Therefore, when an Address size fault occurs, the fault handler does not have to perform any TLB maintenance instructions to remove the faulting entry.

Access flag fault

An Access flag fault can be generated at any level of lookup, and the reported fault code identifies the lookup level. An Access flag fault is generated only if all of the following apply:

- The translation tables support an Access flag bit:
  - The Short-descriptor format supports an Access flag only when SCTLR.AFE is set to 1.
  - The Long-descriptor format always supports an Access flag.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- A translation table descriptor with the Access flag bit set to 0 is loaded.

For more information about the Access flag bit see:

- *VMSAv8-32 Short-descriptor translation table format descriptors* on page G5-5940
- *VMSAv8-32 Long-descriptor translation table format descriptors* on page G5-5949.

The architecture guarantees that any translation table entry that causes an Access flag fault is not cached, meaning the TLB never holds such an entry. Therefore, when an Access flag fault occurs, the fault handler does not have to perform any TLB maintenance instructions to remove the faulting entry.

Whether any cache maintenance instruction by VA can generate Access flag faults is IMPLEMENTATION DEFINED.

Whether branch predictor invalidate by VA operations can generate Access flag faults is IMPLEMENTATION DEFINED.

For more information, see *The Access flag* on page G5-5976.

Domain fault, Short-descriptor format translation tables only

When using the Short-descriptor translation table format, a Domain fault can be generated at level 1or level 2 of lookup. The reported fault code identifies the lookup level. The conditions for generating a Domain fault are:

Level 1 When a level 1 descriptor fetch returns a valid Section level 1 descriptor, the domain field of that descriptor is checked against the DACR. A level 1 Domain fault is generated if this check fails.

Level 2 When a level 2 descriptor fetch returns a valid level 2 descriptor, the domain field of the level 1 descriptor that required the level 2 fetch is checked against the DACR, and a level 2 Domain fault is generated if this check fails.

For more information, see *Domains, Short-descriptor format only* on page G5-5976.

Domain faults cannot occur on cache or branch predictor maintenance operations.

A TLB might hold a translation table entry that cause a Domain fault. Therefore, if the handling of a Domain fault results in an update to the associated translation tables, the software that updates the translation tables must invalidate the appropriate TLB entry, to prevent the stale information in the TLB being used on a subsequent memory access. For more information, see the translation table entry update examples in *Ordering and completion of TLB maintenance instructions* on page G5-5998.

Any change to the DACR must be synchronized by a *Context synchronization event*. For more information see *Synchronization of changes to AArch32 System registers* on page G8-6100.

G5.11.2 VMSAv8-32 MMU fault terminology

The Armv7 Large Physical Address Extension introduced new terminology for faults on a stage of address translation, to provide consistent terminology across all implementations. Table G5-24 shows the terminology used in this manual for an MMU faults, compared with older Arm documentation. The current terms are the same for faults that occur with the Short-descriptor translation table format and with the Long-descriptor format, and also apply to faults in a level 3 lookup when using the Long-descriptor translation table format.

Table G5-24 MMU fault terminology

Current term	Old term	Note
Level 1 Translation fault	Section Translation fault	-
Level 2 Translation fault	Page Translation fault	-
Level 3 Translation fault	-	Long-descriptor translation table format only.
Level 1 Access flag fault	Section Access flag fault	-
Level 2 Access flag fault	Page Access flag fault	-
Level 3 Access flag fault	-	Long-descriptor translation table format only.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G5-24 MMU fault terminology (continued)

Current term	Old term	Note
Level 1 Domain fault	Section Domain fault	Short-descriptor translation table format only, except for reporting faults on address translation instructions in the 64-bit PAR, see *Determining the PAR format* on page G5-6047. Cannot occur at level 3.
Level 2 Domain fault	Page Domain fault	
Level 1 Permission fault	Section Permission fault	-
Level 2 Permission fault	Page Permission fault	-
Level 3 Permission fault	-	Long-descriptor translation table format only.

In an implementation that includes EL2, MMU faults are also classified by the translation stage at which the fault is generated. This means that a memory access from a Non-secure PL1 or EL0 mode can generate:

- A stage 1 MMU fault, for example, a stage 1 Translation fault.
- A stage 2 MMU fault, for example, a stage 2 Translation fault.

G5.11.3 The MMU fault-checking sequence

This section describes the MMU checks made for the memory accesses required for instruction fetches and for explicit memory accesses:

- If an instruction fetch faults it generates a Prefetch Abort exception.
- If an data memory access faults it generates a Data Abort exception.

For more information about Prefetch Abort exceptions and Data Abort exceptions see *Handling exceptions that are taken to an Exception level using AArch32* on page G1-5703.

In VMSAv8-32, all memory accesses require VA to PA translation. Therefore, when a corresponding stage of address translation is enabled, each access requires a lookup of the translation table descriptor for the accessed VA. For more information, see *Translation tables* on page G5-5934 and subsequent sections of this chapter. MMU fault checking is performed for each level of translation table lookup. If an implementation includes EL2 and is operating in Non-secure state, MMU fault checking is performed for each stage of address translation.

Note

In an implementation that includes EL2, if a PE is executing in Non-secure state, the operating system or similar Non-secure system software defines the stage 1 translation tables in the IPA address map, and typically is unaware of the stage 2 translation from IPA to PA. However, each Non-secure stage 1 translation table access is subject to stage 2 address translation, and might be faulted at that stage.

The MMU fault checking sequence is largely independent of the translation table format, as the figures in this section show. The differences are:

When using the Short-descriptor format

- There are one or two levels of lookup.
- Lookup always starts at level 1.
- The final level of lookup checks the Domain field of the descriptor and:
  - Faults if there is no access to the Domain.
  - Checks the access permissions only for Client domains.

When using the Long-descriptor format

- There are one, two, or three levels of lookup.
- Lookup starts at either level 1 or level 2.
- Domains are not supported. All accesses are treated as Client domain accesses.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The fault-checking sequence shows a translation from an Input address to an Output address. For more information about this terminology, see *About address translation for VMSAv8-32* on page G5-5925.

Note

The descriptions in this section do not include the possibility that the attempted address translation generates a TLB conflict abort, as described in *TLB conflict aborts* on page G5-5993.

Types of MMU faults on page G5-6013 describes the faults that an MMU fault-checking sequence can report.

Figure G5-15 shows the process of fetching a descriptor from the translation table. For the top-level fetch for any translation, the descriptor is fetched only if the input address passes any required alignment check. As the figure shows, in an implementation that includes EL2, if the translation is stage 1 of the Non-secure PL1&0 translation regime, then the descriptor address is in the IPA address map, and is subject to a stage 2 translation to obtain the required PA. This stage 2 translation requires a recursive entry to the fault checking sequence.

Note

Figure G5-15 and Figure G5-16 on page G5-6018 give an overview of the fault checking performed by the MMU. See *AArch32 state prioritization of synchronous aborts from a single stage of address translation* on page G5-6021 for the complete set of possible faults and their prioritization.

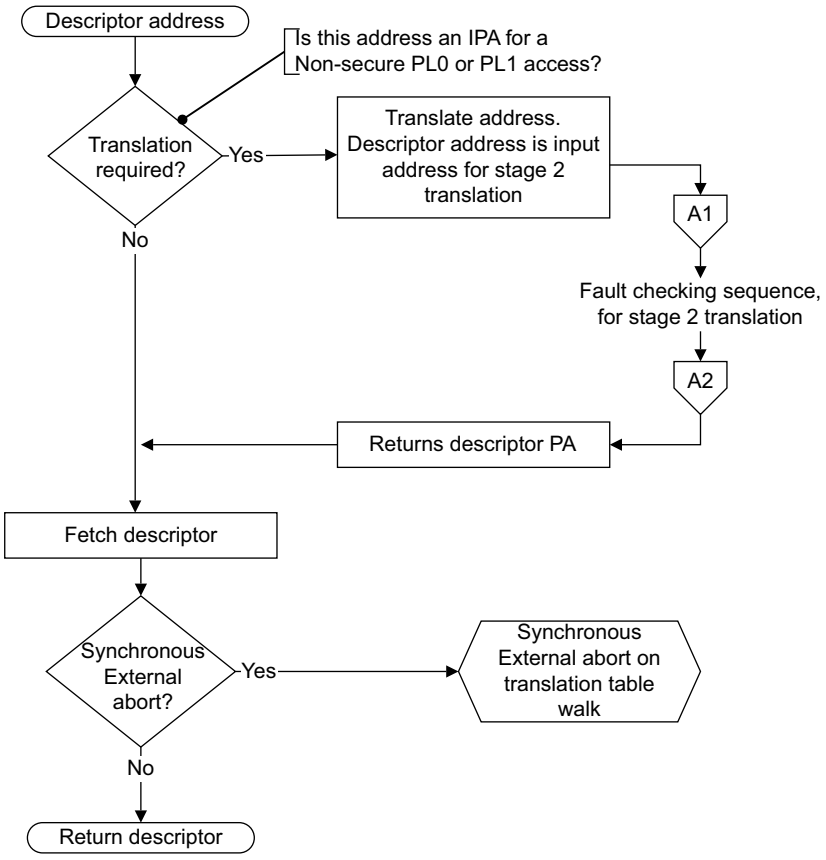
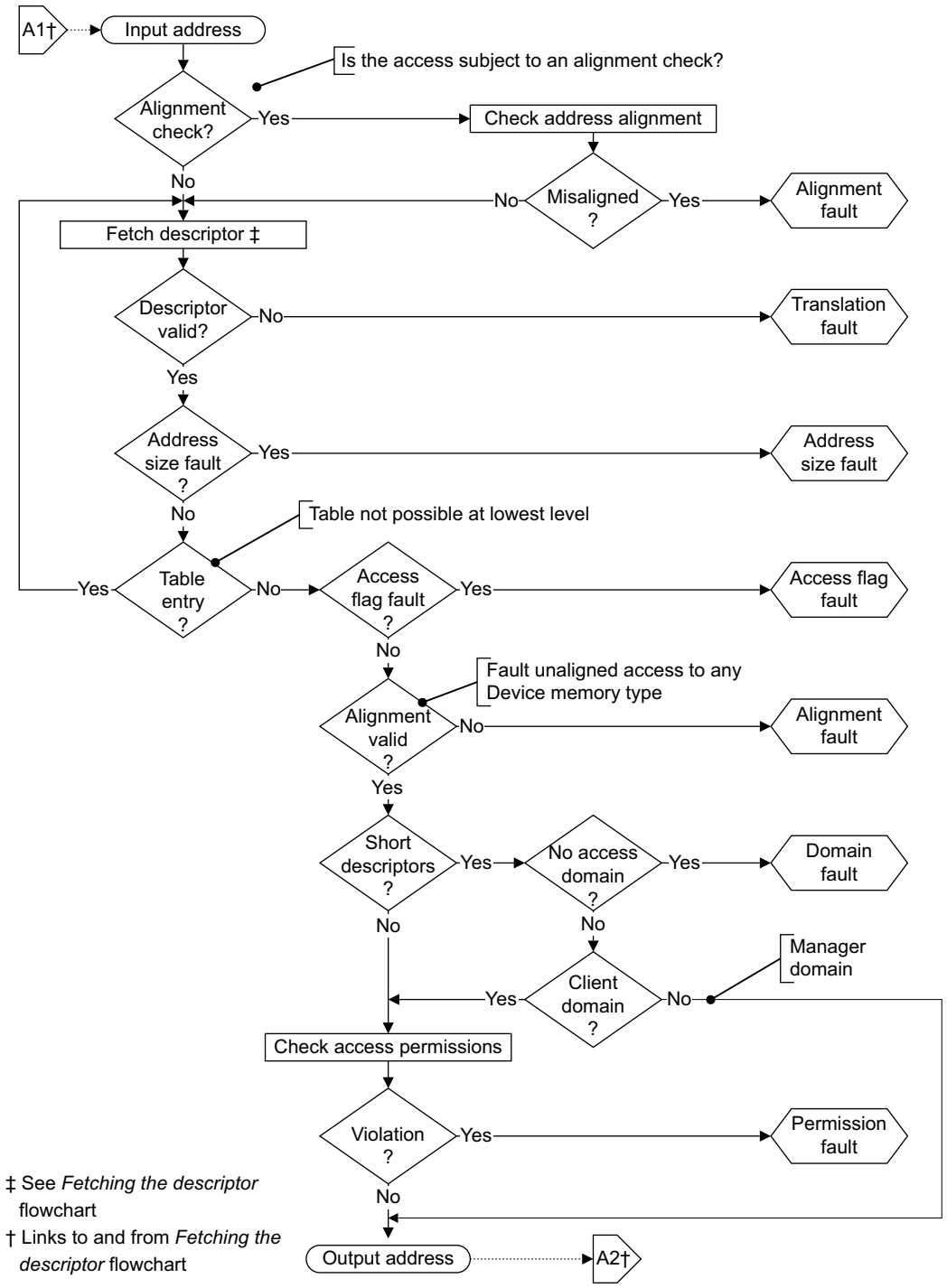


Figure G5-15 Fetching the descriptor in a VMSAv8-32 translation table walk

Figure G5-16 on page G5-6018 shows the full VMSAv8-32 fault checking sequence, including the alignment check on the initial access.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta



‡ See *Fetching the descriptor* flowchart

† Links to and from *Fetching the descriptor* flowchart

Figure G5-16 VMSAv8-32 fault checking sequence

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Stage 2 fault on a stage 1 translation table walk

When an implementation that includes EL2 is operating in a Non-secure PL1 or EL0 mode, any memory access goes through two stages of translation:

- Stage 1, from VA to IPA.
- Stage 2, from IPA to PA.

Note

In a virtualized system that is using AArch32, typically, a Guest OS operating in a Non-secure PL1 mode defines the translation tables and translation table register entries controlling the Non-secure PL1&0 stage 1 translations. A Guest OS has no awareness of the stage 2 address translation, and therefore believes it is specifying translation table addresses in the PA map. However, it actually specifies these addresses in its IPA map. Therefore, to support virtualization, translation table addresses for the Non-secure PL1&0 stage 1 translations are always defined in the IPA address map.

On performing a translation table walk for the stage 1 translations, the descriptor addresses must be translated from IPA to PA, using a stage 2 translation. This means that a memory access made as part of a stage 1 translation table lookup might generate, on a stage 2 translation:

- A Translation fault, Access flag fault, or Permission fault.
- A synchronous External abort on the memory access.

If SCR.EA is set to 1, a synchronous External abort is taken to EL3, and if EL3 is using AArch32 it is taken to Secure Monitor mode. Otherwise, these faults are reported as stage 2 memory aborts. When EL2 is using AArch32, HSR.ISS[7] is set to 1, to indicate a stage 2 fault during a stage 1 translation table walk, and the part of the ISS field that might contain details of the instruction is invalid. For more information see *Use of the HSR* on page G5-6038.

Alternatively, a memory access made as part of a stage 1 translation table lookup might target an area of memory with the Device memory attribute assigned on the stage 2 translation of the address accessed. When the value of the HCR.PTW bit is 1, such an access generates a stage 2 Permission fault.

Note

- On most systems, such a mapping to a Device memory type on the stage 2 translation is likely to indicate a Guest OS error, where the stage 1 translation table is corrupted. Therefore, it is appropriate to trap this access to the hypervisor.

A TLB might hold entries that depend on the effect of HCR.PTW. Therefore, if HCR.PTW is changed without changing the current VMID, the TLBs must be invalidated before executing in a Non-secure PL1 or EL0 mode. For more information see *Changing HCR.PTW* on page G5-6001.

A cache maintenance instruction executed at Non-secure PL1 can cause a stage 1 translation table walk that might generate a stage 2 Permission fault, as described in this section. However:

- If the Point of Coherency is before any level of cache, it is IMPLEMENTATION DEFINED whether a cache maintenance by VA instruction can generate a Permission fault in this way.
- If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether a data or unified cache clean by VA to the Point of Unification instruction can generate a Permission fault in this way.

Note

This is an exception to the general rule that a cache maintenance instruction cannot generate a Permission fault.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The level associated with MMU faults

When an MMU fault is from a stage of translation that is using Long-descriptor translation table format, Table G5-25 shows how the LL bits in the STATUS field of DFSR, IFSR, and HSR encode the lookup level associated with the fault.

Table G5-25 Use of LL bits to encode the lookup level at which the fault occurred

LL bits	Meaning
00	Level 0 of translation or translation table base register.
01	Level 1.
10	Level 2.
11	Level 3. When xFSR.STATUS indicates a Domain fault, this value is reserved.

The lookup level associated with a fault is:

- For a fault generated on a translation table walk, the lookup level of the walk being performed.
- For a Translation fault, the lookup level of the translation table that gave the fault. If a fault occurs because a stage of address translation is disabled, or because the input address is outside the range specified by the appropriate base address register or registers, the fault is reported as a level 1 fault.
- For an Access flag fault, the lookup level of the translation table that gave the fault.
- For a Permission fault, including a Permission fault caused by hierarchical permissions, the lookup level of the final level of translation table accessed for the translation. That is, the lookup level of the translation table that returned a Block or Page descriptor.

Also see *Synchronous External abort errors from address translation caching structures* on page G5-6023.

G5.11.4 Alignment faults

The Arm memory architecture requires support for strict alignment checking. This checking is controlled by:

- SCTLR.A, for accesses made from any PE mode other than Hyp mode.
- HSCTLR.A, for accesses made from Hyp mode.

In addition, some instructions do not support unaligned accesses, regardless of the value of SCTLR.A or HSCTLR.A.

Unaligned data access on page E2-4004:

- Defines when Alignment faults are generated, for both values of SCTLR.A or HSCTLR.A.
- Describes the possible generation of Alignment faults on accesses to Device memory by AArch32 Load Multiple or Store Multiple instructions when ARMv8.2-LSMAOC is implemented.

An Alignment fault can occur on an access for which the stage of address translation is disabled.

Any unaligned access to memory region with any Device memory type attribute generates an Alignment fault.

Routing of aborts taken to AArch32 state on page G1-5722 defines the mode to which an Alignment fault is taken.

The prioritization of Alignment faults depends on whether the fault was generated because of an access to a Device memory type, or for another reason. For more information see *AArch32 state prioritization of synchronous aborts from a single stage of address translation* on page G5-6021.

G5.11.5 External abort on a translation table walk

An External abort on a translation table walk can be either synchronous or asynchronous. For more information on External aborts, see *External aborts* on page G4-5914.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

An External abort on a translation table walk is reported:

- If the External abort is synchronous, using:
  - A synchronous Prefetch Abort exception if the translation table walk is for an instruction fetch.
  - A synchronous Data Abort exception if the translation table walk is for a data access.
- If the External abort is asynchronous, using an SError interrupt, which is taken as an asynchronous Data Abort exception.

If an implementation reports the error in the translation table walk asynchronously from executing the instruction whose instruction fetch or memory access caused the translation table walk, these aborts behave essentially as interrupts. The aborts are masked when PSTATE.A is set to 1, otherwise they are reported using the Data Abort exception.

Behavior of External aborts on a translation table walk caused by address translation instructions

The address translation instructions summarized in *Address translation system instructions* on page K14-8090 require translation table walks. An External abort can occur in the translation table walk. The abort generates a Data Abort exception, and can be synchronous or asynchronous. For more information, see *Handling of faults and aborts during an address translation instruction* on page G5-6047.

G5.11.6 AArch32 state prioritization of synchronous aborts from a single stage of address translation

Exception prioritization for exceptions taken to AArch32 state on page G1-5706 describes the prioritization of exceptions taken from an Exception level that is using AArch32. This section gives additional information about the prioritization of MMU faults from VMSAv8-32 translation regimes.

If a single instruction generates aborts on more than one memory access, the architecture does not define any prioritization between those aborts.

In general, the Arm architecture does not define when asynchronous events are taken, and therefore the prioritization of asynchronous events is IMPLEMENTATION DEFINED.

Note

The priority numbering in this list only shows the relative priorities of aborts from a single stage of address translation in a VMSAv8-32 translation regime. This numbering has no global significance and, for example, does not correlate with the equivalent AArch64 list in *AArch64 state prioritization of synchronous aborts from a single stage of address translation* on page D5-2659.

For a single stage of translation in a VMSAv8-32 translation regime, the following numbered list shows the priority of the possible memory management faults on a memory access. In this list:

- For memory accesses that undergo two stages of translation, the *italic entries show where the faults from the stage 2 translation can occur*. A stage 2 fault within a stage 1 translation table walk follows the same prioritization of faults.
- For synchronous External aborts from translation table walks see also *Synchronous External abort errors from address translation caching structures* on page G5-6023.

The priority order, from highest priority to lowest priority, is:

1. Alignment fault not caused by memory type. This is possible for a stage 1 translation only.
2. Translation fault due to the input address being out of the address range to be translated or requiring an AArch32 TTBR that is disabled. This includes VTCR.SL0 being inconsistent with VTCR.T0SZ or programmed to a reserved value.
3. Address size fault on an AArch32 TTBR caused by the PA being out of the range implemented.
4. *Second stage abort on a level 1 lookup of a a stage 1 table walk. When stage 2 address translation is enabled this includes an Address size fault caused by the PA being out of the range implemented*. This is second stage abort during a first stage translation table walk.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

5. Synchronous parity or ECC error on a level 1 lookup of a translation table walk.
6. Synchronous External abort on a level 1 lookup level of a translation table walk.
7. Translation fault on a level 1 translation table entry.
8. Address size fault on a level 1 lookup translation table entry caused by the output address being out of the range implemented.
9. *Second stage abort on a level 2 lookup of a a stage 1 table walk. When stage 2 address translation is enabled this includes an Address size fault caused by the PA being out of the range implemented*. This is second stage abort during a first stage translation table walk.
10. Synchronous parity or ECC error on a level 2 lookup of a translation table walk.
11. Synchronous External abort on a level 2 lookup level of a translation table walk.
12. Translation fault on a level 2 translation table entry.
13. Address size fault on a level 2 lookup translation table entry caused by the output address being out of the range implemented.
14. *Second stage abort on a level 3 lookup of a a stage 1 table walk. When stage 2 address translation is enabled this includes an Address size fault caused by the PA being out of the range implemented*. This is second stage abort during a first stage translation table walk.
15. Synchronous parity or ECC error on a level 3 lookup of a translation table walk.
16. Synchronous External abort on a level 3 lookup level of a translation table walk.
17. Translation fault on a level 3 translation table entry.
18. Address size fault on a level 3 lookup translation table entry caused by the output address being out of the range implemented.
19. Access Flag fault.
20. Alignment fault caused by the memory type.
21. Domain fault.

    Note

    Domain faults are possible only when using the VMSAv8-32 Short-descriptor translation table format, see *Domain fault, Short-descriptor format translation tables only* on page G5-6015.

22. Permission fault.
23. *A fault from the stage 2 translation of the memory access. When stage 2 address translation is enabled this includes an Address size fault caused by the PA being out of the range implemented*.
24. Synchronous parity or ECC error on the memory access.
25. Synchronous External abort on the memory access.

Note

- The prioritization of TLB Conflict aborts is IMPLEMENTATION DEFINED, as the exact cause of these aborts depends on the form of TLBs implemented. However, the TLB conflict abort must have higher priority than any abort that depends on a value held in the TLB.
- The prioritization of IMPLEMENTATION DEFINED MMU faults for a Load-Exclusive or Store-Exclusive to an unsupported memory type is IMPLEMENTATION DEFINED.

See also *The MMU fault-checking sequence* on page G5-6016.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Synchronous External abort errors from address translation caching structures

A caching structure used for caching translation table walks might support:

- An arbitrary number of levels of translation table lookup.
- One or more stages of translation, that might not correspond to the stages of an address translation lookup.

This might mean that, on a synchronous External abort arising from the caching structure, such as from a parity or ECC error, the PE cannot precisely determine one or both of the translation stage and level of lookup at which the error occurred. In this case:

- If the PE cannot determine precisely the translation stage at which the error occurred, it is reported and prioritized as a stage 1 error.
- If the PE cannot determine precisely the lookup level at which the error occurred, the level is reported and prioritized as either:
  - The lowest-numbered level that could have given rise to the error.
  - Level 1 if it the PE cannot determine any information about the level.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G5.12 Exception reporting in a VMSAv8-32 implementation

This section describes exception reporting, in AArch32 state, in a VMSAv8-32 implementation. That is, it describes only the reporting of exceptions that are taken to an Exception level that is using AArch32. EL2 provides an enhanced reporting mechanism for exceptions taken to the Non-secure EL2 mode, Hyp mode. This means that, for VMSAv8-32, the exception reporting depends on the mode to which the exception is taken.

Note

The enhanced reporting mechanism for exceptions that are taken to Hyp mode is generally similar to the reporting of exceptions that are taken to an Exception level that is using AArch64.

About exception reporting introduces the general approach to exception reporting, and the following sections then describe exception reporting at different privilege levels:

- *Reporting exceptions taken to PL1 modes* on page G5-6025.
- *Fault reporting in PL1 modes* on page G5-6028.
- *Summary of register updates on faults taken to PL1 modes* on page G5-6033.
- *Reporting exceptions taken to Hyp mode* on page G5-6034.
- *Use of the HSR* on page G5-6038.
- *Summary of register updates on exceptions taken to Hyp mode* on page G5-6041.

Note

The registers used for exception reporting also report information about debug exceptions. For more information see:

- *Data Abort exceptions, taken to a PL1 mode* on page G5-6026.
- *Prefetch Abort exceptions, taken to a PL1 mode* on page G5-6028.
- *Reporting exceptions taken to Hyp mode* on page G5-6034.

G5.12.1 About exception reporting

In an implementation that includes EL2 and EL3, exceptions can be taken to:

- Monitor mode, if EL3 is using AArch32.
- Hyp mode, if EL2 is using AArch32.
- A Secure or Non-secure PL1 mode.

Monitor mode is a PL1 mode, but:

- It is accessible only when EL3 is using AArch32.
- It is present only in Secure state.
- When EL3 is using AArch32, System register controls route some exceptions from Non-secure state to Monitor mode. These are the only cases where taking an exception to an Exception level that is using AArch32 changes the Security state of the PE.

Exception reporting in Hyp mode differs significantly from that in the other modes, but in general, exception reporting returns:

- Information about the exception:
  - On taking an exception to Hyp mode, the *Hyp Syndrome Register*, HSR, returns syndrome information.
  - On taking an exception to any other mode, a *Fault Status Register* (FSR) returns status information.
- For synchronous exceptions, one or more addresses associated with the exceptions, returned in *Fault Address Registers* (FARs). For a permitted exception to this requirement see *Fault address reporting on synchronous External aborts* on page G5-6025.

In all modes, additional IMPLEMENTATION DEFINED registers can provide additional information about exceptions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

- *PE mode for taking exceptions* on page G1-5713 describes how the mode to which an exception is taken is determined.
- EL2 provides:
  - Specific exception types, that can only be taken from Non-secure PL1 and EL0 modes, and are always taken to Hyp mode.
  - Routing controls that can route some exceptions from Non-secure PL1 and EL0 modes to Hyp mode.

  These exceptions are reported using the same mechanism as the Hyp mode reporting of VMSAv8-32 memory aborts, as described in this section.

Memory system faults generate either a Data Abort exception or a Prefetch Abort exception, as summarized in:

- *Reporting exceptions taken to PL1 modes*.
- *Memory fault reporting in Hyp mode* on page G5-6036.

On an access that might have multiple aborts, the MMU fault checking sequence and the prioritization of aborts determine which abort occurs. For more information, see *The MMU fault-checking sequence* on page G5-6016 and *AArch32 state prioritization of synchronous aborts from a single stage of address translation* on page G5-6021.

Fault address reporting on synchronous External aborts

The general architectural requirement is that, on a synchronous abort, the faulting address is recorded in a *Fault Address Register* (FAR). This requirement is relaxed for the case of a synchronous External abort that is not a synchronous External abort on a translation table walk. In this case only:

- It is IMPLEMENTATION DEFINED whether the faulting address is recorded in a FAR.
- A bit in a fault reporting register, the FnV bit, indicates whether a valid address is recorded.

For exceptions taken to an Exception level that is using AArch32, the details of this reporting depend on whether the exception is taken to:

- A PL1 mode, as described in *Reporting exceptions taken to PL1 modes*.
- Hyp mode, as described in *Reporting exceptions taken to Hyp mode* on page G5-6034.

G5.12.2 Reporting exceptions taken to PL1 modes

The following sections give general information about the reporting of exceptions when they are taken to a Secure or Non-secure PL1 mode:

- *Registers used for reporting exceptions taken to PL1 modes*.
- *Data Abort exceptions, taken to a PL1 mode* on page G5-6026.
- *Prefetch Abort exceptions, taken to a PL1 mode* on page G5-6028.

Fault reporting in PL1 modes on page G5-6028 then describes the fault reporting in these modes, including the encodings used for reporting the faults.

Note

Security state, Exception levels, and AArch32 execution privilege on page G1-5682 describes how the Secure and Non-secure PL1 modes map onto the Exception levels.

Registers used for reporting exceptions taken to PL1 modes

AArch32 state defines the following registers, and register encodings, for exceptions taken to PL1 modes:

- The DFSR holds information about a Data Abort exception.
- The DFAR holds the faulting address for some synchronous Data Abort exceptions.
- The IFSR holds information about a Prefetch Abort exception.
- The IFAR holds the faulting address for some synchronous Prefetch Abort exceptions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In addition, if implemented, the optional ADFSR and AIFSR can provide additional fault information, see *Auxiliary Fault Status Registers*.

Auxiliary Fault Status Registers

AArch32 state defines the following Auxiliary Fault Status Registers:

- The Auxiliary Data Fault Status Register, ADFSR.
- The Auxiliary Instruction Fault Status Register, AIFSR.

The position of these registers is architecturally-defined, but the content and use of the registers is IMPLEMENTATION DEFINED. An implementation can use these registers to return additional fault status information. An example use of these registers is to return more information for diagnosing parity or ECC errors.

An implementation that does not need to report additional fault information must implement these registers as RES0. This ensures that an attempt to access these registers from software executing at PL1 does not cause an Undefined Instruction exception.

Data Abort exceptions, taken to a PL1 mode

On taking a Data Abort exception to a PL1 mode:

- If the exception is on an instruction cache or branch predictor maintenance operation by VA, its reporting depends on the value of TTBCR.EAE. For more information about the registers used when reporting the exception, see *Data Abort on an instruction cache or branch predictor maintenance instruction by VA* on page G5-6027.

- Otherwise, the DFSR is updated with details of the fault, including the appropriate Fault status code. If the Data Abort exception is synchronous, DFSR.WnR is updated to indicate whether the faulted instruction was a read or a write. However, if the fault is on a cache maintenance instruction, or on an address translation instruction, WnR is set to 1, to indicate a fault on a write instruction, and the CM bit is set to 1.

  If the Data Abort is external, then DFSR provides fields for additional classification of the abort, see *Provision for classification of External aborts* on page G4-5914.

  If the RAS Extension is implemented, and the exception is a virtual SError interrupt exception, the classification reported in DFSR is taken from VDFSR or VSESR_EL2. For more information, see the *ARM® Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for the ARMv8-A architecture profile*.

  See the register description for more information about the returned fault information. See also *Data Abort on a Watchpoint exception* on page G5-6027.

  If the Data Abort exception is

  — Synchronous, the DFAR is updated with the VA that caused the exception, but see *Fault address reporting on synchronous External aborts* on page G5-6025 for a permitted exception to this requirement.

  — Asynchronous, the DFAR becomes UNKNOWN.

DFSR.WnR and DFSR.CM are UNKNOWN on an asynchronous Data Abort exception.

For all Data Abort exceptions, if the implementation includes EL3, the Security state of the PE in the mode to which the Data Abort exception is taken determines whether the Secure or Non-secure DFSR and DFAR are updated.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Data Abort on an instruction cache or branch predictor maintenance instruction by VA

If an instruction cache invalidation by VA or branch predictor invalidation by VA operation generates a Data Abort exception that is taken to a PL1 mode, the DFAR is updated to hold the faulting VA. However, the reporting of the fault depends on the value of TTBCR.EAE:

TTBCR.EAE == 0

When the value of TTBCR.EAE is 0, it is IMPLEMENTATION DEFINED which of the following is used when reporting the fault:

- The DFSR indicates an Instruction cache maintenance instruction fault, and the IFSR is valid and indicates the cause of the fault, a Translation fault or Access flag fault.
- The DFSR indicates the cause of the fault, a Translation fault or Access flag fault. The IFSR is UNKNOWN.

In either case:

- DFSR.WnR is set to 1.
- DFSR.CM is set to 1, to indicate a fault on a cache maintenance instruction.

TTBCR.EAE == 1

When the value of TTBCR.EAE is 1:

- DFSR.CM is set to 1, to indicate a fault on a cache maintenance instruction.
- DFSR.STATUS indicates the cause of the fault, a Translation or Access flag fault.
- DFSR.WnR is set to 1.
- The IFSR is UNKNOWN.

Data Abort on a Watchpoint exception

On taking a Data Abort exception caused by a watchpoint:

- DFSR.FS is updated to indicate a debug exception.
- DFSR.{WnR, Domain} are UNKNOWN.
- DFAR is set to the address that generated the watchpoint

Note

- LR_abt indicates the address of the instruction that triggered the watchpoint.
- In some Armv7 AArch32 implementations, the DBGWFAR is set to the address of the instruction that triggered the watchpoint. In Armv8 this register is RES0.

A watchpointed address can be any byte-aligned address. The address reported in DFAR might not be the watchpointed address, and:

- For a watchpoint due to an operation other than a Data Cache maintenance instruction, can be any address between and including:
  - The lowest address accessed by the instruction that triggered the watchpoint.
  - The highest watchpointed address accessed by that instruction.

  If multiple watchpoints are set in this range, there is no guarantee of which watchpoint is generated.

  The address must also be within a naturally-aligned block of memory of an IMPLEMENTATION DEFINED power-of-two size, containing a watchpoint address accessed by that location.

  Note

  - In particular, there is no guarantee of generating the watchpoint with the lowest address in the range.
  - The IMPLEMENTATION DEFINED power-of-two size must be no larger than the block size of the AArch64 DC ZVA operation.

- For a watchpoint due to a Data Cache operation, the address is the address passed to the instruction. This might be an address that is above the watchpointed location.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Prefetch Abort exceptions, taken to a PL1 mode

For a Prefetch Abort exception generated by an instruction fetch, the Prefetch Abort exception is taken synchronously with the instruction that the abort is reported on. This means:

- If the PE attempts to execute the instruction a Prefetch Abort exception is generated.
- If an instruction fetch is issued but the PE does not attempt to execute the prefetched instruction, no Prefetch Abort exception is generated for that instruction. For example, if the execution flow branches round a prefetched instruction, no Prefetch Abort exception is generated.

In addition, Breakpoint Instruction, Breakpoint, and Vector Catch exceptions, generate a Prefetch Abort exception, see the following for more information:

- *Exception syndrome information and preferred return address for a BKPT instruction* on page G2-5828.
- *Exception syndrome information and preferred return address for a Breakpoint exception* on page G2-5853.
- *Exception syndrome information and preferred return address for a Vector Catch exception* on page G2-5874.

Note

Usually, the term *exception syndrome* is used only for exceptions taken to Hyp mode, or to AArch64 state. The referenced sections use the term more generally, to include exception information reported in the IFSR.

On taking a Prefetch Abort exception to a PL1 mode:

- The IFSR is updated with details of the fault, including the appropriate fault code. If appropriate, the fault code indicates that the exception was generated by a debug exception.

  See the register description for more information about the returned fault information.
- For a Prefetch Abort exception generated by an instruction fetch, the IFAR is updated with the VA that caused the exception, but see *Fault address reporting on synchronous External aborts* on page G5-6025 for a permitted exception to this requirement.
- For a Prefetch Abort exception generated by a debug exception, the IFAR is UNKNOWN.

If the implementation includes EL3, the security state of the PE in the mode to which it takes the Prefetch Abort exception determines whether the exception updates the Secure or Non-secure IFSR and IFAR.

G5.12.3 Fault reporting in PL1 modes

The FSRs provide fault information, including an indication of the fault that occurred. The following subsections describe fault reporting in PL1 modes for each of the translation table formats:

- *PL1 fault reporting with the Short-descriptor translation table format* on page G5-6029.
- *PL1 fault reporting with the Long-descriptor translation table format* on page G5-6031.

Reserved encoding in the IFSR and DFSR encodings tables on page G5-6032 gives some additional information about the encodings for both formats.

Summary of register updates on faults taken to PL1 modes on page G5-6033 shows which registers are updated on each of the reported faults.

Reporting of External aborts taken from Non-secure state to Monitor mode describes how the fault status register format is determined for those aborts. For all other aborts, the current translation table format determines the format of the fault status registers.

Reporting of External aborts taken from Non-secure state to Monitor mode

When an External abort is taken from Non-secure state to Monitor mode:

- For a Data Abort exception, the Secure DFSR and DFAR hold information about the abort.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- For a Prefetch Abort exception, the Secure IFSR and IFAR hold information about the abort.
- The abort does not affect the contents of the Non-secure copies of the fault reporting registers.

Normally, the current translation table format determines the format of the DFSR and IFSR. However, when SCR.EA is set to 1, to route External aborts to Monitor mode, and an External abort is taken from Non-secure state, this section defines the DFSR and IFSR format.

For an External abort taken from Non-secure state to Monitor mode, the DFSR or IFSR uses the format associated with the Long-descriptor translation table format, as described in *PL1 fault reporting with the Long-descriptor translation table format* on page G5-6031, if any of the following applies:

- The value of the Secure TTBCR.EAE field is 1.
- The External abort is synchronous and is taken from either:
  - Hyp mode.
  - A Non-secure PL1 or EL0 mode, and the value of the Non-secure TTBCR.EAE field is 1.

Otherwise:

- For a synchronous External abort from a stage 2 translation routed to Monitor mode when the value of the Secure TTBCR.EAE field is 0 it is IMPLEMENTATION DEFINED whether:
  - The format associated with the Long-descriptor translation table format is used, as described in *PL1 fault reporting with the Long-descriptor translation table format* on page G5-6031.
  - The format associated with the Short-descriptor translation table format is used, as described in *PL1 fault reporting with the Short-descriptor translation table format*. Arm deprecates using this format.

    When this format is used, the value of DFSR.FS[1] or IFSR.FS[1] is UNKNOWN when reporting a synchronous External abort, or a synchronous parity or ECC error, on the stage 2 translation.
- In all other cases the DFSR or IFSR uses the format associated with the Short-descriptor translation table format, as described in *PL1 fault reporting with the Short-descriptor translation table format*.

PL1 fault reporting with the Short-descriptor translation table format

This subsection describes the fault reporting for a fault taken to a PL1 when address translation is using the Short-descriptor translation table format.

On taking an exception, bit[9] of the FSR is RAZ, or set to 0, if the PE is using this FSR format.

An FSR encodes the fault in a 5-bit FS field, that comprises FSR[10, 3:0]. Table G5-26 shows the encoding of that field. *Summary of register updates on faults taken to PL1 modes* on page G5-6033 shows:

- Whether the corresponding FAR is updated on the fault. That is:
  - For a fault reported in the IFSR, whether the IFAR holds a valid address.
  - For a fault reported in the DFSR, whether the DFAR holds a valid address.
- For faults that update DFSR, whether DFSR.Domain is valid

When reading Table G5-26:

- FS values not shown in the table are reserved.
- FS values shown as DFSR only are reserved for the IFSR.

Table G5-26 FSR encodings when using the Short-description translation table format

FS	Source		Notes
00001	Alignment fault		DFSR only. Fault on initial lookup
00100	Fault on instruction cache maintenance		DFSR only
01100 01110	Synchronous External abort on translation table walk[a, b]	Level 1 Level 2	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G5-26 FSR encodings when using the Short-description translation table format (continued)

FS	Source		Notes
11100 11110	Synchronous parity or ECC error on translation table walk[a, b]	Level 1 Level 2	-
00101 00111	Translation fault[a]	Level 1 Level 2	MMU fault
00011[c] 00110	Access flag fault[a]	Level 1 Level 2	MMU fault
01001 01011	Domain fault[a]	Level 1 Level 2	MMU fault
01101 01111	Permission fault[a]	Level 1 Level 2	MMU fault
00010	Debug exception		See Chapter G2 *AArch32 Self-hosted Debug*
01000	Synchronous External abort		-
10000	TLB conflict abort		See *TLB conflict aborts* on page G5-5993
10100	IMPLEMENTATION DEFINED		Lockdown
10101	IMPLEMENTATION DEFINED		Unsupported Exclusive access
11001	Synchronous parity or ECC error on memory access		-
10110	SError interrupt[d]		DFSR only
11000	SError interrupt[d] from a parity or ECC error on memory access		DFSR only

a. See *The level associated with MMU faults on a Short-descriptor translation table lookup*.
b. FS[1] is UNKNOWN if the reported error is from a stage 2 translation.
c. Previously, this encoding was a deprecated encoding for Alignment fault. The extensive changes in the memory model in VMSAv8-32 mean there should be no possibility of confusing the new use of this encoding with its previous use
d. Including asynchronous External abort on a data access, a translation table walk, or an instruction fetch.

The level associated with MMU faults on a Short-descriptor translation table lookup

The lookup level associated with a fault is:

- For a fault generated on a translation table walk, the lookup level of the walk being performed.
- For a Translation fault, the lookup level of the translation table that gave the fault. If a fault occurs because a stage of address translation is disabled, or because the input address is outside the range specified by the appropriate base address register or registers, the fault is reported as a level 1 fault.
- For an Access flag fault, Permission fault, or Domain fault, the lookup level of the final level of translation table accessed for the translation. That is, the lookup level of the translation table that returned a Supersection, Section, or Page descriptor.

Also see *Synchronous External abort errors from address translation caching structures* on page G5-6023.

The Domain field in the DFSR

The DFSR includes a Domain field. This is inherited from previous versions of the VMSA. The IFSR does not include a Domain field. *Summary of register updates on faults taken to PL1 modes* on page G5-6033 describes when DFSR.Domain is valid.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Arm deprecates any use of the Domain field in the DFSR. The Long-descriptor translation table format does not support a Domain field, and future versions of the Arm architecture might not support a Domain field in the Short-descriptor translation table format. Arm strongly recommends that new software does not use this field.

For both Data Abort exceptions and Prefetch Abort exceptions, software can find the domain information by performing a translation table read for the faulting address and extracting the Domain field from the translation table entry.

PL1 fault reporting with the Long-descriptor translation table format

This subsection describes the fault reporting for a fault taken to a PL1mode when address translation is using the Long-descriptor translation table format.

When the PE takes an exception, bit[9] of the FSR is set to 1 if the PE is using this FSR format.

The FSRs encode the fault in a 6-bit STATUS field, that comprises FSR[5:0]. Table G5-27 shows the encoding of that field. In addition:

- For a fault taken to a PL1 mode, *Summary of register updates on faults taken to PL1 modes* on page G5-6033 shows whether the corresponding FAR is updated on the fault. That is:
  - For a fault reported in the IFSR, whether the IFAR holds a valid address.
  - For a fault reported in the DFSR, whether the DFAR holds a valid address.
- For a fault taken to the Hyp mode, *Summary of register updates on exceptions taken to Hyp mode* on page G5-6041 shows what registers are updated on the fault.

Table G5-27 FSR encodings when using the Long-descriptor translation table format

STATUS[a]	Source	Notes
0000LL	Address size fault. LL bits indicate level[b].	MMU fault
0001LL	Translation fault. LL bits indicate level[b].	MMU fault
0010LL	Access flag fault. LL bits indicate level[b].	MMU fault
0011LL	Permission fault. LL bits indicate level[b].	MMU fault
010000	Synchronous External abort.	-
011000	Synchronous parity or ECC error on memory access.	-
010001	SError interrupt[c].	DFSR only
011001	SError interrupt[c] from a parity or ECC error on memory access.	DFSR only
0101LL	Synchronous External abort on translation table walk. LL bits indicate level[b].	-
0111LL	Synchronous parity or ECC error on memory access on translation table walk. LL bits indicate level[b].	-
100001	Alignment fault.	Fault on initial lookup
100010	Debug exception.	See Chapter G2 *AArch32 Self-hosted Debug*
110000	TLB conflict abort.	See *TLB conflict aborts* on page G5-5993

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G5-27 FSR encodings when using the Long-descriptor translation table format (continued)

STATUS[a]	Source	Notes
110100	IMPLEMENTATION DEFINED.	Lockdown, DFSR only
110101	IMPLEMENTATION DEFINED.	Unsupported Exclusive access
1111LL	Domain fault. LL bits indicate level[b].	MMU fault. 64-bit PAR only, level 1 or level 2 only. Never used in DFSR, IFSR, or HSR[d]

a. STATUS values not shown in this table are reserved. STATUS values not supported in the IFSR or DFSR are reserved for the register or registers in which they are not supported.

b. See *The level associated with MMU faults on a Long-descriptor translation table lookup*.

c. Including asynchronous External abort on a data access, a translation table walk, or an instruction fetch.

d. A Domain fault can be reported using the Long-descriptor STATUS encodings only as a result of a fault on an address translation instruction. For more information see *MMU fault on an address translation instruction* on page G5-6047.

The level associated with MMU faults on a Long-descriptor translation table lookup

For MMU faults, Table G5-28 shows how the LL bits in the xFSR.STATUS field encode the lookup level associated with the fault.

Table G5-28 Use of LL bits to encode the lookup level at which the fault occurred

LL bits	Meaning	
00	**Address size fault**	Address size fault in TTBR0 or TTBR1.
	All other faults	Reserved.
01	Level 1.	
10	Level 2.	
11	Level 3. When xFSR.STATUS indicates a Domain fault, this value is reserved.	

The lookup level associated with a fault is:

- For a fault generated on a translation table walk, the lookup level of the walk being performed.
- For a Translation fault, the lookup level of the translation table that gave the fault. If a fault occurs because a stage of address translation is disabled, or because the input address is outside the range specified by the appropriate base address register or registers, the fault is reported as a level 1 fault.
- For an Access flag fault, the lookup level of the translation table that gave the fault.
- For a Permission fault, including a Permission fault caused by hierarchical permissions, the lookup level of the final level of translation table accessed for the translation. That is, the lookup level of the translation table that returned a Block or Page descriptor.

Also see *Synchronous External abort errors from address translation caching structures* on page G5-6023.

Reserved encoding in the IFSR and DFSR encodings tables

With both the Short-descriptor and the Long-descriptor FSR format, the fault encodings reserve a single encoding for Cache and TLB lockdown faults. The details of these faults and any associated subsidiary registers are IMPLEMENTATION DEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G5.12.4 Summary of register updates on faults taken to PL1 modes

For faults that generate exceptions that are taken to a PL1 mode, Table G5-29 shows the registers affected by each fault. In this table:

- Yes indicates that the register is updated.
- UNK indicates that the fault makes the register value UNKNOWN.
- A null entry, -, indicates that the fault does not affect the register.

For faults that update the DFSR using the Short-descriptor format FSR encodings, Table G5-30 on page G5-6034 shows whether DFSR.Domain is valid.

Table G5-29 Effect of a fault taken to a PL1 mode on the reporting registers

Fault		IFSR	IFAR	DFSR	DFAR
Faults reported as Prefetch Abort exceptions:					
MMU fault, always synchronous		Yes	Yes	-	-
Synchronous External abort on translation table walk		Yes	Yes	-	-
Synchronous parity or ECC error on translation table walk		Yes	Yes	-	-
Synchronous External abort		Yes	IMP DEF[a]	-	-
Synchronous parity or ECC error on memory access		Yes	Yes	-	-
TLB conflict abort		Yes	Yes	-	-
Fault reported as Data Abort exception:					
Alignment fault, always synchronous		-	-	Yes	Yes
MMU fault, always synchronous		-	-	Yes	Yes
Fault on instruction cache maintenance, when using Long-descriptor translation table format[b]		UNK	-	Yes	Yes
Fault on instruction cache maintenance, when using Short descriptor translation table format[c]	*either*	Yes	-	Yes	Yes
	or	UNK	-	Yes	Yes
Synchronous External abort on translation table walk		-	-	Yes	Yes
Synchronous parity or ECC error on translation table walk		-	-	Yes	Yes
Synchronous External abort		-	-	Yes	IMP DEF[a]
Synchronous parity or ECC error on memory access		-	-	Yes	Yes
SError interrupt		-	-	Yes	UNK
SError interrupt from a parity or ECC error on memory access		-	-	Yes	UNK
TLB conflict abort		-	-	Yes	Yes
Debug exceptions:					
Breakpoint, Breakpoint Instruction, or Vector Catch[d]		Yes	UNK	-	-
Watchpoint[e]		-	-	Yes	Yes

a. IMPLEMENTATION DEFINED. The IFSR.FnV or DFSR.FnV bit indicates whether the register holds a valid address. See *Fault address reporting on synchronous External aborts* on page G5-6025.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

b. When using the Long-descriptor translation table format, there is not a specific fault code for a fault on an instruction cache maintenance instruction. For more information see *Data Abort on an instruction cache or branch predictor maintenance instruction by VA* on page G5-6027.

c. The two lines of this entry show the alternative ways of reporting the fault when using the Short-descriptor translation table format. It is IMPLEMENTATION DEFINED which methods is used, see *Data Abort on an instruction cache or branch predictor maintenance instruction by VA* on page G5-6027.

d. Generates a Prefetch Abort exception.

e. Generates a Data Abort exception.

For those faults for which Table G5-29 on page G5-6033 shows that the DFSR is updated, if the fault is reported using the Short-descriptor FSR encodings, Table G5-30 shows whether DFSR.Domain is valid. In this table, UNK indicates that the fault makes DFSR.Domain UNKNOWN.

Table G5-30 Validity of Domain field on faults that update the DFSR when using the Short-descriptor encodings

DFSR.FS	Source		DFSR.Domain	Notes
00001	Alignment fault		UNK	-
00100	Fault on instruction cache maintenance instruction		UNK	-
01100 01110	Synchronous External abort on translation table walk	Level 1 Level 2	UNK Valid	-
11100 11110	Synchronous parity or ECC error on translation table walk	Level 1 Level 2	UNK Valid	-
00101 00111	Translation fault	Level 1 Level 2	UNK Valid	MMU fault
00011[a] 00110	Access flag fault	Level 1 Level 2	UNK Valid	MMU fault
01001 01011	Domain fault	Level 1 Level 2	Valid Valid	MMU fault
01101 01111	Permission fault	Level 1 Level 2	UNK UNK	MMU fault
01000	Synchronous External abort		UNK	-
10000	TLB conflict abort		UNK	-
11001	Synchronous parity or ECC error on memory access		UNK	-
10110	SError interrupt[b]		UNK	-
11000	SError interrupt[b] from a parity or ECC error on memory access		UNK	-
00010	Watchpoint		UNK	

a. Previously, this encoding was a deprecated encoding for Alignment fault. The extensive changes in the memory model in VMSAv8-32 mean there should be no possibility of confusing the new use of this encoding with its previous use

b. Including asynchronous External abort on a data access, a translation table walk, or an instruction fetch.

G5.12.5 Reporting exceptions taken to Hyp mode

Hyp mode is the Non-secure EL2 mode. It is entered by taking an exception to Hyp mode.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

Software executing in Monitor mode, or at EL3 when EL3 is using AArch64, can perform an exception return to Hyp mode. This means Hyp mode can be entered either by taking an exception, or by a permitted exception return.

When EL2 is using AArch32, the following exceptions are taken to Hyp mode:

- SError interrupt exceptions, IRQ exceptions, and FIQ exceptions, from Non-secure PL1 and EL0 modes, if not routed to Secure Monitor mode, and if configured by the AMO, FMO or IMO bits. For more information see *Asynchronous exception routing controls* on page G1-5732.
- When HCR.TGE is set to 1, all exceptions that would be routed to Non-secure PL1 modes.

  For more information, see *Routing exceptions from Non-secure EL0 to EL2* on page G1-5718.
- When HDCR.TDE is set to 1, any debug exception that would otherwise be taken to a Non-secure PL1 mode, see *Routing debug exceptions to EL2 using AArch32* on page G1-5720.
- The privilege rules for taking exceptions mean that any exception taken from Hyp mode, if not routed to EL3, must be taken to Hyp mode.
- An abort that *Routing of aborts taken to AArch32 state* on page G1-5722 identifies as taken to Hyp mode.
- Hypervisor Call exceptions, and Hyp Trap exceptions, are always taken to Hyp mode. These exceptions are supported only as part of EL2.

  When EL2 is implemented, various operations from Non-secure PL1 and EL0 modes can be *trapped* to Hyp mode, using the Hyp Trap exception. For more information, see *EL2 configurable controls* on page G1-5787.

Synchronous exceptions taken to Hyp mode provide syndrome information in the HSR.

On an abort exception taken to Hyp mode, the syndrome information in the HSR includes the Fault status code otherwise provided by the fault status register, and extends the fault reporting compared to that available for an exception taken to a PL1 mode.

In addition, for a Debug exception taken to Hyp mode, DBGDSCRint.MOE or DBGDSCRext.MOE shows what caused the Debug exception. This field is valid regardless of whether the Debug exception was taken from Hyp mode or from another Non-secure mode.

For more information, see the following subsections:

- *Registers used for reporting exceptions taken to Hyp mode*.
- *Memory fault reporting in Hyp mode* on page G5-6036.
- *Use of the HSR* on page G5-6038

Registers used for reporting exceptions taken to Hyp mode

The following registers are used for reporting exceptions taken to Hyp mode:

- The HSR holds syndrome information for the exception.
- The HDFAR holds the VA associated with a Data Abort exception.
- The HIFAR holds the VA associated with a Prefetch Abort exception.
- The HPFAR holds bits[39:12] of the IPA associated with some aborts on stage 2 address translations.

In addition, if implemented, the optional HADFSR and HAIFSR can provide additional fault information, see *Hyp Auxiliary Fault Syndrome Registers*.

Hyp Auxiliary Fault Syndrome Registers

EL2 also defines encodings for the following Hyp Auxiliary Fault Syndrome Registers:

- The Hyp Auxiliary Data Fault Syndrome Register, HADFSR.
- The Hyp Auxiliary Instruction Fault Syndrome Register, HAIFSR.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

An implementation can use these registers to return additional fault status information for aborts taken to Hyp mode. They are the Hyp mode equivalents of the registers described in *Auxiliary Fault Status Registers* on page G5-6026. An example use of these registers is to return more information for diagnosing parity or ECC errors.

The architectural requirements for the HADFSR and HAIFSR are:

- The position of these registers is architecturally-defined, but the content and use of the registers is IMPLEMENTATION DEFINED.
- An implementation with no requirement for additional fault reporting can implement these registers as RES0, but the architecture does not require it to do so.

Memory fault reporting in Hyp mode

Prefetch Abort and Data Abort exceptions taken to Hyp mode report memory faults. For these aborts, the HSR contains the following fault status information:

- The HSR.EC field indicates the type of abort, as Table G5-31 shows.
- The HSR.ISS field holds more information about the abort. In particular:
  - Bits[5:0] of this field hold the STATUS field for the abort, using the encodings defined in *PL1 fault reporting with the Long-descriptor translation table format* on page G5-6031.
  - Other subfields of the ISS give more information about the exception, equivalent to the information returned in the FSR for a memory fault reported at PL1.

  See the descriptions of the ISS fields for the memory faults, referenced from the *Syndrome description* column of Table G5-31, for information about the returned fault information.

Table G5-31 HSR.EC encodings for aborts taken to Hyp mode

HSR.EC	Abort	Syndrome description
0x20	Prefetch Abort taken from Non-secure PL1 or EL0 mode	*ISS encoding for an exception from a Prefetch Abort* on page G8-6301
0x21	Prefetch Abort taken from Hyp mode	
0x24	Data Abort taken from Non-secure PL1 or EL0 mode	*ISS encoding for an exception from a Data Abort* on page G8-6303
0x25	Data Abort taken from Hyp mode	

For more information, see *Use of the HSR* on page G5-6038.

A Prefetch Abort exception is taken synchronously with the instruction that the abort is reported on. This means:

- If the PE attempts to execute the instruction a Prefetch Abort exception is generated.
- If an instruction fetch is issued but the PE does not attempt to execute the prefetched instruction, no Prefetch Abort exception is generated for that instruction. For example, if the execution flow branches round a prefetched instruction that would abort if the PE attempted to execute it, no Prefetch Abort exception is generated.

Register updates on exception reporting in Hyp mode

The use of the HSR, and of the other registers listed in *Registers used for reporting exceptions taken to Hyp mode* on page G5-6035, depends on the cause of the Abort. In reporting these faults, in general:

- If the fault generates a synchronous Data Abort exception, the HDFAR holds the associated VA, but see *Fault address reporting on synchronous External aborts* on page G5-6025 for a permitted exception to this requirement.
- If the fault generates a Prefetch Abort exception, the HIFAR holds the associated VA, but see *Fault address reporting on synchronous External aborts* on page G5-6025 for a permitted exception to this requirement.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- In the following cases, the HPFAR holds the faulting IPA:
  - A Translation or Access flag fault on a stage 2 translation.
  - A Translation, Access flag, or Permission fault on the stage 2 translation of an address accessed in a stage 1 translation table walk.
  - A stage 2 Address size fault.

  In all other cases, the HPFAR is UNKNOWN.
- On a Data Abort exception that is taken to Hyp mode, the HIFAR is UNKNOWN.
- On a Prefetch Abort exception that is taken to Hyp mode, the HDFAR is UNKNOWN.

In addition, the reporting of particular aborts is as follows:

Abort on the stage 1 translation for a memory access from Hyp mode

The HDFAR or HIFAR holds the VA that caused the fault. The STATUS subfield of HSR.ISS indicates the type of fault, Translation, Address size, Access flag, or Permission. The HPFAR is UNKNOWN.

Abort on the stage 2 translation for a memory access from a Non-secure PL1 or EL0 mode

This includes aborts on the stage 2 translation of a memory access made as part of a translation table walk for a stage 1 translation. The HDFAR or HIFAR holds the VA that caused the fault. The STATUS subfield of HSR.ISS indicates the type of fault, Translation, Address size, Access flag, or Permission.

For any Access flag fault or Translation fault, and also for any Permission fault on the stage 2 translation of a memory access made as part of a translation table walk for a stage 1 translation, the HPFAR holds the IPA that caused the fault. Otherwise, the HPFAR is UNKNOWN.

Abort caused by a synchronous External abort, or synchronous parity or ECC error, and taken to Hyp mode

The HDFAR or HIFAR holds the VA that caused the fault, but see *Fault address reporting on synchronous External aborts* on page G5-6025 for a permitted exception to this requirement. The HPFAR is UNKNOWN.

Data Abort caused by a Watchpoint exception and routed to Hyp mode because HDCR.TDE is set to 1

When HDCR.TDE is set to 1, a Watchpoint exception generated in a Non-secure PL1 or EL0 mode, that would otherwise generate a Data Abort exception, is routed to Hyp mode and generates a Hyp Trap exception.

HDFAR is set to the address that generated the watchpoint.

Note

ELR_hyp indicates the address of the instruction that triggered the watchpoint.

A watchpointed address can be any byte-aligned address. The address reported in HDFAR might not be the watchpointed address, and, for a watchpoint due to an operation other than a Data Cache maintenance instruction, can be any address between and including:

- The lowest address accessed by the instruction that triggered the watchpoint.
- The highest watchpointed address accessed by that instruction.

If multiple watchpoints are set in this range, there is no guarantee of which watchpoint is generated.

Note

In particular, there is no guarantee of generating the watchpoint with the lowest address in the range.

The address must also be within a naturally-aligned block of memory of an IMPLEMENTATION DEFINED power-of-two size, containing a watchpoint address accessed by that location.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

The IMPLEMENTATION DEFINED power-of-two size must be no larger than the block size of the AArch64 DC ZVA operation.

See also *Watchpoint exceptions* on page G2-5855.

In all cases, HPFAR is UNKNOWN.

Prefetch Abort caused by a Breakpoint Instruction exception and taken to Hyp mode

This abort is generated if a `BKPT` instruction is executed in Hyp mode. The abort leaves the HIFAR and HPFAR UNKNOWN.

See also *Breakpoint Instruction exceptions* on page G2-5827.

Prefetch Abort caused by a Breakpoint Instruction, Breakpoint, or Vector Catch exception, and routed to Hyp mode because HDCR.TDE is set to 1

When HDCR.TDE is set to 1, a debug exception, generated in a Non-secure PL1 or EL0 mode, that would otherwise generate a Prefetch Abort exception, is routed to Hyp mode and generates a Hyp Trap exception.

The abort leaves the HIFAR and HPFAR UNKNOWN. This is identical to the reporting of a Prefetch Abort exception caused by a Debug exception on a `BKPT` instruction that is executed in Hyp mode.

Note

The difference between these two cases is:

- The Debug exception on a `BKPT` instruction executed in Hyp mode generates a Prefetch Abort exception, taken to Hyp mode, and reported in the HSR using EC value `0x21`.
- Aborts generated because HDCR.TDE is set to 1 generate a Hyp Trap exception, and are reported in the HSR using EC value `0x20`.

Use of the HSR

The HSR holds syndrome information for any synchronous exception taken to Hyp mode. Compared with the reporting of exceptions taken to PL1 modes, the HSR:

- Always provides details of the fault. The DFSR and IFSR are not used.
- Provides more extensive information, for a wider range of exceptions.

Note

IRQ and FIQ exceptions taken to Hyp mode do not report any syndrome information in the HSR.

This section summarizes the general form of the HSR register, to show how it encodes exception syndrome information, see the register description for more information. The register comprises:

- A 6-bit Exception class field, EC, that indicates the cause of the exception.
- An instruction length bit, IL. When an exception is caused by trapping an instruction to Hyp mode, this bit indicates the length of the trapped instruction, as follows:

  0 16-bit instruction trapped.

  1 32-bit instruction trapped.

  In other cases the IL field is not valid and is RES1.
- An instruction specific syndrome field, ISS. Architecturally, this field could be defined independently for each defined Exception class (EC), but in practice several ISS formats are common to more than one EC.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The format of the HSR depends on the value of the EC field, as follows:

0b000000<EC≤0b001100

The ISS part of the returned value includes the CV and COND fields described in *Encoding of ISS[24:20] when 0b000000<EC≤0b001100*. Figure G5-17 shows the HSR format in this case.

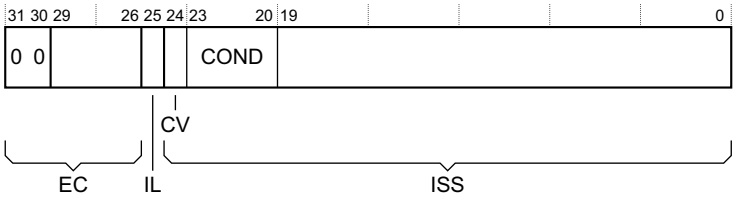


Figure G5-17 HSR format when the ISS includes CV and COND fields

EC==0b000000 or EC0b001110 There are no generic fields within the ISS. Figure G5-18 shows the HSR format in this case.

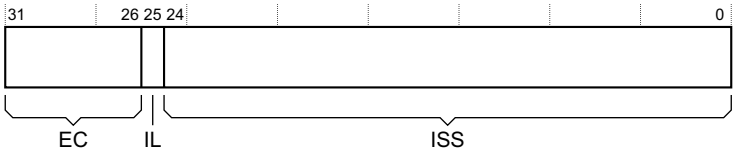


Figure G5-18 HSR format when the ISS does not include a COND field

Encoding of ISS[24:20] when 0b000000<EC≤0b001100

For EC values that are nonzero and less than or equal to `0b001100`, ISS[24:20] provides the Condition code field for the trapped instruction, together with a valid flag for this field. The encoding of this part of the ISS field is:

CV, ISS[24] Condition code valid. Possible values of this bit are:

0 The COND field is not valid.

1 The COND field is valid.

COND, ISS[23:20]

The Condition code for the trapped instruction. This field is valid only when CV is set to 1.

If CV is set to 0, this field is RES0.

The full descriptions of the HSR.ISS formats give more information about the CV field.

Note

In some circumstances, it is IMPLEMENTATION DEFINED whether a conditional instruction that fails its Condition code check generates an Undefined Instruction exception, see *Conditional execution of undefined instructions* on page G1-5740.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

HSR exception classes

Table G5-32 shows the encoding of the HSR exception class field, EC. Values of EC not shown in the table are reserved. For each EC value, the table references a subsection of the description of the HSR that describes the associated ISS format and gives information about the cause of the exception, for example the configuration required to enable the associated trap.

Table G5-32 HSR.EC field encoding

EC	Exception class	ISS description, or notes
`0b000000`	Unknown reason	*ISS encoding for exceptions with an unknown reason* on page G8-6290.
`0b000001`	Trapped `WFI` or `WFE` instruction	*ISS encoding for an exception from a WFI or WFE instruction* on page G8-6291.
`0b000011`	Trapped `MCR` or `MRC` access with (`coproc==0b1111`)	*ISS encoding for an exception from an MCR or MRC access* on page G8-6292.
`0b000100`	Trapped `MCRR` or `MRRC` access with (`coproc==0b1111`)	*ISS encoding for an exception from an MCRR or MRRC access* on page G8-6294.
`0b000101`	Trapped `MCR` or `MRC` access with (`coproc==0b1110`)	*ISS encoding for an exception from an MCR or MRC access* on page G8-6292.
`0b000110`	Trapped `LDC` or `STC` access	*ISS encoding for an exception from an LDC or STC instruction* on page G8-6296.
`0b000111`	Advanced SIMD or floating-point functionality trapped by a HCPTR.{TASE, TCP10} control	*ISS encoding for an exception from an access to SIMD or floating-point functionality, resulting from HCPTR* on page G8-6298.
`0b001000`	Trapped `VMRS` access, from ID group traps, that is not reported using EC `0b000111`	*ISS encoding for an exception from an MCR or MRC access* on page G8-6292. This trap is not taken if the HCPTR settings trap the access.
`0b001100`	Trapped `MRRC` access with (`coproc==0b1110`)	*ISS encoding for an exception from an MCRR or MRRC access* on page G8-6294.
`0b001110`	Illegal exception return to AArch32 state	*ISS encoding for an exception from an Illegal state or PC alignment fault* on page G8-6302.
`0b010001`	Exception on `SVC` execution in AArch32 state routed to EL2	*ISS encoding for an exception from HVC or SVC instruction execution* on page G8-6299.
`0b010010`	`HVC` instruction execution in AArch32 state, when `HVC` is not disabled	
`0b010011`	Trapped execution of `SMC` instruction in AArch32 state	*ISS encoding for an exception from SMC instruction execution* on page G8-6300.
`0b100000`	Prefetch Abort from a lower Exception level	*ISS encoding for an exception from a Prefetch Abort* on page G8-6301.
`0b100001`	Prefetch Abort taken without a change in Exception level	
`0b100010`	PC alignment exception.	*ISS encoding for an exception from an Illegal state or PC alignment fault* on page G8-6302.
`0b100100`	Data Abort from a lower Exception level	*ISS encoding for an exception from a Data Abort* on page G8-6303.
`0b100101`	Data Abort taken without a change in Exception level	

All EC encodings not shown in Table G5-31 on page G5-6036 are reserved by Arm.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G5.12.6 Summary of register updates on exceptions taken to Hyp mode

For memory system faults that generate exceptions that are taken to Hyp mode, Table G5-33 shows the registers affected by each fault. In this table:

- Yes indicates that the register is updated.
- UNK indicates that the fault makes the register value UNKNOWN.
- A null entry, -, indicates that the fault does not affect the register.

Note

For a list of the MMU faults see *Types of MMU faults* on page G5-6013.

Table G5-33 Effect of an exception taken to Hyp mode on the reporting registers

Fault	HSR	HIFAR	HDFAR	HPFAR
Faults reported as Prefetch Abort exceptions:				
MMU fault[a] at stage 1.	Yes	Yes	UNK	UNK
Translation or Access flag MMU fault[a] at stage 2.	Yes	Yes	UNK	Yes
Other[b] MMU fault[a] at stage 2.	Yes	Yes	UNK	UNK
Stage 2 MMU fault[a] on a stage 1 translation.	Yes	Yes	UNK	Yes
Synchronous External abort on translation table walk.	Yes	Yes	UNK	UNK
Synchronous parity or ECC error on translation table walk.	Yes	Yes	UNK	UNK
Synchronous External abort.	Yes	IMP DEF[c]	UNK	UNK
Synchronous parity or ECC error on memory access.	Yes	Yes	UNK	UNK
Fault reported as Data Abort exception:				
MMU fault[a] at stage 1.	Yes	UNK	Yes	UNK
Translation or Access flag MMU fault[a] at stage 2.	Yes	UNK	Yes	Yes
Other[b] MMU fault[a] at stage 2.	Yes	UNK	Yes	UNK
Stage 2 MMU fault[a] on a stage 1 translation.	Yes	UNK	Yes	Yes
Synchronous External abort on translation table walk.	Yes	UNK	Yes	UNK
Synchronous parity or ECC error on translation table walk.	Yes	UNK	Yes	UNK
Synchronous External abort.	Yes	UNK	IMP DEF[c]	UNK
Synchronous parity or ECC error on memory access.	Yes	UNK	Yes	UNK
SError interrupt	Yes	UNK	UNK	UNK
SError interrupt from a parity or ECC error on memory access.	Yes	UNK	UNK	UNK
Debug exception:				
Breakpoint Instruction[d], generates a Prefetch Abort exception.	Yes	UNK	-	UNK

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G5-33 Effect of an exception taken to Hyp mode on the reporting registers (continued)

Fault	HSR	HIFAR	HDFAR	HPFAR
Debug exception routed to Hyp mode because HDCR.TDE is set to 1. Generates a Hyp Trap exception.				
Breakpoint Instruction or Vector Catch	Yes	UNK	-	UNK
Watchpoint	Yes	-	Yes	UNK

a. For more information see *Classification of MMU faults taken to Hyp mode*
b. MMU fault other than a Translation fault or an Access flag fault.
c. IMPLEMENTATION DEFINED. The FnV bit in the HSR.ISS field indicates whether the register holds a valid address. See *Fault address reporting on synchronous External aborts* on page G5-6025.
d. All other debug exceptions are not permitted in Hyp mode.

Note

Unlike Table G5-29 on page G5-6033, the Hyp mode fault reporting table does not include an entry for a fault on an instruction cache maintenance instruction. That is because, when the fault is taken to Hyp mode, the reporting indicates the cause of the fault, for example a Translation fault, and ISS.CM is set to 1 to indicate that the fault was on a cache maintenance instruction, see *ISS encoding for an exception from a Data Abort* on page G8-6303.

Classification of MMU faults taken to Hyp mode

This subsection gives more information about the MMU faults shown in Table G5-33 on page G5-6041.

Note

All MMU faults are synchronous.

The table uses the following descriptions for MMU faults taken to Hyp mode:

MMU fault at stage 1

This is an MMU fault generated on a stage 1 translation performed in the Non-secure EL2 translation regime.

MMU fault at stage 2

This is an MMU fault generated on a stage 2 translation performed in the Non-secure PL1&0 translation regime.

As the table shows, for the faults in this group:

- Translation and Access flag faults update the HPFAR
- Permission faults leave the HPFAR UNKNOWN.

MMU stage 2 fault on a stage 1 translation

This is an MMU fault generated on the stage 2 translation of an address accessed in a stage 1 translation table walk performed in the Non-secure PL1&0 translation regime. For more information about these faults see *Stage 2 fault on a stage 1 translation table walk* on page G5-6019.

Figure G5-1 on page G5-5924 shows the different translation regimes and associated stages of translation.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G5.13 Address translation instructions

The System register encoding space includes encodings for instructions that either:

- Translate a *virtual address* (VA) to a *physical address* (PA).
- Translate a *virtual address* (VA) to an *intermediate physical address* (IPA).

Address translation system instructions on page K14-8090 summarizes these instructions.

When using the Short-descriptor translation table format, all translations performed by these instructions take account of TEX remap when this is enabled, see *Short-descriptor format memory region attributes, with TEX remap* on page G5-5982.

An address translation instruction that executes successfully returns the output address, a PA or an IPA, in the PAR. This is a 64-bit register, that can hold addresses of up to 40 bits.

It is IMPLEMENTATION DEFINED whether the address translation instructions return the values held in a TLB or the result of a translation table walk. Therefore, Arm recommends that these instructions are not used at a time when the TLB entries might be different from the underlying translation tables held in memory.

The following sections give more information about these instructions:

- *Address translation instruction naming and operation summary*.
- *Encoding and availability of the address translation instructions* on page G5-6045.
- *Determining the PAR format* on page G5-6047.
- *Handling of faults and aborts during an address translation instruction* on page G5-6047.

G5.13.1 Address translation instruction naming and operation summary

Some older documentation uses the original names for the address translation instructions that were included in the original Armv7 documentation. Table G5-34 summarizes the instructions that are available in AArch32 state, and relates the old instruction names to the current names.

Table G5-34 Naming of address translation instructions

Name	Old name	Description
ATS1CPR, ATS1CPW, ATS1CUR, ATS1CUW	V2PCWPR, V2PCWPW, V2PCWUR, V2PCWUW	See *ATS1C**, Address translation stage 1, current security state* on page G5-6044
ATS1CPRP, ATS1CPWP	Not applicable[a]	
ATS12NSOPR, ATS12NSOPW, ATS12NSOUR, ATS12NSOUW	V2POWPR, V2POWPW, V2POWUR, V2POWUW	See *ATS12NSO**, Address translation stages 1 and 2, Non-secure state only* on page G5-6044
ATS1HR, ATS1HW	Not applicable[b]	See *ATS1H*, Address translation stage 1, Hyp mode* on page G5-6045

a. Instructions are added by ARMv8.2-ATS1E1 and do not have a previous name.
b. Instructions are part of EL2 and have no equivalent in the older descriptions.

In an implementation that does not include EL2, there is no distinction between stage 1 translations and stage 1 and 2 combined translations.

For the *stage 1 current state* and *stages 1 and 2 Non-secure state only* instructions, the meanings of the final letters of the names are:

PR PL1 mode, read operation.
PRP PL1 mode, read operation, taking account of PSTATE.PAN.
PW PL1 mode, write operation.
PWP PL1 mode, write operation, taking account of PSTATE.PAN.
UR User mode, read operation.
UW User mode, write operation.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

User mode can be described as the unprivileged mode. It is the only EL0 mode.

For the *stage 1 Hyp mode* instructions, the last letter of the instruction name is **R** for the read operation and **W** for the write operation.

See also *Encoding and availability of the address translation instructions* on page G5-6045.

ATS1C**, Address translation stage 1, current security state

Any VMSAv8-32 implementation supports the ATS1C** instructions. They can be executed by any software executing at PL1 or higher, in either Security state.

The ATS1C** instructions are ATS1CPR, ATS1CPW, ATS1CUR, and ATS1CUW and, when ARMv8.2-ATS1E1 is implemented, ATS1CPRP and ATS1CPWP. These instructions perform the address translations of the PL1&0 translation regime.

In an implementation that includes EL2, when executed in Non-secure state, these instructions return the IPA that is the output address of the stage 1 translation. Figure G5-1 on page G5-5924 shows the different translation regimes.

Note

The Non-secure PL1 and EL0 modes have no visibility of the stage 2 address translations, that can be defined only at EL2, and translate IPAs to be PAs.

See *Determining the PAR format* on page G5-6047 for the format used when returning the result of these instructions.

ATS12NSO**, Address translation stages 1 and 2, Non-secure state only

A VMSAv8-32 implementation supports the ATS12NSO** instructions only if it includes EL2. In an implementation that includes EL2, in AArch32 state, they can be executed:

- By software executing in Non-secure state at EL2. This means by software executing in Hyp mode.
- If the implementation includes EL3, when EL3 is using AArch32, by software executing in Secure state at PL1.

The ATS12NSO** instructions are ATS12NSOPR, ATS12NSOPW, ATS12NSOUR, and ATS12NSOUW.

In an implementation that includes EL3, when EL3 is using AArch64 and EL1 is using AArch32, any execution of an ATS12NSO** instruction at Secure EL1 is trapped as an exception that is taken to EL3.

In an implementation that does not include EL2, but includes EL3, when EL3 is using AArch32 these instructions are not UNDEFINED but each instruction behaves in the same way as the equivalent ATS1C** instruction.

If an implementation does not include EL2 and does not include EL3 then these instructions are CONSTRAINED UNPREDICTABLE, with the permitted behavior that the instructions are UNDEFINED, see *Unallocated System register access instructions* on page K1-7834.

Arm deprecates use of these instructions from any Secure PL1 mode other than Monitor mode.

In Secure state and in Non-secure Hyp mode these instructions perform the translations made by the Non-secure PL1&0 translation regime.

These instructions always return the PA and final attributes generated by the translation. That is, for an implementation that includes EL2, they return:

- The result of the two stages of address translation for the specified Non-secure input address.
- The memory attributes obtained by the combination of the stage 1 and stage 2 attributes.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

From Hyp mode, the ATS1C** and ATS12NSO** instructions both return the results of address translations that would be performed in the Non-secure modes other than Hyp mode. The difference is:

- The ATS1C** instructions return the Non-secure PL1 view of the associated address translation. That is, they return the IPA output address corresponding to the VA input address.
- The ATS12NSO** instructions return the EL2, or Hyp mode, view of the associated address translation. That is, they return the PA output address corresponding to the VA input address, generated by two stages of translation.

See *Determining the PAR format* on page G5-6047 for the format used when returning the result of these instructions.

ATS1H*, Address translation stage 1, Hyp mode

A VMSAv8-32 implementation supports the ATS1H* instructions only if it includes EL2. They can be executed by:

- Software executing in Non-secure state at EL2. This means by software executing in Hyp mode.
- Software executing in Secure state in Monitor mode.

The ATS1H* instructions are ATS1HR and ATS1HW. In an implementation that includes EL3, these instructions are CONSTRAINED UNPREDICTABLE if executed in a Secure PL1 mode other than Monitor mode, see *Hyp mode VA to PA address translation instructions* on page K1-7852.

If an implementation does not include EL2 then these instructions are CONSTRAINED UNPREDICTABLE, with the permitted behavior that the instructions are UNDEFINED, see *Unallocated System register access instructions* on page K1-7834.

These instructions perform the translations made by the Non-secure EL2 translation regime. The instruction takes a VA input address and returns a PA output address.

These instructions always return a result in a 64-bit format PAR.

G5.13.2 Encoding and availability of the address translation instructions

Software executing at EL0 never has any visibility of the address translation instructions, but software executing at PL1 or higher can use the unprivileged address translation instructions to find the address translations used for memory accesses by software executing at PL1 and EL0.

Note

For information about translations when the stage of address translation is disabled see *The effects of disabling address translation stages on VMSAv8-32 behavior* on page G5-5930.

Table G5-35 shows the encodings for the address translation instructions, and their availability in different implementations in different PE modes and states.

Table G5-35 Address translation instructions in AArch32 state

opc1	CRm	opc2	Name	Type	Description
All VMSAv8-32 implementations, in all modes, at PL1 or higher, see *ATS1C**, Address translation stage 1, current security state* on page G5-6044					

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G5-35 Address translation instructions in AArch32 state (continued)

opc1	CRm	opc2	Name	Type	Description
0	c8	0	ATS1CPR	WO	PL1 stage 1 read translation, current state
		1	ATS1CPW	WO	PL1 stage 1 write translation, current state
		2	ATS1CUR	WO	Unprivileged stage 1 read translation, current state
		3	ATS1CUW	WO	Unprivileged stage 1 write translation, current state
	c9	0	ATS1CPRP[a]	WO	PL1 stage 1 read translation, current state, PSTATE.PAN[a]
		1	ATS1CPWP[a]	WO	PL1 stage 1 write translation, current state, PSTATE.PAN[a]
Implementation includes EL2, in Non-secure Hyp mode and Secure PL1 modes, see *ATS12NSO**, Address translation stages 1 and 2, Non-secure state only* on page G5-6044					
0	c8	4	ATS12NSOPR	WO	Non-secure PL1 stage 1 and 2 read translation
		5	ATS12NSOPW	WO	Non-secure PL1 stage 1 and 2 write translation
		6	ATS12NSOUR	WO	Non-secure unprivileged stage 1 and 2 read translation
		7	ATS12NSOUW	WO	Non-secure unprivileged stage 1 and 2 write translation
Implementation includes EL2, in Non-secure Hyp mode and Secure Monitor mode, see *ATS1H*, Address translation stage 1, Hyp mode* on page G5-6045					
4	c8	0	ATS1HR	WO	Hyp mode stage 1 read translation
		1	ATS1HW	WO	Hyp mode stage 1 write translation

a. Instruction only supported when ARMv8.2-ATS1E1 is implemented.

The result of an instruction is always returned in the PAR. The PAR is a RW register and:

- In all implementations, the 32-bit format PAR is accessed using an `MCR` or `MRC` instruction with CRn set to c7, CRm set to c4, and opc1 and opc2 both set to 0.
- The 64-bit format PAR is accessed using an `MCRR` or `MRRC` instruction with CRm set to c7, and opc1 set to 0.

Address translation instructions that are not available in a particular implementation are reserved and CONSTRAINED UNPREDICTABLE. For example:

- In an implementation that does not include EL2, the encodings with an `opc1` value of 4 are reserved and CONSTRAINED UNPREDICTABLE. These are the ATS1H* instructions.
- In an implementation that does not include either EL2 or EL3, the encodings with `opc2` values of 4-7 are reserved and CONSTRAINED UNPREDICTABLE. These are the ATS12NSO** instructions.

The CONSTRAINED UNPREDICTABLE behavior of these encodings is that they are UNDEFINED, see *Unallocated System register access instructions* on page K1-7834.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G5.13.3 Determining the PAR format

The PAR is a 64-bit register, that supports both 32-bit and 64-bit PAR formats. This section describes how the PAR format is determined, for returning a result from each of the groups of address translation instructions. The returned result might be the translated address, or might indicate a fault on the translation, see *Handling of faults and aborts during an address translation instruction*.

ATS1C instructions**

Address translations for the current state. From modes other than Hyp mode:

- TTBCR.EAE determines whether the result is returned using the 32-bit or the 64-bit PAR format.
- If the implementation includes EL3, the translation performed is for the current security state and, depending on that state:
  - The Secure or Non-secure TTBCR.EAE determines the PAR format.
  - The result is returned to the Secure or Non-secure instance of the PAR

Instructions executed in Hyp mode always return a result to the Non-secure PAR, using the 64-bit format.

ATS12NSO instructions**

Address translations for the Non-secure PL1 and EL0 modes. These instructions return a result using the 64-bit PAR format if at least one of the following is true:

- The Non-secure TTBCR.EAE bit is set to 1.
- The implementation includes EL2, and the value of HCR.VM is 1.

Otherwise, the instruction returns a result using the 32-bit PAR format.

Instructions executed in a Secure PL1 mode return a result to the Secure PAR. Instructions executed in Hyp mode return a result to the Non-secure PAR.

ATS1H* instructions

Address translations from Hyp mode. These instructions always return a result using the 64-bit PAR format.

Instructions executed in Secure Monitor mode return a result to the Secure PAR. Instructions executed in Non-secure Hyp mode return a result to the Non-secure PAR.

G5.13.4 Handling of faults and aborts during an address translation instruction

When a stage of address translation is enabled, any corresponding address translation instruction requires a translation table lookup, and this might require a translation table walk. However, the input address for the translation might be a faulting address, either because:

- The translation table entries used for the translation indicate a fault.
- A stage 2 fault or an External abort occurs on the required translation table walk.

VMSAv8-32 memory aborts on page G5-6012 describes the faults that might occur on a translation table walk in AArch32 state.

How the fault is handled, and whether it generates an exception, depends on the cause of the fault, as described in:

- *MMU fault on an address translation instruction*.
- *External abort during an address translation instruction* on page G5-6048.
- *Stage 2 fault on a current state address translation instruction* on page G5-6048.

MMU fault on an address translation instruction

In the following cases, an MMU fault on an address translation is reported in the PAR, and no abort is taken. This applies:

- For a faulting address translation instruction executed in Hyp mode, or in a Secure PL1 mode.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- For a faulting address translation instruction executed in a Non-secure PL1 mode, for cases where the fault would generate a stage 1 abort if it occurred on the equivalent load or store operation.

Using the PAR to report a fault on an address translation instruction gives more information about how these faults are reported.

Note

- The Domain fault encodings shown in Table G5-27 on page G5-6031 are used only for reporting a fault on an address translation instruction that uses the 64-bit PAR format. That is, they are used only in an implementation that includes EL2, and are used for reporting a Domain fault on either:
  - An ATS1C** instruction executed in Hyp mode.
  - An ATS12NSO** instruction executed when the value of HCR.VM is 1.

  These encodings are never used for fault reporting in the DFSR, IFSR, or HSR.
- For an address translation instruction executed in a Non-secure PL1 mode, for a fault that would generate a stage 2 abort if it occurred on the equivalent load or store operation, the stage 2 abort is generated as described in *Stage 2 fault on a current state address translation instruction*.

Using the PAR to report a fault on an address translation instruction

For a fault on an address translation instruction for which no abort is taken, the PAR is updated with the following information, to indicate the fault:

- The fault code, that would normally be written to the Fault status register. The code used depends on the current translation table format, as described in either:
  - *PL1 fault reporting with the Short-descriptor translation table format* on page G5-6029.
  - *PL1 fault reporting with the Long-descriptor translation table format* on page G5-6031.

  See also the Note at the start of *Determining the PAR format* on page G5-6047 about the Domain fault encodings shown in Table G5-27 on page G5-6031.
- A status bit, that indicates that the translation operation failed.

The fault does not update any Fault Address Register.

External abort during an address translation instruction

As stated in *External abort on a translation table walk* on page G5-6020, an External abort on a translation table walk generates a Data Abort exception. The abort can be synchronous or asynchronous, and behaves as follows:

Synchronous External abort on a translation table walk

The fault status and fault address registers of the Security state to which the abort is taken are updated. The fault status register indicates the appropriate External abort on a Translation fault, and the fault address register indicates the input address for the translation.

The PAR is UNKNOWN.

Asynchronous External abort on a translation table walk

The fault status register of the Security state to which the abort is taken is updated, to indicate the asynchronous External abort. No fault address registers are updated.

The PAR is UNKNOWN.

Stage 2 fault on a current state address translation instruction

If the PE is in a Non-secure PL1 mode and executes one of the ATS1C** instructions, then a fault in the stage 2 translation of an address accessed in a stage 1 translation table lookup generates an exception. This is equivalent to the case described in *Stage 2 fault on a stage 1 translation table walk* on page G5-6019. When this fault occurs on an ATS1C** address translation instruction:

- A Hyp Trap exception is taken to Hyp mode.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The PAR is UNKNOWN.
- The HSR indicates that:
  - The fault occurred on a translation table walk.
  - The operation that faulted was a cache maintenance instruction.
- The HPFAR holds the IPA that faulted.
- The HDFAR holds the VA that the executing software supplied to the address translation instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G5.14 Pseudocode description of VMSAv8-32 memory system operations

This section contains a list of pseudocode functions describing VMSAv8-32 memory operations. The following subsections describe the pseudocode functions:

- *Alignment fault*.
- *Address translation*.
- *Domain checking*.
- *TLB operations*.
- *Translation table walk*.
- *Reporting syndrome information* on page G5-6051.
- *Memory access decode when TEX remap is enabled* on page G5-6051.

See also the descriptions of pseudocode for general memory system operations in *Pseudocode description of general memory System instructions* on page G4-5917.

G5.14.1 Alignment fault

The `AArch32.AlignmentFault()` pseudocode function describes the generation of an Alignment fault Data Abort exception.

See also *Abort exceptions* on page G4-5919.

G5.14.2 Address translation

The `AArch32.TranslateAddress()` and `AArch32.FullTranslate()` pseudocode functions describe a VMSAv8-32 address translation.

The `AArch32.FullTranslate()` function calls either:

- The function described in *Address translation when the stage 1 address translation is disabled*.
- One of the functions described in *Translation table walk*.

See also *Stage 2 translation table walk* on page G5-6051.

Address translation when the stage 1 address translation is disabled

The `AArch32.TranslateAddressS1Off()` pseudocode function describes the address translation performed when the stage 1 address translation is disabled.

G5.14.3 Domain checking

The `AArch32.CheckDomain()` pseudocode function describes domain checking.

G5.14.4 TLB operations

The `TLBRecord` type represents the contents of a TLB entry:

G5.14.5 Translation table walk

Because of the complexity of a translation table walk, the following sections describe the different cases:

- *Translation table walk using the Short-descriptor translation table format for stage 1*.
- *Translation table walk using the Long-descriptor translation table format for stage 1* on page G5-6051.
- *Stage 2 translation table walk* on page G5-6051.

Translation table walk using the Short-descriptor translation table format for stage 1

The `AArch32.TranslationTableWalkSD()` pseudocode function describes the translation table walk when the stage 1 translation tables use the Short-descriptor format. It calls the function described in *Stage 2 translation table walk* on page G5-6051 if necessary.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The `ShortConvertAttrsHints()` pseudocode function converts the Normal memory cacheability attribute, from the TTBR or the translation table TEX field, into the separate cacheability attribute and cache allocation hint defined in a Long-descriptor translation table descriptor.

Translation table walk using the Long-descriptor translation table format for stage 1

The `AArch32.TranslationTableWalkLD()` pseudocode function describes the translation table walk when the stage 1 translation tables use the Long-descriptor format. It calls the function described in *Stage 2 translation table walk* if necessary.

`AArch32.TranslationTableWalkLD()` calls the `ConvertAttrsHints()` pseudocode function that is defined in *Translation table walk using the Short-descriptor translation table format for stage 1* on page G5-6050.

The `AArch32.S1AttrDecode()` pseudocode function uses the MAIR0 and MAIR1 registers to decode the Attr[2:0] value from a stage 1 translation table descriptor.

The `S2AttrDecode()` pseudocode function decodes the Attr[3:0] value from a stage 2 translation table descriptor.

Stage 2 translation table walk

In the Non-secure EL1&0 translation regime, a descriptor address returned by stage 1 lookup is in the IPA address map, and must be mapped to a PA by a stage 2 translation. When EL2 is using AArch32, function `AArch32.SecondStageWalk()` performs this translation, by calling the `AArch32.SecondStageTranslate()` function. When called from `AArch32.SecondStageWalk()`, the `AArch32.SecondStageTranslate()` function performs a second stage translation, from IPA to PA, of the supplied address, including checking that the access has read permission at the second stage. If the access does not have second stage read permission it generates a second stage Permission fault on the first stage translation table walk. The second stage translation might hit in a TLB, or might involve a translation table walk, which will use the algorithm described in this section. Stage 2 translations tables always use the Long-descriptor translation table format.

The `AArch32.CheckPermission()` pseudocode function checks the access permissions for the stage 1 translation.

The `AArch32.CheckS2Permission()` pseudocode function checks the access permissions for the stage 2 translation.

The `AArch32.CombineS1S2Desc()` pseudocode function combines the stage 1 and stage 2 access descriptors:

G5.14.6 Reporting syndrome information

The `AArch32.ReportHypEntry()`, `AArch32.ReportDataAbort()`, and `AArch32.ReportPrefetchAbort()` pseudocode functions write syndrome value information to the appropriate registers for the current mode.

G5.14.7 Memory access decode when TEX remap is enabled

When using the Short-descriptor translation table format, the function `AArch32.RemappedTEXDecode()` decodes the `texcb` and `S` attributes derived from the translation tables when TEX remap is enabled. *Short-descriptor format memory region attributes, with TEX remap* on page G5-5982 shows the interpretation of the arguments.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G5.15 About the System registers for VMSAv8-32

The System registers and System instructions that are accessible in AArch32 state are almost all in the encoding space described in *The AArch32 System register interface* on page G1-5769. This section gives general information about these registers, which comprise:

- Registers in the (`coproc==0b1111`) encoding space, that provide control and status information for the PE in Non-debug state.
- Registers in the (`coproc==0b1110`) encoding space, including:
  - Debug registers.
  - Trace registers.
  - Legacy execution environment registers.

VMSAv8-32 organization of registers in the (coproc==0b1110) encoding space on page G7-6075 summarizes the registers in the (`coproc==0b1110`) encoding space, and indicates where these registers are described, either in this manual or in other architecture specifications.

VMSAv8-32 organization of registers in the (coproc==0b1111) encoding space on page G7-6078 summarizes the registers in the (`coproc==0b1111`) encoding space, and indicates where in this manual these registers are described.

Note

Many implementations include other interfaces to some System registers, for example a memory-mapped interface to some debug System registers. These are described in the appropriate sections of this manual.

G5.15.1 Classification of System registers

Features provided by EL3 and EL2 integrate with many features of the architecture. Therefore, the descriptions of the individual System registers include information about how these Exception levels affect the register. This section:

- Summarizes how EL3 and EL2 affect the implementation of the System registers, and the classification of those registers.
- Summarizes how EL3 controls access to the System registers.
- Describes an EL3 signal that can control access to some registers in the (`coproc==0b1111`) encoding space.

It contains the following subsections:

- *Banked System registers*.
- *Restricted access System registers* on page G5-6053.
- *Configurable access System registers* on page G5-6053.
- *EL2-mode System registers* on page G5-6054.
- *Common System registers* on page G5-6055.
- *Access to registers from Monitor mode* on page G5-6055.
- *The CP15SDISABLE and CP15SDISABLE2 input signals* on page G5-6056.

Note

EL3 defines the register classifications of Banked, Restricted access, Configurable, and Common. EL2 defines the EL2-mode classification.

It is IMPLEMENTATION DEFINED whether each IMPLEMENTATION DEFINED register is Banked, Restricted access, Configurable, EL2-mode, or Common.

Banked System registers

In an implementation that includes EL3 using AArch32, some System registers are banked. Banked System registers have two copies, one Secure and one Non-secure. The SCR.NS bit selects the Secure or Non-secure instance of the register.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A Banked System register can contain a mixture of:

- Fields that are banked.
- Fields that are read-only in Non-secure PL1 or EL2 modes but read/write in the Secure state.

The System Control Register SCTLR is an example of a register of that contains this mixture of fields.

The Secure copies of the Banked System registers are sometimes referred to as the Secure Banked System registers. The Non-secure copies of the Banked System registers are sometimes referred to as the Non-secure Banked System registers.

Restricted access System registers

In an implementation that includes EL3, some System registers are present only in Secure state. These are called *Restricted access* registers, and their read/write access permissions are:

- In Non-secure state, software cannot modify Restricted access registers.
- For the NSACR, in Non-secure state:
  - Software running at PL1 or higher can read the register.
  - Unprivileged software, meaning software running at EL0, cannot read the register.

  This means that Non-secure software running at PL1 or higher can read the access permissions for System registers that have Configurable access.

  If EL3 is using AArch64, then any read of the NSACR from Non-secure EL2 using AArch32, or Non-secure EL1 using AArch32, returns the value `0x00000C00`.
- For all other Restricted access registers, Non-secure software cannot read the register.

In an implementation that does not include EL3:

- SDER is implemented only in Secure state.
- Any read of the NSACR returns the value `0x00000C00`.
- All other accesses to Restricted access System registers are UNDEFINED.

Configurable access System registers

Secure software can configure the access to some System registers. These registers are called Configurable access registers, and the control can be:

- A bit in the control register determines whether the register is:
  - Accessible from Secure state only.
  - Accessible from both Secure and Non-secure states.
- A bit in the control register changes the accessibility of a register bit or field. For example, setting a bit in the control register might mean that an RW field behaves as RAZ/WI when accessed from Non-secure state.

Bits in the NSACR control access.

In an AArch32 implementation that includes EL3:

- There are no Configurable access System registers in the (`coproc==0b1110`) encoding space.
- The only required Configurable access register in the (`coproc==0b1111`) encoding space is the CPACR.

— Floating-point Status and Control Register, FPSCR
  - Floating-point Exception register, FPEXC.
  - Floating-point System ID register, FPSID.
  - Media and VFP Feature Register 0, MVFR0.
  - Media and VFP Feature Register 1, MVFR1.
  - Media and VFP Feature Register 2, MVFR2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

EL2-mode System registers

In an implementation that includes EL2, if EL2 can use AArch32, the implementation provides a number of registers for use in the EL2 mode, Hyp mode. As with other System register encodings, some of these register encodings provide write-only operations. When the implementation includes EL3 and EL3 is using AArch32, these registers are also accessible from Monitor mode when the value of SCR.NS is 1.

The following subsections describe the EL2-mode registers:

- *Hyp mode read/write registers in the (coproc==0b1111) encoding space*.
- *Hyp mode encodings for shared (coproc==0b1111) System registers*.
- *Hyp mode (coproc==0b1111) write-only System instructions* on page G5-6055.

There are no EL2-mode registers in the (`coproc==0b1110`) encoding space.

Hyp mode read/write registers in the (`coproc==0b1111`) encoding space

These registers are implemented only in Non-secure state, and in Non-secure state they are accessible only from Hyp mode.

Except for accesses to CNTVOFF in an implementation that includes EL3 but not EL2, the behavior of accesses to these registers is as follows:

- In Secure state, the registers can be accessed from EL3 when SCR.NS is set to 1, see *Access to registers from Monitor mode* on page G5-6055.
- The following accesses are UNDEFINED:
  - Accesses from Non-secure PL1 modes.
  - Accesses in Secure state when SCR.NS is set to 0.

In an implementation that includes EL3 but not EL2, the behavior of accesses to CNTVOFF is as follows:

- Any access from Secure Monitor mode is CONSTRAINED UNPREDICTABLE, regardless of the value of SCR.NS. The CONSTRAINED UNPREDICTABLE behavior is that the access is UNDEFINED, see *Unallocated System register access instructions* on page K1-7834.
- All other accesses are UNDEFINED.

Note

Except for CNTVOFF, the Hyp mode registers are part of EL2, meaning they are implemented only if the implementation includes EL2. However, conceptually, CNTVOFF is part of any implementation of the Generic Timer, see *The virtual offset register* on page G6-6067. This means the behavior of CNTVOFF in an implementation that does not include EL2 is not covered by the general definition of the behavior of the Hyp mode (`coproc==0b1111`) read/write registers.

Hyp mode encodings for shared (`coproc==0b1111`) System registers

Some Hyp mode registers share the Secure instance of an existing banked register. In this case, the implementation includes an encoding for the register that is accessible only in Hyp mode, or in Monitor mode when SCR.NS is set to 1.

For these registers, the following accesses are UNDEFINED:

- Accesses from Non-secure PL1 modes.
- Accesses in Secure state when SCR.NS is set to 0.

In Monitor mode, the Secure copies of these registers can be accessed either:

- Using the DFAR or IFAR encoding with SCR.NS set to 0.
- Using the HDFAR or HIFAR encoding with SCR.NS set to 1.

However, between accessing a register using one alias and accessing the register using the other alias, a *Context synchronization event* is required to ensure the ordering of the accesses.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Hyp mode (`coproc`==`0b1111`) write-only System instructions

Architecturally, these encodings are an extension of the banked register encodings described in *Banked System registers* on page G5-6052, where:

- The implementation does not implement the operation in Secure state.
- In Non-secure state, the operation is accessible only at EL2, that is, only from Hyp mode.

In Secure state:

- These instructions can be accessed from Monitor mode regardless of the value of SCR.NS, see *Access to registers from Monitor mode*.
- Accesses to these instructions are CONSTRAINED UNPREDICTABLE if executed in a Secure mode other than Monitor mode, see *Hyp mode TLB maintenance instructions* on page K1-7852 and *Hyp mode VA to PA address translation instructions* on page K1-7852.

Accesses to these instructions are UNDEFINED if accessed from a Non-secure PL1 mode.

Common System registers

Some System registers and operations are common to the Secure and Non-secure Security states. These are described as the *Common access* registers, or simply as the *Common* registers. These registers include:

- Read-only registers that hold configuration information.
- Register encodings used for various memory system operations, rather than to access registers.
- The ISR.
- All System registers in the (`coproc==0b1110`) encoding space.

Secure System registers for the (`coproc`==`0b1111`) encoding space

The Secure System registers in the (`coproc==0b1111`) encoding space comprise:

- The Secure copies of the Banked System registers in the (`coproc==0b1111`) encoding space.
- The Restricted access System registers in the (`coproc==0b1111`) encoding space.
- The Configurable access System registers in the (`coproc==0b1111`) encoding space that are configured to be accessible only from Secure state.

In an implementation that includes EL3, the Non-secure System registers are the System registers other than the Secure System registers.

Access to registers from Monitor mode

When the PE is in Monitor mode, the PE is in Secure state regardless of the value of the SCR.NS bit. In Monitor mode, the SCR.NS bit determines whether, for System registers in the (`coproc==0b1111`) encoding space, valid uses of the `MRC`, `MCR`, `MRRC`, and `MCRR` instructions access the Secure Banked System registers or the Non-secure Banked System registers. That is, when:

NS == 0 Common, Restricted access, and Secure Banked System registers are accessed by `MRC`, `MCR`, `MRRC`, and `MCRR` instructions that target the (`coproc==0b1111`) encoding space.

If the implementation includes EL2, the registers listed in *Hyp mode read/write registers in the (coproc==0b1111) encoding space* on page G5-6054 and *Hyp mode encodings for shared (coproc==0b1111) System registers* on page G5-6054 are not accessible, and any attempt to access them generates an Undefined Instruction exception.

Note

The operations listed in *Hyp mode (coproc==0b1111) write-only System instructions* are accessible in Monitor mode regardless of the value of SCR.NS.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

System instructions in the (`coproc==0b1111`) encoding space use the Security state to determine all resources used, that is, all operations performed by these instructions are performed in Secure state.

NS == 1 Common, Restricted access and Non-secure Banked System registers are accessed by `MRC`, `MCR`, `MRRC`, and `MCRR` instructions that target the (`coproc==0b1111`) encoding space.

If the implementation includes EL2, all the registers and operations listed in the subsections of *EL2-mode System registers* on page G5-6054 are accessible, using the `MRC`, `MCR`, `MRRC`, or `MCRR` instructions required to access them from Hyp mode.

System instructions in the (`coproc==0b1111`) encoding space use the Security state to determine all resources used, that is, all operations by these instructions are performed in Secure state.

The Security state determines whether the Secure or Non-secure banked registers determine the control state.

Note

Where the contents of a register select the value accessed by an `MRC` or `MCR` access to a different register, then the register that is used for selection is being used as control state. For example, CSSELR selects the current Cache Size Identification Register, and therefore CSSELR is used as control state. Therefore, in Monitor mode:

- SCR.NS determines whether the Secure or Non-secure CSSELR is accessible.
- Because the PE is in Secure state, the Secure CSSELR selects the current Cache Size Identification Register.

From Armv8.3, it is possible to have multiple Cache Size Identification Registers. For more details, see *Possible formats of the Cache Size Identification Registers, CCSIDR and CCSIDR2* on page G4-5891.

The CP15SDISABLE and CP15SDISABLE2 input signals

When EL3 is using AArch32, it provides an input signal, **CP15SDISABLE**, that disables write access to some of the Secure registers when asserted HIGH. The **CP15SDISABLE** signal has no effect on:

- Register accesses from AArch64 state.
- Register accesses from Secure EL1 when EL3 is using AArch64 and EL1 is using AArch32.

Note

When EL3 is using AArch32, the interaction between **CP15SDISABLE** and any IMPLEMENTATION DEFINED register is IMPLEMENTATION DEFINED.

On a reset by the external system that resets the PE into EL3 using AArch32, the **CP15SDISABLE** input signal must be taken LOW. This permits the Reset code to set up the configuration of EL3 features. When the input is asserted HIGH, any attempt to write to the Secure registers that are affected by **CP15SDISABLE** results in an Undefined Instruction exception.

The **CP15SDISABLE** input does not affect reading Secure registers, or reading or writing Non-secure registers. It is IMPLEMENTATION DEFINED how the input is changed and when changes to this input are reflected in the PE, and an implementation might not provide any mechanism for driving the **CP15SDISABLE** input HIGH. However, in an implementation in which the **CP15SDISABLE** input can be driven HIGH, changes in the state of **CP15SDISABLE** must be reflected as quickly as possible. Any change must occur before completion of an Instruction Synchronization Barrier operation, issued after the change, is visible to the PE with respect to instruction execution boundaries. Software must perform an Instruction Synchronization Barrier operation meeting the above conditions to ensure all subsequent instructions are affected by the change to **CP15SDISABLE**.

When EL3 is using AArch32, use of **CP15SDISABLE** means key Secure features that are accessible only at PL1 can be locked in a known state. This provides an additional level of overall system security. Arm expects control of **CP15SDISABLE** to reside in the system, in a block dedicated to security.

When ARMv8.0-CP15SDISABLE2 is implemented and EL3 is using AArch32, EL3 provides a second input signal, **CP15SDISABLE2**. **CP15SDISABLE2** has all of the properties of **CP15SDISABLE** described above. The difference between **CP15SDISABLE** and **CP15SDISABLE2** is only in the set of registers each signal affects.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Information on whether a given register is affected by **CP15SDISABLE**, or **CP15SDISABLE2** when it is implemented, can be found in the access pseudocode for that register, as described in Chapter G8 *AArch32 System Register Descriptions*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G5.16 Functional grouping of VMSAv8-32 System registers

This section describes how the System registers in an VMSAv8-32 implementation divide into functional groups. The functional groups of AArch32 registers are:

- Special-purpose registers.
- VMSA-specific registers.
- ID registers.
- Performance monitors registers.
- Activity monitors registers.
- Debug registers.
- The Reliability, Availability, and Serviceability Extension (RAS Extension) registers.
- Generic timer registers.
- Cache maintenance System instructions.
- Address translation System instructions.
- TLB maintenance System instructions.
- Base system registers.
- Legacy feature registers and System instructions.

For a list of these functional groups and the registers in each group, see *Functional index of AArch32 registers and System instructions* on page K14-8084.

Chapter G8 *AArch32 System Register Descriptions* describes each of these registers.

Note

- Table G7-3 on page G7-6082 lists all of the VMSAv8-32 System registers in the (`coproc==0b1111`) encoding space, ordered by:
  1. The `CRn` primary register used when using a 32-bit access to the register.

     For 64-bit register accesses using an `MCRR` or `MRRC` instruction, the instruction arguments that identify the target register are {`coproc`, `Rm`, `opc1`} The value of `Rm` determines where these registers appear in Table G7-3 on page G7-6082, so that these registers appear with the 32-bit registers accessed using that value for `CRn`. So, for example, the 64-bit access to TTBR0, that uses (`CRm==c2`), appears with the 32-bit access to TTBR0, that uses (`CRn==c2`).
  2. The `opc1` value used when accessing the register.
  3. For 32-bit registers, the {`CRm`, `opc2`} values used when accessing the register.
- The functional groups defined in this section mainly consist of the VMSAv8-32 System registers, but include some additional System registers.
- Some registers belong to more than one functional group.

For other related information see:

- *The AArch32 System register interface* on page G1-5769 for general information about the access to the AArch32 System registers, including the main register access instructions `MRC` and `MCR`.
- *About the System registers for VMSAv8-32* on page G5-6052.
- *VMSAv8-32 organization of registers in the (coproc==0b1110) encoding space* on page G7-6075.
- *VMSAv8-32 organization of registers in the (coproc==0b1111) encoding space* on page G7-6078.
- *About the AArch32 System registers* on page G8-6096.

The register descriptions in Chapter G8 *AArch32 System Register Descriptions*, assume you are familiar with these functional groups, and use conventions and other information from them without any explanation.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter G6
The Generic Timer in AArch32 state

This chapter describes the implementation of the Arm Generic Timer as an extension to an Armv8 implementation. It includes an overview of the AArch32 System register interface to an Arm Generic Timer.

It contains the following sections:

- *About the Generic Timer in AArch32 state* on page G6-6060.
- *The AArch32 view of the Generic Timer* on page G6-6064.

Chapter D11 *The Generic Timer in AArch64 state* describes the AArch64 view of the Generic Timer, including an additional timer that can be implemented in AArch64 state, and Chapter I2 *System Level Implementation of the Generic Timer* describes the system level implementation of the Generic Timer.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G6.1 About the Generic Timer in AArch32 state

Figure G6-1 shows an example system-on-chip that uses the Generic Timer as a system timer. In this figure:

- This manual defines the architecture of the individual PEs in the multiprocessor blocks.
- The *ARM Generic Interrupt Controller Architecture Specification* defines a possible architecture for the interrupt controllers.
- Generic Timer functionality is distributed across multiple components.

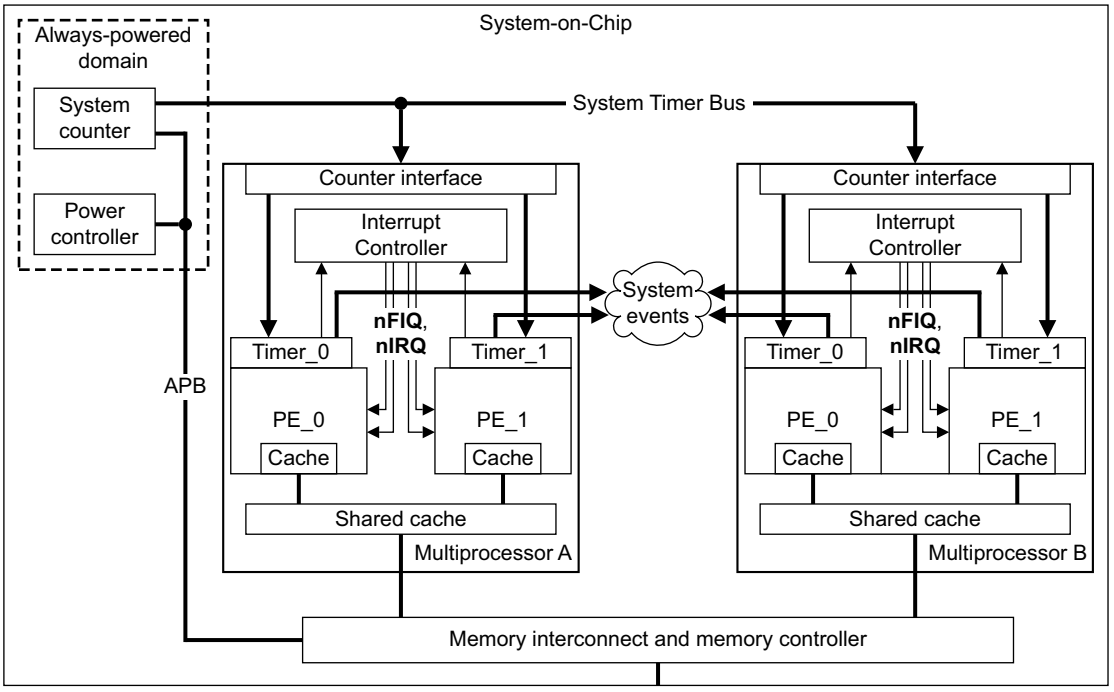


Figure G6-1 Generic Timer example

The Generic Timer:

- Provides a system counter, that measures the passing of time in real-time.

  Note

  The Generic Timer can also provide other components at a system level, but Figure G6-1 does not show any such components.

- Supports *virtual counters* that measure the passing of virtual-time. That is, a virtual counter can measure the passing of time on a particular virtual machine.
- Timers, that can trigger events after a period of time has passed. The timers:
  - Can be used as count-up or as count-down timers.
  - Can operate in real-time or in virtual-time.

This chapter describes an instance of the Generic Timer component that Figure G6-1 shows as Timer_0 or Timer_1 within the Multiprocessor A or Multiprocessor B block. This component can be accessed from AArch64 state or AArch32 state, and this chapter describes access from AArch32 state. Chapter D11 *The Generic Timer in AArch64 state* describes access to this component from AArch64 state.

Note

The reset requirements of Generic Timer registers are more strict when they are accessed from AArch32 state than when they are accessed from AArch64 state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A Generic Timer implementation must also include a memory-mapped system component, see *The full set of Generic Timer components*.

G6.1.1 The full set of Generic Timer components

Within a system that might include multiple PEs, a full set of Generic Timer components is as follows:

The system counter

This provides a uniform view of system time, see *The system counter* on page G6-6062. Because this must be implemented at the system level, it is accessed through *The system level memory-mapped implementation of the Generic Timer*. However, during initialization, a status register in each implemented timer in the system must be programmed with the frequency of the system counter, so that software can read this frequency.

PE implementations of the Generic Timer

Each PE implementation of the Generic Timer provides the following components:

- A physical counter, that gives access to the count value of the system counter.
- A virtual counter, that gives access to virtual time. In AArch32 state, the CNTVOFF register defines the offset between physical time, as defined by the value of the system counter, and virtual time.
- A number of timers. In an implementation where all Exception levels are implemented and can use AArch32 state, the timers that are accessible from AArch64 state are:
  - A Secure PL1 physical timer.
  - A Non-secure EL1 physical timer.
  - An EL2 physical timer.
  - A virtual timer.

  Note

  The Secure PL1 physical timer uses the Secure banked instances of the CNTP_CTL, CNTP_CVAL, and CNTP_TVAL registers, and the Non-secure EL1 physical timer uses the Non-secure instances of the same registers.

The AArch32 view of the Generic Timer on page G6-6064 describes these components.

The system level memory-mapped implementation of the Generic Timer

The memory-mapped registers that control the components of the system level implementation of the Generic Timer are grouped into *frames*. The Generic Timer architecture defines the offset of each register within its frame, but the base address of each frame is IMPLEMENTATION DEFINED, and defined by the system.

Each system level component has one or two register frames. The possible system level components are:

The memory-mapped counter module, required

This module controls the system counter. It has two frames:

- A control frame, CNTControlBase.
- A status frame, CNTReadBase.

The memory-mapped timer control module, required

The system level implementation of the Generic Timer can provide up to eight timers, and the memory-mapped timer control module identifies:

- Which timers are implemented.
- The features of each implemented timer.

This module has a single frame, CNTCTLBase.

Memory-mapped timers, optional

An implemented memory-mapped timer:

- Must provide a privileged view of the timer, in the CNTBase*N* frame.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Optionally. provides an unprivileged view of the timer in the CNTEL0Base*N* frame.

N is the timer number, and the corresponding frame number, in the range 0-7.

Chapter I2 *System Level Implementation of the Generic Timer* describes these components.

G6.1.2 The system counter

The Generic Timer provides a system counter with the following specification:

Width From Armv8.0 to Armv8.5 inclusive, at least 56 bits wide. The value returned by any 64-bit read of the counter is zero-extended to 64 bits.

From Armv8.6, must be 64 bits wide.

Frequency From Armv8.0 to Armv8.5 inclusive, increments at a fixed frequency, typically in the range 1-50MHz. It can support one or more alternative operating modes in which it increments by larger amounts at a lower frequency, typically for power-saving.

From Armv8.6, increments at a fixed frequency of 1GHz.

Roll-over Roll-over time of not less than 40 years.

Accuracy Arm does not specify a required accuracy, but recommends that the counter does not gain or lose more than ten seconds in a 24-hour period.

Use of lower-frequency modes must not affect the implemented accuracy.

Start-up Starts operating from zero.

The system counter, once configured and running, must provide a uniform view of system time. More precisely, it must be impossible for the following sequence of events to show system time going backwards:

1. Device A reads the time from the system counter.
2. Device A communicates with another agent in the system, Device B.
3. After recognizing the communication from Device A, Device B reads the time from the system counter.

The system counter must be implemented in an always-on power domain.

To support lower-power operating modes in architectures from Armv8.0 to Armv8.5, the counter can increment by larger amounts at a lower frequency. For example, a 10MHz system counter might either increment:

- By 1 at 10MHz.
- By 500 at 20kHz, when the system lowers the clock frequency, to reduce power consumption.

In this case, the counter must support transitions between high-frequency, high-precision operation, and lower-frequency, lower-precision operation, without any impact on the required accuracy of the counter.

From Armv8.6 the counter operates at a higher fixed frequency of 1GHz. This implies a resolution of 1ns.

Note

Though each unit of the counter is set to 1ns, this does not require that the counter is incremented every 1ns. A step in the counter might be more than a single bit increment. It is recommended that the count is not incremented at a rate that is less than 50MHz in normal running operation.

The CNTFRQ register is intended to hold a copy of the current clock frequency to allow fast reference to this frequency by software running on the PE. For more information see *Initializing and reading the system counter frequency* on page G6-6063.

The mechanism by which the count from the system counter is distributed to system components is IMPLEMENTATION DEFINED, but each PE with a System register interface to the system counter must have a counter input that can capture each increment of the counter.

Note

So that the system counter can be clocked independently from the PE hardware, the count value might be distributed using a Gray code sequence. *Gray-count scheme for timer distribution scheme* on page K5-7916 gives more information about this possibility.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Initializing and reading the system counter frequency

The CNTFRQ register must be programmed to the clock frequency of the system counter. Typically, this is done only during the system boot process, by using the System register interface to write the system counter frequency to the CNTFRQ register. Only software executing at the highest implemented Exception level can write to CNTFRQ.

Note

The CNTFRQ register is UNKNOWN at reset, and therefore the counter frequency must be set as part of the system boot process.

Software can read the CNTFRQ register, to determine the current system counter frequency, in the following states and modes:

- Hyp mode.
- Secure PL1 modes and Non-secure EL1 modes.
- When CNTKCTL.PL0PCTEN is set to 1, Secure and Non-secure EL0 modes.

Memory-mapped controls of the system counter

Some system counter controls are accessible only through the memory-mapped interface to the system counter. These controls are:

- Enabling and disabling the counter.
- Setting the counter value.
- Changing the operating mode, to change the update frequency and increment value.
- Enabling Halt-on-debug, that a debugger can then use to suspend counting.

For descriptions of these controls, see Chapter I2 *System Level Implementation of the Generic Timer*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G6.2 The AArch32 view of the Generic Timer

The following sections describe the components and features of a PE implementation of the Generic Timer, as seen from AArch32 state:

- *The physical counter*.
- *The virtual counter* on page G6-6065.
- *Event streams* on page G6-6067.
- *Timers* on page G6-6068.

If EL2 is executing in AArch64 state, the function of the AArch32 view of the Generic Timer at EL0 and EL1 are affected by settings in AArch64 state, see *The AArch64 view of the Generic Timer* on page D11-2842.

G6.2.1 The physical counter

The PE includes a physical counter that contains the count value of the system counter. The CNTPCT register holds the current physical counter value.

Accessing the physical counter

Software with sufficient privilege can read CNTPCT using a 64-bit System register read.

CNTPCT:

- Is always accessible from Secure PL1 modes and from Non-secure Hyp mode.
- Is accessible from Non-secure EL1 modes when the value of CNTHCTL.PL1PCTEN is 1. When the value of CNTHCTL.PL1PCTEN is 0, any attempt to access CNTPCT from Non-secure EL1 modes is trapped to Hyp mode.
- Is accessible from Secure User mode when the value of CNTKCTL.PL0PCTEN is 1. When the value of CNTKCTL.PL0PCTEN is 0, any attempt to access CNTPCT generates an UNDEFINED exception.
- Is accessible from Non-secure User mode when the value of CNTHCTL.PL1PCTEN is 1 and the value of CNTKCTL.PL0PCTEN is 1. Otherwise:
  - When the value of CNTKCTL.PL0PCTEN is 0, any attempt to access CNTPCT from Non-secure User mode generates an UNDEFINED exception.
  - When the value of CNTKCTL.PL0PCTEN is 1 and the value of CNTHCTL.PL1PCTEN is 0, any attempt to access CNTPCT from Non-secure User mode is trapped to Hyp mode.

Reads of CNTPCT can occur speculatively and out of order relative to other instructions executed on the same PE.

The self-synchronized view of the physical counter

When ARMv8.6-ECV is implemented, an alternative way to read the physical counter is supported. The CNTPCTSS register is a non-speculative view of the physical counter, as seen from the Exception level that CNTPCTSS is read from.

Accesses to the CNTPCTSS are subject to the same traps as accesses to the CNTPCT.

Reads of CNTPCT occur in program order relative to reads of CNTPCT or CNTPCTSS.

Reads of CNTPCTSS occur in program order relative to reads of CNTPCT or CNTPCTSS.

Example G6-1 Ensuring read of the physical counter occurs after signal read from memory

If a read from memory is used to obtain a signal from another agent that indicates that CNTPCT must be read, an `ISB` is used to ensure that the read of CNTPCT occurs after the signal has been read from memory, as shown in the following code sequence:

```
loop                ; polling for some communication to indicate a requirement to read the timer
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    LDR R1, [R2]
    CMP R1, #1      ; has had the value 1 written to it
    BNE loop
    ISB             ; without this, the CNTPCT could be read before the memory location in [R2]
    MRC R1, CNTPCT
```

When ARMv8.6-ECV is implemented, an access to CNTPCTSS can be used in place of the CNTPCT, which because it cannot be accessed speculatively, allows the ISB to be removed. Therefore the following code sequence can be used:

```
loop                    ; polling for some communication to indicate a requirement to read the timer
    LDR R1, [R2]
    CMP R1, #1          ; has had the value 1 written to it
    BNE loop
    MRC R1, CNTPCTSS
```

Similarly where a read of the physical counter is required to take place after the completion of all loads and stores appearing in program order before the read of the counter, then the following code sequences can be used:

```
    ...             ; earlier loads and stores
    DSB             ; completes earlier loads and stores
    ISB             ; without this the CNTPCT_EL0 could be read before the completion of the earlier
                    ; loads and stores
    MRC R1, CNTPCT
```

Or, if ARMv8.6-ECV is implemented:

```
    ...                 ; earlier loads and stores
    DSB                 ; completes earlier loads and stores
    MRC R1, CNTPCTSS
```

Neither view of the physical counter ensures that:

- Context changes occurring in program order before the read of the counter have been synchronised.
- Accesses to memory appearing in program order after the read of the counter are executed before the counter has been read.

Example G6-2 Ensuring reads of the physical counter occur after previous memory accesses

---

To ensure that all previous memory accesses have completed and all previous context changes have been synchronised before the read of the counter, the following sequence should be used:

```
    DSB
    ISB
    MRC Rn, CNTPCT{SS}      ; either view of the physical counter has the same effect in this example
```

To ensure that a memory access only occurs after a read of the counter, the following sequence should be used:

```
    MRC Rn, CNTPCT{SS}      ; either view of the physical counter has the same effect in this example
    ISB
    LDR Ra, [Xb]            ; this load will be executed after the timer has been read
```

G6.2.2 The virtual counter

An implementation of the Generic Timer always includes a virtual counter, that indicates virtual time.

The virtual counter contains the value of the physical counter minus a 64-bit virtual offset. When executing in a Non-secure EL1 or EL0 mode, the virtual offset value relates to the current virtual machine.

The CNTVOFF register contains the virtual offset. CNTVOFF is only accessible:

- From Hyp mode.
- From Monitor mode only when SCR.NS is set to 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For more information see *The virtual offset register* on page G6-6067.

The CNTVCT register holds the current virtual counter value.

Accessing the virtual counter

Software with sufficient privilege can read CNTVCT using a 64-bit System register read.

CNTVCT is always accessible from Secure PL1 modes and from Non-secure EL1 and EL2 modes.

In addition, when CNTKCTL.PL0VCTEN is set to 1, CNTVCT is accessible from EL0.

When CNTKCTL.PL0VCTEN is set to 0, any attempt to access CNTVCT from EL0 is UNDEFINED.

Reads of CNTVCT can occur speculatively and out of order relative to other instructions executed on the same PE.

The self-synchronized view of the virtual timer

When ARMv8.6-ECV is implemented, an alternative way to read the physical counter is supported. The CNTVCTSS register is a non-speculative view of the physical counter, as seen from the Exception level that CNTVCTSS is read from.

Accesses to the CNTVCTSS are subject to the same traps as accesses to the CNTVCT.

Reads of CNTVCT occur in program order relative to reads of CNTVCT or CNTVCTSS.

Reads of CNTVCTSS occur in program order relative to reads of CNTVCT or CNTVCTSS.

Example G6-3 Ensuring read of the virtual counter occurs after signal read from memory

If a read from memory is used to obtain a signal from another agent that indicates that CNTVCT must be read, an ISB is used to ensure that the read of CNTVCT occurs after the signal has been read from memory, as shown in the following code sequence:

```
loop                ; polling for some communication to indicate a requirement to read the timer
    LDR R1, [R2]
    CMP R1, #1      ; has had the value 1 written to it
    BNE loop
    ISB             ; without this, the CNTVCT could be read before the memory location in [R2]
    MRC R1, CNTVCT
```

When ARMv8.6-ECV is implemented, an access to CNTPCTSS can be used in place of the CNTPCT, which because it cannot be accessed speculatively, allows the ISB to be removed. Therefore the following code sequence can be used:

```
loop                        ; polling for some communication to indicate a requirement to read the timer
    LDR R1, [R2]
    CMP R1, #1              ; has had the value 1 written to it
    BNE loop
    MRC R1, CNTVCTSS
```

Similarly where a read of the physical counter is required to take place after the completion of all loads and stores appearing in program order before the read of the counter, then the following code sequences can be used:

```
    ...             ; earlier loads and stores
    DSB             ; completes earlier loads and stores
    ISB             ; without this the CNTPCT_EL0 could be read before the completion of the earlier
                    ; loads and stores
    MRC R1, CNTVCT
```

Or, if ARMv8.6-ECV is implemented:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
...                 ; earlier loads and stores
DSB                 ; completes earlier loads and stores
MRC R1, CNTVCTSS
```

Neither view of the virtual counter ensures that:

- Context changes occurring in program order before the read of the counter have been synchronised.
- Accesses to memory appearing in program order after the read of the counter are executed before the counter has been read.

Example G6-4 Ensuring reads of the virtual counter occur after previous memory accesses

To ensure that all previous memory accesses have completed and all previous context changes have been synchronised before the read of the counter, the following sequence should be used:

```
DSB
ISB
MRC Rn, CNTVCT{SS}      ; either view of the physical counter has the same effect in this example
```

To ensure that a memory access only occurs after a read of the counter, the following sequence should be used:

```
MRC Rn, CNTVCT{SS}      ; either view of the physical counter has the same effect in this example
ISB
LDR Ra, [Xb]            ; this load will be executed after the timer has been read
```

The virtual offset register

The virtual counter is a counter that has a virtual offset relative to the physical counter as viewed from EL2 and EL3. This virtual offset is held in the register CNTVOFF. The virtual counter value is the count compared by the EL1 virtual timer.

If EL2 is not implemented and enabled, then the virtual counter uses a fixed offset of zero.

G6.2.3 Event streams

Any implementation of the Generic Timer can use the system counter to generate one or more *event streams*, to generate periodic wake-up events as part of the mechanism described in *Wait for Event mechanism and Send event* on page D1-2397.

Note

An event stream might be used:

- To impose a time-out on a Wait For Event polling loop.
- To safeguard against any programming error that means an expected event is not generated.

An event stream is configured by:

- Selecting which bit, from the bottom 16 bits of a counter, triggers the event. This determines the frequency of the events in the stream.
- Selecting whether the event is generated on each 0 to 1 transition, or each 1 to 0 transition, of the selected counter bit.

The CNTKCTL.{EVNTEN, EVNTDIR, EVNTI, EVNTIS} fields define an event stream that is generated from the virtual counter.

In all implementations the CNTHCTL.{EVNTEN, EVNTDIR, EVNTI, EVNTIS} fields define an event stream that is generated from the physical counter.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The operation of an event stream is as follows:

- The pseudocode variables `PreviousCNTVCT` and `PreviousCNTPCT` are initialized as:

```
// Variables used for generation of the timer event stream.
bits(64) PreviousCNTVCT = bits(64) UNKNOWN;
bits(64) PreviousCNTPCT = bits(64) UNKNOWN;
```

- The pseudocode functions `TestEventCNTV()` and `TestEventCNTP()` are called on each cycle of the PE clock.
- The `TestEventCNTx()` pseudocode template defines the functions `TestEventCNTV()` and `TestEventCNTP()`:

```
// TestEventCNTx()
// ===============

// Template for the TestEventCNTV() and TestEventCNTP() functions
// Describes operation when all Exception Levels are using AArch32:
//   CNTxCT         is  CNTVCT          or  CNTPCT          64-bit count value
//   CNTxCTL        is  CNTHCTL         or  CNTKCTL         Control register
//   PreviousCNTxCT is  PreviousCNTVCT  or  PreviousCNTPCT

TestEventCNTx()
    if CNTxCTL.EVNTEN == '1' then
        n = UInt(CNTxCTL.EVNTI);

        if CNTxCTL.EVNTIS == '1' then
            n = n + 8;

        SampleBit   = CNTxCT<n>;
        PreviousBit = PreviousCNTxCT<n>;

        if CNTx_CTL.EVNTDIR == '0' then
            if PreviousBit == '0' && SampleBit == '1' then EventRegisterSet();
        else
            if PreviousBit == '1' && SampleBit == '0' then EventRegisterSet();

    PreviousCNTxCT = CNTxCT;

    return;
```

G6.2.4 Timers

In an implementation that includes EL3, in any implementation of the Generic Timer, the following timers are accessible from AArch32 state, provided the appropriate Exception level can use AArch32:

- A Non-secure EL1 physical timer. A Non-secure EL1 control determines whether this register is accessible from Non-secure EL0.
- A Secure PL1 physical timer. This timer:
  - Is accessible from Secure EL1 using AArch32 when EL3 is using AArch64.
  - Is accessible from Secure EL3 when EL3 is using AArch32.

  A Secure PL1 control determines whether this register is accessible from Secure EL0.
- A Non-secure EL2 physical timer.
- A virtual timer.

The output of each implemented timer:

- Provides an output signal to the system.
- If the PE interfaces to a *Generic Interrupt Controller* (GIC), signals a *Private Peripheral Interrupt* (PPI) to that GIC. In a multiprocessor implementation, each PE must use the same interrupt number for each timer.

Each timer:

- Is based around a 64-bit CompareValue that provides a 64-bit unsigned upcounter.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Provides an alternative view of the CompareValue, called the TimerValue, that appears to operate as a 32-bit downcounter.
- Has, in addition, a 32-bit Control register.

In all implementations, the AArch32 System registers for the EL1 (or PL1) physical timer are banked, to provide the Secure and Non-secure implementations of the timer. Table G6-1 shows the Timer registers.

Table G6-1 Timer registers summary for the Generic Timer

Timer register	Secure PL1 or Non-secure EL1 physical timer	EL2 physical timer	Virtual timer
CompareValue register	CNTP_CVAL[a]	CNTHP_CVAL	CNTV_CVAL
TimerValue register	CNTP_TVAL[a]	CNTHP_TVAL	CNTV_TVAL
Control register	CNTP_CTL[a]	CNTHP_CTL	CNTV_CTL

a. In AArch32 state, these registers are banked to provide the Non-secure EL1 physical timer and the Secure PL1 physical timer.

The following sections describe:

- *Accessing the timer registers*
- *Operation of the CompareValue views of the timers* on page G6-6070
- *Operation of the TimerValue views of the timers* on page G6-6070.

Accessing the timer registers

For each timer, all timer registers have the same access permissions, as follows:

Secure PL1 and Non-secure EL1 physical timer

The Secure PL1 physical timer is accessible from Secure PL1 modes.

Non-secure software executing at EL2 controls access to the Non-secure EL1 physical timer from Non-secure EL1 modes. The Non-secure EL1 physical timer is accessible from Monitor mode when the value of SCR.NS is 1.

When access from PL1 or EL1 modes is permitted, CNTKCTL.PL0PTEN determines whether the registers are accessible from EL0. If an access is not permitted because CNTKCTL.PL0PTEN is set to 0, an attempted access from EL0 is UNDEFINED.

In all implementations:

- Except for accesses from Monitor mode, accesses are to the registers for the current Security state.
- For accesses from Monitor mode, the value of SCR.NS determines whether accesses are to the Secure or the Non-secure registers.

  Note

  Monitor mode is present only when EL3 is using AArch32.

- The Non-secure registers are accessible from Hyp mode.
- CNTHCTL.PL1PCEN determines whether the Non-secure registers are accessible from Non-secure EL1 modes. If this bit is set to 1, to enable access from Non-secure EL1 modes, CNTKCTL.PL0PTEN determines whether the registers are accessible from Non-secure EL0.

  If an access is not permitted because CNTHCTL.PL1PCEN is set to 0, an attempted access from a Non-secure EL1 or EL0 mode generates a Hyp Trap exception. However, if CNTKCTL.PL0PTEN is set to 0, this control takes priority, and an attempted access from EL0 is UNDEFINED.

EL2 physical timer Accessible from Hyp mode, and from Secure Monitor mode when SCR_EL3.NS is set to 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Virtual timer Accessible from Secure PL1 modes and Non-secure EL1 modes, and from Hyp mode.

CNTKCTL.PL0VTEN determines whether the registers are accessible from EL0 modes. If an access is not permitted because CNTKCTL.PL0VTEN is set to 0, an attempted access from EL0 is UNDEFINED.

Operation of the CompareValue views of the timers

The CompareValue view of a timer operates as a 64-bit upcounter. The timer condition is met when the appropriate counter reaches the value programmed into its CompareValue register. When the timer condition is met an interrupt is generated if the interrupt is not masked in the corresponding timer control register, CNTP_CTL, CNTHP_CTL, or CNTV_CTL. For CNTP_CTL, the interrupt is the same as the interrupt asserted by the Non-secure instance of the AArch64 register CNTP_CTL_EL0.

The operation of this view of a timer is:

```
TimerConditionMet = (((Counter[63:0] – Offset[63:0])[63:0] - CompareValue[63:0]) >= 0)
```

Where:

`TimerConditionMet` Is TRUE if the timer condition for this counter is met, and FALSE otherwise.

`Counter` The physical counter value, that can be read from the CNTPCT register.

Note

The virtual counter value, that can be read from the CNTVCT register, is the value:

```
(Counter - Offset)
```

`Offset` For a physical timer, if EL2 is in AArch64 state and ARMv8.6-ECV is implemented, it is the physical offset held in the CNTPOFF_EL2. Otherwise it is zero.

For the virtual timer it is the virtual offset, held in the CNTVOFF register.

`CompareValue` The value of the appropriate CompareValue register, CNTP_CVAL, CNTHP_CVAL, or CNTV_CVAL.

In this view of a timer, `Counter`, `Offset`, and `CompareValue` are all 64-bit unsigned values.

Note

This means that a timer with a `CompareValue` of, or close to, `0xFFFF_FFFF_FFFF_FFFF` might never meet its timer condition. However, there is no practical requirement to use values close to the counter wrap value.

Operation of the TimerValue views of the timers

The TimerValue view of a timer appears to operate as a signed 32-bit downcounter. A TimerValue register is programmed with a count value. This value decrements on each increment of the appropriate counter, and the timer condition is met when the value reaches zero. When the timer condition is met, an interrupt is generated if the interrupt is not masked in the corresponding timer control register, CNTP_CTL, CNTHP_CTL, or CNTV_CTL.

This view of a timer depends on the following behavior of accesses to TimerValue registers:

Reads `TimerValue = (CompareValue – (Counter - Offset))[31:0]`

Writes `CompareValue = ((Counter - Offset)[63:0] + SignExtend(TimerValue))[63:0]`

Where the arguments other than `TimerValue` have the definitions used in *Operation of the CompareValue views of the timers*, and in addition:

`TimerValue` The value of a TimerValue register, CNTP_TVAL, CNTHP_TVAL, or CNTV_TVAL.

In this view of a timer, values are signed, in standard two's complement form.

A read of a TimerValue register after the timer condition has been met indicates the time since the timer condition was met.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

- *Operation of the CompareValue views of the timers* on page G6-6070 gives a strict definition of `TimerConditionMet`. However, provided that the TimerValue is not expected to wrap as a 32-bit signed value when decremented from `0x80000000`, the TimerValue view can be used as giving an effect equivalent to:

  `TimerConditionMet = (TimerValue ≤ 0)`

- Programming TimerValue to a negative number with magnitude greater than (Counter−Offset) can lead to an arithmetic overflow that causes the CompareValue to be an extremely large positive value. This potentially delays meeting the timer condition for an extremely long period of time.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter G7
AArch32 System Register Encoding

This chapter describes the AArch32 System register encoding space. It contains the following sections:

- *The AArch32 System register encoding space* on page G7-6074.
- *VMSAv8-32 organization of registers in the (coproc==0b1110) encoding space* on page G7-6075.
- *VMSAv8-32 organization of registers in the (coproc==0b1111) encoding space* on page G7-6078.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G7.1 The AArch32 System register encoding space

The T32 and A32 instruction sets includes instructions that access the System register encoding space. These instructions provide:

- Access to *System registers*, including the debug registers, that provide system control, and system status information.
- The cache, branch predictor, and TLB maintenance instructions, and address translation instructions.

The AArch32 System register interface on page G1-5769 describes the instructions that provide access to these registers and instructions. Chapter G8 *AArch32 System Register Descriptions* describes these registers and encodings.

When accessing 32-bit registers, or executing these instructions, entries in the encoding space are characterized by the parameter set {`coproc`, `CRn`, `opc1`, `CRm`, `opc2`}. In Armv8, this encoding space is defined only for the `coproc` values `0b1110` and `0b1111`.

Note

- When accessing 64-bit registers entries in the encoding space are characterized by the parameter set {`coproc`, `CRm`, `opc1`}, for the `coproc` values `0b1110` and `0b1111`. A `CRm` value in this parameter set is equivalent to a `CRn` value in the parameter set for accessing 32-bit registers.
- *Background to the System register interface* on page G1-5770 gives more information about this encoding model.

The following describe this encoding space:

- *VMSAv8-32 organization of registers in the (coproc==0b1110) encoding space* on page G7-6075.
- *VMSAv8-32 organization of registers in the (coproc==0b1111) encoding space* on page G7-6078.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G7.2 VMSAv8-32 organization of registers in the (coproc==0b1110) encoding space

The System registers in the (coproc==0b1110) encoding space provide a number of distinct control functions, covering:

- Debug.
- Trace.
- Execution environment control, for identification of the trivial Jazelle implementation.

Because these functions are distinct, the descriptions of these registers are distributed, as follows:

- In this manual, *Debug registers* on page G8-6586 describes the Debug registers.
- The *Embedded Trace Macrocell Architecture Specification* describes the Trace registers.

This section summarizes the allocation of the System registers in the (coproc==0b1110) encoding space between these different functions, and the register encodings in this space that are reserved.

The 32-bit System register encodings are classified by the {opc1, CRn, opc2, CRm} values required to access them using an MCR or an MRC instruction. The 64-bit System register encodings are classified by the {opc1, CRm} values required to access them using an MCRR or an MRRC instruction. For the registers in the (coproc==0b1110) encoding space, the opc1 value determines the primary allocation of these registers, as follows:

opc1==0 Debug registers.

opc1==1 Trace registers.

opc1==7 Jazelle registers. Jazelle registers are implemented as required for a trivial Jazelle implementation.

Other opc1 values
Reserved.

Note

Primary allocation of (coproc==0b1110) register function by opc1 value differs from the allocation of (coproc==0b1111) registers, where primary allocation is by CRn value for 32-bit register accesses, or CRm value for 64-bit register accesses.

For the Debug and Jazelle registers, Table G7-1 on page G7-6076 defines:

- The {opc1, CRn, opc2, CRm} values used for accessing the 32-bit registers using the MRC and MCR instructions.
- The {opc1, CRm} values used for accessing the 64-bit register using the MRRC instruction.

Some Debug registers can also be accessed using the LDC and STC instructions. Table G7-2 on page G7-6077 defines the CRn values used for accessing the registers using these instructions.

Note

The only permitted uses of the LDC and STC instructions are:

- An LDC access to load data from memory to DBGDTRTXint.
- An STC access to store data to memory from DBGDTRRXint.

In the LDC and STC syntax descriptions in this Manual, the required coproc value of p14 and CRn value of c5 are given explicitly.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G7.2.1 Register access instruction arguments, (coproc==0b1110) registers

Table G7-1 shows the MCR, MRC, and MRRC instruction arguments required for accesses to each register that can be visible in the System register interface in the (coproc==0b1110) encoding.

Table G7-1 Mapping of (coproc==0b1110) MCR, MRC, and MRRC instruction arguments to System registers

Name	Width	opc1	CRn	opc2	CRm
DBGDIDR[a]	32-bit	0	c0	0	c0
DBGDSCRint	32-bit				c1
DBGDCCINT	32-bit				c2
DBGDTRRXint	32-bit				c5
DBGDTRTXint	32-bit				c5
-	32-bit				c6
DBGVCR	32-bit				c7
DBGDTRRXext	32-bit			2	c0
DBGDSCRext	32-bit				c2
DBGDTRTXext	32-bit				c3
DBGOSECCR	32-bit				c6
DBGBVR<n>	32-bit			4	c0-15[b]
DBGBCR<n>	32-bit			5	c0-15[b]
DBGWVR<n>	32-bit			6	c0-15[b]
DBGWCR<n>	32-bit			7	c0-15[b]
DBGDRAR	32-bit		c1	0	c0
	64-bit		-	-	c1
DBGBXVR<n>	32-bit		c1	1	c0-15[b]
DBGOSLAR	32-bit			4	c0
DBGOSLSR	32-bit				c1
DBGOSDLR	32-bit				c3
DBGPRCR	32-bit				c4
DBGDSAR	32-bit		c2	0	c0
	64-bit		-	-	c2
-			c4	0-3	c0-15

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G7-1 Mapping of (coproc==0b1110) MCR, MRC, and MRRC instruction arguments to System registers (continued)

Name	Width	opc1	CRn	opc2	CRm
DBGCLAIMSET	32-bit	0	c7	6	c8
DBGCLAIMCLR	32-bit				c9
DBGAUTHSTATUS	32-bit				c14
DBGDEVID2	32-bit			7	c0
DBGDEVID1	32-bit				c1
DBGDEVID	32-bit				c2
-	32-bit	1	c0-c7	0-7	c0-c15
JIDR[c]	32-bit	7	c0	0	c0
JOSCR[c]	32-bit		c1	0	c0
JMCR[c]	32-bit		c2	0	c0
-	32-bit	All other encodings			

a. If EL1 cannot use AArch32, this register is OPTIONAL and deprecated. See the register description for details.

b. Not implemented breakpoint and watchpoint register access instructions are unallocated. If EL2 is not implemented or breakpoint *n* is not context-aware, DBGBXVR<n> is unallocated. CRm encodes <*n*>, the breakpoint or watchpoint number.

c. Legacy register.

Table G7-2 shows the LDC and STC instruction arguments required for accesses to the Debug registers that can be accessed using these instructions.

Table G7-2 Mapping of LDC and STC instruction arguments to System registers

Name	Width	CRn	Instruction	Description
DBGDTRTXint	32-bit	c5	LDC	Debug Data Transfer Register, Transmit, Internal View
DBGDTRRXint	32-bit	c5	STC	Debug Data Transfer Register, Receive, Internal View

Note

In the instruction syntax descriptions for the LDC and STC instructions, the required coproc and CRn values are given explicitly as coproc==p14, CRn==c5.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G7.3 VMSAv8-32 organization of registers in the (`coproc==0b1111`) encoding space

For 32-bit accesses to the System registers in the (`coproc==0b1111`) encoding space, the ordered set of parameters {`CRn`, `opc1`, `CRm`, `opc2`} determine the register order. Within this ordering, the `CRn` value originally provided a functional grouping of these registers. As the number of System registers has increased this ordering has become less appropriate.

This document now:

- Groups the Armv8.0 System registers in the (`coproc==0b1111`) encoding space by functional group, see *Functional index of AArch32 registers and System instructions* on page K14-8084.
- Describes all of the Armv8.0 System registers for VMSAv8-32, in Chapter G8 *AArch32 System Register Descriptions*.
- Gives additional information about the organization of the VMSAv8-32 System registers in the (`coproc==0b1111`) encoding space, in the remainder of this section.

Note

Not all System registers introduced by architectural extensions to Armv8.0 are described in Chapter G8 *AArch32 System Register Descriptions*. For information about the System registers introduced by architectural extensions to Armv8.0, see Chapter A2 *Armv8-A Architecture Extensions*.

This section presents information about the register ordering by {`CRn`, `opc1`, `CRm`, `opc2`}. It contains the following subsections:

- *System register summary for (coproc==0b1111) encodings by CRn value*.
- *Full list of VMSAv8-32 System registers in the (coproc==0b1111) encoding space* on page G7-6081.

Note

The ordered listing of (`coproc==0b1111`) registers by the {`CRn`, `opc1`, `CRm`, `opc2`} encoding of the 32-bit registers is most likely to be useful to those implementing AArch32 state, and to those validating such implementations. However, otherwise, the grouping of registers by function is more logical.

In addition, the indexes in Appendix K14 *Registers Index* include all of the System registers.

G7.3.1 System register summary for (`coproc==0b1111`) encodings by CRn value

Figure G7-1 on page G7-6079 summarizes the grouping of the System registers in the (`coproc==0b1111`) encoding space, for a VMSAv8-32 implementation, by the value of `CRn` used for a 32-bit access to the register.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

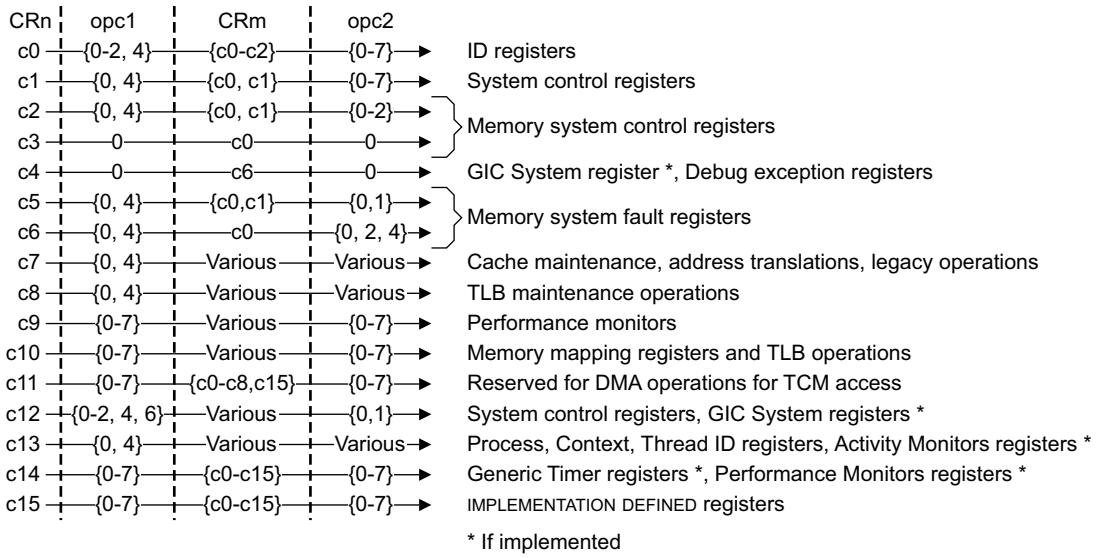


Figure G7-1 AArch32 System register groupings for (`coproc==0b1111`), for 32-bit registers

Note

For the System registers in the (`coproc==0b1111`) encoding space, Figure G7-1 gives only an overview of the assigned encodings for 32-bit registers for each of the `CRn` values `c0-c15`. For more information, see:

- The full list of registers in the (`coproc==0b1111`) encoding space, in *Full list of VMSAv8-32 System registers in the (coproc==0b1111) encoding space* on page G7-6081, for the definition of the assigned and unassigned encodings for that register.
- The register definitions in Chapter G8 *AArch32 System Register Descriptions* for any dependencies on the implemented Exception levels.

In general, System register accesses using an unallocated set of {`CRn`, `opc1`, `CRm`, `opc2`} values are UNDEFINED. *Behavior of VMSAv8-32 32-bit System registers with (coproc==0b1111, CRn==c0)* described the only exceptions to this rule.

The 32-bit System registers with (`coproc==0b1111`, `CRn==c15`), and the corresponding 64-bit System registers, are reserved for implementation defined registers. For more information see *Reserved encodings in the VMSAv8-32 System register (coproc==0b1111) space* on page G7-6080.

The HSTR.Tn trap on (`coproc==0b1111`) System registers

As *General trapping to Hyp mode of Non-secure EL0 and EL1 accesses to System registers in the (coproc==0b1111) encoding space* on page G1-5801 describes, when the value of HSTR.T*n* is 1, Non-secure PL1 accesses to System registers in the (`coproc==0b1111`) encoding space using a `CR`*n* or `CR`*m* value that corresponds to the value of T*n* are trapped to EL2, even if the encoding is UNDEFINED when the value of HSTR.T*n* is 0. This applies:

- For 32 bit register accesses when the value of `Rn` in the `MCR` or `MRC` instruction corresponds to T*n*.
- For 64 bit register accesses when the value of `Rm` in the `MCRR` or `MRRC` instruction corresponds to T*n*.

If there are matching System register encodings that are accessible from Non-secure EL0 then those accesses are also trapped to EL2 when the value of HSTR.T*n* is 1.

Behavior of VMSAv8-32 32-bit System registers with (`coproc==0b1111`, `CRn==c0`)

In the (`coproc==0b1111`) encoding space, the 32-bit System registers with (`CRn==c0`) provide device and feature identification.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G7-3 on page G7-6082 shows all of the architecturally required System registers with {coproc==0b1111, CRn==c0}. The behavior of 32-bit System register encodings in this group that are not shown in the table, and encodings that are part of an unimplemented Exception level, depends on the value of opc1, and possibly on the value of CRm and opc2, as follows:

opc1 == 0 All write accesses to the encodings are UNDEFINED.

For read accesses:

- The following encodings return an UNKNOWN value:
  - CRm==3, opc2=={0, 1, 2}.
  - CRm==={4, 6, 7}, opc2==={0, 1}.
  - CRm==5, opc2==={0, 1, 4, 5}.
- All other encodings are RES0.

opc1 > 0 All accesses to the encodings are UNDEFINED.

See also *Accesses to unallocated encodings in the (coproc==0b111x) encoding space* on page G8-6098.

Note

Some of these registers were previously described as being part of the CPUID identification scheme, see *The CPUID identification scheme* on page G8-6097.

Reserved encodings in the VMSAv8-32 System register (coproc==0b1111) space

AArch32 state reserves a number of regions in the (coproc==0b1111) encoding space for IMPLEMENTATION DEFINED System registers. These reservations are defined in terms of the encoding of 32-bit accesses to the System register encoding space. That is, they are defined by the reserved 32-bit {CRn, opc1, CRm, opc2} encodings.

In Armv8, reserved encodings that do not have an IMPLEMENTATION DEFINED function are UNDEFINED.

The following subsections give more information about these reserved encodings:

- *Reserved 32-bit encodings with {coproc==0b1111, CRn==c9}*.
- *Reserved 32-bit encodings with {coproc==0b1111, CRn==c10}*.
- *Reserved 32-bit encodings with {coproc==0b1111, CRn==c11}* on page G7-6081.
- *Reserved 32-bit encodings with {coproc==0b1111, CRn==c15}* on page G7-6081.

Reserved 32-bit encodings with {coproc==0b1111, CRn==c9}

In the AArch32 encoding space, for 32-bit encodings with {coproc==0b1111, CRn==c9}, the following encodings are reserved for IMPLEMENTATION DEFINED purposes:

- Encodings with {coproc==0b1111, CRn==c9, opc1==={0-7}, opc2==={0-7}, CRm=={c0-c2, c5-c8}} are reserved for IMPLEMENTATION DEFINED branch predictor, cache, and TCM operations.
- Encodings with {coproc==0b1111, CRn==c9, opc1==={0-7}, opc2==={0-7}, CRm==c15} are reserved for IMPLEMENTATION DEFINED performance monitors.

  Note

  These are distinct from the OPTIONAL Arm Performance Monitors Extension, the registers for which use the encoding space {coproc==0b1111, CRn==c9, opc1==={0-7}, opc2==={0-7}, CRm=={c12-c14}}.

Reserved 32-bit encodings with {coproc==0b1111, CRn==c10}

In the AArch32 encoding space, for 32-bit encodings with {coproc==0b1111, CRn==c10}, the following encodings are reserved for IMPLEMENTATION DEFINED purposes:

- Encodings with {coproc==0b1111, CRn==c10, opc==={0-7}, CRm=={c0, c1, c4, c8}} are reserved for IMPLEMENTATION DEFINED TLB lockdown operations.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Reserved 32-bit encodings with {coproc==0b1111, CRn==c11}

In the AArch32 encoding space, for 32-bit encodings with {coproc==0b1111, CRn==c11}, the following encodings are reserved for IMPLEMENTATION DEFINED purposes:

- Encodings with {coproc==0b1111, CRn==c11, opc=={0-7}, CRm=={c0-c8, c15}} are reserved for IMPLEMENTATION DEFINED DMA operations for TCM access.

In Armv8, the remainder of the AArch32 {coproc==0b1111, CRn==c11} encoding space is UNDEFINED.

Reserved 32-bit encodings with {coproc==0b1111, CRn==c15}

Armv8 reserves the AArch32 System register encodings with (coproc==0b1111, CRn==c15) for IMPLEMENTATION DEFINED purposes, and does not impose any restrictions on the use of these encodings. The documentation of the Arm implementation must describe fully any registers implemented in the {coproc==0b1111, CRn==c15} encoding space. Normally, for processor implementations by Arm, this information is included in the *Technical Reference Manual* for the processor.

Typically, an implementation uses the {coproc==0b1111, CRn==c15} encodings to provide test features, and any required configuration options that are not covered by this Manual.

This reservation means that the AArch32 64-bit encodings with {coproc==0b1111, CRm==c15} are also reserved for IMPLEMENTATION DEFINED purposes, without any restrictions on the use of these encodings.

G7.3.2 Full list of VMSAv8-32 System registers in the (coproc==0b1111) encoding space

Table G7-3 on page G7-6082 shows the System registers in the (coproc==0b1111) encoding space in VMSAv8-32, in the order of the {CRn, opc1, CRm, opc2} parameter values used in MCR or MRC accesses to the 32-bit registers:

- For MCR or MRC accesses to the 32-bit registers, CRn is the primary identifier of the target System register for the access. This applies, also, to MCR or MRC instructions that provide 32-bit accesses to a single word of a 64-bit System register.
- For MCRR or MRRC accesses to the 64-bit registers, CRm is the primary identifier of the target System register for the access. Table G7-3 on page G7-6082 orders the 64-bit registers with the 32-bit registers accessed using the same primary register identifier. For example, the 64-bit encoding of TTBR0, that is accessed with (CRm==c2), is listed with the 32-bit registers that are accessed with (CRn==c2).

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G7-3 VMSAv8-32 (`coproc==0b1111`) register summary, in `MCR/MRC` parameter order

Name	Width	CRn	opc1	CRm	opc2	Source
MIDR	32-bit	c0	0	c0	0	v8.0
CTR	32-bit				1	v8.0
TCMTR	32-bit				2	v8.0
TLBTR	32-bit				3	v8.0
MIDR	32-bit				4, 6[a], 7	v8.0
MPIDR	32-bit				5	v8.0
REVIDR	32-bit				6[a]	v8.0
ID_PFR0	32-bit			c1	0	v8.0
ID_PFR1	32-bit				1	v8.0
ID_DFR0	32-bit				2	v8.0
ID_AFR0	32-bit				3	v8.0
ID_MMFR0	32-bit				4	v8.0
ID_MMFR1	32-bit				5	v8.0
ID_MMFR2	32-bit				6	v8.0
ID_MMFR3	32-bit				7	v8.0
ID_ISAR0	32-bit			c2	0	v8.0
ID_ISAR1	32-bit				1	v8.0
ID_ISAR2	32-bit				2	v8.0
ID_ISAR3	32-bit				3	v8.0
ID_ISAR4	32-bit				4	v8.0
ID_ISAR5	32-bit				5	v8.0
ID_MMFR4	32-bit				6	v8.0
ID_PFR2	32-bit			c3	4	v8.0
ID_MMFR5	32-bit				6	v8.0
CCSIDR	32-bit		1	c0	0	v8.0
CLIDR	32-bit				1	v8.0
CCSIDR2	32-bit				2	v8.3[b]
AIDR	32-bit				7	v8.0
CSSELR	32-bit		2	c0	0	v8.0
VPIDR[c]	32-bit		4	c0	0	v8.0
VMPIDR[c]	32-bit				5	v8.0

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G7-3 VMSAv8-32 (`coproc==0b1111`) register summary, in `MCR/MRC` parameter order (continued)

Name	Width	CRn	opc1	CRm	opc2	Source
SCTLR	32-bit	c1	0	c0	0	v8.0
ACTLR	32-bit				1	v8.0
CPACR	32-bit				2	v8.0
ACTLR2	32-bit				3	v8.0
SCR[d]	32-bit			c1	0	v8.0
SDER[d]	32-bit				1	v8.0
NSACR[d]	32-bit				2	v8.0
TRFCR	32-bit			c2	1	v8.4
SDCR	32-bit			c3	1	v8.0
HSCTLR[c]	32-bit		4	c0	0	v8.0
HACTLR[c]	32-bit				1	v8.0
HACTLR2[c]	32-bit				3	v8.0
HCR[c]	32-bit			c1	0	v8.0
HDCR[c]	32-bit				1	v8.0
HCPTR[c]	32-bit				2	v8.0
HSTR[c]	32-bit				3	v8.0
HCR2[c]	32-bit				4	v8.0
HACR[c]	32-bit				7	v8.0
HTRFCR	32-bit			c2	1	v8.4
TTBR0	32-bit	c2	0	c0	0	v8.0
TTBR0	64-bit	-	0	c2	-	v8.0
TTBR1	32-bit	c2	0	c0	1	v8.0
TTBR1	64-bit	-	1	c2	-	v8.0
TTBCR	32-bit	c2	0	c0	2	v8.0
TTBCR2	32-bit				3	v8.2
HTCR[c]	32-bit		4	c0	2	v8.0
VTCR[c]	32-bit			c1	2	v8.0
HTTBR[c]	64-bit	-	4	c2	-	v8.0
VTTBR[c]	64-bit	-	6	c2	-	v8.0
DACR	32-bit	c3	0	c0	0	v8.0
ICC_PMR ICV_PMR	32-bit	c4	0	c6	0	GIC[e]

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G7-3 VMSAv8-32 (`coproc==0b1111`) register summary, in `MCR/MRC` parameter order (continued)

Name	Width	CRn	opc1	CRm	opc2	Source
DSPSR[f]	32-bit	c4	3	c5	0	v8.0
DLR	32-bit				1	v8.0
DFSR	32-bit	c5	0	c0	0	v8.0
IFSR	32-bit				1	v8.0
ADFSR	32-bit			c1	0	v8.0
AIFSR	32-bit				1	v8.0
ERRIDR	32-bit			c3	0	RAS[g]
ERRSELR	32-bit				1	RAS[g]
ERXFR	32-bit			c4	0	RAS[g]
ERXCTLR	32-bit				1	RAS[g]
ERXSTATUS	32-bit				2	RAS[g]
ERXADDR	32-bit				3	RAS[g]
ERXFR2	32-bit				4	RAS[g]
ERXCTLR2	32-bit				5	RAS[g]
ERXADDR2	32-bit				7	RAS[g]
ERXMISC0	32-bit			c5	0	RAS[g]
ERXMISC1	32-bit				1	RAS[g]
ERXMISC4	32-bit				2	RAS[g]
ERXMISC5	32-bit				3	RAS[g]
ERXMISC2	32-bit				4	RAS[g]
ERXMISC3	32-bit				5	RAS[g]
ERXMISC6	32-bit				6	RAS[g]
ERXMISC7	32-bit				7	RAS[g]
HADFSR[c]	32-bit		4	c1	0	v8.0
HAIFSR	32-bit				1	v8.0
HSR[c]	32-bit			c2	0	v8.0
VDFSR	32-bit				3	RAS[g]
DFAR	32-bit	c6	0	c0	0	v8.0
IFAR	32-bit				2	v8.0
HDFAR[c]	32-bit		4	c0	0	v8.0
HIFAR[c]	32-bit				2	v8.0

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G7-3 VMSAv8-32 (`coproc==0b1111`) register summary, in `MCR`/`MRC` parameter order (continued)

Name	Width	CRn	opc1	CRm	opc2	Source
HPFAR[c]	32-bit	c6	4	c0	4	v8.0
ICIALLUIS	32-bit	c7	0	c1	0	v8.0
BPIALLIS	32-bit				6	v8.0
CFPRCTX	32-bit			c3	4	v8.0[h]
DVPRCTX	32-bit				5	v8.0[h]
CPPRCTX	32-bit				7	v8.0[h]
PAR	32-bit			c4	0	v8.0
PAR	64-bit	-	0	c7	-	v8.0
ICIALLU	32-bit	c7	0	c5	0	v8.0
ICIMVAU	32-bit				1	v8.0
CP15ISB[i]	32-bit				4	v8.0
BPIALL	32-bit				6	v8.0
BPIMVA	32-bit				7	v8.0
DCIMVAC	32-bit			c6	1	v8.0
DCISW	32-bit				2	v8.0
ATS1CPR	32-bit			c8	0	v8.0
ATS1CPW	32-bit				1	v8.0
ATS1CUR	32-bit				2	v8.0
ATS1CUW	32-bit				3	v8.0
ATS12NSOPR[d]	32-bit				4	v8.0
ATS12NSOPW[d]	32-bit				5	v8.0
ATS12NSOUR[d]	32-bit				6	v8.0
ATS12NSOUW[d]	32-bit				7	v8.0
DCCMVAC	32-bit			c10	1	v8.0
DCCSW	32-bit				2	v8.0
CP15DSB[i]	32-bit				4	v8.0
CP15DMB[i]	32-bit				5	v8.0
DCCMVAU	32-bit			c11	1	v8.0
DCCIMVAC	32-bit			c14	1	v8.0
DCCISW	32-bit				2	v8.0
ATS1HR[c]	32-bit		4	c8	0	v8.0
ATS1HW[c]	32-bit				1	v8.0

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G7-3 VMSAv8-32 (`coproc==0b1111`) register summary, in `MCR/MRC` parameter order (continued)

Name	Width	CRn	opc1	CRm	opc2	Source
TLBIALLIS	32-bit	c8	0	c3	0	v8.0
TLBIMVAIS	32-bit				1	v8.0
TLBIASIDIS	32-bit				2	v8.0
TLBIMVAAIS	32-bit				3	v8.0
TLBIMVALIS	32-bit				5	v8.0
TLBIMVAALIS	32-bit				7	v8.0
ITLBIALL	32-bit			c5	0	v8.0
ITLBIMVA	32-bit				1	v8.0
ITLBIASID	32-bit				2	v8.0
DTLBIALL	32-bit			c6	0	v8.0
DTLBIMVA	32-bit				1	v8.0
DTLBIASID	32-bit				2	v8.0
TLBIALL	32-bit			c7	0	v8.0
TLBIMVA	32-bit				1	v8.0
TLBIASID	32-bit				2	v8.0
TLBIMVAA	32-bit				3	v8.0
TLBIMVAL	32-bit				5	v8.0
TLBIMVAAL	32-bit				7	v8.0
TLBIIPAS2IS	32-bit		4	c0	1	v8.0
TLBIIPAS2LIS	32-bit				5	v8.0
TLBIALLHIS[c]	32-bit			c3	0	v8.0
TLBIMVAHIS[c]	32-bit				1	v8.0
TLBIALLNSNHIS[c]	32-bit				4	v8.0
TLBIMVALHIS	32-bit				5	v8.0
TLBIIPAS2	32-bit			c4	1	v8.0
TLBIIPAS2L	32-bit				5	v8.0
TLBIALLH[c]	32-bit			c7	0	v8.0
TLBIMVAH[c]	32-bit				1	v8.0
TLBIALLNSNH[c]	32-bit				4	v8.0
TLBIMVALH	32-bit				5	v8.0
Reserved[j]	32-bit	c9	0-7	c0- c2	0-7	-
Reserved[j]	32-bit			c5- c8	0-7	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G7-3 VMSAv8-32 (`coproc==0b1111`) register summary, in `MCR`/`MRC` parameter order (continued)

Name	Width	CRn	opc1	CRm	opc2	Source
PMCR[k]	32-bit	c9	0	c12	0	v8.0
PMCNTENSET[k]	32-bit				1	v8.0
PMCNTENCLR[k]	32-bit				2	v8.0
PMOVSR[k]	32-bit				3	v8.0
PMSWINC[k]	32-bit				4	v8.0
PMSELR[k]	32-bit				5	v8.0
PMCEID0[k]	32-bit				6	v8.0
PMCEID1[k]	32-bit				7	v8.0
PMCCNTR[k]	32-bit			c13	0	v8.0
PMCCNTR_EL0[k]	64-bit	-	0	c9	-	v8.0
PMXEVTYPER[k]	32-bit	c9	0	c13	1	v8.0
PMXEVCNTR[k]	32-bit				2	v8.0
PMUSERENR[k]	32-bit			c14	0	v8.0
PMINTENSET[k]	32-bit				1	v8.0
PMINTENCLR[k]	32-bit				2	v8.0
PMOVSSET[c, k]	32-bit				3	v8.0
PMCEID2[k]	32-bit				4	v8.1
PMCEID3[k]	32-bit				5	v8.1
PMMIR	32-bit				6	v8.4
Reserved[l]	32-bit		0-7	c15	0-7	-
Reserved[m]	32-bit	c10	0	c0- c1	0-7	-
PRRR[n]	32-bit			c2	0	v8.0
MAIR0[n]	32-bit					v8.0
NMRR[n]	32-bit				1	v8.0
MAIR1[n]	32-bit					v8.0
AMAIR0	32-bit			c3	0	v8.0
AMAIR1	32-bit				1	v8.0
Reserved[m]	32-bit			c4, c8	0-7	-
Reserved[m]	32-bit		1-3	c0, c1, c4, c8	0-7	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G7-3 VMSAv8-32 (`coproc==0b1111`) register summary, in `MCR/MRC` parameter order (continued)

Name	Width	CRn	opc1	CRm	opc2	Source
Reserved[m]	32-bit	c10	4	c0, c1	0-7	-
HMAIR0[c]	32-bit			c2	0	v8.0
HMAIR1[c]	32-bit				1	v8.0
HAMAIR0[c]	32-bit			c3	0	v8.0
HAMAIR1[c]	32-bit				1	v8.0
Reserved[m]	32-bit			c4, c8	0-7	-
Reserved[m]	32-bit		5-7	c0, c1, c4, c8	0-7	-
Reserved[o]	32-bit	c11	0-7	c0-c8	0-7	-
Reserved[o]	32-bit			c15	0-7	-
ICC_SGI1R	64-bit	-	0	c12	-	GIC[e]
VBAR	32-bit	c12	0	c0	0	v8.0
MVBAR[d]	32-bit				1	v8.0
RVBAR	32-bit					v8.0
RMR[p]	32-bit				2	v8.0
ISR[d]	32-bit			c1	0	v8.0
DISR	32-bit				1	RAS[g]
VDISR	32-bit		4	c1	1	RAS[g]
ICC_IAR0 ICV_IAR0	32-bit		0	c8	0	GIC[e]
ICC_EOIR0 ICV_EOIR0	32-bit				1	GIC[e]
ICC_HPPIR0 ICV_HPPIR0	32-bit				2	GIC[e]
ICC_BPR0 ICV_BPR0	32-bit				3	GIC[e]
ICC_AP0R0 ICV_AP0R0	32-bit				4	GIC[e]
ICC_AP0R1 ICV_AP0R1	32-bit				5	GIC[e]
ICC_AP0R2 ICV_AP0R2	32-bit				6	GIC[e]
ICC_AP0R3 ICV_AP0R3	32-bit				7	GIC[e]

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G7-3 VMSAv8-32 (`coproc==0b1111`) register summary, in `MCR`/`MRC` parameter order (continued)

Name	Width	CRn	opc1	CRm	opc2	Source
ICC_AP1R0 ICV_AP1R0	32-bit	c12	0	c9	0	GIC[e]
ICC_AP1R1 ICV_AP1R1	32-bit				1	GIC[e]
ICC_AP1R2 ICV_AP1R2	32-bit				2	GIC[e]
ICC_AP1R3 ICV_AP1R3	32-bit				3	GIC[e]
ICC_DIR ICV_DIR	32-bit			c11	1	GIC[e]
ICC_RPR ICV_RPR	32-bit				3	GIC[e]
ICC_IAR1 ICV_IAR1	32-bit			c12	0	GIC[e]
ICC_EOIR1 ICV_EOIR1	32-bit				1	GIC[e]
ICC_HPPIR1 ICV_HPPIR1	32-bit				2	GIC[e]
ICC_BPR1 ICV_BPR1	32-bit				3	GIC[e]
ICC_CTLR ICV_CTLR	32-bit				4	GIC[e]
ICC_SRE	32-bit				5	GIC[e]
ICC_IGRPEN0 ICV_IGRPEN0	32-bit				6	GIC[e]
ICC_IGRPEN1 ICV_IGRPEN1	32-bit				7	GIC[e]
ICC_ASGI1R	64 bit	-	1	c12	-	GIC[e]
ICC_SGI0R	64 bit	-	2	c12	-	GIC[e]
HVBAR[c]	32-bit	c12	4	c0	0	v8.0[e]
HRMR[p]	32-bit				2	v8.0[e]
ICH_AP0R0	32-bit			c8	0	GIC[e]
ICH_AP0R1	32-bit				1	GIC[e]
ICH_AP0R2	32-bit				2	GIC[e]
ICH_AP0R3	32-bit				3	GIC[e]

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G7-3 VMSAv8-32 (`coproc==0b1111`) register summary, in `MCR/MRC` parameter order (continued)

Name	Width	CRn	opc1	CRm	opc2	Source
ICH_AP1R0	32-bit	c12	4	c9	0	GIC[e]
ICH_AP1R1	32-bit				1	GIC[e]
ICH_AP1R2	32-bit				2	GIC[e]
ICH_AP1R3	32-bit				3	GIC[e]
ICC_HSRE	32-bit				5	GIC[e]
ICH_HCR	32-bit			c11	0	GIC[e]
ICH_VTR	32-bit				1	GIC[e]
ICH_MISR	32-bit				2	GIC[e]
ICH_EISR	32-bit				3	GIC[e]
ICH_ELRSR	32-bit				5	GIC[e]
ICH_VMCR	32-bit				7	GIC[e]
ICH_LR<n>, for n==0 to 7	32-bit			c12	0-7	GIC[e]
ICH_LR<n>, for n==8 to 15	32-bit			c13	0-7	GIC[e]
ICH_LRC<n>, for n==0 to 7	32-bit			c14	0-7	GIC[e]
ICH_LRC<n>, for n==8 to 15	32-bit			c15	0-7	GIC[e]
ICC_MCTLR	32-bit		6	c12	4	GIC[e]
ICC_MSRE	32-bit				5	GIC[e]
ICC_MGRPEN1	32-bit				7	GIC[e]
FCSEIDR	32-bit	c13	0	c0	0	v8.0
CONTEXTIDR	32-bit				1	v8.0
TPIDRURW	32-bit				2	v8.0
TPIDRURO	32-bit				3	v8.0
TPIDRPRW	32-bit				4	v8.0
AMCR	32-bit			c2	0	AMU[q]
AMCFGR	32-bit			c2	1	AMU[q]
AMCGCR	32-bit			c2	2	AMU[q]
AMUSERENR	32-bit			c2	3	AMU[q]
AMCNTENCLR0	32-bit			c2	4	AMU[q]
AMCNTENSET0	32-bit			c2	5	AMU[q]
AMCNTENCLR1	32-bit			c3	0	AMU[q]
AMCNTENSET1	32-bit			c3	1	AMU[q]

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G7-3 VMSAv8-32 (`coproc==0b1111`) register summary, in `MCR/MRC` parameter order (continued)

Name	Width	CRn	opc1	CRm	opc2	Source
AMEVTYPER0<n>, for n==0 to 7	32-bit	c13	0	c6	0-7	AMU[q]
AMEVTYPER0<n>, for n==8 to 15	32-bit			c7		AMU[q]
AMEVTYPER1<n>, for n==0 to 7	32-bit			c14		AMU[q]
AMEVTYPER1<n>, for n==8 to 15	32-bit			c15		AMU[q]
AMEVCNTR0<n>, for n==0 to 7	64-bit	-	0-7	c0	-	AMU[q]
AMEVCNTR0<n>, for n==8 to 15	64-bit	-		c1		AMU[q]
AMEVCNTR1<n> for n==0 to 7	64-bit	-		c4		AMU[q]
AMEVCNTR1<n>, for n==8 to 15	64-bit	-		c5		AMU[q]
HTPIDR[c]	32-bit	c13	4	c0	2	v8.0
CNTPCT[r]	64-bit	-	0	c14	-	v8.0
CNTFRQ[r]	32-bit	c14	0	c0	0	v8.0
CNTKCTL[r]	32-bit			c1	0	v8.0
CNTP_TVAL[r]	32-bit			c2	0	v8.0
CNTP_CTL[r]	32-bit				1	v8.0
CNTV_TVAL[r]	32-bit			c3	0	v8.0
CNTV_CTL[r]	32-bit				1	v8.0
PMEVCNTR<n>, for n==0 to 7[k]	32-bit			c8	0-7	v8.0
PMEVCNTR<n>, for n==8 to 15[k]	32-bit			c9	0-7	v8.0
PMEVCNTR<n>, for n==16 to 23[k]	32-bit			c10	0-7	v8.0
PMEVCNTR<n>, for n==24 to 30[k]	32-bit			c11	0-6	v8.0
PMEVTYPER<n>, for n==0 to 7[k]	32-bit			c12	0-7	v8.0
PMEVTYPER<n>, for n==8 to 15[k]	32-bit			c13	0-7	v8.0
PMEVTYPER<n>, for n==16 to 23[k]	32-bit			c14	0-7	v8.0
PMEVTYPER<n>, for n==17 to 30[k]	32-bit			c15	0-6	v8.0
PMCCFILTR[k]	32-bit			c15	7	v8.0
CNTVCT[r]	64-bit	-	1	c14	-	v8.0
CNTP_CVAL[r]	64-bit	-	2	c14	-	v8.0
CNTV_CVAL[r]	64-bit	-	3	c14	-	v8.0
CNTVOFF[s]	64-bit	-	4	c14	-	v8.0
CNTHCTL[r]	32-bit	c14	4	c1	0	v8.0

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G7-3 VMSAv8-32 (coproc==0b1111) register summary, in MCR/MRC parameter order (continued)

Name	Width	CRn	opc1	CRm	opc2	Source
CNTHP_TVAL[r]	32-bit	c14	4	c2	0	v8.0
CNTHP_CTL[r]	32-bit				1	v8.0
CNTHP_CVAL[r]	64-bit	-	6	c14	-	v8.0
CNTPCTSS[r]	64-bit	-	8	c14	-	v8.6
CNTVCTSS[r]	64-bit	-	9	c14	-	v8.6
Reserved[t]	32-bit	c15	0-7	c0-c15	0-7	-

a. REVIDR is an optional register. If it is not implemented, the encoding with opc2 set to 2 is an alias of MIDR.
b. When ARMv8.3-CCIDX is implemented, CCSIDR2 is implemented.
c. Implemented only as part of EL2 when EL2 is using AArch32. Otherwise, encoding is unallocated and UNDEFINED.
d. Implemented only as part of EL3 when EL3 is using AArch32. Otherwise, encoding is unallocated and UNDEFINED.
e. GIC System register, see *About the GIC System registers*. As that subsection describes, each ICV_* register uses the same encoding as the corresponding ICC_* register.
f. This register is accessible only in Debug state.
g. RAS Extension System registers, see *The Reliability, Availability, and Serviceability Extension (RAS Extension)* on page A2-97.
h. When ARMv8.0-PredInv is implemented, the execution and data prediction restriction instructions are implemented, see *Execution and data prediction restriction System instructions* on page G4-5910.
i. For performance reasons, Arm deprecates any use of these memory barrier operations.
j. Reserved for IMPLEMENTATION DEFINED branch predictor, cache, and TCM operations, see *Reserved 32-bit encodings with {coproc==0b1111, CRn==c9}* on page G7-6080.
k. Performance Monitors Extension System register, see *Performance Monitors registers* on page G8-6694.
l. Reserved for IMPLEMENTATION DEFINED performance monitors, see *Reserved 32-bit encodings with {coproc==0b1111, CRn==c9}* on page G7-6080.
m. Reserved for IMPLEMENTATION DEFINED TLB lockdown operations, see *Reserved 32-bit encodings with {coproc==0b1111, CRn==c10}* on page G7-6080.
n. When an implementation is using the Long descriptor translation table format, these encodings access the MAIR0 and MAIR1 registers. Otherwise, they use PRRR and NMRR.
o. Reserved for IMPLEMENTATION DEFINED DMA operations for TCM access, see *Reserved 32-bit encodings with {coproc==0b1111, CRn==c11}* on page G7-6081.
p. Only one of RMR and HRMR is implemented, corresponding to the highest implemented Exception level, and the register is implemented only if that Exception level is using AArch32.
q. Activity Monitors System register, see *Activity Monitors registers* on page G8-6759.
r. Generic Timer System register, see *Generic Timer registers* on page D13-3839.
s. Implemented as RW as part of the Generic Timer on an implementation that includes EL2 and when EL2 is using AArch32. For more information, see *The virtual offset register* on page G6-6067.
t. Reserved for IMPLEMENTATION DEFINED purposes, see *Reserved 32-bit encodings with {coproc==0b1111, CRn==c15}* on page G7-6081.

About the GIC System registers

From version 3.0 of the GIC architecture specification, the specification defines three groups of System registers, identified by the prefix of the register name:

ICC_ GIC physical CPU interface System registers.
ICH_ GIC virtual interface control System registers.
ICV_ GIC Virtual CPU interface System registers.

Note

These registers are in addition to the GIC memory-mapped register groups GICC_, GICD_, GICH_, GICR_, GICV_, and GITS_.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In VMSAv8-32, the GIC System registers are all in the (`coproc==0b1111`) encoding space with (`CRn==c12`). The ICV_* registers have the same {`CRn`, `opc1`, `CRm`, `op2`} encodings as the corresponding ICC_* registers. For these encodings, GIC register configuration fields determine which register is accessed.

When implemented, the GIC System registers form part of an Arm processor implementation, and therefore these registers are included in the register summaries. However, the registers are defined only in the GIC Architecture Specification.

For more information see the *ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0 and version 4.0* (ARM IHI 0069).

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter G8
AArch32 System Register Descriptions

This chapter describes each of the AArch32 System registers.

It contains the following sections:

- *About the AArch32 System registers* on page G8-6096.
- *General system control registers* on page G8-6111.
- *Debug registers* on page G8-6586.
- *Performance Monitors registers* on page G8-6694.
- *Activity Monitors registers* on page G8-6759.
- *RAS registers* on page G8-6792.
- *Generic Timer registers* on page G8-6840.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.1 About the AArch32 System registers

For general information about the AArch32 System registers, see:

In Chapter G5:

- *About the System registers for VMSAv8-32* on page G5-6052.
- *Functional grouping of VMSAv8-32 System registers* on page G5-6058.

In Chapter G7:

- *VMSAv8-32 organization of registers in the (coproc==0b1110) encoding space* on page G7-6075.
- *VMSAv8-32 organization of registers in the (coproc==0b1111) encoding space* on page G7-6078.

In this chapter:

- *Fixed values in the System register descriptions*.
- *General behavior of System registers*.
- *Principles of the ID scheme for fields in ID registers* on page G8-6106.
- *About AArch32 System register accesses* on page G8-6108.

The remainder of this chapter describes the AArch32 System registers, in the following sections:

- *General system control registers* on page G8-6111.
- *Debug registers* on page G8-6586.
- *Performance Monitors registers* on page G8-6694.
- *Generic Timer registers* on page G8-6840.

G8.1.1 Fixed values in the System register descriptions

See *Fixed values in AArch32 instruction and System register descriptions* on page F2-4087. This section defines how the glossary terms RAZ, RES0, RAO, and RES1 can be represented in the System register descriptions.

G8.1.2 General behavior of System registers

Except where indicated, System registers are 32-bits wide. As stated in *About the System registers for VMSAv8-32* on page G5-6052, there are some 64-bit registers, and these include cases where software can access either a 32-bit view or a 64-bit view of a register. The register summaries, and the individual register descriptions, identify the 64-bit registers and how they can be accessed.

The following sections give information about the general behavior of these registers:

- *Register names*.
- *Read-only bits in read/write registers* on page G8-6097.
- *The CPUID identification scheme* on page G8-6097.
- *IMPLEMENTATION DEFINED performance monitors* on page G8-6097.
- *UNPREDICTABLE, CONSTRAINED UNPREDICTABLE, and UNDEFINED behavior for AArch32 System register accesses* on page G8-6097.
- *Read-only and write-only register encodings* on page G8-6099.
- *Reset behavior of AArch32 System registers* on page G8-6100.
- *Synchronization of changes to AArch32 System registers* on page G8-6100.

Unless otherwise indicated, information in the listed sections applies to all AArch32 System registers

See also *About AArch32 System register accesses* on page G8-6108.

Register names

The Arm architecture guarantees not to define any register name prefixed with *IMP_* as part of the standard Arm architecture.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

Arm strongly recommends that any register names created in the IMPLEMENTATION DEFINED register spaces be prefixed with *IMP_*, where appropriate.

Read-only bits in read/write registers

Some read/write registers include bits that are read-only. These bits ignore writes.

The CPUID identification scheme

The ID_* registers were originally called the CPUID identification scheme registers. However, functionally, there is no value in separating these registers from the slightly larger Identification registers functional group. See Table K14-22 on page K14-8085 for a list of the ID_ * registers.

IMPLEMENTATION DEFINED performance monitors

VMSAv8-32 reserves some additional System register encodings in the (`coproc==0b1111`) encoding space for optional additional IMPLEMENTATION DEFINED performance monitors. Table G8-1 shows the allocation of these encodings:

Table G8-1 Performance Monitors System register encoding allocations

CRn	opc1	CRm	opc2	Name	Width	Type
c9	0-7	c12-c14	0-7	Performance Monitors Extension registers, see Table K14-23 on page K14-8086	32-bit	RW or RO[a]
		c15	0-7	IMPLEMENTATION DEFINED	32-bit	[b]

a. The table referenced in the *Name* entry shows the type of each of the OPTIONAL Performance Monitors Extension registers.

b. Access depends on the register or operation, and is IMPLEMENTATION DEFINED.

UNPREDICTABLE, CONSTRAINED UNPREDICTABLE, and UNDEFINED behavior for AArch32 System register accesses

This section defines UNPREDICTABLE and UNDEFINED behaviors for accesses to System registers, including those cases where the Armv8 behavior is CONSTRAINED UNPREDICTABLE.

In AArch32 state the following operations are UNDEFINED:

- All `LDC` and `STC` accesses, except for the `LDC` access to DBGDTRTXint and the `STC` access to DBGDTRRXint specified in Table G7-2 on page G7-6077.
- All `MCRR` and `MRRC` operations to the (`coproc==0b111x`) encoding space, except for those explicitly defined as accessing 64-bit System registers specified in Table G7-1 on page G7-6076 and Table G7-3 on page G7-6082.

Unless otherwise indicated in the individual register descriptions:

- Reserved fields in registers are RES0.
- Assigning a reserved value to a field has a CONSTRAINED UNPREDICTABLE effect, see *Reserved values in System and memory-mapped registers and translation table entries* on page K1-7853.

The following subsections give more information about UNPREDICTABLE, CONSTRAINED UNPREDICTABLE, and UNDEFINED behavior for accesses to the (`coproc==0b111x`) encoding space:

- *Accesses to unallocated encodings in the (coproc==0b111x) encoding space* on page G8-6098.
- *Additional rules for MCR and MRC accesses to System registers* on page G8-6098.
- *Effects of EL3 and EL2 on System register accesses* on page G8-6098.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accesses to unallocated encodings in the (coproc==0b111x) encoding space

In Armv8-A, accesses to unallocated register encodings in the (coproc==0b111x) encoding space are UNDEFINED.

Note

In Armv7, except for 32-bit registers encoded with a CRn value of c12, accesses to unallocated 32-bit registers were UNPREDICTABLE. The Armv8 CONSTRAINED UNPREDICTABLE behavior of these accesses is that they are UNDEFINED, see *Unallocated System register access instructions* on page K1-7834.

Additional rules for MCR and MRC accesses to System registers

The following operations are CONSTRAINED UNPREDICTABLE for all encodings in the (coproc==0b111x) encoding space:

- All MCR operations from the PC.
- All MRC operations to APSR_nzcv, except for the (coproc==0b1110) MRC operation to APSR_nzcv from DBGDSCRint.

The CONSTRAINED UNPREDICTABLE behavior of these operations is that they are UNDEFINED, see *Unallocated System register access instructions* on page K1-7834.

For registers and operations that are accessible from a particular Privilege level, any attempt to access those registers from a lower Privilege level is UNDEFINED.

Some individual registers can be made inaccessible by setting configuration bits, possibly including IMPLEMENTATION DEFINED configuration bits, to disable access to the register. The effects of the architecturally-defined configuration bits are defined individually in this manual. Unless explicitly stated otherwise in this manual, setting a configuration bit to disable access to a register results in the register becoming UNDEFINED for MRC and MCR accesses.

See also *Read-only and write-only register encodings* on page G8-6099.

Effects of EL3 and EL2 on System register accesses

EL2 and EL3 introduce classes of System registers, described in *Classification of System registers* on page G5-6052. Some of these classes of register are either:

- Accessible only from certain modes or states.
- Accessible from certain modes or states only when configuration settings permit the access.

Accesses to these registers that are not permitted are UNDEFINED, meaning execution of the register access instruction generates an Undefined Instruction exception.

Note

This section applies only to registers that are accessible from some modes and states. That is, it applies only to register access instructions using an encoding that, under some circumstances, would perform a valid register access.

The following register classes restrict access in this way:

Restricted access System registers

This register class is defined in any implementation that includes EL3.

Restricted access registers other than the NSACR are accessible only from Secure EL3 modes. All other accesses to these registers are UNDEFINED.

The NSACR is a special case of a Restricted access register and:

- The NSACR is:
  - Read/write accessible from Secure PL1 modes.
  - Is Read-only accessible from Non-secure PL2 and PL1 modes.
- All other accesses to the NSACR are UNDEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For more information, including behavior when EL3 is using AArch64 or is not implemented, see *Restricted access System registers* on page G5-6053.

Configurable access System registers

This register class is defined in any implementation that includes EL3.

Most Configurable access registers are accessible from Non-secure state only if control bits in the NSACR permit Non-secure access to the register. Otherwise, a Non-secure access to the register is UNDEFINED.

For other Configurable access registers, control bits in the NSACR control the behavior of bits or fields in the register when it is accessed from Non-secure state. That is, Non-secure accesses to the register are permitted, but the NSACR controls how they behave. The only architecturally-defined register of this type is the CPACR.

For more information, see *Configurable access System registers* on page G5-6053.

EL2-mode System registers

This register class is defined only in an implementation that includes EL2.

EL2-mode registers are accessible only from:

- The Non-secure EL2 mode, Hyp mode.
- Secure Monitor mode when SCR.NS is set to 1.

All other accesses to these registers are UNDEFINED.

For more information, see *Hyp mode read/write registers in the (coproc==0b1111) encoding space* on page G5-6054 and *Hyp mode encodings for shared (coproc==0b1111) System registers* on page G5-6054.

EL2-mode write-only operations

This register class is defined only in an implementation that includes EL2.

EL2-mode write-only operations are accessible only from:

- The Non-secure EL2 mode, Hyp mode.
- Secure Monitor mode, regardless of the value of SCR.NS.

Write accesses to these operations are:

- CONSTRAINED UNPREDICTABLE in Secure EL3 modes other than Monitor mode.
- UNDEFINED in Non-secure modes other than Hyp mode.

For more information, see *Hyp mode (coproc==0b1111) write-only System instructions* on page G5-6055.

In addition, in any implementation that includes EL3, when EL3 is using AArch32, if write access to a register is disabled by the **CP15SDISABLE** signal then any `MCR` access to that register is UNDEFINED.

Read-only and write-only register encodings

Some System registers are *read-only* (RO) or *write-only* (WO). For example:

- Most identification registers are read-only.
- Most encodings that perform an operation, such as a cache maintenance instruction, are write-only.

If a particular Privilege level defines a register to be:

- RO, then any attempt to write to that register, at that Privilege level, is UNDEFINED. This means that any access to that register with L == 0 is UNDEFINED.
- WO, then any attempt to read from that register, at that Privilege level, is UNDEFINED. This means that any access to that register with L== 1 is UNDEFINED.

For IMPLEMENTATION DEFINED encoding spaces, the treatment of the encodings is IMPLEMENTATION DEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

This section applies only to registers that this manual defines as RO or WO. It does not apply to registers for which other access permissions are explicitly defined.

Reset behavior of AArch32 System registers

Reset values apply only to RW registers and fields, however:

- Some RO registers or fields, including feature ID registers and some status registers or register fields, always return a known value.
- Some RW and RO registers or register fields return status information about the PE. Unless the register description indicates that the value is UNKNOWN on reset, a read of the register immediately after a reset returns valid information.
- Some RW and RO registers and fields are aliases of other registers or fields. In these cases, the reset behavior of the aliased register or field determines the value returned by a read of the register immediately after a reset.
- WO registers that only have an effect on writes do not have meaningful reset values. However, an access to a WO register might affect underlying state, and that state might have a defined reset value.
- IMPLEMENTATION DEFINED registers have IMPLEMENTATION DEFINED reset behavior.

After a reset, only a limited subset of the PE state is guaranteed to be set to defined values. Also, for debug and trace System registers, reset requirements must take account of different levels of reset. For more information about the reset behavior of System registers when the PE resets into an Exception level that is using AArch32, see:

- *PE state on reset into AArch32 state* on page G1-5761.
- The appropriate Trace architecture specification, for the Trace System registers.

When the PE resets into an Exception level that is using AArch64, PE state that relates to execution in AArch32 state, including the System register values, is UNKNOWN. The only exception to this is state that applies to execution in both AArch64 state and AArch32 state and that has a defined reset value on the reset into AArch64 state. An example of such PE state is the EDPRSR.SR bit.

For a PE reset into an Exception level that is using AArch32, the architecture defines which AArch32 System registers have a defined reset value, and when that defined reset value applies. The register descriptions include this information, and *PE state on reset into AArch32 state* on page G1-5761 summarizes these architectural requirements. Otherwise, RW registers reset to an architecturally unknown value.

Note

In an implementation that includes EL3, unless this manual explicitly states otherwise, only the Secure instance of a banked register is reset to the defined value. This means that software must program the Non-secure instance of the register with the required values. Typically, this programming is part of the PE boot sequence.

Pseudocode description of resetting System registers

The `AArch32.ResetControlRegisters()` pseudocode function resets all System registers, and register fields, that have defined reset values, as described in this section and *PE state on reset into AArch32 state* on page G1-5761.

Note

For debug and trace System registers, this function resets registers as defined for the appropriate level of reset.

Synchronization of changes to AArch32 System registers

In this section, *this PE* means the PE on which accesses are being synchronized.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

See *Definitions of direct and indirect reads and writes and their side-effects* on page G8-6105 for definitions of the terms *direct write*, *direct read*, *indirect write*, and *indirect read*.

A *direct write* to a System register might become visible at any point after the change to the register, but without a *Context synchronization event* there is no guarantee that the change becomes visible.

Any direct write to a System register is guaranteed not to affect any instruction that appears, in program order, before the instruction that performed the direct write, and any direct write to a System register must be synchronized before any instruction that appears after the direct write, in program order, can rely on the effect of that write. The only exceptions to this are:

- All direct writes to the same register, using the same encoding, are guaranteed to occur in program order.
- All direct writes to a register are guaranteed to occur in program order relative to all direct reads of the same register using the same encoding.
- Any System register access that an Arm *Architecture Specification* or equivalent specification defines as not requiring synchronization.
- If an instruction that appears in program order before the direct write performs a memory access, such as a memory-mapped register access, that causes an indirect read or write to a register, that memory access is subject to the memory order model. In this case, if permitted by the memory order model, the instruction that appears in program order before the direct write can be affected by the direct write. For information about the memory order model, see *Definition of the Armv8 memory model* on page E2-3980.

These rules mean that an instruction that writes to one of the address translation instructions described in *Address translation instructions* on page G5-6043 must be explicitly synchronized to guarantee that the result of the address translation instruction is visible in the PAR.

Note

In this case, the direct write to the encoding of the address translation instruction causes an *indirect write* to the PAR. Without a *Context synchronization event* after the direct write, there is no guarantee that the indirect write to the PAR is visible.

Conceptually, the explicit synchronization occurs as the first step of any *Context synchronization event*. This means that if the operation uses the state that had been changed but not synchronized before the operation occurred, the operation is guaranteed to use the state as if it had been synchronized.

Note

- This explicit synchronization is applied as the first step of the execution of any instruction that causes the synchronization operation. This means it does not synchronize any effect of changes to the System registers that might affect the fetch and decode of the instructions that cause the operation, such as breakpoints or changes to translation tables.
- For a synchronous exception, the control state in use at the time the exception is generated determines the exception syndrome information, and this syndrome information is not changed by this synchronization at the start of taking the exception.

Except for the register reads listed in *Registers with some architectural guarantee of ordering or observability* on page G8-6103, if no *Context synchronization event* is performed, direct reads of System registers can occur in any order.

Table G8-2 on page G8-6102 shows the synchronization requirement between two reads or writes that access the same System register. In the column headings, *First* and *Second* refer to:

- Program order, for any read or write caused by the execution of an instruction by this PE, other than a read or write caused by a memory access made by that instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The order of arrival of asynchronous reads or writes made by this PE relative to the execution of instructions by this PE.

In addition:

- For indirect reads or writes caused by an external agent, such as a debugger, the mechanism that determines the order of the reads or writes is defined by that external agent. The external agent can provide mechanisms that ensure that any read or write it makes arrives at the PE. These indirect reads and writes are asynchronous to software execution on the PE.
- For indirect reads or writes caused by memory-mapped reads or writes made by this PE, the ordering of the memory accesses is subject to the memory order model, including the effect of the memory type of the accessed memory address. This applies, for example, if this PE reads or writes one of its registers in a memory-mapped register interface.

  The mechanism for ensuring completion of these memory accesses, including ensuring the arrival of the asynchronous read or write at the PE, is defined by the system.

  Note

  Such accesses are likely to be given a Device memory attribute, but requiring this is outside the scope of the architecture.

- For indirect reads or writes caused by autonomous asynchronous events that are counted, for example events caused by the passage of time, the events are ordered so that:
  — Counts progress monotonically.
  — The events arrive at the PE in finite time and without undue delay.

Table G8-2 Synchronization requirements for updates to System registers

First read or write	Second read or write	*Context synchronization event* required
Direct read	Direct read	No
	Direct write	No
	Indirect read	No[a]
	Indirect write	No[a], but see text in this section for exceptions
Direct write	Direct read	No
	Direct write	No
	Indirect read	Yes[a]
	Indirect write	No, but see text in this section for exceptions
Indirect read	Direct read	No
	Direct write	No
	Indirect read	No
	Indirect write	No
Indirect write	Direct read	Yes, but see text in this section for exceptions
	Direct write	No, but see text in this section for exceptions
	Indirect read	Yes, but see text in this section for exceptions
	Indirect write	No, but see text in this section for exceptions

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

a. Although no synchronization is required between a Direct write and a Direct read, or between a Direct read and an Indirect write, this does not imply that a Direct read causes synchronization of a previous Direct write. This means that the sequence Direct write followed by Direct read followed by Indirect read, with no intervening context synchronization, does not guarantee that the Indirect read observes the result of the Direct write.

If the indirect write is to a register that *Registers with some architectural guarantee of ordering or observability* shows as having some guarantee of the visibility of an indirect write, synchronization might not be required.

If a direct read or a direct write to a register is followed by an indirect write to that register that is caused by an external agent, or by an autonomous asynchronous event, or as a result of a memory-mapped write, then synchronization is required to guarantee the ordering of the indirect write relative to the direct read or direct write.

If an indirect write caused by a direct write is followed by an indirect write caused by an external agent, or by an autonomous asynchronous event, or as a result of a memory-mapped write, then synchronization is required to guarantee the ordering of the two indirect writes.

Where an indirect write occurs as a side-effect of an access, this happens atomically with the access, meaning no other accesses are allowed between the register access and its side-effect. For other information about indirect writes after a direct read or a direct write, see *Definitions of direct and indirect reads and writes and their side-effects* on page G8-6105

Note

Where a register has more that one encoding, a direct write to the register using a particular encoding is not an indirect write to the same register with a different encoding.

Where an indirect write is caused by the action of an external agent, such as a debugger, or by a memory-mapped read or write by the PE, then an indirect write by that agent to a register using a particular access mechanism, followed by an indirect read by that agent to the same register using the same access mechanism and address does not need synchronization.

Without explicit synchronization to guarantee the order of the accesses, where the same register is accessed by two or more of a System register access instruction, and external agent, and autonomous asynchronous event, or as a result of a memory-mapped access, the behavior must be as if the accesses occurred atomically and in any order. This applies even if the accesses occur simultaneously.

For information about the additional synchronization requirements for memory-mapped registers, see *Synchronization requirements for AArch64 System registers* on page D13-2869.

To guarantee the visibility of changes to some registers, additional operations might be required before the *Context synchronization event*. For such a register, the definition of the register identifies these additional requirements.

In this manual, unless the context indicates otherwise:

- *Accessing* a System register refers to a direct read or write of the register.
- *Using* a System register refers to an indirect read or write of the register.

Registers with some architectural guarantee of ordering or observability

For the registers for which Table G8-3 on page G8-6104 shows that the ordering of direct reads is guaranteed, multiple direct reads of a single register, using the same encoding, occur in program order without any explicit ordering.

For the registers for which Table G8-3 on page G8-6104 shows that some observability of indirect writes is guaranteed, an indirect write to the register caused by an external agent, an autonomous asynchronous event, or as a result of a memory-mapped write, is both:

- Observable to direct reads of the register, in finite time, without explicit synchronization.
- Observable to subsequent indirect reads of the register without explicit synchronization.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

These two sets of registers are similar, as Table G8-3 shows:

Table G8-3 Registers with a guarantee of ordering or observability, VMSAv8-32

Register	Ordering of direct reads	Observability of indirect writes	Notes
ISR	Guaranteed	Guaranteed	Interrupt Status Register
DBGCLAIMCLR	Guaranteed	Guaranteed	Debug CLAIM registers
DBGCLAIMSET	-	Guaranteed	
DBGDTRRXint	Guaranteed	Guaranteed	Debug Communication Channel registers
DBGDTRTXint	-	Guaranteed	
The DCC flags in DBGDSCRint	Guaranteed	Guaranteed	
CNTPCT	Guaranteed	Guaranteed	Generic Timer registers
CNTP_TVAL	Guaranteed	Guaranteed	
CNTVCT	Guaranteed	Guaranteed	
CNTV_TVAL	Guaranteed	Guaranteed	
CNTHP_TVAL	Guaranteed	Guaranteed	
PMCCNTR	Guaranteed	Guaranteed	Performance Monitors Extension registers, if the implementation includes the extension
PMEVCNTR<n>	Guaranteed	Guaranteed	
PMXEVCNTR	Guaranteed	Guaranteed	
PMOVSSET	Guaranteed	Guaranteed	
PMOVSR	Guaranteed	Guaranteed	
EDSCR.PipeAdv and the DCC flags in EDSCR	-	Guaranteed	Fields of the External Debug Status and Control Register

In addition to the requirements shown in Table G8-3:

- Indirect writes to the following registers as a result of memory-mapped writes, including accesses by external agents, are required to be observable to the indirect read made in determining the response to a subsequent memory-mapped access without explicit synchronization:
  - OSLAR_EL1. OSLAR_EL1 is indirectly read to determine whether the subsequent access is permitted.

    Note

    OSLAR_EL1 maps to the AArch32 System register DBGOSLAR.

  - EDLAR, if implemented. EDLAR is indirectly read to determine whether a subsequent write or side-effect of an access is ignored.

  Note

  This requirement is stricter than the general requirement for the observability of indirect writes.

- The requirement that an indirect write to the registers in Table G8-3 is observable to direct reads in finite time does not imply that all observers will observe the indirect write at the same time.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For example, an increment of the system counter is an autonomous asynchronous event that performs an indirect write to the counter. This asynchronous event might generate a timer interrupt request, resulting in a *Context synchronization event*. When a GIC is used, the timer interrupt might arrive at the GIC after the PE has taken an interrupt request from another source, but before software reads the current interrupt ID from the GIC. This means that the GIC might identify the timer interrupt as the current interrupt. Software must not assume that a subsequent direct read of the counter register is guaranteed to observe the updated value of that register.

Although this example uses the counter-timer registers, it applies equally to other registers that might be linked to interrupt requests, including the PMU and Statistical Profiling status registers.

- When the PE is in Debug state, there are synchronization requirements for the Debug Communication Channel and Instruction Transfer registers. See *DCC and ITR access in Debug state* on page H4-7006.

The possibility that direct reads can occur *early*, in the absence of context synchronization, described in *Ordering of reads of System registers* on page G8-6108, still applies to the registers listed in Table G8-3 on page G8-6104.

Definitions of direct and indirect reads and writes and their side-effects

Direct and indirect reads and writes are defined as follows:

Direct read Is a read of a register, using an `MRC`, `MRRC`, or `STC` instruction, that the architecture permits for the current PE state.

If a direct read of a register has a side-effect of changing the value of a register, the effect of a direct read on that register is defined to be an *indirect write*, and has the synchronization requirements of an indirect write. This means the indirect write is guaranteed to have occurred, and to be visible to subsequent direct or indirect reads and writes only if synchronization is performed after the direct read.

Note

The indirect write described here can affect either the register written to by the direct write, or some other register. The synchronization requirement is the same in both cases.

Direct write Is a write to a register, using an `MCR`, `MCRR`, or `LDC` instruction, that the architecture permits for the current PE state.

In the following cases, the side-effect of the direct write is defined to be an indirect write of the affected register, and has the synchronization requirements of an indirect write:

- If the direct write has a side-effect of changing the value of a register other than the register accessed by the direct write.
- If the direct write has a side-effect of changing the value of the register accessed by the direct write, so that the value in that register might not be the value that the direct write wrote to the register.

In both cases, this means that the indirect write is not guaranteed to be visible to subsequent direct or indirect reads and writes unless synchronization is performed after the direct write.

Note

- As an example of a direct write to a register having an effect that is an indirect write of that register, writing 1 to a PMCNTENCLR.P*x* bit is also an indirect write, because if the P*x* bit had the value 1 before the direct write, the side-effect of the write changes the value of that bit to 0.
- The indirect write described here can affect either the register written to by the direct write, or some other register. The synchronization requirement is the same in both cases.

  For example, writing 1 to a PMCNTENCLR.P*x* bit that is set to 1 also changes the corresponding PMCNTENSET.P*x* bit from 1 to 0. This means that the direct write to the PMCNTENCLR defines indirect writes to both itself and to the PMCNTENSET.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Indirect read Is a use of the register by an instruction to establish the operating conditions for the instruction. Examples of operating conditions that might be determined by an indirect read are the translation table base address, or whether memory accesses are forced to be Non-cacheable.

Indirect reads include situations where the value of one register determines what value is returned by a second register. This means that any read of the second register is an indirect read of the register that determines what value is returned.

Indirect reads also include:

- Reads of the System registers by external agents, such as debuggers, as described in *Debug registers* on page G8-6586.
- Memory-mapped reads of the System registers made by the PE on which the System registers are implemented.

Where an indirect read of a register has a side-effect of changing the value of a register, that change is defined to be an indirect write, and has the synchronization requirements of an indirect write.

Indirect write Is an update to the value of a register as a consequence of either:

- An exception, operation, or execution of an instruction that is not a direct write to that register.
- The asynchronous operation of an external agent.

This can include:

- The passage of time, as seen in counters or timers, including performance counters.
- The assertion of an interrupt.
- A write from an external agent, such as a debugger.

However, for some registers, the architecture gives some guarantee of visibility without any explicit synchronization, see *Registers with some architectural guarantee of ordering or observability* on page G8-6103.

Note

Taking an exception is a *Context synchronization event*. Any indirect write performed as part of an exception entry does not require additional synchronization. This includes the indirect writes to the registers that report the exception, as described in *Exception reporting in a VMSAv8-32 implementation* on page G5-6024.

G8.1.3 Principles of the ID scheme for fields in ID registers

The Arm architecture specifies a number of *ID registers* that are characterized as comprising a set of 4-bit *ID fields,* Each ID field identifies the presence, and possibly the level of support for, a particular feature in an implementation of the architecture. These fields follow an architectural model that aids their use by software and provides future compatibility. This section describes that model. *AArch32 ID registers to which this scheme applies* on page G8-6107 identifies the set of ID registers that are accessible from AArch32 state.

A small number of ID fields do not follow the scheme described in this section. In these cases, the field description states that it does not follow this scheme.

Note

- The ID fields described here are distinct from register fields that enumerate the number of resources, such as the number of breakpoints, watchpoints, or performance monitors, or the amount of memory.
- ID fields that do not follow this scheme include the ID_AA64DFR0_EL1.PMUVer, ID_DFR0_EL1.PerfMon, ID_DFR0.PerfMon and EDDFR.PMUVer fields, see *Alternative ID scheme used for the Performance Monitors Extension version* on page G8-6108.
- The presence of an ID field for a feature does not imply that the feature is optional.

To provide forward compatibility, software can rely on the features of these fields that are described in this section.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The ID fields, which are either signed or unsigned, use increasing numerical values to indicate increases in functionality. Therefore, if a value of `0x1` indicates the presence of some instructions, then the value `0x2` will indicate the presence of those instructions plus some additional instructions or functionality. This means software can be written in the form:

```
if (value >= number) {    // do something that relies on the value of the feature}
```

For ID fields where the value `0x0` defines that a feature is not present, the field holds an unsigned value. This covers the vast majority of such fields.

In a few cases, the architecture has been changed to permit implementations to exclude a feature that has previously been required and for which no ID field has been defined. In these cases, a new ID field is defined and:

- The field holds a signed value.
- The field value `0xF` indicates that the feature is not implemented.
- The field value `0x0` indicates that the feature is implemented.
- Software that depends on the feature can use the test:

```
    if value >= 0 {        // Software features that depend on the presence of the hardware
feature    }
```

In some cases, it has been decided retrospectively that the increase in functionality between two consecutive numerical values is too great, and it is desirable to permit an intermediate degree of functionality, and the means to discover this. This is done by the introduction of a *fractional* field that both:

- Is referred to in the definition of the original field.
- Applies only when the original field is at the lower value of the step.

In principle, a fractional field can be used for two different fractional steps, with different meanings associated with each of these steps. For this reason, a fractional field must be interpreted in the context of the field to which it relates and the value of that field. Example G8-1 shows the use of such a field.

Example G8-1 Example of the use of a fractional field

For a field describing some class of functionality:

- The value `0x1` was defined as indicating that item A is present.
- The value `0x2` was defined as indicating that items B and C are present, in addition to item A.

Subsequently, it might be necessary to introduce a second ID field to indicate that A and B only are present. This new field is a fractional field, and might be defined as having the value `0x1` when A and B only are present. This fractional field is valid only when the original ID field has the value `0x1`.

This approach means that:

- Software that depends on the test if `(value >= 0x2)` can rely on features A, B, and C being present,
- Software that depends on the test `if (value >= 0x1)` can rely on feature A being present.
- If new software needs to check only that features A and B are present, then it can test:

```
    if (value >= 0x2 || (value == 0x1 && fractional_value >= 0x1)) {        // Software features
that depend on A and B only    }
```

A fractional field uses the same approach of increasing numerical values indicating increasing functionality, and the fractional approach can also be applied recursively to fractional fields.

Unused ID fields, and fractional fields that are not applicable, are RES0 to allow their future use when features, or fractional implementation options, are added.

AArch32 ID registers to which this scheme applies

- The Auxiliary Feature register ID_AFR0.
- The Processor Feature registers ID_PFR0 and ID_PFR1.
- The Debug Feature register ID_DFR0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The Memory Model Feature registers ID_MMFR0, ID_MMFR1, ID_MMFR2, ID_MMFR3, and ID_MMFR4.
- The Instruction Set Attribute registers ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR3, ID_ISAR4, and ID_ISAR5.
- The Media and VFP Feature registers MVFR0, MVFR1, and MVFR2.

Note

Principles of the ID scheme for fields in ID registers on page D13-2873 includes information about the AArch64 System registers and the memory-mapped registers to which this scheme applies.

Alternative ID scheme used for the Performance Monitors Extension version

The ID_AA64DFR0_EL1.PMUVer, ID_DFR0_EL1.PerfMon, ID_DFR0.PerfMon, and EDDFR.PMUVer fields, which identify the version of the Performance Monitors Extension, do not follow the standard ID scheme. Software must treat these fields as follows:

- The value `0xF` indicates that the Arm-architected Performance Monitors Extension is not implemented.
- If the field value is not `0xF` the field is treated as an unsigned value, as described for the standard ID scheme.

This means that software that depends on the implementation of a particular version of the Arm Performance Monitors Extension must be written in the form:

```
    if (value != 0xF and value >= number) {        // do something that relies on version 'number' of the
feature    }
```

For these fields, Arm deprecates use of the value `0xF` in new implementations.

G8.1.4 About AArch32 System register accesses

The following subsections give more information about accesses to the AArch32 System registers:

- *Ordering of reads of System registers*.
- *Accessing 32-bit System registers* on page G8-6109.
- *Accessing 64-bit System registers* on page G8-6110.

Ordering of reads of System registers

Reads of the System registers can occur out of order with respect to earlier instructions executed on the same PE, provided that both:

- Any data dependencies between the instructions, as specified in *Synchronization of changes to AArch32 System registers* on page G8-6100, including read-after-read dependencies, are respected.
- The reads to the register do not occur earlier than the most recent *Context synchronization event* to its architectural position in the instruction stream.

Note

In particular, the values read from System registers that hold self-incrementing counts, such as the Performance Monitors counters or the Generic Timer counter or timers, could be accessed from any time after the previous *Context synchronization event*. For example, where a memory access is used to communicate a read of such a counter, an `ISB` must be inserted between the read of the memory location that is known to have returned its data, either as a result of a condition on that data or of the read having completed, and the read of the counter, if it is necessary that the counter returns a count value after the memory communication.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing 32-bit System registers

Software accesses most 32-bit System registers using the generic MCR and MRC System register access instructions, specifying some or all of the parameters {coproc, CRn, opc1, CRm, opc2}, where:

coproc
: Identifies the primary region of the System register encoding space. Takes one of the values:
  - p14 — Encoded as 0b1110.
  - p15 — Encoded as 0b1111.

CRn
: Takes a value in the range c0-c15, encoded the corresponding 4-bit binary value, 0b0000-0b1111.

  In the (coproc==0b1110) encoding space, the opc1 value identifies the System register functional group, and CRn is the most significant identifier for the required register within that group.

  In the (coproc==0b1111) encoding space, CRn is the most significant identifier for the required register.

opc1
: Takes a value in the range 0-7, encoded as its 3-bit binary value.

  In the (coproc==0b1110) encoding space, the opc1 value identifies the System register functional group, and can take the following values:
  - 0 — Debug System registers.
  - 1 — Trace System registers.
  - 7 — Legacy Jazelle System registers.

  In the (coproc==0b1111) encoding space, opc1 can take any value in the range 0-7.

CRm
: Takes a value in the range c0-c15, encoded the corresponding 4-bit binary value, 0b0000-0b1111.

opc2
: Takes a value in the range 0-7, encoded as its 3-bit binary value.

  opc2 is optional in the MCR and MRC instruction syntax, and if no value is specified the encoding defaults to 0b000.

Rt
: A general-purpose register to hold a 32-bit value to transfer to or from the System register. Takes a value in the range R0-R14, encoded as the corresponding 4-bit binary value, 0b0000-0b1110.

This means an MCR or MRC access to a specific 32-bit System register uses:

- A unique combination of coproc, CRn, opc1, CRm, and opc2, to specify the required System register.
- A general-purpose register, Rt, for the transferred 32-bit value.

See also:

- *MCR* on page F5-4505.
- *MRC* on page F5-4528.

A small number of AArch32 debug System registers are accessed using LDC or STC instructions. In these cases, the register to be accessed is identified in the instruction syntax by the use of p14, c5 where:

p14
: Identifies that the access is to the (coproc==0b1110) encoding space.

c5
: Identifies the target debug System register.

See the instruction descriptions:

- *LDC (immediate)* on page F5-4394.
- *LDC (literal)* on page F5-4396.
- *STC* on page F5-4745.

The only uses of LDC and STC permitted in Armv8-A are:

- An LDC access to load data from memory to DBGDTRTXint, see *LDC (immediate)* on page F5-4394 and *LDC (literal)* on page F5-4396.
- An STC access to store data to memory from DBGDTRRXint, see *STC* on page F5-4745.

A small number of AArch32 System registers are accessed using MRS, MSR, VMRS, or VMSR instructions, see the appropriate register and instruction description for more information, see:

- *MRS* on page F5-4532.
- *MSR (immediate)* on page F5-4542,
- *MSR (register)* on page F5-4544.
- *VMRS* on page F6-5346.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *VMSR* on page F6-5349.

Note

- For example:
  - The APSR, CPSR, and SPSR are accessed using MRS or MSR instructions.
  - The MVFR0, MVFR1, and MVFR2 are accessed using VMRS or VMSR instructions.
- In addition, the banked register forms of the MRS and MSR instructions can be used to access some System registers associated with PE modes other than the mode in which the PE is currently executing, see *MRS (Banked register)* on page F5-4534 and *MSR (Banked register)* on page F5-4538.

Accessing 64-bit System registers

Software accesses a 64-bit System register using the generic MCRR and MRRC System register access instructions, specifying the parameters {coproc, CRm, opc1}, where:

coproc
: Identifies the primary region of the System register encoding space. Takes one of the values:
  - p14 Encoded as 0b1110.
  - p15 Encoded as 0b1111.

CRm
: Takes a value in the range c0-c15, encoded the corresponding 4-bit binary value, 0b0000-0b1111.

  In the (coproc==0b1110) encoding space, the opc1 value identifies the System register functional group, and CRm is the most significant identifier for the required register within that group.

  In the (coproc==0b1111) encoding space, CRm is the most significant identifier for the required register.

opc1
: Takes a value in the range 0-15, encoded as its 3-bit binary value.

  In the (coproc==0b1110) encoding space, the opc1 value identifies the System register functional group, and can take the following values:
  - 0 Debug System registers.
  - 1 Trace System registers.

  In the (coproc==0b1111) encoding space, opc1 can take any value in the range 0-15.

Rt
: A general-purpose register to hold bits[31:0] of the value to transfer to or from the System register. Takes a value in the range R0-R14, encoded as the corresponding 4-bit binary value, 0b0000-0b1110.

Rt2
: A general-purpose register to hold bits[63:32] of the value to transfer to or from the System register. Takes a value in the range R0-R14, encoded as the corresponding 4-bit binary value, 0b0000-0b1110.

This means an MCRR or MRRC access to a specific 64-bit System register uses:

- A unique combination of coproc, CRm and opc1, to specify the required 64-bit System register.
- Two general-purpose registers, each holding 32 bits of the value to transfer.

This means a PE can access a 64-bit System register using:

- An MCRR instruction to write to a System register, see *MCRR* on page F5-4507.
- An MRRC instruction to read a System register, see *MRRC* on page F5-4530.

When using an MCRR or MRRC instruction the System register access is 64-bit atomic.

Some 64-bit registers also have an MCR and MRC encoding. The MCR and MRC encodings for these registers access the least significant 32 bits of the register. For example, to access the PAR, software can:

- Use the following instructions to access all 64 bits of the register:

```
MRRC p15, 0, <Rt>, <Rt2>, c7    ; Read 64-bit PAR into Rt (low word) and Rt2 (high word)
MCRR p15, 0, <Rt>, <Rt2>, c7    ; Write Rt (low word) and Rt2 (high word) to 64-bit PAR
```

- Use the following instructions to access the least-significant 32 bits of the register:

```
MRC p15, 0, <Rt>, c7, c4, 0    ; Read PAR[31:0] into Rt
MCR p15, 0, <Rt>, c7, c4, 0    ; Write Rt to PAR[31:0]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2 General system control registers

This section lists the System registers in AArch32 state that are not part of one of the other listed groups.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.1 ACTLR, Auxiliary Control Register

The ACTLR characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED configuration and control options for execution at EL1 and EL0.

Configurations

AArch32 System register ACTLR[31:0] is architecturally mapped to AArch64 System register ACTLR_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ACTLR are UNKNOWN.

Some bits might define global configuration settings, and be common to the Secure and Non-secure instances of the register.

Attributes

ACTLR is a 32-bit register.

Field descriptions

The ACTLR bit assignments are:

31 … 0
IMPLEMENTATION DEFINED

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

Accessing the ACTLR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TACR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TAC == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return ACTLR_NS;
    else
        return ACTLR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return ACTLR_NS;
    else
        return ACTLR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return ACTLR_S;
    else
        return ACTLR_NS;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TACR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TAC == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        ACTLR_NS = R[t];
    else
        ACTLR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        ACTLR_NS = R[t];
    else
        ACTLR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        ACTLR_S = R[t];
    else
        ACTLR_NS = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.2 ACTLR2, Auxiliary Control Register 2

The ACTLR2 characteristics are:

Purpose

Provides additional space to the ACTLR register to hold IMPLEMENTATION DEFINED trap functionality for execution at EL1 and EL0.

Configurations

AArch32 System register ACTLR2[31:0] is architecturally mapped to AArch64 System register ACTLR_EL1[63:32].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ACTLR2 are UNKNOWN.

In Armv8.0 and Armv8.1, it is IMPLEMENTATION DEFINED whether this register is implemented, or whether it causes UNDEFINED exceptions when accessed. The implementation of this register can be detected by examining ID_MMFR4.AC2.

From Armv8.2 this register must be implemented.

Attributes

ACTLR2 is a 32-bit register.

Field descriptions

The ACTLR2 bit assignments are:

31 … 0
IMPLEMENTATION DEFINED

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

Accessing the ACTLR2

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0000	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TACR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TAC == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return ACTLR2_NS;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        return ACTLR2;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return ACTLR2_NS;
    else
        return ACTLR2;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return ACTLR2_S;
    else
        return ACTLR2_NS;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0000	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TACR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TAC == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        ACTLR2_NS = R[t];
    else
        ACTLR2 = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        ACTLR2_NS = R[t];
    else
        ACTLR2 = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        ACTLR2_S = R[t];
    else
        ACTLR2_NS = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.3 ADFSR, Auxiliary Data Fault Status Register

The ADFSR characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED fault status information for Data Abort exceptions taken to EL1 modes, and EL3 modes when EL3 is implemented and is using AArch32.

Configurations

AArch32 System register ADFSR[31:0] is architecturally mapped to AArch64 System register AFSR0_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ADFSR are UNKNOWN.

Attributes

ADFSR is a 32-bit register.

Field descriptions

The ADFSR bit assignments are:

31 … 0
IMPLEMENTATION DEFINED

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

Accessing the ADFSR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TRVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return ADFSR_NS;
    else
        return ADFSR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return ADFSR_NS;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        return ADFSR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return ADFSR_S;
    else
        return ADFSR_NS;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        ADFSR_NS = R[t];
    else
        ADFSR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        ADFSR_NS = R[t];
    else
        ADFSR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        ADFSR_S = R[t];
    else
        ADFSR_NS = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.4 AIDR, Auxiliary ID Register

The AIDR characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED identification information.

The value of this register must be used in conjunction with the value of MIDR.

Configurations

AArch32 System register AIDR[31:0] is architecturally mapped to AArch64 System register AIDR_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to AIDR are UNKNOWN.

Attributes

AIDR is a 32-bit register.

Field descriptions

The AIDR bit assignments are:

31 … 0
IMPLEMENTATION DEFINED

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

Accessing the AIDR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b001	0b0000	0b0000	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        return AIDR;
elsif PSTATE.EL == EL2 then
    return AIDR;
elsif PSTATE.EL == EL3 then
    return AIDR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.5 AIFSR, Auxiliary Instruction Fault Status Register

The AIFSR characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED fault status information for Prefetch Abort exceptions taken to EL1 modes, and EL3 modes when EL3 is implemented and is using AArch32.

Configurations

AArch32 System register AIFSR[31:0] is architecturally mapped to AArch64 System register AFSR1_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to AIFSR are UNKNOWN.

Attributes

AIFSR is a 32-bit register.

Field descriptions

The AIFSR bit assignments are:

31 … 0
IMPLEMENTATION DEFINED

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

Accessing the AIFSR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TRVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return AIFSR_NS;
    else
        return AIFSR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return AIFSR_NS;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        return AIFSR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return AIFSR_S;
    else
        return AIFSR_NS;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        AIFSR_NS = R[t];
    else
        AIFSR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        AIFSR_NS = R[t];
    else
        AIFSR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        AIFSR_S = R[t];
    else
        AIFSR_NS = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.6 AMAIR0, Auxiliary Memory Attribute Indirection Register 0

The AMAIR0 characteristics are:

Purpose

When using the Long-descriptor format translation tables for stage 1 translations, provides IMPLEMENTATION DEFINED memory attributes for the memory regions specified by MAIR0.

Configurations

AArch32 System register AMAIR0[31:0] is architecturally mapped to AArch64 System register AMAIR_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to AMAIR0 are UNKNOWN.

Attributes

AMAIR0 is a 32-bit register.

Field descriptions

The AMAIR0 bit assignments are:

31 … 0
IMPLEMENTATION DEFINED

This register is RES0 in the following cases:

- When an implementation does not provide any IMPLEMENTATION DEFINED memory attributes.
- When the Long-descriptor translation table format is not used.

If EL3 is implemented and is using AArch32:

- AMAIR0(S) gives the value for memory accesses from Secure state.
- AMAIR0(NS) gives the value for memory accesses from Non-secure states other than Hyp mode.

Any IMPLEMENTATION DEFINED memory attributes are additional qualifiers for the memory locations and must not change the architected behavior specified by MAIR0 and MAIR1.

In a typical implementation, AMAIR0 and AMAIR1 split into eight one-byte fields, corresponding to the MAIRn.Attr<n> fields, but the architecture does not require them to do so.

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

Accessing the AMAIR0

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1010	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T10 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T10 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TRVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return AMAIR0_NS;
    else
        return AMAIR0;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return AMAIR0_NS;
    else
        return AMAIR0;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return AMAIR0_S;
    else
        return AMAIR0_NS;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1010	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T10 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T10 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        AMAIR0_NS = R[t];
    else
        AMAIR0 = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        AMAIR0_NS = R[t];
    else
        AMAIR0 = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' && CP15SDISABLE == HIGH then
        UNDEFINED;
    elsif SCR.NS == '0' && CP15SDISABLE2 == HIGH then
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
else
    if SCR.NS == '0' then
        AMAIR0_S = R[t];
    else
        AMAIR0_NS = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.7 AMAIR1, Auxiliary Memory Attribute Indirection Register 1

The AMAIR1 characteristics are:

Purpose

When using the Long-descriptor format translation tables for stage 1 translations, provides IMPLEMENTATION DEFINED memory attributes for the memory regions specified by MAIR1.

Configurations

AArch32 System register AMAIR1[31:0] is architecturally mapped to AArch64 System register AMAIR_EL1[63:32].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to AMAIR1 are UNKNOWN.

When EL3 is using AArch32, write access to AMAIR1(S) is disabled when the CP15SDISABLE signal is asserted HIGH.

Attributes

AMAIR1 is a 32-bit register.

Field descriptions

The AMAIR1 bit assignments are:

31 … 0
IMPLEMENTATION DEFINED

This register is RES0 in the following cases:

- When an implementation does not provide any IMPLEMENTATION DEFINED memory attributes.
- When the Long-descriptor translation table format is not used.

If EL3 is implemented and is using AArch32:

- AMAIR1(S) gives the value for memory accesses from Secure state.
- AMAIR1(NS) gives the value for memory accesses from Non-secure states other than Hyp mode.

Any IMPLEMENTATION DEFINED memory attributes are additional qualifiers for the memory locations and must not change the architected behavior specified by MAIR0 and MAIR1.

In a typical implementation, AMAIR0 and AMAIR1 split into eight one-byte fields, corresponding to the MAIRn.Attr<n> fields, but the architecture does not require them to do so.

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

Accessing the AMAIR1

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1010	0b0011	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T10 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T10 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TRVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return AMAIR1_NS;
    else
        return AMAIR1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return AMAIR1_NS;
    else
        return AMAIR1;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return AMAIR1_S;
    else
        return AMAIR1_NS;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1010	0b0011	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T10 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T10 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        AMAIR1_NS = R[t];
    else
        AMAIR1 = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        AMAIR1_NS = R[t];
    else
        AMAIR1 = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' && CP15SDISABLE == HIGH then
        UNDEFINED;
    elsif SCR.NS == '0' && CP15SDISABLE2 == HIGH then
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
else
    if SCR.NS == '0' then
        AMAIR1_S = R[t];
    else
        AMAIR1_NS = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.8 APSR, Application Program Status Register

The APSR characteristics are:

Purpose

Hold program status and control information.

Configurations

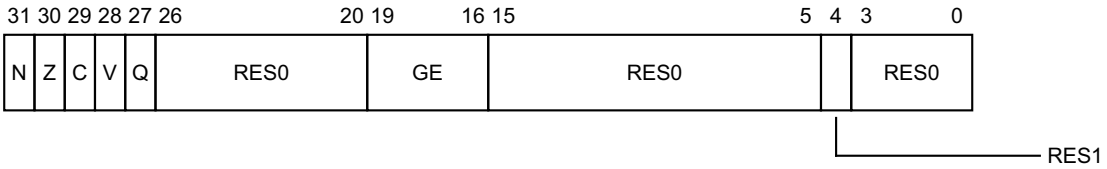
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to APSR are UNKNOWN.

Attributes

APSR is a 32-bit register.

Field descriptions

The APSR bit assignments are:



N, bit [31]

Negative condition flag. Set to bit[31] of the result of the last flag-setting instruction. If the result is regarded as a two's complement signed integer, then N is set to 1 if the result was negative, and N is set to 0 if the result was positive or zero.

Z, bit [30]

Zero condition flag. Set to 1 if the result of the last flag-setting instruction was zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.

C, bit [29]

Carry condition flag. Set to 1 if the last flag-setting instruction resulted in a carry condition, for example an unsigned overflow on an addition.

V, bit [28]

Overflow condition flag. Set to 1 if the last flag-setting instruction resulted in an overflow condition, for example a signed overflow on an addition.

Q, bit [27]

Cumulative saturation bit. Set to 1 to indicate that overflow or saturation occurred in some instructions.

Bits [26:20]

Reserved, RES0.

GE, bits [19:16]

Greater than or Equal flags, for parallel addition and subtraction.

Bits [15:5]

Reserved, RES0.

Bit [4]

Reserved, RES1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bits [3:0]

Reserved, RES0.

It is permitted that, on a read of APSR:

- Bit[22] returns the value of PSTATE.PAN
- Bit[9] returns the value of PSTATE.E.
- Bits[8:6] return the value of PSTATE.{A, I, F}, the mask bits.
- Bit[4:0] returns the value of PSTATE.M[4:0]

Note

This is an exception to the general rule that an UNKNOWN field must not return information that cannot be obtained, at the current Privilege level, by an architected mechanism.

For more information see *The Application Program Status Register, APSR* on page E1-3955.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.9 ATS12NSOPR, Address Translate Stages 1 and 2 Non-secure Only PL1 Read

The ATS12NSOPR characteristics are:

Purpose

Performs stage 1 and 2 address translations as defined for PL1 and the Non-secure state, with permissions as if reading from the given virtual address.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ATS12NSOPR are UNKNOWN.

Attributes

ATS12NSOPR is a 32-bit System instruction.

Field descriptions

The ATS12NSOPR input value bit assignments are:

31 … 0
Input address for translation

Bits [31:0]

Input address for translation. The resulting address can be read from the PAR.

This System instruction takes a VA as input. The resulting address is the PA that is the output address of the stage 2 translation.

Executing the ATS12NSOPR instruction

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1000	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif !ELUsingAArch32(EL2) && SCR_EL3.<NS,EEL2> == '01' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    ATS12NSOPR(R[t]);
elsif PSTATE.EL == EL3 then
    ATS12NSOPR(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.10 ATS12NSOPW, Address Translate Stages 1 and 2 Non-secure Only PL1 Write

The ATS12NSOPW characteristics are:

Purpose

Performs stage 1 and 2 address translations as defined for PL1 and the Non-secure state, with permissions as if writing to the given virtual address.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ATS12NSOPW are UNKNOWN.

Attributes

ATS12NSOPW is a 32-bit System instruction.

Field descriptions

The ATS12NSOPW input value bit assignments are:

31	0
Input address for translation	

Bits [31:0]

Input address for translation. The resulting address can be read from the PAR.

This System instruction takes a VA as input. The resulting address is the PA that is the output address of the stage 2 translation.

Executing the ATS12NSOPW instruction

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1000	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif !ELUsingAArch32(EL2) && SCR_EL3.<NS,EEL2> == '01' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    ATS12NSOPW(R[t]);
elsif PSTATE.EL == EL3 then
    ATS12NSOPW(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.11 ATS12NSOUR, Address Translate Stages 1 and 2 Non-secure Only Unprivileged Read

The ATS12NSOUR characteristics are:

Purpose

Performs stage 1 and 2 address translations as defined for PL0 and the Non-secure state, with permissions as if reading from the given virtual address.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ATS12NSOUR are UNKNOWN.

Attributes

ATS12NSOUR is a 32-bit System instruction.

Field descriptions

The ATS12NSOUR input value bit assignments are:

31 … 0
Input address for translation

Bits [31:0]

Input address for translation. The resulting address can be read from the PAR.

This System instruction takes a VA as input. The resulting address is the PA that is the output address of the stage 2 translation.

Executing the ATS12NSOUR instruction

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1000	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif !ELUsingAArch32(EL2) && SCR_EL3.<NS,EEL2> == '01' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    ATS12NSOUR(R[t]);
elsif PSTATE.EL == EL3 then
    ATS12NSOUR(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.12 ATS12NSOUW, Address Translate Stages 1 and 2 Non-secure Only Unprivileged Write

The ATS12NSOUW characteristics are:

Purpose

Performs stage 1 and 2 address translations as defined for PL0 and the Non-secure state, with permissions as if writing to the given virtual address.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ATS12NSOUW are UNKNOWN.

Attributes

ATS12NSOUW is a 32-bit System instruction.

Field descriptions

The ATS12NSOUW input value bit assignments are:

31	0
Input address for translation	

Bits [31:0]

Input address for translation. The resulting address can be read from the PAR.

This System instruction takes a VA as input. The resulting address is the PA that is the output address of the stage 2 translation.

Executing the ATS12NSOUW instruction

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1000	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif !ELUsingAArch32(EL2) && SCR_EL3.<NS,EEL2> == '01' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    ATS12NSOUW(R[t]);
elsif PSTATE.EL == EL3 then
    ATS12NSOUW(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.13 ATS1CPR, Address Translate Stage 1 Current state PL1 Read

The ATS1CPR characteristics are:

Purpose

Performs stage 1 address translation as defined for PL1 and the current Security state, with permissions as if reading from the given virtual address.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ATS1CPR are UNKNOWN.

Attributes

ATS1CPR is a 32-bit System instruction.

Field descriptions

The ATS1CPR input value bit assignments are:

31	0
Input address for translation	

Bits [31:0]

Input address for translation. The resulting address can be read from the PAR.

This System instruction takes a VA as input. If EL2 is implemented and enabled in the current Security state, the resulting address is the IPA that is the output address of the stage 1 translation. Otherwise, the resulting address is a PA.

Executing the ATS1CPR instruction

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        ATS1CPR(R[t]);
elsif PSTATE.EL == EL2 then
    ATS1CPR(R[t]);
elsif PSTATE.EL == EL3 then
    ATS1CPR(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.14 ATS1CPRP, Address Translate Stage 1 Current state PL1 Read PAN

The ATS1CPRP characteristics are:

Purpose

Performs a stage 1 address translation at PL1 and in the current Security state, where the value of PSTATE.PAN determines if a read from a location will generate a permission fault for a privileged access.

Configurations

This instruction is present only when AArch32 is supported at any Exception level and ARMv8.2-ATS1E1 is implemented. Otherwise, direct accesses to ATS1CPRP are UNDEFINED.

Attributes

ATS1CPRP is a 32-bit System instruction.

Field descriptions

The ATS1CPRP input value bit assignments are:

31	0
Input address for translation	

Bits [31:0]

Input address for translation. The resulting address can be read from the PAR.

This System instruction takes a VA as input. If EL2 is implemented and enabled in the current Security state, the resulting address is the IPA that is the output address of the stage 1 translation. Otherwise, the resulting address is a PA.

Executing the ATS1CPRP instruction

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        ATS1CPRP(R[t]);
elsif PSTATE.EL == EL2 then
    ATS1CPRP(R[t]);
elsif PSTATE.EL == EL3 then
    ATS1CPRP(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.15 ATS1CPW, Address Translate Stage 1 Current state PL1 Write

The ATS1CPW characteristics are:

Purpose

Performs stage 1 address translation as defined for PL1 and the current Security state, with permissions as if writing to the given virtual address.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ATS1CPW are UNKNOWN.

Attributes

ATS1CPW is a 32-bit System instruction.

Field descriptions

The ATS1CPW input value bit assignments are:

31 … 0
Input address for translation

Bits [31:0]

Input address for translation. The resulting address can be read from the PAR.

This System instruction takes a VA as input. If EL2 is implemented and enabled in the current Security state, the resulting address is the IPA that is the output address of the stage 1 translation. Otherwise, the resulting address is a PA.

Executing the ATS1CPW instruction

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        ATS1CPW(R[t]);
elsif PSTATE.EL == EL2 then
    ATS1CPW(R[t]);
elsif PSTATE.EL == EL3 then
    ATS1CPW(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.16 ATS1CPWP, Address Translate Stage 1 Current state PL1 Write PAN

The ATS1CPWP characteristics are:

Purpose

When ARMv8.2-ATS1E1 is implemented, performs a stage 1 address translation at PL1 and in the current Security state, where the value of PSTATE.PAN determines if a write to the location will generate a permission fault for a privileged access.

Configurations

This instruction is present only when AArch32 is supported at any Exception level and ARMv8.2-ATS1E1 is implemented. Otherwise, direct accesses to ATS1CPWP are UNDEFINED.

Attributes

ATS1CPWP is a 32-bit System instruction.

Field descriptions

The ATS1CPWP input value bit assignments are:

31	0
Input address for translation	

Bits [31:0]

Input address for translation. The resulting address can be read from the PAR.

This System instruction takes a VA as input. If EL2 is implemented and enabled in the current Security state, the resulting address is the IPA that is the output address of the stage 1 translation. Otherwise, the resulting address is a PA.

Executing the ATS1CPWP instruction

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        ATS1CPWP(R[t]);
elsif PSTATE.EL == EL2 then
    ATS1CPWP(R[t]);
elsif PSTATE.EL == EL3 then
    ATS1CPWP(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.17 ATS1CUR, Address Translate Stage 1 Current state Unprivileged Read

The ATS1CUR characteristics are:

Purpose

Performs stage 1 address translation as defined for PL0 and the current Security state, with permissions as if reading from the given virtual address.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ATS1CUR are UNKNOWN.

Attributes

ATS1CUR is a 32-bit System instruction.

Field descriptions

The ATS1CUR input value bit assignments are:

31	0
Input address for translation	

Bits [31:0]

Input address for translation. The resulting address can be read from the PAR.

This System instruction takes a VA as input. If EL2 is implemented and enabled in the current Security state, the resulting address is the IPA that is the output address of the stage 1 translation. Otherwise, the resulting address is a PA.

Executing the ATS1CUR instruction

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        ATS1CUR(R[t]);
elsif PSTATE.EL == EL2 then
    ATS1CUR(R[t]);
elsif PSTATE.EL == EL3 then
    ATS1CUR(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.18 ATS1CUW, Address Translate Stage 1 Current state Unprivileged Write

The ATS1CUW characteristics are:

Purpose

Performs stage 1 address translation as defined for PL0 and the current Security state, with permissions as if writing to the given virtual address.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ATS1CUW are UNKNOWN.

Attributes

ATS1CUW is a 32-bit System instruction.

Field descriptions

The ATS1CUW input value bit assignments are:

31	0
Input address for translation	

Bits [31:0]

Input address for translation. The resulting address can be read from the PAR.

This System instruction takes a VA as input. If EL2 is implemented and enabled in the current Security state, the resulting address is the IPA that is the output address of the stage 1 translation. Otherwise, the resulting address is a PA.

Executing the ATS1CUW instruction

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1000	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        ATS1CUW(R[t]);
elsif PSTATE.EL == EL2 then
    ATS1CUW(R[t]);
elsif PSTATE.EL == EL3 then
    ATS1CUW(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.19 ATS1HR, Address Translate Stage 1 Hyp mode Read

The ATS1HR characteristics are:

Purpose

Performs stage 1 address translation as defined for PL2 and the Non-secure state, with permissions as if reading from the given virtual address.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ATS1HR are UNKNOWN.

Attributes

ATS1HR is a 32-bit System instruction.

Field descriptions

The ATS1HR input value bit assignments are:

31	0
Input address for translation	

Bits [31:0]

Input address for translation. The resulting address can be read from the PAR.

This System instruction takes a VA as input. The resulting address is the PA that is the output address of the translation.

Executing the ATS1HR instruction

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0111	0b1000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        ATS1HR(R[t]);
elsif PSTATE.EL == EL2 then
    ATS1HR(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    ATS1HR(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.20 ATS1HW, Address Translate Stage 1 Hyp mode Write

The ATS1HW characteristics are:

Purpose

Performs stage 1 address translation as defined for PL2 and the Non-secure state, with permissions as if writing to the given virtual address.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ATS1HW are UNKNOWN.

Attributes

ATS1HW is a 32-bit System instruction.

Field descriptions

The ATS1HW input value bit assignments are:

31	0
Input address for translation	

Bits [31:0]

Input address for translation. The resulting address can be read from the PAR.

This System instruction takes a VA as input. The resulting address is the PA that is the output address of the translation.

Executing the ATS1HW instruction

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0111	0b1000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        ATS1HW(R[t]);
elsif PSTATE.EL == EL2 then
    ATS1HW(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    ATS1HW(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.21 BPIALL, Branch Predictor Invalidate All

The BPIALL characteristics are:

Purpose

Invalidate all entries from branch predictors.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to BPIALL are UNKNOWN.

In an implementation where the branch predictors are architecturally invisible, this instruction can execute as a NOP.

Attributes

BPIALL is a 32-bit System instruction.

Field descriptions

BPIALL ignores the value in the register specified by the instruction. Software does not have to write a value to the register before issuing this instruction.

Executing the BPIALL instruction

The PE ignores the value of <Rt>. Software does not have to write a value to this register before issuing this instruction.

When HCR.FB is 1, at Non-secure EL1 this instruction executes as a BPIALLIS.

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0101	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.FB == '1' then
        BPIALLIS();
    else
        BPIALL();
elsif PSTATE.EL == EL2 then
    BPIALL();
elsif PSTATE.EL == EL3 then
    BPIALL();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.22 BPIALLIS, Branch Predictor Invalidate All, Inner Shareable

The BPIALLIS characteristics are:

Purpose

Invalidate all entries from branch predictors Inner Shareable.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to BPIALLIS are UNKNOWN.

In an implementation where the branch predictors are architecturally invisible, this instruction can execute as a NOP.

Attributes

BPIALLIS is a 32-bit System instruction.

Field descriptions

BPIALLIS ignores the value in the register specified by the instruction. Software does not have to write a value to the register before issuing this instruction.

Executing the BPIALLIS instruction

The PE ignores the value of <Rt>. Software does not have to write a value to this register before issuing this instruction.

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0001	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        BPIALL();
elsif PSTATE.EL == EL2 then
    BPIALL();
elsif PSTATE.EL == EL3 then
    BPIALL();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.23 BPIMVA, Branch Predictor Invalidate by VA

The BPIMVA characteristics are:

Purpose

Invalidate virtual address from branch predictors.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to BPIMVA are UNKNOWN.

In an implementation where the branch predictors are architecturally invisible, this instruction can execute as a NOP.

Attributes

BPIMVA is a 32-bit System instruction.

Field descriptions

The BPIMVA input value bit assignments are:

31 … 0
Virtual address to use

Bits [31:0]

Virtual address to use.

Executing the BPIMVA instruction

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0101	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        BPIMVA(R[t]);
elsif PSTATE.EL == EL2 then
    BPIMVA(R[t]);
elsif PSTATE.EL == EL3 then
    BPIMVA(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.24 CCSIDR, Current Cache Size ID Register

The CCSIDR characteristics are:

Purpose

Provides information about the architecture of the currently selected cache.

When ARMv8.3-CCIDX is implemented, this register is used in conjunction with CCSIDR2.

Configurations

AArch32 System register CCSIDR[31:0] is architecturally mapped to AArch64 System register CCSIDR_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to CCSIDR are UNKNOWN.

The implementation includes one CCSIDR for each cache that it can access. CSSELR and the Security state select which Cache Size ID Register is accessible.

Attributes

CCSIDR is a 32-bit register.

Field descriptions

The CCSIDR bit assignments are:

When ARMv8.3-CCIDX is implemented:

31 – 24	23 – 3	2 – 0
RES0	Associativity	LineSize

Note

The parameters NumSets, Associativity, and LineSize in these registers define the architecturally visible parameters that are required for the cache maintenance by Set/Way instructions. They are not guaranteed to represent the actual microarchitectural features of a design. You cannot make any inference about the actual sizes of caches based on these parameters.

Bits [31:24]

Reserved, RES0.

Associativity, bits [23:3]

(Associativity of cache) - 1, therefore a value of 0 indicates an associativity of 1. The associativity does not have to be a power of 2.

LineSize, bits [2:0]

(Log_2(Number of bytes in cache line)) - 4. For example:

For a line length of 16 bytes: $Log_2(16) = 4$, LineSize entry = 0. This is the minimum line length.

For a line length of 32 bytes: $Log_2(32) = 5$, LineSize entry = 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

31 – 28	27 – 13	12 – 3	2 – 0
UNKNOWN	NumSets	Associativity	LineSize

Note

The parameters NumSets, Associativity, and LineSize in these registers define the architecturally visible parameters that are required for the cache maintenance by Set/Way instructions. They are not guaranteed to represent the actual microarchitectural features of a design. You cannot make any inference about the actual sizes of caches based on these parameters.

UNKNOWN, bits [31:28]

Reserved, UNKNOWN.

NumSets, bits [27:13]

(Number of sets in cache) - 1, therefore a value of 0 indicates 1 set in the cache. The number of sets does not have to be a power of 2.

Associativity, bits [12:3]

(Associativity of cache) - 1, therefore a value of 0 indicates an associativity of 1. The associativity does not have to be a power of 2.

LineSize, bits [2:0]

(Log_2(Number of bytes in cache line)) - 4. For example:

For a line length of 16 bytes: $Log_2(16) = 4$, LineSize entry = 0. This is the minimum line length.

For a line length of 32 bytes: $Log_2(32) = 5$, LineSize entry = 1.

Accessing the CCSIDR

If CSSELR.Level is programmed to a cache level that is not implemented, then on a read of the CCSIDR the behavior is CONSTRAINED UNPREDICTABLE, and can be one of the following:

- The CCSIDR read is treated as NOP.
- The CCSIDR read is UNDEFINED.
- The CCSIDR read returns an UNKNOWN value.

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b001	0b0000	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID4 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID2 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TID4 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        return CCSIDR;
elsif PSTATE.EL == EL2 then
    return CCSIDR;
elsif PSTATE.EL == EL3 then
    return CCSIDR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.25 CCSIDR2, Current Cache Size ID Register 2

The CCSIDR2 characteristics are:

Purpose

When ARMv8.3-CCIDX is implemented, in conjunction with CCSIDR, provides information about the architecture of the currently selected cache.

When ARMv8.3-CCIDX is not implemented, this register is not implemented.

Configurations

AArch32 System register CCSIDR2[31:0] is architecturally mapped to AArch64 System register CCSIDR2_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level and ARMv8.3-CCIDX is implemented. Otherwise, direct accesses to CCSIDR2 are UNDEFINED.

The implementation includes one CCSIDR2 for each cache that it can access. CSSELR and the Security state select which Cache Size ID Register is accessible.

Attributes

CCSIDR2 is a 32-bit register.

Field descriptions

The CCSIDR2 bit assignments are:

31 24	23 0
RES0	NumSets

Bits [31:24]

Reserved, RES0.

NumSets, bits [23:0]

(Number of sets in cache) - 1, therefore a value of 0 indicates 1 set in the cache. The number of sets does not have to be a power of 2.

Accessing the CCSIDR2

If CSSELR.Level is programmed to a cache level that is not implemented, then on a read of the CCSIDR2 the behavior is CONSTRAINED UNPREDICTABLE, and can be one of the following:

- The CCSIDR2 read is treated as NOP.
- The CCSIDR2 read is UNDEFINED.
- The CCSIDR2 read returns an UNKNOWN value.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b001	0b0000	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID4 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID2 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TID4 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        return CCSIDR2;
elsif PSTATE.EL == EL2 then
    return CCSIDR2;
elsif PSTATE.EL == EL3 then
    return CCSIDR2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.26 CFPRCTX, Control Flow Prediction Restriction by Context

The CFPRCTX characteristics are:

Purpose

Control Flow Prediction Restriction by Context applies to all Control Flow Prediction Resources that predict execution based on information gathered within the target execution context or contexts.

When this instruction is complete and synchronized, control flow prediction does not permit later speculative execution within the target execution context to be observable through side channels.

This instruction is guaranteed to be complete following a DSB that covers both read and write behavior on the same PE as executed the original restriction instruction, and a subsequent context synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

Note

This instruction does not require the invalidation of prediction structures so long as the behavior described for completion of this instruction is met by the implementation.

On some implementations the instruction is likely to take a significant number of cycles to execute. This instruction is expected to be used very rarely, such as on the roll-over of an ASID or VMID, but should not be used on every context switch.

Configurations

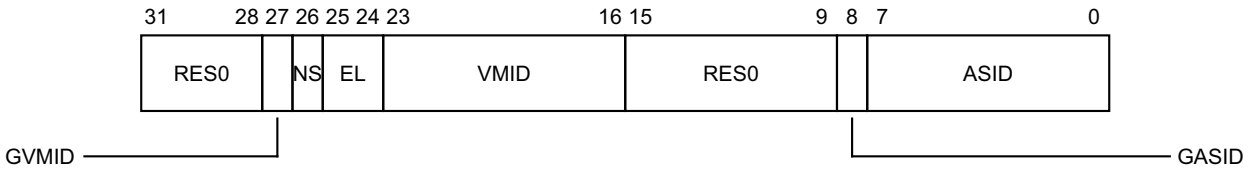
This instruction is present only when AArch32 is supported at any Exception level and ARMv8.0-PredInv is implemented. Otherwise, direct accesses to CFPRCTX are UNDEFINED.

Attributes

CFPRCTX is a 32-bit System instruction.

Field descriptions

The CFPRCTX input value bit assignments are:



Bits [31:28]

Reserved, RES0.

GVMID, bit [27]

Execution of this instruction applies to all VMIDs or a specified VMID.

`0b0` Applies to specified VMID for an EL0 or EL1 context. For all other contexts this field is RES0.

`0b1` Applies to all VMIDs for an EL0 or EL1 context. For all other contexts this field is RES0.

If the instruction is executed at EL0 or EL1, then this field has an Effective value of 0.

NS, bit [26]

Security State.

`0b0` Secure state.

`0b1` Non-secure state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the instruction is executed in Non-secure state, this field has an Effective value of 1.

EL, bits [25:24]

Exception Level.

`0b00` EL0.

`0b01` EL1.

`0b10` EL2.

`0b11` EL3.

If the instruction is executed at an exception level lower than the specified level, this instruction is treated as a NOP.

VMID, bits [23:16]

Only applies when bit[27] is 0 and either of:

- an EL1 context.
- an EL0 context when (HCR_EL2.E2H==0 or HCR_EL2.TGE==0) or EL2 is using AArch32 state.

Otherwise this field is RES0.

When the instruction is executed at EL1 then this field is treated as the current VMID.

When the instruction is executed at EL0 and (HCR_EL2.E2H==0 or HCR_EL2.TGE==0 or ELUsingAArch32(EL2)) then this field is treated as the current VMID.

When the instruction is executed at EL0 and (HCR_EL2.E2H==1 and HCR_EL2.TGE==1 and !ELUsingAArch32(EL2)) then this field is ignored.

Bits [15:9]

Reserved, RES0.

GASID, bit [8]

Execution of this instruction applies to all ASIDs or a specified ASID.

`0b0` Applies to specified ASID for an EL0 context. For all other contexts this field is RES0.

`0b1` Applies to all ASID for an EL0 context. For all other contexts this field is RES0.

If the instruction is executed at EL0, then this field is treated as 0.

ASID, bits [7:0]

Only applies for an EL0 context and when bit[8] is 0.

Otherwise this field is RES0.

When the instruction is executed at EL0 then this field is treated as the current ASID.

Executing the CFPRCTX instruction

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0011	0b100

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.EnRCTX == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && SCTLR.EnRCTX == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.EnRCTX == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        CFPRCTX(R[t]);
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x03);
    else
        CFPRCTX(R[t]);
elsif PSTATE.EL == EL2 then
    CFPRCTX(R[t]);
elsif PSTATE.EL == EL3 then
    CFPRCTX(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.27 CLIDR, Cache Level ID Register

The CLIDR characteristics are:

Purpose

Identifies the type of cache, or caches, that are implemented at each level and can be managed using the architected cache maintenance instructions that operate by set/way, up to a maximum of seven levels. Also identifies the Level of Coherence (LoC) and Level of Unification (LoU) for the cache hierarchy.

Configurations

AArch32 System register CLIDR[31:0] is architecturally mapped to AArch64 System register CLIDR_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to CLIDR are UNKNOWN.

Attributes

CLIDR is a 32-bit register.

Field descriptions

The CLIDR bit assignments are:

31 30	29 27	26 24	23 21	20 18	17 15	14 12	11 9	8 6	5 3	2 0
ICB	LoUU	LoC	LoUIS	Ctype7	Ctype6	Ctype5	Ctype4	Ctype3	Ctype2	Ctype1

ICB, bits [31:30]

Inner cache boundary. This field indicates the boundary for caching Inner Cacheable memory regions.

The possible values are:

`0b00` Not disclosed by this mechanism.

`0b01` L1 cache is the highest Inner Cacheable level.

`0b10` L2 cache is the highest Inner Cacheable level.

`0b11` L3 cache is the highest Inner Cacheable level.

LoUU, bits [29:27]

Level of Unification Uniprocessor for the cache hierarchy.

LoC, bits [26:24]

Level of Coherence for the cache hierarchy.

LoUIS, bits [23:21]

Level of Unification Inner Shareable for the cache hierarchy.

Ctype<n>, bits [3(n-1)+2:3(n-1)], for n = 1 to 7

Cache Type fields. Indicate the type of cache that is implemented and can be managed using the architected cache maintenance instructions that operate by set/way at each level, from Level 1 up to a maximum of seven levels of cache hierarchy. Possible values of each field are:

`0b000` No cache.

`0b001` Instruction cache only.

`0b010` Data cache only.

`0b011` Separate instruction and data caches.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b100` Unified cache.

All other values are reserved.

If software reads the Cache Type fields from Ctype1 upwards, once it has seen a value of 000, no caches that can be managed using the architected cache maintenance instructions that operate by set/way exist at further-out levels of the hierarchy. So, for example, if Ctype3 is the first Cache Type field with a value of 000, the values of Ctype4 to Ctype7 must be ignored.

Accessing the CLIDR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b001	0b0000	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID4 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID2 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TID4 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        return CLIDR;
elsif PSTATE.EL == EL2 then
    return CLIDR;
elsif PSTATE.EL == EL3 then
    return CLIDR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.28 CONTEXTIDR, Context ID Register

The CONTEXTIDR characteristics are:

Purpose

Identifies the current Process Identifier and, when using the Short-descriptor translation table format, the Address Space Identifier.

The value of the whole of this register is called the Context ID and is used by:

- The debug logic, for Linked and Unlinked Context ID matching.
- The trace logic, to identify the current process.

The significance of this register is for debug and trace use only.

Configurations

AArch32 System register CONTEXTIDR[31:0] is architecturally mapped to AArch64 System register CONTEXTIDR_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to CONTEXTIDR are UNKNOWN.

The register format depends on whether address translation is using the Long-descriptor or the Short-descriptor translation table format.

Attributes

CONTEXTIDR is a 32-bit register.

Field descriptions

The CONTEXTIDR bit assignments are:

When TTBCR.EAE == 0:

31 … 8	7 … 0
PROCID	ASID

PROCID, bits [31:8]

Process Identifier. This field must be programmed with a unique value that identifies the current process.

This field resets to an architecturally UNKNOWN value.

ASID, bits [7:0]

Address Space Identifier. This field is programmed with the value of the current ASID.

This field resets to an architecturally UNKNOWN value.

When TTBCR.EAE == 1:

31 … 0
PROCID

PROCID, bits [31:0]

Process Identifier. This field must be programmed with a unique value that identifies the current process.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

Accessing the CONTEXTIDR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TRVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return CONTEXTIDR_NS;
    else
        return CONTEXTIDR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return CONTEXTIDR_NS;
    else
        return CONTEXTIDR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return CONTEXTIDR_S;
    else
        return CONTEXTIDR_NS;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        CONTEXTIDR_NS = R[t];
    else
        CONTEXTIDR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        CONTEXTIDR_NS = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        else
            CONTEXTIDR = R[t];
    elsif PSTATE.EL == EL3 then
        if SCR.NS == '0' then
            CONTEXTIDR_S = R[t];
        else
            CONTEXTIDR_NS = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.29 CP15DMB, Data Memory Barrier System instruction

The CP15DMB characteristics are:

Purpose

Performs a Data Memory Barrier.

Arm deprecates any use of this System instruction, and strongly recommends that software use the DMB instruction instead.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to CP15DMB are UNKNOWN.

Attributes

CP15DMB is a 32-bit System instruction.

Field descriptions

CP15DMB ignores the value in the register specified by the instruction. Software does not have to write a value to the register before issuing this instruction.

Executing the CP15DMB instruction

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1010	0b101

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.CP15BEN == '0'
then
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.CP15BEN == '0'
then
        UNDEFINED;
    elsif ELUsingAArch32(EL1) && SCTLR.CP15BEN == '0' then
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        CP15DMB();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif ELUsingAArch32(EL1) && SCTLR.CP15BEN == '0' then
        UNDEFINED;
    else
        CP15DMB();
elsif PSTATE.EL == EL2 then
    if HSCTLR.CP15BEN == '0' then
        UNDEFINED;
    else
        CP15DMB();
elsif PSTATE.EL == EL3 then
    if SCTLR.CP15BEN == '0' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            UNDEFINED;
        else
            CP15DMB();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.30 CP15DSB, Data Synchronization Barrier System instruction

The CP15DSB characteristics are:

Purpose

Performs a Data Synchronization Barrier.

Arm deprecates any use of this System instruction, and strongly recommends that software use the DSB instruction instead.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to CP15DSB are UNKNOWN.

Attributes

CP15DSB is a 32-bit System instruction.

Field descriptions

CP15DSB ignores the value in the register specified by the instruction. Software does not have to write a value to the register before issuing this instruction.

Executing the CP15DSB instruction

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1010	0b100

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.CP15BEN == '0'
then
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.CP15BEN == '0'
then
        UNDEFINED;
    elsif ELUsingAArch32(EL1) && SCTLR.CP15BEN == '0' then
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        CP15DSB();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif ELUsingAArch32(EL1) && SCTLR.CP15BEN == '0' then
        UNDEFINED;
    else
        CP15DSB();
elsif PSTATE.EL == EL2 then
    if HSCTLR.CP15BEN == '0' then
        UNDEFINED;
    else
        CP15DSB();
elsif PSTATE.EL == EL3 then
    if SCTLR.CP15BEN == '0' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        UNDEFINED;
    else
        CP15DSB();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.31 CP15ISB, Instruction Synchronization Barrier System instruction

The CP15ISB characteristics are:

Purpose

Performs an Instruction Synchronization Barrier.

Arm deprecates any use of this System instruction, and strongly recommends that software use the ISB instruction instead.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to CP15ISB are UNKNOWN.

Attributes

CP15ISB is a 32-bit System instruction.

Field descriptions

CP15ISB ignores the value in the register specified by the instruction. Software does not have to write a value to the register before issuing this instruction.

Executing the CP15ISB instruction

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0101	0b100

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.CP15BEN == '0'
then
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.CP15BEN == '0'
then
        UNDEFINED;
    elsif ELUsingAArch32(EL1) && SCTLR.CP15BEN == '0' then
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        CP15ISB();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif ELUsingAArch32(EL1) && SCTLR.CP15BEN == '0' then
        UNDEFINED;
    else
        CP15ISB();
elsif PSTATE.EL == EL2 then
    if HSCTLR.CP15BEN == '0' then
        UNDEFINED;
    else
        CP15ISB();
elsif PSTATE.EL == EL3 then
    if SCTLR.CP15BEN == '0' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        UNDEFINED;
    else
        CP15ISB();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.32 CPACR, Architectural Feature Access Control Register

The CPACR characteristics are:

Purpose

Controls access to trace, and to Advanced SIMD and floating-point functionality from EL0, EL1, and EL3.

In an implementation that includes EL2, the CPACR has no effect on instructions executed at EL2.

Configurations

AArch32 System register CPACR[31:0] is architecturally mapped to AArch64 System register CPACR_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to CPACR are UNKNOWN.

Bits in the NSACR control Non-secure access to the CPACR fields. See the field descriptions for more information.

Note

In the register field descriptions, controls are described as applying at specified Privilege levels. This is because, in Secure state, a PL1 control:

- Applies to execution in a Secure EL3 mode when EL3 is using AArch32.
- Applies to execution in a Secure EL1 mode when EL3 is using AArch64.

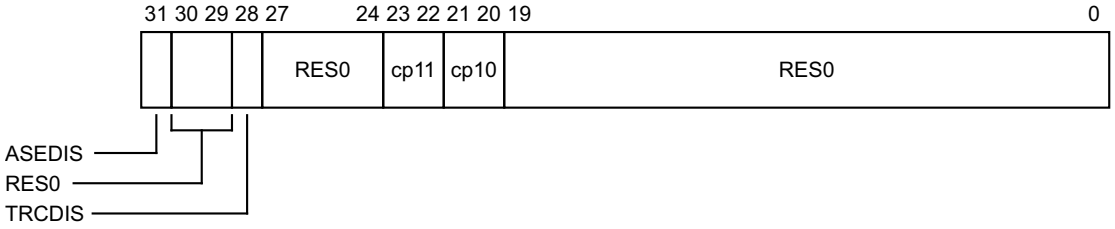
See *Security state, Exception levels, and AArch32 execution privilege* on page G1-5682.

Attributes

CPACR is a 32-bit register.

Field descriptions

The CPACR bit assignments are:



ASEDIS, bit [31]

Disables PL0 and PL1 execution of Advanced SIMD instructions.

`0b0` This control permits execution of Advanced SIMD instructions at PL0 and PL1.

`0b1` All instruction encodings that are Advanced SIMD instruction encodings, but are not also floating-point instruction encodings, are UNDEFINED at PL0 and PL1.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES0. Otherwise, it is IMPLEMENTATION DEFINED whether this field is implemented as a RW field. If it is not implemented as a RW field, it is RAZ/WI.

If EL3 is implemented and is using AArch32, and the value of NSACR.NSASEDIS is 1, this field behaves as RAO/WI in Non-secure state, regardless of its actual value. This applies even if the field is implemented as RAZ/WI.

For the list of instructions affected by this field, see *Controls of Advanced SIMD operation that do not apply to floating-point operation* on page E1-3966.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

See the description of CPACR.cp10 for a list of other controls that can disable or trap execution of Advanced SIMD instructions in AArch32 state.

This field resets to `0`.

Bits [30:29]

Reserved, RES0.

TRCDIS, bit [28]

Traps PL0 and PL1 System register accesses to all implemented trace registers to Undefined mode.

`0b0` This control has no effect on PL0 and PL1 System register accesses to trace registers.

`0b1` PL0 and PL1 System register accesses to all implemented trace registers are trapped to Undefined mode.

If the implementation does not include a PE trace unit, or does not include a System register interface to the PE trace unit registers, this field is RES0. Otherwise, it is IMPLEMENTATION DEFINED whether this field is implemented as a RW field. If it is not implemented as a RW field, it is RAZ/WI.

If EL3 is implemented and is using AArch32, and the value of NSACR.NSTRCDIS is 1, this field behaves as RAO/WI in Non-secure state, regardless of its actual value. This applies even if the field is implemented as RAZ/WI.

Note

- The ETMv4 architecture does not permit EL0 to access the trace registers. If the implementation includes an ETMv4 implementation, EL0 accesses to the trace registers are UNDEFINED.
- The architecture does not provide traps on trace register accesses through the optional memory-mapped external debug interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

This field resets to an architecturally UNKNOWN value.

Bits [27:24]

Reserved, RES0.

cp11, bits [23:22]

The value of this field is ignored. If this field is programmed with a different value to the cp10 field then this field is UNKNOWN on a direct read of the CPACR.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES0.

In Non-secure state, if EL3 is implemented and is using AArch32, when the value of NSACR.cp10 is 0, this field behaves as RAZ/WI, regardless of its actual value.

This field resets to `0`.

cp10, bits [21:20]

Defines the access rights for the floating-point and Advanced SIMD functionality. Possible values of the field are:

`0b00` PL0 and PL1 accesses to floating-point and Advanced SIMD registers or instructions are UNDEFINED.

`0b01` PL0 accesses to floating-point and Advanced SIMD registers or instructions are UNDEFINED.

`0b10` Reserved. The effect of programming this field to this value is CONSTRAINED UNPREDICTABLE. See *Handling of System register control fields for Advanced SIMD and floating-point operation* on page K1-7836.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b11` This control permits full access to the floating-point and Advanced SIMD functionality from PL0 and PL1.

The floating-point and Advanced SIMD features controlled by these fields are:

- Execution of any floating-point or Advanced SIMD instruction.
- Any access to the Advanced SIMD and floating-point registers D0-D31 and their views as S0-S31 and Q0-Q15.
- Any access to the FPSCR, FPSID, MVFR0, MVFR1, MVFR2, or FPEXC System registers.

Note

The CPACR has no effect on floating-point and Advanced SIMD accesses from PL2. These can be disabled by the HCPTR.TCP10 field.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES0.

In Non-secure state, if EL3 is implemented and is using AArch32, when the value of NSACR.cp10 is 0, this field behaves as RAZ/WI, regardless of its actual value.

Execution of floating-point and Advanced SIMD instructions in AArch32 state can be disabled or trapped by the following controls:

- CPACR.cp10, or, if executing at EL0, CPACR_EL1.FPEN.
- FPEXC.EN.
- If executing in Non-secure state:
  - HCPTR.TCP10, or if EL2 is using AArch64, CPTR_EL2.TFP.
  - NSACR.cp10, or if EL3 is using AArch64, CPTR_EL3.TFP.
- For Advanced SIMD instructions only:
  - CPACR.ASEDIS.
  - If executing in Non-secure state, HCPTR.TASE and NSACR.NSTRCDIS.

See the descriptions of the controls for more information.

This field resets to `0`.

Bits [19:0]

Reserved, RES0.

Accessing the CPACR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TCPAC == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TCPAC == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return CPACR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return CPACR;
elsif PSTATE.EL == EL3 then
    return CPACR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TCPAC == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TCPAC == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        CPACR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        CPACR = R[t];
elsif PSTATE.EL == EL3 then
    CPACR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.33 CPSR, Current Program Status Register

The CPSR characteristics are:

Purpose

Holds PE status and control information.

Configurations

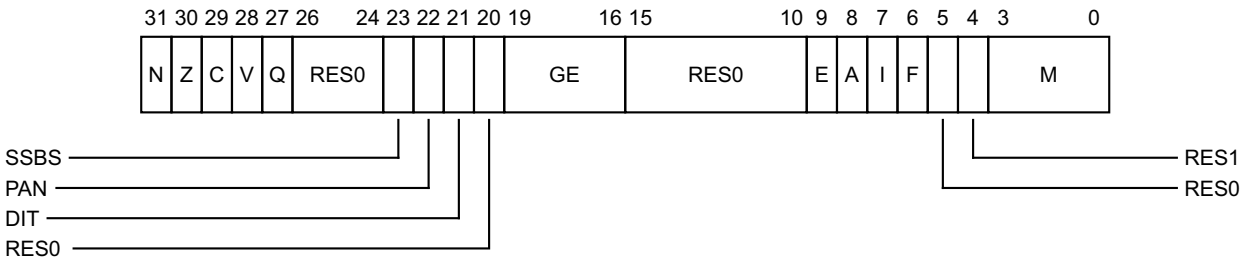
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to CPSR are UNKNOWN.

Attributes

CPSR is a 32-bit register.

Field descriptions

The CPSR bit assignments are:



N, bit [31]

Negative condition flag. Set to bit[31] of the result of the last flag-setting instruction. If the result is regarded as a two's complement signed integer, then N is set to 1 if the result was negative, and N is set to 0 if the result was positive or zero.

Z, bit [30]

Zero condition flag. Set to 1 if the result of the last flag-setting instruction was zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.

C, bit [29]

Carry condition flag. Set to 1 if the last flag-setting instruction resulted in a carry condition, for example an unsigned overflow on an addition.

V, bit [28]

Overflow condition flag. Set to 1 if the last flag-setting instruction resulted in an overflow condition, for example a signed overflow on an addition.

Q, bit [27]

Cumulative saturation bit. Set to 1 to indicate that overflow or saturation occurred in some instructions.

Bits [26:24]

Reserved, RES0.

SSBS, bit [23]

When ARMv8.0-SSBS is implemented:

Speculative Store Bypass Safe.

Prohibits speculative loads or stores which might practically allow a cache timing side channel.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A cache timing side channel might be exploited where a load or store uses an address that is derived from a register that is being loaded from memory using a load instruction speculatively read from a memory location. If PSTATE.SSBS is enabled, the address derived from the load instruction might be from earlier in the coherence order than the latest store to that memory location with the same virtual address.

`0b0` Hardware is not permitted to load or store speculatively in the manner described.

`0b1` Hardware is permitted to load or store speculatively in the manner described.

The value of this bit is usually set to the value described by the SCTLR.DSSBS bit on exceptions to any mode except Hyp mode, and the value described by HSCTLR.DSSBS on exceptions to Hyp mode.

This field resets to an IMPLEMENTATION DEFINED value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When ARMv8.1-PAN is implemented:

Privileged Access Never.

`0b0` The translation system is the same as Armv8.0.

`0b1` Disables privileged read and write accesses to addresses accessible at EL0.

The value of this bit is usually preserved on taking an exception, except in the following situations:

- When the target of the exception is EL1, and the value of the SCTLR.SPAN bit for the current Security state is 0, this bit is set to 1.
- When the target of the exception is EL3, from Secure state, and the value of the Secure SCTLR.SPAN is 0, this bit is set to 1.
- When the target of the exception is EL3, from Non-secure state, this bit is set to 0 regardless of the value of the Secure SCTLR.SPAN bit.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When ARMv8.4-DIT is implemented:

Data Independent Timing.

`0b0` The architecture makes no statement about the timing properties of any instructions.

`0b1` The architecture requires that:

- The timing of every load and store instruction is insensitive to the value of the data being loaded or stored.
- For certain data processing instructions, the instruction takes a time which is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- For certain data processing instructions, the response of the instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

The data processing instructions affected by this bit are:

- All cryptographic instructions. These instructions are:
  - `AESD`, `AESE`, `AESIMC`, `AESMC`, `SHA1C`, `SHA1H`, `SHA1M`, `SHA1P`, `SHA1SU0`, `SHA1SU1`, `SHA256H`, `SHA256H2`, `SHA256SU0`, and `SHA256SU1`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- A subset of those instructions which use the general-purpose register file. For these instructions, the effects of CPSR.DIT apply only if they do not use R15 as either their source or destination and pass their condition execution check. The instructions are:
  - BFI, BFC, CLZ, CMN, CMP, MLA, MLAS, MLS, MOVT, MUL, MULS, NOP, PKHBT, PKHTB, RBIT, REV, REV16, REVSH, RRX, SADD16, SADD8, SASX, SBFX, SHADD16, SHADD8, SHASX, SHSAX, SHSUB16, SHSUB8, SMLAL**, SMLAW*, SMLSD*, SMMLA*, SMMLS*, SMMUL*, SMUAD*, SMUL*, SSAX, SSUB16, SSUB8, SXTAB*, SXTAH, SXTB*, SXTH, TEQ, TST, UADD*, UASX, UBFX, UHADD*, UHASX, UHSAX, UHSUB*, UMAAL, UMLAL, UMLALS, UMULL, UMULLS, USADA8, USAX, USUB*, UXTAB*, UXTAH, UXTB*, UXTH, ADC (register-shifted register), ADCS (register-shifted register), ADD (register-shifted register), ADDS (register-shifted register), AND (register-shifted register), ANDS (register-shifted register), ASR (register-shifted register), ASRS (register-shifted register), BIC (register-shifted register), BICS (register-shifted register), EOR (register-shifted register), EORS (register-shifted register), LSL (register-shifted register), LSLS (register-shifted register), LSR (register-shifted register), LSRS (register-shifted register), MOV (register-shifted register), MOVS (register-shifted register), MVN (register-shifted register), MVNS (register-shifted register), ORR (register-shifted register), ORRS (register-shifted register), ROR (register-shifted register), RORS (register-shifted register), RSB (register-shifted register), RSBS (register-shifted register), RSC (register-shifted register), RSCS (register-shifted register), SBC (register-shifted register), SBCS (register-shifted register), SUB (register-shifted register), and SUBS (register-shifted register).
- A subset of those instructions which use the general-purpose register file. For these instructions, the effects of CPSR.DIT apply only if they do not use R15 as either their source or destination. The effects of CPSR.DIT do not depend on these instructions passing their condition execution check. These instructions are:
  - ADC (immediate), ADC (register), ADCS (immediate), ADCS (register), ADD (immediate), ADD (register), ADDS (immediate), ADDS (register), AND (immediate), AND (register), ANDS (immediate), ANDS (register), ASR (immediate), ASR (register), ASRS (immediate), ASRS (register), BIC (immediate), BIC (register), BICS (immediate), BICS (register), EOR (immediate), EOR (register), EORS (immediate), EORS (register), LSL (immediate), LSL (register), LSLS (immediate), LSLS (register), LSR (immediate), LSR (register), LSRS (immediate), LSRS (register), MOV (immediate), MOV (register), MOVS (immediate), MOVS (register), MVN (immediate), MVN (register), MVNS (immediate), MVNS (register), ORR (immediate), ORR (register), ORRS (immediate), ORRS (register), ROR (immediate), ROR (register), RORS (immediate), RORS (register), RSB (immediate), RSB (register), RSBS (immediate), RSBS (register), RSC (immediate), RSC (register), RSCS (immediate), RSCS (register), SBC (immediate), SBC (register), SBCS (immediate), SBCS (register), SUB (immediate), SUB (register), SUBS (immediate), and SUBS (register).
- A subset of those instructions which use the SIMD&FP register file. For these instructions, the effects of CPSR.DIT apply only if they pass their condition execution check. These instructions are:
  - CRC32B, CRC32H, CRC32W, CRC32CB, CRC32CH, CRC32CW, VABA*, VABD*, VABS, VACGE, VACGT, VACLE, VACLT, VADD (integer), VADDHN, VADDL, VADDW, VAND, VBIC, VBIF, VBIT, VBSL, VCGE, VCGT, VCLE, VCLS, VCLT, VCLZ, VCMP, VCMPE, VCNT, VDUP, VEOR, VEXT, VHADD, VHSUB, VMAX (integer), VMIN (integer), VMLA (integer), VMLAL, VMLS (integer), VMLSL, VMOV, VMOVL, VMOVN, VMUL (integer and polynomial), VMULL (integer and polynomial), VMVN, VNEG, VORN, VORR, VPADAL, VPADD (integer), VPADDL, VPMAX (integer), VPMIN (integer), VRADDHN, VREV*, VRHADD, VRSHL, VRSHR, VRSHRN, VRSRA, VRSUBHN, VSELEQ, VSELGE, VSELGT, VSELVS, VSHL, VSHLL, VSHR, VSLI, VSRA, VSRI, VSUB (integer), VSUBHN, VSUBL, VSUBW, VSWP, VTBL, VTBX, VTRN, VTST, VUZP, and VZIP

This field resets to 0.

Otherwise:

Reserved, RES0.

Bit [20]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

GE, bits [19:16]

Greater than or Equal flags, for parallel addition and subtraction.

Bits [15:10]

Reserved, RES0.

E, bit [9]

Endianness state bit. Controls the load and store endianness for data accesses:

`0b0` Little-endian operation

`0b1` Big-endian operation.

Instruction fetches ignore this bit.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.

When the reset value of the SCTLR.EE bit is defined by a configuration input signal, that value also applies to the CPSR.E bit on reset, and therefore applies to software execution from reset.

A, bit [8]

SError interrupt mask bit. The possible values of this bit are:

`0b0` Exception not masked.

`0b1` Exception masked.

I, bit [7]

IRQ mask bit. The possible values of this bit are:

`0b0` Exception not masked.

`0b1` Exception masked.

F, bit [6]

FIQ mask bit. The possible values of this bit are:

`0b0` Exception not masked.

`0b1` Exception masked.

Bit [5]

Reserved, RES0.

Bit [4]

Reserved, RES1.

M, bits [3:0]

Current PE mode. Possible values are:

`0b0000` User.

`0b0001` FIQ.

`0b0010` IRQ.

`0b0011` Supervisor.

`0b0110` Monitor.

`0b0111` Abort.

`0b1010` Hyp.

`0b1011` Undefined.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1111` System.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.34 CPPRCTX, Cache Prefetch Prediction Restriction by Context

The CPPRCTX characteristics are:

Purpose

Cache Prefetch Prediction Restriction by Context applies to all Cache Allocation Resources that predict cache allocations based on information gathered within the target execution context or contexts.

When this instruction is complete and synchronized, cache prefetch prediction does not permit later speculative execution within the target execution context to be observable through side channels.

This instruction applies to all:

- Instruction caches.
- Data caches.
- TLB prefetching hardware used by the executing PE that applies to the supplied context or contexts.

This instruction is guaranteed to be complete following a DSB that covers both read and write behavior on the same PE as executed the original restriction instruction, and a subsequent context synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

Note

This instruction does not require the invalidation of Cache Allocation Resources so long as the behavior described for completion of this instruction is met by the implementation.

On some implementations the instruction is likely to take a significant number of cycles to execute. This instruction is expected to be used very rarely, such as on the roll-over of an ASID or VMID, but should not be used on every context switch.

Configurations

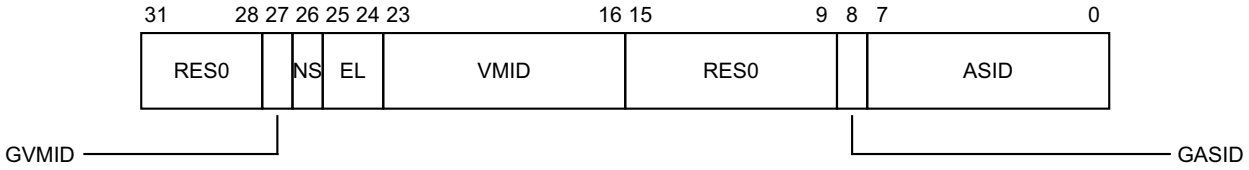
This instruction is present only when AArch32 is supported at any Exception level and ARMv8.0-PredInv is implemented. Otherwise, direct accesses to CPPRCTX are UNDEFINED.

Attributes

CPPRCTX is a 32-bit System instruction.

Field descriptions

The CPPRCTX input value bit assignments are:



Bits [31:28]

Reserved, RES0.

GVMID, bit [27]

Execution of this instruction applies to all VMIDs or a specified VMID.

`0b0` Applies to specified VMID for an EL0 or EL1 context. For all other contexts this field is RES0.

`0b1` Applies to all VMIDs for an EL0 or EL1 context. For all other contexts this field is RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the instruction is executed at EL0 or EL1, then this field has an Effective value of 0.

NS, bit [26]

Security State.

`0b0` Secure state.

`0b1` Non-secure state.

If the instruction is executed in Non-secure state, this field is treated as 1.

EL, bits [25:24]

Exception Level.

`0b00` EL0.

`0b01` EL1.

`0b10` EL2.

`0b11` EL3.

If the instruction is executed at an exception level lower than the specified level, this instruction is treated as a NOP.

VMID, bits [23:16]

Only applies when bit[27] is 0 and either:

- an EL1 context.
- an EL0 context when (HCR_EL2.E2H==0 or HCR_EL2.TGE==0) or EL2 is using AArch32 state.

Otherwise this field is RES0.

When the instruction is executed at EL1 then this field is treated as the current VMID.

When the instruction is executed at EL0 and (HCR_EL2.E2H==0 or HCR_EL2.TGE==0 or ELUsingAArch32(EL2)) then this field is treated as the current VMID.

When the instruction is executed at EL0 and (HCR_EL2.E2H==1 and HCR_EL2.TGE==1 and !ELUsingAArch32(EL2)) then this field is ignored.

Bits [15:9]

Reserved, RES0.

GASID, bit [8]

Execution of this instruction applies to all ASIDs or a specified ASID.

`0b0` Applies to specified ASID for an EL0 context. For all other contexts this field is RES0.

`0b1` Applies to all ASID for an EL0 context. For all other contexts this field is RES0.

If the instruction is executed at EL0, then this field has an Effective value of 0.

ASID, bits [7:0]

Only applies for an EL0 context and when bit[8] is 0.

Otherwise this field is RES0.

When the instruction is executed at EL0 then this field is treated as the current ASID.

Executing the CPPRCTX instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0011	0b111

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.EnRCTX == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && SCTLR.EnRCTX == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.EnRCTX == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        CPPRCTX(R[t]);
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x03);
    else
        CPPRCTX(R[t]);
elsif PSTATE.EL == EL2 then
    CPPRCTX(R[t]);
elsif PSTATE.EL == EL3 then
    CPPRCTX(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.35 CSSELR, Cache Size Selection Register

The CSSELR characteristics are:

Purpose

Selects the current Cache Size ID Register, CCSIDR, by specifying the required cache level and the cache type, which is either instruction cache or data cache.

If ARMv8.3-CCIDX is implemented, CSSELR also selects the current CCSIDR2.

Configurations

AArch32 System register CSSELR[31:0] is architecturally mapped to AArch64 System register CSSELR_EL1[31:0].

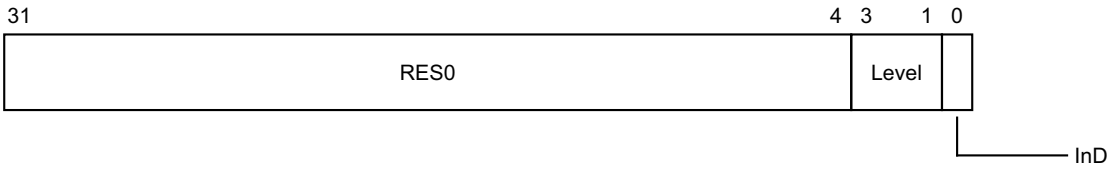
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to CSSELR are UNKNOWN.

Attributes

CSSELR is a 32-bit register.

Field descriptions

The CSSELR bit assignments are:



Bits [31:4]

Reserved, RES0.

Level, bits [3:1]

Cache level of required cache. Permitted values are:

`0b000` Level 1 cache.

`0b001` Level 2 cache.

`0b010` Level 3 cache.

`0b011` Level 4 cache.

`0b100` Level 5 cache.

`0b101` Level 6 cache.

`0b110` Level 7 cache.

All other values are reserved.

If CSSELR.Level is programmed to a cache level that is not implemented, then the value for this field on a read of CSSELR is UNKNOWN.

This field resets to an architecturally UNKNOWN value.

InD, bit [0]

Instruction not Data bit. Permitted values are:

`0b0` Data or unified cache.

`0b1` Instruction cache.

If CSSELR.Level is programmed to a cache level that is not implemented, then the value for this field on a read of CSSELR is UNKNOWN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

Accessing the CSSELR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b010	0b0000	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID4 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID2 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TID4 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return CSSELR_NS;
    else
        return CSSELR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return CSSELR_NS;
    else
        return CSSELR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return CSSELR_S;
    else
        return CSSELR_NS;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b010	0b0000	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID4 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID2 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TID4 == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        CSSELR_NS = R[t];
    else
        CSSELR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        CSSELR_NS = R[t];
    else
        CSSELR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        CSSELR_S = R[t];
    else
        CSSELR_NS = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.36 CTR, Cache Type Register

The CTR characteristics are:

Purpose

Provides information about the architecture of the caches.

Configurations

AArch32 System register CTR[31:0] is architecturally mapped to AArch64 System register CTR_EL0[31:0].

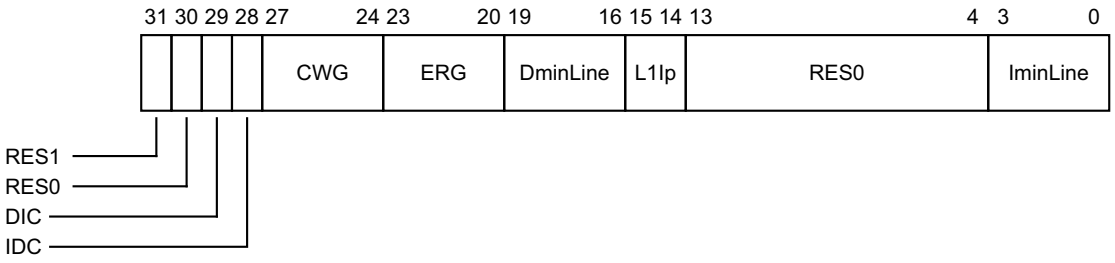
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to CTR are UNKNOWN.

Attributes

CTR is a 32-bit register.

Field descriptions

The CTR bit assignments are:



Bit [31]

Reserved, RES1.

Bit [30]

Reserved, RES0.

DIC, bit [29]

Instruction cache invalidation requirements for data to instruction coherence.

`0b0` Instruction cache invalidation to the Point of Unification is required for data to instruction coherence.

`0b1` Instruction cache invalidation to the Point of Unification is not required for data to instruction coherence.

IDC, bit [28]

Data cache clean requirements for instruction to data coherence. The meaning of this bit is:

`0b0` Data cache clean to the Point of Unification is required for instruction to data coherence, unless CLIDR.LoC == `0b000` or (CLIDR.LoUIS == `0b000` && CLIDR.LoUU == `0b000`).

`0b1` Data cache clean to the Point of Unification is not required for instruction to data coherence.

CWG, bits [27:24]

Cache writeback granule. Log_2 of the number of words of the maximum size of memory that can be overwritten as a result of the eviction of a cache entry that has had a memory location in it modified.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A value of `0b0000` indicates that this register does not provide Cache writeback granule information and either:

- The architectural maximum of 512 words (2KB) must be assumed.
- The Cache writeback granule can be determined from maximum cache line size encoded in the Cache Size ID Registers.

Values greater than `0b1001` are reserved.

Arm recommends that an implementation that does not support cache write-back implements this field as `0b0001`. This applies, for example, to an implementation that supports only write-through caches.

ERG, bits [23:20]

Exclusives reservation granule. Log_2 of the number of words of the maximum size of the reservation granule that has been implemented for the Load-Exclusive and Store-Exclusive instructions.

The use of the value `0b0000` is deprecated.

The value `0b0001` and values greater than `0b1001` are reserved.

DminLine, bits [19:16]

Log_2 of the number of words in the smallest cache line of all the data caches and unified caches that are controlled by the PE.

L1Ip, bits [15:14]

Level 1 instruction cache policy. Indicates the indexing and tagging policy for the L1 instruction cache. Possible values of this field are:

`0b00` VMID aware Physical Index, Physical tag (VPIPT)

`0b01` ASID-tagged Virtual Index, Virtual Tag (AIVIVT)

`0b10` Virtual Index, Physical Tag (VIPT)

`0b11` Physical Index, Physical Tag (PIPT)

The value `0b01` is reserved in Armv8.

The value `0b00` is permitted only in an implementation that includes ARMv8.2-VPIPT, otherwise the value is reserved.

Bits [13:4]

Reserved, RES0.

IminLine, bits [3:0]

Log_2 of the number of words in the smallest cache line of all the instruction caches that are controlled by the PE.

Accessing the CTR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID2 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        return CTR;
elsif PSTATE.EL == EL2 then
    return CTR;
elsif PSTATE.EL == EL3 then
    return CTR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.37 DACR, Domain Access Control Register

The DACR characteristics are:

Purpose

Defines the access permission for each of the sixteen memory domains.

Configurations

AArch32 System register DACR[31:0] is architecturally mapped to AArch64 System register DACR32_EL2[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DACR are UNKNOWN.

This register has no function when TTBCR.EAE is set to 1, to select the Long-descriptor translation table format.

Attributes

DACR is a 32-bit register.

Field descriptions

The DACR bit assignments are:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D15		D14		D13		D12		D11		D10		D9		D8		D7		D6		D5		D4		D3		D2		D1		D0	

D<n>, bits [2n+1:2n], for n = 0 to 15

Domain n access permission, where n = 0 to 15. Permitted values are:

`0b00` No access. Any access to the domain generates a Domain fault.

`0b01` Client. Accesses are checked against the permission bits in the translation tables.

`0b11` Manager. Accesses are not checked against the permission bits in the translation tables.

The value `0b10` is reserved.

This field resets to an architecturally UNKNOWN value.

Accessing the DACR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0011	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T3 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TRVM == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return DACR_NS;
    else
        return DACR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return DACR_NS;
    else
        return DACR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return DACR_S;
    else
        return DACR_NS;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0011	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T3 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        DACR_NS = R[t];
    else
        DACR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        DACR_NS = R[t];
    else
        DACR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' && CP15SDISABLE == HIGH then
        UNDEFINED;
    elsif SCR.NS == '0' && CP15SDISABLE2 == HIGH then
        UNDEFINED;
    else
        if SCR.NS == '0' then
            DACR_S = R[t];
        else
            DACR_NS = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.38 DCCIMVAC, Data Cache line Clean and Invalidate by VA to PoC

The DCCIMVAC characteristics are:

Purpose

Clean and Invalidate data or unified cache line by virtual address to PoC.

Configurations

AArch32 System instruction DCCIMVAC performs the same function as AArch64 System instruction DC CIVAC.

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DCCIMVAC are UNKNOWN.

Attributes

DCCIMVAC is a 32-bit System instruction.

Field descriptions

The DCCIMVAC input value bit assignments are:

31 … 0
Virtual address to use

Bits [31:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing the DCCIMVAC instruction

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see *AArch32 data cache maintenance instructions (DC*)* on page G4-5901.

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1110	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TPCP == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TPC == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        DCCIMVAC(R[t]);
elsif PSTATE.EL == EL2 then
    DCCIMVAC(R[t]);
elsif PSTATE.EL == EL3 then
    DCCIMVAC(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.39 DCCISW, Data Cache line Clean and Invalidate by Set/Way

The DCCISW characteristics are:

Purpose

Clean and Invalidate data or unified cache line by set/way.

Configurations

AArch32 System instruction DCCISW performs the same function as AArch64 System instruction DC CISW.

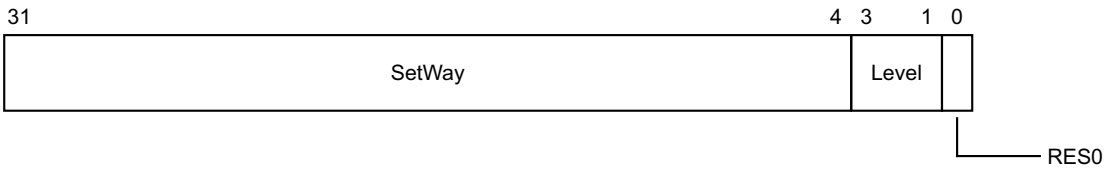
This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DCCISW are UNKNOWN.

Attributes

DCCISW is a 32-bit System instruction.

Field descriptions

The DCCISW input value bit assignments are:



SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log_2(ASSOCIATIVITY), L = Log_2(LINELEN), B = (L + S), S = Log_2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing the DCCISW instruction

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1110	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TSW == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TSW == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        DCCISW(R[t]);
elsif PSTATE.EL == EL2 then
    DCCISW(R[t]);
elsif PSTATE.EL == EL3 then
    DCCISW(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.40 DCCMVAC, Data Cache line Clean by VA to PoC

The DCCMVAC characteristics are:

Purpose

Clean data or unified cache line by virtual address to PoC.

Configurations

AArch32 System instruction DCCMVAC performs the same function as AArch64 System instruction DC CVAC.

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DCCMVAC are UNKNOWN.

Attributes

DCCMVAC is a 32-bit System instruction.

Field descriptions

The DCCMVAC input value bit assignments are:

31 … 0
Virtual address to use

Bits [31:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing the DCCMVAC instruction

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see *AArch32 data cache maintenance instructions (DC*)* on page G4-5901.

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TPCP == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TPC == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        DCCMVAC(R[t]);
elsif PSTATE.EL == EL2 then
    DCCMVAC(R[t]);
elsif PSTATE.EL == EL3 then
    DCCMVAC(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.41 DCCMVAU, Data Cache line Clean by VA to PoU

The DCCMVAU characteristics are:

Purpose

Clean data or unified cache line by virtual address to PoU.

Configurations

AArch32 System instruction DCCMVAU performs the same function as AArch64 System instruction DC CVAU.

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DCCMVAU are UNKNOWN.

Attributes

DCCMVAU is a 32-bit System instruction.

Field descriptions

The DCCMVAU input value bit assignments are:

31 … 0
Virtual address to use

Bits [31:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing the DCCMVAU instruction

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see *AArch32 data cache maintenance instructions (DC*)* on page G4-5901.

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1011	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TPU == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TOCU == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TPU == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TOCU == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        DCCMVAU(R[t]);
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        DCCMVAU(R[t]);
    elsif PSTATE.EL == EL3 then
        DCCMVAU(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.42 DCCSW, Data Cache line Clean by Set/Way

The DCCSW characteristics are:

Purpose

Clean data or unified cache line by set/way.

Configurations

AArch32 System instruction DCCSW performs the same function as AArch64 System instruction DC CSW.

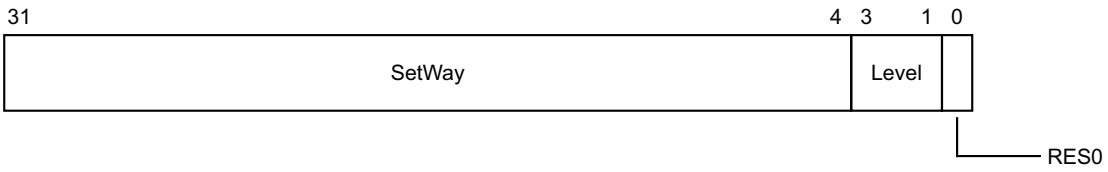
This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DCCSW are UNKNOWN.

Attributes

DCCSW is a 32-bit System instruction.

Field descriptions

The DCCSW input value bit assignments are:



SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log_2(ASSOCIATIVITY), L = Log_2(LINELEN), B = (L + S), S = Log_2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing the DCCSW instruction

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1010	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TSW == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TSW == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        DCCISW(R[t]);
elsif PSTATE.EL == EL2 then
    DCCISW(R[t]);
elsif PSTATE.EL == EL3 then
    DCCISW(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.43 DCIMVAC, Data Cache line Invalidate by VA to PoC

The DCIMVAC characteristics are:

Purpose

Invalidate data or unified cache line by virtual address to PoC.

Configurations

AArch32 System instruction DCIMVAC performs the same function as AArch64 System instruction DC IVAC.

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DCIMVAC are UNKNOWN.

Attributes

DCIMVAC is a 32-bit System instruction.

Field descriptions

The DCIMVAC input value bit assignments are:

31 … 0
Virtual address to use

Bits [31:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing the DCIMVAC instruction

It is IMPLEMENTATION DEFINED whether, when this instruction is executed, it can generate a watchpoint. If this instruction can generate a watchpoint this is prioritized in the same way as other watchpoints.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see *AArch32 data cache maintenance instructions (DC*)* on page G4-5901.

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0110	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TPCP == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TPC == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<DC,VM> != '00' then
        DCCIMVAC(R[t]);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.<DC,VM> != '00' then
        DCCIMVAC(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        DCIMVAC(R[t]);
elsif PSTATE.EL == EL2 then
    DCIMVAC(R[t]);
elsif PSTATE.EL == EL3 then
    DCIMVAC(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.44 DCISW, Data Cache line Invalidate by Set/Way

The DCISW characteristics are:

Purpose

Invalidate data or unified cache line by set/way.

Configurations

AArch32 System instruction DCISW performs the same function as AArch64 System instruction DC ISW.

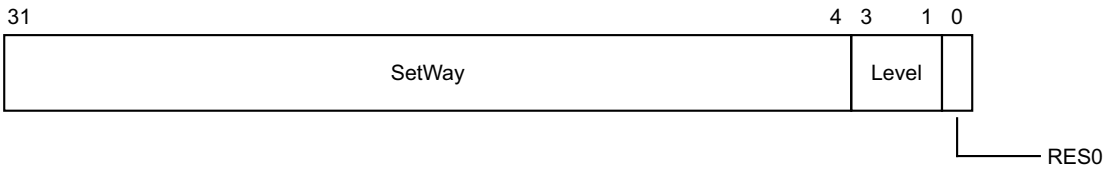
This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DCISW are UNKNOWN.

Attributes

DCISW is a 32-bit System instruction.

Field descriptions

The DCISW input value bit assignments are:



SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log_2(ASSOCIATIVITY), L = Log_2(LINELEN), B = (L + S), S = Log_2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing the DCISW instruction

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0110	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TSW == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TSW == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.SWIO == '1' then
        DCCISW(R[t]);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<DC,VM> != '00' then
        DCCISW(R[t]);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.SWIO == '1' then
        DCCISW(R[t]);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.<DC,VM> != '00' then
        DCCISW(R[t]);
    else
        DCISW(R[t]);
elsif PSTATE.EL == EL2 then
    DCISW(R[t]);
elsif PSTATE.EL == EL3 then
    DCISW(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.45 DFAR, Data Fault Address Register

The DFAR characteristics are:

Purpose

Holds the virtual address of the faulting address that caused a synchronous Data Abort exception.

Configurations

AArch32 System register DFAR[31:0] is architecturally mapped to AArch64 System register FAR_EL1[31:0].

AArch32 System register DFAR[31:0](S) is architecturally mapped to AArch32 System register HDFAR[31:0] when EL2 is implemented, EL3 is implemented and the highest implemented Exception level is using AArch32 state.

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DFAR are UNKNOWN.

Attributes

DFAR is a 32-bit register.

Field descriptions

The DFAR bit assignments are:

31 … 0
VA of faulting address of synchronous Data Abort exception

Bits [31:0]

VA of faulting address of synchronous Data Abort exception.

This field resets to an architecturally UNKNOWN value.

Accessing the DFAR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T6 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T6 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TRVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return DFAR_NS;
    else
        return DFAR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return DFAR_NS;
    else
        return DFAR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return DFAR_S;
    else
        return DFAR_NS;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T6 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T6 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        DFAR_NS = R[t];
    else
        DFAR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        DFAR_NS = R[t];
    else
        DFAR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        DFAR_S = R[t];
    else
        DFAR_NS = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.46 DFSR, Data Fault Status Register

The DFSR characteristics are:

Purpose

Holds status information about the last data fault.

Configurations

AArch32 System register DFSR[31:0] is architecturally mapped to AArch64 System register ESR_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DFSR are UNKNOWN.

The current translation table format determines which format of the register is used.

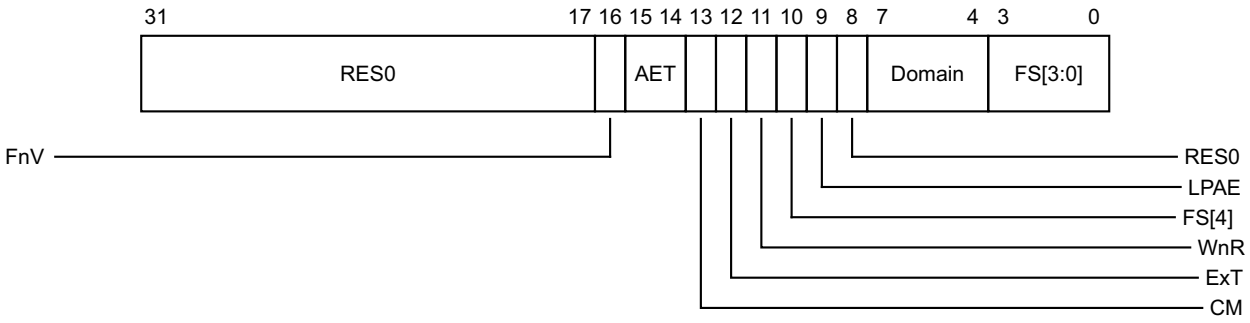
Attributes

DFSR is a 32-bit register.

Field descriptions

The DFSR bit assignments are:

When TTBCR.EAE == 0:



Bits [31:17]

Reserved, RES0.

FnV, bit [16]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

`0b0` DFAR is valid.

`0b1` DFAR is not valid, and holds an UNKNOWN value.

This field is only valid for a synchronous External abort other than a synchronous External abort on a translation table walk. It is RES0 for all other Data Abort exceptions.

This field resets to an architecturally UNKNOWN value.

AET, bits [15:14]

Asynchronous Error Type. When the RAS Extension is implemented, this field describes the state of the PE after taking an asynchronous Data Abort exception. Possible values are:

`0b00` Uncontainable error (UC) or uncategorized.

`0b01` Unrecoverable error (UEU).

`0b10` Restartable error (UEO) or Corrected error (CE).

`0b11` Recoverable error (UER).

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When the RAS Extension is not implemented, or on a synchronous Data Abort, this field is RES0.

Note

Armv8.2 requires the implementation of the RAS Extension.

In the event of multiple errors taken as a single SError interrupt exception, the overall state of the PE is reported.

Note

Software can use this information to determine what recovery might be possible. The recovery software must also examine any implemented fault records to determine the location and extent of the error.

This field resets to an architecturally UNKNOWN value.

CM, bit [13]

Cache maintenance fault. For synchronous faults, this bit indicates whether a cache maintenance instruction generated the fault. The possible values of this bit are:

`0b0` Abort not caused by execution of a cache maintenance instruction.

`0b1` Abort caused by execution of a cache maintenance instruction, or on an address translation.

On a synchronous Data Abort on a translation table walk, this bit is UNKNOWN.

On an asynchronous fault, this bit is UNKNOWN.

This field resets to an architecturally UNKNOWN value.

ExT, bit [12]

External abort type. This bit can be used to provide an IMPLEMENTATION DEFINED classification of External aborts.

In an implementation that does not provide any classification of External aborts, this bit is RES0.

For aborts other than External aborts this bit always returns 0.

This field resets to an architecturally UNKNOWN value.

WnR, bit [11]

Write not Read bit. Indicates whether the abort was caused by a write or a read instruction. The possible values of this bit are:

`0b0` Abort caused by a read instruction.

`0b1` Abort caused by a write instruction.

For faults on the cache maintenance and address translation System instructions in the (coproc==`0b1111`) encoding space this bit always returns a value of 1.

This field resets to an architecturally UNKNOWN value.

FS[4], bit [10]

This field is bit[4] of FS[4:0].

Fault status bits. Possible values of FS[4:0] are:

`0b00001` Alignment fault.

`0b00010` Debug exception.

`0b00011` Access flag fault, level 1.

`0b00100` Fault on instruction cache maintenance.

`0b00101` Translation fault, level 1.

`0b00110` Access flag fault, level 2.

`0b00111` Translation fault, level 2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b01000` Synchronous External abort, not on translation table walk.

`0b01001` Domain fault, level 1.

`0b01011` Domain fault, level 2.

`0b01100` Synchronous External abort, on translation table walk, level 1.

`0b01101` Permission fault, level 1.

`0b01110` Synchronous External abort, on translation table walk, level 2.

`0b01111` Permission fault, level 2.

`0b10000` TLB conflict abort.

`0b10100` IMPLEMENTATION DEFINED fault (Lockdown fault).

`0b10101` IMPLEMENTATION DEFINED fault (Unsupported Exclusive access fault).

`0b10110` SError interrupt.

`0b11000` **When RAS is not implemented** SError interrupt, from a parity or ECC error on memory access.

`0b11001` **When RAS is not implemented** Synchronous parity or ECC error on memory access, not on translation table walk.

`0b11100` **When RAS is not implemented** Synchronous parity or ECC error on translation table walk, level 1.

`0b11110` **When RAS is not implemented** Synchronous parity or ECC error on translation table walk, level 2.

All other values are reserved.

For more information about the lookup level associated with a fault, see *The level associated with MMU faults on a Short-descriptor translation table lookup* on page G5-6030.

The FS field is split as follows:

- FS[4] is DFSR[10].
- FS[3:0] is DFSR[3:0].

This field resets to an architecturally UNKNOWN value.

LPAE, bit [9]

On taking a Data Abort exception, this bit is set as follows:

`0b0` Using the Short-descriptor translation table formats.

`0b1` Using the Long-descriptor translation table formats.

Hardware does not interpret this bit to determine the behavior of the memory system, and therefore software can set this bit to 0 or 1 without affecting operation.

This field resets to an architecturally UNKNOWN value.

Bit [8]

Reserved, RES0.

Domain, bits [7:4]

The domain of the fault address.

Arm deprecates any use of this field, see *The Domain field in the DFSR* on page G5-6030.

This field is UNKNOWN for certain faults where the DFSR is updated and reported using the Short-descriptor FSR encodings, see Table G5-30 on page G5-6034.

This field resets to an architecturally UNKNOWN value.

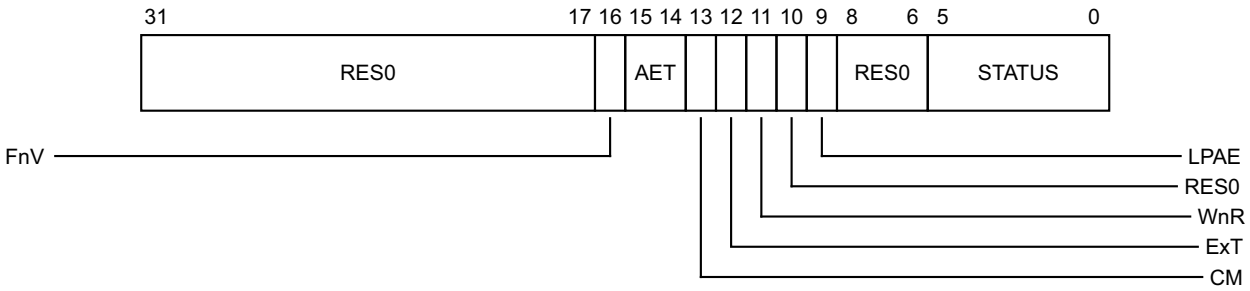
FS[3:0], bits [3:0]

This field is bits[3:0] of FS[4:0].

See FS[4] for the field description.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When TTBCR.EAE == 1:



Bits [31:17]

Reserved, RES0.

FnV, bit [16]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

`0b0` DFAR is valid.

`0b1` DFAR is not valid, and holds an UNKNOWN value.

This field is only valid for a synchronous External abort other than a synchronous External abort on a translation table walk. It is RES0 for all other Data Abort exceptions.

This field resets to an architecturally UNKNOWN value.

AET, bits [15:14]

Asynchronous Error Type. When the RAS Extension is implemented, this field describes the state of the PE after taking an asynchronous Data Abort exception. Possible values are:

`0b00` Uncontainable error (UC) or uncategorized.

`0b01` Unrecoverable error (UEU).

`0b10` Restartable error (UEO) or Corrected error (CE).

`0b11` Recoverable error (UER).

When the RAS Extension is not implemented, or on a synchronous Data Abort, this field is RES0.

Note

Armv8.2 requires the implementation of the RAS Extension.

In the event of multiple errors taken as a single SError interrupt exception, the overall state of the PE is reported.

Note

Software can use this information to determine what recovery might be possible. The recovery software must also examine any implemented fault records to determine the location and extent of the error.

This field resets to an architecturally UNKNOWN value.

CM, bit [13]

Cache maintenance fault. For synchronous faults, this bit indicates whether a cache maintenance instruction generated the fault. The possible values of this bit are:

`0b0` Abort not caused by execution of a cache maintenance instruction.

`0b1` Abort caused by execution of a cache maintenance instruction.

On a synchronous Data Abort on a translation table walk, this bit is UNKNOWN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

On an asynchronous fault, this bit is UNKNOWN.

This field resets to an architecturally UNKNOWN value.

ExT, bit [12]

External abort type. This bit can be used to provide an IMPLEMENTATION DEFINED classification of External aborts.

In an implementation that does not provide any classification of External aborts, this bit is RES0.

For aborts other than External aborts this bit always returns 0.

This field resets to an architecturally UNKNOWN value.

WnR, bit [11]

Write not Read bit. Indicates whether the abort was caused by a write or a read instruction. The possible values of this bit are:

`0b0` Abort caused by a read instruction.

`0b1` Abort caused by a write instruction.

For faults on the cache maintenance and address translation System instructions in the (coproc==`0b1111`) encoding space this bit always returns a value of 1.

This field resets to an architecturally UNKNOWN value.

Bit [10]

Reserved, RES0.

LPAE, bit [9]

On taking a Data Abort exception, this bit is set as follows:

`0b0` Using the Short-descriptor translation table formats.

`0b1` Using the Long-descriptor translation table formats.

Hardware does not interpret this bit to determine the behavior of the memory system, and therefore software can set this bit to 0 or 1 without affecting operation.

This field resets to an architecturally UNKNOWN value.

Bits [8:6]

Reserved, RES0.

STATUS, bits [5:0]

Fault status bits. Possible values of this field are:

`0b000000` Address size fault in translation table base register.

`0b000001` Address size fault, level 1.

`0b000010` Address size fault, level 2.

`0b000011` Address size fault, level 3.

`0b000101` Translation fault, level 1.

`0b000110` Translation fault, level 2.

`0b000111` Translation fault, level 3.

`0b001001` Access flag fault, level 1.

`0b001010` Access flag fault, level 2.

`0b001011` Access flag fault, level 3.

`0b001101` Permission fault, level 1.

`0b001110` Permission fault, level 2.

`0b001111` Permission fault, level 3.

`0b010000` Synchronous External abort, not on translation table walk.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b010001` Asynchronous SError interrupt.

`0b010101` Synchronous External abort, on translation table walk, level 1.

`0b010110` Synchronous External abort, on translation table walk, level 2.

`0b010111` Synchronous External abort, on translation table walk, level 3.

`0b011000` **When RAS is not implemented** Synchronous parity or ECC error on memory access, not on translation table walk.

`0b011001` **When RAS is not implemented** Asynchronous SError interrupt, from a parity or ECC error on memory access.

`0b011101` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk, level 1.

`0b011110` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk, level 2.

`0b011111` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk, level 3.

`0b100001` Alignment fault.

`0b100010` Debug exception.

`0b110000` TLB conflict abort.

`0b110100` IMPLEMENTATION DEFINED fault (Lockdown).

`0b110101` IMPLEMENTATION DEFINED fault (Unsupported Exclusive access).

All other values are reserved.

For more information about the lookup level associated with a fault, see *The level associated with MMU faults on a Long-descriptor translation table lookup* on page G5-6032.

This field resets to an architecturally UNKNOWN value.

Accessing the DFSR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TRVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return DFSR_NS;
    else
        return DFSR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return DFSR_NS;
    else
        return DFSR;
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if SCR.NS == '0' then
        return DFSR_S;
    else
        return DFSR_NS;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        DFSR_NS = R[t];
    else
        DFSR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        DFSR_NS = R[t];
    else
        DFSR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        DFSR_S = R[t];
    else
        DFSR_NS = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.47 DTLBIALL, Data TLB Invalidate All

The DTLBIALL characteristics are:

Purpose

Invalidate all cached copies of translation table entries from data TLBs that are from any level of the translation table walk. The entries that are invalidated are as follows:

- If executed at EL1, all entries that:
  - Would be required for the EL1&0 translation regime.
  - Match the current VMID, if EL2 is implemented and enabled in the current Security state.
- If executed in Secure state when EL3 is using AArch32, all entries that would be required for the Secure PL1&0 translation regime.
- If executed at EL2, and if EL2 is enabled in the current Security state, the stage 1 or stage 2 translation table entries that would be required for the Non-secure PL1&0 translation regime and matches the current VMID.

The invalidation only applies to the PE that executes this System instruction.

Arm deprecates the use of this System instruction. It is only provided for backwards compatibility with earlier versions of the Arm architecture.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DTLBIALL are UNKNOWN.

Attributes

DTLBIALL is a 32-bit System instruction.

Field descriptions

DTLBIALL ignores the value in the register specified by the instruction. Software does not have to write a value to the register before issuing this instruction.

Executing the DTLBIALL instruction

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0110	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TTLB == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        DTLBIALL();
elsif PSTATE.EL == EL2 then
    DTLBIALL();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    DTLBIALL();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.48 DTLBIASID, Data TLB Invalidate by ASID match

The DTLBIASID characteristics are:

Purpose

Invalidate all cached copies of translation table entries from data TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used for the specified ASID, and either:
  - Is from a level of lookup above the final level.
  - Is a non-global entry from the final level of lookup.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Arm deprecates the use of this System instruction. It is only provided for backwards compatibility with earlier versions of the Arm architecture.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DTLBIASID are UNKNOWN.

Attributes

DTLBIASID is a 32-bit System instruction.

Field descriptions

The DTLBIASID input value bit assignments are:

31 … 8	7 … 0
RES0	ASID

Bits [31:8]

Reserved, RES0.

ASID, bits [7:0]

ASID value to match. Any TLB entries for non-global pages that match the ASID values will be affected by this System instruction.

Executing the DTLBIASID instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0110	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TTLB == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        DTLBIASID(R[t]);
elsif PSTATE.EL == EL2 then
    DTLBIASID(R[t]);
elsif PSTATE.EL == EL3 then
    DTLBIASID(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.49 DTLBIMVA, Data TLB Invalidate by VA

The DTLBIMVA characteristics are:

Purpose

Invalidate all cached copies of translation table entries from data TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified address, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Arm deprecates the use of this System instruction. It is only provided for backwards compatibility with earlier versions of the Arm architecture.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DTLBIMVA are UNKNOWN.

Attributes

DTLBIMVA is a 32-bit System instruction.

Field descriptions

The DTLBIMVA input value bit assignments are:

31 – 12	11 – 8	7 – 0
VA	RES0	ASID

VA, bits [31:12]

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Bits [11:8]

Reserved, RES0.

ASID, bits [7:0]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

Executing the DTLBIMVA instruction

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0110	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TTLB == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        DTLBIMVA(R[t]);
elsif PSTATE.EL == EL2 then
    DTLBIMVA(R[t]);
elsif PSTATE.EL == EL3 then
    DTLBIMVA(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.50 DVPRCTX, Data Value Prediction Restriction by Context

The DVPRCTX characteristics are:

Purpose

Data Value Prediction Restriction by Context applies to all Data Value Prediction Resources that predict execution based on information gathered within the target execution context or contexts.

When this instruction is complete and synchronized, data value prediction does not permit later speculative execution within the target execution context to be observable through side channels.

This instruction is guaranteed to be complete following a DSB that covers both read and write behavior on the same PE as executed the original restriction instruction, and a subsequent context synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

Note

This instruction does not require the invalidation of prediction structures so long as the behavior described for completion of this instruction is met by the implementation.

On some implementations the instruction is likely to take a significant number of cycles to execute. This instruction is expected to be used very rarely, such as on the roll-over of an ASID or VMID, but should not be used on every context switch.

Configurations

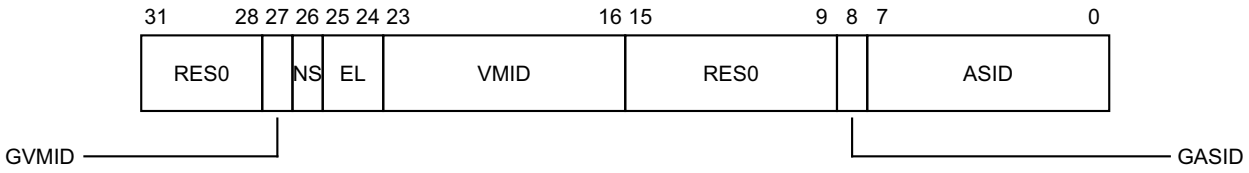
This instruction is present only when AArch32 is supported at any Exception level and ARMv8.0-PredInv is implemented. Otherwise, direct accesses to DVPRCTX are UNDEFINED.

Attributes

DVPRCTX is a 32-bit System instruction.

Field descriptions

The DVPRCTX input value bit assignments are:



Bits [31:28]

Reserved, RES0.

GVMID, bit [27]

Execution of this instruction applies to all VMIDs or a specified VMID.

`0b0` Applies to specified VMID for an EL0 or EL1 context. For all other contexts this field is RES0.

`0b1` Applies to all VMIDs for an EL0 or EL1 context. For all other contexts this field is RES0.

If the instruction is executed at EL0 or EL1, then this field has an Effective value of 0.

NS, bit [26]

Security State.

`0b0` Secure state.

`0b1` Non-secure state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the instruction is executed in Non-secure state, this field has an Effective value of 1.

EL, bits [25:24]

Exception Level

`0b00` EL0.

`0b01` EL1.

`0b10` EL2.

`0b11` EL3.

If the instruction is executed at an exception level lower than the specified level, this instruction is treated as a NOP.

VMID, bits [23:16]

Only applies when bit[27] is 0 and either:

- an EL1 context.
- an EL0 context when (HCR_EL2.E2H==0 or HCR_EL2.TGE==0) or EL2 is using AArch32 state.

Otherwise this field is RES0.

When the instruction is executed at EL1 then this field is treated as the current VMID.

When the instruction is executed at EL0 and (HCR_EL2.E2H==0 or HCR_EL2.TGE==0 or ELUsingAArch32(EL2)) then this field is treated as the current VMID.

When the instruction is executed at EL0 and (HCR_EL2.E2H==1 and HCR_EL2.TGE==1 and !ELUsingAArch32(EL2)) then this field is ignored.

Bits [15:9]

Reserved, RES0.

GASID, bit [8]

Execution of this instruction applies to all ASIDs or a specified ASID.

`0b0` Applies to specified ASID for an EL0 context. For all other contexts this field is RES0.

`0b1` Applies to all ASID for an EL0 context. For all other contexts this field is RES0.

If the instruction is executed at EL0, then this field has an Effective value of 0.

ASID, bits [7:0]

Only applies for an EL0 context and when bit[8] is 0.

Otherwise this field is RES0.

When the instruction is executed at EL0 then this field is treated as the current ASID.

Executing the DVPRCTX instruction

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0011	0b101

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.EnRCTX == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && SCTLR.EnRCTX == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.EnRCTX == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        DVPRCTX(R[t]);
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x03);
    else
        DVPRCTX(R[t]);
elsif PSTATE.EL == EL2 then
    DVPRCTX(R[t]);
elsif PSTATE.EL == EL3 then
    DVPRCTX(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.51 ELR_hyp, Exception Link Register (Hyp mode)

The ELR_hyp characteristics are:

Purpose

When taking an exception to Hyp mode, holds the address to return to.

Configurations

AArch32 System register ELR_hyp[31:0] is architecturally mapped to AArch64 System register ELR_EL2[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ELR_hyp are UNKNOWN.

Attributes

ELR_hyp is a 32-bit register.

Field descriptions

The ELR_hyp bit assignments are:

31 … 0
Return address

Bits [31:0]

Return address.

This field resets to an architecturally UNKNOWN value.

Accessing the ELR_hyp

ELR_hyp is accessible only at Hyp mode and Monitor mode.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS{<c>}{<q>} <Rd>, ELR_hyp

R	M	M1
0b0	0b1	0b1110

MSR{<c>}{<q>} ELR_hyp, <Rn>

R	M	M1
0b0	0b1	0b1110

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.52 FCSEIDR, FCSE Process ID register

The FCSEIDR characteristics are:

Purpose

Identifies whether the Fast Context Switch Extension (FCSE) is implemented.

From Armv8, the FCSE is not implemented, so this register is RAZ/WI. Software can access this register to determine that the implementation does not include the FCSE.

Configurations

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to FCSEIDR are UNKNOWN.

Attributes

FCSEIDR is a 32-bit register.

Field descriptions

The FCSEIDR bit assignments are:

31 … 0
RAZ/WI

Bits [31:0]

Reserved, RAZ/WI.

Accessing the FCSEIDR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        return FCSEIDR;
elsif PSTATE.EL == EL2 then
    return FCSEIDR;
elsif PSTATE.EL == EL3 then
    return FCSEIDR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        FCSEIDR = R[t];
elsif PSTATE.EL == EL2 then
    FCSEIDR = R[t];
elsif PSTATE.EL == EL3 then
    FCSEIDR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.53 FPEXC, Floating-Point Exception Control register

The FPEXC characteristics are:

Purpose

Provides a global enable for the implemented Advanced SIMD and floating-point functionality, and reports floating-point status information.

Configurations

AArch32 System register FPEXC[31:0] is architecturally mapped to AArch64 System register FPEXC32_EL2[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to FPEXC are UNKNOWN.

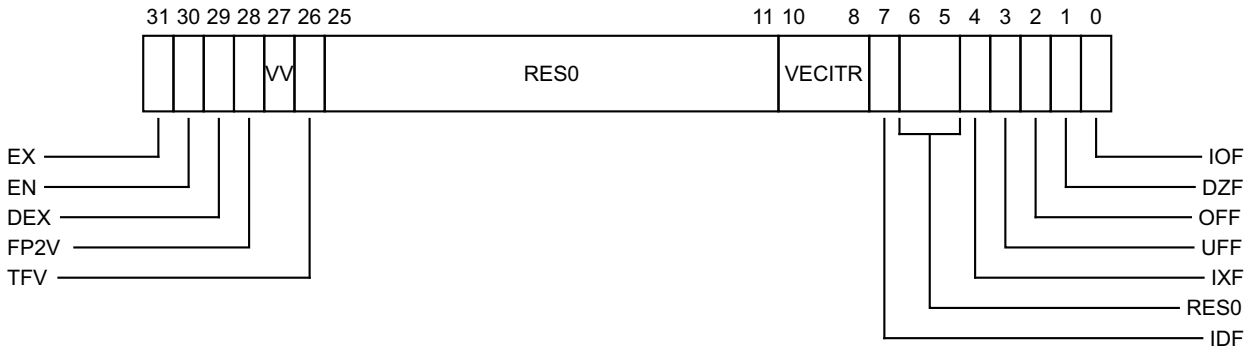
Implemented only if the implementation includes the Advanced SIMD and floating-point functionality.

Attributes

FPEXC is a 32-bit register.

Field descriptions

The FPEXC bit assignments are:



EX, bit [31]

Exception bit. From Armv8, this bit is RAZ/WI.

This field resets to an architecturally UNKNOWN value.

EN, bit [30]

Enables access to the Advanced SIMD and floating-point functionality from all Exception levels, except that setting this field to 0 does not disable the following:

- VMSR accesses to the FPEXC or FPSID.
- VMRS accesses from the FPEXC, FPSID, MVFR0, MVFR1, or MVFR2.

`0b0` Accesses to the FPSCR, and any of the SIMD and floating-point registers Q0-Q15, including their views as D0-D31 registers or S0-S31 registers, are UNDEFINED at all Exception levels.

`0b1` This control permits access to the Advanced SIMD and floating-point functionality at all Exception levels.

Execution of floating-point and Advanced SIMD instructions in AArch32 state can be disabled or trapped by the following controls:

- CPACR.cp10, or, if executing at EL0, CPACR_EL1.FPEN.
- FPEXC.EN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If executing in Non-secure state:
  - HCPTR.TCP10, or if EL2 is using AArch64, CPTR_EL2.TFP.
  - NSACR.cp10, or if EL3 is using AArch64, CPTR_EL3.TFP.
- For Advanced SIMD instructions only:
  - CPACR.ASEDIS.
  - If executing in Non-secure state, HCPTR.TASE and NSACR.NSTRCDIS.

See the descriptions of the controls for more information.

Note

When executing at EL0 using AArch32:

- If EL1 is using AArch64 then behavior is as if the value of FPEXC.EN is 1.
- If EL2 is using AArch64 and enabled in the current Security state, and the value of HCR_EL2.{RW, TGE} is {1, 1}, then the behavior is as if the value of FPEXC.EN is 1.
- If EL2 is using AArch64 and enabled in the current Security state, and the value of HCR_EL2.{RW, TGE} is {0, 1}, then it is IMPLEMENTATION DEFINED whether the behavior is:
  - As if the value of FPEXC.EN is 1.
  - Determined by the value of FPEXC.EN, as described in this field description. However, Arm deprecates using the value of FPEXC.EN to determine behavior.

This field resets to `0`.

DEX, bit [29]

Defined synchronous exception on floating-point execution.

This field identifies whether a synchronous exception generated by the attempted execution of an instruction was generated by an unallocated encoding. The instruction must be in the encoding space that is identified by the pseudocode function ExecutingCP10or11Instr() returning TRUE. This field also indicates whether the FPEXC.TFV field is valid.

The meaning of this bit is:

`0b0` The exception was generated by the attempted execution of an unallocated instruction in the encoding space that is identified by the pseudocode function ExecutingCP10or11Instr(). If FPEXC.TFV is RW then it is invalid and UNKNOWN. If FPEXC.{IDF, IXF, UFF, OFF, DZF, IOF} are RW then they are invalid and UNKNOWN.

`0b1` The exception was generated during the execution of an unallocated encoding. FPEXC.TFV is valid and indicates the cause of the exception.

On an exception that sets this bit to 1 the exception-handling routine must clear this bit to 0.

On an implementation that both does not support trapping of floating-point exceptions and implements the FPSCR.{Stride, Len} fields as RAZ, this bit is RES0.

This field resets to an architecturally UNKNOWN value.

FP2V, bit [28]

FPINST2 instruction valid bit. From Armv8, this bit is RES0.

This field resets to an architecturally UNKNOWN value.

VV, bit [27]

VECITR valid bit. From Armv8, this bit is RES0.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TFV, bit [26]

Trapped Fault Valid bit. Valid only when the value of FPEXC.DEX is 1. When valid, it indicates the cause of the exception and therefore whether the FPEXC.{IDF, IXF, UFF, OFF, DZF, IOF} bits are valid.

`0b0` The exception was caused by the execution of a floating-point VABS, VADD, VDIV, VFMA, VFMS, VFNMA, VFNMS, VMLA, VMLS, VMOV, VMUL, VNEG, VNMLA, VNMLS, VNMUL, VSQRT, or VSUB instruction when one or both of FPSCR.{Stride, Len} was non-zero. If the FPEXC.{IDF, IXF, UFF, OFF, DZF, IOF} bits are RW then they are invalid and UNKNOWN.

`0b1` FPEXC.{IDF, IXF, UFF, OFF, DZF, IOF} indicate the presence of trapped floating-point exceptions that had occurred at the time of the exception. Bits are set for all trapped exceptions that had occurred at the time of the exception.

This bit returns a status value and ignores writes.

When the value of FPEXC.DEX is 0 and this bit is RW, this bit is invalid and UNKNOWN.

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

On an implementation that supports the trapping of floating-point exceptions and implements FPSCR.{Stride, Len} as RAZ, this bit is RAO/WI.

This field resets to an architecturally UNKNOWN value.

Bits [25:11]

Reserved, RES0.

VECITR, bits [10:8]

Vector iteration count. From Armv8, this field is RES1.

This field resets to an architecturally UNKNOWN value.

IDF, bit [7]

Input Denormal trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Input Denormal exception occurred while FPSCR.IDE was 1:

`0b0` Input Denormal exception has not occurred.

`0b1` Input Denormal exception has occurred.

Input Denormal exceptions can occur only when FPSCR.FZ is 1.

Note

A half-precision floating-point value that is flushed to zero because the value of FPSCR.FZ16 is 1 does not generate an Input Denormal exception.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

This field resets to an architecturally UNKNOWN value.

Bits [6:5]

Reserved, RES0.

IXF, bit [4]

Inexact trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Inexact exception occurred while FPSCR.IXE was 1:

`0b0` Inexact exception has not occurred.

`0b1` Inexact exception has occurred.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

This field resets to an architecturally UNKNOWN value.

UFF, bit [3]

Underflow trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Underflow exception occurred while FPSCR.UFE was 1:

`0b0` Underflow exception has not occurred.

`0b1` Underflow exception has occurred.

Underflow trapped exceptions can occur:

- On half-precision data-processing instructions only when FPSCR.FZ16 is 0.
- Otherwise only when FPSCR.FZ is 0.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

This field resets to an architecturally UNKNOWN value.

OFF, bit [2]

Overflow trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Overflow exception occurred while FPSCR.OFE was 1:

`0b0` Overflow exception has not occurred.

`0b1` Overflow exception has occurred.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

This field resets to an architecturally UNKNOWN value.

DZF, bit [1]

Divide by Zero trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether a Divide by Zero exception occurred while FPSCR.DZE was 1:

`0b0` Divide by Zero exception has not occurred.

`0b1` Divide by Zero exception has occurred.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

This field resets to an architecturally UNKNOWN value.

IOF, bit [0]

Invalid Operation trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Invalid Operation exception occurred while FPSCR.IOE was 1:

`0b0` Invalid Operation exception has not occurred.

`0b1` Invalid Operation exception has occurred.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

This field resets to an architecturally UNKNOWN value.

Accessing the FPEXC

Accesses to this register use the following encodings in the System instruction encoding space:

VMRS{<c>}{<q>} <Rt>, <spec_reg>

reg
0b1000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if ELUsingAArch32(EL1) && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 == '0') || CPACR.cp10
== '00') then
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TFP == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 ==
'0') || HCPTR.TCP10 == '1') then
        AArch32.TakeHypTrapException(0x08);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x08);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
        AArch32.TakeHypTrapException(0x08);
    else
        return FPEXC;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TFP == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 ==
'0') || HCPTR.TCP10 == '1') then
        AArch32.TakeHypTrapException(0x00);
    else
        return FPEXC;
elsif PSTATE.EL == EL3 then
    if CPACR.cp10 == '00' then
        UNDEFINED;
    else
        return FPEXC;
```

VMSR{<c>}{<q>} <spec_reg>, <Rt>

reg
0b1000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if ELUsingAArch32(EL1) && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 == '0') || CPACR.cp10
== '00') then
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TFP == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 ==
'0') || HCPTR.TCP10 == '1') then
        AArch32.TakeHypTrapException(0x08);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x08);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
        AArch32.TakeHypTrapException(0x08);
    else
        FPEXC = R[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TFP == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 ==
'0') || HCPTR.TCP10 == '1') then
        AArch32.TakeHypTrapException(0x00);
    else
        FPEXC = R[t];
elsif PSTATE.EL == EL3 then
    if CPACR.cp10 == '00' then
        UNDEFINED;
    else
        FPEXC = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.54 FPSCR, Floating-Point Status and Control Register

The FPSCR characteristics are:

Purpose

Provides floating-point system status information and control.

Configurations

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to FPSCR are UNKNOWN.

The named fields in this register map to the equivalent fields in the AArch64 FPCR and FPSR.

It is IMPLEMENTATION DEFINED whether the Len and Stride fields can be programmed to non-zero values, which will cause some AArch32 floating-point instruction encodings to be UNDEFINED, or whether these fields are RAZ.

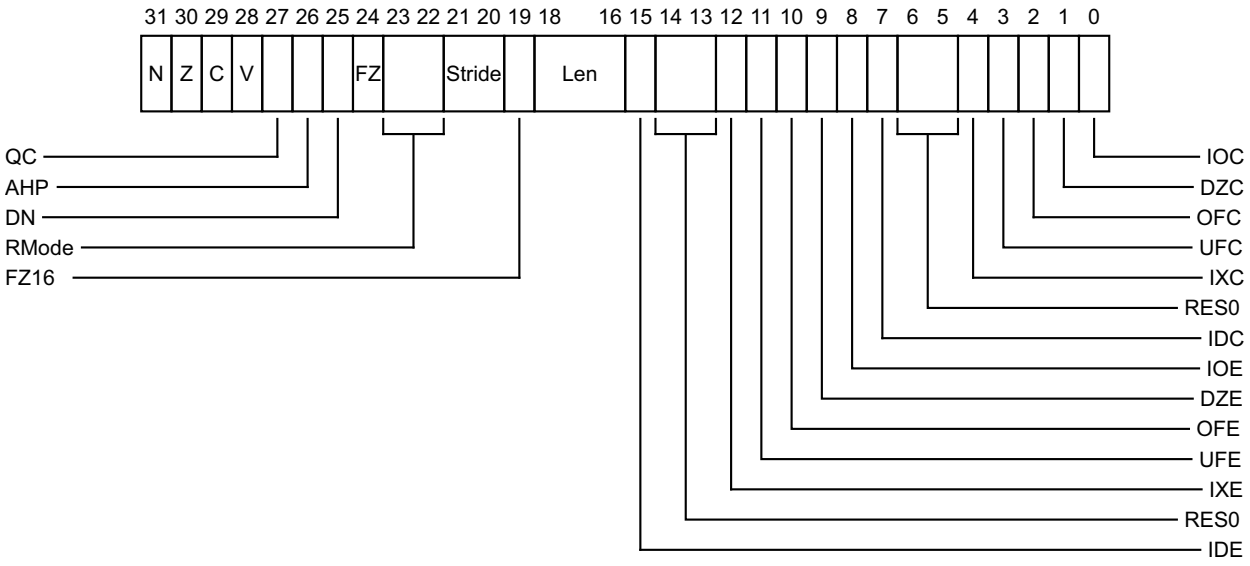
Implemented only if the implementation includes the Advanced SIMD and floating-point functionality.

Attributes

FPSCR is a 32-bit register.

Field descriptions

The FPSCR bit assignments are:



N, bit [31]

Negative condition flag. This is updated by floating-point comparison operations.

This field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero condition flag. This is updated by floating-point comparison operations.

This field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry condition flag. This is updated by floating-point comparison operations.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

V, bit [28]

Overflow condition flag. This is updated by floating-point comparison operations.

This field resets to an architecturally UNKNOWN value.

QC, bit [27]

Cumulative saturation bit, Advanced SIMD only. This bit is set to 1 to indicate that an Advanced SIMD integer operation has saturated since 0 was last written to this bit.

This field resets to an architecturally UNKNOWN value.

AHP, bit [26]

Alternative half-precision control bit:

`0b0` IEEE half-precision format selected.

`0b1` Alternative half-precision format selected.

This bit is only used for conversions between half-precision floating-point and other floating-point formats.

The data-processing instructions added as part of the ARMv8.2-FP16 extension always use the IEEE half-precision format, and ignore the value of this bit.

This field resets to an architecturally UNKNOWN value.

DN, bit [25]

Default NaN mode control bit:

`0b0` NaN operands propagate through to the output of a floating-point operation.

`0b1` Any operation involving one or more NaNs returns the Default NaN.

The value of this bit only controls scalar floating-point arithmetic. Advanced SIMD arithmetic always uses the Default NaN setting, regardless of the value of the DN bit.

This field resets to an architecturally UNKNOWN value.

FZ, bit [24]

Flush-to-zero mode control bit:

`0b0` Flush-to-zero mode disabled. Behavior of the floating-point system is fully compliant with the IEEE 754 standard.

`0b1` Flush-to-zero mode enabled.

The value of this bit only controls scalar floating-point arithmetic. Advanced SIMD arithmetic always uses the Flush-to-zero setting, regardless of the value of the FZ bit.

This bit has no effect on half-precision calculations.

This field resets to an architecturally UNKNOWN value.

RMode, bits [23:22]

Rounding Mode control field. The encoding of this field is:

`0b00` Round to Nearest (RN) mode.

`0b01` Round towards Plus Infinity (RP) mode.

`0b10` Round towards Minus Infinity (RM) mode.

`0b11` Round towards Zero (RZ) mode.

The specified rounding mode is used by almost all scalar floating-point instructions. Advanced SIMD arithmetic always uses the Round to Nearest setting, regardless of the value of the RMode bits.

This field resets to an architecturally UNKNOWN value.

Stride, bits [21:20]

It is IMPLEMENTATION DEFINED whether this field is RW or RAZ.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If this field is RW and is set to a value other than zero, some floating-point instruction encodings are UNDEFINED. The instruction pseudocode identifies these instructions.

Arm strongly recommends that software never sets this field to a value other than zero.

The value of this field is ignored when processing Advanced SIMD instructions.

This field resets to an architecturally UNKNOWN value.

FZ16, bit [19]

When ARMv8.2-FP16 is implemented:

Flush-to-zero mode control bit on half-precision data-processing instructions:

`0b0` Flush-to-zero mode disabled. Behavior of the floating-point system is fully compliant with the IEEE 754 standard.

`0b1` Flush-to-zero mode enabled.

The value of this bit applies to both scalar and Advanced SIMD floating-point half-precision calculations.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Len, bits [18:16]

It is IMPLEMENTATION DEFINED whether this field is RW or RAZ.

If this field is RW and is set to a value other than zero, some floating-point instruction encodings are UNDEFINED. The instruction pseudocode identifies these instructions.

Arm strongly recommends that software never sets this field to a value other than zero.

The value of this field is ignored when processing Advanced SIMD instructions.

This field resets to an architecturally UNKNOWN value.

IDE, bit [15]

Input Denormal floating-point exception trap enable. Possible values are:

`0b0` Untrapped exception handling selected. If the floating-point exception occurs then the IDC bit is set to 1.

`0b1` Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the IDC bit. The trap handling software can decide whether to set the IDC bit to 1.

This bit is RW only if the implementation supports the trapping of floating-point exceptions. In an implementation that does not support floating-point exception trapping, this bit is RAZ/WI.

When this bit is RW, it applies only to floating-point operations. Advanced SIMD operations always use untrapped floating-point exception handling in AArch32 state.

This field resets to an architecturally UNKNOWN value.

Bits [14:13]

Reserved, RES0.

IXE, bit [12]

Inexact floating-point exception trap enable. Possible values are:

`0b0` Untrapped exception handling selected. If the floating-point exception occurs then the IXC bit is set to 1.

`0b1` Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the IXC bit. The trap handling software can decide whether to set the IXC bit to 1.

This bit is RW only if the implementation supports the trapping of floating-point exceptions. In an implementation that does not support floating-point exception trapping, this bit is RAZ/WI.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When this bit is RW, it applies only to floating-point operations. Advanced SIMD operations always use untrapped floating-point exception handling in AArch32 state.

This field resets to an architecturally UNKNOWN value.

UFE, bit [11]

Underflow floating-point exception trap enable. Possible values are:

`0b0` Untrapped exception handling selected. If the floating-point exception occurs then the UFC bit is set to 1.

`0b1` Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the UFC bit. The trap handling software can decide whether to set the UFC bit to 1.

This bit is RW only if the implementation supports the trapping of floating-point exceptions. In an implementation that does not support floating-point exception trapping, this bit is RAZ/WI.

When this bit is RW, it applies only to floating-point operations. Advanced SIMD operations always use untrapped floating-point exception handling in AArch32 state.

This field resets to an architecturally UNKNOWN value.

OFE, bit [10]

Overflow floating-point exception trap enable. Possible values are:

`0b0` Untrapped exception handling selected. If the floating-point exception occurs then the OFC bit is set to 1.

`0b1` Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the OFC bit. The trap handling software can decide whether to set the OFC bit to 1.

This bit is RW only if the implementation supports the trapping of floating-point exceptions. In an implementation that does not support floating-point exception trapping, this bit is RAZ/WI.

When this bit is RW, it applies only to floating-point operations. Advanced SIMD operations always use untrapped floating-point exception handling in AArch32 state.

This field resets to an architecturally UNKNOWN value.

DZE, bit [9]

Divide by Zero floating-point exception trap enable. Possible values are:

`0b0` Untrapped exception handling selected. If the floating-point exception occurs then the DZC bit is set to 1.

`0b1` Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the DZC bit. The trap handling software can decide whether to set the DZC bit to 1.

This bit is RW only if the implementation supports the trapping of floating-point exceptions. In an implementation that does not support floating-point exception trapping, this bit is RAZ/WI.

When this bit is RW, it applies only to floating-point operations. Advanced SIMD operations always use untrapped floating-point exception handling in AArch32 state.

This field resets to an architecturally UNKNOWN value.

IOE, bit [8]

Invalid Operation floating-point exception trap enable. Possible values are:

`0b0` Untrapped exception handling selected. If the floating-point exception occurs then the IOC bit is set to 1.

`0b1` Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the IOC bit. The trap handling software can decide whether to set the IOC bit to 1.

This bit is RW only if the implementation supports the trapping of floating-point exceptions. In an implementation that does not support floating-point exception trapping, this bit is RAZ/WI.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When this bit is RW, it applies only to floating-point operations. Advanced SIMD operations always use untrapped floating-point exception handling in AArch32 state.

This field resets to an architecturally UNKNOWN value.

IDC, bit [7]

Input Denormal cumulative floating-point exception bit. This bit is set to 1 to indicate that the Input Denormal floating-point exception has occurred since 0 was last written to this bit.

How VFP instructions update this bit depends on the value of the IDE bit.

Advanced SIMD instructions set this bit if the Input Denormal floating-point exception occurs in one or more of the floating-point calculations performed by the instruction, regardless of the value of the IDE bit.

This field resets to an architecturally UNKNOWN value.

Bits [6:5]

Reserved, RES0.

IXC, bit [4]

Inexact cumulative floating-point exception bit. This bit is set to 1 to indicate that the Inexact floating-point exception has occurred since 0 was last written to this bit.

How VFP instructions update this bit depends on the value of the IXE bit.

Advanced SIMD instructions set this bit if the Inexact floating-point exception occurs in one or more of the floating-point calculations performed by the instruction, regardless of the value of the IXE bit.

The criteria for the Inexact floating-point exception to occur are different in Flush-to-zero mode. For details, see *Flush-to-zero* on page A1-55.

This field resets to an architecturally UNKNOWN value.

UFC, bit [3]

Underflow cumulative floating-point exception bit. This bit is set to 1 to indicate that the Underflow floating-point exception has occurred since 0 was last written to this bit.

How VFP instructions update this bit depends on the value of the UFE bit.

Advanced SIMD instructions set this bit if the Underflow floating-point exception occurs in one or more of the floating-point calculations performed by the instruction, regardless of the value of the UFE bit.

The criteria for the Underflow floating-point exception to occur are different in Flush-to-zero mode. For details, see *Flush-to-zero* on page A1-55.

This field resets to an architecturally UNKNOWN value.

OFC, bit [2]

Overflow cumulative floating-point exception bit. This bit is set to 1 to indicate that the Overflow floating-point exception has occurred since 0 was last written to this bit.

How VFP instructions update this bit depends on the value of the OFE bit.

Advanced SIMD instructions set this bit if the Overflow floating-point exception occurs in one or more of the floating-point calculations performed by the instruction, regardless of the value of the OFE bit.

This field resets to an architecturally UNKNOWN value.

DZC, bit [1]

Divide by Zero cumulative floating-point exception bit. This bit is set to 1 to indicate that the Divide by Zero floating-point exception has occurred since 0 was last written to this bit.

How VFP instructions update this bit depends on the value of the DZE bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Advanced SIMD instructions set this bit if the Divide by Zero floating-point exception occurs in one or more of the floating-point calculations performed by the instruction, regardless of the value of the DZE bit.

This field resets to an architecturally UNKNOWN value.

IOC, bit [0]

Invalid Operation cumulative floating-point exception bit. This bit is set to 1 to indicate that the Invalid Operation floating-point exception has occurred since 0 was last written to this bit.

How VFP instructions update this bit depends on the value of the IOE bit.

Advanced SIMD instructions set this bit if the Invalid Operation floating-point exception occurs in one or more of the floating-point calculations performed by the instruction, regardless of the value of the IOE bit.

This field resets to an architecturally UNKNOWN value.

Accessing the FPSCR

Accesses to this register use the following encodings in the System instruction encoding space:

VMRS{<c>}{<q>} <Rt>, <spec_reg>

reg
0b0001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if CPACR_EL1.FPEN == 'x0' then
        AArch64.AArch32SystemAccessTrap(EL1, 0x07);
    elsif ELUsingAArch32(EL1) && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 == '0') ||
CPACR.cp10 == '00') then
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TFP == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 ==
'0') || HCPTR.TCP10 == '1') then
        AArch32.TakeHypTrapException(0x08);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x08);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
        AArch32.TakeHypTrapException(0x08);
    else
        return FPSCR;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TFP == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 ==
'0') || HCPTR.TCP10 == '1') then
        AArch32.TakeHypTrapException(0x00);
    else
        return FPSCR;
elsif PSTATE.EL == EL3 then
    if CPACR.cp10 == '00' then
        UNDEFINED;
    else
        return FPSCR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

VMSR{<c>}{<q>} <spec_reg>, <Rt>

reg
0b0001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if CPACR_EL1.FPEN == 'x0' then
        AArch64.AArch32SystemAccessTrap(EL1, 0x07);
    elsif ELUsingAArch32(EL1) && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 == '0') ||
CPACR.cp10 == '00') then
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TFP == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 ==
'0') || HCPTR.TCP10 == '1') then
        AArch32.TakeHypTrapException(0x08);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x08);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
        AArch32.TakeHypTrapException(0x08);
    else
        FPSCR = R[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TFP == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 ==
'0') || HCPTR.TCP10 == '1') then
        AArch32.TakeHypTrapException(0x00);
    else
        FPSCR = R[t];
elsif PSTATE.EL == EL3 then
    if CPACR.cp10 == '00' then
        UNDEFINED;
    else
        FPSCR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.55 FPSID, Floating-Point System ID register

The FPSID characteristics are:

Purpose

Provides top-level information about the floating-point implementation.

This register largely duplicates information held in the MIDR. Arm deprecates use of it.

Configurations

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to FPSID are UNKNOWN.

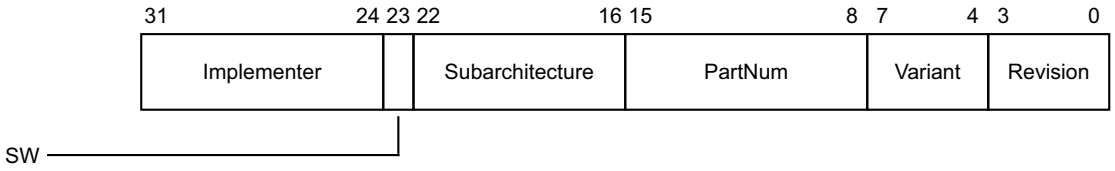
Implemented only if the implementation includes the Advanced SIMD and floating-point functionality.

Attributes

FPSID is a 32-bit register.

Field descriptions

The FPSID bit assignments are:



Implementer, bits [31:24]

Implementer codes are the same as those used for the MIDR.

For an implementation by Arm this field is `0x41`, the ASCII code for A.

This field resets to an architecturally UNKNOWN value.

SW, bit [23]

Software bit. Defined values are:

`0b0` The implementation provides a hardware implementation of the floating-point instructions.

`0b1` The implementation supports only software emulation of the floating-point instructions.

In Armv8-A the only permitted value is `0b0`.

This field resets to an architecturally UNKNOWN value.

Subarchitecture, bits [22:16]

Subarchitecture version number. For an implementation by Arm, defined values are:

`0b0000000` VFPv1 architecture with an IMPLEMENTATION DEFINED subarchitecture.

`0b0000001` VFPv2 architecture with Common VFP subarchitecture v1.

`0b0000010` VFPv3 architecture, or later, with Common VFP subarchitecture v2. The VFP architecture version is indicated by the MVFR0 and MVFR1 registers.

`0b0000011` VFPv3 architecture, or later, with Null subarchitecture. The entire floating-point implementation is in hardware, and no software support code is required. The VFP architecture version is indicated by the MVFR0 and MVFR1 registers. This value can be used only by an implementation that does not support the trap enable bits in the FPSCR.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b0000100` VFPv3 architecture, or later, with Common VFP subarchitecture v3, and support for trap enable bits in FPSCR. The VFP architecture version is indicated by the MVFR0 and MVFR1 registers.

For a subarchitecture designed by Arm the most significant bit of this field, register bit[22], is 0. Values with a most significant bit of 0 that are not listed here are reserved.

When the subarchitecture designer is not Arm, the most significant bit of this field, register bit[22], must be 1. Each implementer must maintain its own list of subarchitectures it has designed, starting at subarchitecture version number `0x40`.

In Armv8-A the permitted values are `0b0000011` and `0b0000100`.

This field resets to an architecturally UNKNOWN value.

PartNum, bits [15:8]

An IMPLEMENTATION DEFINED part number for the floating-point implementation, assigned by the implementer.

This field resets to an architecturally UNKNOWN value.

Variant, bits [7:4]

An IMPLEMENTATION DEFINED variant number. Typically, this field distinguishes between different production variants of a single product.

This field resets to an architecturally UNKNOWN value.

Revision, bits [3:0]

An IMPLEMENTATION DEFINED revision number for the floating-point implementation.

This field resets to an architecturally UNKNOWN value.

Accessing the FPSID

Accesses to this register use the following encodings in the System instruction encoding space:

VMRS{<c>}{<q>} <Rt>, <spec_reg>

reg
0b0000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if ELUsingAArch32(EL1) && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 == '0') || CPACR.cp10
== '00') then
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TFP == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 ==
'0') || HCPTR.TCP10 == '1') then
        AArch32.TakeHypTrapException(0x08);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x08);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
        AArch32.TakeHypTrapException(0x08);
    else
        return FPSID;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TFP == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 ==
'0') || HCPTR.TCP10 == '1') then
        AArch32.TakeHypTrapException(0x00);
    else
        return FPSID;
elsif PSTATE.EL == EL3 then
    if CPACR.cp10 == '00' then
        UNDEFINED;
    else
        return FPSID;
```

VMSR{<c>}{<q>} <spec_reg>, <Rt>

reg
0b0000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if ELUsingAArch32(EL1) && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 == '0') || CPACR.cp10
== '00') then
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TFP == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 ==
'0') || HCPTR.TCP10 == '1') then
        AArch32.TakeHypTrapException(0x08);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x08);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
        AArch32.TakeHypTrapException(0x08);
    else
        //no operation
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TFP == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 ==
'0') || HCPTR.TCP10 == '1') then
        AArch32.TakeHypTrapException(0x00);
    else
        //no operation
elsif PSTATE.EL == EL3 then
    if CPACR.cp10 == '00' then
        UNDEFINED;
    else
        //no operation
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.56 HACR, Hyp Auxiliary Configuration Register

The HACR characteristics are:

Purpose

Controls trapping to Hyp mode of IMPLEMENTATION DEFINED aspects of Non-secure EL1 or EL0 operation.

Configurations

AArch32 System register HACR[31:0] is architecturally mapped to AArch64 System register HACR_EL2[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to HACR are UNKNOWN.

If EL2 is not implemented, this register is RES0 from EL3.

Attributes

HACR is a 32-bit register.

Field descriptions

The HACR bit assignments are:

31	0
IMPLEMENTATION DEFINED	

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

Accessing the HACR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return HACR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        return HACR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    HACR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        HACR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.57 HACTLR, Hyp Auxiliary Control Register

The HACTLR characteristics are:

Purpose

Controls IMPLEMENTATION DEFINED features of Hyp mode operation.

Configurations

AArch32 System register HACTLR[31:0] is architecturally mapped to AArch64 System register ACTLR_EL2[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to HACTLR are UNKNOWN.

If EL2 is not implemented, this register is RES0 from EL3.

Attributes

HACTLR is a 32-bit register.

Field descriptions

The HACTLR bit assignments are:

31 … 0
IMPLEMENTATION DEFINED

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

Accessing the HACTLR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return HACTLR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        return HACTLR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    HACTLR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        HACTLR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.58 HACTLR2, Hyp Auxiliary Control Register 2

The HACTLR2 characteristics are:

Purpose

Provides additional space to the HACTLR register to hold IMPLEMENTATION DEFINED trap functionality.

Configurations

AArch32 System register HACTLR2[31:0] is architecturally mapped to AArch64 System register ACTLR_EL2[63:32].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to HACTLR2 are UNKNOWN.

In Armv8.0 and Armv8.1, it is IMPLEMENTATION DEFINED whether this register is implemented, or whether it causes UNDEFINED exceptions when accessed. The implementation of this register can be detected by examining ID_MMFR4.AC2.

From Armv8.2 this register must be implemented.

Attributes

HACTLR2 is a 32-bit register.

Field descriptions

The HACTLR2 bit assignments are:

31 … 0
IMPLEMENTATION DEFINED

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

Accessing the HACTLR2

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0000	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return HACTLR2;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        UNDEFINED;
    else
        return HACTLR2;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0000	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    HACTLR2 = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        HACTLR2 = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.59 HADFSR, Hyp Auxiliary Data Fault Status Register

The HADFSR characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED syndrome information for Data Abort exceptions taken to Hyp mode.

Configurations

AArch32 System register HADFSR[31:0] is architecturally mapped to AArch64 System register AFSR0_EL2[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to HADFSR are UNKNOWN.

This is an optional register. An implementation that does not require this register can implement it as RES0.

If EL2 is not implemented, this register is RES0 from EL3.

Attributes

HADFSR is a 32-bit register.

Field descriptions

The HADFSR bit assignments are:

31 … 0
IMPLEMENTATION DEFINED

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

Accessing the HADFSR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0101	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return HADFSR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        return HADFSR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0101	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    HADFSR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        HADFSR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.60 HAIFSR, Hyp Auxiliary Instruction Fault Status Register

The HAIFSR characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED syndrome information for Prefetch Abort exceptions taken to Hyp mode.

Configurations

AArch32 System register HAIFSR[31:0] is architecturally mapped to AArch64 System register AFSR1_EL2[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to HAIFSR are UNKNOWN.

This is an optional register. An implementation that does not require this register can implement it as RES0.

If EL2 is not implemented, this register is RES0 from EL3.

Attributes

HAIFSR is a 32-bit register.

Field descriptions

The HAIFSR bit assignments are:

31	0
IMPLEMENTATION DEFINED	

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

Accessing the HAIFSR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0101	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return HAIFSR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        return HAIFSR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0101	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    HAIFSR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        HAIFSR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.61 HAMAIR0, Hyp Auxiliary Memory Attribute Indirection Register 0

The HAMAIR0 characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED memory attributes for the memory attribute encodings defined by HMAIR0. These IMPLEMENTATION DEFINED attributes can only provide additional qualifiers for the memory attribute encodings, and cannot change the memory attributes defined in HMAIR0.

Configurations

AArch32 System register HAMAIR0[31:0] is architecturally mapped to AArch64 System register AMAIR_EL2[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to HAMAIR0 are UNKNOWN.

If EL2 is not implemented, this register is RES0 from EL3.

Attributes

HAMAIR0 is a 32-bit register.

Field descriptions

The HAMAIR0 bit assignments are:

31 … 0
IMPLEMENTATION DEFINED

If an implementation does not provide any IMPLEMENTATION DEFINED memory attributes, this register is RES0.

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

Accessing the HAMAIR0

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1010	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T10 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T10 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return HAMAIR0;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        return HAMAIR0;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1010	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T10 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T10 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    HAMAIR0 = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        HAMAIR0 = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.62 HAMAIR1, Hyp Auxiliary Memory Attribute Indirection Register 1

The HAMAIR1 characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED memory attributes for the memory attribute encodings defined by HMAIR1. These IMPLEMENTATION DEFINED attributes can only provide additional qualifiers for the memory attribute encodings, and cannot change the memory attributes defined in HMAIR1.

Configurations

AArch32 System register HAMAIR1[31:0] is architecturally mapped to AArch64 System register AMAIR_EL2[63:32].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to HAMAIR1 are UNKNOWN.

If EL2 is not implemented, this register is RES0 from EL3.

Attributes

HAMAIR1 is a 32-bit register.

Field descriptions

The HAMAIR1 bit assignments are:

31	0
IMPLEMENTATION DEFINED	

If an implementation does not provide any IMPLEMENTATION DEFINED memory attributes, this register is RES0.

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

Accessing the HAMAIR1

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1010	0b0011	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T10 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T10 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return HAMAIR1;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        return HAMAIR1;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1010	0b0011	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T10 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T10 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    HAMAIR1 = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        HAMAIR1 = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.63 HCPTR, Hyp Architectural Feature Trap Register

The HCPTR characteristics are:

Purpose

Controls:

- Trapping to Hyp mode of Non-secure access, at EL1 or EL0, to trace, and to Advanced SIMD and floating-point functionality.
- Hyp mode access to trace, and to Advanced SIMD and floating-point functionality.

Note

Accesses to this functionality:

- From Non-secure modes other than Hyp mode are also affected by settings in the CPACR and NSACR.
- From Hyp mode are also affected by settings in the NSACR.

Exceptions generated by the CPACR and NSACR controls are higher priority than those generated by the HCPTR controls.

Configurations

AArch32 System register HCPTR[31:0] is architecturally mapped to AArch64 System register CPTR_EL2[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to HCPTR are UNKNOWN.

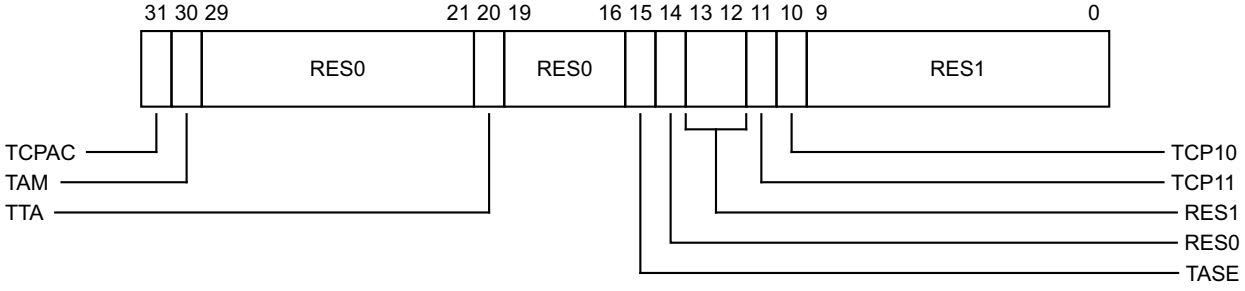
If EL2 is not implemented, this register is RES0 from EL3.

Attributes

HCPTR is a 32-bit register.

Field descriptions

The HCPTR bit assignments are:



TCPAC, bit [31]

Traps Non-secure EL1 accesses to the CPACR to Hyp mode.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Non-secure EL1 accesses to the CPACR are trapped to Hyp mode.

Note

The CPACR is not accessible at EL0.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TAM, bit [30]

When AMUv1 is implemented:

Trap Activity Monitor access. Traps Non-secure EL1 and EL0 accesses to all Activity Monitor registers to EL2.

`0b0` Accesses from Non-secure EL1 and EL0 to Activity Monitor registers are not trapped.

`0b1` Accesses from Non-secure EL1 and EL0 to Activity Monitor registers are trapped to Hyp mode.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

Otherwise:

Reserved, RES0.

Bits [29:21]

Reserved, RES0.

TTA, bit [20]

Traps Non-secure System register accesses to all implemented trace registers to Hyp mode.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Any Non-secure System register access to an implemented trace register is trapped to Hyp mode, unless the access is trapped to EL1 by a CPACR or NSACR control, or the access is from Non-secure EL0 and the definition of the register in the appropriate trace architecture specification indicates that the register is not accessible from EL0. A trapped instruction generates:

- A Hyp Trap exception, if the exception is taken from Non-secure EL0 or EL1.
- An Undefined Instruction exception taken to Hyp mode, if the exception is taken from Hyp mode.

If the implementation does not include a PE trace unit, or does not include a System register interface to the PE trace unit registers, it is IMPLEMENTATION DEFINED whether this bit:

- Is RES0.
- Is RES1.
- Can be written from Hyp mode, and from Secure Monitor mode when SCR.NS is 1.

If EL3 is implemented and is using AArch32, and the value of NSACR.NSTRCDIS is 1, in Non-secure state this field behaves as RAO/WI, regardless of its actual value.

Note

- The ETMv4 architecture does not permit EL0 to access the trace registers. If the implementation includes an ETMv4 implementation, EL0 accesses to the trace registers are UNDEFINED, and a resulting Undefined Instruction exception is higher priority than a HCPTR.TTA Hyp Trap exception.
- The architecture does not provide traps on trace register accesses through the optional memory-mapped debug interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

Bits [19:16]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TASE, bit [15]

Traps Non-secure execution of Advanced SIMD instructions to Hyp mode when the value of HCPTR.TCP10 is 0.

`0b0` This control does not cause any instructions to be trapped.

`0b1` When the value of HCPTR.TCP10 is 0, any attempt to execute an Advanced SIMD instruction in Non-secure state is trapped to Hyp mode, unless it is trapped to EL1 by a CPACR or NSACR control. A trapped instruction generates:

- A Hyp Trap exception, if the exception is taken from Non-secure EL0 or EL1.
- An Undefined Instruction exception taken to Hyp mode, if the exception is taken from Hyp mode.

When the value of HCPTR.TCP10 is 1, the value of this field is ignored.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES1. Otherwise, it is IMPLEMENTATION DEFINED whether this field is implemented as a RW field. If it is not implemented as a RW field, then it is RAZ/WI.

If EL3 is implemented and is using AArch32, and the value of NSACR.NSASEDIS is 1, in Non-secure state this field behaves as RAO/WI, regardless of its actual value. This applies even if the field is implemented as RAZ/WI.

For the list of instructions affected by this field, see *Controls of Advanced SIMD operation that do not apply to floating-point operation* on page E1-3966.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

Bit [14]

Reserved, RES0.

Bits [13:12]

Reserved, RES1.

TCP11, bit [11]

The value of this field is ignored. If this field is programmed with a different value to the TCP10 bit then this field is UNKNOWN on a direct read of the HCPTR.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES1.

If EL3 is implemented and is using AArch32, and the value of NSACR.cp10 is 0, in Non-secure state this field behaves as RAO/WI, regardless of its actual value.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

TCP10, bit [10]

Trap Non-secure accesses to Advanced SIMD and floating-point functionality to Hyp mode:

`0b0` This control does not cause any instructions to be trapped.

`0b1` Any attempted access to Advanced SIMD and floating-point functionality from Non-secure state is trapped to Hyp mode, unless it is trapped to EL1 by a CPACR or NSACR control. A trapped instruction generates:

- A Hyp Trap exception, if the exception is taken from Non-secure EL0 or EL1.
- An Undefined Instruction exception taken to Hyp mode, if the exception is taken from Hyp mode.

The Advanced SIMD and floating-point features controlled by these fields are:

- Execution of any floating-point or Advanced SIMD instruction.
- Any access to the Advanced SIMD and floating-point registers D0-D31 and their views as S0-S31 and Q0-Q15.
- Any access to the FPSCR, FPSID, MVFR0, MVFR1, MVFR2, or FPEXC System registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES1.

If EL3 is implemented and is using AArch32, and the value of NSACR.cp10 is 0, in Non-secure state this field behaves as RAO/WI, regardless of its actual value.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

Bits [9:0]

Reserved, RES1.

Accessing the HCPTR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return HCPTR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        return HCPTR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        HCPTR = R[t];
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if SCR.NS == '0' then
    UNDEFINED;
else
    HCPTR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.64 HCR, Hyp Configuration Register

The HCR characteristics are:

Purpose

Provides configuration controls for virtualization, including defining whether various Non-secure operations are trapped to Hyp mode.

Configurations

AArch32 System register HCR[31:0] is architecturally mapped to AArch64 System register HCR_EL2[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to HCR are UNKNOWN.

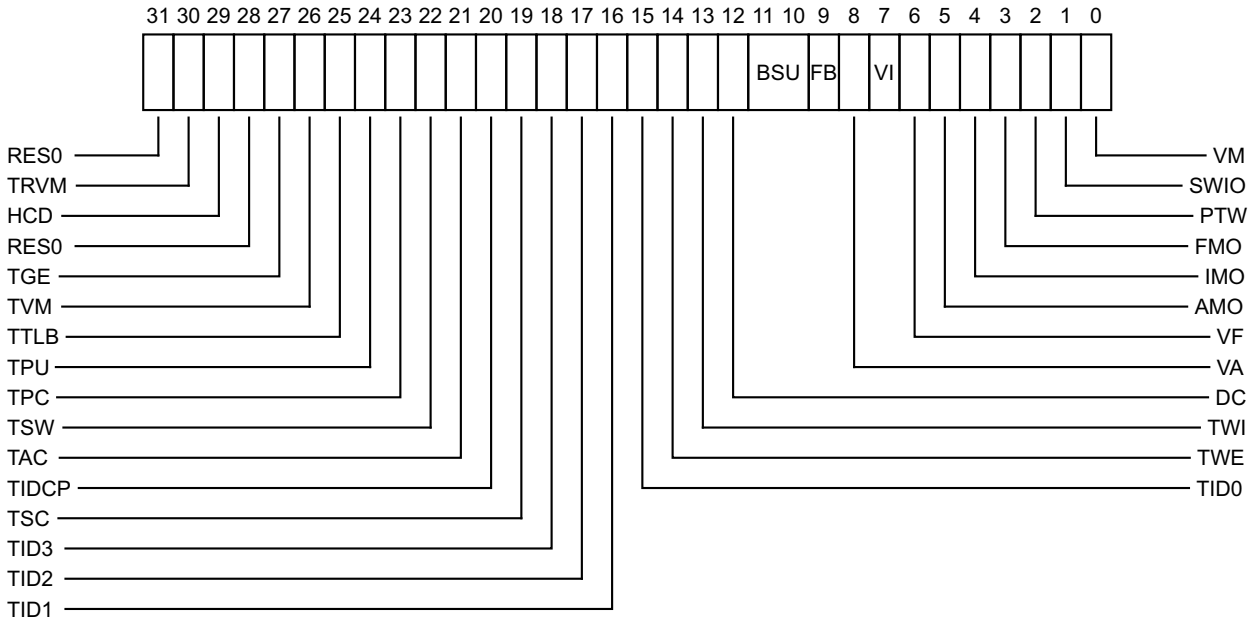
If EL2 is not implemented, this register is RES0 from EL3.

Attributes

HCR is a 32-bit register.

Field descriptions

The HCR bit assignments are:



Bit [31]

Reserved, RES0.

TRVM, bit [30]

Trap Reads of Virtual Memory controls. Traps Non-secure EL1 reads of the virtual memory control registers to EL2, when EL2 is enabled in the current Security state.

The registers for which read accesses are trapped are as follows:

SCTLR, TTBR0, TTBR1, TTBCR, TTBCR2, DACR, DFSR, IFSR, DFAR, IFAR, ADFSR, AIFSR, PRRR, NMRR, MAIR0, MAIR1, AMAIR0, AMAIR1, CONTEXTIDR.

`0b0` This control does not cause any instructions to be trapped.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` Non-secure EL1 read accesses to the specified Virtual Memory controls are trapped to EL2.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

HCD, bit [29]

When EL3 is not implemented:

HVC instruction disable. Disables Non-secure EL1 and EL2 execution of HVC instructions, when EL2 is enabled in the current Security state.

`0b0` HVC instruction execution is enabled at EL2 and EL1.

`0b1` HVC instructions are UNDEFINED at EL2 and Non-secure EL1.

The Undefined Instruction exception is taken to the Exception level at which the HVC instruction is executed.

Note

HVC instructions are always UNDEFINED at EL0.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

Otherwise:

Reserved, RES0.

Bit [28]

Reserved, RES0.

TGE, bit [27]

Trap General Exceptions, from Non-secure EL0.

`0b0` This control has no effect on execution at EL0.

`0b1` When EL2 is not enabled in the current Security state, this control has no effect on execution at EL0.

When EL2 is enabled in the current Security state, then:

- All exceptions that would be routed to EL1 are routed to EL2.
- The SCTLR.M bit is treated as being 0 for all purposes other than returning the result of a direct read of SCTLR.
- The HCR.{FMO, IMO, AMO} bits are treated as being 1 for all purposes other than returning the result of a direct read of HCR.
- All virtual interrupts are disabled.
- Any IMPLEMENTATION DEFINED mechanisms for signaling virtual interrupts are disabled.
- An exception return to EL1 is treated as an illegal exception return.
- Monitor mode execution of an MSR or CPS instruction that changes CPSR.M to a Non-secure EL1 mode is an illegal change to PSTATE.M. For more information see *Illegal changes to PSTATE.M* on page G1-5699.

Also, when HCR.TGE is 1:

- If EL3 is using AArch32, an attempt to change from a Secure PL1 mode to a Non-secure EL1 mode by changing SCR.NS from 0 to 1 results in SCR.NS remaining as 0.
- The HDCR.{TDRA, TDOSA, TDA, TDE} bits are ignored and treated as being 1 other than for the purpose of a direct read of HDCR.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

TVM, bit [26]

Trap Virtual Memory controls. Traps Non-secure EL1 writes to the virtual memory control registers to EL2, when EL2 is enabled in the current Security state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The registers for which write accesses are trapped are as follows:

SCTLR, TTBR0, TTBR1, TTBCR, TTBCR2, DACR, DFSR, IFSR, DFAR, IFAR, ADFSR, AIFSR, PRRR, NMRR, MAIR0, MAIR1, AMAIR0, AMAIR1, CONTEXTIDR.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Non-secure EL1 write accesses to the specified virtual memory control registers are trapped to EL2.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

TTLB, bit [25]

Trap TLB maintenance instructions. Traps Non-secure EL1 execution of a TLBI instruction to EL2, when EL2 is enabled in the current Security state.

This applies to the following instructions:

TLBIALLIS, TLBIMVAIS, TLBIASIDIS, TLBIMVAAIS, TLBIMVALIS, TLBIMVAALIS, ITLBIALL, ITLBIMVA, ITLBIASID, DTLBIALL, DTLBIMVA, DTLBIASID, TLBIALL, TLBIMVA, TLBIASID, TLBIMVAA, TLBIMVAL, TLBIMVAAL

`0b0` This control does not cause any instructions to be trapped.

`0b1` Non-secure EL1 accesses to the specified TLB maintenance instructions are trapped to EL2.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

TPU, bit [24]

Trap cache maintenance instructions that operate to the Point of Unification. Traps Non-secure EL1 execution of those cache maintenance instructions to EL2, when EL2 is enabled in the current Security state.

This applies to the following instructions:

- ICIMVAU, ICIALLU, ICIALLUIS, DCCMVAU.

Note

An Undefined Instruction exception generated at EL0 is higher priority than this trap to EL2, and these instructions are always UNDEFINED at EL0.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Non-secure EL1 execution of the specified cache maintenance instructions is trapped to EL2.

If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean by VA to the Point of Unification instruction can be trapped when the value of this control is 1.

If the Point of Unification is before any level of instruction cache, it is IMPLEMENTATION DEFINED whether the execution of any instruction cache invalidate to the Point of Unification instruction can be trapped when the value of this control is 1.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

TPC, bit [23]

Trap data or unified cache maintenance instructions that operate to the Point of Coherency. Traps Non-secure EL1 execution of those cache maintenance instructions to EL2, when EL2 is enabled in the current Security state.

This applies to the following instructions:

- DCIMVAC, DCCIMVAC, DCCMVAC.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

An Undefined Instruction exception generated at EL0 is higher priority than this trap to EL2, and these instructions are always UNDEFINED at EL0.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Non-secure EL1 execution of the specified cache maintenance instructions is trapped to EL2.

If the Point of Coherency is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean, invalidate, or clean and invalidate instruction that operates by VA to the point of coherency can be trapped when the value of this control is 1.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

TSW, bit [22]

Trap data or unified cache maintenance instructions that operate by Set/Way. Traps Non-secure EL1 execution of those cache maintenance instructions by set/way to EL2, when EL2 is enabled in the current Security state.

This applies to the following instructions:

- DCISW, DCCSW, DCCISW.

Note

An Undefined Instruction exception generated at EL0 is higher priority than this trap to EL2, and these instructions are always UNDEFINED at EL0.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Non-secure EL1 execution of the specified cache maintenance instructions is trapped to EL2.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

TAC, bit [21]

Trap Auxiliary Control Registers. Traps Non-secure EL1 accesses to the Auxiliary Control Registers to EL2, when EL2 is enabled in the current Security state, from both Execution states.

This applies to the following register accesses:

ACTLR and, if implemented, ACTLR2.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Non-secure EL1 accesses to the specified registers are trapped to EL2.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

TIDCP, bit [20]

Trap IMPLEMENTATION DEFINED functionality. Traps Non-secure EL1 accesses to the encodings for IMPLEMENTATION DEFINED System Registers to EL2, when EL2 is enabled in the current Security state.

MCR and MRC instructions accessing the following encodings:

- All coproc==p15, CRn==c9, Opcode1 = {0-7}, CRm == {c0-c2, c5-c8}, opcode2 == {0-7}.
- All coproc==p15, CRn==c10, Opcode1 ==={0-7}, CRm == {c0, c1, c4, c8}, opcode2 == {0-7}.
- All coproc==p15, CRn==c11, Opcode1==={0-7}, CRm == {c0-c8, c15}, opcode2 == {0-7}.

When HCR.TIDCP is set to 1, it is IMPLEMENTATION DEFINED whether any of this functionality accessed from Non-secure EL0 is trapped to EL2. Otherwise, it is UNDEFINED and the PE takes an Undefined Instruction exception to Non-secure Undefined mode.

`0b0` This control does not cause any instructions to be trapped.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` Non-secure EL1 accesses to the specified System register encodings for IMPLEMENTATION DEFINED functionality are trapped to EL2.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

TSC, bit [19]

Trap SMC instructions. Traps Non-secure EL1 execution of SMC instructions to Hyp mode.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Any attempt to execute an SMC instruction at Non-secure EL1 is trapped to Hyp mode, regardless of the value of SCR.SCD.

The Armv8-A architecture permits, but does not require, this trap to apply to conditional SMC instructions that fail their condition code check, in the same way as with traps on other conditional instructions.

Note

- This trap is only implemented if the implementation includes EL3.
- SMC instructions are always UNDEFINED at PL0.
- This bit traps execution of the SMC instruction. It is not a routing control for the SMC exception. Hyp Trap exceptions and SMC exceptions have different preferred return addresses.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

TID3, bit [18]

Trap ID group 3. Traps Non-secure EL1 reads of the following registers to EL2, when EL2 is enabled in the current Security state, as follows:

- ID_PFR0, ID_PFR1, ID_DFR0, ID_AFR0, ID_MMFR0, ID_MMFR1, ID_MMFR2, ID_MMFR3, ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR3, ID_ISAR4, ID_ISAR5, MVFR0, MVFR1, MVFR2.
- ID_MMFR4 and ID_MMFR5 are trapped unless implemented as RAZ, when it is IMPLEMENTATION DEFINED whether accesses to ID_MMFR4 or ID_MMFR5 are trapped.
- ID_ISAR6 is trapped unless implemented as RAZ, when it is IMPLEMENTATION DEFINED whether accesses to ID_ISAR6 are trapped.
- ID_DFR1 is trapped unless implemented as RAZ, when it is IMPLEMENTATION DEFINED whether accesses to ID_DFR1 are trapped.
- It is IMPLEMENTATION DEFINED whether this bit traps MRC accesses to registers not already mentioned, with coproc==p15, opc1 == 0, CRn == c0, CRm == {c2-c7}, opc2 == {0-7}.

`0b0` This control does not cause any instructions to be trapped.

`0b1` The specified Non-secure EL1 read accesses to ID group 3 registers are trapped to EL2.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

TID2, bit [17]

Trap ID group 2. Traps the following register accesses to EL2, when EL2 is enabled in the current Security state:

- Non-secure EL1 and EL0 reads of the CTR, CCSIDR, CCSIDR2, CLIDR, and CSSELR.
- Non-secure EL1 and EL0 writes to the CSSELR.

`0b0` This control does not cause any instructions to be trapped.

`0b1` The specified Non-secure EL1 and EL0 accesses to ID group 2 registers are trapped to EL2.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TID1, bit [16]

Trap ID group 1. Traps Non-secure EL1 reads of the following registers to EL2, when EL2 is enabled in the current Security state:

TCMTR, TLBTR, REVIDR, AIDR.

`0b0` This control does not cause any instructions to be trapped.

`0b1` The specified Non-secure EL1 read accesses to ID group 1 registers are trapped to EL2.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

TID0, bit [15]

Trap ID group 0. Traps the following register accesses to EL2, when EL2 is enabled in the current Security state:

- Non-secure EL1 reads of the JIDR and FPSID.
- If the JIDR is RAZ from Non-secure EL0, Non-secure EL0 reads of the JIDR.

Note

- It is IMPLEMENTATION DEFINED whether the JIDR is RAZ or UNDEFINED at EL0. If it is UNDEFINED at EL0 then the Undefined Instruction exception takes precedence over this trap.
- The FPSID is not accessible at EL0.
- Writes to the FPSID are ignored, and not trapped by this control.

`0b0` This control does not cause any instructions to be trapped.

`0b1` The specified Non-secure EL1 read accesses to ID group 0 registers are trapped to EL2.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

TWE, bit [14]

Traps Non-secure EL0 and EL1 execution of WFE instructions to EL2, when EL2 is enabled in the current Security state.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Any attempt to execute a WFE instruction at Non-secure EL0 or EL1 is trapped to EL2, if the instruction would otherwise have caused the PE to enter a low-power state and it is not trapped by SCTLR.nTWE.

The attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

Note

Since a WFE can complete at any time, even without a Wakeup event, the traps on WFE are not guaranteed to be taken, even if the WFE is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

TWI, bit [13]

Traps Non-secure EL0 and EL1 execution of WFI instructions to EL2, when EL2 is enabled in the current Security state.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Any attempt to execute a WFI instruction at Non-secure EL0 or EL1 is trapped to EL2, if the instruction would otherwise have caused the PE to enter a low-power state and it is not trapped by SCTLR.nTWI.

The attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

Since a WFI can complete at any time, even without a Wakeup event, the traps on WFI are not guaranteed to be taken, even if the WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

DC, bit [12]

Default Cacheability.

`0b0` This control has no effect on the Non-secure EL1&0 translation regime.

`0b1` In Non-secure state:

- The SCTLR.M field behaves as 0 for all purposes other than a direct read of the value of the field.
- The HCR.VM field behaves as 1 for all purposes other than a direct read of the value of the field.
- The memory type produced by the first stage of the EL1&0 translation regime is Normal Non-Shareable, Inner Write-Back Read-Allocate Write-Allocate, Outer Write-Back Read-Allocate Write-Allocate.

This field has no effect on the EL2 and EL3 translation regimes.

This field is permitted to be cached in a TLB.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

BSU, bits [11:10]

Barrier Shareability upgrade. This field determines the minimum shareability domain that is applied to any barrier instruction executed from Non-secure EL1 or Non-secure EL0:

`0b00` No effect.

`0b01` Inner Shareable.

`0b10` Outer Shareable.

`0b11` Full system.

This value is combined with the specified level of the barrier held in its instruction, using the same principles as combining the shareability attributes from two stages of address translation.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

FB, bit [9]

Force broadcast. Causes the following instructions to be broadcast within the Inner Shareable domain when executed from Non-secure EL1:

BPIALL, TLBIALL, TLBIMVA, TLBIASID, DTLBIALL, DTLBIMVA, DTLBIASID, ITLBIALL, ITLBIMVA, ITLBIASID, TLBIMVAA, ICIALLU, TLBIMVAL, TLBIMVAAL.

`0b0` This field has no effect on the operation of the specified instructions.

`0b1` When one of the specified instruction is executed at Non-secure EL1, the instruction is broadcast within the Inner Shareable shareability domain.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

VA, bit [8]

Virtual SError interrupt exception.

`0b0` This mechanism is not making a virtual SError interrupt pending.

`0b1` A virtual SError interrupt is pending because of this mechanism.

The virtual SError interrupt is enabled only when the value of HCR.{TGE, AMO} is {0, 1}.

The Guest OS cannot distinguish the virtual exception from the corresponding physical exception.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

VI, bit [7]

Virtual IRQ exception.

`0b0` This mechanism is not making a virtual IRQ pending.

`0b1` A virtual IRQ is pending because of this mechanism.

The virtual IRQ is enabled only when the value of HCR.{TGE, IMO} is {0, 1}.

The Guest OS cannot distinguish the virtual exception from the corresponding physical exception.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

VF, bit [6]

Virtual FIQ exception.

`0b0` This mechanism is not making a virtual FIQ pending.

`0b1` A virtual FIQ is pending because of this mechanism.

The virtual FIQ is enabled only when the value of HCR.{TGE, FMO} is {0, 1}.

The Guest OS cannot distinguish the virtual exception from the corresponding physical exception.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

AMO, bit [5]

SError interrupt Mask Override. When this bit is set to 1, it overrides the effect of CPSR.A, and enables virtual exception signaling by the VA bit.

If the value of HCR.TGE is 0, then virtual SError interrupts are enabled in Non-secure state.

If the value of HCR.TGE is 1, then in Non-secure state the HCR.AMO bit behaves as 1 for all purposes other than a direct read of the value of the bit.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

IMO, bit [4]

IRQ Mask Override. When this bit is set to 1, it overrides the effect of CPSR.I, and enables virtual exception signaling by the VI bit.

If the value of HCR.TGE is 0, then Virtual IRQ interrupts are enabled in the Non-secure state.

If the value of HCR.TGE is 1, then in Non-secure state the HCR.IMO bit behaves as 1 for all purposes other than a direct read of the value of the bit.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

FMO, bit [3]

FIQ Mask Override. When this bit is set to 1, it overrides the effect of CPSR.F, and enables virtual exception signaling by the VF bit.

If the value of HCR.TGE is 0, then Virtual FIQ interrupts are enabled in the Non-secure state.

If the value of HCR.TGE is 1, then in Non-secure state the HCR.FMO bit behaves as 1 for all purposes other than a direct read of the value of the bit.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

PTW, bit [2]

Protected Table Walk. In the Non-secure PL1&0 translation regime, a translation table access made as part of a stage 1 translation table walk is subject to a stage 2 translation. The combining of the memory type attributes from the two stages of translation means the access might be made to a type of Device memory. If this occurs then the value of this bit determines the behavior:

`0b0` The translation table walk occurs as if it is to Normal Non-cacheable memory. This means it can be made speculatively.

`0b1` The memory access generates a stage 2 Permission fault.

This field is permitted to be cached in a TLB.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

SWIO, bit [1]

Set/Way Invalidation Override. Causes Non-secure EL1 execution of the data cache invalidate by set/way instructions to perform a data cache clean and invalidate by set/way.

`0b0` This control has no effect on the operation of data cache invalidate by set/way instructions.

`0b1` Data cache invalidate by set/way instructions perform a data cache clean and invalidate by set/way.

When this bit is set to 1, DCISW performs the same invalidation as a DCCISW instruction.

As a result of changes to the behavior of DCISW, this bit is redundant in Armv8. This bit can be implemented as RES1.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

VM, bit [0]

Virtualization enable. Enables stage 2 address translation for the Non-secure EL1&0 translation regime.

`0b0` Non-secure EL1&0 stage 2 address translation disabled.

`0b1` Non-secure EL1&0 stage 2 address translation enabled.

If the HCR.DC bit is set to 1, then the behavior of the PE when executing in a Non-secure mode other than Hyp mode is consistent with HCR.VM being 1, regardless of the actual value of HCR.VM, other than the value returned by an explicit read of HCR.VM.

When the value of this bit is 1, data cache invalidate instructions executed at Non-secure EL1 perform a data cache clean and invalidate. For the invalidate by set/way instruction this behavior applies regardless of the value of the HCR.SWIO bit.

This bit is permitted to be cached in a TLB.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

Accessing the HCR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return HCR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        return HCR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    HCR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        HCR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.65 HCR2, Hyp Configuration Register 2

The HCR2 characteristics are:

Purpose

Provides additional configuration controls for virtualization.

Configurations

AArch32 System register HCR2[31:0] is architecturally mapped to AArch64 System register HCR_EL2[63:32].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to HCR2 are UNKNOWN.

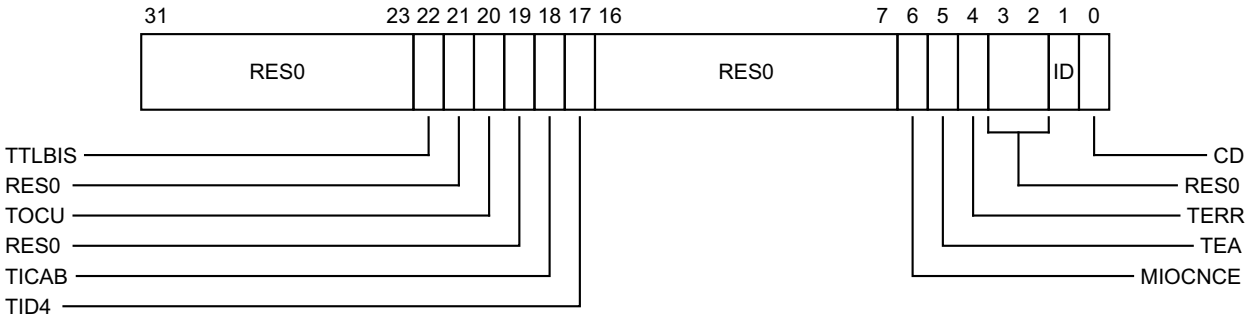
If EL2 is not implemented, this register is RES0 from EL3.

Attributes

HCR2 is a 32-bit register.

Field descriptions

The HCR2 bit assignments are:



Bits [31:23]

Reserved, RES0.

TTLBIS, bit [22]

When ARMv8.2-EVT is implemented:

Trap TLB maintenance instructions that operate on the Inner Shareable domain. Traps execution of the following TLB maintenance instructions at EL1 to EL2:

TLBIALLIS, TLBIMVAIS, TLBIASIDIS, TLBIMVAAIS, TLBIMVALIS, TLBIMVAALIS

`0b0` This control does not cause any instructions to be trapped.

`0b1` Non-secure EL1 execution of the specified TLB maintenance instructions is trapped to EL2.

If ARMv8.2-EVT is not implemented, this field is RES0.

When ARMv8.1-VHE and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

Otherwise:

Reserved, RES0.

Bit [21]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TOCU, bit [20]

When ARMv8.2-EVT is implemented:

Trap cache maintenance instructions that operate to the Point of Unification. Traps execution of those cache maintenance instructions at EL1 or EL0 using AArch64, and at EL1 using AArch32, to EL2.

This applies to the following instructions:

- When Non-secure EL0 is using AArch64, IC IVAU, DC CVAU. However, if the value of SCTLR_EL1.UCI is 0 these instructions are UNDEFINED at EL0 and any resulting exception is higher priority than this trap to EL2.
- When EL1 is using AArch64, IC IVAU, IC IALLU, DC CVAU.
- When Non-secure EL1 is using AArch32, ICIMVAU, ICIALLU, DCCMVAU.

Note

An exception generated because an instruction is UNDEFINED at EL0 is higher priority than this trap to EL2. In addition:

- IC IALLUIS and IC IALLU are always UNDEFINED at EL0 using AArch64.
- ICIMVAU, ICIALLU, ICIALLUIS, and DCCMVAU are always UNDEFINED at EL0 using AArch32.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Non-secure execution of the specified cache maintenance instructions is trapped to EL2.

If ARMv8.2-EVT is not implemented, this field is RES0.

If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean by VA to the Point of Unification instruction can be trapped when the value of this control is 1.

If the Point of Unification is before any level of instruction cache, it is IMPLEMENTATION DEFINED whether the execution of any instruction cache invalidate to the Point of Unification instruction can be trapped when the value of this control is 1.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

Otherwise:

Reserved, RES0.

Bit [19]

Reserved, RES0.

TICAB, bit [18]

When ARMv8.2-EVT is implemented:

Trap ICIALLUIS cache maintenance instructions. Traps execution of those cache maintenance instructions at EL1 to EL2.

This applies to the following instructions:

ICIALLUIS.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Non-secure EL1 execution of the specified cache maintenance instructions is trapped to EL2.

If ARMv8.2-EVT is not implemented, this field is RES0.

If the Point of Unification is before any level of instruction cache, it is IMPLEMENTATION DEFINED whether the execution of any instruction cache invalidate to the Point of Unification instruction can be trapped when the value of this control is 1.

When ARMv8.1-VHE and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

Reserved, RES0.

TID4, bit [17]

When ARMv8.2-EVT is implemented:

Trap ID group 4. Traps the following register accesses to EL2:

- EL1 reads of CCSIDR, CCSIDR2, CLIDR, and CSSELR.
- EL1 writes to CSSELR.

`0b0` This control does not cause any instructions to be trapped.

`0b1` The specified Non-secure EL1 and EL0 accesses to ID group 4 registers are trapped to EL2.

If ARMv8.2-EVT is not implemented, this field is RES0.

When ARMv8.1-VHE is implemented and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

Otherwise:

Reserved, RES0.

Bits [16:7]

Reserved, RES0.

MIOCNCE, bit [6]

Mismatched Inner/Outer Cacheable Non-Coherency Enable, for the Non-secure PL1&0 translation regime.

`0b0` For the Non-secure PL1&0 translation regime, for permitted accesses to a memory location that use a common definition of the Shareability and Cacheability of the location, there must be no loss of coherency if the Inner Cacheability attribute for those accesses differs from the Outer Cacheability attribute.

`0b1` For the Non-secure PL1&0 translation regime, for permitted accesses to a memory location that use a common definition of the Shareability and Cacheability of the location, there might be a loss of coherency if the Inner Cacheability attribute for those accesses differs from the Outer Cacheability attribute.

For more information see *Mismatched memory attributes* on page E2-4020.

This field can be implemented as RAZ/WI.

In a system where the PE resets into EL2 or EL3, this field resets to an architecturally UNKNOWN value.

TEA, bit [5]

Route synchronous External abort exceptions from EL0 and EL1 to EL2. If the RAS Extension is implemented, the possible values of this bit are:

`0b0` Does not route synchronous External abort exceptions from Non-secure EL0 and EL1 to EL2.

`0b1` Route synchronous External abort exceptions from Non-secure EL0 and EL1 to EL2, if not routed to EL3.

When the RAS Extension is not implemented, this field is RES0.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

TERR, bit [4]

When RAS is implemented:

Trap Error record accesses from EL1 to EL2. Trap accesses to the following registers from EL1 to EL2:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ERRIDR, ERRSELR, ERXADDR, ERXADDR2, ERXCTLR, ERXCTLR2, ERXFR, ERXFR2, ERXMISC0, ERXMISC1, ERXMISC2, ERXMISC3, and ERXSTATUS. When ARMv8.4-RAS is implemented, ERXMISC4, ERXMISC5, ERXMISC6, and ERXMISC7.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Accesses to the specified registers from EL1 generate a Trap exception to EL2.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

Otherwise:

Reserved, RES0.

Bits [3:2]

Reserved, RES0.

ID, bit [1]

Stage 2 Instruction access cacheability disable. For the Non-secure PL1&0 translation regime, when HCR.VM==1, this control forces all stage 2 translations for instruction accesses to Normal memory to be Non-cacheable.

`0b0` This control has no effect on stage 2 of the Non-secure PL1&0 translation regime.

`0b1` For the Non-secure PL1&0 translation regime, forces all stage 2 translations for instruction accesses to Normal memory to be Non-cacheable.

This bit has no effect on the EL2 translation regime.

On a Warm reset, in a system where the PE resets into EL2 or EL3, this field resets to `0`.

CD, bit [0]

Stage 2 Data access cacheability disable. When HCR.VM==1, this forces all stage 2 translations for data accesses and translation table walks to Normal memory to be Non-cacheable for the Non-secure PL1&0 translation regime.

`0b0` This control has no effect on stage 2 of the Non-secure PL1&0 translation regime for data accesses and translation table walks.

`0b1` For the Non-secure PL1&0 translation regime, forces all stage 2 translations for data accesses and translation table walks to Normal memory to be Non-cacheable.

This bit has no effect on the EL2 translation regime.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

Accessing the HCR2

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return HCR2;
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if SCR.NS == '0' then
        UNDEFINED;
    else
        return HCR2;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    HCR2 = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        HCR2 = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.66 HDFAR, Hyp Data Fault Address Register

The HDFAR characteristics are:

Purpose

Holds the virtual address of the faulting address that caused a synchronous Data Abort exception that is taken to Hyp mode.

Configurations

AArch32 System register HDFAR[31:0] is architecturally mapped to AArch64 System register FAR_EL2[31:0].

AArch32 System register HDFAR[31:0] is architecturally mapped to AArch32 System register DFAR[31:0] (S) when EL2 is implemented, EL3 is implemented and the highest implemented Exception level is using AArch32 state.

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to HDFAR are UNKNOWN.

If EL2 is not implemented, this register is RES0 from EL3.

Attributes

HDFAR is a 32-bit register.

Field descriptions

The HDFAR bit assignments are:

31 … 0
VA of faulting address of synchronous Data Abort exception taken to Hyp mode

Bits [31:0]

VA of faulting address of synchronous Data Abort exception taken to Hyp mode.

On a Prefetch Abort exception, this register is UNKNOWN.

Any execution in a Non-secure EL1 or Non-secure EL0 mode makes this register UNKNOWN.

This field resets to an architecturally UNKNOWN value.

Accessing the HDFAR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T6 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T6 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    return HDFAR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        return HDFAR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T6 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T6 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    HDFAR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        HDFAR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.67 HIFAR, Hyp Instruction Fault Address Register

The HIFAR characteristics are:

Purpose

Holds the virtual address of the faulting address that caused a synchronous Prefetch Abort exception that is taken to Hyp mode.

Configurations

AArch32 System register HIFAR[31:0] is architecturally mapped to AArch64 System register FAR_EL2[63:32].

AArch32 System register HIFAR[31:0] is architecturally mapped to AArch32 System register IFAR[31:0] (S) when EL2 is implemented, EL3 is implemented and the highest implemented Exception level is using AArch32 state.

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to HIFAR are UNKNOWN.

If EL2 is not implemented, this register is RES0 from EL3.

Attributes

HIFAR is a 32-bit register.

Field descriptions

The HIFAR bit assignments are:

31 … 0
VA of faulting address of synchronous Prefetch Abort exception taken to Hyp mode

Bits [31:0]

VA of faulting address of synchronous Prefetch Abort exception taken to Hyp mode.

On a Data Abort exception, this register is UNKNOWN.

Any execution in a Non-secure EL1 or Non-secure EL0 mode makes this register UNKNOWN.

This field resets to an architecturally UNKNOWN value.

Accessing the HIFAR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0110	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T6 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T6 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    return HIFAR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        return HIFAR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0110	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T6 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T6 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    HIFAR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        HIFAR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.68 HMAIR0, Hyp Memory Attribute Indirection Register 0

The HMAIR0 characteristics are:

Purpose

Along with HMAIR1, provides the memory attribute encodings corresponding to the possible AttrIndx values in a Long-descriptor format translation table entry for stage 1 translations for memory accesses from Hyp mode.

AttrIndx[2] indicates the HMAIR register to be used:

- When AttrIndx[2] is 0, HMAIR0 is used.
- When AttrIndx[2] is 1, HMAIR1 is used.

Configurations

AArch32 System register HMAIR0[31:0] is architecturally mapped to AArch64 System register MAIR_EL2[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to HMAIR0 are UNKNOWN.

If EL2 is not implemented, this register is RES0 from EL3.

Attributes

HMAIR0 is a 32-bit register.

Field descriptions

The HMAIR0 bit assignments are:

When TTBCR.EAE == 1:

31 – 24	23 – 16	15 – 8	7 – 0
Attr3	Attr2	Attr1	Attr0

Attr<n>, bits [8n+7:8n], for n = 0 to 3

The memory attribute encoding for an AttrIndx[2:0] entry in a Long descriptor format translation table entry, where:

- AttrIndx[2:0] gives the value of <n> in Attr<n>.
- AttrIndx[2] defines which MAIR to access. Attr7 to Attr4 are in MAIR1, and Attr3 to Attr0 are in MAIR0.

Bits [7:4] are encoded as follows:

Attr<n>[7:4]	Meaning
`0b0000`	Device memory. See encoding of Attr<n>[3:0] for the type of Device memory.
0b00RW, RW not `0b00`	Normal memory, Outer Write-Through Transient.
`0b0100`	Normal memory, Outer Non-cacheable.
0b01RW, RW not `0b00`	Normal memory, Outer Write-Back Transient.
0b10RW	Normal memory, Outer Write-Through Non-transient.
0b11RW	Normal memory, Outer Write-Back Non-transient.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

R = Outer Read-Allocate policy, W = Outer Write-Allocate policy.

The meaning of bits [3:0] depends on the value of bits [7:4]:

Attr<n>[3:0]	Meaning when Attr<n>[7:4] is 0b0000	Meaning when Attr<n>[7:4] is not 0b0000
0b0000	Device-nGnRnE memory	UNPREDICTABLE
0b00RW, RW not 0b00	UNPREDICTABLE	Normal memory, Inner Write-Through Transient
0b0100	Device-nGnRE memory	Normal memory, Inner Non-cacheable
0b01RW, RW not 0b00	UNPREDICTABLE	Normal memory, Inner Write-Back Transient
0b1000	Device-nGRE memory	Normal memory, Inner Write-Through Non-transient (RW=0b00)
0b10RW, RW not 0b00	UNPREDICTABLE	Normal memory, Inner Write-Through Non-transient
0b1100	Device-GRE memory	Normal memory, Inner Write-Back Non-transient (RW=0b00)
0b11RW, RW not 0b00	UNPREDICTABLE	Normal memory, Inner Write-Back Non-transient

R = Inner Read-Allocate policy, W = Inner Write-Allocate policy.

The R and W bits in some Attr<n> fields have the following meanings:

R or W	Meaning
0b0	No Allocate
0b1	Allocate

This field resets to an architecturally UNKNOWN value.

Accessing the HMAIR0

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1010	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T10 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T10 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return HMAIR0;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        return HMAIR0;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1010	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T10 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T10 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    HMAIR0 = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        HMAIR0 = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.69 HMAIR1, Hyp Memory Attribute Indirection Register 1

The HMAIR1 characteristics are:

Purpose

Along with HMAIR0, provides the memory attribute encodings corresponding to the possible AttrIndx values in a Long-descriptor format translation table entry for stage 1 translations for memory accesses from Hyp mode.

AttrIndx[2] indicates the HMAIR register to be used:

- When AttrIndx[2] is 0, HMAIR0 is used.
- When AttrIndx[2] is 1, HMAIR1 is used.

Configurations

AArch32 System register HMAIR1[31:0] is architecturally mapped to AArch64 System register MAIR_EL2[63:32].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to HMAIR1 are UNKNOWN.

If EL2 is not implemented, this register is RES0 from EL3.

Attributes

HMAIR1 is a 32-bit register.

Field descriptions

The HMAIR1 bit assignments are:

When TTBCR.EAE == 1:

31 – 24	23 – 16	15 – 8	7 – 0
Attr7	Attr6	Attr5	Attr4

Attr<n>, bits [8(n-4)+7:8(n-4)], for n = 4 to 7

The memory attribute encoding for an AttrIndx[2:0] entry in a Long descriptor format translation table entry, where:

- AttrIndx[2:0] gives the value of <n> in Attr<n>.
- AttrIndx[2] defines which MAIR to access. Attr7 to Attr4 are in MAIR1, and Attr3 to Attr0 are in MAIR0.

Bits [7:4] are encoded as follows:

Attr<n>[7:4]	Meaning
`0b0000`	Device memory. See encoding of Attr<n>[3:0] for the type of Device memory.
0b00RW, RW not `0b00`	Normal memory, Outer Write-Through Transient.
`0b0100`	Normal memory, Outer Non-cacheable.
0b01RW, RW not `0b00`	Normal memory, Outer Write-Back Transient.
0b10RW	Normal memory, Outer Write-Through Non-transient.
0b11RW	Normal memory, Outer Write-Back Non-transient.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

R = Outer Read-Allocate policy, W = Outer Write-Allocate policy.

The meaning of bits [3:0] depends on the value of bits [7:4]:

Attr<n>[3:0]	Meaning when Attr<n>[7:4] is `0b0000`	Meaning when Attr<n>[7:4] is not `0b0000`
`0b0000`	Device-nGnRnE memory	UNPREDICTABLE
0b00RW, RW not `0b00`	UNPREDICTABLE	Normal memory, Inner Write-Through Transient
`0b0100`	Device-nGnRE memory	Normal memory, Inner Non-cacheable
0b01RW, RW not `0b00`	UNPREDICTABLE	Normal memory, Inner Write-Back Transient
`0b1000`	Device-nGRE memory	Normal memory, Inner Write-Through Non-transient (RW=`0b00`)
0b10RW, RW not `0b00`	UNPREDICTABLE	Normal memory, Inner Write-Through Non-transient
`0b1100`	Device-GRE memory	Normal memory, Inner Write-Back Non-transient (RW=`0b00`)
0b11RW, RW not `0b00`	UNPREDICTABLE	Normal memory, Inner Write-Back Non-transient

R = Inner Read-Allocate policy, W = Inner Write-Allocate policy.

The R and W bits in some Attr<n> fields have the following meanings:

R or W	Meaning
`0b0`	No Allocate
`0b1`	Allocate

This field resets to an architecturally UNKNOWN value.

Accessing the HMAIR1

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1010	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T10 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T10 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return HMAIR1;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        return HMAIR1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1010	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T10 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T10 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    HMAIR1 = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        HMAIR1 = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.70 HPFAR, Hyp IPA Fault Address Register

The HPFAR characteristics are:

Purpose

Holds the faulting IPA for some aborts on a stage 2 translation taken to Hyp mode.

Configurations

AArch32 System register HPFAR[31:0] is architecturally mapped to AArch64 System register HPFAR_EL2[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to HPFAR are UNKNOWN.

If EL2 is not implemented, this register is RES0 from EL3.

Attributes

HPFAR is a 32-bit register.

Field descriptions

The HPFAR bit assignments are:

31 … 4	3 … 0
FIPA[39:12]	RES0

Execution in any Non-secure mode other than Hyp mode makes this register UNKNOWN.

FIPA[39:12], bits [31:4]

Bits [39:12] of the faulting intermediate physical address.

This field resets to an architecturally UNKNOWN value.

Bits [3:0]

Reserved, RES0.

Accessing the HPFAR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0110	0b0000	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T6 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T6 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return HPFAR;
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if SCR.NS == '0' then
        UNDEFINED;
    else
        return HPFAR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0110	0b0000	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T6 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T6 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    HPFAR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        HPFAR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.71 HRMR, Hyp Reset Management Register

The HRMR characteristics are:

Purpose

If EL2 is the highest implemented Exception level and this register is implemented:

- A write to the register at EL2 can request a Warm reset.
- If EL2 can use AArch32 and AArch64, this register specifies the Execution state that the PE boots into on a Warm reset.

Configurations

AArch32 System register HRMR[31:0] is architecturally mapped to AArch64 System register RMR_EL2[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to HRMR are UNKNOWN.

Only implemented if EL2 is the highest implemented Exception level. In this case:

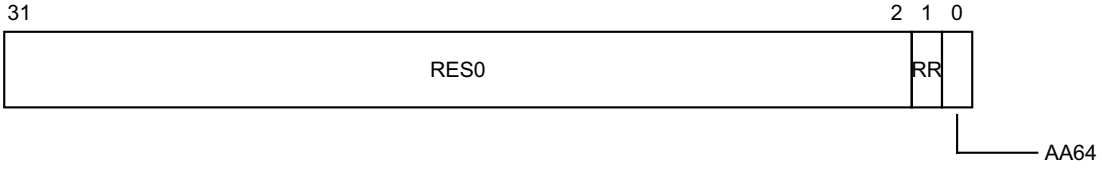
- If EL2 can use AArch32 and AArch64 then this register must be implemented.
- If EL2 cannot use AArch64 then it is IMPLEMENTATION DEFINED whether the register is implemented.

Attributes

HRMR is a 32-bit register.

Field descriptions

The HRMR bit assignments are:



Bits [31:2]

Reserved, RES0.

RR, bit [1]

Reset Request. Setting this bit to 1 requests a Warm reset.

This field resets to `0`.

AA64, bit [0]

When EL2 can use AArch64, determines which Execution state the PE boots into after a Warm reset:

`0b0` AArch32.

`0b1` AArch64.

On coming out of the Warm reset, execution starts at the IMPLEMENTATION DEFINED reset vector address of the specified Execution state.

If EL2 cannot use AArch64 this bit is RAZ/WI.

When implemented as a RW field, this field resets to 0 on a Cold reset.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the HRMR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1100	0b0000	0b010

```
if PSTATE.EL == EL1 && EL2Enabled() && IsHighestEL(EL2) && !ELUsingAArch32(EL2) && HSTR_EL2.T12 == '1'
then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif PSTATE.EL == EL1 && EL2Enabled() && IsHighestEL(EL2) && ELUsingAArch32(EL2) && HSTR.T12 == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif PSTATE.EL == EL2 && IsHighestEL(EL2) then
    return HRMR;
else
    UNDEFINED;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1100	0b0000	0b010

```
if PSTATE.EL == EL1 && EL2Enabled() && IsHighestEL(EL2) && !ELUsingAArch32(EL2) && HSTR_EL2.T12 == '1'
then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif PSTATE.EL == EL1 && EL2Enabled() && IsHighestEL(EL2) && ELUsingAArch32(EL2) && HSTR.T12 == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif PSTATE.EL == EL2 && IsHighestEL(EL2) then
    HRMR = R[t];
else
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.72 HSCTLR, Hyp System Control Register

The HSCTLR characteristics are:

Purpose

Provides top level control of the system operation in Hyp mode.

Configurations

AArch32 System register HSCTLR[31:0] is architecturally mapped to AArch64 System register SCTLR_EL2[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to HSCTLR are UNKNOWN.

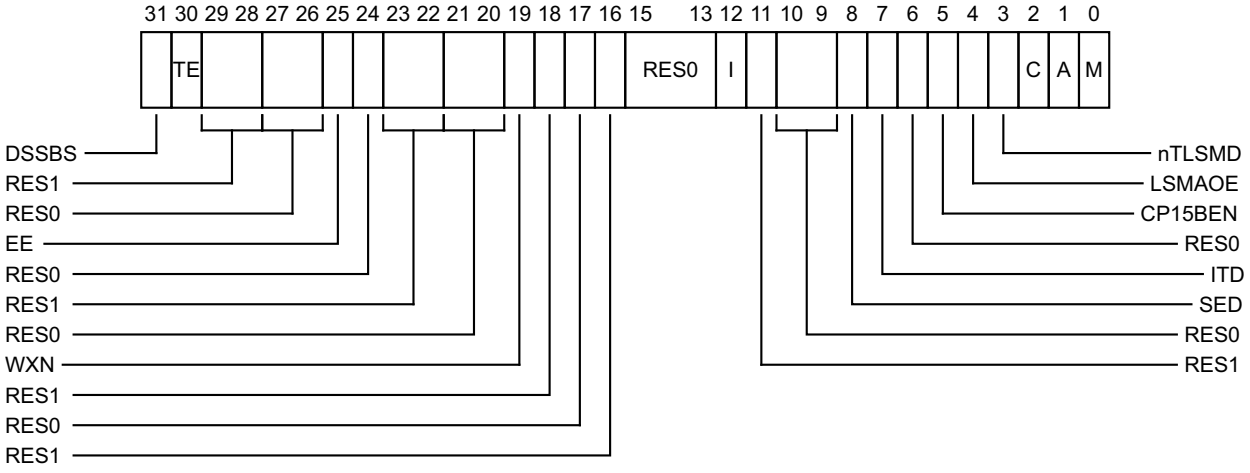
If EL2 is not implemented, this register is RES0 from EL3.

Attributes

HSCTLR is a 32-bit register.

Field descriptions

The HSCTLR bit assignments are:



DSSBS, bit [31]

When ARMv8.0-SSBS is implemented:

Default PSTATE.SSBS value on Exception Entry. The defined values are:

`0b0` PSTATE.SSBS is set to 0 on an exception to Hyp mode.

`0b1` PSTATE.SSBS is set to 1 on an exception to Hyp mode.

In a system where the PE resets into EL2, this field resets to an IMPLEMENTATION DEFINED value.

Otherwise:

Reserved, RES0.

TE, bit [30]

T32 Exception Enable. This bit controls whether exceptions to EL2 are taken to A32 or T32 state:

`0b0` Exceptions, including reset, taken to A32 state.

`0b1` Exceptions, including reset, taken to T32 state.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bits [29:28]

Reserved, RES1.

Bits [27:26]

Reserved, RES0.

EE, bit [25]

The value of the PSTATE.E bit on entry to Hyp mode, the endianness of stage 1 translation table walks in the EL2 translation regime, and the endianness of stage 2 translation table walks in the PL1&0 translation regime.

The possible values of this bit are:

`0b0` Little-endian. PSTATE.E is cleared to 0 on entry to Hyp mode. Stage 1 translation table walks in the EL2 translation regime, and stage 2 translation table walks in the PL1&0 translation regime are little-endian.

`0b1` Big-endian. PSTATE.E is set to 1 on entry to Hyp mode. Stage 1 translation table walks in the EL2 translation regime, and stage 2 translation table walks in the PL1&0 translation regime are big-endian.

If an implementation does not provide Big-endian support at Exception Levels higher than EL0, this bit is RES0.

If an implementation does not provide Little-endian support at Exception Levels higher than EL0, this bit is RES1.

In a system where the PE resets into EL2, this field resets to an IMPLEMENTATION DEFINED value.

Bit [24]

Reserved, RES0.

Bits [23:22]

Reserved, RES1.

Bits [21:20]

Reserved, RES0.

WXN, bit [19]

Write permission implies XN (Execute-never). For the EL2 translation regime, this bit can force all memory regions that are writable to be treated as XN. The possible values of this bit are:

`0b0` This control has no effect on memory access permissions.

`0b1` Any region that is writable in the EL2 translation regime is forced to XN for accesses from software executing at EL2.

This bit applies only when HSCTLR.M bit is set.

The WXN bit is permitted to be cached in a TLB.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Bit [18]

Reserved, RES1.

Bit [17]

Reserved, RES0.

Bit [16]

Reserved, RES1.

Bits [15:13]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I, bit [12]

Instruction access Cacheability control, for accesses at EL2:

`0b0` All instruction access to Normal memory from EL2 are Non-cacheable for all levels of instruction and unified cache.

If the value of HSCTLR.M is 0, instruction accesses from stage 1 of the EL2 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.

`0b1` All instruction access to Normal memory from EL2 can be cached at all levels of instruction and unified cache.

If the value of HSCTLR.M is 0, instruction accesses from stage 1 of the EL2 translation regime are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.

This bit has no effect on the PL1&0 translation regime.

In a system where the PE resets into EL2, this field resets to `0`.

Bit [11]

Reserved, RES1.

Bits [10:9]

Reserved, RES0.

SED, bit [8]

SETEND instruction disable. Disables SETEND instructions at EL2.

`0b0` SETEND instruction execution is enabled at EL2.

`0b1` SETEND instructions are UNDEFINED at EL2.

If the implementation does not support mixed-endian operation at EL2, this bit is RES1.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

ITD, bit [7]

IT Disable. Disables some uses of IT instructions at EL2.

`0b0` All IT instruction functionality is enabled at EL2.

`0b1` Any attempt at EL2 to execute any of the following is UNDEFINED:

- All encodings of the IT instruction with hw1[3:0]!=1000.
- All encodings of the subsequent instruction with the following values for hw1:
  - 11xxxxxxxxxxxxxx: All 32-bit instructions, and the 16-bit instructions B, UDF, SVC, LDM, and STM.
  - 1011xxxxxxxxxxxx: All instructions in *Miscellaneous 16-bit instructions* on page F3-4115.
  - 10100xxxxxxxxxxx: ADD Rd, PC, #imm
  - 01001xxxxxxxxxxx: LDR Rd, [PC, #imm]
  - 0100x1xxx1111xxx: ADD Rdn, PC; CMP Rn, PC; MOV Rd, PC; BX PC; BLX PC.
  - 010001xx1xxxx111: ADD PC, Rm; CMP PC, Rm; MOV PC, Rm. This pattern also covers unpredictable cases with BLX Rn.

These instructions are always UNDEFINED, regardless of whether they would pass or fail the condition code check that applies to them as a result of being in an IT block.

It is IMPLEMENTATION DEFINED whether the IT instruction is treated as:

- A 16-bit instruction, that can only be followed by another 16-bit instruction.
- The first half of a 32-bit instruction.

This means that, for the situations that are UNDEFINED, either the second 16-bit instruction or the 32-bit instruction is UNDEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

An implementation might vary dynamically as to whether IT is treated as a 16-bit instruction or the first half of a 32-bit instruction.

If an instruction in an active IT block that would be disabled by this field sets this field to 1 then behavior is CONSTRAINED UNPREDICTABLE. For more information see *Changes to an ITD control by an instruction in an IT block* on page E1-3958

ITD is optional, but if it is implemented in the SCTLR then it must also be implemented in the HSCTLR. If it is not implemented then this bit is RAZ/WI.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Bit [6]

Reserved, RES0.

CP15BEN, bit [5]

System instruction memory barrier enable. Enables accesses to the DMB, DSB, and ISB System instructions in the (coproc==`0b1111`) encoding space from EL2:

`0b0` EL2 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is UNDEFINED.

`0b1` EL2 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is enabled.

CP15BEN is optional, but if it is implemented in the SCTLR then it must also be implemented in the HSCTLR. If it is not implemented then this bit is RAO/WI.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

LSMAOE, bit [4]

When ARMv8.2-LSMAOC is implemented:

Load Multiple and Store Multiple Atomicity and Ordering Enable.

`0b0` For all memory accesses at EL2, A32 and T32 Load Multiple and Store Multiple can have an interrupt taken during the sequence memory accesses, and the memory accesses are not required to be ordered.

`0b1` The ordering and interrupt behavior of A32 and T32 Load Multiple and Store Multiple at EL2 is as defined for Armv8.0.

This bit is permitted to be cached in a TLB.

In a system where the PE resets into EL2, this field resets to `1`.

Otherwise:

Reserved, RES1.

nTLSMD, bit [3]

When ARMv8.2-LSMAOC is implemented:

No Trap Load Multiple and Store Multiple to Device-nGRE/Device-nGnRE/Device-nGnRnE memory.

`0b0` All memory accesses by A32 and T32 Load Multiple and Store Multiple at EL2 that are marked at stage 1 as Device-nGRE/Device-nGnRE/Device-nGnRnE memory are trapped and generate a stage 1 Alignment fault.

`0b1` All memory accesses by A32 and T32 Load Multiple and Store Multiple at EL2 that are marked at stage 1 as Device-nGRE/Device-nGnRE/Device-nGnRnE memory are not trapped.

This bit is permitted to be cached in a TLB.

In a system where the PE resets into EL2, this field resets to `1`.

Otherwise:

Reserved, RES1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

C, bit [2]

Cacheability control, for data accesses at EL2:

`0b0` All data access to Normal memory from EL2, and all accesses to the EL2 translation tables, are Non-cacheable for all levels of data and unified cache.

`0b1` All data access to Normal memory from EL2, and all accesses to the EL2 translation tables, can be cached at all levels of data and unified cache.

This bit has no effect on the PL1&0 translation regime.

In a system where the PE resets into EL2, this field resets to `0`.

A, bit [1]

Alignment check enable. This is the enable bit for Alignment fault checking at EL2:

`0b0` Alignment fault checking disabled when executing at EL2.

Instructions that load or store one or more registers, other than load/store exclusive and load-acquire/store-release, do not check that the address being accessed is aligned to the size of the data element or data elements being accessed.

`0b1` Alignment fault checking enabled when executing at EL2.

All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element or data elements being accessed. If this check fails it causes an Alignment fault, which is taken as a Data Abort exception.

Load/store exclusive and load-acquire/store-release instructions have an alignment check regardless of the value of the A bit.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

M, bit [0]

MMU enable for EL2 stage 1 address translation. Possible values of this bit are:

`0b0` EL2 stage 1 address translation disabled.

See the HSCTLR.I field for the behavior of instruction accesses to Normal memory.

`0b1` EL2 stage 1 address translation enabled.

In a system where the PE resets into EL2, this field resets to `0`.

Accessing the HSCTLR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return HSCTLR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        return HSCTLR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    HSCTLR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        HSCTLR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.73 HSR, Hyp Syndrome Register

The HSR characteristics are:

Purpose

Holds syndrome information for an exception taken to Hyp mode.

Configurations

AArch32 System register HSR[31:0] is architecturally mapped to AArch64 System register ESR_EL2[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to HSR are UNKNOWN.

If EL2 is not implemented, this register is RES0 from EL3.

Attributes

HSR is a 32-bit register.

Field descriptions

The HSR bit assignments are:

31 – 26	25	24 – 0
EC	IL	ISS

Execution in any Non-secure PE mode other than Hyp mode makes this register UNKNOWN.

When an UNPREDICTABLE instruction is treated as UNDEFINED, and the exception is taken to EL2, the value of HSR is UNKNOWN. The value written to HSR must be consistent with a value that could be created as a result of an exception from the same Exception level that generated the exception as a result of a situation that is not UNPREDICTABLE at that Exception level, in order to avoid the possibility of a privilege violation.

EC, bits [31:26]

Exception Class. Indicates the reason for the exception that this register holds information about. Possible values of this field are:

EC == 0b000000

Unknown reason.

See ISS encoding for exceptions with an unknown reason.

EC == 0b000001

Trapped WFI or WFE instruction execution.

Conditional WFE and WFI instructions that fail their condition code check do not cause an exception.

See ISS encoding for an exception from a WFI or WFE instruction.

EC == 0b000011

Trapped MCR or MRC access with (coproc==`0b1111`) that is not reported using EC `0b000000`.

See ISS encoding for an exception from an MCR or MRC access.

EC == 0b000100

Trapped MCRR or MRRC access with (coproc==`0b1111`) that is not reported using EC `0b000000`.

See ISS encoding for an exception from an MCRR or MRRC access.

EC == 0b000101

Trapped MCR or MRC access with (coproc==`0b1110`).

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

See ISS encoding for an exception from an MCR or MRC access.

EC == 0b000110

Trapped LDC or STC access.

The only architected uses of these instructions are:

- An STC to write data to memory from DBGDTRRXint.
- An LDC to read data from memory to DBGDTRTXint.

See ISS encoding for an exception from an LDC or STC instruction.

EC == 0b000111

Access to Advanced SIMD or floating-point functionality trapped by a HCPTR.{TASE, TCP10} control.

Excludes exceptions generated because Advanced SIMD and floating-point are not implemented. These are reported with EC value `0b000000`.

See ISS encoding for an exception from an access to SIMD or floating-point functionality, resulting from HCPTR.

EC == 0b001000

Trapped VMRS access, from ID group trap, that is not reported using EC `0b000111`.

See ISS encoding for an exception from an MCR or MRC access.

EC == 0b001100

Trapped MRRC access with (coproc==`0b1110`).

See ISS encoding for an exception from an MCRR or MRRC access.

EC == 0b001110

Illegal exception return to AArch32 state.

See ISS encoding for an exception from an Illegal state or PC alignment fault.

EC == 0b010001

Exception on SVC instruction execution in AArch32 state routed to EL2.

See ISS encoding for an exception from HVC or SVC instruction execution.

EC == 0b010010

HVC instruction execution in AArch32 state, when HVC is not disabled.

See ISS encoding for an exception from HVC or SVC instruction execution.

EC == 0b010011

Trapped execution of SMC instruction in AArch32 state.

See ISS encoding for an exception from SMC instruction execution.

EC == 0b100000

Prefetch Abort from a lower Exception level.

See ISS encoding for an exception from a Prefetch Abort.

EC == 0b100001

Prefetch Abort taken without a change in Exception level.

See ISS encoding for an exception from a Prefetch Abort.

EC == 0b100010

PC alignment fault exception.

See ISS encoding for an exception from an Illegal state or PC alignment fault.

EC == 0b100100

Data Abort from a lower Exception level.

See ISS encoding for an exception from a Data Abort.

EC == 0b100101

Data Abort taken without a change in Exception level.

See ISS encoding for an exception from a Data Abort.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

All other EC values are reserved by Arm, and:

- Unused values in the range `0b000000` - `0b101100` (`0x00` - `0x2C`) are reserved for future use for synchronous exceptions.
- Unused values in the range `0b101101` - `0b111111` (`0x2D` - `0x3F`) are reserved for future use, and might be used for synchronous or asynchronous exceptions.

The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-7871.

This field resets to an architecturally UNKNOWN value.

IL, bit [25]

Instruction length bit. Indicates the size of the instruction that has been trapped to Hyp mode. When this bit is valid, possible values of this bit are:

`0b0` 16-bit instruction trapped.

`0b1` 32-bit instruction trapped.

This field is RES1 and not valid for the following cases:

- When the EC field is `0b000000`, indicating an exception with an unknown reason.
- Prefetch Aborts.
- Data Aborts for which the HSR.ISS.ISV field is 0.
- When the EC value is `0b001110`, indicating an Illegal state exception.

Note

This is a change from the behavior in Armv7, where the IL field is UNK/SBZP for the corresponding cases.

The IL field is not valid and is UNKNOWN on an exception from a PC alignment fault.

This field resets to an architecturally UNKNOWN value.

ISS, bits [24:0]

Instruction Specific Syndrome. Architecturally, this field can be defined independently for each defined Exception class. However, in practice, some ISS encodings are used for more than one Exception class.

The following subsections describe each ISS format.

ISS encoding for exceptions with an unknown reason

24 … 0
RES0

Bits [24:0]

Reserved, RES0.

This EC code is used for all exceptions that are not covered by any other EC value. This includes exceptions that are generated in the following situations:

- The attempted execution of an instruction bit pattern that has no allocated instruction or is not accessible in the current PE mode in the current Security state, including:
  - A read access using a System register encoding pattern that is not allocated for reads or that does not permit reads in the current PE mode and Security state.
  - A write access using a System register encoding pattern that is not allocated for writes or that does not permit writes in the current PE mode and Security state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

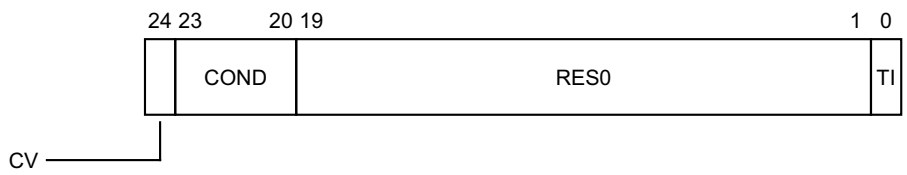
  - Instruction encodings that are unallocated.
  - Instruction encodings for instructions not implemented in the implementation.
- In Debug state, the attempted execution of an instruction bit pattern that not accessible in Debug state.
- In Non-debug state, the attempted execution of an instruction bit pattern that not accessible in Non-debug state.
- The attempted execution of a short vector floating-point instruction.
- In an implementation that does not include Advanced SIMD and floating-point functionality, an attempted access to Advanced SIMD or floating-point functionality under conditions where that access would be permitted if that functionality was present. This includes the attempted execution of an Advanced SIMD or floating-point instruction, and attempted accesses to Advanced SIMD and floating-point System registers.
- An exception generated because of the value of one of the SCTLR.{ITD, SED, CP15BEN} control bits.
- Attempted execution of:
  - An HVC instruction when disabled by HCR.HCD, SCR.HCE, or SCR_EL3.HCE.
  - An SMC instruction when disabled by SCR.SCD or SCR_EL3.SMD.
  - An HLT instruction when disabled by EDSCR.HDE.
- An HVC instruction when disabled by HCR.HCD, SCR.HCE, or SCR_EL3.HCE.An SMC instruction when disabled by SCR.SCD or SCR_EL3.SMD.An HLT instruction when disabled by EDSCR.HDE.
- An exception generated because of the attempted execution of an MSR (Banked register) or MRS (Banked register) instruction that would access a Banked register that is not accessible from the Security state and PE mode at which the instruction was executed.

An exception is generated only if the CONSTRAINED UNPREDICTABLE behavior of the instruction is that it is UNDEFINED, see *MSR (banked register) and MRS (banked register)* on page K1-7853.

- Attempted execution, in Debug state, of:
  - A DCPS1 instruction in Non-secure state from EL0 when EL2 is using AArch32 and the value of HCR.TGE is 1.
  - A DCPS2 instruction at EL1 or EL0 when EL2 is not implemented, or when EL3 is using AArch32 and the value of SCR.NS is 0, or when EL3 is using AArch64 and the value of SCR_EL3.NS is 0.
  - A DCPS3 instruction when EL3 is not implemented, or when the value of EDSCR.SDD is 1.
- In Debug state when the value of EDSCR.SDD is 1, the attempted execution at EL2, EL1, or EL0 of an instruction that is configured to trap to EL3.

Undefined Instruction exception, when the value of HCR.TGE is 1 on page G1-5719 describes the configuration settings for a trap that returns an HSR.EC value of `0b000000`.

ISS encoding for an exception from a WFI or WFE instruction



CV, bit [24]

Condition code valid. Possible values of this bit are:

`0b0` The COND field is not valid.

`0b1` The COND field is valid.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When an A32 instruction is trapped, CV is set to 1.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction.

When an A32 instruction is trapped, CV is set to 1 and:

- If the instruction is conditional, COND is set to the condition code field value from the instruction.
- If the instruction is unconditional, COND is set to `0b1110`.

A conditional A32 instruction that is known to pass its condition code check can be presented either:

- With COND set to `0b1110`, the value for unconditional.
- With the COND value held in the instruction.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:

- CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
- CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

This field resets to an architecturally UNKNOWN value.

Bits [19:1]

Reserved, RES0.

TI, bit [0]

Trapped instruction. Possible values of this bit are:

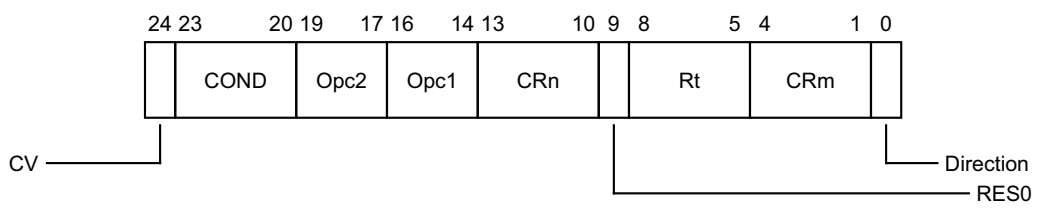
`0b0` WFI trapped.

`0b1` WFE trapped.

This field resets to an architecturally UNKNOWN value.

Traps to Hyp mode of Non-secure EL0 and EL1 execution of WFE and WFI instructions on page G1-5797 describes the configuration settings for this trap.

ISS encoding for an exception from an MCR or MRC access



CV, bit [24]

Condition code valid. Possible values of this bit are:

`0b0` The COND field is not valid.

`0b1` The COND field is valid.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When an A32 instruction is trapped, CV is set to 1.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction.

When an A32 instruction is trapped, CV is set to 1 and:

- If the instruction is conditional, COND is set to the condition code field value from the instruction.
- If the instruction is unconditional, COND is set to `0b1110`.

A conditional A32 instruction that is known to pass its condition code check can be presented either:

- With COND set to `0b1110`, the value for unconditional.
- With the COND value held in the instruction.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:

- CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
- CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

This field resets to an architecturally UNKNOWN value.

Opc2, bits [19:17]

The Opc2 value from the issued instruction.

For a trapped VMRS access, holds the value `0b000`.

This field resets to an architecturally UNKNOWN value.

Opc1, bits [16:14]

The Opc1 value from the issued instruction.

For a trapped VMRS access, holds the value `0b111`.

This field resets to an architecturally UNKNOWN value.

CRn, bits [13:10]

The CRn value from the issued instruction.

For a trapped VMRS access, holds the reg field from the VMRS instruction encoding.

This field resets to an architecturally UNKNOWN value.

Bit [9]

Reserved, RES0.

Rt, bits [8:5]

The Rt value from the issued instruction, the general-purpose register used for the transfer.

This field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

For a trapped VMRS access, holds the value `0b0000`.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Direction, bit [0]

Indicates the direction of the trapped instruction. The possible values of this bit are:

`0b0` Write to System register space. MCR instruction.

`0b1` Read from System register space. MRC or VMRS instruction.

This field resets to an architecturally UNKNOWN value.

The following sections describe configuration settings for traps that are reported using EC value `0b000011`:

- *Traps to Hyp mode of Non-secure EL0 and EL1 accesses to the ID registers* on page G1-5794.
- *Traps to Hyp mode of Non-secure EL0 and EL1 accesses to lockdown, DMA, and TCM operations* on page G1-5793.
- *Traps to Hyp mode of Non-secure EL1 execution of cache maintenance instructions* on page G1-5792.
- *Traps to Hyp mode of Non-secure EL1 execution of TLB maintenance instructions* on page G1-5791.
- *Traps to Hyp mode of Non-secure EL1 accesses to the Auxiliary Control Register* on page G1-5792.
- *Traps to Hyp mode of Non-secure EL0 and EL1 accesses to Performance Monitors registers* on page G1-5805.
- *Traps to Hyp mode of Non-secure EL0 and EL1 accesses to Activity Monitors registers* on page G1-5797.
- *Traps to Hyp mode of Non-secure EL1 accesses to the CPACR* on page G1-5799.
- *Traps to Hyp mode of Non-secure EL1 accesses to virtual memory control registers* on page G1-5790.
- *General trapping to Hyp mode of Non-secure EL0 and EL1 accesses to System registers in the (coproc==0b1111) encoding space* on page G1-5801.

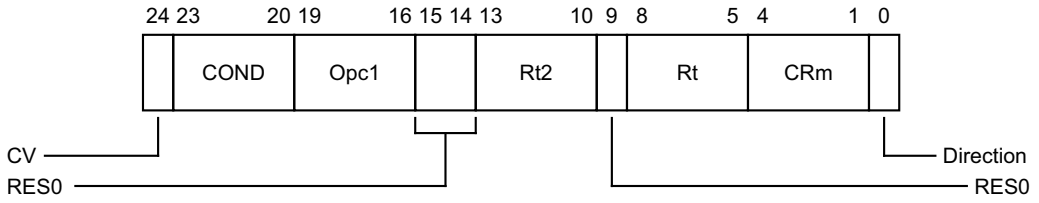
The following sections describe configuration settings for traps that are reported using EC value `0b000101`:

- *ID group 0, Primary device identification registers* on page G1-5795.
- *Traps to Hyp mode of Non-secure System register accesses to trace registers* on page G1-5800.
- *Trapping Non-secure System register accesses to Debug ROM registers* on page G1-5803.
- *Trapping Non-secure System register accesses to powerdown debug registers* on page G1-5803.
- *Trapping general Non-secure System register accesses to debug registers* on page G1-5803.

The following sections describes configuration settings for traps that are reported using EC value `0b001000`:

- *ID group 0, Primary device identification registers* on page G1-5795.
- *ID group 3, Detailed feature identification registers* on page G1-5796.

ISS encoding for an exception from an MCRR or MRRC access



CV, bit [24]

Condition code valid. Possible values of this bit are:

`0b0` The COND field is not valid.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` The COND field is valid.

When an A32 instruction is trapped, CV is set to 1.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction.

When an A32 instruction is trapped, CV is set to 1 and:

- If the instruction is conditional, COND is set to the condition code field value from the instruction.
- If the instruction is unconditional, COND is set to `0b1110`.

A conditional A32 instruction that is known to pass its condition code check can be presented either:

- With COND set to `0b1110`, the value for unconditional.
- With the COND value held in the instruction.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:

- CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
- CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

This field resets to an architecturally UNKNOWN value.

Opc1, bits [19:16]

The Opc1 value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Bits [15:14]

Reserved, RES0.

Rt2, bits [13:10]

The Rt2 value from the issued instruction, the second general-purpose register used for the transfer.

This field resets to an architecturally UNKNOWN value.

Bit [9]

Reserved, RES0.

Rt, bits [8:5]

The Rt value from the issued instruction, the first general-purpose register used for the transfer.

This field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction. The possible values of this bit are:

`0b0` Write to System register space. MCRR instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` Read from System register space. MRRC instruction.

This field resets to an architecturally UNKNOWN value.

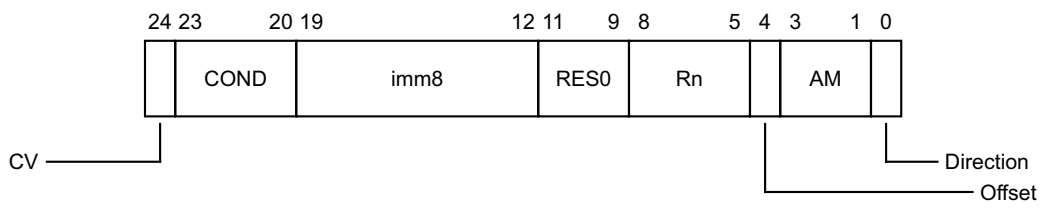
The following sections describe configuration settings for traps that are reported using EC value `0b000100`:

- *Traps to Hyp mode of Non-secure EL1 accesses to virtual memory control registers* on page G1-5790.
- *Traps to Hyp mode of Non-secure EL0 and EL1 accesses to Performance Monitors registers* on page G1-5805.
- *Traps to Hyp mode of Non-secure EL0 and EL1 accesses to Activity Monitors registers* on page G1-5797.
- *Traps to Hyp mode of Non-secure EL0 and EL1 accesses to the Generic Timer registers* on page G1-5804.
- *General trapping to Hyp mode of Non-secure EL0 and EL1 accesses to System registers in the (coproc==0b1111) encoding space* on page G1-5801.

The following sections describe configuration settings for traps that are reported using EC value `0b001100`:

- *Traps to Hyp mode of Non-secure System register accesses to trace registers* on page G1-5800.
- *Trapping Non-secure System register accesses to Debug ROM registers* on page G1-5803.

ISS encoding for an exception from an LDC or STC instruction



CV, bit [24]

Condition code valid. Possible values of this bit are:

`0b0` The COND field is not valid.

`0b1` The COND field is valid.

When an A32 instruction is trapped, CV is set to 1.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction.

When an A32 instruction is trapped, CV is set to 1 and:

- If the instruction is conditional, COND is set to the condition code field value from the instruction.
- If the instruction is unconditional, COND is set to `0b1110`.

A conditional A32 instruction that is known to pass its condition code check can be presented either:

- With COND set to `0b1110`, the value for unconditional.
- With the COND value held in the instruction.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:

- CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

This field resets to an architecturally UNKNOWN value.

imm8, bits [19:12]

The immediate value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Bits [11:9]

Reserved, RES0.

Rn, bits [8:5]

The Rn value from the issued instruction. Valid only when AM[2] is 0, indicating an immediate form of the LDC or STC instruction.

When AM[2] is 1, indicating a literal form of the LDC or STC instruction, this field is UNKNOWN.

This field resets to an architecturally UNKNOWN value.

Offset, bit [4]

Indicates whether the offset is added or subtracted:

`0b0` Subtract offset.

`0b1` Add offset.

This bit corresponds to the U bit in the instruction encoding.

This field resets to an architecturally UNKNOWN value.

AM, bits [3:1]

Addressing mode. The permitted values of this field are:

`0b000` Immediate unindexed.

`0b001` Immediate post-indexed.

`0b010` Immediate offset.

`0b011` Immediate pre-indexed.

`0b100` Literal unindexed.
LDC instruction in A32 instruction set only.
For a trapped STC instruction or a trapped T32 LDC instruction this encoding is reserved.

`0b110` Literal offset.
LDC instruction only.
For a trapped STC instruction, this encoding is reserved.

The values `0b101` and `0b111` are reserved. The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-7853.

Bit [2] in this subfield indicates the instruction form, immediate or literal.

Bits [1:0] in this subfield correspond to the bits {P, W} in the instruction encoding.

This field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction. The possible values of this bit are:

`0b0` Write to memory. STC instruction.

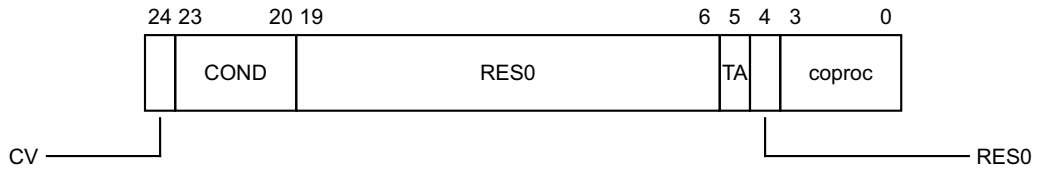
Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` Read from memory. LDC instruction.

This field resets to an architecturally UNKNOWN value.

Trapping general Non-secure System register accesses to debug registers on page G1-5803 describes the configuration settings for the trap that is reported using EC value `0b000110`.

ISS encoding for an exception from an access to SIMD or floating-point functionality, resulting from HCPTR



Excludes exceptions that occur because Advanced SIMD and floating-point functionality is not implemented, or because the value of HCR.TGE or HCR_EL2.TGE is 1. These are reported with EC value `0b000000`.

CV, bit [24]

Condition code valid. Possible values of this bit are:

`0b0` The COND field is not valid.

`0b1` The COND field is valid.

When an A32 instruction is trapped, CV is set to 1.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction.

When an A32 instruction is trapped, CV is set to 1 and:

- If the instruction is conditional, COND is set to the condition code field value from the instruction.
- If the instruction is unconditional, COND is set to `0b1110`.

A conditional A32 instruction that is known to pass its condition code check can be presented either:

- With COND set to `0b1110`, the value for unconditional.
- With the COND value held in the instruction.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:

- CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
- CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

This field resets to an architecturally UNKNOWN value.

Bits [19:6]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TA, bit [5]

Indicates trapped use of Advanced SIMD functionality. The possible values of this bit are:

`0b0` Exception was not caused by trapped use of Advanced SIMD functionality.

`0b1` Exception was caused by trapped use of Advanced SIMD functionality.

Any use of an Advanced SIMD instruction that is not also a floating-point instruction that is trapped to Hyp mode because of a trap configured in the HCPTR sets this bit to 1.

For a list of these instructions, see *Controls of Advanced SIMD operation that do not apply to floating-point operation* on page E1-3966.

This field resets to an architecturally UNKNOWN value.

Bit [4]

Reserved, RES0.

coproc, bits [3:0]

When the TA field returns the value 1, this field returns the value `1010`, otherwise this field is RES0.

This field resets to an architecturally UNKNOWN value.

The following sections describe the configuration settings for the traps that are reported using EC value `0b000111`:

- *General trapping to Hyp mode of Non-secure accesses to the SIMD and floating-point registers* on page G1-5798.
- *Traps to Hyp mode of Non-secure accesses to Advanced SIMD functionality* on page G1-5799.

ISS encoding for an exception from HVC or SVC instruction execution

24 … 16	15 … 0
RES0	imm16

Bits [24:16]

Reserved, RES0.

imm16, bits [15:0]

The value of the immediate field from the HVC or SVC instruction.

For an HVC instruction, this is the value of the imm16 field of the issued instruction.

For an SVC instruction:

- If the instruction is unconditional, then:
  - For the T32 instruction, this field is zero-extended from the imm8 field of the instruction.
  - For the A32 instruction, this field is the bottom 16 bits of the imm24 field of the instruction.
- For the T32 instruction, this field is zero-extended from the imm8 field of the instruction.For the A32 instruction, this field is the bottom 16 bits of the imm24 field of the instruction.
- If the instruction is conditional, this field is UNKNOWN.

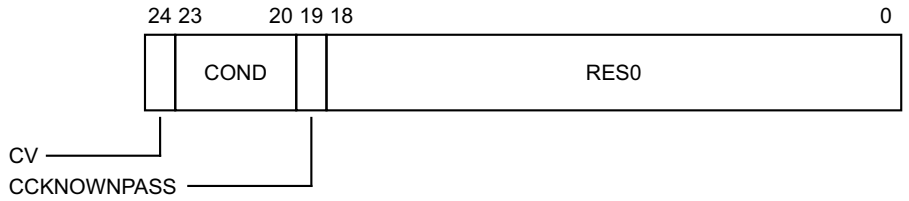
This field resets to an architecturally UNKNOWN value.

The HVC instruction is unconditional, and a conditional SVC instruction generates an exception only if it passes its condition code check. Therefore, the syndrome information for these exceptions does not require conditionality information.

Supervisor Call exception, when the value of HCR.TGE is 1 on page G1-5719 describes the configuration settings for the trap reported with EC value `0b010001`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ISS encoding for an exception from SMC instruction execution



CV, bit [24]

Condition code valid. Possible values of this bit are:

`0b0` The COND field is not valid.

`0b1` The COND field is valid.

When an A32 instruction is trapped, CV is set to 1.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

This field is only valid if CCKNOWNPASS is 1, otherwise it is RES0.

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction.

When an A32 instruction is trapped, CV is set to 1 and:

- If the instruction is conditional, COND is set to the condition code field value from the instruction.
- If the instruction is unconditional, COND is set to `0b1110`.

A conditional A32 instruction that is known to pass its condition code check can be presented either:

- With COND set to `0b1110`, the value for unconditional.
- With the COND value held in the instruction.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:

- CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
- CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

This field is only valid if CCKNOWNPASS is 1, otherwise it is RES0.

This field resets to an architecturally UNKNOWN value.

CCKNOWNPASS, bit [19]

Indicates whether the instruction might have failed its condition code check.

`0b0` The instruction was unconditional, or was conditional and passed its condition code check.

`0b1` The instruction was conditional, and might have failed its condition code check.

This field resets to an architecturally UNKNOWN value.

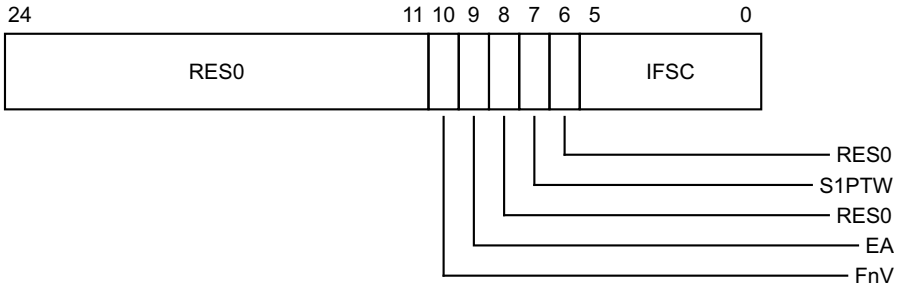
Bits [18:0]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Traps to Hyp mode of Non-secure EL1 execution of SMC instructions on page G1-5794 describes the configuration settings for this trap, for instructions executed in Non-secure EL1.

ISS encoding for an exception from a Prefetch Abort



Bits [24:11]

Reserved, RES0.

FnV, bit [10]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

`0b0` HIFAR is valid.

`0b1` HIFAR is not valid, and holds an UNKNOWN value.

This field is only valid if the IFSC code is `0b010000`. It is RES0 for all other aborts.

This field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

This field resets to an architecturally UNKNOWN value.

Bit [8]

Reserved, RES0.

S1PTW, bit [7]

For a stage 2 fault, indicates whether the fault was a stage 2 fault on an access made for a stage 1 translation table walk:

`0b0` Fault not on a stage 2 translation for a stage 1 translation table walk.

`0b1` Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

This field resets to an architecturally UNKNOWN value.

Bit [6]

Reserved, RES0.

IFSC, bits [5:0]

Instruction Fault Status Code. Possible values of this field are:

`0b000000` Address size fault in translation table base register.

`0b000001` Address size fault, level 1.

`0b000010` Address size fault, level 2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b000011` Address size fault, level 3.

`0b000101` Translation fault, level 1.

`0b000110` Translation fault, level 2.

`0b000111` Translation fault, level 3.

`0b001001` Access flag fault, level 1.

`0b001010` Access flag fault, level 2.

`0b001011` Access flag fault, level 3.

`0b001101` Permission fault, level 1.

`0b001110` Permission fault, level 2.

`0b001111` Permission fault, level 3.

`0b010000` Synchronous External abort, not on translation table walk.

`0b010101` Synchronous External abort, on translation table walk, level 1.

`0b010110` Synchronous External abort, on translation table walk, level 2.

`0b010111` Synchronous External abort, on translation table walk, level 3.

`0b011000` **When RAS is not implemented** Synchronous parity or ECC error on memory access, not on translation table walk.

`0b011101` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk, level 1.

`0b011110` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk, level 2.

`0b011111` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk, level 3.

`0b100010` Debug exception.

`0b110000` TLB conflict abort.

All other values are reserved.

For more information about the lookup level associated with a fault, see *The level associated with MMU faults on a Long-descriptor translation table lookup* on page G5-6032.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.

This field resets to an architecturally UNKNOWN value.

The following sections describe cases where Prefetch Abort exceptions can be routed to Hyp mode, generating exceptions that are reported in the HSR with EC value `0b100000`:

- *Abort exceptions, when the value of HCR.TGE is 1* on page G1-5719.
- *Routing debug exceptions to EL2 using AArch32* on page G1-5720.

ISS encoding for an exception from an Illegal state or PC alignment fault

24 … 0
RES0

Bits [24:0]

Reserved, RES0.

For more information about the Illegal state exception, see:

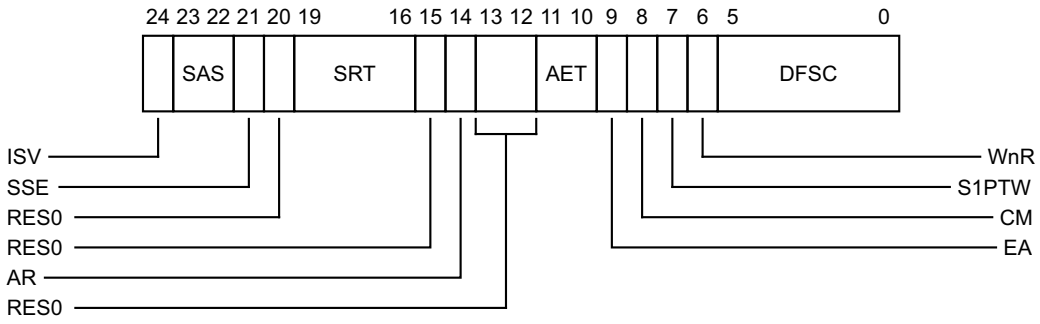
- *Illegal changes to PSTATE.M* on page G1-5699.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *Illegal return events from AArch32 state* on page G1-5726.
- *Legal returns that set PSTATE.IL to 1* on page G1-5728.
- *The Illegal Execution state exception* on page G1-5728.

For more information about the PC alignment fault exception, see *Branching to an unaligned PC* on page K1-7832.

ISS encoding for an exception from a Data Abort



ISV, bit [24]

Instruction syndrome valid. Indicates whether the syndrome information in ISS[23:14] is valid.

`0b0` No valid instruction syndrome. ISS[23:14] are RES0.

`0b1` ISS[23:14] hold a valid instruction syndrome.

This bit is 0 for all faults except Data Aborts generated by stage 2 address translations for which all the following apply to the instruction that generated the Data Abort exception:

- The instruction is an LDR, LDA, LDRT, LDRSH, LDRSHT, LDRH, LDAH, LDRHT, LDRSB, LDRSBT, LDRB, LDAB, LDRBT, STR, STL, STRT, STRH, STLH, STRHT, STRB, STLB, or STRBT instruction.
- The instruction is not performing register writeback.
- The instruction is not using the PC as a source or destination register.

For these cases, ISV is UNKNOWN if the exception was generated in Debug state in memory access mode, as described in *Data Aborts in Memory access mode* on page H4-6996, and otherwise indicates whether ISS[23:14] hold a valid syndrome.

Note

In the A32 instruction set, LDR*T and STR*T instructions always perform register writeback and therefore never return a valid instruction syndrome.

When the RAS Extension is implemented, ISV is 0 for any synchronous External abort.

ISV is set to 0 on a stage 2 abort on a stage 1 translation table walk.

When the RAS Extension is not implemented, it is IMPLEMENTATION DEFINED whether ISV is set to 1 or 0 on a synchronous External abort on a stage 2 translation table walk.

This field resets to an architecturally UNKNOWN value.

SAS, bits [23:22]

Syndrome Access Size. When ISV is 1, indicates the size of the access attempted by the faulting operation.

`0b00` Byte

`0b01` Halfword

`0b10` Word

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b11` Doubleword

This field is UNKNOWN when the value of ISV is UNKNOWN.

This field is RES0 when the value of ISV is 0.

This field resets to an architecturally UNKNOWN value.

SSE, bit [21]

Syndrome Sign Extend. When ISV is 1, for a byte, halfword, or word load operation, indicates whether the data item must be sign extended. For these cases, the possible values of this bit are:

`0b0` Sign-extension not required.

`0b1` Data item must be sign-extended.

For all other operations this bit is 0.

This field is UNKNOWN when the value of ISV is UNKNOWN.

This field is RES0 when the value of ISV is 0.

This field resets to an architecturally UNKNOWN value.

Bit [20]

Reserved, RES0.

SRT, bits [19:16]

Syndrome Register transfer. When ISV is 1, the register number of the Rt operand of the faulting instruction.

This field is UNKNOWN when the value of ISV is UNKNOWN.

This field is RES0 when the value of ISV is 0.

This field resets to an architecturally UNKNOWN value.

Bit [15]

Reserved, RES0.

AR, bit [14]

Acquire/Release. When ISV is 1, the possible values of this bit are:

`0b0` Instruction did not have acquire/release semantics.

`0b1` Instruction did have acquire/release semantics.

This field is UNKNOWN when the value of ISV is UNKNOWN.

This field is RES0 when the value of ISV is 0.

This field resets to an architecturally UNKNOWN value.

Bits [13:12]

Reserved, RES0.

AET, bits [11:10]

Asynchronous Error Type.

When the RAS Extension is implemented and the value returned in the DFSC field is `0b010001`, describes the state of the PE after taking the SError interrupt exception. The possible values of this field are:

`0b00` Uncontainable error (UC) or uncategorized.

`0b01` Unrecoverable error (UEU).

`0b10` Restartable error (UEO) or Corrected error (CE).

`0b11` Recoverable error (UER).

On a synchronous Data Abort, this field is RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If multiple errors are taken as a single SError interrupt exception, the overall state of the PE is reported. For example, if both a Recoverable and Unrecoverable error occurred, the state is Unrecoverable.

Note

Software can use this information to determine what recovery might be possible. The recovery software must also examine any implemented fault records to determine the location and extent of the error.

When the RAS Extension is not implemented, or when DFSC is not `0b010001`:

- Bit[11] is RES0.
- Bit[10] forms the FnV field.

Note

Armv8.2 requires the implementation of the RAS Extension.

This field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

This field resets to an architecturally UNKNOWN value.

CM, bit [8]

Cache maintenance. For a synchronous fault, identifies fault that comes from a cache maintenance or address translation instruction. For synchronous faults, the possible values of this bit are:

`0b0` Fault not generated by a cache maintenance or address translation instruction.

`0b1` Fault generated by a cache maintenance or address translation instruction.

For an asynchronous Data Abort exception, this bit is 0.

This field resets to an architecturally UNKNOWN value.

S1PTW, bit [7]

For a stage 2 fault, indicates whether the fault was a stage 2 fault on an access made for a stage 1 translation table walk:

`0b0` Fault not on a stage 2 translation for a stage 1 translation table walk.

`0b1` Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

This field resets to an architecturally UNKNOWN value.

WnR, bit [6]

Write not Read. Indicates whether a synchronous abort was caused by a write instruction or a read instruction. The possible values of this bit are:

`0b0` Abort caused by a read instruction.

`0b1` Abort caused by a write instruction.

For faults on cache maintenance and address translation instructions, this bit always returns a value of 1.

On an asynchronous Data Abort:

- When the RAS Extension is not implemented, this bit is UNKNOWN.
- When the RAS Extension is implemented, this bit is RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

Armv8.2 requires the implementation of the RAS Extension.

This field resets to an architecturally UNKNOWN value.

DFSC, bits [5:0]

Data Fault Status Code. Possible values of this field are:

`0b000000` Address size fault in translation table base register.

`0b000001` Address size fault, level 1.

`0b000010` Address size fault, level 2.

`0b000011` Address size fault, level 3.

`0b000101` Translation fault, level 1.

`0b000110` Translation fault, level 2.

`0b000111` Translation fault, level 3.

`0b001001` Access flag fault, level 1.

`0b001010` Access flag fault, level 2.

`0b001011` Access flag fault, level 3.

`0b001101` Permission fault, level 1.

`0b001110` Permission fault, level 2.

`0b001111` Permission fault, level 3.

`0b010000` Synchronous External abort, not on translation table walk.

`0b010001` Asynchronous SError interrupt.

`0b010101` Synchronous External abort, on translation table walk, level 1.

`0b010110` Synchronous External abort, on translation table walk, level 2.

`0b010111` Synchronous External abort, on translation table walk, level 3.

`0b011000` **When RAS is not implemented** Synchronous parity or ECC error on memory access, not on translation table walk.

`0b011001` **When RAS is not implemented** Asynchronous SError interrupt, from a parity or ECC error on memory access.

`0b011101` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk, level 1.

`0b011110` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk, level 2.

`0b011111` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk, level 3.

`0b100001` Alignment fault.

`0b100010` Debug exception.

`0b110000` TLB conflict abort.

`0b110100` IMPLEMENTATION DEFINED fault (Lockdown).

`0b110101` IMPLEMENTATION DEFINED fault (Unsupported Exclusive access).

All other values are reserved.

For more information about the lookup level associated with a fault, see *The level associated with MMU faults on a Long-descriptor translation table lookup* on page G5-6032.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following describe cases where Data Abort exceptions can be routed to Hyp mode, generating exceptions that are reported in the HSR with EC value `0b100100`:

- *Abort exceptions, when the value of HCR.TGE is 1* on page G1-5719.
- *Routing debug exceptions to EL2 using AArch32* on page G1-5720.

The following describe cases that can cause a Data Abort exception that is taken to Hyp mode, and reported in the HSR with EC value of `0b100000` or `0b100100`:

- *Hyp mode control of Non-secure access permissions* on page G5-5977.
- *Memory fault reporting in Hyp mode* on page G5-6036.

Accessing the HSR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0101	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return HSR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        return HSR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0101	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    HSR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        else
            HSR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.74 HSTR, Hyp System Trap Register

The HSTR characteristics are:

Purpose

Controls trapping to Hyp mode of Non-secure accesses, at EL1 or lower, to System registers in the coproc == `0b1111` encoding space:

- By the CRn value used to access the register using MCR or MRC instruction.
- By the CRm value used to access the register using MCRR or MRRC instruction.

Configurations

AArch32 System register HSTR[31:0] is architecturally mapped to AArch64 System register HSTR_EL2[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to HSTR are UNKNOWN.

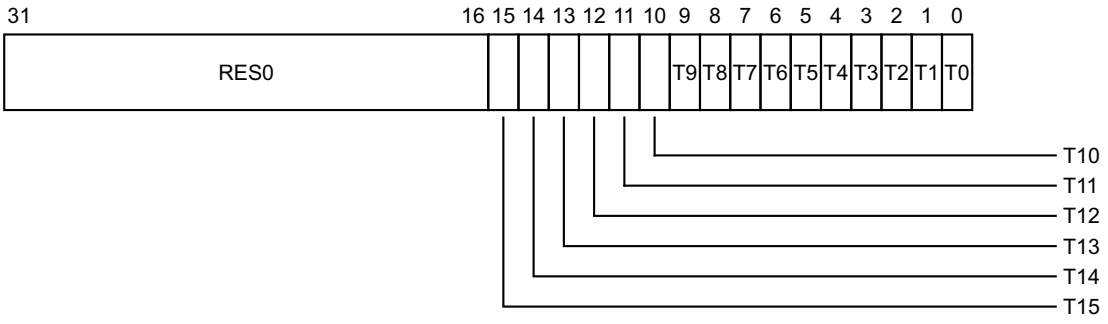
If EL2 is not implemented, this register is RES0 from EL3.

Attributes

HSTR is a 32-bit register.

Field descriptions

The HSTR bit assignments are:



Bits [31:16]

Reserved, RES0.

T<n>, bit [n], for n = 0 to 15

Fields T14 and T4 are RES0.

The remaining fields control whether Non-secure EL0 and EL1 accesses, using MCR, MRC, MCRR, and MRRC instructions, to the System registers in the coproc == `0b1111` encoding space are trapped to Hyp mode:

`0b0` This control has no effect on Non-secure EL0 or EL1 accesses to System registers.

`0b1` Any Non-secure EL1 MCR or MRC access with coproc == `0b1111` and CRn == <n> is trapped to Hyp mode. A Non-secure EL0 MCR or MRC access with these values is trapped to Hyp mode only if the access is not UNDEFINED when the value of this field is 0.

Any Non-secure EL1 MCRR or MRRC access with coproc == `0b1111` and CRm == <n> is trapped to Hyp mode. A Non-secure EL0 MCRR or MRRC access with these values is trapped to Hyp mode only if the access is not UNDEFINED when the value of this field is 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For example, when HSTR.T7 is 1, for instructions executed at Non-secure EL1:

- An MCR or MRC instruction with coproc set to `0b1111` and `<CRn>` set to c7 is trapped to Hyp mode.
- An MCRR or MRRC instruction with coproc set to `0b1111` and `<CRm>` set to c7 is trapped to Hyp mode.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

Accessing the HSTR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return HSTR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        return HSTR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    HSTR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        HSTR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.75 HTCR, Hyp Translation Control Register

The HTCR characteristics are:

Purpose

The control register for stage 1 of the EL2 translation regime.

Note

This stage of translation always uses the Long-descriptor translation table format.

Configurations

AArch32 System register HTCR[31:0] is architecturally mapped to AArch64 System register TCR_EL2[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to HTCR are UNKNOWN.

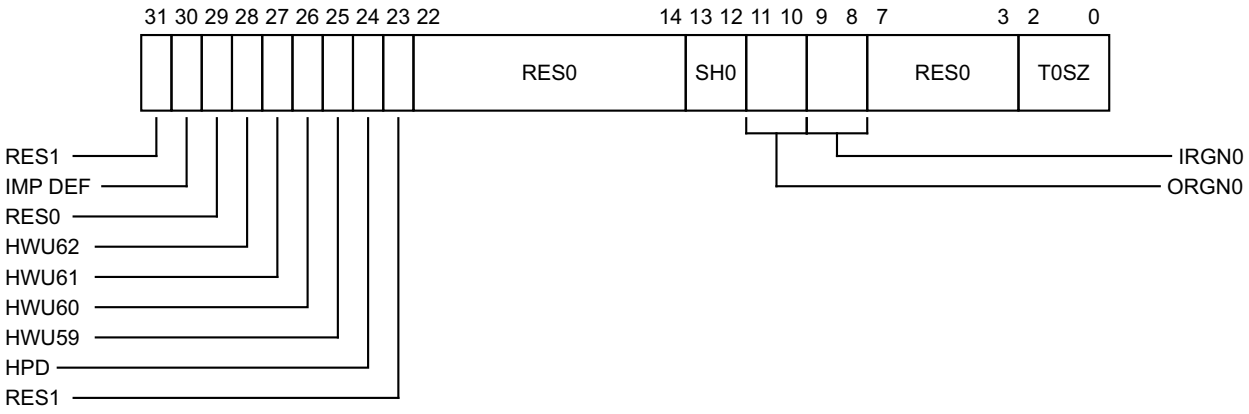
If EL2 is not implemented, this register is RES0 from EL3.

Attributes

HTCR is a 32-bit register.

Field descriptions

The HTCR bit assignments are:



Bit [31]

Reserved, RES1.

IMPLEMENTATION DEFINED, bit [30]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

Bit [29]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

HWU62, bit [28]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 1 translation table Block or Page entry.

`0b0` Bit[62] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` Bit[62] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of HTCR.HPD is 1.

The Effective value of this field is 0 if the value of HTCR.HPD is 0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU61, bit [27]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 1 translation table Block or Page entry.

`0b0` Bit[61] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` Bit[61] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of HTCR.HPD is 1.

The Effective value of this field is 0 if the value of HTCR.HPD is 0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU60, bit [26]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 1 translation table Block or Page entry.

`0b0` Bit[60] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` Bit[60] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of HTCR.HPD is 1.

The Effective value of this field is 0 if the value of HTCR.HPD is 0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU59, bit [25]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 1 translation table Block or Page entry.

`0b0` Bit[59] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` Bit[59] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of HTCR.HPD is 1.

The Effective value of this field is 0 if the value of HTCR.HPD is 0.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

Reserved, RES0.

HPD, bit [24]

When ARMv8.2-AA32HPD is implemented:

Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, XNTable, and PXNTable, in the PL2 translation regime.

`0b0` Hierarchical permissions are enabled.

`0b1` Hierarchical permissions are disabled.

When disabled, the permissions are treated as if the bits are zero.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [23]

Reserved, RES1.

Bits [22:14]

Reserved, RES0.

SH0, bits [13:12]

Shareability attribute for memory associated with translation table walks using HTTBR.

`0b00` Non-shareable.

`0b10` Outer Shareable.

`0b11` Inner Shareable.

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-7853.

This field resets to an architecturally UNKNOWN value.

ORGN0, bits [11:10]

Outer cacheability attribute for memory associated with translation table walks using HTTBR.

`0b00` Normal memory, Outer Non-cacheable.

`0b01` Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.

`0b10` Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.

`0b11` Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.

This field resets to an architecturally UNKNOWN value.

IRGN0, bits [9:8]

Inner cacheability attribute for memory associated with translation table walks using HTTBR.

`0b00` Normal memory, Inner Non-cacheable.

`0b01` Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.

`0b10` Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.

`0b11` Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

This field resets to an architecturally UNKNOWN value.

Bits [7:3]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T0SZ, bits [2:0]

The size offset of the memory region addressed by HTTBR. The region size is $2^{(32-T0SZ)}$ bytes.

This field resets to an architecturally UNKNOWN value.

Accessing the HTCR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0010	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T2 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return HTCR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        return HTCR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0010	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T2 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    HTCR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        HTCR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.76 HTPIDR, Hyp Software Thread ID Register

The HTPIDR characteristics are:

Purpose

Provides a location where software running in Hyp mode can store thread identifying information that is not visible to Non-secure software executing at EL0 or EL1, for hypervisor management purposes.

The PE makes no use of this register.

Configurations

AArch32 System register HTPIDR[31:0] is architecturally mapped to AArch64 System register TPIDR_EL2[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to HTPIDR are UNKNOWN.

If EL2 is not implemented, this register is RES0 from EL3.

Note

The PE never updates this register.

Attributes

HTPIDR is a 32-bit register.

Field descriptions

The HTPIDR bit assignments are:

31 … 0
Thread ID

Bits [31:0]

Thread ID. Thread identifying information stored by software running at this Exception level.

This field resets to an architecturally UNKNOWN value.

Accessing the HTPIDR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1101	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL2 then
    return HTPIDR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        return HTPIDR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1101	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    HTPIDR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        HTPIDR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.77 HTTBR, Hyp Translation Table Base Register

The HTTBR characteristics are:

Purpose

Holds the base address of the translation table for the initial lookup for stage 1 of an address translation in the EL2 translation regime, and other information for this translation regime.

Configurations

AArch32 System register HTTBR[47:1] is architecturally mapped to AArch64 System register TTBR0_EL2[47:1].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to HTTBR are UNKNOWN.

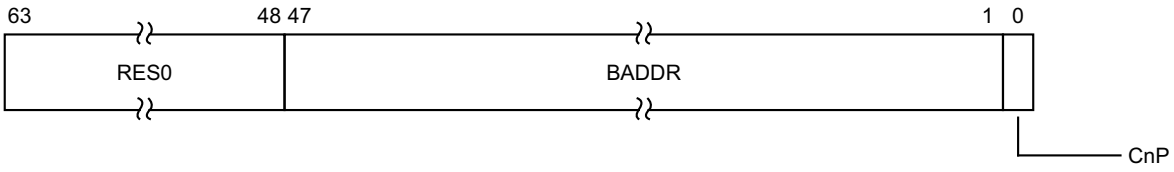
If EL2 is not implemented, this register is RES0 from EL3.

Attributes

HTTBR is a 64-bit register.

Field descriptions

The HTTBR bit assignments are:



Bits [63:48]

Reserved, RES0.

BADDR, bits [47:1]

Translation table base address, bits[47:x], Bits [x-1:1] are RES0, with the additional requirement that if bits[x-1:3] are not all zero, this is a misaligned translation table base address, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Register bits [x-1:3] are treated as if all the bits are zero. The value read back from these bits is either the value written or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

x is determined from the value of HTCR.T0SZ as follows:

- If HTCR.T0SZ is 0 or 1, x = 5 - HTCR.T0SZ.
- If HTCR.T0SZ is greater than 1, x = 14 - HTCR.T0SZ.

If bits[47:40] of the translation table base address are not zero, an Address size fault is generated.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CnP, bit [0]

When ARMv8.2-TTCNP is implemented:

Common not Private. This bit indicates whether each entry that is pointed to by HTTBR is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of HTTBR.CnP is 1.

`0b0` The translation table entries pointed to by HTTBR are permitted to differ from corresponding entries for HTTBR for other PEs in the Inner Shareable domain. This is not affected by the value of HTTBR.CnP on those other PEs.

`0b1` The translation table entries pointed to by HTTBR are the same as the translation table entries pointed to by HTTBR on every other PE in the Inner Shareable domain for which the value of HTTBR.CnP is 1.

Note

If the value of the HTTBR.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those HTTBRs do not point to the same translation table entries when the other conditions specified for the case when the value of CnP is 1 apply, then the results of translations are CONSTRAINED UNPREDICTABLE, see *CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values* on page K1-7835.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing the HTTBR

Accesses to this register use the following encodings in the System instruction encoding space:

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b0010	0b0100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T2 == '1' then
        AArch32.TakeHypTrapException(0x04);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return HTTBR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        return HTTBR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b0010	0b0100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T2 == '1' then
        AArch32.TakeHypTrapException(0x04);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    HTTBR = R[t2]:R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        HTTBR = R[t2]:R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.78 HVBAR, Hyp Vector Base Address Register

The HVBAR characteristics are:

Purpose

Holds the vector base address for any exception that is taken to Hyp mode.

Configurations

AArch32 System register HVBAR[31:0] is architecturally mapped to AArch64 System register VBAR_EL2[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to HVBAR are UNKNOWN.

If EL2 is not implemented, this register is RES0 from EL3.

Attributes

HVBAR is a 32-bit register.

Field descriptions

The HVBAR bit assignments are:

31 … 5	4 … 0
Vector Base Address	RES0

Bits [31:5]

Vector Base Address. Bits[31:5] of the base address of the exception vectors for exceptions taken to this Exception level. Bits[4:0] of an exception vector are the exception offset.

This field resets to an architecturally UNKNOWN value.

Bits [4:0]

Reserved, RES0.

Accessing the HVBAR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1100	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T12 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T12 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return HVBAR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        UNDEFINED;
    else
        return HVBAR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1100	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T12 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T12 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    HVBAR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        HVBAR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.79 ICIALLU, Instruction Cache Invalidate All to PoU

The ICIALLU characteristics are:

Purpose

Invalidate all instruction caches to PoU. If branch predictors are architecturally visible, also flush branch predictors.

Configurations

AArch32 System instruction ICIALLU performs the same function as AArch64 System instruction IC IALLU.

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ICIALLU are UNKNOWN.

Attributes

ICIALLU is a 32-bit System instruction.

Field descriptions

ICIALLU ignores the value in the register specified by the instruction. Software does not have to write a value to the register before issuing this instruction.

Executing the ICIALLU instruction

The PE ignores the value of <Rt>. Software does not have to write a value to this register before issuing this instruction.

When HCR.FB is 1, at Non-secure EL1 this instruction executes as a ICIALLUIS.

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0101	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TPU == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TOCU == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TPU == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TOCU == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.FB == '1' then
        ICIALLUIS();
    else
        ICIALLU();
elsif PSTATE.EL == EL2 then
    ICIALLU();
elsif PSTATE.EL == EL3 then
    ICIALLU();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.80 ICIALLUIS, Instruction Cache Invalidate All to PoU, Inner Shareable

The ICIALLUIS characteristics are:

Purpose

Invalidate all instruction caches Inner Shareable to PoU. If branch predictors are architecturally visible, also flush branch predictors.

Configurations

AArch32 System instruction ICIALLUIS performs the same function as AArch64 System instruction IC IALLUIS.

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ICIALLUIS are UNKNOWN.

Attributes

ICIALLUIS is a 32-bit System instruction.

Field descriptions

ICIALLUIS ignores the value in the register specified by the instruction. Software does not have to write a value to the register before issuing this instruction.

Executing the ICIALLUIS instruction

The PE ignores the value of <Rt>. Software does not have to write a value to this register before issuing this instruction.

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TPU == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TICAB == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TOCU == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TPU == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TICAB == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TOCU == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        ICIALLU();
elsif PSTATE.EL == EL2 then
    ICIALLU();
elsif PSTATE.EL == EL3 then
    ICIALLU();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.81 ICIMVAU, Instruction Cache line Invalidate by VA to PoU

The ICIMVAU characteristics are:

Purpose

Invalidate instruction cache line by virtual address to PoU.

Configurations

AArch32 System instruction ICIMVAU performs the same function as AArch64 System instruction IC IVAU.

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ICIMVAU are UNKNOWN.

Attributes

ICIMVAU is a 32-bit System instruction.

Field descriptions

The ICIMVAU input value bit assignments are:

31 … 0
Virtual address to use

Bits [31:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing the ICIMVAU instruction

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see *AArch32 instruction cache maintenance instructions (IC*)* on page G4-5900.

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0101	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TPU == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TOCU == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TPU == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TOCU == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        ICIMVAU(R[t]);
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    ICIMVAU(R[t]);
elsif PSTATE.EL == EL3 then
    ICIMVAU(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.82 ID_AFR0, Auxiliary Feature Register 0

The ID_AFR0 characteristics are:

Purpose

Provides information about the IMPLEMENTATION DEFINED features of the PE in AArch32 state.

Must be interpreted with the Main ID Register, MIDR.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page G8-6106.

Configurations

AArch32 System register ID_AFR0[31:0] is architecturally mapped to AArch64 System register ID_AFR0_EL1[31:0].

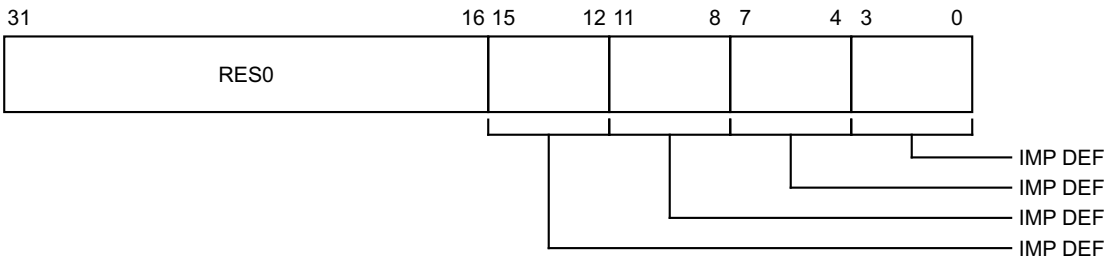
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_AFR0 are UNKNOWN.

Attributes

ID_AFR0 is a 32-bit register.

Field descriptions

The ID_AFR0 bit assignments are:



Bits [31:16]

Reserved, RES0.

IMPLEMENTATION DEFINED, bits [15:12]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED, bits [11:8]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED, bits [7:4]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED, bits [3:0]

IMPLEMENTATION DEFINED.

Accessing the ID_AFR0

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0001	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        return ID_AFR0;
elsif PSTATE.EL == EL2 then
    return ID_AFR0;
elsif PSTATE.EL == EL3 then
    return ID_AFR0;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.83 ID_DFR0, Debug Feature Register 0

The ID_DFR0 characteristics are:

Purpose

Provides top level information about the debug system in AArch32 state.

Must be interpreted with the Main ID Register, MIDR.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page G8-6106.

Configurations

AArch32 System register ID_DFR0[31:0] is architecturally mapped to AArch64 System register ID_DFR0_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_DFR0 are UNKNOWN.

Attributes

ID_DFR0 is a 32-bit register.

Field descriptions

The ID_DFR0 bit assignments are:

31 28	27 24	23 20	19 12	11 8	7 4	3 0
TraceFilt	PerfMon	MProfDbg	RES0	MMapDbg	CopSDbg	CopDbg

TraceFilt, bits [31:28]

Armv8.4 Self-hosted Trace Extension version. Defined values are:

`0b0000` Armv8.4 Self-hosted Trace Extension not implemented.

`0b0001` Armv8.4 Self-hosted Trace Extension implemented.

All other values are reserved.

ARMv8.4-Trace implements the functionality added by the value `0b0001`.

From Armv8.3, the permitted values are `0b0000` and `0b0001`.

PerfMon, bits [27:24]

Performance Monitors Extension version.

This field does not follow the standard ID scheme, but uses the alternative ID scheme described in *Alternative ID scheme used for the Performance Monitors Extension version* on page G8-6108.

Defined values are:

`0b0000` Performance Monitors Extension not implemented.

`0b0001` Performance Monitors Extension version 1 implemented, PMUv1.

`0b0010` Performance Monitors Extension version 2 implemented, PMUv2.

`0b0011` Performance Monitors Extension version 3 implemented, PMUv3.

`0b0100` PMUv3 for Armv8.1. As `0b0011`, and also includes support for:

- Extended 16-bit PMEVTYPER<n>.evtCount field.
- If EL2 is implemented, the HDCR.HPMD control bit.

`0b0101` PMUv3 for Armv8.4. As `0b0100` and also includes support for the PMMIR register.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b0110` PMUv3 for Armv8.5. As `0b0101` and also includes support for:

- 64-bit event counters.
- If EL2 is implemented, the HDCR.HCCD control bit.
- If EL3 is implemented, the SDCR.SCCD control bit.

`0b1111` IMPLEMENTATION DEFINED form of performance monitors supported, PMUv3 not supported. Arm does not recommend this value in new implementations.

ARMv8.1-PMU implements the functionality added by the value `0b0100`.

ARMv8.4-PMU implements the functionality added by the value `0b0101`.

ARMv8.5-PMU implements the functionality added by the value `0b0110`.

All other values are reserved.

In any Armv8 implementation, the values `0b0001` and `0b0010` are not permitted.

From Armv8.1, the value `0b0011` is not permitted.

From Armv8.4, the value `0b0100` is not permitted.

From Armv8.5, the value `0b0101` is not permitted.

Note

In Armv7, the value `0b0000` can mean that PMUv1 is implemented. PMUv1 is not permitted in an Armv8 implementation.

MProfDbg, bits [23:20]

M Profile Debug. Support for memory-mapped debug model for M profile processors. Defined values are:

`0b0000` Not supported.

`0b0001` Support for M profile Debug architecture, with memory-mapped access.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0000`.

Bits [19:12]

Reserved, RES0.

MMapDbg, bits [11:8]

Memory Mapped Debug. Support for v7 memory-mapped debug model, for A and R profile processors.

In Armv8-A, this field is RES0.

The optional memory map defined by Armv8 is not compatible with Armv7.

CopSDbg, bits [7:4]

Support for a System registers-based Secure debug model, using registers in the coproc = `0b1110` encoding space, for an A profile processor that includes EL3.

If EL3 is not implemented and the implemented Security state is Non-secure state, this field is RES0. Otherwise, this field reads the same as bits [3:0].

CopDbg, bits [3:0]

Support for System registers-based debug model, using registers in the coproc == `0b1110` encoding space, for A and R profile processors. Defined values are:

`0b0000` Not supported.

`0b0010` Support for Armv6, v6 Debug architecture, with System registers access.

`0b0011` Support for Armv6, v6.1 Debug architecture, with System registers access.

`0b0100` Support for Armv7, v7 Debug architecture, with System registers access.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b0101` Support for Armv7, v7.1 Debug architecture, with System registers access.

`0b0110` Support for Armv8 debug architecture, with System registers access.

`0b0111` Support for Armv8 debug architecture, with System registers access, and Virtualization Host extensions.

`0b1000` Support for Armv8.2 debug architecture.

`0b1001` Support for Armv8.4 debug architecture.

All other values are reserved.

In any Armv8 implementation, the values `0b0000`, `0b0010`, `0b0011`, `0b0100`, and `0b0101` are not permitted.

If ARMv8.1-VHE is not implemented, the only permitted value is `0b0110`.

In an Armv8.0 implementation, the value `0b1000` is not permitted.

Accessing the ID_DFR0

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0001	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        return ID_DFR0;
elsif PSTATE.EL == EL2 then
    return ID_DFR0;
elsif PSTATE.EL == EL3 then
    return ID_DFR0;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.84 ID_DFR1, Debug Feature Register 1

The ID_DFR1 characteristics are:

Purpose

Provides top level information about the debug system in AArch32.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page G8-6106.

Configurations

AArch32 System register ID_DFR1[31:0] is architecturally mapped to AArch64 System register ID_DFR1_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_DFR1 are RES0.

Note

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.

Attributes

ID_DFR1 is a 32-bit register.

Field descriptions

The ID_DFR1 bit assignments are:

31 … 4	3 … 0
RES0	MTPMU

Bits [31:4]

Reserved, RES0.

MTPMU, bits [3:0]

Multi-threaded PMU extension. Defined values are:

`0b0000` ARMv8.6-MTPMU not implemented. If PMUv3 is implemented, it is IMPLEMENTATION DEFINED whether PMEVTYPER<n>.MT are read/write or RES0.

`0b0001` ARMv8.6-MTPMU implemented and PMEVTYPER<n>.MT are read/write. When ARMv8.6-MTPMU is disabled, the Effective values of PMEVTYPER<n>.MT are 0.

`0b1111` ARMv8.6-MTPMU not implemented. If PMUv3 is implemented, PMEVTYPER<n>.MT are RES0.

All other values are reserved.

ARMv8.6-MTPMU implements the functionality identified by the value 0b0001.

In an Armv8.6-compliant implementation that includes PMUv3, the value `0b0000` is not permitted.

In an implementation that does not include PMUv3, the value `0b0001` is not permitted.

Accessing the ID_DFR1

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0011	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!IsZero(ID_DFR1) || boolean IMPLEMENTATION_DEFINED
"ID_DFR1 trapped by HCR_EL2.TID3") && HCR_EL2.TID3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && (!IsZero(ID_DFR1) || boolean IMPLEMENTATION_DEFINED
"ID_DFR1 trapped by HCR.TID3") && HCR.TID3 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        return ID_DFR1;
elsif PSTATE.EL == EL2 then
    return ID_DFR1;
elsif PSTATE.EL == EL3 then
    return ID_DFR1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.85 ID_ISAR0, Instruction Set Attribute Register 0

The ID_ISAR0 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR1, ID_ISAR2, ID_ISAR3, ID_ISAR4, and ID_ISAR5.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page G8-6106.

Configurations

AArch32 System register ID_ISAR0[31:0] is architecturally mapped to AArch64 System register ID_ISAR0_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_ISAR0 are UNKNOWN.

Attributes

ID_ISAR0 is a 32-bit register.

Field descriptions

The ID_ISAR0 bit assignments are:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
RES0	Divide	Debug	Coproc	CmpBranch	BitField	BitCount	Swap

Bits [31:28]

Reserved, RES0.

Divide, bits [27:24]

Indicates the implemented Divide instructions. Defined values are:

`0b0000` None implemented.

`0b0001` Adds SDIV and UDIV in the T32 instruction set.

`0b0010` As for `0b0001`, and adds SDIV and UDIV in the A32 instruction set.

All other values are reserved.

In Armv8-A the only permitted value is `0b0010`.

Debug, bits [23:20]

Indicates the implemented Debug instructions. Defined values are:

`0b0000` None implemented.

`0b0001` Adds BKPT.

All other values are reserved.

In Armv8-A the only permitted value is `0b0001`.

Coproc, bits [19:16]

Indicates the implemented System register access instructions. Defined values are:

`0b0000` None implemented, except for instructions separately attributed by the architecture to provide access to AArch32 System registers and System instructions.

`0b0001` Adds generic CDP, LDC, MCR, MRC, and STC.

`0b0010` As for `0b0001`, and adds generic CDP2, LDC2, MCR2, MRC2, and STC2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b0011` As for `0b0010`, and adds generic MCRR and MRRC.

`0b0100` As for `0b0011`, and adds generic MCRR2 and MRRC2.

All other values are reserved.

In Armv8-A the only permitted value is `0b0000`.

CmpBranch, bits [15:12]

Indicates the implemented combined Compare and Branch instructions in the T32 instruction set. Defined values are:

`0b0000` None implemented.

`0b0001` Adds CBNZ and CBZ.

All other values are reserved.

In Armv8-A the only permitted value is `0b0001`.

BitField, bits [11:8]

Indicates the implemented BitField instructions. Defined values are:

`0b0000` None implemented.

`0b0001` Adds BFC, BFI, SBFX, and UBFX.

All other values are reserved.

In Armv8-A the only permitted value is `0b0001`.

BitCount, bits [7:4]

Indicates the implemented Bit Counting instructions. Defined values are:

`0b0000` None implemented.

`0b0001` Adds CLZ.

All other values are reserved.

In Armv8-A the only permitted value is `0b0001`.

Swap, bits [3:0]

Indicates the implemented Swap instructions in the A32 instruction set. Defined values are:

`0b0000` None implemented.

`0b0001` Adds SWP and SWPB.

All other values are reserved.

In Armv8-A the only permitted value is `0b0000`.

Accessing the ID_ISAR0

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        return ID_ISAR0;
elsif PSTATE.EL == EL2 then
    return ID_ISAR0;
elsif PSTATE.EL == EL3 then
    return ID_ISAR0;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.86 ID_ISAR1, Instruction Set Attribute Register 1

The ID_ISAR1 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0, ID_ISAR2, ID_ISAR3, ID_ISAR4, and ID_ISAR5.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page G8-6106.

Configurations

AArch32 System register ID_ISAR1[31:0] is architecturally mapped to AArch64 System register ID_ISAR1_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_ISAR1 are UNKNOWN.

Attributes

ID_ISAR1 is a 32-bit register.

Field descriptions

The ID_ISAR1 bit assignments are:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
Jazelle	Interwork	Immediate	IfThen	Extend	Except_AR	Except	Endian

Jazelle, bits [31:28]

Indicates the implemented Jazelle extension instructions. Defined values are:

`0b0000` No support for Jazelle.

`0b0001` Adds the BXJ instruction, and the J bit in the PSR. This setting might indicate a trivial implementation of the Jazelle extension.

All other values are reserved.

In Armv8-A the only permitted value is `0b0001`.

Interwork, bits [27:24]

Indicates the implemented Interworking instructions. Defined values are:

`0b0000` None implemented.

`0b0001` Adds the BX instruction, and the T bit in the PSR.

`0b0010` As for `0b0001`, and adds the BLX instruction. PC loads have BX-like behavior.

`0b0011` As for `0b0010`, and guarantees that data-processing instructions in the A32 instruction set with the PC as the destination and the S bit clear have BX-like behavior.

All other values are reserved.

In Armv8-A the only permitted value is `0b0011`.

Immediate, bits [23:20]

Indicates the implemented data-processing instructions with long immediates. Defined values are:

`0b0000` None implemented.

`0b0001` Adds:

- The MOVT instruction

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The MOV instruction encodings with zero-extended 16-bit immediates.
- The T32 ADD and SUB instruction encodings with zero-extended 12-bit immediates, and the other ADD, ADR, and SUB encodings cross-referenced by the pseudocode for those encodings.

All other values are reserved.

In Armv8-A the only permitted value is `0b0001`.

IfThen, bits [19:16]

Indicates the implemented If-Then instructions in the T32 instruction set. Defined values are:

`0b0000` None implemented.

`0b0001` Adds the IT instructions, and the IT bits in the PSRs.

All other values are reserved.

In Armv8-A the only permitted value is `0b0001`.

Extend, bits [15:12]

Indicates the implemented Extend instructions. Defined values are:

`0b0000` No scalar sign-extend or zero-extend instructions are implemented, where scalar instructions means non-Advanced SIMD instructions.

`0b0001` Adds the SXTB, SXTH, UXTB, and UXTH instructions.

`0b0010` As for `0b0001`, and adds the SXTB16, SXTAB, SXTAB16, SXTAH, UXTB16, UXTAB, UXTAB16, and UXTAH instructions.

All other values are reserved.

In Armv8-A the only permitted value is `0b0010`.

Except_AR, bits [11:8]

Indicates the implemented A and R profile exception-handling instructions. Defined values are:

`0b0000` None implemented.

`0b0001` Adds the SRS and RFE instructions, and the A and R profile forms of the CPS instruction.

All other values are reserved.

In Armv8-A the only permitted value is `0b0001`.

Except, bits [7:4]

Indicates the implemented exception-handling instructions in the A32 instruction set. Defined values are:

`0b0000` Not implemented. This indicates that the User bank and Exception return forms of the LDM and STM instructions are not implemented.

`0b0001` Adds the LDM (exception return), LDM (user registers), and STM (user registers) instruction versions.

All other values are reserved.

In Armv8-A the only permitted value is `0b0001`.

Endian, bits [3:0]

Indicates the implemented Endian instructions. Defined values are:

`0b0000` None implemented.

`0b0001` Adds the SETEND instruction, and the E bit in the PSRs.

All other values are reserved.

In Armv8-A the permitted values are `0b0000` and `0b0001`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the ID_ISAR1

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        return ID_ISAR1;
elsif PSTATE.EL == EL2 then
    return ID_ISAR1;
elsif PSTATE.EL == EL3 then
    return ID_ISAR1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.87 ID_ISAR2, Instruction Set Attribute Register 2

The ID_ISAR2 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0, ID_ISAR1, ID_ISAR3, ID_ISAR4, and ID_ISAR5.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page G8-6106.

Configurations

AArch32 System register ID_ISAR2[31:0] is architecturally mapped to AArch64 System register ID_ISAR2_EL1[31:0].

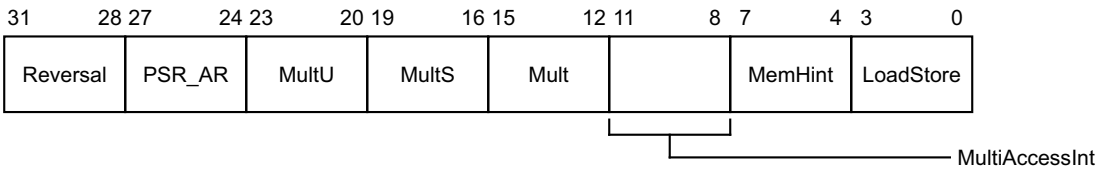
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_ISAR2 are UNKNOWN.

Attributes

ID_ISAR2 is a 32-bit register.

Field descriptions

The ID_ISAR2 bit assignments are:



Reversal, bits [31:28]

Indicates the implemented Reversal instructions. Defined values are:

`0b0000` None implemented.

`0b0001` Adds the REV, REV16, and REVSH instructions.

`0b0010` As for `0b0001`, and adds the RBIT instruction.

All other values are reserved.

In Armv8-A the only permitted value is `0b0010`.

PSR_AR, bits [27:24]

Indicates the implemented A and R profile instructions to manipulate the PSR. Defined values are:

`0b0000` None implemented.

`0b0001` Adds the MRS and MSR instructions, and the exception return forms of data-processing instructions.

All other values are reserved.

In Armv8-A the only permitted value is `0b0001`.

The exception return forms of the data-processing instructions are:

- In the A32 instruction set, data-processing instructions with the PC as the destination and the S bit set. These instructions might be affected by the WithShifts attribute.
- In the T32 instruction set, the SUBS PC,LR,#N instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MultU, bits [23:20]

Indicates the implemented advanced unsigned Multiply instructions. Defined values are:

`0b0000` None implemented.

`0b0001` Adds the UMULL and UMLAL instructions.

`0b0010` As for `0b0001`, and adds the UMAAL instruction.

All other values are reserved.

In Armv8-A the only permitted value is `0b0010`.

MultS, bits [19:16]

Indicates the implemented advanced signed Multiply instructions. Defined values are:

`0b0000` None implemented.

`0b0001` Adds the SMULL and SMLAL instructions.

`0b0010` As for `0b0001`, and adds the SMLABB, SMLABT, SMLALBB, SMLALBT, SMLALTB, SMLALTT, SMLATB, SMLATT, SMLAWB, SMLAWT, SMULBB, SMULBT, SMULTB, SMULTT, SMULWB, and SMULWT instructions. Also adds the Q bit in the PSRs.

`0b0011` As for `0b0010`, and adds the SMLAD, SMLADX, SMLALD, SMLALDX, SMLSD, SMLSDX, SMLSLD, SMLSLDX, SMMLA, SMMLAR, SMMLS, SMMLSR, SMMUL, SMMULR, SMUAD, SMUADX, SMUSD, and SMUSDX instructions.

All other values are reserved.

In Armv8-A the only permitted value is `0b0011`.

Mult, bits [15:12]

Indicates the implemented additional Multiply instructions. Defined values are:

`0b0000` No additional instructions implemented. This means only MUL is implemented.

`0b0001` Adds the MLA instruction.

`0b0010` As for `0b0001`, and adds the MLS instruction.

All other values are reserved.

In Armv8-A the only permitted value is `0b0010`.

MultiAccessInt, bits [11:8]

Indicates the support for interruptible multi-access instructions. Defined values are:

`0b0000` No support. This means the LDM and STM instructions are not interruptible.

`0b0001` LDM and STM instructions are restartable.

`0b0010` LDM and STM instructions are continuable.

All other values are reserved.

In Armv8-A the only permitted value is `0b0000`.

MemHint, bits [7:4]

Indicates the implemented Memory Hint instructions. Defined values are:

`0b0000` None implemented.

`0b0001` Adds the PLD instruction.

`0b0010` Adds the PLD instruction. (`0b0001` and `0b0010` have identical effects.)

`0b0011` As for `0b0001` (or `0b0010`), and adds the PLI instruction.

`0b0100` As for `0b0011`, and adds the PLDW instruction.

All other values are reserved.

In Armv8-A the only permitted value is `0b0100`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

LoadStore, bits [3:0]

Indicates the implemented additional load/store instructions. Defined values are:

`0b0000` No additional load/store instructions implemented.

`0b0001` Adds the LDRD and STRD instructions.

`0b0010` As for `0b0001`, and adds the Load Acquire (LDAB, LDAH, LDA, LDAEXB, LDAEXH, LDAEX, LDAEXD) and Store Release (STLB, STLH, STL, STLEXB, STLEXH, STLEX, STLEXD) instructions.

All other values are reserved.

In Armv8-A the only permitted value is `0b0010`.

Accessing the ID_ISAR2

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0010	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        return ID_ISAR2;
elsif PSTATE.EL == EL2 then
    return ID_ISAR2;
elsif PSTATE.EL == EL3 then
    return ID_ISAR2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.88 ID_ISAR3, Instruction Set Attribute Register 3

The ID_ISAR3 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR4, and ID_ISAR5.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page G8-6106.

Configurations

AArch32 System register ID_ISAR3[31:0] is architecturally mapped to AArch64 System register ID_ISAR3_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_ISAR3 are UNKNOWN.

Attributes

ID_ISAR3 is a 32-bit register.

Field descriptions

The ID_ISAR3 bit assignments are:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
T32EE	TrueNOP	T32Copy	TabBranch	SynchPrim	SVC	SIMD	Saturate

T32EE, bits [31:28]

Indicates the implemented T32EE instructions. Defined values are:

`0b0000` None implemented.

`0b0001` Adds the ENTERX and LEAVEX instructions, and modifies the load behavior to include null checking.

All other values are reserved.

In Armv8-A the only permitted value is `0b0000`.

TrueNOP, bits [27:24]

Indicates the implemented true NOP instructions. Defined values are:

`0b0000` None implemented. This means there are no NOP instructions that do not have any register dependencies.

`0b0001` Adds true NOP instructions in both the T32 and A32 instruction sets. This also permits additional NOP-compatible hints.

All other values are reserved.

In Armv8-A the only permitted value is `0b0001`.

T32Copy, bits [23:20]

Indicates the support for T32 non flag-setting MOV instructions. Defined values are:

`0b0000` Not supported. This means that in the T32 instruction set, encoding T1 of the MOV (register) instruction does not support a copy from a low register to a low register.

`0b0001` Adds support for T32 instruction set encoding T1 of the MOV (register) instruction, copying from a low register to a low register.

All other values are reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In Armv8-A the only permitted value is `0b0001`.

TabBranch, bits [19:16]

Indicates the implemented Table Branch instructions in the T32 instruction set. Defined values are:

`0b0000` None implemented.

`0b0001` Adds the TBB and TBH instructions.

All other values are reserved.

In Armv8-A the only permitted value is `0b0001`.

SynchPrim, bits [15:12]

Used in conjunction with ID_ISAR4.SynchPrim_frac to indicate the implemented Synchronization Primitive instructions. Defined values are:

`0b0000` If SynchPrim_frac == `0b000`, no Synchronization Primitives implemented.

`0b0001` If SynchPrim_frac == `0b000`, adds the LDREX and STREX instructions.
If SynchPrim_frac == `0b011`, also adds the CLREX, LDREXB, STREXB, and STREXH instructions.

`0b0010` If SynchPrim_frac == `0b000`, as for [`0b001`, `0b011`] and also adds the LDREXD and STREXD instructions.

All other combinations of SynchPrim and SynchPrim_frac are reserved.

In Armv8-A the only permitted value is `0b0010`.

SVC, bits [11:8]

Indicates the implemented SVC instructions. Defined values are:

`0b0000` Not implemented.

`0b0001` Adds the SVC instruction.

All other values are reserved.

In Armv8-A the only permitted value is `0b0001`.

SIMD, bits [7:4]

Indicates the implemented SIMD instructions. Defined values are:

`0b0000` None implemented.

`0b0001` Adds the SSAT and USAT instructions, and the Q bit in the PSRs.

`0b0011` As for `0b0001`, and adds the PKHBT, PKHTB, QADD16, QADD8, QASX, QSUB16, QSUB8, QSAX, SADD16, SADD8, SASX, SEL, SHADD16, SHADD8, SHASX, SHSUB16, SHSUB8, SHSAX, SSAT16, SSUB16, SSUB8, SSAX, SXTAB16, SXTB16, UADD16, UADD8, UASX, UHADD16, UHADD8, UHASX, UHSUB16, UHSUB8, UHSAX, UQADD16, UQADD8, UQASX, UQSUB16, UQSUB8, UQSAX, USAD8, USADA8, USAT16, USUB16, USUB8, USAX, UXTAB16, and UXTB16 instructions. Also adds support for the GE[3:0] bits in the PSRs.

All other values are reserved.

In Armv8-A the only permitted value is `0b0011`.

The SIMD field relates only to implemented instructions that perform SIMD operations on the general-purpose registers. In an implementation that supports floating-point and Advanced SIMD instructions, MVFR0, MVFR1, and MVFR2 give information about the implemented Advanced SIMD instructions.

Saturate, bits [3:0]

Indicates the implemented Saturate instructions. Defined values are:

`0b0000` None implemented. This means no non-Advanced SIMD saturate instructions are implemented.

`0b0001` Adds the QADD, QDADD, QDSUB, and QSUB instructions, and the Q bit in the PSRs.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

All other values are reserved.

In Armv8-A the only permitted value is `0b0001`.

Accessing the ID_ISAR3

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0010	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        return ID_ISAR3;
elsif PSTATE.EL == EL2 then
    return ID_ISAR3;
elsif PSTATE.EL == EL3 then
    return ID_ISAR3;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.89 ID_ISAR4, Instruction Set Attribute Register 4

The ID_ISAR4 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR3, and ID_ISAR5.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page G8-6106.

Configurations

AArch32 System register ID_ISAR4[31:0] is architecturally mapped to AArch64 System register ID_ISAR4_EL1[31:0].

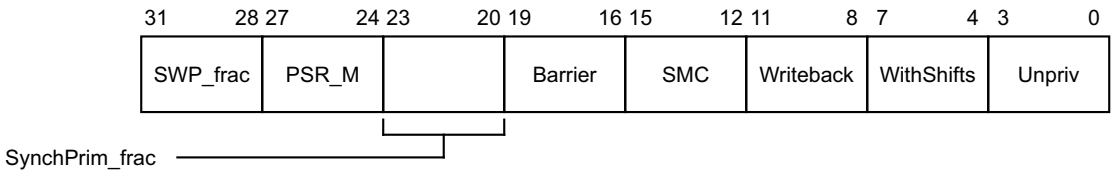
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_ISAR4 are UNKNOWN.

Attributes

ID_ISAR4 is a 32-bit register.

Field descriptions

The ID_ISAR4 bit assignments are:



SWP_frac, bits [31:28]

Indicates support for the memory system locking the bus for SWP or SWPB instructions. Defined values are:

`0b0000` SWP or SWPB instructions not implemented.

`0b0001` SWP or SWPB implemented but only in a uniprocessor context. SWP and SWPB do not guarantee whether memory accesses from other masters can come between the load memory access and the store memory access of the SWP or SWPB.

All other values are reserved. This field is valid only if ID_ISAR0.Swap is `0b0000`.

In Armv8-A, the only permitted value is `0b0000`.

PSR_M, bits [27:24]

Indicates the implemented M profile instructions to modify the PSRs. Defined values are:

`0b0000` None implemented.

`0b0001` Adds the M profile forms of the CPS, MRS, and MSR instructions.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0000`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

SynchPrim_frac, bits [23:20]

Used in conjunction with ID_ISAR3.SynchPrim to indicate the implemented Synchronization Primitive instructions. Possible values are:

`0b0000` If SynchPrim == `0b0000`, no Synchronization Primitives implemented. If SynchPrim == `0b0001`, adds the LDREX and STREX instructions. If SynchPrim == `0b0010`, also adds the CLREX, LDREXB, LDREXH, STREXB, STREXH, LDREXD, and STREXD instructions.

`0b0011` If SynchPrim == `0b0001`, adds the LDREX, STREX, CLREX, LDREXB, LDREXH, STREXB, and STREXH instructions.

All other combinations of SynchPrim and SynchPrim_frac are reserved.

In Armv8-A, the only permitted value is `0b0000`.

Barrier, bits [19:16]

Indicates the implemented Barrier instructions in the A32 and T32 instruction sets. Defined values are:

`0b0000` None implemented. Barrier operations are provided only as System instructions in the (coproc==`0b1111`) encoding space.

`0b0001` Adds the DMB, DSB, and ISB barrier instructions.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0001`.

SMC, bits [15:12]

Indicates the implemented SMC instructions. Defined values are:

`0b0000` None implemented.

`0b0001` Adds the SMC instruction.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0001`.

Writeback, bits [11:8]

Indicates the support for Writeback addressing modes. Defined values are:

`0b0000` Basic support. Only the LDM, STM, PUSH, POP, SRS, and RFE instructions support writeback addressing modes. These instructions support all of their writeback addressing modes.

`0b0001` Adds support for all of the writeback addressing modes.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0001`.

WithShifts, bits [7:4]

Indicates the support for instructions with shifts. Defined values are:

`0b0000` Nonzero shifts supported only in MOV and shift instructions.

`0b0001` Adds support for shifts of loads and stores over the range LSL 0-3.

`0b0011` As for `0b0001`, and adds support for other constant shift options, both on load/store and other instructions.

`0b0100` As for `0b0011`, and adds support for register-controlled shift options.

All other values are reserved.

In Armv8-A the only permitted value is `0b0100`.

Unpriv, bits [3:0]

Indicates the implemented unprivileged instructions. Defined values are:

`0b0000` None implemented. No T variant instructions are implemented.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b0001` Adds the LDRBT, LDRT, STRBT, and STRT instructions.

`0b0010` As for `0b0001`, and adds the LDRHT, LDRSBT, LDRSHT, and STRHT instructions.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0010`.

Accessing the ID_ISAR4

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0010	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        return ID_ISAR4;
elsif PSTATE.EL == EL2 then
    return ID_ISAR4;
elsif PSTATE.EL == EL3 then
    return ID_ISAR4;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.90 ID_ISAR5, Instruction Set Attribute Register 5

The ID_ISAR5 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR3, and ID_ISAR4.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page G8-6106.

Configurations

AArch32 System register ID_ISAR5[31:0] is architecturally mapped to AArch64 System register ID_ISAR5_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_ISAR5 are UNKNOWN.

Attributes

ID_ISAR5 is a 32-bit register.

Field descriptions

The ID_ISAR5 bit assignments are:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
VCMA	RDM	RES0	CRC32	SHA2	SHA1	AES	SEVL

VCMA, bits [31:28]

Indicates AArch32 support for complex number addition and multiplication where numbers are stored in vectors. Defined values are:

`0b0000` The VCMLA and VCADD instructions are not implemented in AArch32.

`0b0001` The VCMLA and VCADD instructions are implemented in AArch32.

All other values are reserved.

ARMv8.3-CompNum implements the functionality identified by `0b0001`.

From Armv8.3, the only permitted value is `0b0001`.

RDM, bits [27:24]

Indicates support for the VQRDMLAH and VQRDMLSH instructions in AArch32 state. Defined values are:

`0b0000` No VQRDMLAH and VQRDMLSH instructions implemented.

`0b0001` VQRDMLAH and VQRDMLSH instructions implemented.

All other values are reserved.

ARMv8.1-RDMA implements the functionality identified by the value `0b0001`.

From Armv8.1, the only permitted value is `0b0001`.

Bits [23:20]

Reserved, RES0.

CRC32, bits [19:16]

Indicates supports for the CRC32 instructions in AArch32 state. Defined values are:

`0b0000` No CRC32 instructions implemented.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b0001` CRC32B, CRC32H, CRC32W, CRC32CB, CRC32CH, and CRC32CW instructions implemented.

All other values are reserved.

In Armv8.0 the permitted values are `0b0000` and `0b0001`.

From Armv8.1, the only permitted value is `0b0001`.

SHA2, bits [15:12]

Indicates supports for the SHA2 instructions in AArch32 state.

`0b0000` No SHA2 instructions implemented.

`0b0001` SHA256H, SHA256H2, SHA256SU0, and SHA256SU1 implemented.

All other values are reserved.

In Armv8-A, the permitted values are `0b0000` and `0b0001`.

SHA1, bits [11:8]

Indicates support for the SHA1 instructions are implemented in AArch32 state. Defined values are:

`0b0000` No SHA1 instructions implemented.

`0b0001` SHA1C, SHA1P, SHA1M, SHA1H, SHA1SU0, and SHA1SU1 implemented.

All other values are reserved.

In Armv8-A the permitted values are `0b0000` and `0b0001`.

AES, bits [7:4]

Indicates support for the AES instructions in AArch32 state. Defined values are:

`0b0000` No AES instructions implemented.

`0b0001` AESE, AESD, AESMC, and AESIMC implemented.

`0b0010` As for `0b0001`, plus VMULL (polynomial) instructions operating on 64-bit data quantities.

All other values are reserved.

In Armv8-A the permitted values are `0b0000` and `0b0010`.

SEVL, bits [3:0]

Indicates support for the SEVL instruction in AArch32 state. Defined values are:

`0b0000` SEVL is implemented as a NOP.

`0b0001` SEVL is implemented as Send Event Local.

All other values are reserved.

In Armv8-A the only permitted value is `0b0001`.

Accessing the ID_ISAR5

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0010	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        return ID_ISAR5;
elsif PSTATE.EL == EL2 then
    return ID_ISAR5;
elsif PSTATE.EL == EL3 then
    return ID_ISAR5;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.91 ID_ISAR6, Instruction Set Attribute Register 6

The ID_ISAR6 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR3, ID_ISAR4 and ID_ISAR5.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page G8-6106.

Configurations

AArch32 System register ID_ISAR6[31:0] is architecturally mapped to AArch64 System register ID_ISAR6_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_ISAR6 are UNKNOWN.

Note

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.

Attributes

ID_ISAR6 is a 32-bit register.

Field descriptions

The ID_ISAR6 bit assignments are:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
RES0	I8MM	BF16	SPECRES	SB	FHM	DP	JSCVT

Bits [31:28]

Reserved, RES0.

I8MM, bits [27:24]

Indicates support for Advanced SIMD and floating-point Int8 matrix multiplication instructions in AArch32 state. Defined values are:

`0b0000` Int8 matrix multiplication instructions are not implemented.

`0b0001` VSMMLA, VSUDOT, VUMMLA, VUSMMLA, and VUSDOT instructions are implemented.

All other values are reserved.

ARMv8.2-AA32I8MM implements the functionality identified by `0b0001`.

From Armv8.2, the permitted values are `0b0000` and `0b0001`.

BF16, bits [23:20]

Indicates support for Advanced SIMD and floating-point BFloat16 instructions in AArch32 state. Defined values are:

`0b0000` BFloat16 instructions are not implemented.

`0b0001` VCVT, VCVTB, VCVTT, VDOT, VFMAL, and VMMLA instructions with BF16 operand or result types are implemented.

All other values are reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ARMv8.2-AA32BF16 implements the functionality identified by `0b0001`.

From Armv8.2, the permitted values are `0b0000` and `0b0001`.

SPECRES, bits [19:16]

Indicates support for Speculation invalidation instructions in AArch32 state. Defined values are:

`0b0000` CFPRCTX, DVPRCTX, and CPPRCTX instructions are not implemented.

`0b0001` CFPRCTX, DVPRCTX, and CPPRCTX instructions are implemented.

All other values are reserved.

From Armv8.5, the only permitted value is `0b0001`.

SB, bits [15:12]

Indicates support for SB instruction in AArch32 state. Defined values are:

`0b0000` SB instruction is not implemented.

`0b0001` SB instruction is implemented.

All other values are reserved.

From Armv8.5, the only permitted value is `0b0001`.

FHM, bits [11:8]

Indicates support for Advanced SIMD and floating-point VFMAL and VFMSL instructions in AArch32 state. Defined values are:

`0b0000` VFMAL and VMFSL instructions not implemented.

`0b0001` VFMAL and VMFSL instructions implemented.

ARMv8.2-FHM implements the functionality identified by the value `0b0001`.

DP, bits [7:4]

Indicates support for dot product instructions in AArch32 state. Defined values are:

`0b0000` No dot product instructions implemented.

`0b0001` VUDOT and VSDOT instructions implemented.

All other values are reserved.

ARMv8.2-DotProd implements the functionality identified by the value `0b0001`.

JSCVT, bits [3:0]

Indicates support for the Javascript conversion instruction in AArch32 state. Defined values are:

`0b0000` The VJCVT instruction is not implemented.

`0b0001` The VJCVT instruction is implemented.

All other values are reserved.

In Armv8.0, Armv8.1 and Armv8.2 the only permitted value is `0b0000`.

ARMv8.3-JSconv implements the functionality identified by `0b0001`.

From Armv8.3, if Advanced SIMD or Floating-point is implemented, the only permitted value is `0b0001`.

From Armv8.3, if Advanced SIMD or Floating-point is not implemented, the only permitted value is `0b0000`.

Accessing the ID_ISAR6

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0010	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!IsZero(ID_ISAR6) || boolean IMPLEMENTATION_DEFINED
"ID_ISAR6 trapped by HCR_EL2.TID3") && HCR_EL2.TID3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && (!IsZero(ID_ISAR6) || boolean IMPLEMENTATION_DEFINED
"ID_ISAR6 trapped by HCR.TID3") && HCR.TID3 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        return ID_ISAR6;
elsif PSTATE.EL == EL2 then
    return ID_ISAR6;
elsif PSTATE.EL == EL3 then
    return ID_ISAR6;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.92 ID_MMFR0, Memory Model Feature Register 0

The ID_MMFR0 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

Must be interpreted with ID_MMFR1, ID_MMFR2, ID_MMFR3, and ID_MMFR4.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page G8-6106.

Configurations

AArch32 System register ID_MMFR0[31:0] is architecturally mapped to AArch64 System register ID_MMFR0_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_MMFR0 are UNKNOWN.

Attributes

ID_MMFR0 is a 32-bit register.

Field descriptions

The ID_MMFR0 bit assignments are:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
InnerShr	FCSE	AuxReg	TCM	ShareLvl	OuterShr	PMSA	VMSA

InnerShr, bits [31:28]

Innermost Shareability. Indicates the innermost shareability domain implemented. Defined values are:

`0b0000` Implemented as Non-cacheable.

`0b0001` Implemented with hardware coherency support.

`0b1111` Shareability ignored.

All other values are reserved.

From Armv8 the permitted values are `0b0000`, `0b0001`, and `0b1111`.

This field is valid only if the implementation supports two levels of shareability, as indicated by ID_MMFR0.ShareLvl having the value `0b0001`.

When ID_MMFR0.ShareLvl is zero, this field is UNKNOWN.

FCSE, bits [27:24]

Indicates whether the implementation includes the FCSE. Defined values are:

`0b0000` Not supported.

`0b0001` Support for FCSE.

All other values are reserved.

From Armv8 the only permitted value is `0b0000`.

AuxReg, bits [23:20]

Auxiliary Registers. Indicates support for Auxiliary registers. Defined values are:

`0b0000` None supported.

`0b0001` Support for Auxiliary Control Register only.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b0010` Support for Auxiliary Fault Status Registers (AIFSR and ADFSR) and Auxiliary Control Register.

All other values are reserved.

From Armv8 the only permitted value is `0b0010`.

Note

Accesses to unimplemented Auxiliary registers are UNDEFINED.

TCM, bits [19:16]

Indicates support for TCMs and associated DMAs. Defined values are:

`0b0000` Not supported.

`0b0001` Support is IMPLEMENTATION DEFINED. Armv7 requires this setting.

`0b0010` Support for TCM only, Armv6 implementation.

`0b0011` Support for TCM and DMA, Armv6 implementation.

All other values are reserved.

In Armv8-A the only permitted value is `0b0000`.

ShareLvl, bits [15:12]

Shareability Levels. Indicates the number of shareability levels implemented. Defined values are:

`0b0000` One level of shareability implemented.

`0b0001` Two levels of shareability implemented.

All other values are reserved.

From Armv8 the only permitted value is `0b0001`.

OuterShr, bits [11:8]

Outermost Shareability. Indicates the outermost shareability domain implemented. Defined values are:

`0b0000` Implemented as Non-cacheable.

`0b0001` Implemented with hardware coherency support.

`0b1111` Shareability ignored.

All other values are reserved.

From Armv8 the permitted values are `0b0000`, `0b0001`, and `0b1111`.

PMSA, bits [7:4]

Indicates support for a PMSA. Defined values are:

`0b0000` Not supported.

`0b0001` Support for IMPLEMENTATION DEFINED PMSA.

`0b0010` Support for PMSAv6, with a Cache Type Register implemented.

`0b0011` Support for PMSAv7, with support for memory subsections. Armv7-R profile.

All other values are reserved.

In Armv8-A the only permitted value is `0b0000`.

VMSA, bits [3:0]

Indicates support for a VMSA. Defined values are:

`0b0000` Not supported.

`0b0001` Support for IMPLEMENTATION DEFINED VMSA.

`0b0010` Support for VMSAv6, with Cache and TLB Type Registers implemented.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b0011` Support for VMSAv7, with support for remapping and the Access flag. ARMv7-A profile.

`0b0100` As for `0b0011`, and adds support for the PXN bit in the Short-descriptor translation table format descriptors.

`0b0101` As for `0b0100`, and adds support for the Long-descriptor translation table format.

All other values are reserved.

In Armv8-A the only permitted value is `0b0101`.

Accessing the ID_MMFR0

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0001	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        return ID_MMFR0;
elsif PSTATE.EL == EL2 then
    return ID_MMFR0;
elsif PSTATE.EL == EL3 then
    return ID_MMFR0;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.93 ID_MMFR1, Memory Model Feature Register 1

The ID_MMFR1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

Must be interpreted with ID_MMFR0, ID_MMFR2, ID_MMFR3, and ID_MMFR4.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page G8-6106.

Configurations

AArch32 System register ID_MMFR1[31:0] is architecturally mapped to AArch64 System register ID_MMFR1_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_MMFR1 are UNKNOWN.

Attributes

ID_MMFR1 is a 32-bit register.

Field descriptions

The ID_MMFR1 bit assignments are:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
BPred	L1TstCln	L1Uni	L1Hvd	L1UniSW	L1HvdSW	L1UniVA	L1HvdVA

BPred, bits [31:28]

Branch Predictor. Indicates branch predictor management requirements. Defined values are:

`0b0000` No branch predictor, or no MMU present. Implies a fixed MPU configuration.

`0b0001` Branch predictor requires flushing on:

- Enabling or disabling a stage of address translation.
- Writing new data to instruction locations.
- Writing new mappings to the translation tables.
- Changes to the TTBR0, TTBR1, or TTBCR registers.
- Changes to the ContextID or ASID, or to the FCSE ProcessID if this is supported.

`0b0010` Branch predictor requires flushing on:

- Enabling or disabling a stage of address translation.
- Writing new data to instruction locations.
- Writing new mappings to the translation tables.
- Any change to the TTBR0, TTBR1, or TTBCR registers without a change to the corresponding ContextID or ASID, or FCSE ProcessID if this is supported.

`0b0011` Branch predictor requires flushing only on writing new data to instruction locations.

`0b0100` For execution correctness, branch predictor requires no flushing at any time.

All other values are reserved.

In Armv8-A the permitted values are `0b0010`, `0b0011`, or `0b0100`. For values other than `0b0000` and `0b0100` the Arm Architecture Reference Manual, or the product documentation, might give more information about the required maintenance.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

L1TstCln, bits [27:24]

Level 1 cache Test and Clean. Indicates the supported Level 1 data cache test and clean operations, for Harvard or unified cache implementations. Defined values are:

`0b0000` None supported.

`0b0001` Supported Level 1 data cache test and clean operations are:

- Test and clean data cache.

`0b0010` As for 0001, and adds:

- Test, clean, and invalidate data cache.

All other values are reserved.

In Armv8-A the only permitted value is `0b0000`.

L1Uni, bits [23:20]

Level 1 Unified cache. Indicates the supported entire Level 1 cache maintenance operations for a unified cache implementation. Defined values are:

`0b0000` None supported.

`0b0001` Supported entire Level 1 cache operations are:

- Invalidate cache, including branch predictor if appropriate.
- Invalidate branch predictor, if appropriate.

`0b0010` As for 0001, and adds:

- Clean cache, using a recursive model that uses the cache dirty status bit.
- Clean and invalidate cache, using a recursive model that uses the cache dirty status bit.

All other values are reserved.

In Armv8-A the only permitted value is `0b0000`.

L1Hvd, bits [19:16]

Level 1 Harvard cache. Indicates the supported entire Level 1 cache maintenance operations for a Harvard cache implementation. Defined values are:

`0b0000` None supported.

`0b0001` Supported entire Level 1 cache operations are:

- Invalidate instruction cache, including branch predictor if appropriate.
- Invalidate branch predictor, if appropriate.

`0b0010` As for 0001, and adds:

- Invalidate data cache.
- Invalidate data cache and instruction cache, including branch predictor if appropriate.

`0b0011` As for 0010, and adds:

- Clean data cache, using a recursive model that uses the cache dirty status bit.
- Clean and invalidate data cache, using a recursive model that uses the cache dirty status bit.

All other values are reserved.

In Armv8-A the only permitted value is `0b0000`.

L1UniSW, bits [15:12]

Level 1 Unified cache by Set/Way. Indicates the supported Level 1 cache line maintenance operations by set/way, for a unified cache implementation. Defined values are:

`0b0000` None supported.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b0001` Supported Level 1 unified cache line maintenance operations by set/way are:

- Clean cache line by set/way.

`0b0010` As for 0001, and adds:

- Clean and invalidate cache line by set/way.

`0b0011` As for 0010, and adds:

- Invalidate cache line by set/way.

All other values are reserved.

In Armv8-A the only permitted value is `0b0000`.

L1HvdSW, bits [11:8]

Level 1 Harvard cache by Set/Way. Indicates the supported Level 1 cache line maintenance operations by set/way, for a Harvard cache implementation. Defined values are:

`0b0000` None supported.

`0b0001` Supported Level 1 Harvard cache line maintenance operations by set/way are:

- Clean data cache line by set/way.
- Clean and invalidate data cache line by set/way.

`0b0010` As for 0001, and adds:

- Invalidate data cache line by set/way.

`0b0011` As for 0010, and adds:

- Invalidate instruction cache line by set/way

All other values are reserved.

In Armv8-A the only permitted value is `0b0000`.

L1UniVA, bits [7:4]

Level 1 Unified cache by Virtual Address. Indicates the supported Level 1 cache line maintenance operations by VA, for a unified cache implementation. Defined values are:

`0b0000` None supported.

`0b0001` Supported Level 1 unified cache line maintenance operations by VA are:

- Clean cache line by VA.
- Invalidate cache line by VA.
- Clean and invalidate cache line by VA.

`0b0010` As for 0001, and adds:

- Invalidate branch predictor by VA, if branch predictor is implemented.

All other values are reserved.

In Armv8-A the only permitted value is `0b0000`.

L1HvdVA, bits [3:0]

Level 1 Harvard cache by Virtual Address. Indicates the supported Level 1 cache line maintenance operations by VA, for a Harvard cache implementation. Defined values are:

`0b0000` None supported.

`0b0001` Supported Level 1 Harvard cache line maintenance operations by VA are:

- Clean data cache line by VA.
- Invalidate data cache line by VA.
- Clean and invalidate data cache line by VA.
- Clean instruction cache line by VA.

`0b0010` As for 0001, and adds:

- Invalidate branch predictor by VA, if branch predictor is implemented.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

All other values are reserved.

In Armv8-A the only permitted value is `0b0000`.

Accessing the ID_MMFR1

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0001	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        return ID_MMFR1;
elsif PSTATE.EL == EL2 then
    return ID_MMFR1;
elsif PSTATE.EL == EL3 then
    return ID_MMFR1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.94 ID_MMFR2, Memory Model Feature Register 2

The ID_MMFR2 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

Must be interpreted with ID_MMFR0, ID_MMFR1, ID_MMFR3, and ID_MMFR4.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page G8-6106.

Configurations

AArch32 System register ID_MMFR2[31:0] is architecturally mapped to AArch64 System register ID_MMFR2_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_MMFR2 are UNKNOWN.

Attributes

ID_MMFR2 is a 32-bit register.

Field descriptions

The ID_MMFR2 bit assignments are:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
HWAccFlg	WFIStall	MemBarr	UniTLB	HvdTLB	L1HvdRng	L1HvdBG	L1HvdFG

HWAccFlg, bits [31:28]

Hardware Access Flag. In earlier versions of the Arm Architecture, this field indicates support for a Hardware Access flag, as part of the VMSAv7 implementation. Defined values are:

`0b0000` Not supported.

`0b0001` Support for VMSAv7 Access flag, updated in hardware.

All other values are reserved.

From Armv8 the only permitted value is `0b000`.

WFIStall, bits [27:24]

Wait For Interrupt Stall. Indicates the support for Wait For Interrupt (WFI) stalling. Defined values are:

`0b0000` Not supported.

`0b0001` Support for WFI stalling.

All other values are reserved.

From Armv8 the permitted values are `0b000` and `0b001`.

MemBarr, bits [23:20]

Memory Barrier. Indicates the supported memory barrier System instructions in the (coproc==1111) encoding space:

`0b0000` None supported.

`0b0001` Supported memory barrier System instructions are:

- Data Synchronization Barrier (DSB).

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b0010` As for `0b001`, and adds:

- Instruction Synchronization Barrier (ISB).
- Data Memory Barrier (DMB).

All other values are reserved.

From Armv8 the only permitted value is `0b010`.

Arm deprecates the use of these operations. ID_ISAR4.Barrier_instrs indicates the level of support for the preferred barrier instructions.

UniTLB, bits [19:16]

Unified TLB. Indicates the supported TLB maintenance operations, for a unified TLB implementation. Defined values are:

`0b0000` Not supported.

`0b0001` Supported unified TLB maintenance operations are:

- Invalidate all entries in the TLB.
- Invalidate TLB entry by VA.

`0b0010` As for `0b001`, and adds:

- Invalidate TLB entries by ASID match.

`0b0011` As for `0b010`, and adds:

- Invalidate instruction TLB and data TLB entries by VA All ASID. This is a shared unified TLB operation

`0b0100` As for `0b011`, and adds:

- Invalidate Hyp mode unified TLB entry by VA.
- Invalidate entire Non-secure PL1&0 unified TLB.
- Invalidate entire Hyp mode unified TLB.

`0b0101` As for `0b100`, and adds the following operations: TLBIMVALIS, TLBIMVAALIS, TLBIMVALHIS, TLBIMVAL, TLBIMVAAL,TLBIMVALH.

`0b0110` As for `0b101`, and adds the following operations: TLBIIPAS2IS, TLBIIPAS2LIS, TLBIIPAS2, TLBIIPAS2L.

All other values are reserved.

In Armv8-A the only permitted value is `0b110`.

HvdTLB, bits [15:12]

If the Unified TLB field (UniTLB, bits [19:16]) is not 0000, then the meaning of this field is IMPLEMENTATION DEFINED. Arm deprecates the use of this field by software.

L1HvdRng, bits [11:8]

Level 1 Harvard cache Range. Indicates the supported Level 1 cache maintenance range operations, for a Harvard cache implementation. Defined values are:

`0b0000` Not supported.

`0b0001` Supported Level 1 Harvard cache maintenance range operations are:

- Invalidate data cache range by VA.
- Invalidate instruction cache range by VA.
- Clean data cache range by VA.
- Clean and invalidate data cache range by VA.

All other values are reserved.

From Armv8 the only permitted value is `0b0000`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

L1HvdBG, bits [7:4]

Level 1 Harvard cache Background fetch. Indicates the supported Level 1 cache background fetch operations, for a Harvard cache implementation. When supported, background fetch operations are non-blocking operations. Defined values are:

`0b0000` Not supported.

`0b0001` Supported Level 1 Harvard cache background fetch operations are:

- Fetch instruction cache range by VA.
- Fetch data cache range by VA.

All other values are reserved.

From Armv8 the only permitted value is `0b0000`.

L1HvdFG, bits [3:0]

Level 1 Harvard cache Foreground fetch. Indicates the supported Level 1 cache foreground fetch operations, for a Harvard cache implementation. When supported, foreground fetch operations are blocking operations. Defined values are:

`0b0000` Not supported.

`0b0001` Supported Level 1 Harvard cache foreground fetch operations are:

- Fetch instruction cache range by VA.
- Fetch data cache range by VA.

All other values are reserved.

From Armv8 the only permitted value is `0b0000`.

Accessing the ID_MMFR2

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0001	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        return ID_MMFR2;
elsif PSTATE.EL == EL2 then
    return ID_MMFR2;
elsif PSTATE.EL == EL3 then
    return ID_MMFR2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.95 ID_MMFR3, Memory Model Feature Register 3

The ID_MMFR3 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

Must be interpreted with ID_MMFR0, ID_MMFR1, ID_MMFR2, and ID_MMFR4.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page G8-6106.

Configurations

AArch32 System register ID_MMFR3[31:0] is architecturally mapped to AArch64 System register ID_MMFR3_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_MMFR3 are UNKNOWN.

Attributes

ID_MMFR3 is a 32-bit register.

Field descriptions

The ID_MMFR3 bit assignments are:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
Supersec	CMemSz	CohWalk	PAN	MaintBcst	BPMaint	CMaintSW	CMaintVA

Supersec, bits [31:28]

Supersections. On a VMSA implementation, indicates whether Supersections are supported. Defined values are:

`0b0000` Supersections supported.

`0b1111` Supersections not supported.

All other values are reserved.

In Armv8-A the permitted values are `0b0000` and `0b1111`.

CMemSz, bits [27:24]

Cached Memory Size. Indicates the physical memory size supported by the caches. Defined values are:

`0b0000` 4GB, corresponding to a 32-bit physical address range.

`0b0001` 64GB, corresponding to a 36-bit physical address range.

`0b0010` 1TB or more, corresponding to a 40-bit or larger physical address range.

All other values are reserved.

In Armv8-A the permitted values are `0b0000`, `0b0001`, and `0b0010`.

CohWalk, bits [23:20]

Coherent Walk. Indicates whether Translation table updates require a clean to the Point of Unification. Defined values are:

`0b0000` Updates to the translation tables require a clean to the Point of Unification to ensure visibility by subsequent translation table walks.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b0001` Updates to the translation tables do not require a clean to the Point of Unification to ensure visibility by subsequent translation table walks.

All other values are reserved.

In Armv8-A the only permitted value is `0b0001`.

PAN, bits [19:16]

Privileged Access Never. Indicates support for the PAN bit in CPSR, SPSR, and DSPSR in AArch32 state. Defined values are:

`0b0000` PAN not supported.

`0b0001` PAN supported.

`0b0010` PAN supported and ATS1CPRP and ATS1CPWP instructions supported.

All other values are reserved.

ARMv8.1-PAN implements the functionality identified by the value `0b0001`.

ARMv8.2-ATS1E1 implements the functionality added by the value `0b0010`.

In Armv8.1 the value `0b0000` is not permitted.

From Armv8.2, the only permitted value is `0b0010`.

MaintBcst, bits [15:12]

Maintenance Broadcast. Indicates whether Cache, TLB, and branch predictor operations are broadcast. Defined values are:

`0b0000` Cache, TLB, and branch predictor operations only affect local structures.

`0b0001` Cache and branch predictor operations affect structures according to shareability and defined behavior of instructions. TLB operations only affect local structures.

`0b0010` Cache, TLB, and branch predictor operations affect structures according to shareability and defined behavior of instructions.

All other values are reserved.

In Armv8-A the only permitted value is `0b0010`.

BPMaint, bits [11:8]

Branch Predictor Maintenance. Indicates the supported branch predictor maintenance operations in an implementation with hierarchical cache maintenance operations. Defined values are:

`0b0000` None supported.

`0b0001` Supported branch predictor maintenance operations are:

- Invalidate all branch predictors.

`0b0010` As for 0001, and adds:

- Invalidate branch predictors by VA.

All other values are reserved.

In Armv8-A the only permitted value is `0b0010`.

CMaintSW, bits [7:4]

Cache Maintenance by Set/Way. Indicates the supported cache maintenance operations by set/way, in an implementation with hierarchical caches. Defined values are:

`0b0000` None supported.

`0b0001` Supported hierarchical cache maintenance instructions by set/way are:

- Invalidate data cache by set/way.
- Clean data cache by set/way.
- Clean and invalidate data cache by set/way.

All other values are reserved.

In Armv8-A the only permitted value is `0b0001`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In a unified cache implementation, the data cache maintenance operations apply to the unified caches.

CMaintVA, bits [3:0]

Cache Maintenance by Virtual Address. Indicates the supported cache maintenance operations by VA, in an implementation with hierarchical caches. Defined values are:

`0b0000` None supported.

`0b0001` Supported hierarchical cache maintenance operations by VA are:

- Invalidate data cache by VA.
- Clean data cache by VA.
- Clean and invalidate data cache by VA.
- Invalidate instruction cache by VA.
- Invalidate all instruction cache entries.

All other values are reserved.

In Armv8-A the only permitted value is `0b0001`.

In a unified cache implementation, data cache maintenance operations apply to the unified caches, and the instruction cache maintenance instructions are not implemented.

Accessing the ID_MMFR3

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0001	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        return ID_MMFR3;
elsif PSTATE.EL == EL2 then
    return ID_MMFR3;
elsif PSTATE.EL == EL3 then
    return ID_MMFR3;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.96 ID_MMFR4, Memory Model Feature Register 4

The ID_MMFR4 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

Must be interpreted with ID_MMFR0, ID_MMFR1, ID_MMFR2, and ID_MMFR3.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page G8-6106.

Configurations

AArch32 System register ID_MMFR4[31:0] is architecturally mapped to AArch64 System register ID_MMFR4_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_MMFR4 are UNKNOWN.

Attributes

ID_MMFR4 is a 32-bit register.

Field descriptions

The ID_MMFR4 bit assignments are:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
EVT	CCIDX	LSM	HPDS	CnP	XNX	AC2	SpecSEI

EVT, bits [31:28]

Enhanced Virtualization Traps. If EL2 is implemented, indicates support for the HCR2.{TTLBIS, TOCU, TICAB, TID4} traps. Defined values are:

`0b0000` HCR2.{TTLBIS, TOCU, TICAB, TID4} traps are not supported.

`0b0001` HCR2.{TOCU, TICAB, TID4} traps are supported. HCR2.TTLBIS trap is not supported.

`0b0010` HCR2.{TTLBIS, TOCU, TICAB, TID4} traps are supported.

All other values are reserved.

ARMv8.2-EVT implements the functionality identified by the values `0b0001` and `0b0010`.

If EL2 is not implemented, the only permitted value is `0b0000`.

From Armv8.1, the permitted values are `0b0000` and `0b0001`.

From Armv8.5, if EL2 is implemented, the only permitted value is `0b0010`.

CCIDX, bits [27:24]

Support for use of the revised CCSIDR format and the presence of the CCSIDR2 is indicated. Defined values are:

`0b0000` 32-bit format implemented for all levels of the CCSIDR, and the CCSIDR2 register is not implemented.

`0b0001` 64-bit format implemented for all levels of the CCSIDR, and the CCSIDR2 register is implemented.

All other values are reserved.

ARMv8.3-CCIDX implements the functionality identified by `0b0001`.

From Armv8.3, the permitted values are `0b0000` and `0b0001`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

LSM, bits [23:20]

Indicates support for LSMAOE and nTLSMD bits in HSCTLR and SCTLR. Defined values are:

`0b0000` LSMAOE and nTLSMD bits not supported.

`0b0001` LSMAOE and nTLSMD bits supported.

All other values are reserved.

ARMv8.2-LSMAOC implements the functionality identified by the value `0b0001`.

From Armv8.2, the permitted values are `0b0000` and `0b0001`.

HPDS, bits [19:16]

Hierarchical permission disables bits in translation tables. Defined values are:

`0b0000` Disabling of hierarchical controls not supported.

`0b0001` Supports disabling of hierarchical controls using the TTBCR2.HPD0, TTBCR2.HPD1, and HTCR.HPD bits.

`0b0010` As for value `0b0001`, and adds possible hardware allocation of bits[62:59] of the translation table descriptors from the final lookup level for IMPLEMENTATION DEFINED use.

All other values are reserved.

ARMv8.2-AA32HPD implements the functionality identified by the value `0b0001`.

ARMv8.2-TTPBHA implements the functionality added by the value `0b0010`.

Note

The value `0b0000` implies that the encoding for TTBCR2 is UNDEFINED.

CnP, bits [15:12]

Common not Private translations. Defined values are:

`0b0000` Common not Private translations not supported.

`0b0001` Common not Private translations supported.

All other values are reserved.

ARMv8.2-TTCNP implements the functionality identified by the value `0b0001`.

From Armv8.2 the only permitted value is `0b0001`.

XNX, bits [11:8]

Support for execute-never control distinction by Exception level at stage 2. Defined values are:

`0b0000` Distinction between EL0 and EL1 execute-never control at stage 2 not supported.

`0b0001` Distinction between EL0 and EL1 execute-never control at stage 2 supported.

All other values are reserved.

ARMv8.2-TTS2UXN implements the functionality identified by the value `0b0001`.

When ARMv8.2-TTS2UXN is implemented:

- If all of the following conditions are true, it is IMPLEMENTATION DEFINED whether the value of ID_MMFR4.XNX is `0b0000` or `0b0001`:
  - ID_AA64MMFR1_EL1.XNX ==1.
  - EL2 cannot use AArch32.
  - EL1 can use AArch32.
- If EL2 can use AArch32 then the only permitted value is `0b0001`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

AC2, bits [7:4]

Indicates the extension of the ACTLR and HACTLR registers using ACTLR2 and HACTLR2. Defined values are:

`0b0000` ACTLR2 and HACTLR2 are not implemented.

`0b0001` ACTLR2 and HACTLR2 are implemented.

All other values are reserved.

In Armv8.0 and Armv8.1 the permitted values are `0b0000` and `0b0001`.

From Armv8.2, the only permitted value is `0b0001`.

SpecSEI, bits [3:0]

Describes whether the PE can generate SError interrupt exceptions from speculative reads of memory, including speculative instruction fetches. The defined values of this field are:

`0b0000` The PE never generates an SError interrupt due to an External abort on a speculative read.

`0b0001` The PE might generate an SError interrupt due to an External abort on a speculative read.

All other values are reserved.

Accessing the ID_MMFR4

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0010	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!IsZero(ID_MMFR4) || boolean IMPLEMENTATION_DEFINED
"ID_MMFR4 trapped by HCR_EL2.TID3") && HCR_EL2.TID3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && (!IsZero(ID_MMFR4) || boolean IMPLEMENTATION_DEFINED
"ID_MMFR4 trapped by HCR.TID3") && HCR.TID3 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        return ID_MMFR4;
elsif PSTATE.EL == EL2 then
    return ID_MMFR4;
elsif PSTATE.EL == EL3 then
    return ID_MMFR4;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.97 ID_MMFR5, Memory Model Feature Register 5

The ID_MMFR5 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page G8-6106.

Configurations

AArch32 System register ID_MMFR5[31:0] is architecturally mapped to AArch64 System register ID_MMFR5_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_MMFR5 are UNKNOWN.

Attributes

ID_MMFR5 is a 32-bit register.

Field descriptions

The ID_MMFR5 bit assignments are:

31 … 4	3 … 0
RES0	ETS

Bits [31:4]

Reserved, RES0.

ETS, bits [3:0]

Support for Enhanced Translation Synchronization. Defined values are:

`0b0000` Enhanced Translation Synchronization is not supported.

`0b0001` Enhanced Translation Synchronization is supported.

All other values are reserved.

ARMv8.0-ETS implements the functionality identified by the value `0b0001`.

From Armv8.0, the permitted values are `0b0000` and `0b0001`.

Accessing the ID_MMFR5

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0011	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!IsZero(ID_MMFR5) || boolean IMPLEMENTATION_DEFINED
"ID_MMFR5 trapped by HCR_EL2.TID3") && HCR_EL2.TID3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && (!IsZero(ID_MMFR5) || boolean IMPLEMENTATION_DEFINED
"ID_MMFR5 trapped by HCR.TID3") && HCR.TID3 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        return ID_MMFR5;
elsif PSTATE.EL == EL2 then
    return ID_MMFR5;
elsif PSTATE.EL == EL3 then
    return ID_MMFR5;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.98 ID_PFR0, Processor Feature Register 0

The ID_PFR0 characteristics are:

Purpose

Gives top-level information about the instruction sets and other features supported by the PE in AArch32 state.

Must be interpreted with ID_PFR1.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page G8-6106.

Configurations

AArch32 System register ID_PFR0[31:0] is architecturally mapped to AArch64 System register ID_PFR0_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_PFR0 are UNKNOWN.

Attributes

ID_PFR0 is a 32-bit register.

Field descriptions

The ID_PFR0 bit assignments are:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
RAS	DIT	AMU	CSV2	State3	State2	State1	State0

RAS, bits [31:28]

RAS Extension version. Defined values are:

`0b0000` No RAS Extension.

`0b0001` RAS Extension present.

`0b0010` ARMv8.4-RAS present. As `0b0001`, and adds support for additional ERXMISC<m> System registers.

Error records accessed through System registers conform to RAS System Architecture v1.1, which includes simplifications to ERR<n>STATUS and support for the optional RAS Timestamp Extension.

All other values are reserved.

From Armv8.4, when ARMv8.4-DFE is not implemented, and ERRIDR.NUM is zero, the permitted values are IMPLEMENTATION DEFINED `0b0001` or `0b0010`. Otherwise from Armv8.4 the only permitted value is `0b0010`.

ARMv8.4-RAS implements the functionality identified by the value `0b0010`.

In Armv8.2, the only permitted value is `0b0001`.

In Armv8.1 and Armv8.0, the permitted values are `0b0000` and `0b0001`.

DIT, bits [27:24]

Data Independent Timing. Defined values are:

`0b0000` AArch32 does not guarantee constant execution time of any instructions.

`0b0001` AArch32 provides the CPSR.DIT mechanism to guarantee constant execution time of certain instructions.

All other values are reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ARMv8.4-DIT implements the functionality identified by the value `0b0001`.

From Armv8.4, the only permitted value is `0b0001`.

AMU, bits [23:20]

Activity Monitors Extension. Defined values are:

`0b0000` Activity Monitors Extension is not implemented.

`0b0001` AMUv1 for Armv8.4 is implemented.

`0b0010` AMUv1 for Armv8.6 is implemented. As `0b0001` and adds support for virtualization of the activity monitor event counters.

All other values are reserved.

AMUv1 implements the functionality identified by the value `0b0001`.

ARMv8.6-AMU implements the functionality identified by the value `0b0010`.

CSV2, bits [19:16]

Speculative use of out of context branch targets. Defined values are:

`0b0000` This Device does not disclose whether branch targets trained in one hardware described context can affect speculative execution in a different hardware described context.

`0b0001` Branch targets trained in one hardware described context can only affect speculative execution in a different hardware described context in a hard-to-determine way.

All other values are reserved.

ARMv8.0-CSV2 implements the functionality identified by `0b0001`.

From Armv8.5, the only permitted value is `0b0001`.

State3, bits [15:12]

T32EE instruction set support. Defined values are:

`0b0000` Not implemented.

`0b0001` T32EE instruction set implemented.

All other values are reserved.

In Armv8-A the only permitted value is `0b0000`.

State2, bits [11:8]

Jazelle extension support. Defined values are:

`0b0000` Not implemented.

`0b0001` Jazelle extension implemented, without clearing of JOSCR.CV on exception entry.

`0b0010` Jazelle extension implemented, with clearing of JOSCR.CV on exception entry.

All other values are reserved.

In Armv8-A the only permitted value is `0b0001`.

State1, bits [7:4]

T32 instruction set support. Defined values are:

`0b0000` T32 instruction set not implemented.

`0b0001` T32 encodings before the introduction of Thumb-2 technology implemented:

- All instructions are 16-bit.
- A BL or BLX is a pair of 16-bit instructions
- 32-bit instructions other than BL and BLX cannot be encoded.

`0b0011` T32 encodings after the introduction of Thumb-2 technology implemented, for all 16-bit and 32-bit T32 basic instructions.

All other values are reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In Armv8-A the only permitted value is `0b0011`.

State0, bits [3:0]

A32 instruction set support. Defined values are:

`0b0000` A32 instruction set not implemented.

`0b0001` A32 instruction set implemented.

All other values are reserved.

In Armv8-A the only permitted value is `0b0001`.

Accessing the ID_PFR0

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        return ID_PFR0;
elsif PSTATE.EL == EL2 then
    return ID_PFR0;
elsif PSTATE.EL == EL3 then
    return ID_PFR0;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.99 ID_PFR1, Processor Feature Register 1

The ID_PFR1 characteristics are:

Purpose

Gives information about the AArch32 programmers' model.

Must be interpreted with ID_PFR0.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page G8-6106.

Configurations

AArch32 System register ID_PFR1[31:0] is architecturally mapped to AArch64 System register ID_PFR1_EL1[31:0].

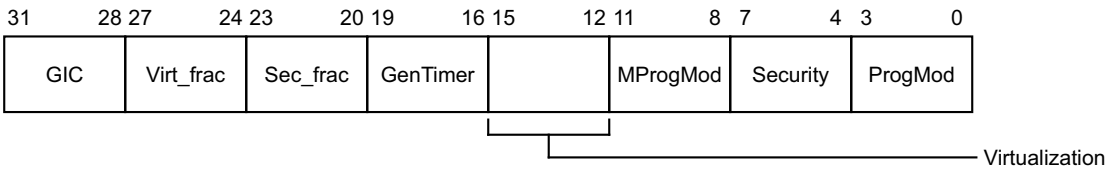
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_PFR1 are UNKNOWN.

Attributes

ID_PFR1 is a 32-bit register.

Field descriptions

The ID_PFR1 bit assignments are:



GIC, bits [31:28]

System register GIC CPU interface. Defined values are:

`0b0000` GIC CPU interface system registers not implemented.

`0b0001` System register interface to versions 3.0 and 4.0 of the GIC CPU interface is supported.

`0b0011` System register interface to version 4.1 of the GIC CPU interface is supported.

All other values are reserved.

Virt_frac, bits [27:24]

Virtualization fractional field. When the Virtualization field is `0b0000`, determines the support for features from the ARMv7 Virtualization Extensions. Defined values are:

`0b0000` No features from the ARMv7 Virtualization Extensions are implemented.

`0b0001` The following features of the ARMv7 Virtualization Extensions are implemented:

- The SCR.SIF bit, if EL3 is implemented.
- The modifications to the SCR.AW and SCR.FW bits described in the Virtualization Extensions, if EL3 is implemented.
- The MSR (banked register) and MRS (banked register) instructions.
- The ERET instruction.

All other values are reserved.

In Armv8-A the permitted values are:

- `0b0000` when EL2 is implemented.
- `0b0001` when EL2 is not implemented.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field is only valid when the value of ID_PFR1.Virtualization is 0, otherwise it holds the value `0b0000`.

Note

The ID_ISAR registers do not identify whether the instructions added by the ARMv7 Virtualization Extensions are implemented.

Sec_frac, bits [23:20]

Security fractional field. When the Security field is `0b0000`, determines the support for features from the ARMv7 Security Extensions. Defined values are:

`0b0000` No features from the ARMv7 Security Extensions are implemented.

`0b0001` The following features from the ARMv7 Security Extensions are implemented:

- The VBAR register.
- The TTBCR.PD0 and TTBCR.PD1 bits.

`0b0010` As for `0b0001`, plus the ability to access Secure or Non-secure physical memory is supported.

All other values are reserved.

In Armv8-A the permitted values are:

- `0b0000` when EL3 is implemented.
- `0b0001` or `0b0010` when EL3 is not implemented.

This field is only valid when the value of ID_PFR1.Security is 0, otherwise it holds the value `0b0000`.

GenTimer, bits [19:16]

Generic Timer support. Defined values are:

`0b0000` Generic Timer is not implemented.

`0b0001` Generic Timer is implemented.

`0b0010` Generic Timer is implemented, and also includes support for CNTHCTL.EVNTIS and CNTKCTL.EVNTIS fields, and CNTPCTSS and CNTVCTSS counter views.

All other values are reserved.

ARMv8.6-ECV implements the functionality identified by the value `0b0010`.

From Armv8.0 to Armv8.4, the only permitted value is `0b0001`.

From Armv8.6, the only permitted value is `0b0010`.

Virtualization, bits [15:12]

Virtualization support. Defined values are:

`0b0000` EL2, Hyp mode, and the HVC instruction not implemented.

`0b0001` EL2, Hyp mode, the HVC instruction, and all the features described by Virt_frac == `0b0001` implemented.

All other values are reserved.

In Armv8-A the permitted values are:

- `0b0000` when EL2 is not implemented.
- `0b0001` when EL2 is implemented.

In an implementation that includes EL2, if EL2 cannot use AArch32 but EL1 can use AArch32 then this field has the value `0b0001`.

Note

The ID_ISARs do not identify whether the HVC instruction is implemented.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MProgMod, bits [11:8]

M profile programmers' model support. Defined values are:

`0b0000` Not supported.

`0b0010` Support for two-stack programmers' model.

All other values are reserved.

In Armv8-A the only permitted value is `0b0000`.

Security, bits [7:4]

Security support. Defined values are:

`0b0000` EL3, Monitor mode, and the SMC instruction not implemented.

`0b0001` EL3, Monitor mode, the SMC instruction, and all the features described by Sec_frac == `0b0001` implemented.

`0b0010` As for `0b0001`, and adds the ability to set the NSACR.RFR bit. Not permitted in Armv8 as the NSACR.RFR bit is RES0.

All other values are reserved.

In Armv8-A the permitted values are:

- `0b0000` when EL3 is not implemented.
- `0b0001` when EL3 is implemented.

In an implementation that includes EL3, if EL3 cannot use AArch32 but EL1 can use AArch32 then this field has the value `0b0001`.

ProgMod, bits [3:0]

Support for the standard programmers' model for ARMv4 and later. Model must support User, FIQ, IRQ, Supervisor, Abort, Undefined, and System modes. Defined values are:

`0b0000` Not supported.

`0b0001` Supported.

All other values are reserved.

In Armv8-A the only permitted value is `0b0001`.

Accessing the ID_PFR1

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        return ID_PFR1;
elsif PSTATE.EL == EL2 then
    return ID_PFR1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    return ID_PFR1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.100 ID_PFR2, Processor Feature Register 2

The ID_PFR2 characteristics are:

Purpose

Gives information about the AArch32 programmers' model.

Must be interpreted with ID_PFR0 and ID_PFR1.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page G8-6106.

Configurations

AArch32 System register ID_PFR2[31:0] is architecturally mapped to AArch64 System register ID_PFR2_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ID_PFR2 are UNKNOWN.

Attributes

ID_PFR2 is a 32-bit register.

Field descriptions

The ID_PFR2 bit assignments are:

31 … 12	11 … 8	7 … 4	3 … 0
RES0	RAS_frac	SSBS	CSV3

Bits [31:12]

Reserved, RES0.

RAS_frac, bits [11:8]

From ARMv8.4:

RAS Extension fractional field.

`0b0000` If ID_PFR0.RAS == `0b0001`, RAS Extension implemented.

`0b0001` If ID_PFR0.RAS == `0b0001`, as `0b0000` and adds support for additional ERXMISC<m> System registers.

Error records accessed through System registers conform to RAS System Architecture v1.1, which includes simplifications to ERR<n>STATUS and support for the optional RAS Timestamp Extension.

All other values are reserved.

This field is valid only if ID_PFR0.RAS == `0b0001`.

Otherwise:

Reserved, RES0.

SSBS, bits [7:4]

From ARMv8.5:

Speculative Store Bypassing controls in AArch64 state. Defined values are:

`0b0000` AArch32 provides no mechanism to control the use of Speculative Store Bypassing.

`0b0001` AArch32 provides the PSTATE.SSBS mechanism to mark regions that are Speculative Store Bypass Safe.

From Armv8.0, the permitted values are `0b0000` and `0b0001`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

From Armv8.5, the only permitted value is `0b0001`.

All other values are reserved.

Otherwise:

Reserved, RES0.

CSV3, bits [3:0]

From ARMv8.5:

Speculative use of faulting data. Defined values are:

`0b0000` This Device does not disclose whether data loaded under speculation with a permission or domain fault can be used to form an address or generate condition codes or SVE predicate values to be used by instructions newer than the load in the speculative sequence

`0b0001` Data loaded under speculation with a permission or domain fault cannot be used to form an address or generate condition codes or SVE predicate values to be used by instructions newer than the load in the speculative sequence

From Armv8.0, the permitted values are `0b0000` and `0b0001`.

From Armv8.5, the only permitted value is `0b0001`.

All other values are reserved.

Otherwise:

Reserved, RES0.

Accessing the ID_PFR2

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0011	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        return ID_PFR2;
elsif PSTATE.EL == EL2 then
    return ID_PFR2;
elsif PSTATE.EL == EL3 then
    return ID_PFR2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.101 IFAR, Instruction Fault Address Register

The IFAR characteristics are:

Purpose

Holds the virtual address of the faulting address that caused a synchronous Prefetch Abort exception.

Configurations

AArch32 System register IFAR[31:0] is architecturally mapped to AArch64 System register FAR_EL1[63:32].

AArch32 System register IFAR[31:0](S) is architecturally mapped to AArch32 System register HIFAR[31:0] when EL2 is implemented, EL3 is implemented and the highest implemented Exception level is using AArch32 state.

AArch32 System register IFAR[31:0](S) is architecturally mapped to AArch64 System register FAR_EL2[63:32] when EL2 is implemented.

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to IFAR are UNKNOWN.

Attributes

IFAR is a 32-bit register.

Field descriptions

The IFAR bit assignments are:

31 … 0
VA of faulting address of synchronous Prefetch Abort exception

Bits [31:0]

VA of faulting address of synchronous Prefetch Abort exception.

This field resets to an architecturally UNKNOWN value.

Accessing the IFAR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0110	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T6 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T6 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TRVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        return IFAR_NS;
    else
        return IFAR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return IFAR_NS;
    else
        return IFAR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return IFAR_S;
    else
        return IFAR_NS;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0110	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T6 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T6 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        IFAR_NS = R[t];
    else
        IFAR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        IFAR_NS = R[t];
    else
        IFAR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        IFAR_S = R[t];
    else
        IFAR_NS = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.102 IFSR, Instruction Fault Status Register

The IFSR characteristics are:

Purpose

Holds status information about the last instruction fault.

Configurations

AArch32 System register IFSR[31:0] is architecturally mapped to AArch64 System register IFSR32_EL2[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to IFSR are UNKNOWN.

The current translation table format determines which format of the register is used.

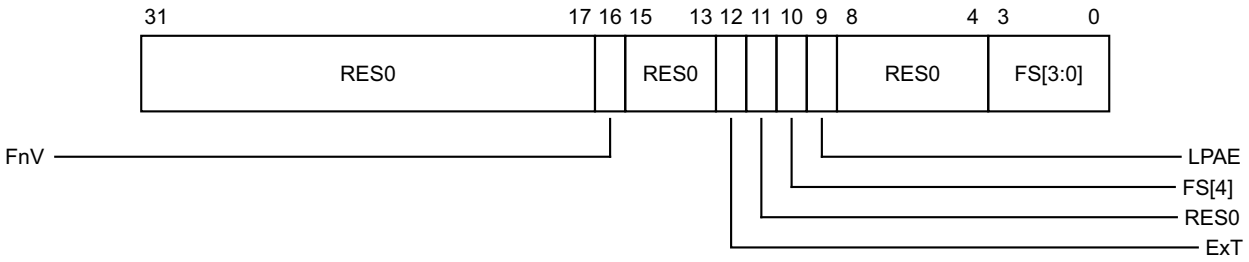
Attributes

IFSR is a 32-bit register.

Field descriptions

The IFSR bit assignments are:

When TTBCR.EAE == 0:



Bits [31:17]

Reserved, RES0.

FnV, bit [16]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

`0b0` IFAR is valid.

`0b1` IFAR is not valid, and holds an UNKNOWN value.

This field is only valid for a synchronous External abort other than a synchronous External abort on a translation table walk. It is RES0 for all other Prefetch Abort exceptions.

This field resets to an architecturally UNKNOWN value.

Bits [15:13]

Reserved, RES0.

ExT, bit [12]

External abort type. This bit can be used to provide an IMPLEMENTATION DEFINED classification of External aborts.

In an implementation that does not provide any classification of External aborts, this bit is RES0.

For aborts other than External aborts this bit always returns 0.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bit [11]

Reserved, RES0.

FS[4], bit [10]

This field is bit[4] of FS[4:0].

Fault Status bits. Bits [10] and [3:0] are interpreted together.

`0b00001` PC alignment fault.

`0b00010` Debug exception.

`0b00011` Access flag fault, level 1.

`0b00101` Translation fault, level 1.

`0b00110` Access flag fault, level 2.

`0b00111` Translation fault, level 2.

`0b01000` Synchronous External abort, not on translation table walk.

`0b01001` Domain fault, level 1.

`0b01011` Domain fault, level 2.

`0b01100` Synchronous External abort, on translation table walk, level 1.

`0b01101` Permission fault, level 1.

`0b01110` Synchronous External abort, on translation table walk, level 2.

`0b01111` Permission fault, level 2.

`0b10000` TLB conflict abort.

`0b10100` IMPLEMENTATION DEFINED fault (Lockdown fault).

`0b11001` **When RAS is not implemented** Synchronous parity or ECC error on memory access, not on translation table walk.

`0b11100` **When RAS is not implemented** Synchronous parity or ECC error on translation table walk, level 1.

`0b11110` **When RAS is not implemented** Synchronous parity or ECC error on translation table walk, level 2.

All other values are reserved.

For more information about the lookup level associated with a fault, see *The level associated with MMU faults on a Short-descriptor translation table lookup* on page G5-6030.

The FS field is split as follows:

- FS[4] is IFSR[10].
- FS[3:0] is IFSR[3:0].

This field resets to an architecturally UNKNOWN value.

LPAE, bit [9]

On taking a Data Abort exception, this bit is set as follows:

`0b0` Using the Short-descriptor translation table formats.

`0b1` Using the Long-descriptor translation table formats.

Hardware does not interpret this bit to determine the behavior of the memory system, and therefore software can set this bit to 0 or 1 without affecting operation.

This field resets to an architecturally UNKNOWN value.

Bits [8:4]

Reserved, RES0.

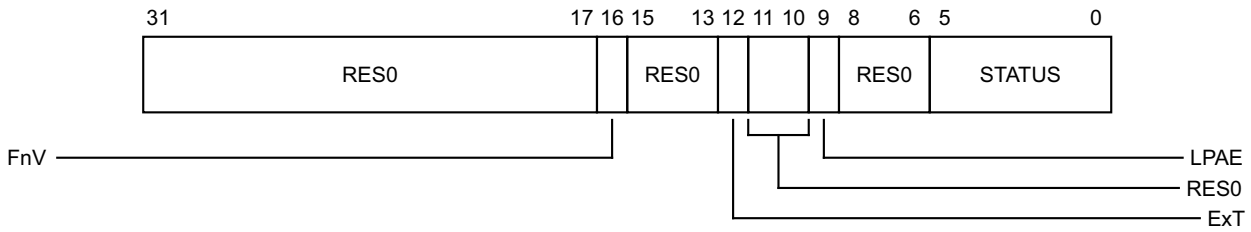
FS[3:0], bits [3:0]

This field is bits[3:0] of FS[4:0].

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

See FS[4] for the field description.

When TTBCR.EAE == 1:



Bits [31:17]

Reserved, RES0.

FnV, bit [16]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

`0b0` IFAR is valid.

`0b1` IFAR is not valid, and holds an UNKNOWN value.

This field is only valid for a synchronous External abort other than a synchronous External abort on a translation table walk. It is RES0 for all other Prefetch Abort exceptions.

This field resets to an architecturally UNKNOWN value.

Bits [15:13]

Reserved, RES0.

ExT, bit [12]

External abort type. This bit can be used to provide an IMPLEMENTATION DEFINED classification of External aborts.

In an implementation that does not provide any classification of External aborts, this bit is RES0.

For aborts other than External aborts this bit always returns 0.

This field resets to an architecturally UNKNOWN value.

Bits [11:10]

Reserved, RES0.

LPAE, bit [9]

On taking a Data Abort exception, this bit is set as follows:

`0b0` Using the Short-descriptor translation table formats.

`0b1` Using the Long-descriptor translation table formats.

Hardware does not interpret this bit to determine the behavior of the memory system, and therefore software can set this bit to 0 or 1 without affecting operation.

This field resets to an architecturally UNKNOWN value.

Bits [8:6]

Reserved, RES0.

STATUS, bits [5:0]

Fault status bits. Possible values of this field are:

`0b000000` Address size fault in translation table base register.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b000001` Address size fault, level 1.

`0b000010` Address size fault, level 2.

`0b000011` Address size fault, level 3.

`0b000101` Translation fault, level 1.

`0b000110` Translation fault, level 2.

`0b000111` Translation fault, level 3.

`0b001001` Access flag fault, level 1.

`0b001010` Access flag fault, level 2.

`0b001011` Access flag fault, level 3.

`0b001101` Permission fault, level 1.

`0b001110` Permission fault, level 2.

`0b001111` Permission fault, level 3.

`0b010000` Synchronous External abort, not on translation table walk.

`0b010101` Synchronous External abort, on translation table walk, level 1.

`0b010110` Synchronous External abort, on translation table walk, level 2.

`0b010111` Synchronous External abort, on translation table walk, level 3.

`0b011000` **When RAS is not implemented** Synchronous parity or ECC error on memory access, not on translation table walk.

`0b011101` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk, level 1.

`0b011110` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk, level 2.

`0b011111` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk, level 3.

`0b100001` PC alignment fault.

`0b100010` Debug exception.

`0b110000` TLB conflict abort.

All other values are reserved.

When the RAS Extension is implemented, `0b011000`, `0b011101`, `0b011110`, and `0b011111`, are reserved.

For more information about the lookup level associated with a fault, see *The level associated with MMU faults on a Long-descriptor translation table lookup* on page G5-6032.

This field resets to an architecturally UNKNOWN value.

Accessing the IFSR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TRVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return IFSR_NS;
    else
        return IFSR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return IFSR_NS;
    else
        return IFSR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return IFSR_S;
    else
        return IFSR_NS;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        IFSR_NS = R[t];
    else
        IFSR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        IFSR_NS = R[t];
    else
        IFSR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        IFSR_S = R[t];
    else
        IFSR_NS = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.103 ISR, Interrupt Status Register

The ISR characteristics are:

Purpose

Shows the pending status of the IRQ, FIQ, or SError.

When executing at EL2, EL3, or Secure EL1, when SCR_EL3.EEL2 == `0b0`, this shows the pending status of the physical interrupts.

When executing at Non-secure EL1, or at Secure EL1, when SCR_EL3.EEL2 == `0b01`:

- If the HCR.{IMO,FMO,AMO} bit has a value of 1, the corresponding ISR.{I,F,A} bit shows the pending status of the virtual IRQ, FIQ, or SError.
- If the HCR.{IMO,FMO,AMO} bit has a value of 0, the corresponding ISR.{I,F,A} bit shows the pending status of the physical IRQ, FIQ, or SError.

Configurations

AArch32 System register ISR[31:0] is architecturally mapped to AArch64 System register ISR_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ISR are UNKNOWN.

Attributes

ISR is a 32-bit register.

Field descriptions

The ISR bit assignments are:

31 … 9	8	7	6	5 … 0
RES0	A	I	F	RES0

Bits [31:9]

Reserved, RES0.

A, bit [8]

SError interrupt pending bit:

`0b0` No pending SError interrupt.

`0b1` An SError interrupt is pending.

If the SError interrupt is edge-triggered, this field is cleared to zero when the physical SError interrupt is taken.

I, bit [7]

IRQ pending bit. Indicates whether an IRQ interrupt is pending:

`0b0` No pending IRQ.

`0b1` An IRQ interrupt is pending.

F, bit [6]

FIQ pending bit. Indicates whether an FIQ interrupt is pending.

`0b0` No pending FIQ.

`0b1` An FIQ interrupt is pending.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bits [5:0]

Reserved, RES0.

Accessing the ISR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1100	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T12 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T12 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        return ISR;
elsif PSTATE.EL == EL2 then
    return ISR;
elsif PSTATE.EL == EL3 then
    return ISR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.104 ITLBIALL, Instruction TLB Invalidate All

The ITLBIALL characteristics are:

Purpose

Invalidate all cached copies of translation table entries from instruction TLBs that are from any level of the translation table walk. The entries that are invalidated are as follows:

- If executed at EL1, all entries that:
  - Would be required for the EL1&0 translation regime.
  - Match the current VMID, if EL2 is implemented and enabled in the current Security state.
- If executed in Secure state when EL3 is using AArch32, all entries that would be required for the Secure PL1&0 translation regime.
- If executed at EL2, and if EL2 is enabled in the current Security state, the stage 1 or stage 2 translation table entries that would be required for the Non-secure PL1&0 translation regime and matches the current VMID.

The invalidation only applies to the PE that executes this System instruction.

Arm deprecates the use of this System instruction. It is only provided for backwards compatibility with earlier versions of the Arm architecture.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ITLBIALL are UNKNOWN.

Attributes

ITLBIALL is a 32-bit System instruction.

Field descriptions

ITLBIALL ignores the value in the register specified by the instruction. Software does not have to write a value to the register before issuing this instruction.

Executing the ITLBIALL instruction

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0101	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TTLB == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        ITLBIALL();
elsif PSTATE.EL == EL2 then
    ITLBIALL();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    ITLBIALL();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.105 ITLBIASID, Instruction TLB Invalidate by ASID match

The ITLBIASID characteristics are:

Purpose

Invalidate all cached copies of translation table entries from instruction TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used for the specified ASID, and either:
  - — Is from a level of lookup above the final level.
  - — Is a non-global entry from the final level of lookup.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Arm deprecates the use of this System instruction. It is only provided for backwards compatibility with earlier versions of the Arm architecture.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ITLBIASID are UNKNOWN.

Attributes

ITLBIASID is a 32-bit System instruction.

Field descriptions

The ITLBIASID input value bit assignments are:

31 … 8	7 … 0
RES0	ASID

Bits [31:8]

Reserved, RES0.

ASID, bits [7:0]

ASID value to match. Any TLB entries for non-global pages that match the ASID values will be affected by this System instruction.

Executing the ITLBIASID instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0101	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TTLB == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        ITLBIASID(R[t]);
elsif PSTATE.EL == EL2 then
    ITLBIASID(R[t]);
elsif PSTATE.EL == EL3 then
    ITLBIASID(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.106 ITLBIMVA, Instruction TLB Invalidate by VA

The ITLBIMVA characteristics are:

Purpose

Invalidate all cached copies of translation table entries from instruction TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified address, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Arm deprecates the use of this System instruction. It is only provided for backwards compatibility with earlier versions of the Arm architecture.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to ITLBIMVA are UNKNOWN.

Attributes

ITLBIMVA is a 32-bit System instruction.

Field descriptions

The ITLBIMVA input value bit assignments are:

31 … 12	11 … 8	7 … 0
VA	RES0	ASID

VA, bits [31:12]

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Bits [11:8]

Reserved, RES0.

ASID, bits [7:0]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Global TLB entries that match the VA value will be affected by this operation, regardless of the value of the ASID field.

Executing the ITLBIMVA instruction

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0101	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TTLB == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        ITLBIASID(R[t]);
elsif PSTATE.EL == EL2 then
    ITLBIASID(R[t]);
elsif PSTATE.EL == EL3 then
    ITLBIASID(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.107 JIDR, Jazelle ID Register

The JIDR characteristics are:

Purpose

A Jazelle register, which identified the Jazelle architecture version.

Configurations

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to JIDR are UNKNOWN.

Attributes

JIDR is a 32-bit register.

Field descriptions

The JIDR bit assignments are:

31	0
RAZ	

Bits [31:0]

Reserved, RAZ.

Accessing the JIDR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b111	0b0000	0b0000	0b000

```
if PSTATE.EL == EL0 then
    if boolean IMPLEMENTATION_DEFINED "JIDR UNDEFINED at EL0" then
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TID0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID0 == '1' then
        AArch32.TakeHypTrapException(0x05);
    else
        return JIDR;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID0 == '1' then
        AArch32.TakeHypTrapException(0x05);
    else
        return JIDR;
elsif PSTATE.EL == EL2 then
    return JIDR;
elsif PSTATE.EL == EL3 then
    return JIDR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.108 JMCR, Jazelle Main Configuration Register

The JMCR characteristics are:

Purpose

A Jazelle register, which provides control of the Jazelle extension.

Configurations

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to JMCR are UNKNOWN.

Attributes

JMCR is a 32-bit register.

Field descriptions

The JMCR bit assignments are:

31 … 0
RAZ/WI

Bits [31:0]

Reserved, RAZ/WI.

Accessing the JMCR

For accesses from EL0 it is IMPLEMENTATION DEFINED whether the register is RW or UNDEFINED.

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b111	0b0010	0b0000	0b000

```
if PSTATE.EL == EL0 then
    if boolean IMPLEMENTATION_DEFINED "JMCR UNDEFINED at EL0" then
        UNDEFINED;
    else
        return JMCR;
elsif PSTATE.EL == EL1 then
    return JMCR;
elsif PSTATE.EL == EL2 then
    return JMCR;
elsif PSTATE.EL == EL3 then
    return JMCR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b111	0b0010	0b0000	0b000

```
if PSTATE.EL == EL0 then
    if boolean IMPLEMENTATION_DEFINED "JMCR UNDEFINED at EL0" then
        UNDEFINED;
    else
        //no operation
elsif PSTATE.EL == EL1 then
    //no operation
elsif PSTATE.EL == EL2 then
    //no operation
elsif PSTATE.EL == EL3 then
    //no operation
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.109 JOSCR, Jazelle OS Control Register

The JOSCR characteristics are:

Purpose

A Jazelle register, which provides operating system control of the Jazelle Extension.

Configurations

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to JOSCR are UNKNOWN.

Attributes

JOSCR is a 32-bit register.

Field descriptions

The JOSCR bit assignments are:

31	0
RAZ/WI	

Bits [31:0]

Reserved, RAZ/WI.

Accessing the JOSCR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b111	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    if boolean IMPLEMENTATION_DEFINED "JOSCR UNDEFINED at EL0" then
        UNDEFINED;
    else
        return JOSCR;
elsif PSTATE.EL == EL1 then
    return JOSCR;
elsif PSTATE.EL == EL2 then
    return JOSCR;
elsif PSTATE.EL == EL3 then
    return JOSCR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b111	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    if boolean IMPLEMENTATION_DEFINED "JOSCR UNDEFINED at EL0" then
        UNDEFINED;
    else
        //no operation
elsif PSTATE.EL == EL1 then
    //no operation
elsif PSTATE.EL == EL2 then
    //no operation
elsif PSTATE.EL == EL3 then
    //no operation
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.110 MAIR0, Memory Attribute Indirection Register 0

The MAIR0 characteristics are:

Purpose

Along with MAIR1, provides the memory attribute encodings corresponding to the possible AttrIndx values in a Long-descriptor format translation table entry for stage 1 translations.

AttrIndx[2] indicates the MAIR register to be used:

- When AttrIndx[2] is 0, MAIR0 is used.
- When AttrIndx[2] is 1, MAIR1 is used.

Configurations

AArch32 System register MAIR0[31:0] is architecturally mapped to AArch64 System register MAIR_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to MAIR0 are UNKNOWN.

MAIR0 and PRRR are the same register, with a different view depending on the value of TTBCR.EAE:

- When it is set to 0, the register is as described in PRRR.
- When it is set to 1, the register is as described in MAIR0.

When EL3 is using AArch32, write access to MAIR0(S) is disabled when the CP15SDISABLE signal is asserted HIGH.

Attributes

MAIR0 is a 32-bit register.

Field descriptions

The MAIR0 bit assignments are:

When TTBCR.EAE == 1:

31 – 24	23 – 16	15 – 8	7 – 0
Attr3	Attr2	Attr1	Attr0

Attr<n>, bits [8n+7:8n], for n = 0 to 3

The memory attribute encoding for an AttrIndx[2:0] entry in a Long descriptor format translation table entry, where:

- AttrIndx[2:0] gives the value of <n> in Attr<n>.
- AttrIndx[2] defines which MAIR to access. Attr7 to Attr4 are in MAIR1, and Attr3 to Attr0 are in MAIR0.

Bits [7:4] are encoded as follows:

Attr<n>[7:4]	Meaning
`0b0000`	Device memory. See encoding of Attr<n>[3:0] for the type of Device memory.
0b00RW, RW not `0b00`	Normal memory, Outer Write-Through Transient.
`0b0100`	Normal memory, Outer Non-cacheable.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Attr<n>[7:4]	Meaning
0b01RW, RW not `0b00`	Normal memory, Outer Write-Back Transient.
0b10RW	Normal memory, Outer Write-Through Non-transient.
0b11RW	Normal memory, Outer Write-Back Non-transient.

R = Outer Read-Allocate policy, W = Outer Write-Allocate policy.

The meaning of bits [3:0] depends on the value of bits [7:4]:

Attr<n>[3:0]	Meaning when Attr<n>[7:4] is `0b0000`	Meaning when Attr<n>[7:4] is not `0b0000`
`0b0000`	Device-nGnRnE memory	UNPREDICTABLE
0b00RW, RW not `0b00`	UNPREDICTABLE	Normal memory, Inner Write-Through Transient
`0b0100`	Device-nGnRE memory	Normal memory, Inner Non-cacheable
0b01RW, RW not `0b00`	UNPREDICTABLE	Normal memory, Inner Write-Back Transient
`0b1000`	Device-nGRE memory	Normal memory, Inner Write-Through Non-transient (RW=`0b00`)
0b10RW, RW not `0b00`	UNPREDICTABLE	Normal memory, Inner Write-Through Non-transient
`0b1100`	Device-GRE memory	Normal memory, Inner Write-Back Non-transient (RW=`0b00`)
0b11RW, RW not `0b00`	UNPREDICTABLE	Normal memory, Inner Write-Back Non-transient

R = Inner Read-Allocate policy, W = Inner Write-Allocate policy.

The R and W bits in some Attr<n> fields have the following meanings:

R or W	Meaning
`0b0`	No Allocate
`0b1`	Allocate

This field resets to an architecturally UNKNOWN value.

Accessing the MAIR0

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1010	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T10 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T10 == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TRVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return MAIR0_NS;
    else
        return MAIR0;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return MAIR0_NS;
    else
        return MAIR0;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return MAIR0_S;
    else
        return MAIR0_NS;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1010	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T10 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T10 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        MAIR0_NS = R[t];
    else
        MAIR0 = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        MAIR0_NS = R[t];
    else
        MAIR0 = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' && CP15SDISABLE == HIGH then
        UNDEFINED;
    elsif SCR.NS == '0' && CP15SDISABLE2 == HIGH then
        UNDEFINED;
    else
        if SCR.NS == '0' then
            MAIR0_S = R[t];
        else
            MAIR0_NS = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.111 MAIR1, Memory Attribute Indirection Register 1

The MAIR1 characteristics are:

Purpose

Along with MAIR0, provides the memory attribute encodings corresponding to the possible AttrIndx values in a Long-descriptor format translation table entry for stage 1 translations.

AttrIndx[2] indicates the MAIR register to be used:

- When AttrIndx[2] is 0, MAIR0 is used.
- When AttrIndx[2] is 1, MAIR1 is used.

Configurations

AArch32 System register MAIR1[31:0] is architecturally mapped to AArch64 System register MAIR_EL1[63:32].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to MAIR1 are UNKNOWN.

MAIR1 and NMRR are the same register, with a different view depending on the value of TTBCR.EAE:

- When it is set to 0, the register is as described in NMRR.
- When it is set to 1, the register is as described in MAIR1.

When EL3 is using AArch32, write access to MAIR1(S) is disabled when the CP15SDISABLE signal is asserted HIGH.

Attributes

MAIR1 is a 32-bit register.

Field descriptions

The MAIR1 bit assignments are:

When TTBCR.EAE == 1:

31 – 24	23 – 16	15 – 8	7 – 0
Attr7	Attr6	Attr5	Attr4

Attr<n>, bits [8(n-4)+7:8(n-4)], for n = 4 to 7

The memory attribute encoding for an AttrIndx[2:0] entry in a Long descriptor format translation table entry, where:

- AttrIndx[2:0] gives the value of <n> in Attr<n>.
- AttrIndx[2] defines which MAIR to access. Attr7 to Attr4 are in MAIR1, and Attr3 to Attr0 are in MAIR0.

Bits [7:4] are encoded as follows:

Attr<n>[7:4]	Meaning
`0b0000`	Device memory. See encoding of Attr<n>[3:0] for the type of Device memory.
0b00RW, RW not `0b00`	Normal memory, Outer Write-Through Transient.
`0b0100`	Normal memory, Outer Non-cacheable.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Attr<n>[7:4]	Meaning
0b01RW, RW not 0b00	Normal memory, Outer Write-Back Transient.
0b10RW	Normal memory, Outer Write-Through Non-transient.
0b11RW	Normal memory, Outer Write-Back Non-transient.

R = Outer Read-Allocate policy, W = Outer Write-Allocate policy.

The meaning of bits [3:0] depends on the value of bits [7:4]:

Attr<n>[3:0]	Meaning when Attr<n>[7:4] is 0b0000	Meaning when Attr<n>[7:4] is not 0b0000
0b0000	Device-nGnRnE memory	UNPREDICTABLE
0b00RW, RW not 0b00	UNPREDICTABLE	Normal memory, Inner Write-Through Transient
0b0100	Device-nGnRE memory	Normal memory, Inner Non-cacheable
0b01RW, RW not 0b00	UNPREDICTABLE	Normal memory, Inner Write-Back Transient
0b1000	Device-nGRE memory	Normal memory, Inner Write-Through Non-transient (RW=0b00)
0b10RW, RW not 0b00	UNPREDICTABLE	Normal memory, Inner Write-Through Non-transient
0b1100	Device-GRE memory	Normal memory, Inner Write-Back Non-transient (RW=0b00)
0b11RW, RW not 0b00	UNPREDICTABLE	Normal memory, Inner Write-Back Non-transient

R = Inner Read-Allocate policy, W = Inner Write-Allocate policy.

The R and W bits in some Attr<n> fields have the following meanings:

R or W	Meaning
0b0	No Allocate
0b1	Allocate

This field resets to an architecturally UNKNOWN value.

Accessing the MAIR1

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1010	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T10 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T10 == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TRVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return MAIR1_NS;
    else
        return MAIR1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return MAIR1_NS;
    else
        return MAIR1;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return MAIR1_S;
    else
        return MAIR1_NS;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1010	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T10 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T10 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        MAIR1_NS = R[t];
    else
        MAIR1 = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        MAIR1_NS = R[t];
    else
        MAIR1 = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' && CP15SDISABLE == HIGH then
        UNDEFINED;
    elsif SCR.NS == '0' && CP15SDISABLE2 == HIGH then
        UNDEFINED;
    else
        if SCR.NS == '0' then
            MAIR1_S = R[t];
        else
            MAIR1_NS = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.112 MIDR, Main ID Register

The MIDR characteristics are:

Purpose

Provides identification information for the PE, including an implementer code for the device and a device ID number.

Configurations

AArch32 System register MIDR[31:0] is architecturally mapped to AArch64 System register MIDR_EL1[31:0].

AArch32 System register MIDR[31:0] is architecturally mapped to External register MIDR_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to MIDR are UNKNOWN.

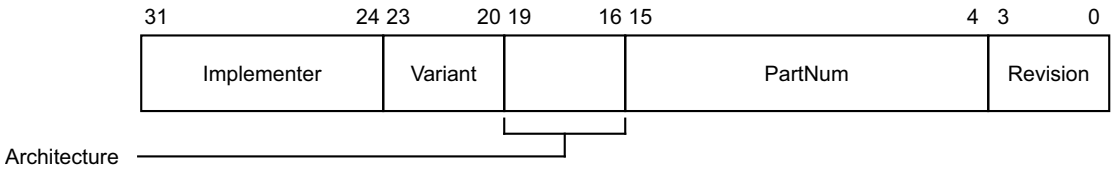
Some fields of the MIDR are IMPLEMENTATION DEFINED. For details of the values of these fields for a particular Armv8 implementation, and any implementation-specific significance of these values, see the product documentation.

Attributes

MIDR is a 32-bit register.

Field descriptions

The MIDR bit assignments are:



Implementer, bits [31:24]

The Implementer code. This field must hold an implementer code that has been assigned by Arm. Assigned codes include the following:

Hex representation	Implementer
`0x00`	Reserved for software use
`0xC0`	Ampere Computing
`0x41`	Arm Limited
`0x42`	Broadcom Corporation
`0x43`	Cavium Inc.
`0x44`	Digital Equipment Corporation
`0x46`	Fujitsu Ltd.
`0x49`	Infineon Technologies AG
`0x4D`	Motorola or Freescale Semiconductor Inc.
`0x4E`	NVIDIA Corporation

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Hex representation	Implementer
0x50	Applied Micro Circuits Corporation
0x51	Qualcomm Inc.
0x56	Marvell International Ltd.
0x69	Intel Corporation

Arm can assign codes that are not published in this manual. All values not assigned by Arm are reserved and must not be used.

Variant, bits [23:20]

An IMPLEMENTATION DEFINED variant number. Typically, this field is used to distinguish between different product variants, or major revisions of a product.

Architecture, bits [19:16]

The permitted values of this field are:

0b0001 Armv4.

0b0010 Armv4T.

0b0011 Armv5 (obsolete).

0b0100 Armv5T.

0b0101 Armv5TE.

0b0110 Armv5TEJ.

0b0111 Armv6.

0b1111 Architectural features are individually identified in the ID_* registers, see *ID registers* on page K14-8085.

All other values are reserved.

PartNum, bits [15:4]

An IMPLEMENTATION DEFINED primary part number for the device.

On processors implemented by Arm, if the top four bits of the primary part number are 0x0 or 0x7, the variant and architecture are encoded differently.

Revision, bits [3:0]

An IMPLEMENTATION DEFINED revision number for the device.

Accessing the MIDR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && !ELUsingAArch32(EL2) then
        return VPIDR_EL2<31:0>;
    elsif EL2Enabled() && ELUsingAArch32(EL2) then
        return VPIDR;
    else
        return MIDR;
elsif PSTATE.EL == EL2 then
    return MIDR;
elsif PSTATE.EL == EL3 then
    return MIDR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.113 MPIDR, Multiprocessor Affinity Register

The MPIDR characteristics are:

Purpose

In a multiprocessor system, provides an additional PE identification mechanism for scheduling purposes.

Configurations

AArch32 System register MPIDR[31:0] is architecturally mapped to AArch64 System register MPIDR_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to MPIDR are UNKNOWN.

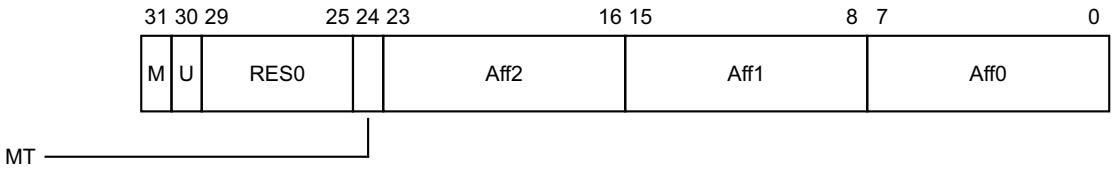
In a uniprocessor system Arm recommends that each Aff<n> field of this register returns a value of 0.

Attributes

MPIDR is a 32-bit register.

Field descriptions

The MPIDR bit assignments are:



M, bit [31]

Indicates whether this implementation includes the functionality introduced by the ARMv7 Multiprocessing Extensions. The possible values of this bit are:

`0b0` This implementation does not include the ARMv7 Multiprocessing Extensions functionality.

`0b1` This implementation includes the ARMv7 Multiprocessing Extensions functionality.

From Armv8, this bit is RAO.

U, bit [30]

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system. The possible values of this bit are:

`0b0` Processor is part of a multiprocessor system.

`0b1` Processor is part of a uniprocessor system.

Bits [29:25]

Reserved, RES0.

MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using a multithreading type approach. See the description of Aff0 for more information about affinity levels. The possible values of this bit are:

`0b0` Performance of PEs at the lowest affinity level, or PEs with MPIDR.MT set to 1, different affinity level 0 values, and the same values for affinity level 1 and higher, is largely independent.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` Performance of PEs at the lowest affinity level, or PEs with MPIDR.MT set to 1, different affinity level 0 values, and the same values for affinity level 1 and higher, is very interdependent.

Aff2, bits [23:16]

Affinity level 2. See the description of Aff0 for more information.

Aff1, bits [15:8]

Affinity level 1. See the description of Aff0 for more information.

Aff0, bits [7:0]

Affinity level 0. This is the affinity level that is most significant for determining PE behavior. Higher affinity levels are increasingly less significant in determining PE behavior. The assigned value of the MPIDR.{Aff2, Aff1, Aff0} or MPIDR_EL1.{Aff3, Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

Accessing the MPIDR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0000	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) then
        return VMPIDR_EL2<31:0>;
    elsif EL2Enabled() && ELUsingAArch32(EL2) then
        return VMPIDR;
    else
        return MPIDR;
elsif PSTATE.EL == EL2 then
    return MPIDR;
elsif PSTATE.EL == EL3 then
    return MPIDR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.114 MVBAR, Monitor Vector Base Address Register

The MVBAR characteristics are:

Purpose

When EL3 is implemented and can use AArch32, holds the vector base address for any exception that is taken to Monitor mode.

Secure software must program the MVBAR with the required initial value as part of the PE boot sequence.

Configurations

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to MVBAR are UNKNOWN.

It is IMPLEMENTATION DEFINED whether MVBAR[0] has a fixed value and ignored writes, or takes the last value written to it.

On a reset into EL3 using AArch32, the reset value of MVBAR is an IMPLEMENTATION DEFINED choice between the following:

- MVBAR[31:5] = an IMPLEMENTATION DEFINED value, which might be UNKNOWN, MVBAR[4:1] = RES0, and MVBAR[0] = 0.
- MVBAR[31:1] = an IMPLEMENTATION DEFINED value that is bits[31:1] of the AArch32 reset address, and MVBAR[0] = 1.

Attributes

MVBAR is a 32-bit register.

Field descriptions

The MVBAR bit assignments are:

When programmed with a vector base address:

31 … 5	4 … 0
Vector Base Address	Reserved

Bits [31:5]

Vector Base Address. Bits[31:5] of the base address of the exception vectors for exceptions taken to this Exception level. Bits[4:0] of an exception vector are the exception offset.

Reserved, bits [4:0]

Reserved, see Configurations.

Accessing the MVBAR

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1100	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsHighestEL(EL1) then
        return RVBAR;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T12 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T12 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif !ELUsingAArch32(EL2) && SCR_EL3.<NS,EEL2> == '01' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if IsHighestEL(EL2) then
        return RVBAR;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    return MVBAR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1100	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T12 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T12 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif !ELUsingAArch32(EL2) && SCR_EL3.<NS,EEL2> == '01' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' && CP15SDISABLE == HIGH then
        UNDEFINED;
    elsif SCR.NS == '0' && CP15SDISABLE2 == HIGH then
        UNDEFINED;
    else
        MVBAR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.115 MVFR0, Media and VFP Feature Register 0

The MVFR0 characteristics are:

Purpose

Describes the features provided by the AArch32 Advanced SIMD and Floating-point implementation.

Must be interpreted with MVFR1 and MVFR2.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page G8-6106.

Configurations

AArch32 System register MVFR0[31:0] is architecturally mapped to AArch64 System register MVFR0_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to MVFR0 are UNKNOWN.

Implemented only if the implementation includes Advanced SIMD and floating-point instructions.

Attributes

MVFR0 is a 32-bit register.

Field descriptions

The MVFR0 bit assignments are:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
FPRound	FPShVec	FPSqrt	FPDivide	FPTrap	FPDP	FPSP	SIMDReg

FPRound, bits [31:28]

Floating-Point Rounding modes. Indicates whether the floating-point implementation provides support for rounding modes. Defined values are:

`0b0000` Not implemented, or only Round to Nearest mode supported, except that Round towards Zero mode is supported for VCVT instructions that always use that rounding mode regardless of the FPSCR setting.

`0b0001` All rounding modes supported.

All other values are reserved.

In Armv8-A the permitted values are `0b0000` and `0b0001`.

FPShVec, bits [27:24]

Short Vectors. Indicates whether the floating-point implementation provides support for the use of short vectors. Defined values are:

`0b0000` Short vectors not supported.

`0b0001` Short vector operation supported.

All other values are reserved.

In Armv8-A the only permitted value is `0b0000`.

FPSqrt, bits [23:20]

Square Root. Indicates whether the floating-point implementation provides support for the ARMv6 VFP square root operations. Defined values are:

`0b0000` Not supported in hardware.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b0001` Supported.

All other values are reserved.

In Armv8-A the permitted values are `0b0000` and `0b0001`.

The VSQRT.F32 instruction also requires the single-precision floating-point attribute, bits [7:4], and the VSQRT.F64 instruction also requires the double-precision floating-point attribute, bits [11:8].

FPDivide, bits [19:16]

Indicates whether the floating-point implementation provides support for VFP divide operations. Defined values are:

`0b0000` Not supported in hardware.

`0b0001` Supported.

All other values are reserved.

In Armv8-A the permitted values are `0b0000` and `0b0001`.

The VDIV.F32 instruction also requires the single-precision floating-point attribute, bits [7:4], and the VDIV.F64 instruction also requires the double-precision floating-point attribute, bits [11:8].

FPTrap, bits [15:12]

Floating Point Exception Trapping. Indicates whether the floating-point implementation provides support for exception trapping. Defined values are:

`0b0000` Not supported.

`0b0001` Supported.

All other values are reserved.

A value of `0b0001` indicates that, when the corresponding trap is enabled, a floating-point exception generates an exception.

FPDP, bits [11:8]

Double Precision. Indicates whether the floating-point implementation provides support for double-precision operations. Defined values are:

`0b0000` Not supported in hardware.

`0b0001` Supported, VFPv2.

`0b0010` Supported, VFPv3, VFPv4, or Armv8. VFPv3 and Armv8 add an instruction to load a double-precision floating-point constant, and conversions between double-precision and fixed-point values.

All other values are reserved.

In Armv8-A the permitted values are `0b0000` and `0b0010`.

A value of `0b0001` or `0b0010` indicates support for all VFP double-precision instructions in the supported version of VFP, except that, in addition to this field being nonzero:

- VSQRT.F64 is only available if the Square root field is `0b0001`.
- VDIV.F64 is only available if the Divide field is `0b0001`.
- Conversion between double-precision and single-precision is only available if the single-precision field is nonzero.

FPSP, bits [7:4]

Single Precision. Indicates whether the floating-point implementation provides support for single-precision operations. Defined values are:

`0b0000` Not supported in hardware.

`0b0001` Supported, VFPv2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b0010` Supported, VFPv3 or VFPv4. VFPv3 adds an instruction to load a single-precision floating-point constant, and conversions between single-precision and fixed-point values.

All other values are reserved.

In Armv8-A the permitted values are `0b0000` and `0b0010`.

A value of `0b0001` or `0b0010` indicates support for all VFP single-precision instructions in the supported version of VFP, except that, in addition to this field being nonzero:

- VSQRT.F32 is only available if the Square root field is `0b0001`.
- VDIV.F32 is only available if the Divide field is `0b0001`.
- Conversion between double-precision and single-precision is only available if the double-precision field is nonzero.

SIMDReg, bits [3:0]

Advanced SIMD registers. Indicates whether the Advanced SIMD and floating-point implementation provides support for the Advanced SIMD and floating-point register bank. Defined values are:

`0b0000` The implementation has no Advanced SIMD and floating-point support.

`0b0001` The implementation includes floating-point support with 16 x 64-bit registers.

`0b0010` The implementation includes Advanced SIMD and floating-point support with 32 x 64-bit registers.

All other values are reserved.

In Armv8-A the permitted values are `0b0000` and `0b0010`.

Accessing the MVFR0

Accesses to this register use the following encodings in the System instruction encoding space:

VMRS{<c>}{<q>} <Rt>, <spec_reg>

reg
0b0111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if ELUsingAArch32(EL1) && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 == '0') || CPACR.cp10
== '00') then
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TFP == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 ==
'0') || HCPTR.TCP10 == '1') then
        AArch32.TakeHypTrapException(0x08);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x08);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
        AArch32.TakeHypTrapException(0x08);
    else
        return MVFR0;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TFP == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && ELUsingAArch32(EL2) && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 ==
'0') || HCPTR.TCP10 == '1') then
        AArch32.TakeHypTrapException(0x00);
    else
        return MVFR0;
elsif PSTATE.EL == EL3 then
    if CPACR.cp10 == '00' then
        UNDEFINED;
    else
        return MVFR0;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.116 MVFR1, Media and VFP Feature Register 1

The MVFR1 characteristics are:

Purpose

Describes the features provided by the AArch32 Advanced SIMD and Floating-point implementation.

Must be interpreted with MVFR0 and MVFR2.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page G8-6106.

Configurations

AArch32 System register MVFR1[31:0] is architecturally mapped to AArch64 System register MVFR1_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to MVFR1 are UNKNOWN.

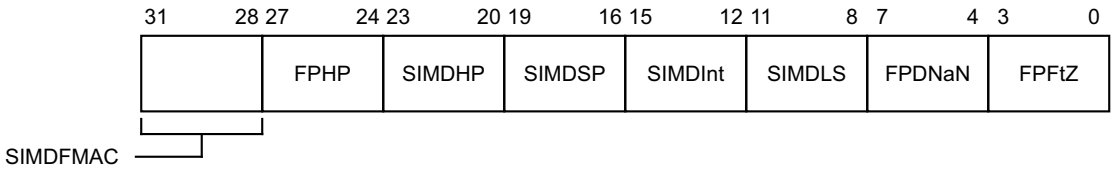
Implemented only if the implementation includes Advanced SIMD and floating-point instructions.

Attributes

MVFR1 is a 32-bit register.

Field descriptions

The MVFR1 bit assignments are:



SIMDFMAC, bits [31:28]

Advanced SIMD Fused Multiply-Accumulate. Indicates whether the Advanced SIMD implementation provides fused multiply accumulate instructions. Defined values are:

`0b0000` Not implemented.

`0b0001` Implemented.

All other values are reserved.

In Armv8-A the permitted values are `0b0000` and `0b0001`.

The Advanced SIMD and floating-point implementations must provide the same level of support for these instructions.

FPHP, bits [27:24]

Floating Point Half Precision. Indicates the level of half-precision floating-point support. Defined values are:

`0b0000` Not supported.

`0b0001` Floating-point half-precision conversion instructions are supported for conversion between single-precision and half-precision.

`0b0010` As for `0b0001`, and adds instructions for conversion between double-precision and half-precision.

`0b0011` As for `0b0010`, and adds support for half-precision floating-point arithmetic.

All other values are reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In Armv8-A the permitted values are:

- `0b0000` in an implementation without floating-point support.
- `0b0010` in an implementation with floating-point support that does not include the ARMv8.2-FP16 extension.
- `0b0011` in an implementation with floating-point support that includes the ARMv8.2-FP16 extension.

The level of support indicated by this field must be equivalent to the level of support indicated by the SIMDHP field, meaning the permitted values are:

Half Precision instructions supported	FPHP	SIMDHP
No support	`0b0000`	`0b0000`
Conversions only	`0b0010`	`0b0001`
Conversions and arithmetic	`0b0011`	`0b0010`

SIMDHP, bits [23:20]

Advanced SIMD Half Precision. Indicates the level of half-precision floating-point support. Defined values are:

`0b0000` Not supported.

`0b0001` SIMD half-precision conversion instructions are supported for conversion between single-precision and half-precision.

`0b0010` As for `0b0001`, and adds support for half-precision floating-point arithmetic.

All other values are reserved.

In Armv8-A the permitted values are:

- `0b0000` in an implementation without SIMD floating-point support.
- `0b0010` in an implementation with SIMD floating-point support that does not include the ARMv8.2-FP16 extension.
- `0b0011` in an implementation with SIMD floating-point support that includes the ARMv8.2-FP16 extension.

The level of support indicated by this field must be equivalent to the level of support indicated by the FPHP field, meaning the permitted values are:

Half Precision instructions supported	FPHP	SIMDHP
No support	`0b0000`	`0b0000`
Conversions only	`0b0010`	`0b0001`
Conversions and arithmetic	`0b0011`	`0b0010`

SIMDSP, bits [19:16]

Advanced SIMD Single Precision. Indicates whether the Advanced SIMD and floating-point implementation provides single-precision floating-point instructions. Defined values are:

`0b0000` Not implemented.

`0b0001` Implemented. This value is permitted only if the SIMDInt field is `0b0001`.

All other values are reserved.

In Armv8-A the permitted values are `0b0000` and `0b0001`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

SIMDInt, bits [15:12]

Advanced SIMD Integer. Indicates whether the Advanced SIMD and floating-point implementation provides integer instructions. Defined values are:

`0b0000` Not implemented.

`0b0001` Implemented.

All other values are reserved.

In Armv8-A the permitted values are `0b0000` and `0b0001`.

SIMDLS, bits [11:8]

Advanced SIMD Load/Store. Indicates whether the Advanced SIMD and floating-point implementation provides load/store instructions. Defined values are:

`0b0000` Not implemented.

`0b0001` Implemented.

All other values are reserved.

In Armv8-A the permitted values are `0b0000` and `0b0001`.

FPDNaN, bits [7:4]

Default NaN mode. Indicates whether the floating-point implementation provides support only for the Default NaN mode. Defined values are:

`0b0000` Not implemented, or hardware supports only the Default NaN mode.

`0b0001` Hardware supports propagation of NaN values.

All other values are reserved.

In Armv8-A the permitted values are `0b0000` and `0b0001`.

FPFtZ, bits [3:0]

Flush to Zero mode. Indicates whether the floating-point implementation provides support only for the Flush-to-Zero mode of operation. Defined values are:

`0b0000` Not implemented, or hardware supports only the Flush-to-Zero mode of operation.

`0b0001` Hardware supports full denormalized number arithmetic.

All other values are reserved.

In Armv8-A the permitted values are `0b0000` and `0b0001`.

Accessing the MVFR1

Accesses to this register use the following encodings in the System instruction encoding space:

VMRS{<c>}{<q>} <Rt>, <spec_reg>

reg
0b0110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if ELUsingAArch32(EL1) && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 == '0') || CPACR.cp10
== '00') then
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TFP == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 ==
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
'0') || HCPTR.TCP10 == '1') then
        AArch32.TakeHypTrapException(0x08);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x08);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
        AArch32.TakeHypTrapException(0x08);
    else
        return MVFR1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TFP == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 ==
'0') || HCPTR.TCP10 == '1') then
        AArch32.TakeHypTrapException(0x00);
    else
        return MVFR1;
elsif PSTATE.EL == EL3 then
    if CPACR.cp10 == '00' then
        UNDEFINED;
    else
        return MVFR1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.117 MVFR2, Media and VFP Feature Register 2

The MVFR2 characteristics are:

Purpose

Describes the features provided by the AArch32 Advanced SIMD and Floating-point implementation.

Must be interpreted with MVFR0 and MVFR1.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page G8-6106.

Configurations

AArch32 System register MVFR2[31:0] is architecturally mapped to AArch64 System register MVFR2_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to MVFR2 are UNKNOWN.

Implemented only if the implementation includes Advanced SIMD and floating-point instructions.

Attributes

MVFR2 is a 32-bit register.

Field descriptions

The MVFR2 bit assignments are:

31 … 8	7 … 4	3 … 0
RES0	FPMisc	SIMDMisc

Bits [31:8]

Reserved, RES0.

FPMisc, bits [7:4]

Indicates whether the floating-point implementation provides support for miscellaneous VFP features.

`0b0000` Not implemented, or no support for miscellaneous features.

`0b0001` Support for Floating-point selection.

`0b0010` As `0b0001`, and Floating-point Conversion to Integer with Directed Rounding modes.

`0b0011` As `0b0010`, and Floating-point Round to Integer Floating-point.

`0b0100` As `0b0011`, and Floating-point MaxNum and MinNum.

All other values are reserved.

In Armv8-A the permitted values are `0b0000` and `0b0100`.

SIMDMisc, bits [3:0]

Indicates whether the Advanced SIMD implementation provides support for miscellaneous Advanced SIMD features.

`0b0000` Not implemented, or no support for miscellaneous features.

`0b0001` Floating-point Conversion to Integer with Directed Rounding modes.

`0b0010` As `0b0001`, and Floating-point Round to Integer Floating-point.

`0b0011` As `0b0010`, and Floating-point MaxNum and MinNum.

All other values are reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In Armv8-A the permitted values are `0b0000` and `0b0011`.

Accessing the MVFR2

Accesses to this register use the following encodings in the System instruction encoding space:

VMRS{<c>}{<q>} <Rt>, <spec_reg>

reg
0b0101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if ELUsingAArch32(EL1) && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 == '0') || CPACR.cp10
== '00') then
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TFP == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 ==
'0') || HCPTR.TCP10 == '1') then
        AArch32.TakeHypTrapException(0x08);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x08);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
        AArch32.TakeHypTrapException(0x08);
    else
        return MVFR2;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TFP == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 ==
'0') || HCPTR.TCP10 == '1') then
        AArch32.TakeHypTrapException(0x00);
    else
        return MVFR2;
elsif PSTATE.EL == EL3 then
    if CPACR.cp10 == '00' then
        UNDEFINED;
    else
        return MVFR2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.118 NMRR, Normal Memory Remap Register

The NMRR characteristics are:

Purpose

Provides additional mapping controls for memory regions that are mapped as Normal memory by their entry in the PRRR.

Used in conjunction with the PRRR.

Configurations

AArch32 System register NMRR[31:0] is architecturally mapped to AArch64 System register MAIR_EL1[63:32].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to NMRR are UNKNOWN.

MAIR1 and NMRR are the same register, with a different view depending on the value of TTBCR.EAE:

- When it is set to 0, the register is as described in NMRR.
- When it is set to 1, the register is as described in MAIR1.

Attributes

NMRR is a 32-bit register.

Field descriptions

The NMRR bit assignments are:

When TTBCR.EAE == 0:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OR7		OR6		OR5		OR4		OR3		OR2		OR1		OR0		IR7		IR6		IR5		IR4		IR3		IR2		IR1		IR0	

OR<n>, bits [2n+17:2n+16], for n = 0 to 7

Outer Cacheable property mapping for memory attributes n, if the region is mapped as Normal memory by the PRRR.TR<n> entry. n is the value of the TEX[0], C, and B bits concatenated. The possible values of this field are:

`0b00` Region is Non-cacheable.

`0b01` Region is Write-Back, Write-Allocate.

`0b10` Region is Write-Through, no Write-Allocate.

`0b11` Region is Write-Back, no Write-Allocate.

The meaning of the field with n = 6 is IMPLEMENTATION DEFINED and might differ from the meaning given here. This is because the meaning of the attribute combination {TEX[0] = 1, C = 1, B = 0} is IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

IR<n>, bits [2n+1:2n], for n = 0 to 7

Inner Cacheable property mapping for memory attributes n, if the region is mapped as Normal memory by the PRRR.TR<n> entry. n is the value of the TEX[0], C, and B bits concatenated. The possible values of this field are:

`0b00` Region is Non-cacheable.

`0b01` Region is Write-Back, Write-Allocate.

`0b10` Region is Write-Through, no Write-Allocate.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b11` Region is Write-Back, no Write-Allocate.

The meaning of the field with n = 6 is IMPLEMENTATION DEFINED and might differ from the meaning given here. This is because the meaning of the attribute combination {TEX[0] = 1, C = 1, B = 0} is IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

Accessing the NMRR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1010	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T10 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T10 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TRVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return NMRR_NS;
    else
        return NMRR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return NMRR_NS;
    else
        return NMRR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return NMRR_S;
    else
        return NMRR_NS;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1010	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T10 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T10 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        NMRR_NS = R[t];
    else
        NMRR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        NMRR_NS = R[t];
    else
        NMRR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' && CP15SDISABLE == HIGH then
        UNDEFINED;
    elsif SCR.NS == '0' && CP15SDISABLE2 == HIGH then
        UNDEFINED;
    else
        if SCR.NS == '0' then
            NMRR_S = R[t];
        else
            NMRR_NS = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.119 NSACR, Non-Secure Access Control Register

The NSACR characteristics are:

Purpose

When EL3 is implemented and can use AArch32, defines the Non-secure access permissions to Trace, Advanced SIMD and floating-point functionality. Also includes IMPLEMENTATION DEFINED bits that can define Non-secure access permissions for IMPLEMENTATION DEFINED functionality.

Configurations

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to NSACR are UNKNOWN.

Note

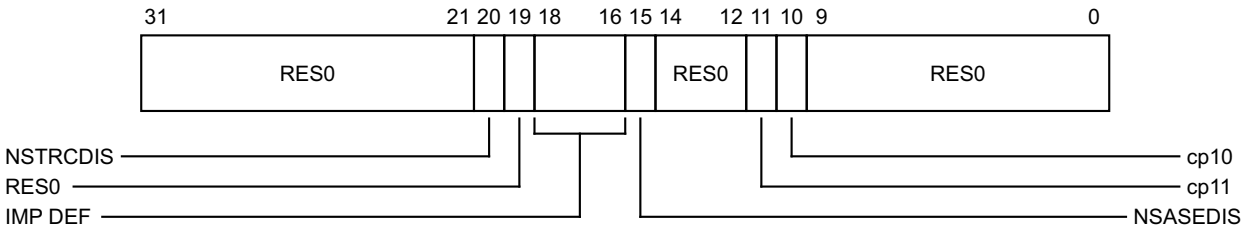
In AArch64 state, the NSACR controls are replaced by controls in CPTR_EL3.

Attributes

NSACR is a 32-bit register.

Field descriptions

The NSACR bit assignments are:



If EL3 is implemented and is using AArch64 then:

- Any read of the NSACR from Non-secure EL2 or Non-secure EL1 returns a value of `0x00000C00`.
- Any read or write to NSACR from Secure EL1 is trapped as an exception to EL3.

If EL3 is not implemented, then any read of the NSACR from EL2 or EL1 returns a value of `0x00000C00`.

Bits [31:21]

Reserved, RES0.

NSTRCDIS, bit [20]

Disables Non-secure System register accesses to all implemented trace registers.

`0b0` This control has no effect on:

- System register access to implemented trace registers.
- The behavior of CPACR.TRCDIS and HCPTR.TTA.

`0b1` Non-secure System register accesses to all implemented trace registers are disabled, meaning:

- CPACR.TRCDIS behaves as RAO/WI in Non-secure state, regardless of its actual value.
- HCPTR.TTA behaves as RAO/WI, regardless of its actual value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The implementation of this field must correspond to the implementation of the CPACR.TRCDIS field:

- If CPACR.TRCDIS is RAZ/WI, this field is RAZ/WI.
- If CPACR.TRCDIS is RW, this field is RW.

Note

- The ETMv4 architecture does not permit EL0 to access the trace registers. If the implementation includes an ETMv4 implementation, EL0 accesses to the trace registers are UNDEFINED.
- The architecture does not provide Non-secure access controls on trace register accesses through the optional memory-mapped external debug interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

In a system where the PE resets into EL3, this field resets to `0`.

Bit [19]

Reserved, RES0.

IMPLEMENTATION DEFINED, bits [18:16]

IMPLEMENTATION DEFINED.

NSASEDIS, bit [15]

Disables Non-secure access to the Advanced SIMD functionality.

`0b0` This control has no effect on:

- Non-secure access to Advanced SIMD functionality.
- The behavior of CPACR.ASEDIS and HCPTR.TASE.

`0b1` Non-secure access to the Advanced SIMD functionality is disabled, meaning:

- CPACR.ASEDIS behaves as RAO/WI in Non-secure state, regardless of its actual value.
- HCPTR.TASE behaves as RAO/WI, regardless of its actual value.

The implementation of this field must correspond to the implementation of the CPACR.ASEDIS field:

- If CPACR.ASEDIS is RES0, this field is RES0. If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES0.
- If CPACR.ASEDIS is RAZ/WI, this field is RAZ/WI.
- If CPACR.ASEDIS is RW, this field is RW.

In a system where the PE resets into EL3, this field resets to `0`.

Bits [14:12]

Reserved, RES0.

cp11, bit [11]

The value of this field is ignored. If this field is programmed with a different value to the cp10 field then this field is UNKNOWN on a direct read of the NSACR.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES0.

In a system where the PE resets into EL3, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

cp10, bit [10]

Enable Non-secure access to the Advanced SIMD and floating-point features. Possible values of the fields are:

`0b0` Advanced SIMD and floating-point features can be accessed only from Secure state. Any attempt to access this functionality from Non-secure state is UNDEFINED.

When the PE is in Non-secure state:

- The CPACR.{cp11, cp10} fields ignore writes and read as `0b00`, access denied.
- The HCPTR.{TCP11, TCP10} fields behave as RAO/WI, regardless of their actual values.

`0b1` Advanced SIMD and floating-point features can be accessed from both Security states.

If Non-secure access to the Advanced SIMD and floating-point functionality is enabled, the CPACR must be checked to determine the level of access that is permitted.

The Advanced SIMD and floating-point features controlled by these fields are:

- Execution of any floating-point or Advanced SIMD instruction.
- Any access to the Advanced SIMD and floating-point registers D0-D31 and their views as S0-S31 and Q0-Q15.
- Any access to the FPSCR, FPSID, MVFR0, MVFR1, MVFR2, or FPEXC System registers.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES0.

In a system where the PE resets into EL3, this field resets to an architecturally UNKNOWN value.

Bits [9:0]

Reserved, RES0.

Accessing the NSACR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0001	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif !ELUsingAArch32(EL2) && SCR_EL3.<NS,EEL2> == '01' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif !HaveEL(EL3) || (!ELUsingAArch32(EL3) && SCR_EL3.NS == '1') then
        return Zeros(20):'1100':Zeros(8);
    else
        return NSACR;
elsif PSTATE.EL == EL2 then
    if !HaveEL(EL3) || (!ELUsingAArch32(EL3) && SCR_EL3.NS == '1') then
        return Zeros(20):'1100':Zeros(8);
    else
        return NSACR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    return NSACR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0001	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif !ELUsingAArch32(EL2) && SCR_EL3.<NS,EEL2> == '01' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' && CP15SDISABLE2 == HIGH then
        UNDEFINED;
    else
        NSACR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.120 PAR, Physical Address Register

The PAR characteristics are:

Purpose

Returns the output address (OA) from an Address translation instruction that executed successfully, or fault information if the instruction did not execute successfully.

Configurations

AArch32 System register PAR[63:0] is architecturally mapped to AArch64 System register PAR_EL1[63:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to PAR are UNKNOWN.

PAR is accessed as a 32-bit value:

- When the PE is not in Hyp mode and is using the Short-descriptor translation table format.
- When the PE is in Hyp mode and executes an ATS12NSOPR, ATS12NSOPW, ATS12NSOUR, or ATS12NSOUW instruction and the value of HCR.VM is 0 and the value of TTBCR.EAE is 0.

In these cases, PAR[63:32] is RES0.

Otherwise, the PAR is accessed as a 64-bit value, if any of the following is true:

- When using the Long-descriptor translation table format.
- If the stage 1 address translation is disabled and TTBCR.EAE is set to 1.
- In an implementation that includes EL2, for the result of an ATS1Cxx instruction performed from Hyp mode.

For PL1&0 stage 1 translations, TTBCR.EAE selects the translation table format.

Attributes

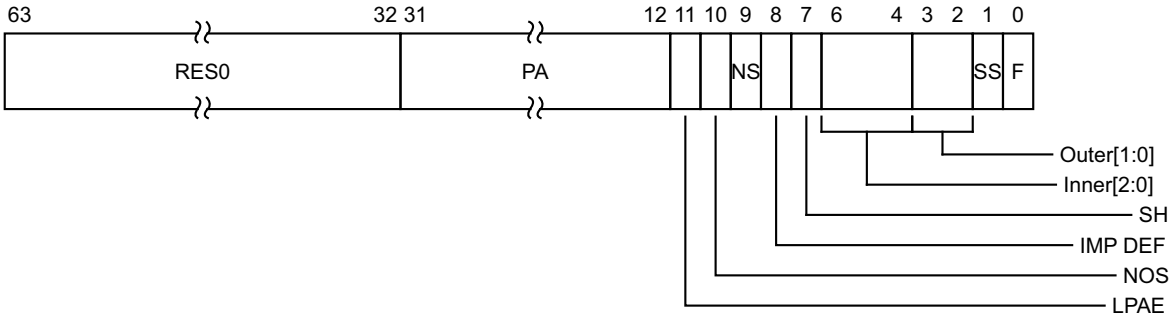
PAR is a 64-bit register that can also be accessed as a 32-bit value. If it is accessed as a 32-bit register, accesses read and write bits[31:0] and do not modify bits[63:32].

The Configurations section specifies the cases where each PAR format is used.

Field descriptions

The PAR bit assignments are:

When the instruction returned a 32-bit value to the PAR, PAR.F==0:



This section describes the register value returned by the successful execution of an Address translation instruction. Software might subsequently write a different value to the register, and that write does not affect the operation of the PE.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

On a successful conversion, the PAR can return a value that indicates the resulting attributes, rather than the values that appear in the translation table descriptors. More precisely:

- Memory attribute fields are permitted to report the resulting attributes, as determined by any permitted implementation choices and any applicable configuration bits, instead of reporting the values that appear in the translation table descriptors. This applies to the NOS, SH, Inner, and Outer fields.
- See the NS bit description for constraints on the value it returns.

Bits [63:32]

Reserved, RES0.

PA, bits [31:12]

Output address. The output address (OA) corresponding to the supplied input address. This field returns address bits[31:12].

This field resets to an architecturally UNKNOWN value.

LPAE, bit [11]

When updating the PAR with the result of the translation operation, this bit is set as follows:

`0b0` Short-descriptor translation table format used. This means the PAR returned a 32-bit value.

This field resets to an architecturally UNKNOWN value.

NOS, bit [10]

Not Outer Shareable. When the returned value of PAR.SH is 1, indicates the Shareability attribute for the physical memory region:

`0b0` Memory region is Outer Shareable.

`0b1` Memory region is Inner Shareable.

When the returned value of PAR.SH is 0 the value returned to this field is UNKNOWN.

The value returned in this field can be the resulting attribute, as determined by any permitted implementation choices and any applicable configuration bits, instead of the value that appears in the translation table descriptor.

This field resets to an architecturally UNKNOWN value.

NS, bit [9]

Non-secure. The NS attribute for a translation table entry from a Secure translation regime.

For a result from a Secure translation regime, this bit reflects the Security state of the physical address space of the translation. This means it reflects the effect of the NSTable bits of earlier levels of the translation table walk if those NSTable bits have an effect on the translation.

For a result from a Non-secure translation regime, this bit is UNKNOWN.

This field resets to an architecturally UNKNOWN value.

IMPLEMENTATION DEFINED, bit [8]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

SH, bit [7]

Shareability. Indicates whether the physical memory region is Non-shareable:

`0b0` Memory is Non-shareable.

`0b1` Memory is shareable, and PAR.NOS indicates whether the region is Outer Shareable or Inner Shareable.

The value returned in this field can be the resulting attribute, as determined by any permitted implementation choices and any applicable configuration bits, instead of the value that appears in the translation table descriptor.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

Inner[2:0], bits [6:4]

Inner cacheability attribute for the region. Permitted values are:

`0b000` Non-cacheable.

`0b001` Device-nGnRnE.

`0b011` Device-nGnRE.

`0b101` Write-Back, Write-Allocate.

`0b110` Write-Through.

`0b111` Write-Back, no Write-Allocate.

The values `0b010` and `0b100` are reserved.

The value returned in this field can be the resulting attribute, as determined by any permitted implementation choices and any applicable configuration bits, instead of the value that appears in the translation table descriptor.

This field resets to an architecturally UNKNOWN value.

Outer[1:0], bits [3:2]

Outer cacheability attribute for the region. Permitted values are:

`0b00` Non-cacheable.

`0b01` Write-Back, Write-Allocate.

`0b10` Write-Through, no Write-Allocate.

`0b11` Write-Back, no Write-Allocate.

The value returned in this field can be the resulting attribute, as determined by any permitted implementation choices and any applicable configuration bits, instead of the value that appears in the translation table descriptor.

This field resets to an architecturally UNKNOWN value.

SS, bit [1]

Supersection. Used to indicate if the result is a Supersection:

`0b0` Result is not a Supersection. PAR[31:12] contains OA[31:12].

`0b1` Result is a Supersection, and:

- PAR[31:24] contains OA[31:24].
- PAR[23:16] contains OA[39:32].
- PAR[15:12] contains `0b0000`.

If an implementation supports less than 40 bits of physical address, the bits in the PAR field that correspond to physical address bits that are not implemented are UNKNOWN.

This field resets to an architecturally UNKNOWN value.

F, bit [0]

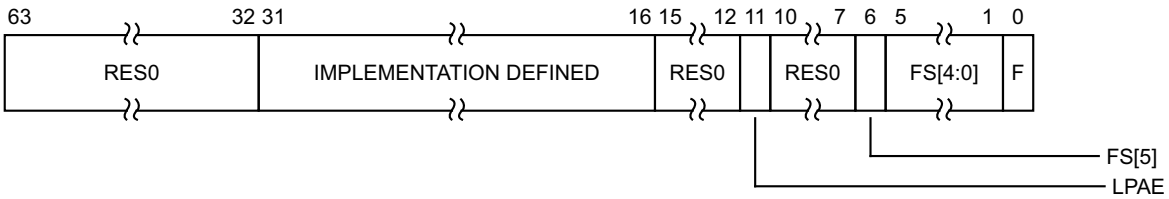
Indicates whether the instruction performed a successful address translation.

`0b0` Address translation completed successfully.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When the instruction returned a 32-bit value to the PAR, PAR.F==1:



This section describes the register value returned by a fault on the execution of an Address translation instruction. Software might subsequently write a different value to the register, and that write does not affect the operation of the PE.

Bits [63:32]

Reserved, RES0.

IMPLEMENTATION DEFINED, bits [31:16]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

Bits [15:12]

Reserved, RES0.

LPAE, bit [11]

When updating the PAR with the result of the translation operation, this bit is set as follows:

`0b0` Short-descriptor translation table format used. This means the PAR returned a 32-bit value.

This field resets to an architecturally UNKNOWN value.

Bits [10:7]

Reserved, RES0.

FS[5], bit [6]

Fault status bits, external abort type. Provides an IMPLEMENTATION DEFINED classification of an External abort. Values are as in the DFSR.ExT field when using the Short-descriptor translation table format.

In an implementation that does not provide any classification of External aborts, this bit is RES0.

For aborts other than External aborts this bit always returns 0.

This field resets to an architecturally UNKNOWN value.

FS[4:0], bits [5:1]

Fault status bits. Values are as in the DFSR.FS field when using the Short-descriptor translation table format.

`0b00001` Alignment fault.

`0b00011` Access flag fault, level 1.

`0b00100` Fault on instruction cache maintenance.

`0b00101` Translation fault, level 1.

`0b00110` Access flag fault, level 2.

`0b00111` Translation fault, level 2.

`0b01001` Domain fault, level 1.

`0b01011` Domain fault, level 2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b01100` Synchronous External abort, on translation table walk, level 1.

`0b01101` Permission fault, level 1.

`0b01110` Synchronous External abort, on translation table walk, level 2.

`0b01111` Permission fault, level 2.

`0b10000` TLB conflict abort.

`0b11001` **When RAS is not implemented** Synchronous parity or ECC error on memory access, not on translation table walk.

`0b11100` **When RAS is not implemented** Synchronous parity or ECC error on translation table walk, level 1.

`0b11110` **When RAS is not implemented** Synchronous parity or ECC error on translation table walk, level 2.

This field resets to an UNKNOWN value.

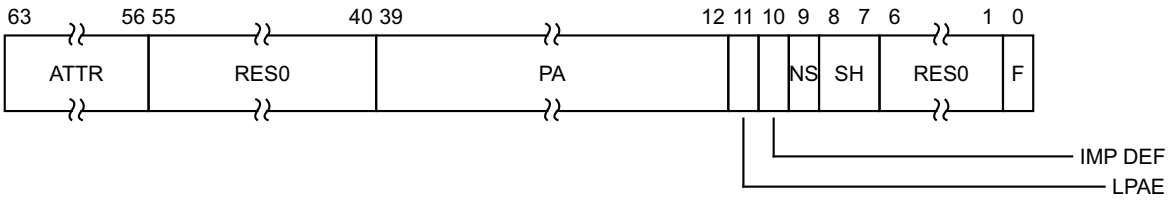
F, bit [0]

Indicates whether the instruction performed a successful address translation.

`0b1` Address translation aborted.

This field resets to an architecturally UNKNOWN value.

When the instruction returned a 64-bit value to the PAR, PAR.F==0:



This section describes the register value returned by the successful execution of an Address translation instruction. Software might subsequently write a different value to the register, and that write does not affect the operation of the PE.

On a successful conversion, the PAR can return a value that indicates the resulting attributes, rather than the values that appear in the translation table descriptors. More precisely:

- Memory attribute fields are permitted to report the resulting attributes, as determined by any permitted implementation choices and any applicable configuration bits, instead of reporting the values that appear in the translation table descriptors. This applies to the ATTR and SH fields.
- See the NS bit description for constraints on the value it returns.

ATTR, bits [63:56]

Memory attributes for the returned output address. This field uses the same encoding as the Attr<n> fields in MAIR0 and MAIR1.

The value returned in this field can be the resulting attribute, as determined by any permitted implementation choices and any applicable configuration bits, instead of the value that appears in the translation table descriptor.

This field resets to an architecturally UNKNOWN value.

Bits [55:40]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

PA, bits [39:12]

Output address. The output address (OA) corresponding to the supplied input address. This field returns address bits[39:12].

This field resets to an architecturally UNKNOWN value.

LPAE, bit [11]

When updating the PAR with the result of the translation operation, this bit is set as follows:

`0b1` Long-descriptor translation table format used. This means the PAR returned a 64-bit value.

This field resets to an architecturally UNKNOWN value.

IMPLEMENTATION DEFINED, bit [10]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

NS, bit [9]

Non-secure. The NS attribute for a translation table entry from a Secure translation regime.

For a result from a Secure translation regime, this bit reflects the Security state of the physical address space of the translation. This means it reflects the effect of the NSTable bits of earlier levels of the translation table walk if those NSTable bits have an effect on the translation.

For a result from a Non-secure translation regime, this bit is UNKNOWN.

This field resets to an architecturally UNKNOWN value.

SH, bits [8:7]

Shareability attribute, for the returned output address. Permitted values are:

`0b00` Non-shareable.

`0b10` Outer Shareable.

`0b11` Inner Shareable.

The value `0b01` is reserved.

Note

This field returns the value `0b10` for:

- Any type of Device memory.
- Normal memory with both Inner Non-cacheable and Outer Non-cacheable attributes.

The value returned in this field can be the resulting attribute, as determined by any permitted implementation choices and any applicable configuration bits, instead of the value that appears in the translation table descriptor.

This field resets to an architecturally UNKNOWN value.

Bits [6:1]

Reserved, RES0.

F, bit [0]

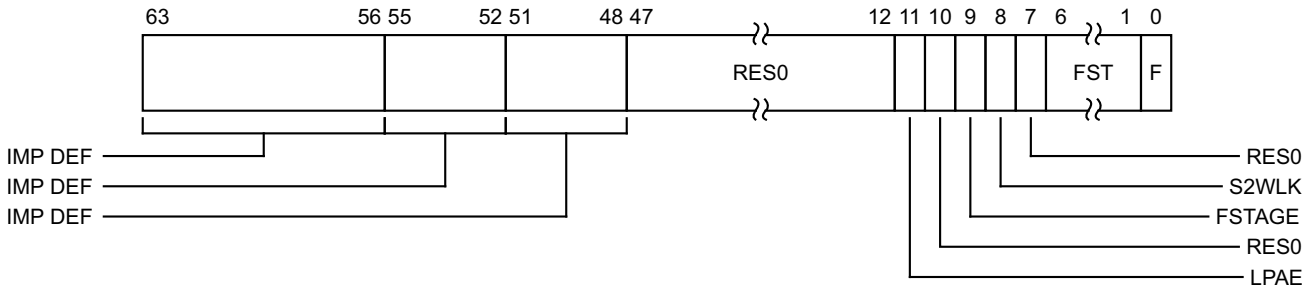
Indicates whether the instruction performed a successful address translation.

`0b0` Address translation completed successfully.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When the instruction returned a 64-bit value to the PAR, PAR.F==1:



This section describes the register value returned by a fault on the execution of an Address translation instruction. Software might subsequently write a different value to the register, and that write does not affect the operation of the PE.

IMPLEMENTATION DEFINED, bits [63:56]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

IMPLEMENTATION DEFINED, bits [55:52]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

IMPLEMENTATION DEFINED, bits [51:48]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

Bits [47:12]

Reserved, RES0.

LPAE, bit [11]

When updating the PAR with the result of the translation operation, this bit is set as follows:

`0b1` Long-descriptor translation table format used. This means the PAR returned a 64-bit value.

This field resets to an architecturally UNKNOWN value.

Bit [10]

Reserved, RES0.

FSTAGE, bit [9]

Indicates the translation stage at which the translation aborted:

`0b0` Translation aborted because of a fault in the stage 1 translation.

`0b1` Translation aborted because of a fault in the stage 2 translation.

This field resets to an architecturally UNKNOWN value.

S2WLK, bit [8]

If this bit is set to 1, it indicates the translation aborted because of a stage 2 fault during a stage 1 translation table walk.

This field resets to an architecturally UNKNOWN value.

Bit [7]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

FST, bits [6:1]

Fault status field. Values are as in the DFSR.STATUS and IFSR.STATUS fields when using the Long-descriptor translation table format.

`0b000000` Address size fault in translation table base register.

`0b000001` Address size fault, level 1.

`0b000010` Address size fault, level 2.

`0b000011` Address size fault, level 3.

`0b000101` Translation fault, level 1.

`0b000110` Translation fault, level 2.

`0b000111` Translation fault, level 3.

`0b001001` Access flag fault, level 1.

`0b001010` Access flag fault, level 2.

`0b001011` Access flag fault, level 3.

`0b001101` Permission fault, level 1.

`0b001110` Permission fault, level 2.

`0b001111` Permission fault, level 3.

`0b010000` Synchronous External abort, not on translation table walk.

`0b010101` Synchronous External abort, on translation table walk, level 1.

`0b010110` Synchronous External abort, on translation table walk, level 2.

`0b010111` Synchronous External abort, on translation table walk, level 3.

`0b011000` **When RAS is not implemented** Synchronous parity or ECC error on memory access, not on translation table walk.

`0b011001` **When RAS is not implemented** Asynchronous SError interrupt, from a parity or ECC error on memory access.

`0b011101` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk, level 1.

`0b011110` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk, level 2.

`0b011111` **When RAS is not implemented** Synchronous parity or ECC error on memory access on translation table walk, level 3.

`0b100001` Alignment fault.

`0b110000` TLB conflict abort.

This field resets to an architecturally UNKNOWN value.

F, bit [0]

Indicates whether the instruction performed a successful address translation.

`0b1` Address translation aborted.

This field resets to an architecturally UNKNOWN value.

Accessing the PAR

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0100	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return PAR_NS<31:0>;
    else
        return PAR<31:0>;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return PAR_NS<31:0>;
    else
        return PAR<31:0>;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return PAR_S<31:0>;
    else
        return PAR_NS<31:0>;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0100	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        PAR_NS = ZeroExtend(R[t]);
    else
        PAR = ZeroExtend(R[t]);
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        PAR_NS = ZeroExtend(R[t]);
    else
        PAR = ZeroExtend(R[t]);
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        PAR_S = ZeroExtend(R[t]);
    else
        PAR_NS = ZeroExtend(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b0111	0b0000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x04);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return PAR_NS;
    else
        return PAR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return PAR_NS;
    else
        return PAR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return PAR_S;
    else
        return PAR_NS;
```

MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b0111	0b0000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x04);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        PAR_NS = R[t2]:R[t];
    else
        PAR = R[t2]:R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        PAR_NS = R[t2]:R[t];
    else
        PAR = R[t2]:R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        PAR_S = R[t2]:R[t];
    else
        PAR_NS = R[t2]:R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.121 PRRR, Primary Region Remap Register

The PRRR characteristics are:

Purpose

Controls the top level mapping of the TEX[0], C, and B memory region attributes.

Configurations

AArch32 System register PRRR[31:0] is architecturally mapped to AArch64 System register MAIR_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to PRRR are UNKNOWN.

MAIR0 and PRRR are the same register, with a different view depending on the value of TTBCR.EAE:

- When it is set to 0, the register is as described in PRRR.
- When it is set to 1, the register is as described in MAIR0.

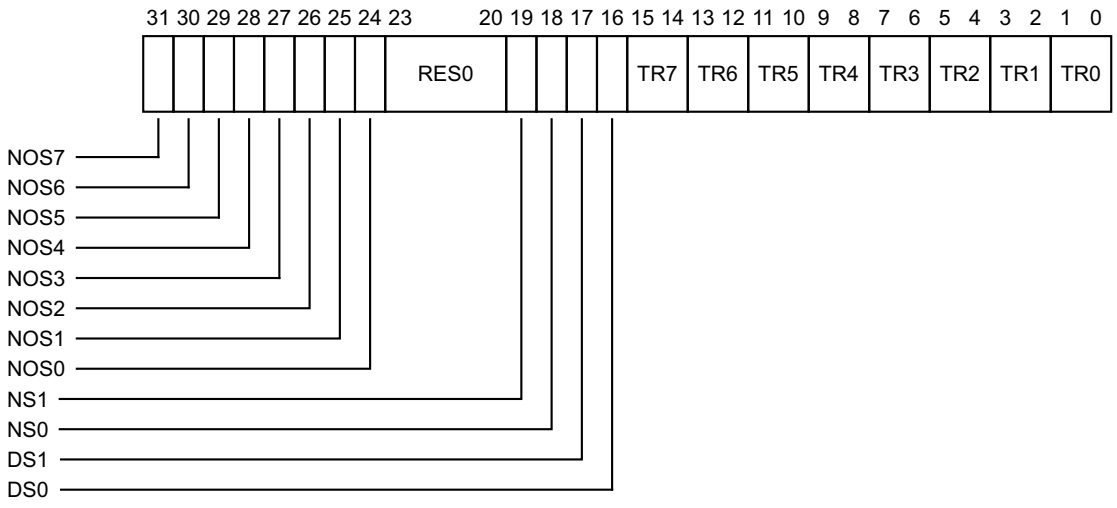
Attributes

PRRR is a 32-bit register.

Field descriptions

The PRRR bit assignments are:

When TTBCR.EAE == 0:



NOS<n>, bit [n+24], for n = 0 to 7

Not Outer Shareable. NOS<n> is the Outer Shareable property for memory attributes n, if the region is mapped as Normal memory that is not Inner Non-cacheable, Outer Non-cacheable, and the appropriate PRRR.{NS0, NS1} field identifies the region as shareable. n is the value of the concatenation of the {TEX[0], C, B} bits from the translation table descriptor. The possible values of each NOS<n> field other than NOS6 are:

`0b0` Memory region is Outer Shareable.

`0b1` Memory region is Inner Shareable.

The value of this bit is ignored if the region is:

- Device memory

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Normal memory that is at least one of:
  - Inner Non-cacheable, Outer Non-cacheable.
  - Identified by the appropriate PRRR.{NS0, NS1} field as Non-shareable.

The meaning of the NOS6 field is IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

Bits [23:20]

Reserved, RES0.

NS1, bit [19]

Mapping of S = 1 attribute for Normal memory regions. This field is used in determining the Shareability of a memory region that is mapped to Normal memory and both:

- Is not Inner Non-cacheable, Outer Non-cacheable.
- Has the S bit in the translation table descriptor set to 1.

The possible values of this bit are:

`0b0` Region is Non-shareable.

`0b1` Region is shareable. The value of the appropriate PRRR.NOS<n> field determines whether the region is Inner Shareable or Outer Shareable.

This field resets to an architecturally UNKNOWN value.

NS0, bit [18]

Mapping of S = 0 attribute for Normal memory regions. This field is used in determining the Shareability of a memory region that is mapped to Normal memory and both:

- Is not Inner Non-cacheable, Outer Non-cacheable.
- Has the S bit in the translation table descriptor set to 0.

The possible values of this bit are:

`0b0` Region is Non-shareable.

`0b1` Region is shareable. The value of the appropriate PRRR.NOS<n> field determines whether the region is Inner Shareable or Outer Shareable.

This field resets to an architecturally UNKNOWN value.

DS1, bit [17]

Mapping of S = 1 attribute for Device memory. From Armv8, all types of Device memory are Outer Shareable, and therefore this bit is RES1.

This field resets to an architecturally UNKNOWN value.

DS0, bit [16]

Mapping of S = 0 attribute for Device memory. From Armv8, all types of Device memory are Outer Shareable, and therefore this bit is RES1.

This field resets to an architecturally UNKNOWN value.

TR<n>, bits [2n+1:2n], for n = 0 to 7

TR<n> is the primary TEX mapping for memory attributes n, and defines the mapped memory type for a region with attributes n. n is the value of the concatenation of the {TEX[0], C, B} bits from the translation table descriptor. The possible values for each field other than TR6 are:

`0b00` Device-nGnRnE memory

`0b01` Device-nGnRE memory

`0b10` Normal memory

The value `0b11` is reserved. The effect of programming a field to `0b11` is CONSTRAINED UNPREDICTABLE, see *Reserved values in System and memory-mapped registers and translation table entries* on page K1-7853.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The meaning of the TR6 field is IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

Accessing the PRRR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1010	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T10 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T10 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TRVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return PRRR_NS;
    else
        return PRRR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return PRRR_NS;
    else
        return PRRR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return PRRR_S;
    else
        return PRRR_NS;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1010	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T10 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T10 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        PRRR_NS = R[t];
    else
        PRRR = R[t];
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        PRRR_NS = R[t];
    else
        PRRR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' && CP15SDISABLE == HIGH then
        UNDEFINED;
    elsif SCR.NS == '0' && CP15SDISABLE2 == HIGH then
        UNDEFINED;
    else
        if SCR.NS == '0' then
            PRRR_S = R[t];
        else
            PRRR_NS = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.122 REVIDR, Revision ID Register

The REVIDR characteristics are:

Purpose

Provides implementation-specific minor revision information.

Configurations

AArch32 System register REVIDR[31:0] is architecturally mapped to AArch64 System register REVIDR_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to REVIDR are UNKNOWN.

If REVIDR has the same value as MIDR, then its contents have no significance.

Attributes

REVIDR is a 32-bit register.

Field descriptions

The REVIDR bit assignments are:

31 … 0
IMPLEMENTATION DEFINED

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

Accessing the REVIDR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0000	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        return REVIDR;
elsif PSTATE.EL == EL2 then
    return REVIDR;
elsif PSTATE.EL == EL3 then
    return REVIDR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.123 RMR, Reset Management Register

The RMR characteristics are:

Purpose

If EL1 or EL3 is the highest implemented Exception level and this register is implemented:

- A write to the register at the highest implemented Exception level can request a Warm reset.
- If the highest implemented Exception level can use AArch32 and AArch64, this register specifies the Execution state that the PE boots into on a Warm reset.

Configurations

AArch32 System register RMR[31:0] is architecturally mapped to AArch64 System register RMR_EL1[31:0] when the highest implemented Exception level is EL1.

AArch32 System register RMR[31:0] is architecturally mapped to AArch64 System register RMR_EL3[31:0] when EL3 is implemented.

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to RMR are UNKNOWN.

Only implemented if EL1 or EL3 is the highest implemented Exception level. In this case:

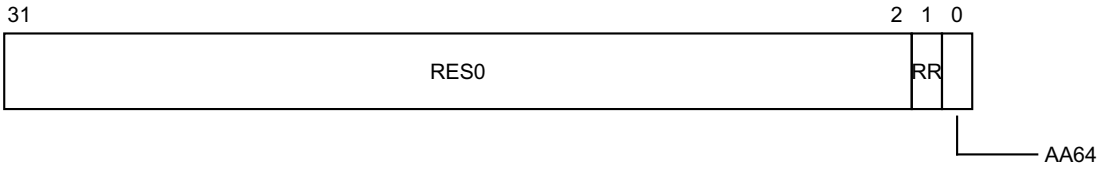
- If the highest implemented Exception level can use AArch32 and AArch64 then this register must be implemented.
- If the highest implemented Exception level cannot use AArch64 then it is IMPLEMENTATION DEFINED whether the register is implemented.

Attributes

RMR is a 32-bit register.

Field descriptions

The RMR bit assignments are:



Bits [31:2]

Reserved, RES0.

RR, bit [1]

Reset Request. Setting this bit to 1 requests a Warm reset.

This field resets to `0`.

AA64, bit [0]

When the highest implemented Exception level can use AArch64, determines which Execution state the PE boots into after a Warm reset:

`0b0` AArch32.

`0b1` AArch64.

On coming out of the Warm reset, execution starts at the IMPLEMENTATION DEFINED reset vector address of the specified Execution state.

If the highest implemented Exception level cannot use AArch64 this bit is RAZ/WI.

When implemented as a RW field, this field resets to 0 on a Cold reset.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the RMR

When EL3 is implemented, Arm deprecates accessing this register from any PE mode other than Monitor mode.

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1100	0b0000	0b010

```
if PSTATE.EL IN {EL3, EL1} && IsHighestEL(PSTATE.EL) then
    return RMR;
else
    UNDEFINED;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1100	0b0000	0b010

```
if PSTATE.EL IN {EL3, EL1} && IsHighestEL(PSTATE.EL) then
    RMR = R[t];
else
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.124 RVBAR, Reset Vector Base Address Register

The RVBAR characteristics are:

Purpose

If EL3 is not implemented, contains the IMPLEMENTATION DEFINED address that execution starts from after reset when executing in AArch32 state.

Configurations

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to RVBAR are UNKNOWN.

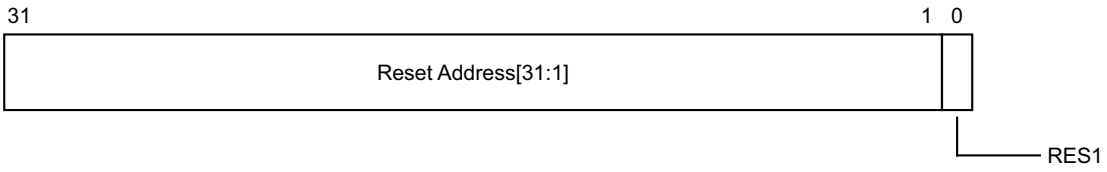
This register is only implemented if the highest Exception level implemented is capable of using AArch32, and is not EL3.

Attributes

RVBAR is a 32-bit register.

Field descriptions

The RVBAR bit assignments are:



Bits [31:1]

Reset Address[31:1]. Bits [31:1] of the IMPLEMENTATION DEFINED address that execution starts from after reset when executing in 32-bit state.

Bit [0]

Reserved, RES1.

Accessing the RVBAR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1100	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsHighestEL(EL1) then
        return RVBAR;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T12 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T12 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif !ELUsingAArch32(EL2) && SCR_EL3.<NS,EEL2> == '01' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if IsHighestEL(EL2) then
        return RVBAR;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    return MVBAR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.125 SCR, Secure Configuration Register

The SCR characteristics are:

Purpose

When EL3 is implemented and can use AArch32, defines the configuration of the current Security state. It specifies:

- The Security state, either Secure or Non-secure.
- What mode the PE branches to if an IRQ, FIQ, or External abort occurs.
- Whether the CPSR.F or CPSR.A bits can be modified when SCR.NS==1.

Configurations

AArch32 System register SCR[31:0] can be mapped to AArch64 System register SCR_EL3[31:0], but this is not architecturally mandated.

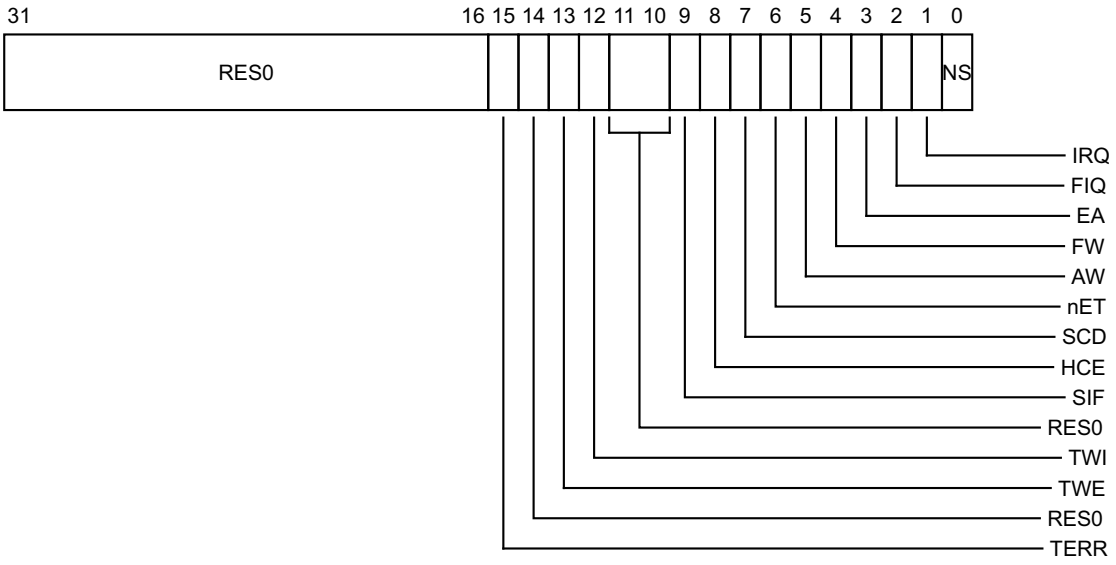
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to SCR are UNKNOWN.

Attributes

SCR is a 32-bit register.

Field descriptions

The SCR bit assignments are:



Bits [31:16]

Reserved, RES0.

TERR, bit [15]

When RAS is implemented:

Trap Error record accesses. Generate a Monitor Trap exception on accesses to the following registers from modes other than Monitor mode:

ERRIDR, ERRSELR, ERXADDR, ERXADDR2, ERXCTLR, ERXCTLR2, ERXFR, ERXFR2, ERXMISC0, ERXMISC1, ERXMISC2, ERXMISC3, and ERXSTATUS. When ARMv8.4-RAS is implemented, ERXMISC4, ERXMISC5, ERXMISC6, ERXMISC7.

`0b0` This control does not cause any instructions to be trapped.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` Accesses to the specified registers from modes other than Monitor mode generate a Monitor Trap exception.

In a system where the PE resets into EL3, this field resets to `0`.

Otherwise:

Reserved, RES0.

Bit [14]

Reserved, RES0.

TWE, bit [13]

Traps WFE instructions to Monitor mode.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Any attempt to execute a WFE instruction in any mode other than Monitor mode is trapped to Monitor mode, if the instruction would otherwise have caused the PE to enter a low-power state and the attempted execution does not generate an exception that is taken to EL1 or EL2 by SCTLR.nTWE or HCR.TWE.

Any exception that is taken to EL1 or to EL2 has priority over this trap.

The attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

Note

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

In a system where the PE resets into EL3, this field resets to `0`.

TWI, bit [12]

Traps WFI instructions to Monitor mode.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Any attempt to execute a WFI instruction in any mode other than Monitor mode is trapped to Monitor mode, if the instruction would otherwise have caused the PE to enter a low-power state and the attempted execution does not generate an exception that is taken to EL1 or EL2 by SCTLR.nTWI or HCR.TWI.

Any exception that is taken to EL1 or to EL2 has priority over this trap.

The attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

Note

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

In a system where the PE resets into EL3, this field resets to `0`.

Bits [11:10]

Reserved, RES0.

SIF, bit [9]

Secure instruction fetch. When the PE is in Secure state, this bit disables instruction fetch from Non-secure memory. The possible values for this bit are:

`0b0` Secure state instruction fetches from Non-secure memory are permitted.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` Secure state instruction fetches from Non-secure memory are not permitted.

This bit is permitted to be cached in a TLB.

In a system where the PE resets into EL3, this field resets to `0`.

HCE, bit [8]

Hypervisor Call instruction enable. If EL2 is implemented, enables execution of HVC instructions at Non-secure EL1 and EL2.

`0b0` HVC instructions are:

- UNDEFINED at Non-secure EL1. The Undefined Instruction exception is taken from PL1 to PL1.
- UNPREDICTABLE at EL2. Behavior is one of the following:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.

`0b1` HVC instructions are enabled at Non-secure EL1 and EL2.

Note

HVC instructions are always UNDEFINED at EL0 and in Secure state.

If EL2 is not implemented, this bit is RES0 and HVC is UNDEFINED.

In a system where the PE resets into EL3, this field resets to `0`.

SCD, bit [7]

Secure Monitor Call disable. Disables SMC instructions.

`0b0` SMC instructions are enabled.

`0b1` In Non-secure state, SMC instructions are UNDEFINED. The Undefined Instruction exception is taken from the current Exception level to the current Exception level.

In Secure state, behavior is one of the following:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.

Note

SMC instructions are always UNDEFINED at PL0.

In a system where the PE resets into EL3, this field resets to `0`.

nET, bit [6]

Not Early Termination. This bit disables early termination. The possible values of this bit are:

`0b0` Early termination permitted. Execution time of data operations can depend on the data values.

`0b1` Disable early termination. The number of cycles required for data operations is forced to be independent of the data values.

This IMPLEMENTATION DEFINED mechanism can disable data dependent timing optimizations from multiplies and data operations. It can provide system support against information leakage that might be exploited by timing correlation types of attack.

On implementations that do not support early termination or do not support disabling early termination, this bit is RES0.

In a system where the PE resets into EL3, this field resets to `0`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

AW, bit [5]

When the value of SCR.EA is 1 and the value of HCR.AMO is 0, this bit controls whether CPSR.A masks an External abort taken from Non-secure state, and the possible values of this bit are:

`0b0` External aborts taken from Non-secure state are not masked by CPSR.A, and are taken to EL3.

External aborts taken from Secure state are masked by CPSR.A.

`0b1` External aborts taken from either Security state are masked by CPSR.A. When CPSR.A is 0, the abort is taken to EL3.

When SCR.EA is 0 or HCR.AMO is 1, this bit has no effect.

In a system where the PE resets into EL3, this field resets to `0`.

FW, bit [4]

When the value of SCR.FIQ is 1 and the value of HCR.FMO is 0, this bit controls whether CPSR.F masks an FIQ interrupt taken from Non-secure state, and the possible values of this bit are:

`0b0` An FIQ taken from Non-secure state is not masked by CPSR.F, and is taken to EL3.

An FIQ taken from Secure state is masked by CPSR.F.

`0b1` An FIQ taken from either Security state is masked by CPSR.F. When CPSR.F is 0, the FIQ is taken to EL3.

When SCR.FIQ is 0 or HCR.FMO is 1, this bit has no effect.

In a system where the PE resets into EL3, this field resets to `0`.

EA, bit [3]

External Abort handler. This bit controls which mode takes External aborts. The possible values of this bit are:

`0b0` External aborts taken to Abort mode.

`0b1` External aborts taken to Monitor mode.

In a system where the PE resets into EL3, this field resets to `0`.

FIQ, bit [2]

FIQ handler. This bit controls which mode takes FIQ exceptions. The possible values of this bit are:

`0b0` FIQs taken to FIQ mode.

`0b1` FIQs taken to Monitor mode.

In a system where the PE resets into EL3, this field resets to `0`.

IRQ, bit [1]

IRQ handler. This bit controls which mode takes IRQ exceptions. The possible values of this bit are:

`0b0` IRQs taken to IRQ mode.

`0b1` IRQs taken to Monitor mode.

In a system where the PE resets into EL3, this field resets to `0`.

NS, bit [0]

Non-secure bit. Except when the PE is in Monitor mode, this bit determines the Security state of the PE:

`0b0` PE is in Secure state.

`0b1` PE is in Non-secure state.

If the HCR.TGE bit is set, an attempt to change from a Secure PL1 mode to a Non-secure EL1 mode by changing the SCR.NS bit from 0 to 1 results in the SCR.NS bit remaining as 0.

In a system where the PE resets into EL3, this field resets to `0`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the SCR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif !ELUsingAArch32(EL2) && SCR_EL3.<NS,EEL2> == '01' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    return SCR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif !ELUsingAArch32(EL2) && SCR_EL3.<NS,EEL2> == '01' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    SCR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.126 SCTLR, System Control Register

The SCTLR characteristics are:

Purpose

Provides the top level control of the system, including its memory system.

Configurations

AArch32 System register SCTLR[31:0] is architecturally mapped to AArch64 System register SCTLR_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to SCTLR are UNKNOWN.

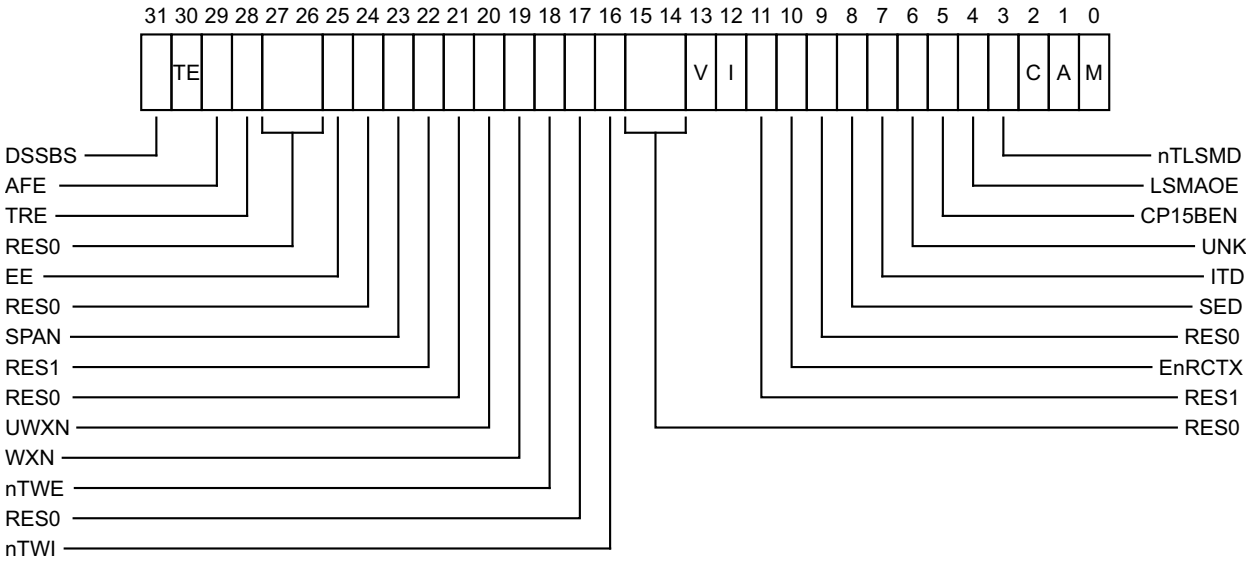
Some bits in the register are read-only. These bits relate to non-configurable features of an implementation, and are provided for compatibility with previous versions of the architecture.

Attributes

SCTLR is a 32-bit register.

Field descriptions

The SCTLR bit assignments are:



DSSBS, bit [31]

When ARMv8.0-SSBS is implemented:

Default PSTATE.SSBS value on Exception Entry. The defined values are:

`0b0` PSTATE.SSBS is set to 0 on an exception to any mode in this security state except Hyp mode

`0b1` PSTATE.SSBS is set to 1 on an exception to any mode in this security state except Hyp mode

Note

When EL3 is implemented and is using AArch32, this bit is banked between the two Security states.

This field resets to an IMPLEMENTATION DEFINED value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

Reserved, RES0.

TE, bit [30]

T32 Exception Enable. This bit controls whether exceptions to an Exception Level that is executing at PL1 are taken to A32 or T32 state:

`0b0` Exceptions, including reset, taken to A32 state.

`0b1` Exceptions, including reset, taken to T32 state.

This field resets to an IMPLEMENTATION DEFINED choice between:

- `0`.
- a value determined by an input configuration signal.

AFE, bit [29]

Access Flag Enable. When using the Short-descriptor translation table format for the PL1&0 translation regime, this bit enables use of the AP[0] bit in the translation descriptors as the Access flag, and restricts access permissions in the translation descriptors to the simplified model. The possible values of this bit are:

`0b0` In the translation table descriptors, AP[0] is an access permissions bit. The full range of access permissions is supported. No Access flag is implemented.

`0b1` In the translation table descriptors, AP[0] is the Access flag. Only the simplified model for access permissions is supported.

When using the Long-descriptor translation table format, the VMSA behaves as if this bit is set to 1, regardless of the value of this bit.

The AFE bit is permitted to be cached in a TLB.

This field resets to `0`.

TRE, bit [28]

TEX remap enable. This bit enables remapping of the TEX[2:1] bits in the PL1&0 translation regime for use as two translation table bits that can be managed by the operating system. Enabling this remapping also changes the scheme used to describe the memory region attributes in the VMSA. The possible values of this bit are:

`0b0` TEX remap disabled. TEX[2:0] are used, with the C and B bits, to describe the memory region attributes.

`0b1` TEX remap enabled. TEX[2:1] are reassigned for use as bits managed by the operating system. The TEX[0], C, and B bits are used to describe the memory region attributes, with the MMU remap registers.

When the value of TTBCR.EAE is 1, this bit is RES1.

The TRE bit is permitted to be cached in a TLB.

This field resets to `0`.

Bits [27:26]

Reserved, RES0.

EE, bit [25]

The value of the PSTATE.E bit on branch to an exception vector or coming out of reset, and the endianness of stage 1 translation table walks in the PL1&0 translation regime.

The possible values of this bit are:

`0b0` Little-endian. PSTATE.E is cleared to 0 on taking an exception or coming out of reset. Stage 1 translation table walks in the PL1&0 translation regime are little-endian.

`0b1` Big-endian. PSTATE.E is set to 1 on taking an exception or coming out of reset. Stage 1 translation table walks in the PL1&0 translation regime are big-endian.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If an implementation does not provide Big-endian support for data accesses at Exception Levels higher than EL0, this bit is RES0.

If an implementation does not provide Little-endian support for data accesses at Exception Levels higher than EL0, this bit is RES1.

This field resets to an IMPLEMENTATION DEFINED choice between:

- `0`.
- a value determined by an input configuration signal.

Bit [24]

Reserved, RES0.

SPAN, bit [23]

When ARMv8.1-PAN is implemented:

Set Privileged Access Never, on taking an exception to EL1 from either Secure or Non-secure state, or to EL3 from Secure state when EL3 is using AArch32.

`0b0` CPSR.PAN is set to 1 in the following situations:

- In Non-secure state, on taking an exception to EL1.
- In Secure state, when EL3 is using AArch64, on taking an exception to EL1.
- In Secure state, when EL3 is using AArch32, on taking an exception to EL3.

`0b1` The value of CPSR.PAN is left unchanged on taking an exception to EL1.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

Bit [22]

Reserved, RES1.

Bit [21]

Reserved, RES0.

UWXN, bit [20]

Unprivileged write permission implies PL1 XN (Execute-never). This bit can force all memory regions that are writable at PL0 to be treated as XN for accesses from software executing at PL1. The possible values of this bit are:

`0b0` This control has no effect on memory access permissions.

`0b1` Any region that is writable at PL0 forced to XN for accesses from software executing at PL1.

The UWXN bit is permitted to be cached in a TLB.

This field resets to `0`.

WXN, bit [19]

Write permission implies XN (Execute-never). For the PL1&0 translation regime, this bit can force all memory regions that are writable to be treated as XN. The possible values of this bit are:

`0b0` This control has no effect on memory access permissions.

`0b1` Any region that is writable in the PL1&0 translation regime is forced to XN for accesses from software executing at PL1 or PL0.

This bit applies only when SCTLR.M bit is set.

The WXN bit is permitted to be cached in a TLB.

This field resets to `0`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

nTWE, bit [18]

Traps EL0 execution of WFE instructions to Undefined mode.

`0b0` Any attempt to execute a WFE instruction at EL0 is trapped to Undefined mode, if the instruction would otherwise have caused the PE to enter a low-power state.

`0b1` This control does not cause any instructions to be trapped.

The attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

Note

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

This field resets to `1`.

Bit [17]

Reserved, RES0.

nTWI, bit [16]

Traps EL0 execution of WFI instructions to Undefined mode.

`0b0` Any attempt to execute a WFI instruction at EL0 is trapped to Undefined mode, if the instruction would otherwise have caused the PE to enter a low-power state.

`0b1` This control does not cause any instructions to be trapped.

The attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

Note

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

This field resets to `1`.

Bits [15:14]

Reserved, RES0.

V, bit [13]

Vectors bit. This bit selects the base address of the exception vectors for exceptions taken to a PE mode other than Monitor mode or Hyp mode:

`0b0` Normal exception vectors. Base address is held in VBAR.

`0b1` High exception vectors (Hivecs), base address `0xFFFF0000`. This base address cannot be remapped.

This field resets to an IMPLEMENTATION DEFINED choice between:

- `0`.
- a value determined by an input configuration signal.

I, bit [12]

Instruction access Cacheability control, for accesses at EL1 and EL0:

`0b0` All instruction access to Normal memory from PL1 and PL0 are Non-cacheable for all levels of instruction and unified cache.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the value of SCTLR.M is 0, instruction accesses from stage 1 of the PL1&0 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.

`0b1` All instruction access to Normal memory from PL1 and PL0 can be cached at all levels of instruction and unified cache.

If the value of SCTLR.M is 0, instruction accesses from stage 1 of the PL1&0 translation regime are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.

Instruction accesses to Normal memory from EL1 and EL0 are Cacheable regardless of the value of the SCTLR.I bit if either:

- EL2 is using AArch32 and the value of HCR.DC is 1.
- EL2 is using AArch64 and the value of HCR_EL2.DC is 1.

This field resets to `0`.

Bit [11]

Reserved, RES1.

EnRCTX, bit [10]

When ARMv8.0-CSV2 is implemented:

Enable EL0 Access to the AArch32 CFPRCTX, DVPRCTX and CPPRCTX instructions. The defined values are:

`0b0` EL0 access to these instructions is disabled, and these instructions are trapped to EL1.

`0b1` EL0 access to these instructions is enabled.

Note

When EL3 is implemented and is using AArch32, this bit is banked between the two Security states.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [9]

Reserved, RES0.

SED, bit [8]

SETEND instruction disable. Disables SETEND instructions at PL0 and PL1.

`0b0` SETEND instruction execution is enabled at PL0 and PL1.

`0b1` SETEND instructions are UNDEFINED at PL0 and PL1.

If the implementation does not support mixed-endian operation at any Exception level, this bit is RES1.

This field resets to `0`.

ITD, bit [7]

IT Disable. Disables some uses of IT instructions at PL1 and PL0.

`0b0` All IT instruction functionality is enabled at PL1 and PL0.

`0b1` Any attempt at PL1 or PL0 to execute any of the following is UNDEFINED:

- All encodings of the IT instruction with hw1[3:0]!=1000.
- All encodings of the subsequent instruction with the following values for hw1:
  - 11xxxxxxxxxxxxxx: All 32-bit instructions, and the 16-bit instructions B, UDF, SVC, LDM, and STM.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— 1011xxxxxxxxxxxx: All instructions in *Miscellaneous 16-bit instructions* on page F3-4115.

— 10100xxxxxxxxxxx: ADD Rd, PC, #imm

— 01001xxxxxxxxxxx: LDR Rd, [PC, #imm]

— 0100x1xxx1111xxx: ADD Rdn, PC; CMP Rn, PC; MOV Rd, PC; BX PC; BLX PC.

— 010001xx1xxxx111: ADD PC, Rm; CMP PC, Rm; MOV PC, Rm. This pattern also covers unpredictable cases with BLX Rn.

These instructions are always UNDEFINED, regardless of whether they would pass or fail the condition code check that applies to them as a result of being in an IT block.

It is IMPLEMENTATION DEFINED whether the IT instruction is treated as:

- A 16-bit instruction, that can only be followed by another 16-bit instruction.
- The first half of a 32-bit instruction.

This means that, for the situations that are UNDEFINED, either the second 16-bit instruction or the 32-bit instruction is UNDEFINED.

An implementation might vary dynamically as to whether IT is treated as a 16-bit instruction or the first half of a 32-bit instruction.

If an instruction in an active IT block that would be disabled by this field sets this field to 1 then behavior is CONSTRAINED UNPREDICTABLE. For more information see *Changes to an ITD control by an instruction in an IT block* on page E1-3958.

ITD is optional, but if it is implemented in the SCTLR then it must also be implemented in the SCTLR_EL1. If it is not implemented then this bit is RAZ/WI.

This field resets to `0`.

UNK, bit [6]

Writes to this bit are IGNORED. Reads of this bit return an UNKNOWN value.

This field resets to an architecturally UNKNOWN value.

CP15BEN, bit [5]

System instruction memory barrier enable. Enables accesses to the DMB, DSB, and ISB System instructions in the (coproc==`0b1111`) encoding space from PL1 and PL0:

`0b0` PL0 and PL1 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is UNDEFINED.

`0b1` PL0 and PL1 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is enabled.

CP15BEN is optional, but if it is implemented in the SCTLR then it must also be implemented in the SCTLR_EL1. If it is not implemented then this bit is RAO/WI.

This field resets to `1`.

LSMAOE, bit [4]

When ARMv8.2-LSMAOC is implemented:

Load Multiple and Store Multiple Atomicity and Ordering Enable.

`0b0` For all memory accesses at EL1 or EL0, A32 and T32 Load Multiple and Store Multiple can have an interrupt taken during the sequence memory accesses, and the memory accesses are not required to be ordered.

`0b1` The ordering and interrupt behavior of A32 and T32 Load Multiple and Store Multiple at EL1 or EL0 is as defined for Armv8.0.

This bit is permitted to be cached in a TLB.

This field resets to `1`.

Otherwise:

Reserved, RES1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

nTLSMD, bit [3]

When ARMv8.2-LSMAOC is implemented:

No Trap Load Multiple and Store Multiple to Device-nGRE/Device-nGnRE/Device-nGnRnE memory.

`0b0` All memory accesses by A32 and T32 Load Multiple and Store Multiple at EL1 or EL0 that are marked at stage 1 as Device-nGRE/Device-nGnRE/Device-nGnRnE memory are trapped and generate a stage 1 Alignment fault.

`0b1` All memory accesses by A32 and T32 Load Multiple and Store Multiple at EL1 or EL0 that are marked at stage 1 as Device-nGRE/Device-nGnRE/Device-nGnRnE memory are not trapped.

This bit is permitted to be cached in a TLB.

This field resets to `1`.

Otherwise:

Reserved, RES1.

C, bit [2]

Cacheability control, for data accesses at EL1 and EL0:

`0b0` All data access to Normal memory from PL1 and PL0, and all accesses to the PL1&0 stage 1 translation tables, are Non-cacheable for all levels of data and unified cache.

`0b1` All data access to Normal memory from PL1 and PL0, and all accesses to the PL1&0 stage 1 translation tables, can be cached at all levels of data and unified cache.

The PE ignores SCLTR.C for Non-secure state and data accesses to Normal memory from EL1 and EL0 are Cacheable if either:

- EL2 is using AArch32 and the value of HCR.DC is 1.
- EL2 is using AArch64 and the value of HCR_EL2.DC is 1.

This field resets to `0`.

A, bit [1]

Alignment check enable. This is the enable bit for Alignment fault checking at PL1 and PL0:

`0b0` Alignment fault checking disabled when executing at PL1 or PL0.

Instructions that load or store one or more registers, other than load/store exclusive and load-acquire/store-release, do not check that the address being accessed is aligned to the size of the data element(s) being accessed.

`0b1` Alignment fault checking enabled when executing at PL1 or PL0.

All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element(s) being accessed. If this check fails it causes an Alignment fault, which is taken as a Data Abort exception.

Load/store exclusive and load-acquire/store-release instructions have an alignment check regardless of the value of the A bit.

This field resets to `0`.

M, bit [0]

MMU enable for EL1 and EL0 stage 1 address translation. Possible values of this bit are:

`0b0` EL1 and EL0 stage 1 address translation disabled.

See the SCTLR.I field for the behavior of instruction accesses to Normal memory.

`0b1` EL1 and EL0 stage 1 address translation enabled.

In the Non-secure state the PE behaves as if the value of the SCTLR.M field is 0 for all purposes other than returning the value of a direct read of the field if either:

- EL2 is using AArch32 and the value of HCR.{DC, TGE} is not {0, 0}.
- EL2 is using AArch64 and the value of HCR_EL2.{DC, TGE} is not {0, 0}.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to 0.

Accessing the SCTLR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TRVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return SCTLR_NS;
    else
        return SCTLR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return SCTLR_NS;
    else
        return SCTLR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return SCTLR_S;
    else
        return SCTLR_NS;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        SCTLR_NS = R[t];
    else
        SCTLR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        SCTLR_NS = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        SCTLR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' && CP15SDISABLE == HIGH then
        UNDEFINED;
    elsif SCR.NS == '0' && CP15SDISABLE2 == HIGH then
        UNDEFINED;
    else
        if SCR.NS == '0' then
            SCTLR_S = R[t];
        else
            SCTLR_NS = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.127 SPSR, Saved Program Status Register

The SPSR characteristics are:

Purpose

Holds the saved process state for the current mode.

Configurations

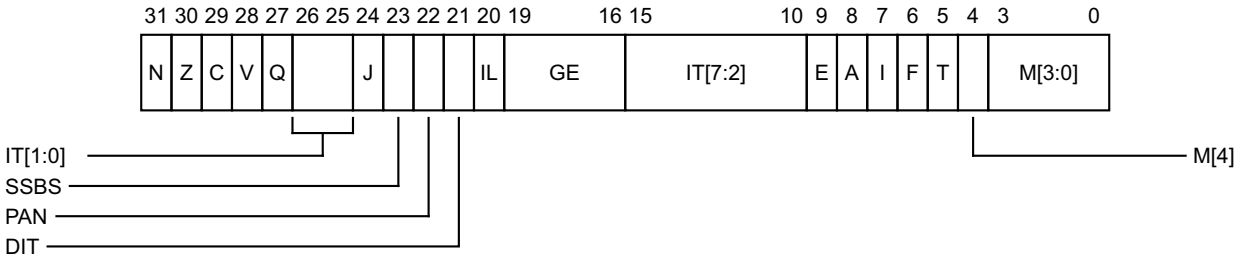
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to SPSR are UNKNOWN.

Attributes

SPSR is a 32-bit register.

Field descriptions

The SPSR bit assignments are:



N, bit [31]

Set to the value of PSTATE.N on taking an exception to the current mode, and copied to PSTATE.N on executing an exception return operation in the current mode.

Z, bit [30]

Set to the value of PSTATE.Z on taking an exception to the current mode, and copied to PSTATE.Z on executing an exception return operation in the current mode.

C, bit [29]

Set to the value of PSTATE.C on taking an exception to the current mode, and copied to PSTATE.C on executing an exception return operation in the current mode.

V, bit [28]

Set to the value of PSTATE.V on taking an exception to the current mode, and copied to PSTATE.V on executing an exception return operation in the current mode.

Q, bit [27]

Set to the value of PSTATE.Q on taking an exception to the current mode, and copied to PSTATE.Q on executing an exception return operation in the current mode.

IT[1:0], bits [26:25]

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

SSBS, bit [23]

When ARMv8.0-SSBS is implemented:

Speculative Store Bypass Safe. This bit is set to the value of PSTATE.SSBS on taking an exception to the current mode, and copied to PSTATE.SSBS on executing an exception return operation in the current mode.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When ARMv8.1-PAN is implemented:

Privileged Access Never. This bit is set to the value of PSTATE.PAN on taking an exception to the current mode, and copied to PSTATE.PAN on executing an exception return operation in the current mode.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When ARMv8.4-DIT is implemented:

Data Independent Timing. This bit is set to the value of PSTATE.DIT on taking an exception to the current mode, and copied to PSTATE.DIT on executing an exception return operation in the current mode.

Otherwise:

Reserved, RES0.

IL, bit [20]

Illegal Execution state bit. Shows the value of PSTATE.IL immediately before the exception was taken.

GE, bits [19:16]

Greater than or Equal flags, for parallel addition and subtraction.

IT[7:2], bits [15:10]

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is `0b00000000` when no IT block is active.

E, bit [9]

Endianness state bit. Controls the load and store endianness for data accesses:

`0b0` Little-endian operation

`0b1` Big-endian operation.

Instruction fetches ignore this bit.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When the reset value of the SCTLR.EE bit is defined by a configuration input signal, that value also applies to the CPSR.E bit on reset, and therefore applies to software execution from reset.

A, bit [8]

SError interrupt mask bit.

`0b0` Exception not masked.

`0b1` Exception masked.

I, bit [7]

IRQ mask bit.

`0b0` Exception not masked.

`0b1` Exception masked.

F, bit [6]

FIQ mask bit.

`0b0` Exception not masked.

`0b1` Exception masked.

T, bit [5]

T32 Instruction set state bit. Determines the AArch32 instruction set state that the exception was taken from.

`0b0` Taken from A32 state.

`0b1` Taken from T32 state.

M[4], bit [4]

Execution state that the exception was taken from.

`0b1` Exception taken from AArch32.

M[3:0], bits [3:0]

AArch32 mode that an exception was taken from.

`0b0000` User.

`0b0001` FIQ.

`0b0010` IRQ.

`0b0011` Supervisor.

`0b0110` **When EL3 is implemented and EL3 is capable of using AArch32** Monitor (only valid in Secure state, if EL3 is implemented and can use AArch32).

`0b0111` Abort.

`0b1010` Hyp.

`0b1011` Undefined.

`0b1111` System.

Other values are reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.128 SPSR_abt, Saved Program Status Register (Abort mode)

The SPSR_abt characteristics are:

Purpose

Holds the saved process state when an exception is taken to Abort mode.

Configurations

AArch32 System register SPSR_abt[31:0] is architecturally mapped to AArch64 System register SPSR_abt[31:0].

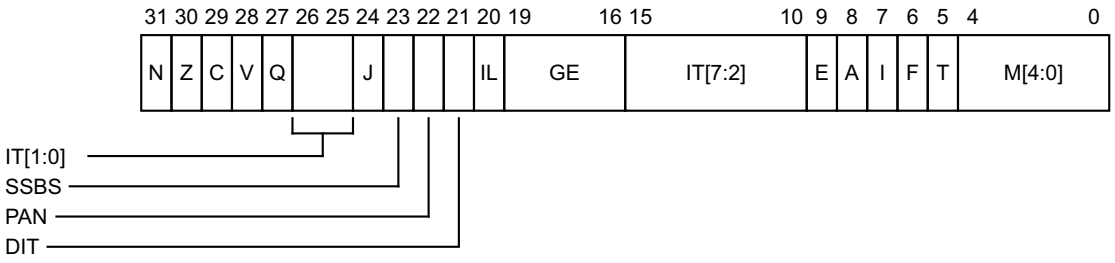
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to SPSR_abt are UNKNOWN.

Attributes

SPSR_abt is a 32-bit register.

Field descriptions

The SPSR_abt bit assignments are:



N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to Abort mode, and copied to PSTATE.N on executing an exception return operation in Abort mode.

This field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to Abort mode, and copied to PSTATE.Z on executing an exception return operation in Abort mode.

This field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to Abort mode, and copied to PSTATE.C on executing an exception return operation in Abort mode.

This field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to Abort mode, and copied to PSTATE.V on executing an exception return operation in Abort mode.

This field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to Abort mode, and copied to PSTATE.Q on executing an exception return operation in Abort mode.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

IT[1:0], bits [26:25]

If-Then. Set to the value of PSTATE.IT[1:0] on taking an exception to Abort mode, and copied to PSTATE.IT[1:0] on executing an exception return operation in Abort mode.

On executing an exception return operation in Abort mode SPSR_abt.IT must contain a value that is valid for the instruction being returned to.

This field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.

Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When ARMv8.0-SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to Abort mode, and copied to PSTATE.SSBS on executing an exception return operation in Abort mode.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When ARMv8.1-PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to Abort mode, and copied to PSTATE.PAN on executing an exception return operation in Abort mode.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When ARMv8.4-DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to Abort mode, and copied to PSTATE.DIT on executing an exception return operation in Abort mode.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to Abort mode, and copied to PSTATE.IL on executing an exception return operation in Abort mode.

This field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to Abort mode, and copied to PSTATE.GE on executing an exception return operation in Abort mode.

This field resets to an architecturally UNKNOWN value.

IT[7:2], bits [15:10]

If-Then. Set to the value of PSTATE.IT[7:2] on taking an exception to Abort mode, and copied to PSTATE.IT[7:2] on executing an exception return operation in Abort mode.

SPSR_abt.IT must contain a value that is valid for the instruction being returned to.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to Abort mode, and copied to PSTATE.E on executing an exception return operation in Abort mode.

If the implementation does not support big-endian operation, SPSR_abt.E is RES0. If the implementation does not support little-endian operation, SPSR_abt.E is RES1. On executing an exception return operation in Abort mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_abt.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_abt.E is RES1.

This field resets to an architecturally UNKNOWN value.

A, bit [8]

SError interrupt mask. Set to the value of PSTATE.A on taking an exception to Abort mode, and copied to PSTATE.A on executing an exception return operation in Abort mode.

This field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to Abort mode, and copied to PSTATE.I on executing an exception return operation in Abort mode.

This field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to Abort mode, and copied to PSTATE.F on executing an exception return operation in Abort mode.

This field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to Abort mode, and copied to PSTATE.T on executing an exception return operation in Abort mode.

This field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to Abort mode, and copied to PSTATE.M[4:0] on executing an exception return operation in Abort mode.

`0b10000` User.

`0b10001` FIQ.

`0b10010` IRQ.

`0b10011` Supervisor.

`0b10111` Abort.

`0b11011` Undefined.

`0b11111` System.

Other values are reserved. If SPSR_abt.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in Abort mode is an illegal return event, as described in *Illegal return events from AArch32 state* on page G1-5726.

This field resets to an architecturally UNKNOWN value.

Accessing the SPSR_abt

SPSR_abt is accessible in all modes other than User mode and Abort mode.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS{<c>}{<q>} <Rd>, SPSR_abt

R	M	M1
0b1	0b1	0b0100

MSR{<c>}{<q>} SPSR_abt, <Rn>

R	M	M1
0b1	0b1	0b0100

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.129 SPSR_fiq, Saved Program Status Register (FIQ mode)

The SPSR_fiq characteristics are:

Purpose

Holds the saved process state when an exception is taken to FIQ mode.

Configurations

AArch32 System register SPSR_fiq[31:0] is architecturally mapped to AArch64 System register SPSR_fiq[31:0].

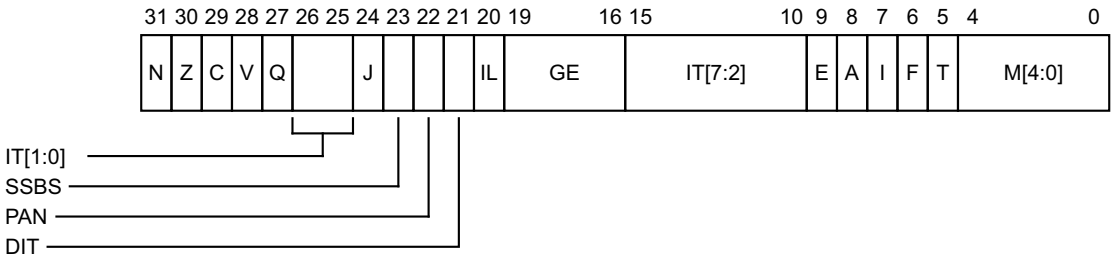
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to SPSR_fiq are UNKNOWN.

Attributes

SPSR_fiq is a 32-bit register.

Field descriptions

The SPSR_fiq bit assignments are:



N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to FIQ mode, and copied to PSTATE.N on executing an exception return operation in FIQ mode.

This field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to FIQ mode, and copied to PSTATE.Z on executing an exception return operation in FIQ mode.

This field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to FIQ mode, and copied to PSTATE.C on executing an exception return operation in FIQ mode.

This field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to FIQ mode, and copied to PSTATE.V on executing an exception return operation in FIQ mode.

This field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to FIQ mode, and copied to PSTATE.Q on executing an exception return operation in FIQ mode.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

IT[1:0], bits [26:25]

If-Then. Set to the value of PSTATE.IT[1:0] on taking an exception to FIQ mode, and copied to PSTATE.IT[1:0] on executing an exception return operation in FIQ mode.

On executing an exception return operation in FIQ mode SPSR_fiq.IT must contain a value that is valid for the instruction being returned to.

This field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.

Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When ARMv8.0-SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to FIQ mode, and copied to PSTATE.SSBS on executing an exception return operation in FIQ mode.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When ARMv8.1-PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to FIQ mode, and copied to PSTATE.PAN on executing an exception return operation in FIQ mode.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When ARMv8.4-DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to FIQ mode, and copied to PSTATE.DIT on executing an exception return operation in FIQ mode.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to FIQ mode, and copied to PSTATE.IL on executing an exception return operation in FIQ mode.

This field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to FIQ mode, and copied to PSTATE.GE on executing an exception return operation in FIQ mode.

This field resets to an architecturally UNKNOWN value.

IT[7:2], bits [15:10]

If-Then. Set to the value of PSTATE.IT[7:2] on taking an exception to FIQ mode, and copied to PSTATE.IT[7:2] on executing an exception return operation in FIQ mode.

SPSR_fiq.IT must contain a value that is valid for the instruction being returned to.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to FIQ mode, and copied to PSTATE.E on executing an exception return operation in FIQ mode.

If the implementation does not support big-endian operation, SPSR_fiq.E is RES0. If the implementation does not support little-endian operation, SPSR_fiq.E is RES1. On executing an exception return operation in FIQ mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_fiq.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_fiq.E is RES1.

This field resets to an architecturally UNKNOWN value.

A, bit [8]

SError interrupt mask. Set to the value of PSTATE.A on taking an exception to FIQ mode, and copied to PSTATE.A on executing an exception return operation in FIQ mode.

This field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to FIQ mode, and copied to PSTATE.I on executing an exception return operation in FIQ mode.

This field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to FIQ mode, and copied to PSTATE.F on executing an exception return operation in FIQ mode.

This field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to FIQ mode, and copied to PSTATE.T on executing an exception return operation in FIQ mode.

This field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to FIQ mode, and copied to PSTATE.M[4:0] on executing an exception return operation in FIQ mode.

`0b10000` User.

`0b10001` FIQ.

`0b10010` IRQ.

`0b10011` Supervisor.

`0b10111` Abort.

`0b11011` Undefined.

`0b11111` System.

Other values are reserved. If SPSR_fiq.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in FIQ mode is an illegal return event, as described in *Illegal return events from AArch32 state* on page G1-5726.

This field resets to an architecturally UNKNOWN value.

Accessing the SPSR_fiq

SPSR_fiq is accessible in all modes other than User mode and FIQ mode.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS{<c>}{<q>} <Rd>, SPSR_fiq

R	M	M1
0b1	0b0	0b1110

MSR{<c>}{<q>} SPSR_fiq, <Rn>

R	M	M1
0b1	0b0	0b1110

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.130 SPSR_hyp, Saved Program Status Register (Hyp mode)

The SPSR_hyp characteristics are:

Purpose

Holds the saved process state when an exception is taken to Hyp mode.

Configurations

AArch32 System register SPSR_hyp[31:0] is architecturally mapped to AArch64 System register SPSR_EL2[31:0].

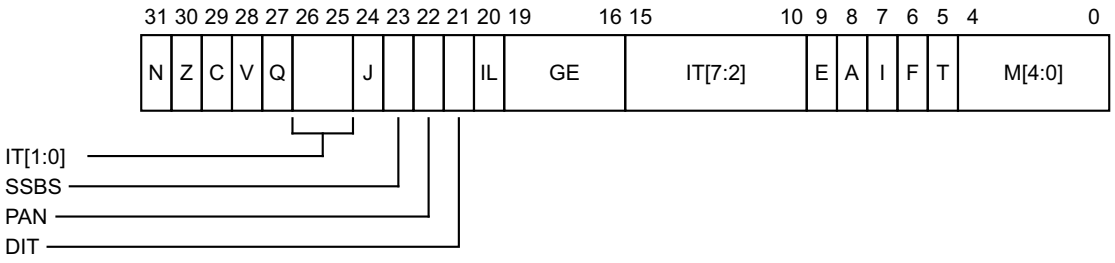
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to SPSR_hyp are UNKNOWN.

Attributes

SPSR_hyp is a 32-bit register.

Field descriptions

The SPSR_hyp bit assignments are:



N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to Hyp mode, and copied to PSTATE.N on executing an exception return operation in Hyp mode.

This field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to Hyp mode, and copied to PSTATE.Z on executing an exception return operation in Hyp mode.

This field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to Hyp mode, and copied to PSTATE.C on executing an exception return operation in Hyp mode.

This field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to Hyp mode, and copied to PSTATE.V on executing an exception return operation in Hyp mode.

This field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to Hyp mode, and copied to PSTATE.Q on executing an exception return operation in Hyp mode.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

IT[1:0], bits [26:25]

If-Then. Set to the value of PSTATE.IT[1:0] on taking an exception to Hyp mode, and copied to PSTATE.IT[1:0] on executing an exception return operation in Hyp mode.

On executing an exception return operation in Hyp mode SPSR_hyp.IT must contain a value that is valid for the instruction being returned to.

This field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.

Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When ARMv8.0-SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to Hyp mode, and copied to PSTATE.SSBS on executing an exception return operation in Hyp mode.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When ARMv8.1-PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to Hyp mode, and copied to PSTATE.PAN on executing an exception return operation in Hyp mode.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When ARMv8.4-DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to Hyp mode, and copied to PSTATE.DIT on executing an exception return operation in Hyp mode.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to Hyp mode, and copied to PSTATE.IL on executing an exception return operation in Hyp mode.

This field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to Hyp mode, and copied to PSTATE.GE on executing an exception return operation in Hyp mode.

This field resets to an architecturally UNKNOWN value.

IT[7:2], bits [15:10]

If-Then. Set to the value of PSTATE.IT[7:2] on taking an exception to Hyp mode, and copied to PSTATE.IT[7:2] on executing an exception return operation in Hyp mode.

SPSR_hyp.IT must contain a value that is valid for the instruction being returned to.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to Hyp mode, and copied to PSTATE.E on executing an exception return operation in Hyp mode.

If the implementation does not support big-endian operation, SPSR_hyp.E is RES0. If the implementation does not support little-endian operation, SPSR_hyp.E is RES1. On executing an exception return operation in Hyp mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_hyp.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_hyp.E is RES1.

This field resets to an architecturally UNKNOWN value.

A, bit [8]

SError interrupt mask. Set to the value of PSTATE.A on taking an exception to Hyp mode, and copied to PSTATE.A on executing an exception return operation in Hyp mode.

This field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to Hyp mode, and copied to PSTATE.I on executing an exception return operation in Hyp mode.

This field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to Hyp mode, and copied to PSTATE.F on executing an exception return operation in Hyp mode.

This field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to Hyp mode, and copied to PSTATE.T on executing an exception return operation in Hyp mode.

This field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to Hyp mode, and copied to PSTATE.M[4:0] on executing an exception return operation in Hyp mode.

`0b10000` User.

`0b10001` FIQ.

`0b10010` IRQ.

`0b10011` Supervisor.

`0b10111` Abort.

`0b11010` Hyp.

`0b11011` Undefined.

`0b11111` System.

Other values are reserved. If SPSR_hyp.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in Hyp mode is an illegal return event, as described in *Illegal return events from AArch32 state* on page G1-5726.

This field resets to an architecturally UNKNOWN value.

Accessing the SPSR_hyp

SPSR_hyp is accessible only in Monitor mode.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accesses to this register use the following encodings in the System instruction encoding space:

MRS{<c>}{<q>} <Rd>, SPSR_hyp

R	M	M1
0b1	0b1	0b1110

MSR{<c>}{<q>} SPSR_hyp, <Rn>

R	M	M1
0b1	0b1	0b1110

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.131 SPSR_irq, Saved Program Status Register (IRQ mode)

The SPSR_irq characteristics are:

Purpose

Holds the saved process state when an exception is taken to IRQ mode.

Configurations

AArch32 System register SPSR_irq[31:0] is architecturally mapped to AArch64 System register SPSR_irq[31:0].

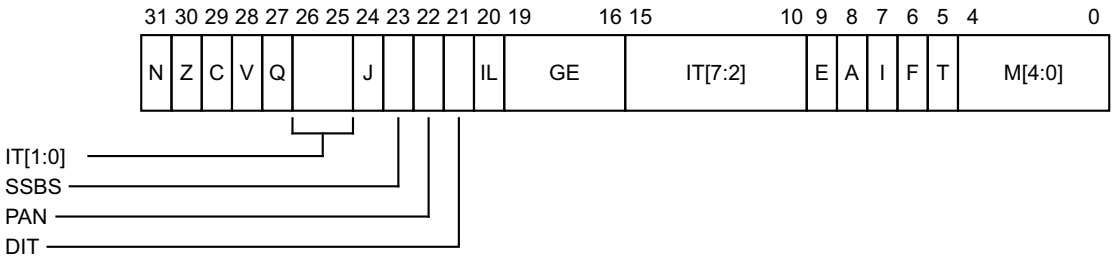
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to SPSR_irq are UNKNOWN.

Attributes

SPSR_irq is a 32-bit register.

Field descriptions

The SPSR_irq bit assignments are:



N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to IRQ mode, and copied to PSTATE.N on executing an exception return operation in IRQ mode.

This field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to IRQ mode, and copied to PSTATE.Z on executing an exception return operation in IRQ mode.

This field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to IRQ mode, and copied to PSTATE.C on executing an exception return operation in IRQ mode.

This field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to IRQ mode, and copied to PSTATE.V on executing an exception return operation in IRQ mode.

This field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to IRQ mode, and copied to PSTATE.Q on executing an exception return operation in IRQ mode.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

IT[1:0], bits [26:25]

If-Then. Set to the value of PSTATE.IT[1:0] on taking an exception to IRQ mode, and copied to PSTATE.IT[1:0] on executing an exception return operation in IRQ mode.

On executing an exception return operation in IRQ mode SPSR_irq.IT must contain a value that is valid for the instruction being returned to.

This field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.

Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When ARMv8.0-SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to IRQ mode, and copied to PSTATE.SSBS on executing an exception return operation in IRQ mode.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When ARMv8.1-PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to IRQ mode, and copied to PSTATE.PAN on executing an exception return operation in IRQ mode.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When ARMv8.4-DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to IRQ mode, and copied to PSTATE.DIT on executing an exception return operation in IRQ mode.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to IRQ mode, and copied to PSTATE.IL on executing an exception return operation in IRQ mode.

This field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to IRQ mode, and copied to PSTATE.GE on executing an exception return operation in IRQ mode.

This field resets to an architecturally UNKNOWN value.

IT[7:2], bits [15:10]

If-Then. Set to the value of PSTATE.IT[7:2] on taking an exception to IRQ mode, and copied to PSTATE.IT[7:2] on executing an exception return operation in IRQ mode.

SPSR_irq.IT must contain a value that is valid for the instruction being returned to.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to IRQ mode, and copied to PSTATE.E on executing an exception return operation in IRQ mode.

If the implementation does not support big-endian operation, SPSR_irq.E is RES0. If the implementation does not support little-endian operation, SPSR_irq.E is RES1. On executing an exception return operation in IRQ mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_irq.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_irq.E is RES1.

This field resets to an architecturally UNKNOWN value.

A, bit [8]

SError interrupt mask. Set to the value of PSTATE.A on taking an exception to IRQ mode, and copied to PSTATE.A on executing an exception return operation in IRQ mode.

This field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to IRQ mode, and copied to PSTATE.I on executing an exception return operation in IRQ mode.

This field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to IRQ mode, and copied to PSTATE.F on executing an exception return operation in IRQ mode.

This field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to IRQ mode, and copied to PSTATE.T on executing an exception return operation in IRQ mode.

This field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to IRQ mode, and copied to PSTATE.M[4:0] on executing an exception return operation in IRQ mode.

`0b10000` User.

`0b10001` FIQ.

`0b10010` IRQ.

`0b10011` Supervisor.

`0b10111` Abort.

`0b11011` Undefined.

`0b11111` System.

Other values are reserved. If SPSR_irq.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in IRQ mode is an illegal return event, as described in *Illegal return events from AArch32 state* on page G1-5726.

This field resets to an architecturally UNKNOWN value.

Accessing the SPSR_irq

SPSR_irq is accessible in all modes other than User mode and IRQ mode.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS{<c>}{<q>} <Rd>, SPSR_irq

R	M	M1
0b1	0b1	0b0000

MSR{<c>}{<q>} SPSR_irq, <Rn>

R	M	M1
0b1	0b1	0b0000

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.132 SPSR_mon, Saved Program Status Register (Monitor mode)

The SPSR_mon characteristics are:

Purpose

Holds the saved process state when an exception is taken to Monitor mode.

Configurations

AArch32 System register SPSR_mon[31:0] can be mapped to AArch64 System register SPSR_EL3[31:0], but this is not architecturally mandated.

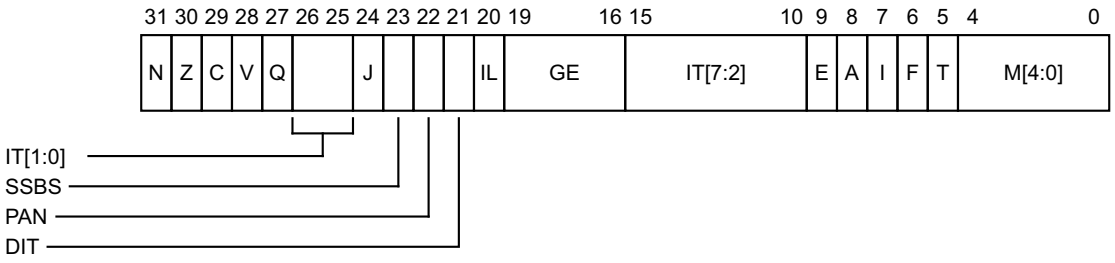
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to SPSR_mon are UNKNOWN.

Attributes

SPSR_mon is a 32-bit register.

Field descriptions

The SPSR_mon bit assignments are:



N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to Monitor mode, and copied to PSTATE.N on executing an exception return operation in Monitor mode.

This field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to Monitor mode, and copied to PSTATE.Z on executing an exception return operation in Monitor mode.

This field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to Monitor mode, and copied to PSTATE.C on executing an exception return operation in Monitor mode.

This field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to Monitor mode, and copied to PSTATE.V on executing an exception return operation in Monitor mode.

This field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to Monitor mode, and copied to PSTATE.Q on executing an exception return operation in Monitor mode.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

IT[1:0], bits [26:25]

If-Then. Set to the value of PSTATE.IT[1:0] on taking an exception to Monitor mode, and copied to PSTATE.IT[1:0] on executing an exception return operation in Monitor mode.

On executing an exception return operation in Monitor mode SPSR_mon.IT must contain a value that is valid for the instruction being returned to.

This field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.

Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When ARMv8.0-SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to Monitor mode, and copied to PSTATE.SSBS on executing an exception return operation in Monitor mode.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When ARMv8.1-PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to Monitor mode, and copied to PSTATE.PAN on executing an exception return operation in Monitor mode.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When ARMv8.4-DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to Monitor mode, and copied to PSTATE.DIT on executing an exception return operation in Monitor mode.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to Monitor mode, and copied to PSTATE.IL on executing an exception return operation in Monitor mode.

This field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to Monitor mode, and copied to PSTATE.GE on executing an exception return operation in Monitor mode.

This field resets to an architecturally UNKNOWN value.

IT[7:2], bits [15:10]

If-Then. Set to the value of PSTATE.IT[7:2] on taking an exception to Monitor mode, and copied to PSTATE.IT[7:2] on executing an exception return operation in Monitor mode.

SPSR_mon.IT must contain a value that is valid for the instruction being returned to.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to Monitor mode, and copied to PSTATE.E on executing an exception return operation in Monitor mode.

If the implementation does not support big-endian operation, SPSR_mon.E is RES0. If the implementation does not support little-endian operation, SPSR_mon.E is RES1. On executing an exception return operation in Monitor mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_mon.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_mon.E is RES1.

This field resets to an architecturally UNKNOWN value.

A, bit [8]

SError interrupt mask. Set to the value of PSTATE.A on taking an exception to Monitor mode, and copied to PSTATE.A on executing an exception return operation in Monitor mode.

This field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to Monitor mode, and copied to PSTATE.I on executing an exception return operation in Monitor mode.

This field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to Monitor mode, and copied to PSTATE.F on executing an exception return operation in Monitor mode.

This field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to Monitor mode, and copied to PSTATE.T on executing an exception return operation in Monitor mode.

This field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to Monitor mode, and copied to PSTATE.M[4:0] on executing an exception return operation in Monitor mode.

`0b10000` User.

`0b10001` FIQ.

`0b10010` IRQ.

`0b10011` Supervisor.

`0b10110` Monitor.

`0b10111` Abort.

`0b11010` Hyp.

`0b11011` Undefined.

`0b11111` System.

Other values are reserved. If SPSR_mon.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in Monitor mode is an illegal return event, as described in *Illegal return events from AArch32 state* on page G1-5726.

This field resets to an architecturally UNKNOWN value.

Accessing the SPSR_mon

SPSR_mon is only accessible in EL3 modes other than Monitor mode.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accesses to this register use the following encodings in the System instruction encoding space:

MRS{<c>}{<q>} <Rd>, SPSR_mon

R	M	M1
0b1	0b1	0b1100

MSR{<c>}{<q>} SPSR_mon, <Rn>

R	M	M1
0b1	0b1	0b1100

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.133 SPSR_svc, Saved Program Status Register (Supervisor mode)

The SPSR_svc characteristics are:

Purpose

Holds the saved process state when an exception is taken to Supervisor mode.

Configurations

AArch32 System register SPSR_svc[31:0] is architecturally mapped to AArch64 System register SPSR_EL1[31:0].

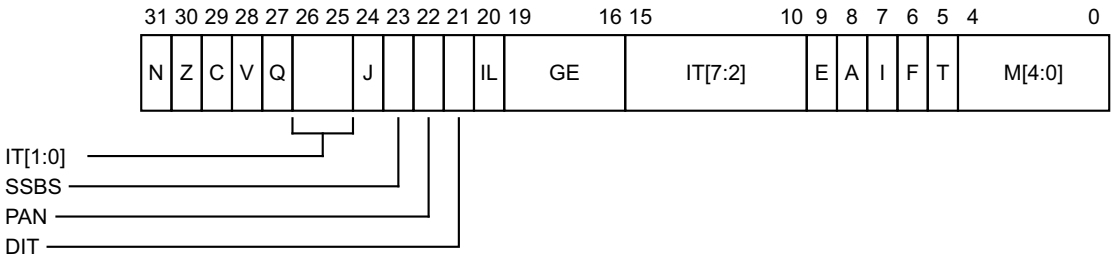
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to SPSR_svc are UNKNOWN.

Attributes

SPSR_svc is a 32-bit register.

Field descriptions

The SPSR_svc bit assignments are:



N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to Supervisor mode, and copied to PSTATE.N on executing an exception return operation in Supervisor mode.

This field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to Supervisor mode, and copied to PSTATE.Z on executing an exception return operation in Supervisor mode.

This field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to Supervisor mode, and copied to PSTATE.C on executing an exception return operation in Supervisor mode.

This field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to Supervisor mode, and copied to PSTATE.V on executing an exception return operation in Supervisor mode.

This field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to Supervisor mode, and copied to PSTATE.Q on executing an exception return operation in Supervisor mode.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

IT[1:0], bits [26:25]

If-Then. Set to the value of PSTATE.IT[1:0] on taking an exception to Supervisor mode, and copied to PSTATE.IT[1:0] on executing an exception return operation in Supervisor mode.

On executing an exception return operation in Supervisor mode SPSR_svc.IT must contain a value that is valid for the instruction being returned to.

This field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.

Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When ARMv8.0-SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to Supervisor mode, and copied to PSTATE.SSBS on executing an exception return operation in Supervisor mode.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When ARMv8.1-PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to Supervisor mode, and copied to PSTATE.PAN on executing an exception return operation in Supervisor mode.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When ARMv8.4-DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to Supervisor mode, and copied to PSTATE.DIT on executing an exception return operation in Supervisor mode.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to Supervisor mode, and copied to PSTATE.IL on executing an exception return operation in Supervisor mode.

This field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to Supervisor mode, and copied to PSTATE.GE on executing an exception return operation in Supervisor mode.

This field resets to an architecturally UNKNOWN value.

IT[7:2], bits [15:10]

If-Then. Set to the value of PSTATE.IT[7:2] on taking an exception to Supervisor mode, and copied to PSTATE.IT[7:2] on executing an exception return operation in Supervisor mode.

SPSR_svc.IT must contain a value that is valid for the instruction being returned to.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to Supervisor mode, and copied to PSTATE.E on executing an exception return operation in Supervisor mode.

If the implementation does not support big-endian operation, SPSR_svc.E is RES0. If the implementation does not support little-endian operation, SPSR_svc.E is RES1. On executing an exception return operation in Supervisor mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_svc.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_svc.E is RES1.

This field resets to an architecturally UNKNOWN value.

A, bit [8]

SError interrupt mask. Set to the value of PSTATE.A on taking an exception to Supervisor mode, and copied to PSTATE.A on executing an exception return operation in Supervisor mode.

This field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to Supervisor mode, and copied to PSTATE.I on executing an exception return operation in Supervisor mode.

This field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to Supervisor mode, and copied to PSTATE.F on executing an exception return operation in Supervisor mode.

This field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to Supervisor mode, and copied to PSTATE.T on executing an exception return operation in Supervisor mode.

This field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to Supervisor mode, and copied to PSTATE.M[4:0] on executing an exception return operation in Supervisor mode.

`0b10000` User.

`0b10001` FIQ.

`0b10010` IRQ.

`0b10011` Supervisor.

`0b10111` Abort.

`0b11011` Undefined.

`0b11111` System.

Other values are reserved. If SPSR_svc.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in Supervisor mode is an illegal return event, as described in *Illegal return events from AArch32 state* on page G1-5726.

This field resets to an architecturally UNKNOWN value.

Accessing the SPSR_svc

SPSR_svc is accessible in all modes other than User mode and Supervisor mode.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS{<c>}{<q>} <Rd>, SPSR_svc

R	M	M1
0b1	0b1	0b0010

MSR{<c>}{<q>} SPSR_svc, <Rn>

R	M	M1
0b1	0b1	0b0010

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.134 SPSR_und, Saved Program Status Register (Undefined mode)

The SPSR_und characteristics are:

Purpose

Holds the saved process state when an exception is taken to Undefined mode.

Configurations

AArch32 System register SPSR_und[31:0] is architecturally mapped to AArch64 System register SPSR_und[31:0].

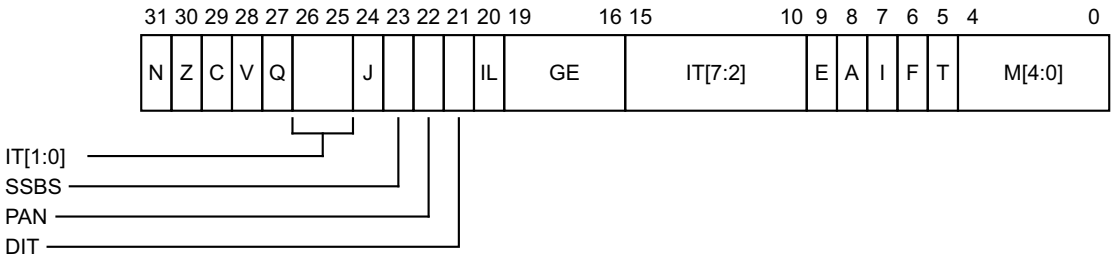
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to SPSR_und are UNKNOWN.

Attributes

SPSR_und is a 32-bit register.

Field descriptions

The SPSR_und bit assignments are:



N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to Undefined mode, and copied to PSTATE.N on executing an exception return operation in Undefined mode.

This field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to Undefined mode, and copied to PSTATE.Z on executing an exception return operation in Undefined mode.

This field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to Undefined mode, and copied to PSTATE.C on executing an exception return operation in Undefined mode.

This field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to Undefined mode, and copied to PSTATE.V on executing an exception return operation in Undefined mode.

This field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to Undefined mode, and copied to PSTATE.Q on executing an exception return operation in Undefined mode.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

IT[1:0], bits [26:25]

If-Then. Set to the value of PSTATE.IT[1:0] on taking an exception to Undefined mode, and copied to PSTATE.IT[1:0] on executing an exception return operation in Undefined mode.

On executing an exception return operation in Undefined mode SPSR_und.IT must contain a value that is valid for the instruction being returned to.

This field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.

Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When ARMv8.0-SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to Undefined mode, and copied to PSTATE.SSBS on executing an exception return operation in Undefined mode.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When ARMv8.1-PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to Undefined mode, and copied to PSTATE.PAN on executing an exception return operation in Undefined mode.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When ARMv8.4-DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to Undefined mode, and copied to PSTATE.DIT on executing an exception return operation in Undefined mode.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to Undefined mode, and copied to PSTATE.IL on executing an exception return operation in Undefined mode.

This field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to Undefined mode, and copied to PSTATE.GE on executing an exception return operation in Undefined mode.

This field resets to an architecturally UNKNOWN value.

IT[7:2], bits [15:10]

If-Then. Set to the value of PSTATE.IT[7:2] on taking an exception to Undefined mode, and copied to PSTATE.IT[7:2] on executing an exception return operation in Undefined mode.

SPSR_und.IT must contain a value that is valid for the instruction being returned to.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to Undefined mode, and copied to PSTATE.E on executing an exception return operation in Undefined mode.

If the implementation does not support big-endian operation, SPSR_und.E is RES0. If the implementation does not support little-endian operation, SPSR_und.E is RES1. On executing an exception return operation in Undefined mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_und.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_und.E is RES1.

This field resets to an architecturally UNKNOWN value.

A, bit [8]

SError interrupt mask. Set to the value of PSTATE.A on taking an exception to Undefined mode, and copied to PSTATE.A on executing an exception return operation in Undefined mode.

This field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to Undefined mode, and copied to PSTATE.I on executing an exception return operation in Undefined mode.

This field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to Undefined mode, and copied to PSTATE.F on executing an exception return operation in Undefined mode.

This field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to Undefined mode, and copied to PSTATE.T on executing an exception return operation in Undefined mode.

This field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to Undefined mode, and copied to PSTATE.M[4:0] on executing an exception return operation in Undefined mode.

`0b10000` User.

`0b10001` FIQ.

`0b10010` IRQ.

`0b10011` Supervisor.

`0b10111` Abort.

`0b11011` Undefined.

`0b11111` System.

Other values are reserved. If SPSR_und.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in Undefined mode is an illegal return event, as described in *Illegal return events from AArch32 state* on page G1-5726.

This field resets to an architecturally UNKNOWN value.

Accessing the SPSR_und

SPSR_und is accessible in all modes other than User mode and Undefined mode.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRS{<c>}{<q>} <Rd>, SPSR_und

R	M	M1
0b1	0b1	0b0110

MSR{<c>}{<q>} SPSR_und, <Rn>

R	M	M1
0b1	0b1	0b0110

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.135 TCMTR, TCM Type Register

The TCMTR characteristics are:

Purpose

Provides information about the implementation of the TCM.

Configurations

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to TCMTR are UNKNOWN.

If EL1 or above can use AArch32 then this register must be implemented.

Attributes

TCMTR is a 32-bit register.

Field descriptions

The TCMTR bit assignments are:

31	0
IMPLEMENTATION DEFINED	

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

Accessing the TCMTR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        return TCMTR;
elsif PSTATE.EL == EL2 then
    return TCMTR;
elsif PSTATE.EL == EL3 then
    return TCMTR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.136 TLBIALL, TLB Invalidate All

The TLBIALL characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that are from any level of the translation table walk. The entries that are invalidated are as follows:

- If executed at EL1, all entries that:
  - Would be required for the EL1&0 translation regime.
  - Match the current VMID, if EL2 is implemented and enabled in the current Security state.
- If executed in Secure state when EL3 is using AArch32, all entries that would be required for the Secure PL1&0 translation regime.
- If executed at EL2, and if EL2 is enabled in the current Security state, the stage 1 or stage 2 translation table entries that would be required for the PL1&0 translation regime and matches the current VMID.

The invalidation only applies to the PE that executes this System instruction.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to TLBIALL are UNKNOWN.

Attributes

TLBIALL is a 32-bit System instruction.

Field descriptions

TLBIALL ignores the value in the register specified by the instruction. Software does not have to write a value to the register before issuing this instruction.

Executing the TLBIALL instruction

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0111	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TTLB == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.FB == '1' then
        TLBIALLIS();
    else
        TLBIALL();
elsif PSTATE.EL == EL2 then
    TLBIALL();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    TLBIALL();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.137 TLBIALLH, TLB Invalidate All, Hyp mode

The TLBIALLH characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that are from any level of the translation table walk that would be required for the Non-secure EL2 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to TLBIALLH are UNKNOWN.

Attributes

TLBIALLH is a 32-bit System instruction.

Field descriptions

TLBIALLH ignores the value in the register specified by the instruction. Software does not have to write a value to the register before issuing this instruction.

Executing the TLBIALLH instruction

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0111	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBIALLH();
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        UNDEFINED;
    else
        TLBIALLH();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.138 TLBIALLHIS, TLB Invalidate All, Hyp mode, Inner Shareable

The TLBIALLHIS characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that are from any level of the translation table walk that would be required for the Non-secure EL2 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to TLBIALLHIS are UNKNOWN.

Attributes

TLBIALLHIS is a 32-bit System instruction.

Field descriptions

TLBIALLHIS ignores the value in the register specified by the instruction. Software does not have to write a value to the register before issuing this instruction.

Executing the TLBIALLHIS instruction

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBIALLHIS();
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        UNDEFINED;
    else
        TLBIALLHIS();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.139 TLBIALLIS, TLB Invalidate All, Inner Shareable

The TLBIALLIS characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that are from any level of the translation table walk. The entries that are invalidated are as follows:

- If executed at EL1, all entries that:
  - Would be required for the EL1&0 translation regime.
  - Match the current VMID, if EL2 is implemented and enabled in the current Security state.
- If executed in Secure state when EL3 is using AArch32, all entries that would be required for the Secure PL1&0 translation regime.
- If executed at EL2 and if EL2 is enabled in the current Security state, the stage 1 or stage 2 translation table entries that would be required for the PL1&0 translation regime and matches the current VMID.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to TLBIALLIS are UNKNOWN.

Attributes

TLBIALLIS is a 32-bit System instruction.

Field descriptions

TLBIALLIS ignores the value in the register specified by the instruction. Software does not have to write a value to the register before issuing this instruction.

Executing the TLBIALLIS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLBIS == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TTLB == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TTLBIS == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        TLBIALLIS();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL2 then
    TLBIALLIS();
elsif PSTATE.EL == EL3 then
    TLBIALLIS();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.140 TLBIALLNSNH, TLB Invalidate All, Non-Secure Non-Hyp

The TLBIALLNSNH characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that are from any level of the translation table walk that would be required for stage 1 or stage 2 of the Non-secure PL1&0 translation regime, regardless of the associated VMID.

The invalidation only applies to the PE that executes this System instruction.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to TLBIALLNSNH are UNKNOWN.

Attributes

TLBIALLNSNH is a 32-bit System instruction.

Field descriptions

TLBIALLNSNH ignores the value in the register specified by the instruction. Software does not have to write a value to the register before issuing this instruction.

Executing the TLBIALLNSNH instruction

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0111	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBIALLNSNH();
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        UNDEFINED;
    else
        TLBIALLNSNH();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.141 TLBIALLNSNHIS, TLB Invalidate All, Non-Secure Non-Hyp, Inner Shareable

The TLBIALLNSNHIS characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that are from any level of the translation table walk that would be required for stage 1 or stage 2 of the Non-secure PL1&0 translation regime, regardless of the associated VMID.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to TLBIALLNSNHIS are UNKNOWN.

Attributes

TLBIALLNSNHIS is a 32-bit System instruction.

Field descriptions

TLBIALLNSNHIS ignores the value in the register specified by the instruction. Software does not have to write a value to the register before issuing this instruction.

Executing the TLBIALLNSNHIS instruction

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0011	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBIALLNSNHIS();
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        UNDEFINED;
    else
        TLBIALLNSNHIS();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.142 TLBIASID, TLB Invalidate by ASID match

The TLBIASID characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used for the specified ASID, and either:
  - Is from a level of lookup above the final level.
  - Is a non-global entry from the final level of lookup.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to TLBIASID are UNKNOWN.

Attributes

TLBIASID is a 32-bit System instruction.

Field descriptions

The TLBIASID input value bit assignments are:

31 … 8	7 … 0
RES0	ASID

Bits [31:8]

Reserved, RES0.

ASID, bits [7:0]

ASID value to match. Any TLB entries for non-global pages that match the ASID values will be affected by this System instruction.

Executing the TLBIASID instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0111	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TTLB == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.FB == '1' then
        TLBIASIDIS(R[t]);
    else
        TLBIASID(R[t]);
elsif PSTATE.EL == EL2 then
    TLBIASID(R[t]);
elsif PSTATE.EL == EL3 then
    TLBIASID(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.143 TLBIASIDIS, TLB Invalidate by ASID match, Inner Shareable

The TLBIASIDIS characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used for the specified ASID, and either:
  - Is from a level of lookup above the final level.
  - Is a non-global entry from the final level of lookup.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to TLBIASIDIS are UNKNOWN.

Attributes

TLBIASIDIS is a 32-bit System instruction.

Field descriptions

The TLBIASIDIS input value bit assignments are:

31 … 8	7 … 0
RES0	ASID

Bits [31:8]

Reserved, RES0.

ASID, bits [7:0]

ASID value to match. Any TLB entries for non-global pages that match the ASID values will be affected by this System instruction.

Executing the TLBIASIDIS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0011	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLBIS == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TTLB == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TTLBIS == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        TLBIASIDIS(R[t]);
elsif PSTATE.EL == EL2 then
    TLBIASIDIS(R[t]);
elsif PSTATE.EL == EL3 then
    TLBIASIDIS(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.144 TLBIIPAS2, TLB Invalidate by Intermediate Physical Address, Stage 2

The TLBIIPAS2 characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 2 only translation table entry, from any level of the translation table walk.
- SCR.NS is 1.
- The entry would be used for the specified IPA.
- The entry would be used with the current VMID.
- The entry would be required for the PL1&0 translation regime.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation only applies to the PE that executes this System instruction.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to TLBIIPAS2 are UNKNOWN.

Note

This System instruction is not implemented in architecture versions before Armv8.

Attributes

TLBIIPAS2 is a 32-bit System instruction.

Field descriptions

The TLBIIPAS2 input value bit assignments are:

31 28	27 0
RES0	IPA[39:12]

Bits [31:28]

Reserved, RES0.

IPA[39:12], bits [27:0]

Bits[39:12] of the intermediate physical address to match.

Executing the TLBIIPAS2 instruction

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0100	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBIIPAS2(R[t]);
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        UNDEFINED;
    elsif SCR.NS == '0' then
        //no operation
    else
        TLBIIPAS2(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.145 TLBIIPAS2IS, TLB Invalidate by Intermediate Physical Address, Stage 2, Inner Shareable

The TLBIIPAS2IS characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 2 only translation table entry, from any level of the translation table walk.
- SCR.NS is 1.
- The entry would be used for the specified IPA.
- The entry would be used with the current VMID.
- The entry would be required for the PL1&0 translation regime.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to TLBIIPAS2IS are UNKNOWN.

Note

This System instruction is not implemented in architecture versions before Armv8.

Attributes

TLBIIPAS2IS is a 32-bit System instruction.

Field descriptions

The TLBIIPAS2IS input value bit assignments are:

31 28	27 0
RES0	IPA[39:12]

Bits [31:28]

Reserved, RES0.

IPA[39:12], bits [27:0]

Bits[39:12] of the intermediate physical address to match.

Executing the TLBIIPAS2IS instruction

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBIIPAS2(R[t]);
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        UNDEFINED;
    elsif SCR.NS == '0' then
        //no operation
    else
        TLBIIPAS2(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.146 TLBIIPAS2L, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level

The TLBIIPAS2L characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 2 only translation table entry, from the final level of the translation table walk.
- SCR.NS is 1.
- The entry would be used for the specified IPA.
- The entry would be used with the current VMID.
- The entry would be required for the PL1&0 translation regime.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation only applies to the PE that executes this System instruction.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to TLBIIPAS2L are UNKNOWN.

Note

This System instruction is not implemented in architecture versions before Armv8.

Attributes

TLBIIPAS2L is a 32-bit System instruction.

Field descriptions

The TLBIIPAS2L input value bit assignments are:

31 28	27 0
RES0	IPA[39:12]

Bits [31:28]

Reserved, RES0.

IPA[39:12], bits [27:0]

Bits[39:12] of the intermediate physical address to match.

Executing the TLBIIPAS2L instruction

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0100	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBIIPAS2(R[t]);
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        UNDEFINED;
    elsif SCR.NS == '0' then
        //no operation
    else
        TLBIIPAS2(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.147 TLBIIPAS2LIS, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, Inner Shareable

The TLBIIPAS2LIS characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 2 only translation table entry, from the final level of the translation table walk.
- SCR.NS is 1.
- The entry would be used for the specified IPA.
- The entry would be used with the current VMID.
- The entry would be required for the PL1&0 translation regime.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to TLBIIPAS2LIS are UNKNOWN.

Note

This System instruction is not implemented in architecture versions before Armv8.

Attributes

TLBIIPAS2LIS is a 32-bit System instruction.

Field descriptions

The TLBIIPAS2LIS input value bit assignments are:

31 28	27 0
RES0	IPA[39:12]

Bits [31:28]

Reserved, RES0.

IPA[39:12], bits [27:0]

Bits[39:12] of the intermediate physical address to match.

Executing the TLBIIPAS2LIS instruction

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0000	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBIIPAS2IS(R[t]);
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        UNDEFINED;
    elsif SCR.NS == '0' then
        //no operation
    else
        TLBIIPAS2IS(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.148 TLBIMVA, TLB Invalidate by VA

The TLBIMVA characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified address, and one of the following applies:
  - — The entry is from a level of lookup above the final level and matches the specified ASID.
  - — The entry is a global entry from the final level of lookup.
  - — The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to TLBIMVA are UNKNOWN.

Attributes

TLBIMVA is a 32-bit System instruction.

Field descriptions

The TLBIMVA input value bit assignments are:

31 … 12	11 … 8	7 … 0
VA	RES0	ASID

VA, bits [31:12]

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this operation.

Bits [11:8]

Reserved, RES0.

ASID, bits [7:0]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Executing the TLBIMVA instruction

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0111	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TTLB == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.FB == '1' then
        TLBIMVAIS(R[t]);
    else
        TLBIMVA(R[t]);
elsif PSTATE.EL == EL2 then
    TLBIMVA(R[t]);
elsif PSTATE.EL == EL3 then
    TLBIMVA(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.149 TLBIMVAA, TLB Invalidate by VA, All ASID

The TLBIMVAA characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be used to translate the specified address.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to TLBIMVAA are UNKNOWN.

Attributes

TLBIMVAA is a 32-bit System instruction.

Field descriptions

The TLBIMVAA input value bit assignments are:

31 … 12	11 … 0
VA	RES0

VA, bits [31:12]

Virtual address to match. Any unlocked TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

Bits [11:0]

Reserved, RES0.

Executing the TLBIMVAA instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0111	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TTLB == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.FB == '1' then
        TLBIMVAAIS(R[t]);
    else
        TLBIMVAA(R[t]);
elsif PSTATE.EL == EL2 then
    TLBIMVAA(R[t]);
elsif PSTATE.EL == EL3 then
    TLBIMVAA(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.150 TLBIMVAAIS, TLB Invalidate by VA, All ASID, Inner Shareable

The TLBIMVAAIS characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be used to translate the specified address.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to TLBIMVAAIS are UNKNOWN.

Attributes

TLBIMVAAIS is a 32-bit System instruction.

Field descriptions

The TLBIMVAAIS input value bit assignments are:

31 … 12	11 … 0
VA	RES0

VA, bits [31:12]

Virtual address to match. Any unlocked TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

Bits [11:0]

Reserved, RES0.

Executing the TLBIMVAAIS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0011	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLBIS == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TTLB == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TTLBIS == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        TLBIMVAAIS(R[t]);
elsif PSTATE.EL == EL2 then
    TLBIMVAAIS(R[t]);
elsif PSTATE.EL == EL3 then
    TLBIMVAAIS(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.151 TLBIMVAAL, TLB Invalidate by VA, All ASID, Last level

The TLBIMVAAL characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.
- The entry would be used to translate the specified address.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to TLBIMVAAL are UNKNOWN.

Note

This System instruction is not implemented in architecture versions before Armv8.

Attributes

TLBIMVAAL is a 32-bit System instruction.

Field descriptions

The TLBIMVAAL input value bit assignments are:

31 … 12	11 … 0
VA	RES0

VA, bits [31:12]

Virtual address to match. Any unlocked TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

Bits [11:0]

Reserved, RES0.

Executing the TLBIMVAAL instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0111	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TTLB == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.FB == '1' then
        TLBIMVAALIS(R[t]);
    else
        TLBIMVAAL(R[t]);
elsif PSTATE.EL == EL2 then
    TLBIMVAAL(R[t]);
elsif PSTATE.EL == EL3 then
    TLBIMVAAL(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.152 TLBIMVAALIS, TLB Invalidate by VA, All ASID, Last level, Inner Shareable

The TLBIMVAALIS characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.
- The entry would be used to translate the specified address.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to TLBIMVAALIS are UNKNOWN.

Note

This System instruction is not implemented in architecture versions before Armv8.

Attributes

TLBIMVAALIS is a 32-bit System instruction.

Field descriptions

The TLBIMVAALIS input value bit assignments are:

31 … 12	11 … 0
VA	RES0

VA, bits [31:12]

Virtual address to match. Any unlocked TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

Bits [11:0]

Reserved, RES0.

Executing the TLBIMVAALIS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0011	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLBIS == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TTLB == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TTLBIS == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        TLBIMVAALIS(R[t]);
elsif PSTATE.EL == EL2 then
    TLBIMVAALIS(R[t]);
elsif PSTATE.EL == EL3 then
    TLBIMVAALIS(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.153 TLBIMVAH, TLB Invalidate by VA, Hyp mode

The TLBIMVAH characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that are from any level of the translation table walk that would be required for the Non-secure EL2 translation regime and used to translate the specified address.

The invalidation only applies to the PE that executes this System instruction.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to TLBIMVAH are UNKNOWN.

Attributes

TLBIMVAH is a 32-bit System instruction.

Field descriptions

The TLBIMVAH input value bit assignments are:

31 … 12	11 … 0
VA	RES0

VA, bits [31:12]

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Bits [11:0]

Reserved, RES0.

Executing the TLBIMVAH instruction

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0111	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBIMVAH(R[t]);
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        UNDEFINED;
    else
        TLBIMVAH(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.154 TLBIMVAHIS, TLB Invalidate by VA, Hyp mode, Inner Shareable

The TLBIMVAHIS characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that are from any level of the translation table walk that would be required for the Non-secure EL2 translation regime and used to translate the specified address.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to TLBIMVAHIS are UNKNOWN.

Attributes

TLBIMVAHIS is a 32-bit System instruction.

Field descriptions

The TLBIMVAHIS input value bit assignments are:

31 … 12	11 … 0
VA	RES0

VA, bits [31:12]

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Bits [11:0]

Reserved, RES0.

Executing the TLBIMVAHIS instruction

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0011	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBIMVAHIS(R[t]);
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        UNDEFINED;
    else
        TLBIMVAHIS(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.155 TLBIMVAIS, TLB Invalidate by VA, Inner Shareable

The TLBIMVAIS characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified address, and one of the following applies:
  - — The entry is from a level of lookup above the final level and matches the specified ASID.
  - — The entry is a global entry from the final level of lookup.
  - — The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to TLBIMVAIS are UNKNOWN.

Attributes

TLBIMVAIS is a 32-bit System instruction.

Field descriptions

The TLBIMVAIS input value bit assignments are:

31 … 12	11 … 8	7 … 0
VA	RES0	ASID

VA, bits [31:12]

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Bits [11:8]

Reserved, RES0.

ASID, bits [7:0]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

Executing the TLBIMVAIS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0011	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLBIS == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TTLB == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TTLBIS == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        TLBIMVAIS(R[t]);
elsif PSTATE.EL == EL2 then
    TLBIMVAIS(R[t]);
elsif PSTATE.EL == EL3 then
    TLBIMVAIS(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.156 TLBIMVAL, TLB Invalidate by VA, Last level

The TLBIMVAL characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified address, and one of the following applies:
  - — The entry is a global entry from the final level of lookup.
  - — The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to TLBIMVAL are UNKNOWN.

This System instruction is not implemented in architecture versions before Armv8.

Attributes

TLBIMVAL is a 32-bit System instruction.

Field descriptions

The TLBIMVAL input value bit assignments are:

31 … 12	11 … 8	7 … 0
VA	RES0	ASID

VA, bits [31:12]

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Bits [11:8]

Reserved, RES0.

ASID, bits [7:0]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Executing the TLBIMVAL instruction

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0111	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TTLB == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.FB == '1' then
        TLBIMVALIS(R[t]);
    else
        TLBIMVAL(R[t]);
elsif PSTATE.EL == EL2 then
    TLBIMVAL(R[t]);
elsif PSTATE.EL == EL3 then
    TLBIMVAL(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.157 TLBIMVALH, TLB Invalidate by VA, Last level, Hyp mode

The TLBIMVALH characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that are from the final level of the translation table walk that would be required for the Non-secure EL2 translation regime and used to translate the specified address.

The invalidation only applies to the PE that executes this System instruction.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to TLBIMVALH are UNKNOWN.

This System instruction is not implemented in architecture versions before Armv8.

Attributes

TLBIMVALH is a 32-bit System instruction.

Field descriptions

The TLBIMVALH input value bit assignments are:

31 … 12	11 … 0
VA	RES0

VA, bits [31:12]

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Bits [11:0]

Reserved, RES0.

Executing the TLBIMVALH instruction

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0111	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBIMVALH(R[t]);
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        UNDEFINED;
    else
        TLBIMVALH(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.158 TLBIMVALHIS, TLB Invalidate by VA, Last level, Hyp mode, Inner Shareable

The TLBIMVALHIS characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that are from the final level of the translation table walk that would be required for the Non-secure EL2 translation regime and used to translate the specified address.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to TLBIMVALHIS are UNKNOWN.

This System instruction is not implemented in architecture versions before Armv8.

Attributes

TLBIMVALHIS is a 32-bit System instruction.

Field descriptions

The TLBIMVALHIS input value bit assignments are:

31 … 12	11 … 0
VA	RES0

VA, bits [31:12]

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Bits [11:0]

Reserved, RES0.

Executing the TLBIMVALHIS instruction

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0011	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TLBIMVALHIS(R[t]);
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        UNDEFINED;
    else
        TLBIMVALHIS(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.159 TLBIMVALIS, TLB Invalidate by VA, Last level, Inner Shareable

The TLBIMVALIS characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified address, and one of the following applies:
  - — The entry is a global entry from the final level of lookup.
  - — The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configurations

This instruction is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to TLBIMVALIS are UNKNOWN.

This System instruction is not implemented in architecture versions before Armv8.

Attributes

TLBIMVALIS is a 32-bit System instruction.

Field descriptions

The TLBIMVALIS input value bit assignments are:

31 … 12	11 … 8	7 … 0
VA	RES0	ASID

VA, bits [31:12]

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Bits [11:8]

Reserved, RES0.

ASID, bits [7:0]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Executing the TLBIMVALIS instruction

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0011	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLBIS == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TTLB == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TTLBIS == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        TLBIMVAL(R[t]);
elsif PSTATE.EL == EL2 then
    TLBIMVAL(R[t]);
elsif PSTATE.EL == EL3 then
    TLBIMVAL(R[t]);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.160 TLBTR, TLB Type Register

The TLBTR characteristics are:

Purpose

Provides information about the TLB implementation. The register must define whether the implementation provides separate instruction and data TLBs, or a unified TLB. Normally, the IMPLEMENTATION DEFINED information in this register includes the number of lockable entries in the TLB.

Configurations

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to TLBTR are UNKNOWN.

Attributes

TLBTR is a 32-bit register.

Field descriptions

The TLBTR bit assignments are:

31 … 1	0
IMPLEMENTATION DEFINED	nU

IMPLEMENTATION DEFINED, bits [31:1]

IMPLEMENTATION DEFINED.

nU, bit [0]

Not Unified TLB. Indicates whether the implementation has a unified TLB:

`0b0` Unified TLB.

`0b1` Separate Instruction and Data TLBs.

Accessing the TLBTR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0000	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        return TLBTR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL2 then
    return TLBTR;
elsif PSTATE.EL == EL3 then
    return TLBTR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.161 TPIDRPRW, PL1 Software Thread ID Register

The TPIDRPRW characteristics are:

Purpose

Provides a location where software executing at EL1 or higher can store thread identifying information that is not visible to software executing at EL0, for OS management purposes.

The PE makes no use of this register.

Configurations

AArch32 System register TPIDRPRW[31:0] is architecturally mapped to AArch64 System register TPIDR_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to TPIDRPRW are UNKNOWN.

Note

The PE never updates this register.

Attributes

TPIDRPRW is a 32-bit register.

Field descriptions

The TPIDRPRW bit assignments are:

31 … 0
Thread ID

Bits [31:0]

Thread ID. Thread identifying information stored by software running at this Exception level.

This field resets to an architecturally UNKNOWN value.

Accessing the TPIDRPRW

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0000	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return TPIDRPRW_NS;
    else
        return TPIDRPRW;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return TPIDRPRW_NS;
    else
        return TPIDRPRW;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return TPIDRPRW_S;
    else
        return TPIDRPRW_NS;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0000	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        TPIDRPRW_NS = R[t];
    else
        TPIDRPRW = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        TPIDRPRW_NS = R[t];
    else
        TPIDRPRW = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        TPIDRPRW_S = R[t];
    else
        TPIDRPRW_NS = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.162 TPIDRURO, PL0 Read-Only Software Thread ID Register

The TPIDRURO characteristics are:

Purpose

Provides a location where software executing at EL1 or higher can store thread identifying information that is visible to software executing at EL0, for OS management purposes.

The PE makes no use of this register.

Configurations

AArch32 System register TPIDRURO[31:0] is architecturally mapped to AArch64 System register TPIDRRO_EL0[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to TPIDRURO are UNKNOWN.

Note

The PE never updates this register.

Attributes

TPIDRURO is a 32-bit register.

Field descriptions

The TPIDRURO bit assignments are:

31 … 0
Thread ID

Bits [31:0]

Thread ID. Thread identifying information stored by software running at this Exception level.

This field resets to an architecturally UNKNOWN value.

Accessing the TPIDRURO

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0000	0b011

```
if PSTATE.EL == EL0 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HFGRTR_EL2.TPIDRRO_EL0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        return TPIDRURO;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return TPIDRURO_NS;
    else
        return TPIDRURO;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return TPIDRURO_NS;
    else
        return TPIDRURO;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return TPIDRURO_S;
    else
        return TPIDRURO_NS;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0000	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        TPIDRURO_NS = R[t];
    else
        TPIDRURO = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        TPIDRURO_NS = R[t];
    else
        TPIDRURO = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        TPIDRURO_S = R[t];
    else
        TPIDRURO_NS = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.163 TPIDRURW, PL0 Read/Write Software Thread ID Register

The TPIDRURW characteristics are:

Purpose

Provides a location where software executing at EL0 can store thread identifying information, for OS management purposes.

The PE makes no use of this register.

Configurations

AArch32 System register TPIDRURW[31:0] is architecturally mapped to AArch64 System register TPIDR_EL0[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to TPIDRURW are UNKNOWN.

Note

The PE never updates this register.

Attributes

TPIDRURW is a 32-bit register.

Field descriptions

The TPIDRURW bit assignments are:

31 … 0
Thread ID

Bits [31:0]

Thread ID. Thread identifying information stored by software running at this Exception level.

This field resets to an architecturally UNKNOWN value.

Accessing the TPIDRURW

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0000	0b010

```
if PSTATE.EL == EL0 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HFGRTR_EL2.TPIDR_EL0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        return TPIDRURW;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return TPIDRURW_NS;
    else
        return TPIDRURW;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return TPIDRURW_NS;
    else
        return TPIDRURW;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return TPIDRURW_S;
    else
        return TPIDRURW_NS;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0000	0b010

```
if PSTATE.EL == EL0 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HFGWTR_EL2.TPIDR_EL0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        TPIDRURW = R[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        TPIDRURW_NS = R[t];
    else
        TPIDRURW = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        TPIDRURW_NS = R[t];
    else
        TPIDRURW = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        TPIDRURW_S = R[t];
    else
        TPIDRURW_NS = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.164 TTBCR, Translation Table Base Control Register

The TTBCR characteristics are:

Purpose

The control register for stage 1 of the PL1&0 translation regime. Its controls include:

- Where the VA range is split between addresses translated using TTBR0 and addresses translated using TTBR1.
- The translation table format used by this stage of translation.

From Armv8.2, when the value of TTBCR.{EAE, T2E} is {1, 1}, TTBCR is used with TTBCR2.

Configurations

AArch32 System register TTBCR[31:0] is architecturally mapped to AArch64 System register TCR_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to TTBCR are UNKNOWN.

The current translation table format determines which format of the register is used.

Some RW fields of this register have defined reset values. These apply only if the PE resets into an Exception level that is using AArch32. If the PE resets into EL3 using AArch32 then:

- The EAE bit resets to 0 in both the Secure and the Non-secure instances of the register.
- Other reset values apply only to the Secure instance of the register.

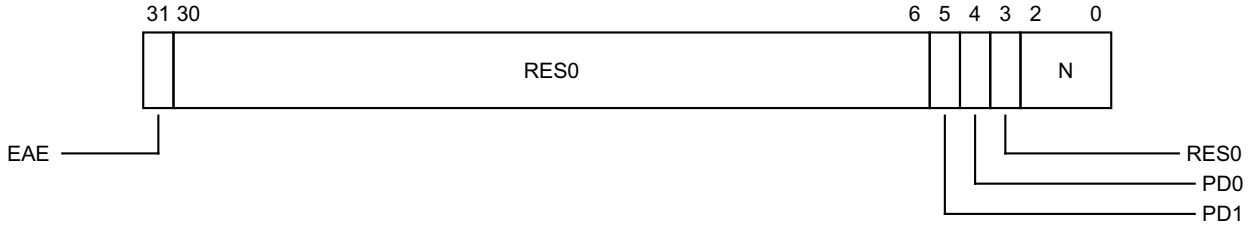
Attributes

TTBCR is a 32-bit register.

Field descriptions

The TTBCR bit assignments are:

When TTBCR.EAE == 0:



EAE, bit [31]

Extended Address Enable. The meanings of the possible values of this bit are:

`0b0` Use the VMSAv8-32 translation system with the Short-descriptor translation table format.

This field resets to `0`.

Bits [30:6]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

PD1, bit [5]

Translation table walk disable for translations using TTBR1. This bit controls whether a translation table walk is performed on a TLB miss, for an address that is translated using TTBR1. The encoding of this bit is:

`0b0` Perform translation table walks using TTBR1.

`0b1` A TLB miss on an address that is translated using TTBR1 generates a Translation fault. No translation table walk is performed.

This field resets to `0`.

PD0, bit [4]

Translation table walk disable for translations using TTBR0. This bit controls whether a translation table walk is performed on a TLB miss for an address that is translated using TTBR0. The encoding of this bit is:

`0b0` Perform translation table walks using TTBR0.

`0b1` A TLB miss on an address that is translated using TTBR0 generates a Translation fault. No translation table walk is performed.

This field resets to `0`.

Bit [3]

Reserved, RES0.

N, bits [2:0]

Indicate the width of the base address held in TTBR0. In TTBR0, the base address field is bits[31:14-N]. The value of N also determines:

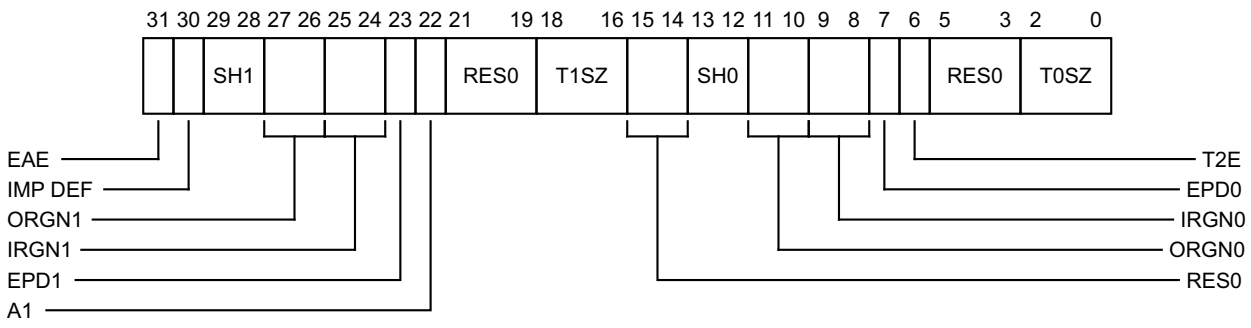
- Whether TTBR0 or TTBR1 is used as the base address for translation table walks.
- The size of the translation table pointed to by TTBR0.

N can take any value from 0 to 7, that is, from `0b000` to `0b111`.

When N has its reset value of 0, the translation table base is compatible with Armv5 and Armv6.

This field resets to `0`.

When TTBCR.EAE == 1:



EAE, bit [31]

Extended Address Enable. The meanings of the possible values of this bit are:

`0b1` Use the VMSAv8-32 translation system with the Long-descriptor translation table format.

This field resets to `0`.

IMPLEMENTATION DEFINED, bit [30]

IMPLEMENTATION DEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to `0`.

SH1, bits [29:28]

Shareability attribute for memory associated with translation table walks using TTBR1. Defined values are:

`0b00` Non-shareable.

`0b10` Outer Shareable.

`0b11` Inner Shareable.

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-7853.

This field resets to `0`.

ORGN1, bits [27:26]

Outer cacheability attribute for memory associated with translation table walks using TTBR1.

`0b00` Normal memory, Outer Non-cacheable.

`0b01` Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.

`0b10` Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.

`0b11` Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.

This field resets to `0`.

IRGN1, bits [25:24]

Inner cacheability attribute for memory associated with translation table walks using TTBR1.

`0b00` Normal memory, Inner Non-cacheable.

`0b01` Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.

`0b10` Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.

`0b11` Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

This field resets to `0`.

EPD1, bit [23]

Translation table walk disable for translations using TTBR1. This bit controls whether a translation table walk is performed on a TLB miss, for an address that is translated using TTBR1. The encoding of this bit is:

`0b0` Perform translation table walks using TTBR1.

`0b1` A TLB miss on an address that is translated using TTBR1 generates a Translation fault. No translation table walk is performed.

This field resets to `0`.

A1, bit [22]

Selects whether TTBR0 or TTBR1 defines the ASID. The encoding of this bit is:

`0b0` TTBR0.ASID defines the ASID.

`0b1` TTBR1.ASID defines the ASID.

This field resets to `0`.

Bits [21:19]

Reserved, RES0.

T1SZ, bits [18:16]

See *Selecting between TTBR0 and TTBR1, VMSAv8-32 Long-descriptor translation table format* on page G5-5957 for how TTBCR.{T1SZ, T0SZ} determine the input address ranges and memory region sizes translated using TTBR0 and TTBR1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to 0.

Bits [15:14]

Reserved, RES0.

SH0, bits [13:12]

Shareability attribute for memory associated with translation table walks using TTBR0.

0b00 Non-shareable

0b10 Outer Shareable

0b11 Inner Shareable

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-7853.

This field resets to 0.

ORGN0, bits [11:10]

Outer cacheability attribute for memory associated with translation table walks using TTBR0.

0b00 Normal memory, Outer Non-cacheable.

0b01 Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.

0b10 Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.

0b11 Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.

This field resets to 0.

IRGN0, bits [9:8]

Inner cacheability attribute for memory associated with translation table walks using TTBR0.

0b00 Normal memory, Inner Non-cacheable.

0b01 Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.

0b10 Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.

0b11 Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

This field resets to 0.

EPD0, bit [7]

Translation table walk disable for translations using TTBR0. This bit controls whether a translation table walk is performed on a TLB miss, for an address that is translated using TTBR0. The encoding of this bit is:

0b0 Perform translation table walks using TTBR0.

0b1 A TLB miss on an address that is translated using TTBR0 generates a Translation fault. No translation table walk is performed.

This field resets to 0.

T2E, bit [6]

When ARMv8.2-AA32HPD is implemented:

TTBCR2 Enable.

0b0 TTBCR2 is disabled. The contents of TTBCR2 are treated as 0 for all purposes other than reading or writing the register.

0b1 TTBCR2 is enabled.

If TTBCR.EAE==0, then the behavior is as if the bit is 0.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bits [5:3]

Reserved, RES0.

T0SZ, bits [2:0]

See *Selecting between TTBR0 and TTBR1, VMSAv8-32 Long-descriptor translation table format* on page G5-5957 for how TTBCR.{T1SZ, T0SZ} determine the input address ranges and memory region sizes translated using TTBR0 and TTBR1.

This field resets to 0.

Accessing the TTBCR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0010	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T2 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TRVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return TTBCR_NS;
    else
        return TTBCR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return TTBCR_NS;
    else
        return TTBCR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return TTBCR_S;
    else
        return TTBCR_NS;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0010	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T2 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        TTBCR_NS = R[t];
    else
        TTBCR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        TTBCR_NS = R[t];
    else
        TTBCR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' && CP15SDISABLE == HIGH then
        UNDEFINED;
    elsif SCR.NS == '0' && CP15SDISABLE2 == HIGH then
        UNDEFINED;
    else
        if SCR.NS == '0' then
            TTBCR_S = R[t];
        else
            TTBCR_NS = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.165 TTBCR2, Translation Table Base Control Register 2

The TTBCR2 characteristics are:

Purpose

The second control register for stage 1 of the PL1&0 translation regime.

If ARMv8.2-AA32HPD is not implemented then this register is not implemented and its encoding is UNDEFINED. Otherwise:

- When the value of TTBCR.{EAE, T2E} is not {1, 1} the contents of TTBCR2 are treated as zero for all purposes other than reading or writing the register.
- When the value of TTBCR.{EAE, T2E} is {1, 1} TTBCR2 is used with TTBCR.

Configurations

AArch32 System register TTBCR2[31:0] is architecturally mapped to AArch64 System register TCR_EL1[63:32].

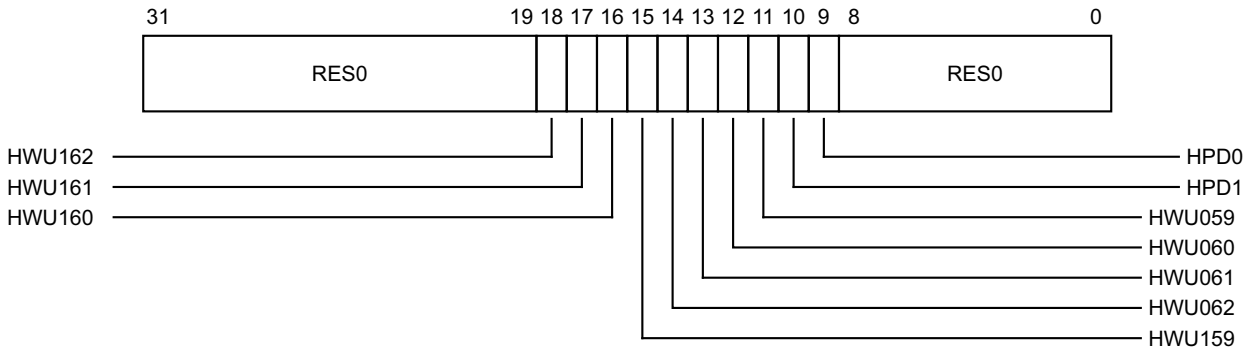
This register is present only when AArch32 is supported at any Exception level and ARMv8.2-AA32HPD is implemented. Otherwise, direct accesses to TTBCR2 are UNDEFINED.

Attributes

TTBCR2 is a 32-bit register.

Field descriptions

The TTBCR2 bit assignments are:



Bits [31:19]

Reserved, RES0.

HWU162, bit [18]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 1 translation table Block or Page entry for translations using TTBR1.

`0b0` For translations using TTBR1, bit[62] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` For translations using TTBR1, bit[62] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TTBCR2.HPD1 is 1.

The Effective value of this field is 0 if the value of TTBCR2.HPD1 is 0 or the value of TTBCR.T2E is 0.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

Reserved, RES0.

HWU161, bit [17]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 1 translation table Block or Page entry for translations using TTBR1.

`0b0` For translations using TTBR1, bit[61] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` For translations using TTBR1, bit[61] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TTBCR2.HPD1 is 1.

The Effective value of this field is 0 if the value of TTBCR2.HPD1 is 0 or the value of TTBCR.T2E is 0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU160, bit [16]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 1 translation table Block or Page entry for translations using TTBR1.

`0b0` For translations using TTBR1, bit[60] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` For translations using TTBR1, bit[60] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TTBCR2.HPD1 is 1.

The Effective value of this field is 0 if the value of TTBCR2.HPD1 is 0 or the value of TTBCR.T2E is 0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU159, bit [15]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 1 translation table Block or Page entry for translations using TTBR1.

`0b0` For translations using TTBR1, bit[59] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` For translations using TTBR1, bit[59] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TTBCR2.HPD1 is 1.

The Effective value of this field is 0 if the value of TTBCR2.HPD1 is 0 or the value of TTBCR.T2E is 0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

HWU062, bit [14]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 1 translation table Block or Page entry for translations using TTBR0.

`0b0` For translations using TTBR0, bit[62] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` For translations using TTBR0, bit[62] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TTBCR2.HPD0 is 1.

The Effective value of this field is 0 if the value of TTBCR2.HPD0 is 0 or the value of TTBCR.T2E is 0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU061, bit [13]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 1 translation table Block or Page entry for translations using TTBR0.

`0b0` For translations using TTBR0, bit[61] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` For translations using TTBR0, bit[61] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TTBCR2.HPD0 is 1.

The Effective value of this field is 0 if the value of TTBCR2.HPD0 is 0 or the value of TTBCR.T2E is 0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU060, bit [12]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 1 translation table Block or Page entry for translations using TTBR0.

`0b0` For translations using TTBR0, bit[60] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` For translations using TTBR0, bit[60] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TTBCR2.HPD0 is 1.

The Effective value of this field is 0 if the value of TTBCR2.HPD0 is 0 or the value of TTBCR.T2E is 0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

HWU059, bit [11]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 1 translation table Block or Page entry for translations using TTBR0.

`0b0` For translations using TTBR0, bit[59] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` For translations using TTBR0, bit[59] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TTBCR2.HPD0 is 1.

The Effective value of this field is 0 if the value of TTBCR2.HPD0 is 0 or the value of TTBCR.T2E is 0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HPD1, bit [10]

Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, XNTable, and PXNTable, in the translation tables pointed to by TTBR1.

`0b0` Hierarchical permissions are enabled.

`0b1` Hierarchical permissions are disabled if TTBCR.T2E == 1.

When disabled, the permissions are treated as if the bits are 0.

The Effective value of this field is 0 if the value of TTBCR.T2E is 0.

This field resets to an architecturally UNKNOWN value.

HPD0, bit [9]

Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, XNTable, and PXNTable, in the translation tables pointed to by TTBR0.

`0b0` Hierarchical permissions are enabled.

`0b1` Hierarchical permissions are disabled if TTBCR.T2E ==1.

When disabled, the permissions are treated is as if the bits are 0.

The Effective value of this field is 0 if the value of TTBCR.T2E is 0.

This field resets to an architecturally UNKNOWN value.

Bits [8:0]

Reserved, RES0.

Accessing the TTBCR2

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0010	0b0000	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T2 == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TRVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return TTBCR2_NS;
    else
        return TTBCR2;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return TTBCR2_NS;
    else
        return TTBCR2;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return TTBCR2_S;
    else
        return TTBCR2_NS;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0010	0b0000	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T2 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        TTBCR2_NS = R[t];
    else
        TTBCR2 = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        TTBCR2_NS = R[t];
    else
        TTBCR2 = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' && CP15SDISABLE == HIGH then
        UNDEFINED;
    elsif SCR.NS == '0' && CP15SDISABLE2 == HIGH then
        UNDEFINED;
    else
        if SCR.NS == '0' then
            TTBCR2_S = R[t];
        else
            TTBCR2_NS = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.166 TTBR0, Translation Table Base Register 0

The TTBR0 characteristics are:

Purpose

Holds the base address of the translation table for the initial lookup for stage 1 of the translation of an address from the lower VA range in the PL1&0 translation regime, and other information for this translation regime.

Configurations

AArch32 System register TTBR0[63:0] is architecturally mapped to AArch64 System register TTBR0_EL1[63:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to TTBR0 are UNKNOWN.

TTBCR.EAE determines which TTBR0 format is used:

- TTBCR.EAE == `0b0`: 32-bit format is used. TTBR0[63:32] are ignored.
- TTBCR.EAE == `0b1`: 64-bit format is used.

When EL3 is using AArch32, write access to TTBR0(S) is disabled when the CP15SDISABLE signal is asserted HIGH.

Used in conjunction with the TTBCR. When the 64-bit TTBR0 format is used, cacheability and shareability information is held in the TTBCR, not in TTBR0.

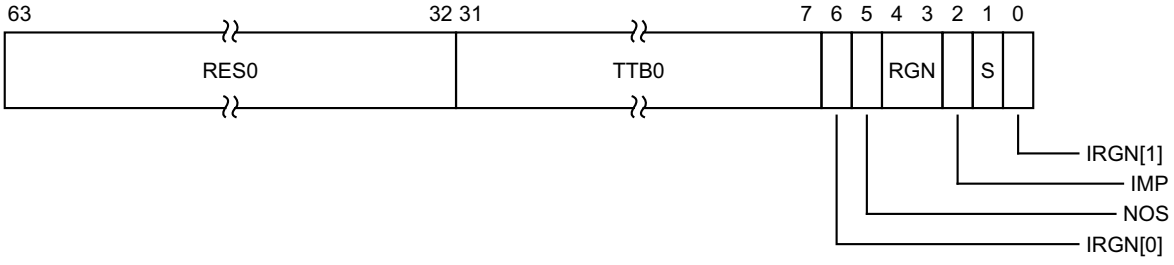
Attributes

TTBR0 is a 64-bit register that can also be accessed as a 32-bit value. If it is accessed as a 32-bit register, accesses read and write bits [31:0] and do not modify bits [63:32].

Field descriptions

The TTBR0 bit assignments are:

When TTBCR.EAE == 0:



Bits [63:32]

Reserved, RES0.

TTB0, bits [31:7]

Translation table base address, bits[31:x], where x is 14-(TTBCR.N). Register bits [x-1:7] are RES0, with the additional requirement that if these bits are not all zero, this is a misaligned translation table base address, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Register bits [x-1:7] are treated as if all the bits are zero. The value read back from these bits is either the value written or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

IRGN[0], bit [6]

This field is bit[0] of IRGN[1:0].

Inner region bits. Bits [0,6] of this register together indicate the Inner Cacheability attributes for the memory associated with the translation table walks. The possible values of IRGN[1:0] are:

`0b00` Normal memory, Inner Non-cacheable.

`0b01` Normal memory, Inner Write-Back Write-Allocate Cacheable.

`0b10` Normal memory, Inner Write-Through Cacheable.

`0b11` Normal memory, Inner Write-Back no Write-Allocate Cacheable.

Note

The encoding of the IRGN bits is counter-intuitive, with register bit[6] being IRGN[0] and register bit[0] being IRGN[1]. This encoding is chosen to give a consistent encoding of memory region types and to ensure that software written for ARMv7 without the Multiprocessing Extensions can run unmodified on an implementation that includes the functionality introduced by the ARMv7 Multiprocessing Extensions.

The IRGN field is split as follows:

- IRGN[0] is TTBR0[6].
- IRGN[1] is TTBR0[0].

This field resets to an architecturally UNKNOWN value.

NOS, bit [5]

Not Outer Shareable. When the value of TTBR0.S is 1, indicates whether the memory associated with a translation table walk is Inner Shareable or Outer Shareable:

`0b0` Memory is Outer Shareable.

`0b1` Memory is Inner Shareable.

This bit is ignored when the value of TTBR0.S is 0.

This field resets to an architecturally UNKNOWN value.

RGN, bits [4:3]

Region bits. Indicates the Outer cacheability attributes for the memory associated with the translation table walks:

`0b00` Normal memory, Outer Non-cacheable.

`0b01` Normal memory, Outer Write-Back Write-Allocate Cacheable.

`0b10` Normal memory, Outer Write-Through Cacheable.

`0b11` Normal memory, Outer Write-Back no Write-Allocate Cacheable.

This field resets to an architecturally UNKNOWN value.

IMP, bit [2]

The effect of this bit is IMPLEMENTATION DEFINED. If the translation table implementation does not include any IMPLEMENTATION DEFINED features this bit is RES0.

This field resets to an architecturally UNKNOWN value.

S, bit [1]

Shareable. Indicates whether the memory associated with the translation table walks is Non-shareable:

`0b0` Memory is Non-shareable.

`0b1` Memory is shareable. The TTBR0.NOS field indicates whether the memory is Inner Shareable or Outer Shareable.

This field resets to an architecturally UNKNOWN value.

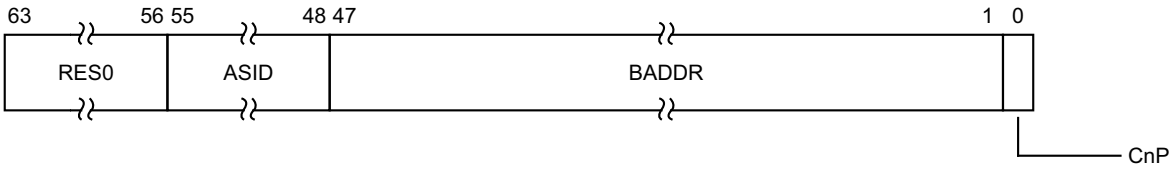
Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

IRGN[1], bit [0]

This field is bit[1] of IRGN[1:0].

See IRGN[0] for the field description.

When TTBCR.EAE == 1:



Bits [63:56]

Reserved, RES0.

ASID, bits [55:48]

An ASID for the translation table base address. The TTBCR.A1 field selects either TTBR0.ASID or TTBR1.ASID.

This field resets to an architecturally UNKNOWN value.

BADDR, bits [47:1]

Translation table base address, bits[47:x], Bits [x-1:1] are RES0, with the additional requirement that if bits[x-1:3] are not all zero, this is a misaligned translation table base address, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Register bits [x-1:3] are treated as if all the bits are zero. The value read back from these bits is either the value written or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

x is determined from the value of TTBCR.T0SZ as follows:

- If TTBCR.T0SZ is 0 or 1, x = 5 - TTBCR.T0SZ.
- If TTBCR.T0SZ is greater than 1, x = 14 - TTBCR.T0SZ.

If bits[47:40] of the translation table base address are not zero, an Address size fault is generated.

This field resets to an architecturally UNKNOWN value.

CnP, bit [0]

When ARMv8.2-TTCNP is implemented:

Common not Private. When TTBCR.EAE ==1, this bit indicates whether each entry that is pointed to by TTBR0 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of TTBR0.CnP is 1.

`0b0` The translation table entries pointed to by this instance of TTBR0, for the current ASID, are permitted to differ from corresponding entries for this instance of TTBR0 for other PEs in the Inner Shareable domain. This is not affected by:

- The value of TTBR0.CnP on those other PEs.
- The value of TTBR0.EAE on those other PEs.
- The value of the current ASID or, for the Non-secure instance of TTBR0, the value of the current VMID.

`0b1` The translation table entries pointed to by this instance of TTBR0 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of TTBR0.CnP is 1 for this instance of TTBR0 and all of the following apply:

- The translation table entries are pointed to by this instance of TTBR0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The value of the applicable TTBCR.EAE field is 1.
- The ASID is the same as the current ASID.
- For the Non-secure instance of TTBR0, the VMID is the same as the current VMID.

When a TLB combines entries from stage 1 translation and stage 2 translation into a single entry, that entry can only be shared between different PEs if the value of the CnP bit is 1 for both stage 1 and stage 2.

Note

If the value of the TTBR0.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those TTBR0s do not point to the same translation table entries when the other conditions specified for the case when the value of CnP is 1 apply, then the results of translations are CONSTRAINED UNPREDICTABLE, see *CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values* on page K1-7835.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing the TTBR0

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0010	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T2 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TRVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return TTBR0_NS<31:0>;
    else
        return TTBR0<31:0>;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return TTBR0_NS<31:0>;
    else
        return TTBR0<31:0>;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return TTBR0_S<31:0>;
    else
        return TTBR0_NS<31:0>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0010	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T2 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        TTBR0_NS = ZeroExtend(R[t]);
    else
        TTBR0 = ZeroExtend(R[t]);
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        TTBR0_NS = ZeroExtend(R[t]);
    else
        TTBR0 = ZeroExtend(R[t]);
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' && CP15SDISABLE == HIGH then
        UNDEFINED;
    else
        if SCR.NS == '0' then
            TTBR0_S = ZeroExtend(R[t]);
        else
            TTBR0_NS = ZeroExtend(R[t]);
```

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b0010	0b0000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T2 == '1' then
        AArch32.TakeHypTrapException(0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TRVM == '1' then
        AArch32.TakeHypTrapException(0x04);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return TTBR0_NS;
    else
        return TTBR0;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return TTBR0_NS;
    else
        return TTBR0;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        return TTBR0_S;
    else
        return TTBR0_NS;
```

MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b0010	0b0000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T2 == '1' then
        AArch32.TakeHypTrapException(0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
        AArch32.TakeHypTrapException(0x04);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        TTBR0_NS = R[t2]:R[t];
    else
        TTBR0 = R[t2]:R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        TTBR0_NS = R[t2]:R[t];
    else
        TTBR0 = R[t2]:R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' && CP15SDISABLE == HIGH then
        UNDEFINED;
    else
        if SCR.NS == '0' then
            TTBR0_S = R[t2]:R[t];
        else
            TTBR0_NS = R[t2]:R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.167 TTBR1, Translation Table Base Register 1

The TTBR1 characteristics are:

Purpose

Holds the base address of the translation table for the initial lookup for stage 1 of the translation of an address from the higher VA range in the PL1&0 translation regime, and other information for this translation regime.

Configurations

AArch32 System register TTBR1[63:0] is architecturally mapped to AArch64 System register TTBR1_EL1[63:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to TTBR1 are UNKNOWN.

TTBCR.EAE determines which TTBR1 format is used:

- TTBCR.EAE == `0b0`: 32-bit format is used. TTBR1[63:32] are ignored.
- TTBCR.EAE == `0b1`: 64-bit format is used.

Used in conjunction with the TTBCR. When the 64-bit TTBR1 format is used, cacheability and shareability information is held in the TTBCR, not in TTBR1.

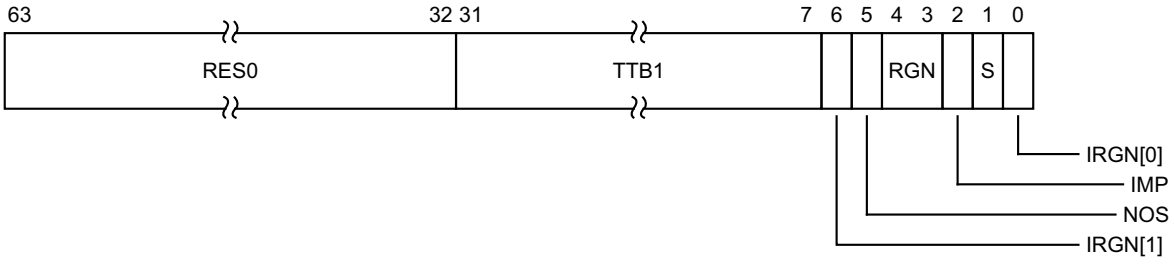
Attributes

TTBR1 is a 64-bit register that can also be accessed as a 32-bit value. If it is accessed as a 32-bit register, accesses read and write bits [31:0] and do not modify bits [63:32].

Field descriptions

The TTBR1 bit assignments are:

When TTBCR.EAE == 0:



Bits [63:32]

Reserved, RES0.

TTB1, bits [31:7]

Translation table base address, bits[31:14]. Register bits [13:7] are RES0, with the additional requirement that if these bits are not all zero, this is a misaligned translation table base address, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Register bits [13:7] are treated as if all the bits are zero. The value read back from these bits is either the value written or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

IRGN[1], bit [6]

This field is bit[1] of IRGN[1:0].

Inner region bits. IRGN[1:0] indicate the Inner Cacheability attributes for the memory associated with the translation table walks. The possible values of IRGN[1:0] are:

`0b00` Normal memory, Inner Non-cacheable.

`0b01` Normal memory, Inner Write-Back Write-Allocate Cacheable.

`0b10` Normal memory, Inner Write-Through Cacheable.

`0b11` Normal memory, Inner Write-Back no Write-Allocate Cacheable.

Note

The encoding of the IRGN bits is counter-intuitive, with register bit[6] being IRGN[0] and register bit[0] being IRGN[1]. This encoding is chosen to give a consistent encoding of memory region types and to ensure that software written for Armv7 without the Multiprocessing Extensions can run unmodified on an implementation that includes the functionality introduced by the ARMv7 Multiprocessing Extensions.

The IRGN field is split as follows:

- IRGN[1] is TTBR1[6].
- IRGN[0] is TTBR1[0].

This field resets to an architecturally UNKNOWN value.

NOS, bit [5]

Not Outer Shareable. When the value of TTBR1.S is 1, indicates whether the memory associated with a translation table walk is Inner Shareable or Outer Shareable:

`0b0` Memory is Outer Shareable.

`0b1` Memory is Inner Shareable.

This bit is ignored when the value of TTBR1.S is 0.

This field resets to an architecturally UNKNOWN value.

RGN, bits [4:3]

Region bits. Indicates the Outer cacheability attributes for the memory associated with the translation table walks:

`0b00` Normal memory, Outer Non-cacheable.

`0b01` Normal memory, Outer Write-Back Write-Allocate Cacheable.

`0b10` Normal memory, Outer Write-Through Cacheable.

`0b11` Normal memory, Outer Write-Back no Write-Allocate Cacheable.

This field resets to an architecturally UNKNOWN value.

IMP, bit [2]

The effect of this bit is IMPLEMENTATION DEFINED. If the translation table implementation does not include any IMPLEMENTATION DEFINED features this bit is RES0.

This field resets to an architecturally UNKNOWN value.

S, bit [1]

Shareable. Indicates whether the memory associated with the translation table walks is Non-shareable:

`0b0` Memory is Non-shareable.

`0b1` Memory is shareable. The TTBR1.NOS field indicates whether the memory is Inner Shareable or Outer Shareable.

This field resets to an architecturally UNKNOWN value.

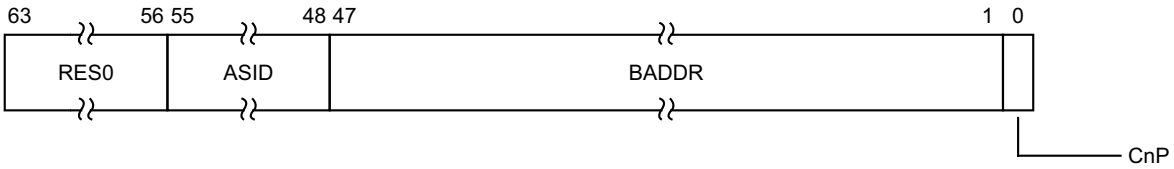
Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

IRGN[0], bit [0]

This field is bit[0] of IRGN[1:0].

See IRGN[1] for the field description.

When TTBCR.EAE == 1:



Bits [63:56]

Reserved, RES0.

ASID, bits [55:48]

An ASID for the translation table base address. The TTBCR.A1 field selects either TTBR0.ASID or TTBR1.ASID.

This field resets to an architecturally UNKNOWN value.

BADDR, bits [47:1]

Translation table base address, bits[47:x], Bits [x-1:1] are RES0, with the additional requirement that if bits[x-1:3] are not all zero, this is a misaligned translation table base address, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Register bits [x-1:3] are treated as if all the bits are zero. The value read back from these bits is either the value written or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

x is determined from the value of TTBCR.T1SZ as follows:

- If TTBCR.T1SZ is 0 or 1, x = 5 - TTBCR.T1SZ.
- If TTBCR.T1SZ is greater than 1, x = 14 - TTBCR.T1SZ.

If bits[47:40] of the translation table base address are not zero, an Address size fault is generated.

This field resets to an architecturally UNKNOWN value.

CnP, bit [0]

When ARMv8.2-TTCNP is implemented:

Common not Private. When TTBCR.EAE ==1, this bit indicates whether each entry that is pointed to by TTBR1 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of TTBR1.CnP is 1.

`0b0` The translation table entries pointed to by this instance of TTBR1, for the current ASID, are permitted to differ from corresponding entries for this instance of TTBR1 for other PEs in the Inner Shareable domain. This is not affected by:

- The value of TTBR1.CnP on those other PEs.
- The value of TTBR1.EAE on those other PEs.
- The value of the current ASID or, for the Non-secure instance of TTBR1, the value of the current VMID.

`0b1` The translation table entries pointed to by this instance of TTBR1 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of TTBR1.CnP is 1 for this instance of TTBR1 and all of the following apply:

- The translation table entries are pointed to by this instance of TTBR1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The value of the applicable TTBCR.EAE field is 1.
- The ASID is the same as the current ASID.
- For the Non-secure instance of TTBR1, the VMID is the same as the current VMID.

When a TLB combines entries from stage 1 translation and stage 2 translation into a single entry, that entry can only be shared between different PEs if the value of the CnP bit is 1 for both stage 1 and stage 2.

Note

If the value of the TTBR1.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those TTBR1s do not point to the same translation table entries when the other conditions specified for the case when the value of CnP is 1 apply, then the results of translations are CONSTRAINED UNPREDICTABLE, see *CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values* on page K1-7835.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing the TTBR1

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0010	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T2 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TRVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return TTBR1_NS<31:0>;
    else
        return TTBR1<31:0>;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return TTBR1_NS<31:0>;
    else
        return TTBR1<31:0>;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return TTBR1_S<31:0>;
    else
        return TTBR1_NS<31:0>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0010	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T2 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        TTBR1_NS = ZeroExtend(R[t]);
    else
        TTBR1 = ZeroExtend(R[t]);
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        TTBR1_NS = ZeroExtend(R[t]);
    else
        TTBR1 = ZeroExtend(R[t]);
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' && CP15SDISABLE == HIGH then
        UNDEFINED;
    elsif SCR.NS == '0' && CP15SDISABLE2 == HIGH then
        UNDEFINED;
    else
        if SCR.NS == '0' then
            TTBR1_S = ZeroExtend(R[t]);
        else
            TTBR1_NS = ZeroExtend(R[t]);
```

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b0010	0b0001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T2 == '1' then
        AArch32.TakeHypTrapException(0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TRVM == '1' then
        AArch32.TakeHypTrapException(0x04);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return TTBR1_NS;
    else
        return TTBR1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return TTBR1_NS;
    else
        return TTBR1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return TTBR1_S;
    else
        return TTBR1_NS;
```

MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b0010	0b0001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T2 == '1' then
        AArch32.TakeHypTrapException(0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
        AArch32.TakeHypTrapException(0x04);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        TTBR1_NS = R[t2]:R[t];
    else
        TTBR1 = R[t2]:R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        TTBR1_NS = R[t2]:R[t];
    else
        TTBR1 = R[t2]:R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' && CP15SDISABLE == HIGH then
        UNDEFINED;
    else
        if SCR.NS == '0' then
            TTBR1_S = R[t2]:R[t];
        else
            TTBR1_NS = R[t2]:R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.168 VBAR, Vector Base Address Register

The VBAR characteristics are:

Purpose

When high exception vectors are not selected, holds the vector base address for exceptions that are not taken to Monitor mode or to Hyp mode.

Software must program VBAR(NS) with the required initial value as part of the PE boot sequence.

Configurations

AArch32 System register VBAR[31:0] is architecturally mapped to AArch64 System register VBAR_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to VBAR are UNKNOWN.

Attributes

VBAR is a 32-bit register.

Field descriptions

The VBAR bit assignments are:

31 … 5	4 … 0
Vector Base Address	RES0

Bits [31:5]

Vector Base Address. Bits[31:5] of the base address of the exception vectors for exceptions taken to this Exception level. Bits[4:0] of an exception vector are the exception offset.

This field resets to an IMPLEMENTATION DEFINED value.

Bits [4:0]

Reserved, RES0.

Accessing the VBAR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1100	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T12 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T12 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return VBAR_NS;
    else
        return VBAR;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return VBAR_NS;
    else
        return VBAR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return VBAR_S;
    else
        return VBAR_NS;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1100	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T12 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T12 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        VBAR_NS = R[t];
    else
        VBAR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        VBAR_NS = R[t];
    else
        VBAR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' && CP15SDISABLE == HIGH then
        UNDEFINED;
    elsif SCR.NS == '0' && CP15SDISABLE2 == HIGH then
        UNDEFINED;
    else
        if SCR.NS == '0' then
            VBAR_S = R[t];
        else
            VBAR_NS = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.169 VMPIDR, Virtualization Multiprocessor ID Register

The VMPIDR characteristics are:

Purpose

Holds the value of the Virtualization Multiprocessor ID. This is the value returned by Non-secure EL1 reads of MPIDR.

Configurations

AArch32 System register VMPIDR[31:0] is architecturally mapped to AArch64 System register VMPIDR_EL2[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to VMPIDR are UNKNOWN.

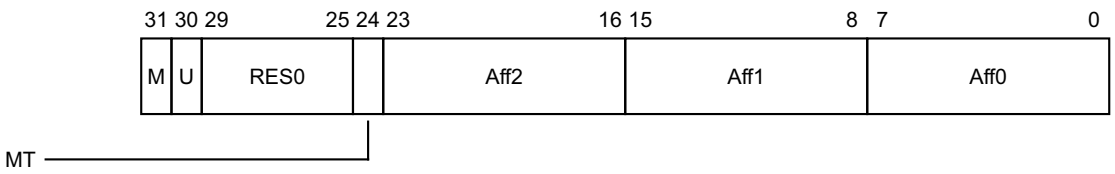
If EL2 is not implemented but EL3 is implemented, this register takes the value of the MPIDR.

Attributes

VMPIDR is a 32-bit register.

Field descriptions

The VMPIDR bit assignments are:



M, bit [31]

Indicates whether this implementation includes the functionality introduced by the ARMv7 Multiprocessing Extensions. The possible values of this bit are:

`0b0` This implementation does not include the ARMv7 Multiprocessing Extensions functionality.

`0b1` This implementation includes the ARMv7 Multiprocessing Extensions functionality.

From Armv8 this bit is RES1.

U, bit [30]

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system. The possible values of this bit are:

`0b0` Processor is part of a multiprocessor system.

`0b1` Processor is part of a uniprocessor system.

In a system where the PE resets into EL2 or EL3, this field resets to the value in MPIDR.U.

Bits [29:25]

Reserved, RES0.

MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using a multithreading type approach. See the description of Aff0 for more information about affinity levels. The possible values of this bit are:

`0b0` Performance of PEs at the lowest affinity level is largely independent.

`0b1` Performance of PEs at the lowest affinity level is very interdependent.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In a system where the PE resets into EL2 or EL3, this field resets to the value in MPIDR.MT.

Aff2, bits [23:16]

Affinity level 2. See the description of Aff0 for more information.

In a system where the PE resets into EL2 or EL3, this field resets to the value in MPIDR.Aff2.

Aff1, bits [15:8]

Affinity level 1. See the description of Aff0 for more information.

In a system where the PE resets into EL2 or EL3, this field resets to the value in MPIDR.Aff1.

Aff0, bits [7:0]

Affinity level 0. This is the affinity level that is most significant for determining PE behavior. Higher affinity levels are increasingly less significant in determining PE behavior. The assigned value of the MPIDR.{Aff2, Aff1, Aff0} or MPIDR_EL1.{Aff3, Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

In a system where the PE resets into EL2 or EL3, this field resets to the value in MPIDR.Aff0.

Accessing the VMPIDR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0000	0b0000	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return VMPIDR;
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        return MPIDR;
    elsif SCR.NS == '0' then
        UNDEFINED;
    else
        return VMPIDR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0000	0b0000	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    VMPIDR = R[t];
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        //no operation
    elsif SCR.NS == '0' then
        UNDEFINED;
    else
        VMPIDR = R[t];
```

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0000	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) then
        return VMPIDR_EL2<31:0>;
    elsif EL2Enabled() && ELUsingAArch32(EL2) then
        return VMPIDR;
    else
        return MPIDR;
elsif PSTATE.EL == EL2 then
    return MPIDR;
elsif PSTATE.EL == EL3 then
    return MPIDR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.170 VPIDR, Virtualization Processor ID Register

The VPIDR characteristics are:

Purpose

Holds the value of the Virtualization Processor ID. This is the value returned by Non-secure EL1 reads of MIDR.

Configurations

AArch32 System register VPIDR[31:0] is architecturally mapped to AArch64 System register VPIDR_EL2[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to VPIDR are UNKNOWN.

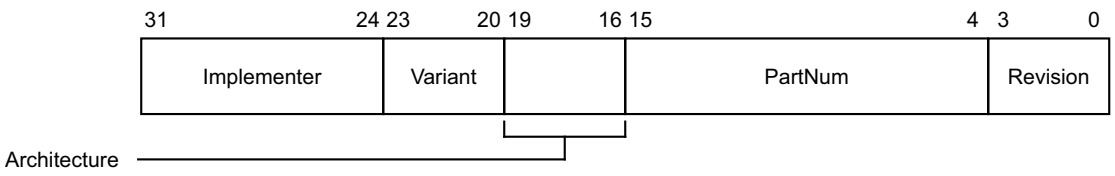
If EL2 is not implemented but EL3 is implemented, this register takes the value of the MIDR.

Attributes

VPIDR is a 32-bit register.

Field descriptions

The VPIDR bit assignments are:



Implementer, bits [31:24]

The Implementer code. This field must hold an implementer code that has been assigned by Arm. Assigned codes include the following:

Hex representation	ASCII representation	Implementer
0x41	A	Arm Limited
0x42	B	Broadcom Corporation
0x43	C	Cavium Inc.
0x44	D	Digital Equipment Corporation
0x49	I	Infineon Technologies AG
0x4D	M	Motorola or Freescale Semiconductor Inc.
0x4E	N	NVIDIA Corporation
0x50	P	Applied Micro Circuits Corporation
0x51	Q	Qualcomm Inc.
0x56	V	Marvell International Ltd.
0x69	i	Intel Corporation

Arm can assign codes that are not published in this manual. All values not assigned by Arm are reserved and must not be used.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In a system where the PE resets into EL2 or EL3, this field resets to the value in MIDR.Implementer.

Variant, bits [23:20]

An IMPLEMENTATION DEFINED variant number. Typically, this field is used to distinguish between different product variants, or major revisions of a product.

In a system where the PE resets into EL2 or EL3, this field resets to the value in MIDR.Variant.

Architecture, bits [19:16]

The permitted values of this field are:

`0b0001` Armv4.

`0b0010` Armv4T.

`0b0011` Armv5 (obsolete).

`0b0100` Armv5T.

`0b0101` Armv5TE.

`0b0110` Armv5TEJ.

`0b0111` Armv6.

`0b1111` Architectural features are individually identified in the ID_* registers, see *ID registers* on page K14-8085.

All other values are reserved.

In a system where the PE resets into EL2 or EL3, this field resets to the value in MIDR.Architecture.

PartNum, bits [15:4]

An IMPLEMENTATION DEFINED primary part number for the device.

On processors implemented by Arm, if the top four bits of the primary part number are `0x0` or `0x7`, the variant and architecture are encoded differently.

In a system where the PE resets into EL2 or EL3, this field resets to the value in MIDR.PartNum.

Revision, bits [3:0]

An IMPLEMENTATION DEFINED revision number for the device.

In a system where the PE resets into EL2 or EL3, this field resets to the value in MIDR.Revision.

Accessing the VPIDR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0000	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return VPIDR;
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        return MIDR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif SCR.NS == '0' then
        UNDEFINED;
    else
        return VPIDR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0000	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    VPIDR = R[t];
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        //no operation
    elsif SCR.NS == '0' then
        UNDEFINED;
    else
        VPIDR = R[t];
```

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) then
        return VPIDR_EL2<31:0>;
    elsif EL2Enabled() && ELUsingAArch32(EL2) then
        return VPIDR;
    else
        return MIDR;
elsif PSTATE.EL == EL2 then
    return MIDR;
elsif PSTATE.EL == EL3 then
    return MIDR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.171 VTCR, Virtualization Translation Control Register

The VTCR characteristics are:

Purpose

The control register for stage 2 of the Non-secure PL1&0 translation regime.

Note

This stage of translation always uses the Long-descriptor translation table format.

Configurations

AArch32 System register VTCR[31:0] is architecturally mapped to AArch64 System register VTCR_EL2[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to VTCR are UNKNOWN.

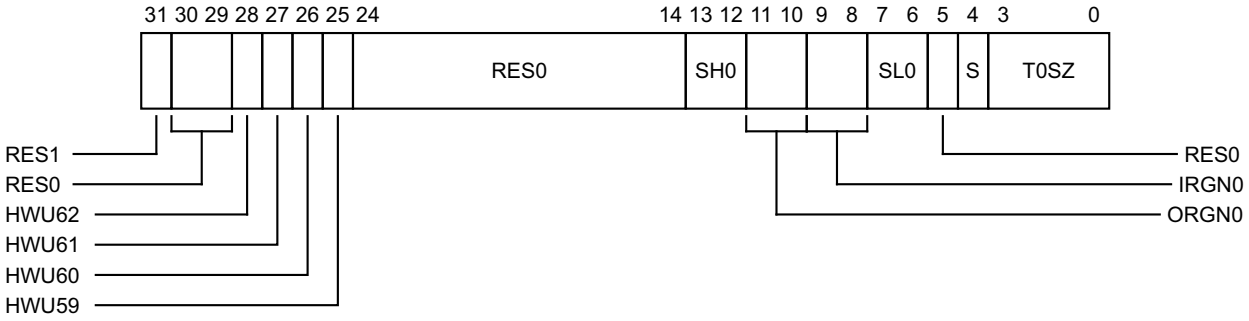
If EL2 is not implemented, this register is RES0 from EL3.

Attributes

VTCR is a 32-bit register.

Field descriptions

The VTCR bit assignments are:



Bit [31]

Reserved, RES1.

Bits [30:29]

Reserved, RES0.

HWU62, bit [28]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 2 translation table Block or Page entry.

`0b0` Bit[62] of each stage 2 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` Bit[62] of each stage 2 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

HWU61, bit [27]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 2 translation table Block or Page entry.

`0b0` Bit[61] of each stage 2 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` Bit[61] of each stage 2 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU60, bit [26]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 2 translation table Block or Page entry.

`0b0` Bit[60] of each stage 2 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` Bit[60] of each stage 2 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU59, bit [25]

When ARMv8.2-TTPBHA is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 2 translation table Block or Page entry.

`0b0` Bit[59] of each stage 2 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

`0b1` Bit[59] of each stage 2 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [24:14]

Reserved, RES0.

SH0, bits [13:12]

Shareability attribute for memory associated with translation table walks using VTTBR.

`0b00` Non-shareable.

`0b10` Outer Shareable.

`0b11` Inner Shareable.

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-7853.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ORGN0, bits [11:10]

Outer cacheability attribute for memory associated with translation table walks using VTTBR.

`0b00` Normal memory, Outer Non-cacheable.

`0b01` Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.

`0b10` Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.

`0b11` Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.

This field resets to an architecturally UNKNOWN value.

IRGN0, bits [9:8]

Inner cacheability attribute for memory associated with translation table walks using VTTBR.

`0b00` Normal memory, Inner Non-cacheable.

`0b01` Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.

`0b10` Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.

`0b11` Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

This field resets to an architecturally UNKNOWN value.

SL0, bits [7:6]

Starting level for translation table walks using VTTBR.

`0b00` Start at level 2

`0b01` Start at level 1

All other values are reserved. If this field is programmed to a reserved value, or to a value that is not consistent with the programming of T0SZ, then a stage 2 level 1 Translation fault is generated.

This field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

S, bit [4]

Sign extension bit. This bit must be programmed to the value of T0SZ[3]. If it is not, then the behavior is CONSTRAINED UNPREDICTABLE and the stage 2 T0SZ value is treated as an UNKNOWN value, see *Misprogramming VTCR.S* on page K1-7850.

This field resets to an architecturally UNKNOWN value.

T0SZ, bits [3:0]

The size offset of the memory region addressed by VTTBR. The region size is $2^{(32-T0SZ)}$ bytes.

This field holds a four-bit signed integer value, meaning it supports values from -8 to 7.

Note

This is different from the other translation control registers, where TnSZ holds a three-bit unsigned integer, supporting values from 0 to 7.

If this field is programmed to a value that is not consistent with the programming of SL0 then a stage 2 level 1 Translation fault is generated.

This field resets to an architecturally UNKNOWN value.

Accessing the VTCR

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0010	0b0001	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T2 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return VTCR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        return VTCR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0010	0b0001	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T2 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    VTCR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        VTCR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.2.172 VTTBR, Virtualization Translation Table Base Register

The VTTBR characteristics are:

Purpose

Holds the base address of the translation table for the initial lookup for stage 2 of an address translation in the Non-secure PL1&0 translation regime, and other information for this translation regime.

Configurations

AArch32 System register VTTBR[63:0] is architecturally mapped to AArch64 System register VTTBR_EL2[63:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to VTTBR are UNKNOWN.

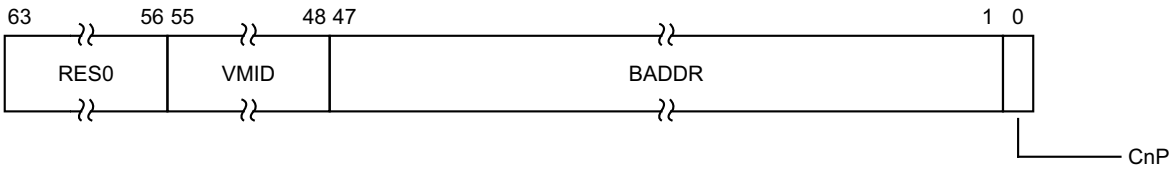
If EL2 is not implemented, this register is RES0 from EL3.

Attributes

VTTBR is a 64-bit register.

Field descriptions

The VTTBR bit assignments are:



Bits [63:56]

Reserved, RES0.

VMID, bits [55:48]

The VMID for the translation table.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

BADDR, bits [47:1]

Translation table base address, bits[47:x], Bits [x-1:1] are RES0, with the additional requirement that if bits[x-1:3] are not all zero, this is a misaligned translation table base address, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Register bits [x-1:3] are treated as if all the bits are zero. The value read back from these bits is either the value written or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

x is determined from the value of VTCR.SL0 and VTCR.T0SZ as follows:

- If VTCR.SL0 is `0b00`, meaning that lookup starts at level 2, then x is 14 - VTCR.T0SZ.
- If VTCR.SL0 is `0b01`, meaning that lookup starts at level 1, then x is 5 - VTCR.T0SZ.
- If VTCR.SL0 is either `0b10` or `0b11` then a stage 2 level 1 Translation fault is generated.

If bits[47:40] of the translation table base address are not zero, an Address size fault is generated.

In a system where the PE resets into EL2 or EL3, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CnP, bit [0]

When ARMv8.2-TTCNP is implemented:

Common not Private. This bit indicates whether each entry that is pointed to by VTTBR is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of VTTBR.CnP is 1.

`0b0` The translation table entries pointed to by VTTBR are permitted to differ from the entries for VTTBR for other PEs in the Inner Shareable domain. This is not affected by the value of the current VMID.

`0b1` The translation table entries pointed to by VTTBR are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of VTTBR.CnP is 1 and the VMID is the same as the current VMID.

When a TLB combines entries from stage 1 translation and stage 2 translation into a single entry, that entry can only be shared between different PEs if the value of the CnP bit is 1 for both stage 1 and stage 2.

Note

If the value of the VTTBR.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those VTTBRs do not point to the same translation table entries when the VMID value is the same as the current VMID, then the results of translations are CONSTRAINED UNPREDICTABLE, see *CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values* on page K1-7835.

In a system where the PE resets into EL2 or EL3, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing the VTTBR

Accesses to this register use the following encodings in the System instruction encoding space:

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b0010	0b0110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T2 == '1' then
        AArch32.TakeHypTrapException(0x04);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return VTTBR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        return VTTBR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b0010	0b0110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T2 == '1' then
        AArch32.TakeHypTrapException(0x04);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    VTTBR = R[t2]:R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        VTTBR = R[t2]:R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3 Debug registers

This section lists the Debug System registers in AArch32 state, in alphabetic order.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.1 DBGAUTHSTATUS, Debug Authentication Status register

The DBGAUTHSTATUS characteristics are:

Purpose

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for debug.

Configurations

AArch32 System register DBGAUTHSTATUS[31:0] is architecturally mapped to AArch64 System register DBGAUTHSTATUS_EL1[31:0].

AArch32 System register DBGAUTHSTATUS[31:0] is architecturally mapped to External register DBGAUTHSTATUS_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DBGAUTHSTATUS are UNKNOWN.

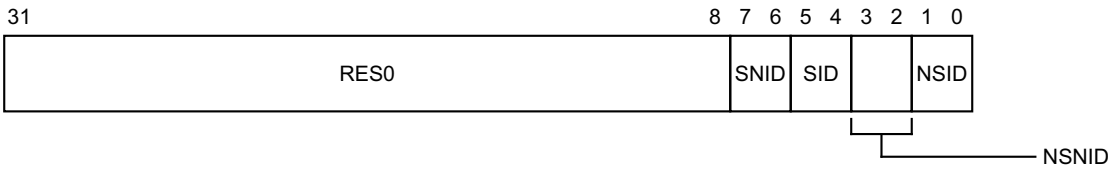
This register is required in all implementations.

Attributes

DBGAUTHSTATUS is a 32-bit register.

Field descriptions

The DBGAUTHSTATUS bit assignments are:



Bits [31:8]

Reserved, RES0.

SNID, bits [7:6]

When ARMv8.4-Debug is implemented:

Secure Non-Invasive Debug.

This field has the same value as DBGAUTHSTATUS.SID.

Otherwise:

Secure Non-Invasive Debug.

`0b00` Not implemented. EL3 is not implemented and the Effective value of SCR.NS is 1.

`0b10` Implemented and disabled. ExternalSecureNoninvasiveDebugEnabled() == FALSE.

`0b11` Implemented and enabled. ExternalSecureNoninvasiveDebugEnabled() == TRUE.

All other values are reserved.

SID, bits [5:4]

Secure Invasive Debug.

`0b00` Not implemented. EL3 is not implemented and the Effective value of SCR_EL3.NS is 1.

`0b10` Implemented and disabled. ExternalSecureInvasiveDebugEnabled() == FALSE.

`0b11` Implemented and enabled. ExternalSecureInvasiveDebugEnabled() == TRUE.

All other values are reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

NSNID, bits [3:2]

When ARMv8.4-Debug is implemented:

Non-secure Non-invasive debug.

`0b00` Not implemented. EL3 is not implemented and the Effective value of SCR.NS is 0.

`0b11` Implemented and enabled. EL3 is implemented or the Effective value of SCR.NS is 1.

All other values are reserved.

Otherwise:

Non-secure Non-Invasive Debug.

`0b00` Not implemented. EL3 is not implemented and the Effective value of SCR.NS is 0

`0b10` Implemented and disabled. ExternalNoninvasiveDebugEnabled() == FALSE.

`0b11` Implemented and enabled. ExternalNoninvasiveDebugEnabled() == TRUE.

All other values are reserved.

NSID, bits [1:0]

Non-secure Invasive Debug.

`0b00` Not implemented. EL3 is not implemented or the Effective value of SCR_EL3.NS is 0.

`0b10` Implemented and disabled. ExternalInvasiveDebugEnabled() == FALSE.

`0b11` Implemented and enabled. ExternalInvasiveDebugEnabled() == TRUE.

All other values are reserved.

Accessing the DBGAUTHSTATUS

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0111	0b1110	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGAUTHSTATUS;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGAUTHSTATUS;
elsif PSTATE.EL == EL3 then
    return DBGAUTHSTATUS;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.2 DBGBCR<n>, Debug Breakpoint Control Registers, n = 0 - 15

The DBGBCR<n> characteristics are:

Purpose

Holds control information for a breakpoint. Forms breakpoint n together with value register DBGBVR<n>. If EL2 is implemented and this breakpoint supports Context matching, DBGBVR<n> can be associated with a Breakpoint Extended Value Register DBGBXVR<n> for VMID matching.

Configurations

AArch32 System register DBGBCR<n>[31:0] is architecturally mapped to AArch64 System register DBGBCR<n>_EL1[31:0].

AArch32 System register DBGBCR<n>[31:0] is architecturally mapped to External register DBGBCR<n>_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DBGBCR<n> are UNKNOWN.

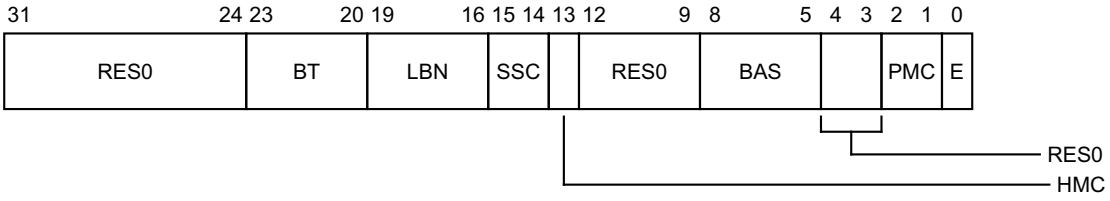
If breakpoint n is not implemented then accesses to this register are UNDEFINED.

Attributes

DBGBCR<n> is a 32-bit register.

Field descriptions

The DBGBCR<n> bit assignments are:



When the E field is zero, all the other fields in the register are ignored.

Bits [31:24]

Reserved, RES0.

BT, bits [23:20]

Breakpoint Type. Possible values are:

`0b0000` Unlinked instruction address match. DBGBVR<n> is the address of an instruction.

`0b0001` As `0b0000` with linking enabled.

`0b0010` Unlinked Context ID match. When ARMv8.1-VHE is implemented, EL2 is using AArch64, and the Effective value of HCR_EL2.E2H is 1, if either the PE is executing at EL0 with HCR_EL2.TGE set to 0 or the PE is executing at EL2, then DBGBVR<n>.ContextID must match the CONTEXTIDR_EL2 value. Otherwise DBGBVR<n>.ContextID must match the CONTEXTIDR value.

`0b0011` As `0b0010` with linking enabled.

`0b0100` Unlinked instruction address mismatch. DBGBVR<n> is the address of an instruction to be stepped.

`0b0101` As `0b0100` with linking enabled.

`0b0110` Unlinked CONTEXTIDR_EL1 match. DBGBVR<n>.ContextID is a Context ID compared against CONTEXTIDR.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b0111` As `0b0110` with linking enabled.

`0b1000` Unlinked VMID match. DBGBXVR<n>.VMID is a VMID compared against VTTBR.VMID.

`0b1001` As `0b1000` with linking enabled.

`0b1010` Unlinked VMID and Context ID match. DBGBVR<n>.ContextID is a Context ID compared against CONTEXTIDR, and DBGBXVR<n>.VMID is a VMID compared against VTTBR.VMID.

`0b1011` As `0b1010` with linking enabled.

`0b1100` Unlinked CONTEXTIDR_EL2 match. DBGBXVR<n>.ContextID2 is a Context ID compared against CONTEXTIDR_EL2.

`0b1101` As `0b1100` with linking enabled.

`0b1110` Unlinked Full Context ID match. DBGBVR<n>.ContextID is compared against CONTEXTIDR, and DBGBXVR<n>.ContextID2 is compared against CONTEXTIDR_EL2.

`0b1111` As `0b1110` with linking enabled.

For more information on Breakpoints and their constraints, see *Breakpoint exceptions* on page G2-5830 and *Reserved DBGBCR<n>.BT values* on page G2-5850.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

LBN, bits [19:16]

Linked breakpoint number. For Linked address matching breakpoints, this specifies the index of the Context-matching breakpoint linked to.

For all other breakpoint types this field is ignored and reads of the register return an UNKNOWN value.

This field is ignored when the value of DBGBCR<n>.E is 0.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

SSC, bits [15:14]

Security state control. Determines the Security states under which a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the HMC and PMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields.

For more information, see *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page G2-5840, and *Reserved DBGBCR<n>.{SSC, HMC, PMC} values* on page G2-5851.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

HMC, bit [13]

Higher mode control. Determines the debug perspective for deciding when a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the SSC and PMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information see the SSC, bits [15:14] description.

For more information on the operation of the SSC, HMC, and PMC fields, see *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page G2-5840.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bits [12:9]

Reserved, RES0.

BAS, bits [8:5]

Byte address select. Defines which half-words an address-matching breakpoint matches, regardless of the instruction set and Execution state.

The permitted values depend on the breakpoint type.

For Address match breakpoints, the permitted values are:

BAS	Match instruction at	Constraint for debuggers
0b0011	DBGBVR<n>	Use for T32 instructions
0b1100	DBGBVR<n>+2	Use for T32 instructions
0b1111	DBGBVR<n>	Use for A32 instructions

All other values are reserved. For more information, see *Reserved DBGBCR<n>.{SSC, HMC, PMC} values* on page G2-5851.

For more information on using the BAS field in Address Match breakpoints, see *Using the BAS field in Address Match breakpoints* on page G2-5843.

For Address mismatch breakpoints in an AArch32 stage 1 translation regime, the permitted values are:

BAS	Step instruction at	Constraint for debuggers
0b0000	-	Use for a match anywhere breakpoint
0b0011	DBGBVR<n>	Use for T32 instructions
0b1100	DBGBVR<n>+2	Use for T32 instructions
0b1111	DBGBVR<n>	Use for A32 instructions

All other values are reserved. For more information, see *Reserved DBGBCR<n>.{SSC, HMC, PMC} values* on page G2-5851.

For more information on using the BAS field in address mismatch breakpoints, see *Using the BAS field in Address Match breakpoints* on page G2-5843.

For Context matching breakpoints, this field is RES1 and ignored.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

Bits [4:3]

Reserved, RES0.

PMC, bits [2:1]

Privilege mode control. Determines the Exception level or levels at which a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the SSC and HMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information see the DBGBCR<n>.SSC description.

For more information on the operation of the SSC, HMC, and PMC fields, see *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page G2-5840.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

E, bit [0]

Enable breakpoint DBGBVR<n>. Possible values are:

`0b0` Breakpoint disabled.

`0b1` Breakpoint enabled.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

Accessing the DBGBCR<n>

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	n[3:0]	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    elsif ELUsingAArch32(EL1) && DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        return DBGBCR[UInt(CRm<3:0>)];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    elsif ELUsingAArch32(EL1) && DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        return DBGBCR[UInt(CRm<3:0>)];
elsif PSTATE.EL == EL3 then
    if ELUsingAArch32(EL1) && DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        return DBGBCR[UInt(CRm<3:0>)];
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	n[3:0]	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    elsif ELUsingAArch32(EL1) && DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGBCR[UInt(CRm<3:0>)] = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    elsif ELUsingAArch32(EL1) && DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGBCR[UInt(CRm<3:0>)] = R[t];
elsif PSTATE.EL == EL3 then
    if ELUsingAArch32(EL1) && DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGBCR[UInt(CRm<3:0>)] = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.3 DBGBVR<n>, Debug Breakpoint Value Registers, n = 0 - 15

The DBGBVR<n> characteristics are:

Purpose

Holds a value for use in breakpoint matching, either the virtual address of an instruction or a context ID. Forms breakpoint n together with control register DBGBCR<n>. If EL2 is implemented and this breakpoint supports Context matching, DBGBVR<n> can be associated with a Breakpoint Extended Value Register DBGBXVR<n> for VMID matching.

Configurations

AArch32 System register DBGBVR<n>[31:0] is architecturally mapped to AArch64 System register DBGBVR<n>_EL1[31:0].

AArch32 System register DBGBVR<n>[31:0] is architecturally mapped to External register DBGBVR<n>_EL1[31:0].

Note

Writes to DBGBVR<n> do not modify DBGBVR<n>_EL1[63:32].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DBGBVR<n> are UNKNOWN.

If breakpoint n is not implemented then accesses to this register are UNDEFINED.

Attributes

How this register is interpreted depends on the value of DBGBCR<n>.BT.

- When DBGBCR<n>.BT is `0b0x0x`, this register holds a virtual address.
- When DBGBCR<n>.BT is `0bxx1x`, this register holds a Context ID.

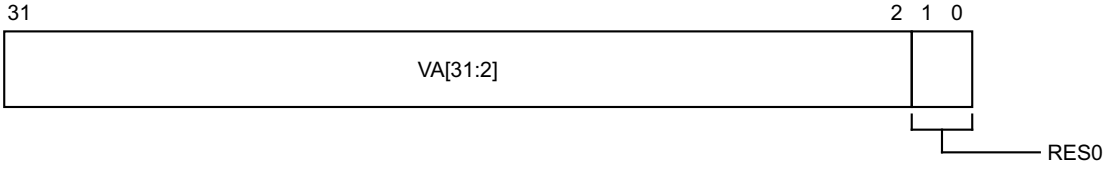
For other values of DBGBCR<n>.BT, this register is RES0.

Some breakpoints might not support Context ID comparison. For more information, see the description of the DBGDIDR.CTX_CMPs field.

Field descriptions

The DBGBVR<n> bit assignments are:

When DBGBCR<n>.BT == 0b0x0x:



VA[31:2], bits [31:2]

Bits[31:2] of the address value for comparison.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

Bits [1:0]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When DBGBCR<n>.BT == 0b001x:

31	0
ContextID	

ContextID, bits [31:0]

Context ID value for comparison.

The value is compared against CONTEXTIDR_EL2 when all of the following are true:

- CONTEXTIDR_EL2 is implemented.
- HCR_EL2.{E2H, TGE} is {1,1}.
- The PE is executing at EL0.
- EL2 is using AArch64 and is enabled in the current Security state.

Otherwise, the value is compared against CONTEXTIDR.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

When DBGBCR<n>.BT == 0b101x and EL2 is implemented:

31	0
ContextID	

ContextID, bits [31:0]

Context ID value for comparison against CONTEXTIDR.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

When DBGBCR<n>.BT == 0bx11x, EL2 is implemented and (ARMv8.1-VHE is implemented or ARMv8.2-Debug is implemented):

31	0
ContextID	

ContextID, bits [31:0]

Context ID value for comparison against CONTEXTIDR.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

Accessing the DBGBVR<n>

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	n[3:0]	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    elsif ELUsingAArch32(EL1) && DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        return DBGBVR[UInt(CRm<3:0>)];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    elsif ELUsingAArch32(EL1) && DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        return DBGBVR[UInt(CRm<3:0>)];
elsif PSTATE.EL == EL3 then
    if ELUsingAArch32(EL1) && DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        return DBGBVR[UInt(CRm<3:0>)];
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	n[3:0]	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    elsif ELUsingAArch32(EL1) && DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGBVR[UInt(CRm<3:0>)] = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    elsif ELUsingAArch32(EL1) && DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGBVR[UInt(CRm<3:0>)] = R[t];
elsif PSTATE.EL == EL3 then
    if ELUsingAArch32(EL1) && DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
else
    DBGBVR[UInt(CRm<3:0>)] = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.4 DBGBXVR<n>, Debug Breakpoint Extended Value Registers, n = 0 - 15

The DBGBXVR<n> characteristics are:

Purpose

Holds a value for use in breakpoint matching, to support VMID matching. Used in conjunction with a control register DBGBCR<n> and a value register DBGBVR<n>, where EL2 is implemented and breakpoint n supports Context matching.

Configurations

AArch32 System register DBGBXVR<n>[31:0] is architecturally mapped to AArch64 System register DBGBVR<n>_EL1[63:32].

AArch32 System register DBGBXVR<n>[31:0] is architecturally mapped to External register DBGBVR<n>_EL1[63:32].

Note

Writes to DBGBXVR<n> do not modify DBGBVR<n>_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DBGBXVR<n> are UNKNOWN.

Accesses to this register are UNDEFINED in any of the following cases:

- Breakpoint n is not implemented.
- Breakpoint n does not support Context matching.
- EL2 is not implemented.

For more information, see the description of the DBGDIDR.CTX_CMPs field.

Attributes

How this register is interpreted depends on the value of DBGBCR<n>.BT.

- When DBGBCR<n>.BT is `0b10xx`, this register holds a VMID.
- When DBGBCR<n>.BT is `0b11xx`, this register holds a Context ID.

For other values of DBGBCR<n>.BT, this register is RES0.

Field descriptions

The DBGBXVR<n> bit assignments are:

When DBGBCR<n>.BT == 0b10xx and EL2 is implemented:

31 … 16	15 … 8	7 … 0
RES0	VMID[15:8]	VMID[7:0]

Bits [31:16]

Reserved, RES0.

VMID[15:8], bits [15:8]

When ARMv8.1-VMID16 is implemented and VTCR_EL2.VS == 1:

Extension to VMID[7:0]. See VMID[7:0] for more details.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

Reserved, RES0.

VMID[7:0], bits [7:0]

VMID value for comparison. The VMID is 8 bits when any of the following are true:

- EL2 is using AArch32.
- VTCR_EL2.VS is 0.
- ARMv8.1-VMID16 is not implemented.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

When DBGBCR<n>.BT == 0b11xx and EL2 is implemented:

31	0
ContextID2	

ContextID2, bits [31:0]

When ARMv8.1-VHE is implemented or ARMv8.2-Debug is implemented:

Context ID value for comparison against CONTEXTIDR_EL2.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

Otherwise:

Reserved, RES0.

Accessing the DBGBXVR<n>

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0001	n[3:0]	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    elsif ELUsingAArch32(EL1) && DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        return DBGBXVR[UInt(CRm<3:0>)];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    elsif ELUsingAArch32(EL1) && DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        Halt(DebugHalt_SoftwareAccess);
    else
        return DBGBXVR[UInt(CRm<3:0>)];
elsif PSTATE.EL == EL3 then
    if ELUsingAArch32(EL1) && DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        return DBGBXVR[UInt(CRm<3:0>)];
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0001	n[3:0]	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    elsif ELUsingAArch32(EL1) && DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGBXVR[UInt(CRm<3:0>)] = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    elsif ELUsingAArch32(EL1) && DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGBXVR[UInt(CRm<3:0>)] = R[t];
elsif PSTATE.EL == EL3 then
    if ELUsingAArch32(EL1) && DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGBXVR[UInt(CRm<3:0>)] = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.5 DBGCLAIMCLR, Debug CLAIM Tag Clear register

The DBGCLAIMCLR characteristics are:

Purpose

Used by software to read the values of the CLAIM tag bits, and to clear CLAIM tag bits to 0.

The architecture does not define any functionality for the CLAIM tag bits.

Note

CLAIM tags are typically used for communication between the debugger and target software.

Used in conjunction with the DBGCLAIMSET register.

Configurations

AArch32 System register DBGCLAIMCLR[31:0] is architecturally mapped to AArch64 System register DBGCLAIMCLR_EL1[31:0].

AArch32 System register DBGCLAIMCLR[31:0] is architecturally mapped to External register DBGCLAIMCLR_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DBGCLAIMCLR are UNKNOWN.

An implementation must include eight CLAIM tag bits.

Attributes

DBGCLAIMCLR is a 32-bit register.

Field descriptions

The DBGCLAIMCLR bit assignments are:

31 … 8	7 … 0
RAZ/SBZ	CLAIM

Bits [31:8]

Reserved, RAZ/SBZ. Software can rely on these bits reading as zero, and must use a should-be-zero policy on writes. Implementations must ignore writes.

CLAIM, bits [7:0]

Read or clear CLAIM tag bits. Reading this field returns the current value of the CLAIM tag bits.

Writing a 1 to one of these bits clears the corresponding CLAIM tag bit to 0. This is an indirect write to the CLAIM tag bits. A single write operation can clear multiple CLAIM tag bits to 0.

Writing 0 to one of these bits has no effect.

The following resets apply:

- On a Cold reset, this field resets to `0`.
- On a Warm reset, the value of this field is unchanged.

Accessing the DBGCLAIMCLR

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0111	0b1001	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGCLAIMCLR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGCLAIMCLR;
elsif PSTATE.EL == EL3 then
    return DBGCLAIMCLR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0111	0b1001	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        DBGCLAIMCLR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        DBGCLAIMCLR = R[t];
elsif PSTATE.EL == EL3 then
    DBGCLAIMCLR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.6 DBGCLAIMSET, Debug CLAIM Tag Set register

The DBGCLAIMSET characteristics are:

Purpose

Used by software to set the CLAIM tag bits to 1.

The architecture does not define any functionality for the CLAIM tag bits.

Note

CLAIM tags are typically used for communication between the debugger and target software.

Used in conjunction with the DBGCLAIMCLR register.

Configurations

AArch32 System register DBGCLAIMSET[31:0] is architecturally mapped to AArch64 System register DBGCLAIMSET_EL1[31:0].

AArch32 System register DBGCLAIMSET[31:0] is architecturally mapped to External register DBGCLAIMSET_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DBGCLAIMSET are UNKNOWN.

An implementation must include eight CLAIM tag bits.

Attributes

DBGCLAIMSET is a 32-bit register.

Field descriptions

The DBGCLAIMSET bit assignments are:

31 … 8	7 … 0
RAZ/SBZ	CLAIM

Bits [31:8]

Reserved, RAZ/SBZ. Software can rely on these bits reading as zero, and must use a should-be-zero policy on writes. Implementations must ignore writes.

CLAIM, bits [7:0]

Set CLAIM tag bits.

This field is RAO.

Writing a 1 to one of these bits sets the corresponding CLAIM tag bit to 1. This is an indirect write to the CLAIM tag bits. A single write operation can set multiple CLAIM tag bits to 1.

Writing 0 to one of these bits has no effect.

On a Cold reset, this field resets to `0`.

Accessing the DBGCLAIMSET

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0111	0b1000	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGCLAIMSET;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGCLAIMSET;
elsif PSTATE.EL == EL3 then
    return DBGCLAIMSET;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0111	0b1000	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        DBGCLAIMSET = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        DBGCLAIMSET = R[t];
elsif PSTATE.EL == EL3 then
    DBGCLAIMSET = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.7 DBGDCCINT, DCC Interrupt Enable Register

The DBGDCCINT characteristics are:

Purpose

Enables interrupt requests to be signaled based on the DCC status flags.

Configurations

AArch32 System register DBGDCCINT[31:0] is architecturally mapped to AArch64 System register MDCCINT_EL1[31:0].

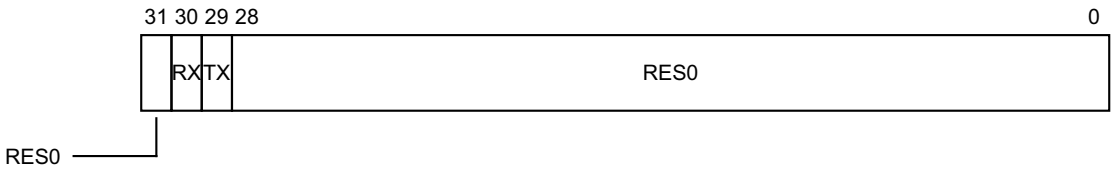
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DBGDCCINT are UNKNOWN.

Attributes

DBGDCCINT is a 32-bit register.

Field descriptions

The DBGDCCINT bit assignments are:



Bit [31]

Reserved, RES0.

RX, bit [30]

DCC interrupt request enable control for DTRRX. Enables a common **COMMIRQ** interrupt request to be signaled based on the DCC status flags.

`0b0` No interrupt request generated by DTRRX.

`0b1` Interrupt request will be generated on RXfull == 1.

If legacy **COMMRX** and **COMMTX** signals are implemented, then these are not affected by the value of this bit.

On a Warm reset, this field resets to `0`.

TX, bit [29]

DCC interrupt request enable control for DTRTX. Enables a common **COMMIRQ** interrupt request to be signaled based on the DCC status flags.

`0b0` No interrupt request generated by DTRTX.

`0b1` Interrupt request will be generated on TXfull == 0.

If legacy **COMMRX** and **COMMTX** signals are implemented, then these are not affected by the value of this bit.

On a Warm reset, this field resets to `0`.

Bits [28:0]

Reserved, RES0.

Accessing the DBGDCCINT

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TDCC == '1' then
        AArch32.TakeHypTrapException(0x05);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
        AArch32.TakeMonitorTrapException();
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGDCCINT;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
        AArch32.TakeMonitorTrapException();
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGDCCINT;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SDCR.TDCC == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return DBGDCCINT;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TDCC == '1' then
        AArch32.TakeHypTrapException(0x05);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
        AArch32.TakeMonitorTrapException();
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        DBGDCCINT = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
        AArch32.TakeMonitorTrapException();
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        DBGDCCINT = R[t];
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SDCR.TDCC == '1' then
        AArch32.TakeMonitorTrapException();
    else
        DBGDCCINT = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.8 DBGDEVID, Debug Device ID register 0

The DBGDEVID characteristics are:

Purpose

Adds to the information given by the DBGDIDR by describing other features of the debug implementation.

Configurations

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DBGDEVID are UNKNOWN.

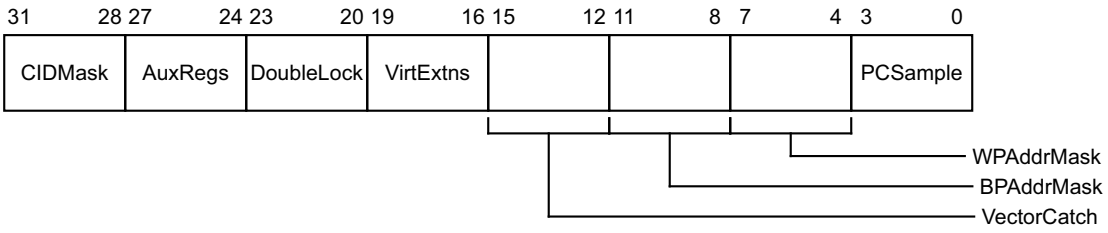
This register is required in all implementations.

Attributes

DBGDEVID is a 32-bit register.

Field descriptions

The DBGDEVID bit assignments are:



CIDMask, bits [31:28]

Indicates the level of support for the Context ID matching breakpoint masking capability. Permitted values of this field are:

`0b0000` Context ID masking is not implemented.

`0b0001` Context ID masking is implemented.

All other values are reserved. The value of this for Armv8 is `0b0000`.

AuxRegs, bits [27:24]

Indicates support for Auxiliary registers. Permitted values for this field are:

`0b0000` None supported.

`0b0001` Support for External Debug Auxiliary Control Register, EDACR.

All other values are reserved.

DoubleLock, bits [23:20]

OS Double Lock implemented. Defined values are:

`0b0000` OS Double Lock is not implemented. DBGOSDLR is RAZ/WI.

`0b0001` OS Double Lock is implemented. DBGOSDLR is RW.

ARMv8.0-DoubleLock implements the functionality identified by the value `0b0001`.

All other values are reserved.

VirtExtns, bits [19:16]

Indicates whether EL2 is implemented. Permitted values of this field are:

`0b0000` EL2 is not implemented.

`0b0001` EL2 is implemented.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

All other values are reserved.

VectorCatch, bits [15:12]

Defines the form of Vector Catch exception implemented. Permitted values of this field are:

`0b0000` Address matching Vector Catch exception implemented.

`0b0001` Exception matching Vector Catch exception implemented.

All other values are reserved.

BPAddrMask, bits [11:8]

Indicates the level of support for the instruction address matching breakpoint masking capability. Permitted values of this field are:

`0b0000` Breakpoint address masking might be implemented. If not implemented, DBGBCR<n>[28:24] is RAZ/WI.

`0b0001` Breakpoint address masking is implemented.

`0b1111` Breakpoint address masking is not implemented. DBGBCR<n>[28:24] is RES0.

All other values are reserved. The value of this for Armv8 is `0b1111`.

WPAddrMask, bits [7:4]

Indicates the level of support for the data address matching watchpoint masking capability. Permitted values of this field are:

`0b0000` Watchpoint address masking might be implemented. If not implemented, DBGWCR<n>.MASK (Address mask) is RAZ/WI.

`0b0001` Watchpoint address masking is implemented.

`0b1111` Watchpoint address masking is not implemented. DBGWCR<n>.MASK (Address mask) is RES0.

All other values are reserved. The value of this for Armv8 is `0b0001`.

PCSample, bits [3:0]

Indicates the level of PC Sample-based Profiling support using external debug registers. Permitted values of this field are:

`0b0000` PC Sample-based Profiling Extension is not implemented in the external debug registers space.

`0b0010` Only EDPCSR and EDCIDSR are implemented. This option is only permitted if EL3 and EL2 are not implemented.

`0b0011` EDPCSR, EDCIDSR, and EDVIDSR are implemented.

All other values are reserved.

When ARMv8.2-PCSample is implemented, the only permitted value is `0b0000`.

Note

ARMv8.2-PCSample implements the PC Sample-based Profiling Extension in the Performance Monitors register space, as indicated by the value of PMDEVID.PCSample.

Accessing the DBGDEVID

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0111	0b0010	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGDEVID;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGDEVID;
elsif PSTATE.EL == EL3 then
    return DBGDEVID;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.9 DBGDEVID1, Debug Device ID register 1

The DBGDEVID1 characteristics are:

Purpose

Adds to the information given by the DBGDIDR by describing other features of the debug implementation.

Configurations

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DBGDEVID1 are UNKNOWN.

This register is required in all implementations.

Attributes

DBGDEVID1 is a 32-bit register.

Field descriptions

The DBGDEVID1 bit assignments are:

31 … 4	3 … 0
RES0	PCSROffset

Bits [31:4]

Reserved, RES0.

PCSROffset, bits [3:0]

This field indicates the offset applied to PC samples returned by reads of EDPCSR. Permitted values of this field in Armv8 are:

`0b0000` EDPCSR is not implemented.

`0b0010` EDPCSR implemented. Samples have no offset applied and do not sample the instruction set state in AArch32 state.

When ARMv8.2-PCSample is implemented, the only permitted value is `0b0000`.

Note

ARMv8.2-PCSample implements the PC Sample-based Profiling Extension in the Performance Monitors register space, as indicated by the value of PMDEVID.PCSample.

Accessing the DBGDEVID1

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0111	0b0001	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
            AArch32.TakeHypTrapException(0x05);
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        else
            return DBGDEVID1;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        else
            return DBGDEVID1;
    elsif PSTATE.EL == EL3 then
        return DBGDEVID1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.10 DBGDEVID2, Debug Device ID register 2

The DBGDEVID2 characteristics are:

Purpose

Reserved for future descriptions of features of the debug implementation.

Configurations

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DBGDEVID2 are UNKNOWN.

Attributes

DBGDEVID2 is a 32-bit register.

Field descriptions

The DBGDEVID2 bit assignments are:

31	0
RES0	

Bits [31:0]

Reserved, RES0.

Accessing the DBGDEVID2

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0111	0b0000	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGDEVID2;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGDEVID2;
elsif PSTATE.EL == EL3 then
    return DBGDEVID2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.11 DBGDIDR, Debug ID Register

The DBGDIDR characteristics are:

Purpose

Specifies which version of the Debug architecture is implemented, and some features of the debug implementation.

Configurations

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DBGDIDR are UNKNOWN.

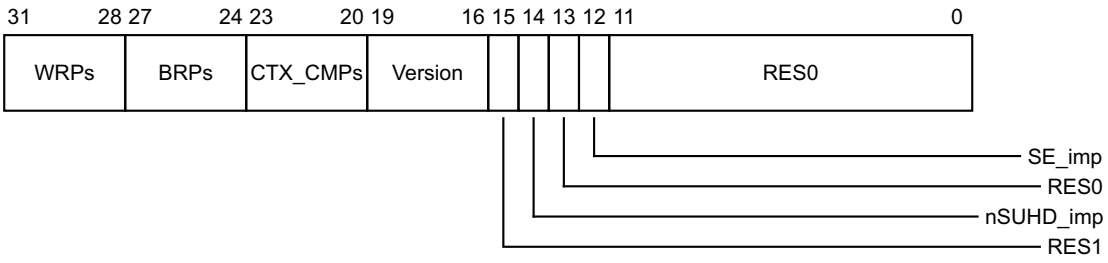
If EL1 cannot use AArch32 then the implementation of this register is OPTIONAL and deprecated.

Attributes

DBGDIDR is a 32-bit register.

Field descriptions

The DBGDIDR bit assignments are:



WRPs, bits [31:28]

The number of watchpoints implemented, minus 1.

Permitted values of this field are from `0b0001` for 2 implemented watchpoints, to `0b1111` for 16 implemented watchpoints.

The value of `0b0000` is reserved.

If AArch64 is implemented, this field has the same value as ID_AA64DFR0_EL1.WRPs.

BRPs, bits [27:24]

The number of breakpoints implemented, minus 1.

Permitted values of this field are from `0b0001` for 2 implemented breakpoint, to `0b1111` for 16 implemented breakpoints.

The value of `0b0000` is reserved.

If AArch64 is implemented, this field has the same value as ID_AA64DFR0_EL1.BRPs.

CTX_CMPs, bits [23:20]

The number of breakpoints that can be used for Context matching, minus 1.

Permitted values of this field are from `0b0000` for 1 Context matching breakpoint, to `0b1111` for 16 Context matching breakpoints.

The Context matching breakpoints must be the highest addressed breakpoints. For example, if six breakpoints are implemented and two are Context matching breakpoints, they must be breakpoints 4 and 5.

If AArch64 is implemented, this field has the same value as ID_AA64DFR0_EL1.CTX_CMPs.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Version, bits [19:16]

The Debug architecture version. Defined values are:

`0b0001` Armv6, v6 Debug architecture.

`0b0010` Armv6, v6.1 Debug architecture.

`0b0011` Armv7, v7 Debug architecture, with baseline CP14 registers implemented.

`0b0100` Armv7, v7 Debug architecture, with all CP14 registers implemented.

`0b0101` Armv7, v7.1 Debug architecture.

`0b0110` Armv8, v8 Debug architecture.

`0b0111` Armv8.1, v8 Debug architecture, with Virtualization Host Extensions.

`0b1000` Armv8.2, v8.2 Debug architecture.

`0b1001` Armv8.4, v8.4 Debug architecture.

All other values are reserved.

In any Armv8 implementation, the values `0b0001`, `0b0010`, `0b0011`, `0b0100`, and `0b0101` are not permitted.

- If ARMv8.1-VHE is not implemented, the only permitted value is `0b0110`.
- In an Armv8.0 implementation, the value `0b1000` or higher is not permitted.

Bit [15]

Reserved, RES1.

nSUHD_imp, bit [14]

In Armv7-A, was Secure User Halting Debug not implemented.

The value of this bit must match the value of the SE_imp bit.

Bit [13]

Reserved, RES0.

SE_imp, bit [12]

EL3 implemented. The meanings of the values of this bit are:

`0b0` EL3 not implemented.

`0b1` EL3 implemented.

The value of this bit must match the value of the nSUHD_imp bit.

Bits [11:0]

Reserved, RES0.

Accessing the DBGDIDR

Arm deprecates any access to this register from EL0.

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0000	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && MDSCR_EL1.TDCC == '1' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x05);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x05);
    elsif ELUsingAArch32(EL1) && DBGDSCRext.UDCCdis == '1' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x05);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (HCR_EL2.TGE == '1' || MDCR_EL2.<TDE,TDA> != '00') then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (HCR.TGE == '1' || HDCR.<TDE,TDA> != '00') then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGDIDR;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGDIDR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGDIDR;
elsif PSTATE.EL == EL3 then
    return DBGDIDR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.12 DBGDRAR, Debug ROM Address Register

The DBGDRAR characteristics are:

Purpose

Defines the base physical address of a 4KB-aligned memory-mapped debug component, usually a ROM table that locates and describes the memory-mapped debug components in the system. Armv8 deprecates any use of this register.

Configurations

AArch32 System register DBGDRAR[63:0] is architecturally mapped to AArch64 System register MDRAR_EL1[63:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DBGDRAR are UNKNOWN.

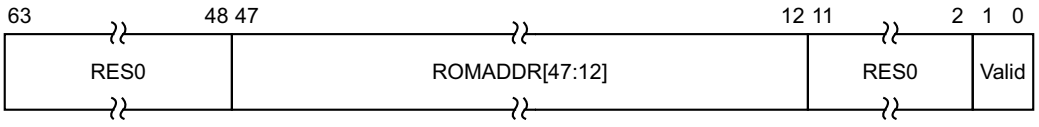
If EL1 cannot use AArch32 then the implementation of this register is OPTIONAL and deprecated.

Attributes

DBGDRAR is a 64-bit register that can also be accessed as a 32-bit value. If it is accessed as a 32-bit register, bits [31:0] are read.

Field descriptions

The DBGDRAR bit assignments are:



Bits [63:48]

Reserved, RES0.

ROMADDR[47:12], bits [47:12]

Bits[47:12] of the ROM table physical address.

If the physical address size in bits (PAsize) is less than 48 then the register bits corresponding to ROMADDR [47:PAsize] are RES0.

Bits [11:0] of the ROM table physical address are zero.

Arm strongly recommends that bits ROMADDR[(PAsize-1):32] are zero in any system that supports AArch32 at the highest implemented Exception level.

In an implementation that includes EL3, ROMADDR is an address in Non-secure memory. It is IMPLEMENTATION DEFINED whether the ROM table is also accessible in Secure memory.

Bits [11:2]

Reserved, RES0.

Valid, bits [1:0]

This field indicates whether the ROM Table address is valid. The permitted values of this field are:

`0b00` ROM Table address is not valid. Software must ignore ROMADDR.

`0b11` ROM Table address is valid.

Other values are reserved.

Accessing the DBGDRAR

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && MDSCR_EL1.TDCC == '1' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x05);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x05);
    elsif ELUsingAArch32(EL1) && DBGDSCRext.UDCCdis == '1' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x05);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (HCR_EL2.TGE == '1' || MDCR_EL2.<TDE,TDRA> != '00')
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && (HCR.TGE == '1' || HDCR.<TDE,TDRA> != '00') then
        AArch32.TakeHypTrapException(0x05);
    else
        return DBGDRAR<31:0>;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDRA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDRA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    else
        return DBGDRAR<31:0>;
elsif PSTATE.EL == EL2 then
    return DBGDRAR<31:0>;
elsif PSTATE.EL == EL3 then
    return DBGDRAR<31:0>;
```

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1110	0b0001	0b0000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && MDSCR_EL1.TDCC == '1' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x0C);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x0C);
    elsif ELUsingAArch32(EL1) && DBGDSCRext.UDCCdis == '1' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x0C);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (HCR_EL2.TGE == '1' || MDCR_EL2.<TDE,TDRA> != '00')
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x0C);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && (HCR.TGE == '1' || HDCR.<TDE,TDRA> != '00') then
        AArch32.TakeHypTrapException(0x0C);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        return DBGDRAR;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDRA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x0C);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDRA> != '00' then
        AArch32.TakeHypTrapException(0x0C);
    else
        return DBGDRAR;
elsif PSTATE.EL == EL2 then
    return DBGDRAR;
elsif PSTATE.EL == EL3 then
    return DBGDRAR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.13 DBGDSAR, Debug Self Address Register

The DBGDSAR characteristics are:

Purpose

In earlier versions of the Arm Architecture, this register defines the offset from the base address defined in DBGDRAR of the physical base address of the debug registers for the PE. Armv8 deprecates any use of this register.

Configurations

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DBGDSAR are UNKNOWN.

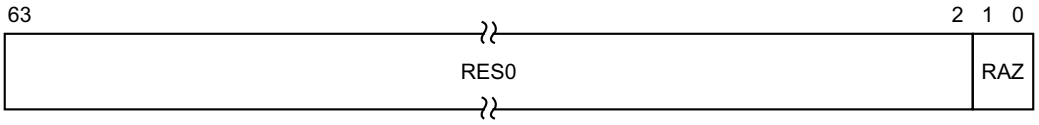
If EL1 cannot use AArch32 then the implementation of this register is OPTIONAL and deprecated.

Attributes

DBGDSAR is a 64-bit register that can also be accessed as a 32-bit value. If it is accessed as a 32-bit register, bits [31:0] are read.

Field descriptions

The DBGDSAR bit assignments are:



Bits [63:2]

Reserved, RES0.

Bits [1:0]

Reserved, RAZ.

This field indicates whether the debug self address offset is valid. For ARMv8, this field is always `0b00`, the offset is not valid.

Accessing the DBGDSAR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0010	0b0000	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && MDSCR_EL1.TDCC == '1' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x05);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x05);
    elsif ELUsingAArch32(EL1) && DBGDSCRext.UDCCdis == '1' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x05);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (HCR_EL2.TGE == '1' || MDCR_EL2.<TDE,TDRA> != '00')
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && (HCR.TGE == '1' || HDCR.<TDE,TDRA> != '00') then
        AArch32.TakeHypTrapException(0x05);
    else
        return DBGDSAR<31:0>;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDRA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDRA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    else
        return DBGDSAR<31:0>;
elsif PSTATE.EL == EL2 then
    return DBGDSAR<31:0>;
elsif PSTATE.EL == EL3 then
    return DBGDSAR<31:0>;
```

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1110	0b0010	0b0000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && MDSCR_EL1.TDCC == '1' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x0C);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x0C);
    elsif ELUsingAArch32(EL1) && DBGDSCRext.UDCCdis == '1' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x0C);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (HCR_EL2.TGE == '1' || MDCR_EL2.<TDE,TDRA> != '00')
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x0C);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && (HCR.TGE == '1' || HDCR.<TDE,TDRA> != '00') then
        AArch32.TakeHypTrapException(0x0C);
    else
        return DBGDSAR;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDRA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x0C);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDRA> != '00' then
        AArch32.TakeHypTrapException(0x0C);
    else
        return DBGDSAR;
elsif PSTATE.EL == EL2 then
    return DBGDSAR;
elsif PSTATE.EL == EL3 then
    return DBGDSAR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.14 DBGDSCRext, Debug Status and Control Register, External View

The DBGDSCRext characteristics are:

Purpose

Main control register for the debug implementation.

Configurations

AArch32 System register DBGDSCRext[31:0] is architecturally mapped to AArch64 System register MDSCR_EL1[31:0].

AArch32 System register DBGDSCRext[15:2] is architecturally mapped to AArch32 System register DBGDSCRint[15:2].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DBGDSCRext are UNKNOWN.

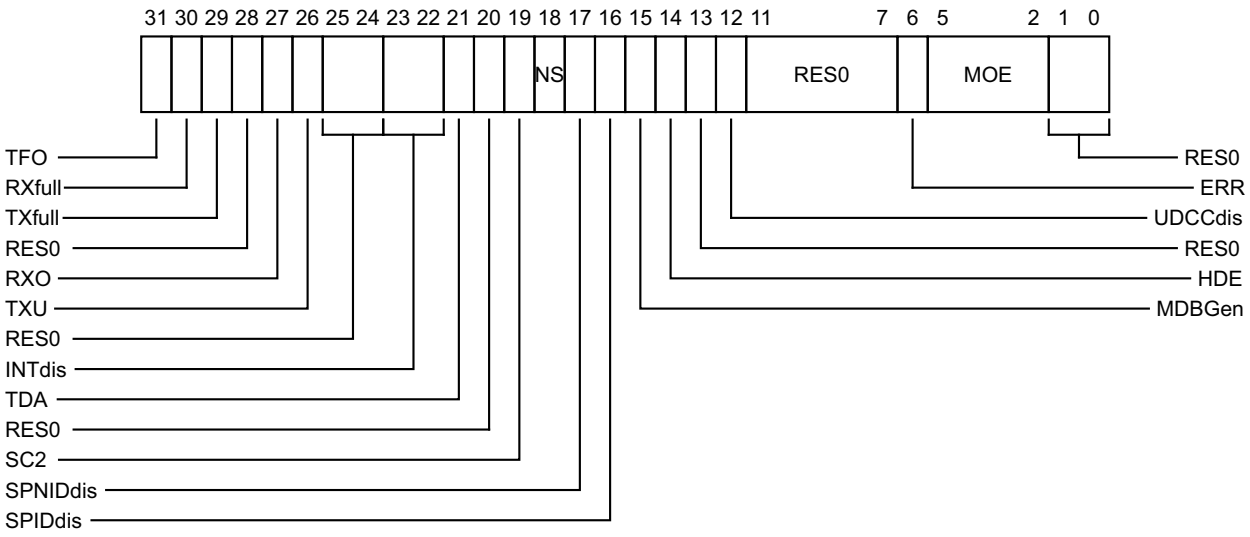
This register is required in all implementations.

Attributes

DBGDSCRext is a 32-bit register.

Field descriptions

The DBGDSCRext bit assignments are:



TFO, bit [31]

When ARMv8.4-Trace is implemented:

Trace Filter override. Used for save/restore of EDSCR.TFO.

When the OS Lock is unlocked, DBGOSLSR.OSLK == 0, software must treat this bit as UNK/SBZP.

When the OS Lock is locked, DBGOSLSR.OSLK == 1, this bit holds the value of EDSCR.TFO. Reads and writes of this bit are indirect accesses to EDSCR.TFO.

Accessing this field has the following behavior:

- When DBGOSLSR.OSLK == 1, access to this field is RW.
- When DBGOSLSR.OSLK == 0, access to this field is RO.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

RXfull, bit [30]

DTRRX full. Used for save/restore of EDSCR.RXfull.

When DBGOSLSR.OSLK == 0, software must treat this bit as UNK/SBZP.

When DBGOSLSR.OSLK == 1, this bit holds the value of EDSCR.RXfull. Reads and writes of this bit are indirect accesses to EDSCR.RXfull.

Arm deprecates use of this bit other than for save/restore. Use DBGDSCRint to access the DTRRX full status.

The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:

- When DBGOSLSR.OSLK == 1, access to this field is RW.
- When DBGOSLSR.OSLK == 0, access to this field is RO.

TXfull, bit [29]

DTRTX full. Used for save/restore of EDSCR.TXfull.

When DBGOSLSR.OSLK == 0, software must treat this bit as UNK/SBZP.

When DBGOSLSR.OSLK == 1, this bit holds the value of EDSCR.TXfull. Reads and writes of this bit are indirect accesses to EDSCR.TXfull.

Arm deprecates use of this bit other than for save/restore. Use DBGDSCRint to access the DTRTX full status.

The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:

- When DBGOSLSR.OSLK == 1, access to this field is RW.
- When DBGOSLSR.OSLK == 0, access to this field is RO.

Bit [28]

Reserved, RES0.

RXO, bit [27]

Used for save/restore of EDSCR.RXO.

When DBGOSLSR.OSLK == 0, software must treat this bit as UNK/SBZP.

When DBGOSLSR.OSLK == 1, this bit holds the value of EDSCR.RXO. Reads and writes of this bit are indirect accesses to EDSCR.RXO.

The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:

- When DBGOSLSR.OSLK == 1, access to this field is RW.
- When DBGOSLSR.OSLK == 0, access to this field is RO.

TXU, bit [26]

Used for save/restore of EDSCR.TXU.

When DBGOSLSR.OSLK == 0, software must treat this bit as UNK/SBZP.

When DBGOSLSR.OSLK == 1, this bit holds the value of EDSCR.TXU. Reads and writes of this bit are indirect accesses to EDSCR.TXU.

The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:

- When DBGOSLSR.OSLK == 1, access to this field is RW.
- When DBGOSLSR.OSLK == 0, access to this field is RO.

Bits [25:24]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

INTdis, bits [23:22]

Used for save/restore of EDSCR.INTdis.

When DBGOSLSR.OSLK == 0, this field is RO, and software must treat it as UNK/SBZP.

When DBGOSLSR.OSLK == 1, this field is RW and holds the value of EDSCR.INTdis. Reads and writes of this field are indirect accesses to EDSCR.INTdis.

The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:

- When DBGOSLSR.OSLK == 1, access to this field is RW.
- When DBGOSLSR.OSLK == 0, access to this field is RO.

TDA, bit [21]

Used for save/restore of EDSCR.TDA.

When DBGOSLSR.OSLK == 0, software must treat this bit as UNK/SBZP.

When DBGOSLSR.OSLK == 1, this bit holds the value of EDSCR.TDA. Reads and writes of this bit are indirect accesses to EDSCR.TDA.

The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:

- When DBGOSLSR.OSLK == 1, access to this field is RW.
- When DBGOSLSR.OSLK == 0, access to this field is RO.

Bit [20]

Reserved, RES0.

SC2, bit [19]

When ARMv8.0-PCSample is implemented, ARMv8.1-VHE is implemented and ARMv8.2-PCSample is not implemented:

Used for save/restore of EDSCR.SC2.

When DBGOSLSR.OSLK == 0, software must treat this bit as UNK/SBZP.

When DBGOSLSR.OSLK == 1, this bit holds the value of EDSCR.SC2. Reads and writes of this bit are indirect accesses to EDSCR.SC2.

Accessing this field has the following behavior:

- When DBGOSLSR.OSLK == 1, access to this field is RW.
- When DBGOSLSR.OSLK == 0, access to this field is RO.

Otherwise:

Reserved, RES0.

NS, bit [18]

Non-secure status. Returns the inverse of IsSecure().

Arm deprecates use of this field.

Access to this field is RO.

SPNIDdis, bit [17]

When EL3 is implemented:

Secure privileged profiling disabled status bit.

`0b0` Profiling allowed in Secure privileged modes.

`0b1` Profiling prohibited in Secure privileged modes.

This field reads as 0 if any of the following applies, and reads as 1 otherwise:

- ARMv8.2-Debug is not implemented and `ExternalSecureNoninvasiveDebugEnabled()` returns TRUE.
- EL3 is using AArch32 and the value of SDCR.SPME is 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- EL3 is using AArch64 and the value of MDCR_EL3.SPME is 1.

Arm deprecates use of this field.

Access to this field is RO.

Otherwise:

Reserved, RES0.

SPIDdis, bit [16]

When EL3 is implemented:

Secure privileged AArch32 invasive self-hosted debug disabled status bit. The value of this bit depends on the value of SDCR.SPD and the pseudocode function `AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()`.

`0b0` Self-hosted debug enabled in Secure privileged AArch32 modes.

`0b1` Self-hosted debug disabled in Secure privileged AArch32 modes.

This bit reads as 1 if any of the following is true and reads as 0 otherwise:

- EL3 is using AArch32 and SDCR.SPD has the value `0b10`.
- EL3 is using AArch64 and MDCR_EL3.SPD32 has the value `0b10`.
- EL3 is using AArch32, SDCR.SPD has the value `0b00`, and `AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()` returns FALSE.
- EL3 is using AArch64, MDCR_EL3.SPD32 has the value `0b00`, and `AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()` returns FALSE.

Arm deprecates use of this field.

Access to this field is RO.

Otherwise:

Reserved, RES0.

MDBGen, bit [15]

Monitor debug events enable. Enable Breakpoint, Watchpoint, and Vector Catch exceptions.

`0b0` Breakpoint, Watchpoint, and Vector Catch exceptions disabled.

`0b1` Breakpoint, Watchpoint, and Vector Catch exceptions enabled.

On a Warm reset, this field resets to `0`.

HDE, bit [14]

Used for save/restore of EDSCR.HDE.

When DBGOSLSR.OSLK == 0, software must treat this bit as UNK/SBZP.

When DBGOSLSR.OSLK == 1, this bit holds the value of EDSCR.HDE. Reads and writes of this bit are indirect accesses to EDSCR.HDE.

The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:

- When DBGOSLSR.OSLK == 1, access to this field is RW.
- When DBGOSLSR.OSLK == 0, access to this field is RO.

Bit [13]

Reserved, RES0.

UDCCdis, bit [12]

Traps EL0 accesses to the DCC registers to Undefined mode.

`0b0` This control does not cause any instructions to be trapped.

`0b1` EL0 accesses to the DBGDSCRint, DBGDTRRXint, DBGDTRTXint, DBGDIDR, DBGDSAR, and DBGDRAR are trapped to Undefined mode.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

All accesses to these registers are trapped, including LDC and STC accesses to DBGDTRTXint and DBGDTRRXint, and MRRC accesses to DBGDSAR and DBGDRAR.

Traps of EL0 accesses to the DBGDTRRXint and DBGDTRTXint are ignored in Debug state.

On a Warm reset, this field resets to `0`.

Bits [11:7]

Reserved, RES0.

ERR, bit [6]

Used for save/restore of EDSCR.ERR.

When DBGOSLSR.OSLK == 0, software must treat this bit as UNK/SBZP.

When DBGOSLSR.OSLK == 1, this bit holds the value of EDSCR.ERR. Reads and writes of this bit are indirect accesses to EDSCR.ERR.

The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:

- When DBGOSLSR.OSLK == 1, access to this field is RW.
- When DBGOSLSR.OSLK == 0, access to this field is RO.

MOE, bits [5:2]

Method of Entry for debug exception. When a debug exception is taken to an Exception level using AArch32, this field is set to indicate the event that caused the exception:

`0b0001` Breakpoint.

`0b0011` Software breakpoint (BKPT) instruction.

`0b0101` Vector catch.

`0b1010` Watchpoint.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [1:0]

Reserved, RES0.

Accessing the DBGDSCRext

Individual fields within this register might have restricted accessibility when the OS lock is unlocked, DBGOSLSR.OSLK == 0. See the field descriptions for more detail.

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0010	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        return DBGDSCRext;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGDSCRext;
elsif PSTATE.EL == EL3 then
    return DBGDSCRext;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0010	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        DBGDSCRext = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        DBGDSCRext = R[t];
elsif PSTATE.EL == EL3 then
    DBGDSCRext = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.15 DBGDSCRint, Debug Status and Control Register, Internal View

The DBGDSCRint characteristics are:

Purpose

Main control register for the debug implementation. This is an internal, read-only view.

Configurations

AArch32 System register DBGDSCRint[30:29] is architecturally mapped to AArch64 System register MDCCSR_EL0[30:29].

AArch32 System register DBGDSCRint[15:2] is architecturally mapped to AArch32 System register DBGDSCRext[15:2].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DBGDSCRint are UNKNOWN.

This register is required in all implementations.

DBGDSCRint.{NS, SPNIDdis, SPIDdis, MDBGen, UDCCdis, MOE} are UNKNOWN when the register is accessed at EL0. However, although these values are not accessible at EL0 by instructions that are neither UNPREDICTABLE nor return UNKNOWN values, it is permissible for an implementation to return the values of DBGDSCRext.{NS, SPNIDdis, SPIDdis, MDBGen, UDCCdis, MOE} for these fields at EL0.

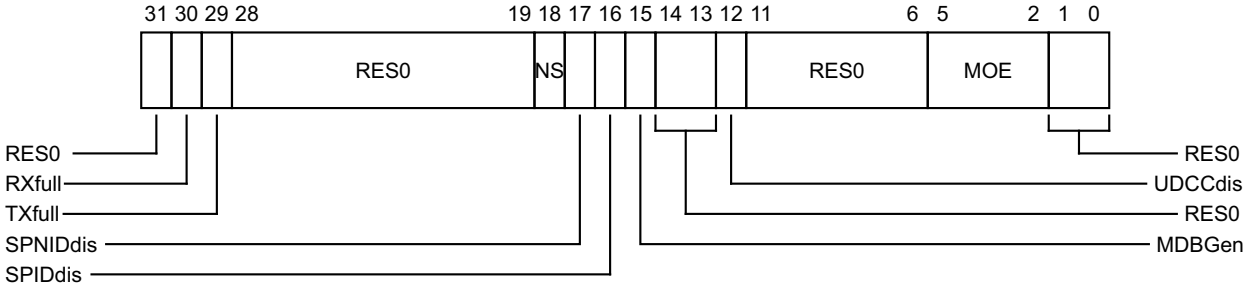
It is also permissible for an implementation to return the same values as defined for a read of DBGDSCRint at EL1 or above. (This is the case even if the implementation does not support AArch32 at EL1 or above.)

Attributes

DBGDSCRint is a 32-bit register.

Field descriptions

The DBGDSCRint bit assignments are:



Bit [31]

Reserved, RES0.

RXfull, bit [30]

DTRRX full. Read-only view of the equivalent bit in the EDSCR.

TXfull, bit [29]

DTRTX full. Read-only view of the equivalent bit in the EDSCR.

Bits [28:19]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

NS, bit [18]

Non-secure status.

Read-only view of the equivalent bit in the DBGDSCRext. Arm deprecates use of this field.

SPNIDdis, bit [17]

Secure privileged non-invasive debug disable.

Read-only view of the equivalent bit in the DBGDSCRext. Arm deprecates use of this field.

SPIDdis, bit [16]

Secure privileged invasive debug disable.

Read-only view of the equivalent bit in the DBGDSCRext. Arm deprecates use of this field.

MDBGen, bit [15]

Monitor debug events enable.

Read-only view of the equivalent bit in the DBGDSCRext.

Bits [14:13]

Reserved, RES0.

UDCCdis, bit [12]

User mode access to Debug Communications Channel disable.

Read-only view of the equivalent bit in the DBGDSCRext. Arm deprecates use of this field.

Bits [11:6]

Reserved, RES0.

MOE, bits [5:2]

Method of Entry for debug exception. When a debug exception is taken to an Exception level using AArch32, this field is set to indicate the event that caused the exception:

`0b0001` Breakpoint

`0b0011` Software breakpoint (BKPT) instruction

`0b0101` Vector catch

`0b1010` Watchpoint

Read-only view of the equivalent bit in the DBGDSCRext.

Bits [1:0]

Reserved, RES0.

Accessing the DBGDSCRint

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0001	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && MDSCR_EL1.TDCC == '1' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x05);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x05);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        elsif ELUsingAArch32(EL1) && DBGDSCRext.UDCCdis == '1' then
            if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
                AArch64.AArch32SystemAccessTrap(EL2, 0x05);
            elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
                AArch32.TakeHypTrapException(0x00);
            else
                UNDEFINED;
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x05);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TDCC == '1' then
            AArch32.TakeHypTrapException(0x05);
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && (HCR_EL2.TGE == '1' || MDCR_EL2.<TDE,TDA> != '00') then
            AArch64.AArch32SystemAccessTrap(EL2, 0x05);
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && (HCR.TGE == '1' || HDCR.<TDE,TDA> != '00') then
            AArch32.TakeHypTrapException(0x05);
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
            AArch32.TakeMonitorTrapException();
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        else
            return DBGDSCRint;
    elsif PSTATE.EL == EL1 then
        if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x05);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TDCC == '1' then
            AArch32.TakeHypTrapException(0x05);
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x05);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
            AArch32.TakeHypTrapException(0x05);
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
            AArch32.TakeMonitorTrapException();
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        else
            return DBGDSCRint;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
            AArch32.TakeMonitorTrapException();
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        else
            return DBGDSCRint;
    elsif PSTATE.EL == EL3 then
        if PSTATE.M != M32_Monitor && SDCR.TDCC == '1' then
            AArch32.TakeMonitorTrapException();
        else
            return DBGDSCRint;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.16 DBGDTRRXext, Debug OS Lock Data Transfer Register, Receive, External View

The DBGDTRRXext characteristics are:

Purpose

Used for save/restore of DBGDTRRXint. It is a component of the Debug Communications Channel.

Configurations

AArch32 System register DBGDTRRXext[31:0] is architecturally mapped to AArch64 System register OSDTRRX_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DBGDTRRXext are UNKNOWN.

Attributes

DBGDTRRXext is a 32-bit register.

Field descriptions

The DBGDTRRXext bit assignments are:

31 … 0
Update DTRRX without side-effect

Bits [31:0]

Update DTRRX without side-effect.

Writes to this register update the value in DTRRX and do not change RXfull.

Reads of this register return the last value written to DTRRX and do not change RXfull.

For the full behavior of the Debug Communications Channel, see Chapter H4 *The Debug Communication Channel and Instruction Transfer Register*.

This field resets to an architecturally UNKNOWN value.

Accessing the DBGDTRRXext

Arm deprecates reads and writes of DBGDTRRXext through the System register interface when the OS Lock is unlocked, DBGOSLSR.OSLK == 0.

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TDCC == '1' then
        AArch32.TakeHypTrapException(0x05);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
        AArch32.TakeMonitorTrapException();
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGDTRRXext;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
        AArch32.TakeMonitorTrapException();
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGDTRRXext;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SDCR.TDCC == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return DBGDTRRXext;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TDCC == '1' then
        AArch32.TakeHypTrapException(0x05);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
        AArch32.TakeMonitorTrapException();
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        DBGDTRRXext = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
        AArch32.TakeMonitorTrapException();
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        DBGDTRRXext = R[t];
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SDCR.TDCC == '1' then
        AArch32.TakeMonitorTrapException();
    else
        DBGDTRRXext = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.17 DBGDTRRXint, Debug Data Transfer Register, Receive

The DBGDTRRXint characteristics are:

Purpose

Transfers data from an external debugger to the PE. For example, it is used by a debugger transferring commands and data to a debug target. See Arch64-DBGDTR_EL0 for additional architectural mappings. It is a component of the Debug Communications Channel.

Configurations

AArch32 System register DBGDTRRXint[31:0] is architecturally mapped to AArch64 System register DBGDTRRX_EL0[31:0].

AArch32 System register DBGDTRRXint[31:0] is architecturally mapped to External register DBGDTRRX_EL0[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DBGDTRRXint are UNKNOWN.

Attributes

DBGDTRRXint is a 32-bit register.

Field descriptions

The DBGDTRRXint bit assignments are:

31 … 0
Update DTRRX

Bits [31:0]

Update DTRRX.

Reads of this register:

- If RXfull is set to 1, return the last value written to DTRRX.
- If RXfull is set to 0, return an UNKNOWN value.

After the read, RXfull is cleared to 0.

For the full behavior of the Debug Communications Channel, see Chapter H4 *The Debug Communication Channel and Instruction Transfer Register*.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing the DBGDTRRXint

Data can be stored to memory from this register using STC.

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0101	0b000

```
if Halted() then
    return DBGDTRRXint;
elsif PSTATE.EL == EL0 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        if !ELUsingAArch32(EL1) && MDSCR_EL1.TDCC == '1' then
            if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
                AArch64.AArch32SystemAccessTrap(EL2, 0x05);
            else
                AArch64.AArch32SystemAccessTrap(EL1, 0x05);
        elsif ELUsingAArch32(EL1) && DBGDSCRext.UDCCdis == '1' then
            if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
                AArch64.AArch32SystemAccessTrap(EL2, 0x05);
            elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
                AArch32.TakeHypTrapException(0x00);
            else
                UNDEFINED;
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x05);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TDCC == '1' then
            AArch32.TakeHypTrapException(0x05);
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && (HCR_EL2.TGE == '1' || MDCR_EL2.<TDE,TDA> != '00') then
            AArch64.AArch32SystemAccessTrap(EL2, 0x05);
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && (HCR.TGE == '1' || HDCR.<TDE,TDA> != '00') then
            AArch32.TakeHypTrapException(0x05);
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
            AArch32.TakeMonitorTrapException();
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        else
            return DBGDTRRXint;
    elsif PSTATE.EL == EL1 then
        if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x05);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TDCC == '1' then
            AArch32.TakeHypTrapException(0x05);
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x05);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
            AArch32.TakeHypTrapException(0x05);
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
            AArch32.TakeMonitorTrapException();
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        else
            return DBGDTRRXint;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
            AArch32.TakeMonitorTrapException();
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        else
            return DBGDTRRXint;
    elsif PSTATE.EL == EL3 then
        if PSTATE.M != M32_Monitor && SDCR.TDCC == '1' then
            AArch32.TakeMonitorTrapException();
        else
            return DBGDTRRXint;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.18 DBGDTRTXext, Debug OS Lock Data Transfer Register, Transmit

The DBGDTRTXext characteristics are:

Purpose

Used for save/restore of DBGDTRTXint. It is a component of the Debug Communication Channel.

Configurations

AArch32 System register DBGDTRTXext[31:0] is architecturally mapped to AArch64 System register OSDTRTX_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DBGDTRTXext are UNKNOWN.

Attributes

DBGDTRTXext is a 32-bit register.

Field descriptions

The DBGDTRTXext bit assignments are:

31	0
Return DTRTX without side-effect	

Bits [31:0]

Return DTRTX without side-effect.

Reads of this register return the value in DTRTX and do not change TXfull.

Writes of this register update the value in DTRTX and do not change TXfull.

For the full behavior of the Debug Communications Channel, see Chapter H4 *The Debug Communication Channel and Instruction Transfer Register*.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing the DBGDTRTXext

Arm deprecates reads and writes of DBGDTRTXext through the System register interface when the OS Lock is unlocked, DBGOSLSR.OSLK == 0.

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0011	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TDCC == '1' then
        AArch32.TakeHypTrapException(0x05);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
        AArch32.TakeMonitorTrapException();
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGDTRTXext;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
        AArch32.TakeMonitorTrapException();
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGDTRTXext;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SDCR.TDCC == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return DBGDTRTXext;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0011	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TDCC == '1' then
        AArch32.TakeHypTrapException(0x05);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
        AArch32.TakeMonitorTrapException();
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        DBGDTRTXext = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
        AArch32.TakeMonitorTrapException();
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        DBGDTRTXext = R[t];
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SDCR.TDCC == '1' then
        AArch32.TakeMonitorTrapException();
    else
        DBGDTRTXext = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.19 DBGDTRTXint, Debug Data Transfer Register, Transmit

The DBGDTRTXint characteristics are:

Purpose

Transfers data from the PE to an external debugger. For example, it is used by a debug target to transfer data to the debugger. See DBGDTR_EL0 for additional architectural mappings. It is a component of the Debug Communication Channel.

Configurations

AArch32 System register DBGDTRTXint[31:0] is architecturally mapped to AArch64 System register DBGDTRTX_EL0[31:0].

AArch32 System register DBGDTRTXint[31:0] is architecturally mapped to External register DBGDTRTX_EL0[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DBGDTRTXint are UNKNOWN.

Attributes

DBGDTRTXint is a 32-bit register.

Field descriptions

The DBGDTRTXint bit assignments are:

31 … 0
Return DTRTX

Bits [31:0]

Return DTRTX.

Writes to this register:

- If TXfull is set to 1, set DTRTX to UNKNOWN.
- If TXfull is set to 0, update the value in DTRTX.

After the write, TXfull is set to 1.

For the full behavior of the Debug Communications Channel, see Chapter H4 *The Debug Communication Channel and Instruction Transfer Register*.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing the DBGDTRTXint

Data can be loaded from memory into this register using LDC (immediate) and LDC (literal).

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0101	0b000

```
if Halted() then
    DBGDTRTXint = R[t];
elsif PSTATE.EL == EL0 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        if !ELUsingAArch32(EL1) && MDSCR_EL1.TDCC == '1' then
            if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
                AArch64.AArch32SystemAccessTrap(EL2, 0x05);
            else
                AArch64.AArch32SystemAccessTrap(EL1, 0x05);
        elsif ELUsingAArch32(EL1) && DBGDSCRext.UDCCdis == '1' then
            if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
                AArch64.AArch32SystemAccessTrap(EL2, 0x05);
            elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
                AArch32.TakeHypTrapException(0x00);
            else
                UNDEFINED;
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x05);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TDCC == '1' then
            AArch32.TakeHypTrapException(0x05);
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && (HCR_EL2.TGE == '1' || MDCR_EL2.<TDE,TDA> != '00') then
            AArch64.AArch32SystemAccessTrap(EL2, 0x05);
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && (HCR.TGE == '1' || HDCR.<TDE,TDA> != '00') then
            AArch32.TakeHypTrapException(0x05);
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
            AArch32.TakeMonitorTrapException();
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        else
            DBGDTRTXint = R[t];
    elsif PSTATE.EL == EL1 then
        if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x05);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TDCC == '1' then
            AArch32.TakeHypTrapException(0x05);
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x05);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
            AArch32.TakeHypTrapException(0x05);
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
            AArch32.TakeMonitorTrapException();
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        else
            DBGDTRTXint = R[t];
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
            AArch32.TakeMonitorTrapException();
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        else
            DBGDTRTXint = R[t];
    elsif PSTATE.EL == EL3 then
        if PSTATE.M != M32_Monitor && SDCR.TDCC == '1' then
            AArch32.TakeMonitorTrapException();
        else
            DBGDTRTXint = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.20 DBGOSDLR, Debug OS Double Lock Register

The DBGOSDLR characteristics are:

Purpose

Locks out the external debug interface.

Configurations

AArch32 System register DBGOSDLR[31:0] is architecturally mapped to AArch64 System register OSDLR_EL1[31:0].

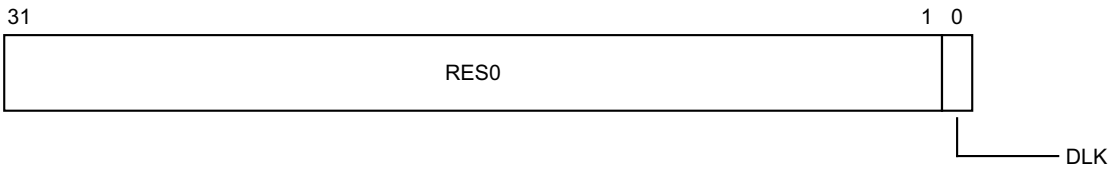
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DBGOSDLR are UNKNOWN.

Attributes

DBGOSDLR is a 32-bit register.

Field descriptions

The DBGOSDLR bit assignments are:



Bits [31:1]

Reserved, RES0.

DLK, bit [0]

When ARMv8.0-DoubleLock is implemented:

OS Double Lock control bit.

`0b0` OS Double Lock unlocked.

`0b1` OS Double Lock locked, if DBGPRCR.CORENPDRQ (Core no powerdown request) bit is set to 0 and the PE is in Non-debug state.

On a Warm reset, this field resets to `0`.

Otherwise:

Reserved, RAZ/WI.

Accessing the DBGOSDLR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0001	0b0011	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDOSA> != '00' &&
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
(IsFeatureImplemented("ARMv8.0-DoubleLock") || boolean IMPLEMENTATION_DEFINED "Trapped by
MDCR_EL2.TDOSA") then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDOSA> != '00' &&
(IsFeatureImplemented("ARMv8.0-DoubleLock") || boolean IMPLEMENTATION_DEFINED "Trapped by HDCR.TDOSA")
then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' &&
(IsFeatureImplemented("ARMv8.0-DoubleLock") || boolean IMPLEMENTATION_DEFINED "Trapped by
MDCR_EL3.TDOSA") then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGOSDLR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' &&
(IsFeatureImplemented("ARMv8.0-DoubleLock") || boolean IMPLEMENTATION_DEFINED "Trapped by
MDCR_EL3.TDOSA") then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGOSDLR;
elsif PSTATE.EL == EL3 then
    return DBGOSDLR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0001	0b0011	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDOSA> != '00' &&
(IsFeatureImplemented("ARMv8.0-DoubleLock") || boolean IMPLEMENTATION_DEFINED "Trapped by
MDCR_EL2.TDOSA") then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDOSA> != '00' &&
(IsFeatureImplemented("ARMv8.0-DoubleLock") || boolean IMPLEMENTATION_DEFINED "Trapped by HDCR.TDOSA")
then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' &&
(IsFeatureImplemented("ARMv8.0-DoubleLock") || boolean IMPLEMENTATION_DEFINED "Trapped by
MDCR_EL3.TDOSA") then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        DBGOSDLR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' &&
(IsFeatureImplemented("ARMv8.0-DoubleLock") || boolean IMPLEMENTATION_DEFINED "Trapped by
MDCR_EL3.TDOSA") then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        DBGOSDLR = R[t];
elsif PSTATE.EL == EL3 then
    DBGOSDLR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.21 DBGOSECCR, Debug OS Lock Exception Catch Control Register

The DBGOSECCR characteristics are:

Purpose

Provides a mechanism for an operating system to access the contents of EDECCR that are otherwise invisible to software, so it can save/restore the contents of EDECCR over powerdown on behalf of the external debugger.

Configurations

AArch32 System register DBGOSECCR[31:0] is architecturally mapped to AArch64 System register OSECCR_EL1[31:0].

AArch32 System register DBGOSECCR[31:0] is architecturally mapped to External register EDECCR[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DBGOSECCR are UNKNOWN.

If DBGOSLSR.OSLK == 0 then DBGOSECCR returns an UNKNOWN value on reads and ignores writes.

Attributes

DBGOSECCR is a 32-bit register.

Field descriptions

The DBGOSECCR bit assignments are:

When DBGOSLSR.OSLK == 1:

31	0
EDECCR	

EDECCR, bits [31:0]

Used for save/restore to EDECCR over powerdown.

Reads or writes to this field are indirect accesses to EDECCR.

Accessing the DBGOSECCR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0110	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        return DBGOSECCR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGOSECCR;
elsif PSTATE.EL == EL3 then
    return DBGOSECCR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0110	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        DBGOSECCR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        DBGOSECCR = R[t];
elsif PSTATE.EL == EL3 then
    DBGOSECCR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.22 DBGOSLAR, Debug OS Lock Access Register

The DBGOSLAR characteristics are:

Purpose

Provides a lock for the debug registers. The OS Lock also disables some debug exceptions and debug events.

Configurations

AArch32 System register DBGOSLAR[31:0] is architecturally mapped to AArch64 System register OSLAR_EL1[31:0].

AArch32 System register DBGOSLAR[31:0] is architecturally mapped to External register OSLAR_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DBGOSLAR are UNKNOWN.

Attributes

DBGOSLAR is a 32-bit register.

Field descriptions

The DBGOSLAR bit assignments are:

31 … 0
OSLA

OSLA, bits [31:0]

OS Lock Access. Writing the value `0xC5ACCE55` to the DBGOSLAR sets the OS lock to 1. Writing any other value sets the OS lock to 0.

Use DBGOSLSR.OSLK to check the current status of the lock.

Accessing the DBGOSLAR

Accesses to this register use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0001	0b0000	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDOSA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDOSA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        DBGOSLAR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        DBGOSLAR = R[t];
elsif PSTATE.EL == EL3 then
    DBGOSLAR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.23 DBGOSLSR, Debug OS Lock Status Register

The DBGOSLSR characteristics are:

Purpose

Provides status information for the OS Lock.

Configurations

AArch32 System register DBGOSLSR[31:0] is architecturally mapped to AArch64 System register OSLSR_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DBGOSLSR are UNKNOWN.

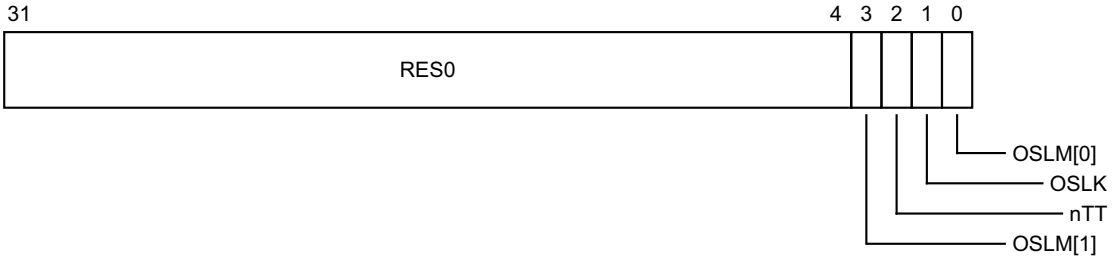
The OS Lock status is also visible in the external debug interface through EDPRSR.

Attributes

DBGOSLSR is a 32-bit register.

Field descriptions

The DBGOSLSR bit assignments are:



Bits [31:4]

Reserved, RES0.

OSLM[1], bit [3]

This field is bit[1] of OSLM[1:0].

OS lock model implemented. Identifies the form of OS save and restore mechanism implemented.

`0b00` OS Lock not implemented.

`0b10` OS Lock implemented.

All other values are reserved. In an Armv8 implementation the value `0b00` is not permitted.

The OSLM field is split as follows:

- OSLM[1] is DBGOSLSR[3].
- OSLM[0] is DBGOSLSR[0].

nTT, bit [2]

Not 32-bit access. This bit is always RAZ. It indicates that a 32-bit access is needed to write the key to the OS Lock Access Register.

OSLK, bit [1]

OS Lock Status. The possible values are:

`0b0` OS Lock unlocked.

`0b1` OS Lock locked.

The OS Lock is locked and unlocked by writing to the OS Lock Access Register.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following resets apply:

- On a Cold reset, this field resets to 1.
- On a Warm reset, the value of this field is unchanged.

OSLM[0], bit [0]

This field is bit[0] of OSLM[1:0].

See OSLM[1] for the field description.

Accessing the DBGOSLSR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0001	0b0001	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDOSA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDOSA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGOSLSR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGOSLSR;
elsif PSTATE.EL == EL3 then
    return DBGOSLSR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.24 DBGPRCR, Debug Power Control Register

The DBGPRCR characteristics are:

Purpose

Controls behavior of the PE on powerdown request.

Configurations

AArch32 System register DBGPRCR[31:0] is architecturally mapped to AArch64 System register DBGPRCR_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DBGPRCR are UNKNOWN.

Bit [0] of this register is mapped to EDPRCR.CORENPDRQ, bit [0] of the external view of this register.

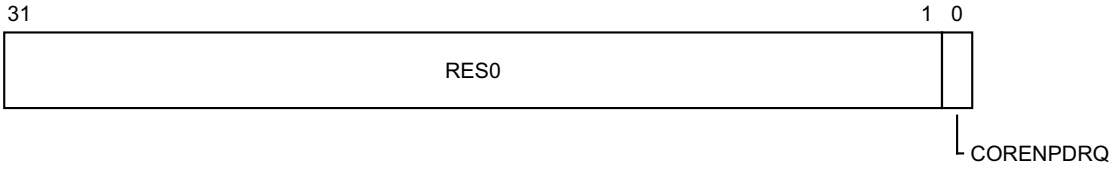
The other bits in these registers are not mapped to each other.

Attributes

DBGPRCR is a 32-bit register.

Field descriptions

The DBGPRCR bit assignments are:



Bits [31:1]

Reserved, RES0.

CORENPDRQ, bit [0]

When ARMv8.3-DoPD is implemented:

Core no powerdown request. Requests emulation of powerdown.

This request is typically passed to an external power controller. This means that whether a request causes power up is dependent on the IMPLEMENTATION DEFINED nature of the system. The power controller must not allow the Core power domain to switch off while this bit is 1.

`0b0` If the system responds to a powerdown request, it powers down Core power domain.

`0b1` If the system responds to a powerdown request, it does not powerdown the Core power domain, but instead emulates a powerdown of that domain.

In an implementation that includes the recommended external debug interface, this bit drives the DBGNOPWRDWN signal.

It is IMPLEMENTATION DEFINED whether this bit is reset to the Cold reset value on exit from an IMPLEMENTATION DEFINED software-visible retention state. For more information about retention states see *Core power domain power states* on page H6-7030.

Note

Writes to this bit are not prohibited by the IMPLEMENTATION DEFINED authentication interface. This means that a debugger can request emulation of powerdown regardless of whether invasive debug is permitted.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following resets apply:

- On a Cold reset, this field is set to 1 if the powerup request is implemented and the powerup request has been asserted, and is set to 0 otherwise.
- On a Warm reset, the value of this field is unchanged.

Otherwise:

Core no powerdown request. Requests emulation of powerdown.

This request is typically passed to an external power controller. This means that whether a request causes power up is dependent on the IMPLEMENTATION DEFINED nature of the system. The power controller must not allow the Core power domain to switch off while this bit is 1.

`0b0` If the system responds to a powerdown request, it powers down Core power domain.

`0b1` If the system responds to a powerdown request, it does not powerdown the Core power domain, but instead emulates a powerdown of that domain.

In an implementation that includes the recommended external debug interface, this bit drives the DBGNOPWRDWN signal.

It is IMPLEMENTATION DEFINED whether this bit is reset to the value of EDPRCR.COREPURQ on exit from an IMPLEMENTATION DEFINED software-visible retention state. For more information about retention states see *Core power domain power states* on page H6-7030.

Note

Writes to this bit are not prohibited by the IMPLEMENTATION DEFINED authentication interface. This means that a debugger can request emulation of powerdown regardless of whether invasive debug is permitted.

The following resets apply:

- On a Cold reset, this field resets to the value in EDPRCR.COREPURQ.
- On a Warm reset, the value of this field is unchanged.

Accessing the DBGPRCR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0001	0b0100	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDOSA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDOSA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGPRCR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGPRCR;
elsif PSTATE.EL == EL3 then
    return DBGPRCR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0001	0b0100	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDOSA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDOSA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        DBGPRCR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        DBGPRCR = R[t];
elsif PSTATE.EL == EL3 then
    DBGPRCR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.25 DBGVCR, Debug Vector Catch Register

The DBGVCR characteristics are:

Purpose

Controls Vector Catch debug events.

Configurations

AArch32 System register DBGVCR[31:0] is architecturally mapped to AArch64 System register DBGVCR32_EL2[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DBGVCR are UNKNOWN.

This register is required in all implementations.

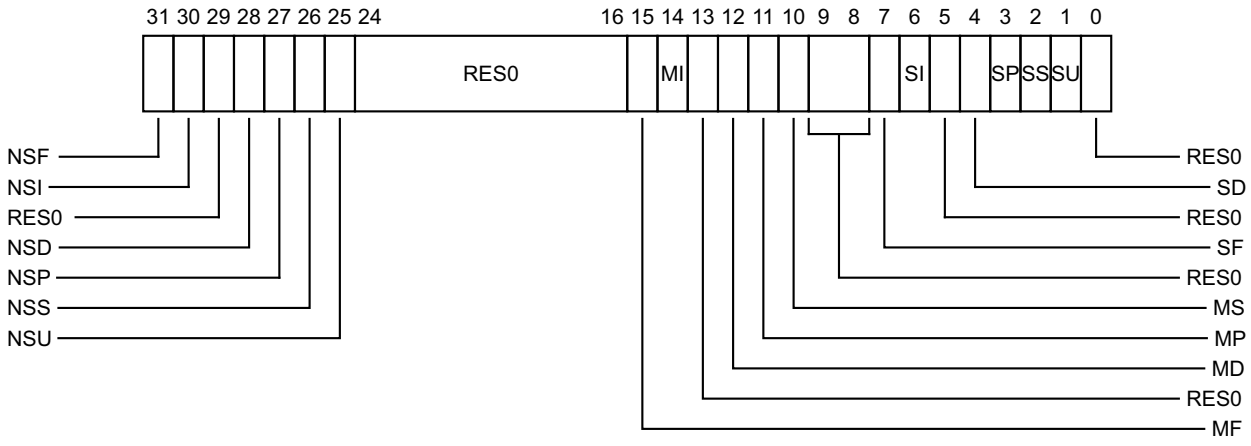
Attributes

DBGVCR is a 32-bit register.

Field descriptions

The DBGVCR bit assignments are:

When EL3 is implemented and EL3 is using AArch32:



NSF, bit [31]

FIQ vector catch enable in Non-secure state.

The exception vector offset is `0x1C`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

NSI, bit [30]

IRQ vector catch enable in Non-secure state.

The exception vector offset is `0x18`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [29]

Reserved, RES0.

NSD, bit [28]

Data Abort vector catch enable in Non-secure state.

The exception vector offset is `0x10`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

On a Warm reset, this field resets to an architecturally UNKNOWN value.

NSP, bit [27]

Prefetch Abort vector catch enable in Non-secure state.

The exception vector offset is `0x0C`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

NSS, bit [26]

Supervisor Call (SVC) vector catch enable in Non-secure state.

The exception vector offset is `0x08`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

NSU, bit [25]

Undefined Instruction vector catch enable in Non-secure state.

The exception vector offset is `0x04`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [24:16]

Reserved, RES0.

MF, bit [15]

FIQ vector catch enable in Monitor mode.

The exception vector offset is `0x1C`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

MI, bit [14]

IRQ vector catch enable in Monitor mode.

The exception vector offset is `0x18`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [13]

Reserved, RES0.

MD, bit [12]

Data Abort vector catch enable in Monitor mode.

The exception vector offset is `0x10`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

MP, bit [11]

Prefetch Abort vector catch enable in Monitor mode.

The exception vector offset is `0x0C`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

MS, bit [10]

Secure Monitor Call (SMC) vector catch enable in Monitor mode.

The exception vector offset is `0x08`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [9:8]

Reserved, RES0.

SF, bit [7]

FIQ vector catch enable in Secure state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The exception vector offset is `0x1C`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

SI, bit [6]

IRQ vector catch enable in Secure state.

The exception vector offset is `0x18`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

SD, bit [4]

Data Abort vector catch enable in Secure state.

The exception vector offset is `0x10`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

SP, bit [3]

Prefetch Abort vector catch enable in Secure state.

The exception vector offset is `0x0C`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

SS, bit [2]

Supervisor Call (SVC) vector catch enable in Secure state.

The exception vector offset is `0x08`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

SU, bit [1]

Undefined Instruction vector catch enable in Secure state.

The exception vector offset is `0x04`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [0]

Reserved, RES0.

When EL3 is implemented and EL3 is using AArch64:

NSF, bit [31]

FIQ vector catch enable in Non-secure state.

The exception vector offset is `0x1C`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

On a Warm reset, this field resets to an architecturally UNKNOWN value.

NSI, bit [30]

IRQ vector catch enable in Non-secure state.

The exception vector offset is `0x18`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [29]

Reserved, RES0.

NSD, bit [28]

Data Abort vector catch enable in Non-secure state.

The exception vector offset is `0x10`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

NSP, bit [27]

Prefetch Abort vector catch enable in Non-secure state.

The exception vector offset is `0x0C`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

NSS, bit [26]

Supervisor Call (SVC) vector catch enable in Non-secure state.

The exception vector offset is `0x08`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

NSU, bit [25]

Undefined Instruction vector catch enable in Non-secure state.

The exception vector offset is `0x04`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [24:8]

Reserved, RES0.

SF, bit [7]

FIQ vector catch enable in Secure state.

The exception vector offset is `0x1C`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

SI, bit [6]

IRQ vector catch enable in Secure state.

The exception vector offset is `0x18`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

SD, bit [4]

Data Abort vector catch enable in Secure state.

The exception vector offset is `0x10`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

SP, bit [3]

Prefetch Abort vector catch enable in Secure state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The exception vector offset is `0x0C`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

SS, bit [2]

Supervisor Call (SVC) vector catch enable in Secure state.

The exception vector offset is `0x08`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

SU, bit [1]

Undefined Instruction vector catch enable in Secure state.

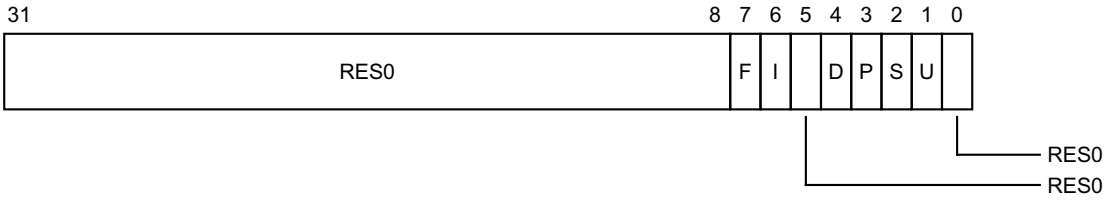
The exception vector offset is `0x04`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [0]

Reserved, RES0.

When EL3 is not implemented:



Bits [31:8]

Reserved, RES0.

F, bit [7]

FIQ vector catch enable.

The exception vector offset is `0x1C`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [6]

IRQ vector catch enable.

The exception vector offset is `0x18`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

D, bit [4]

Data Abort vector catch enable.

The exception vector offset is `0x10`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

P, bit [3]

Prefetch Abort vector catch enable.

The exception vector offset `0x0C`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

S, bit [2]

Supervisor Call (SVC) vector catch enable.

The exception vector offset is `0x08`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

U, bit [1]

Undefined Instruction vector catch enable.

The exception vector offset is `0x04`.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [0]

Reserved, RES0.

Accessing the DBGVCR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0111	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGVCR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGVCR;
elsif PSTATE.EL == EL3 then
    return DBGVCR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0111	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        DBGVCR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        DBGVCR = R[t];
elsif PSTATE.EL == EL3 then
    DBGVCR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.26 DBGWCR<n>, Debug Watchpoint Control Registers, n = 0 - 15

The DBGWCR<n> characteristics are:

Purpose

Holds control information for a watchpoint. Forms watchpoint n together with value register DBGWVR<n>.

Configurations

AArch32 System register DBGWCR<n>[31:0] is architecturally mapped to AArch64 System register DBGWCR<n>_EL1[31:0].

AArch32 System register DBGWCR<n>[31:0] is architecturally mapped to External register DBGWCR<n>_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DBGWCR<n> are UNKNOWN.

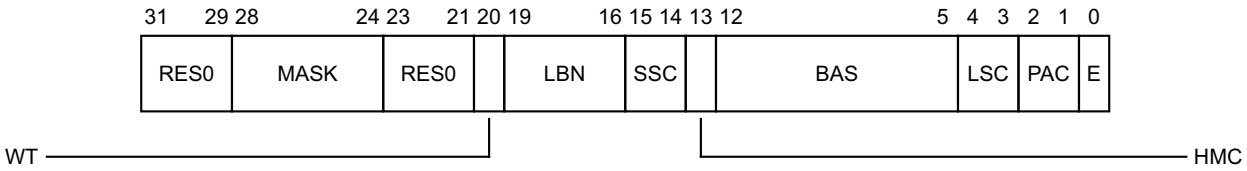
If watchpoint n is not implemented then accesses to this register are UNDEFINED.

Attributes

DBGWCR<n> is a 32-bit register.

Field descriptions

The DBGWCR<n> bit assignments are:



When the E field is zero, all the other fields in the register are ignored.

Bits [31:29]

Reserved, RES0.

MASK, bits [28:24]

Address mask. Only objects up to 2GB can be watched using a single mask.

`0b00000` No mask.

`0b00001` Reserved.

`0b00010` Reserved.

If programmed with a reserved value, a watchpoint must behave as if either:

- MASK has been programmed with a defined value, which might be 0 (no mask), other than for a direct read of DBGWCRn_EL1.
- The watchpoint is disabled.

Software must not rely on this property because the behavior of reserved values might change in a future revision of the architecture.

Other values mask the corresponding number of address bits, from `0b00011` masking 3 address bits (`0x00000007` mask for address) to `0b11111` masking 31 address bits (`0x7FFFFFFF` mask for address).

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bits [23:21]

Reserved, RES0.

WT, bit [20]

Watchpoint type. Possible values are:

`0b0` Unlinked data address match.

`0b1` Linked data address match.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

LBN, bits [19:16]

Linked breakpoint number. For Linked data address watchpoints, this specifies the index of the Context-matching breakpoint linked to.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

SSC, bits [15:14]

Security state control. Determines the Security states under which a Watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the HMC and PAC fields.

For more information, see *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page G2-5840, and *Reserved DBGBCR<n>.{SSC, HMC, PMC} values* on page G2-5851.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

HMC, bit [13]

Higher mode control. Determines the debug perspective for deciding when a Watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the SSC and PAC fields.

For more information on the operation of the SSC, HMC, and PAC fields, see *Execution conditions for which a watchpoint generates Watchpoint exceptions* on page G2-5857.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

BAS, bits [12:5]

Byte address select. Each bit of this field selects whether a byte from within the word or double-word addressed by DBGWVR<n> is being watched.

BAS	Description
`0bxxxxxxx1`	Match byte at DBGWVR<n>
`0bxxxxxx1x`	Match byte at DBGWVR<n>+1
`0bxxxxx1xx`	Match byte at DBGWVR<n>+2
`0bxxxx1xxx`	Match byte at DBGWVR<n>+3

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In cases where DBGWVR<n> addresses a double-word:

BAS	Description, if DBGWVR<n>[2] == 0
`0bxxx1xxxx`	Match byte at DBGWVR<n>+4
`0bxx1xxxxx`	Match byte at DBGWVR<n>+5
`0bx1xxxxxx`	Match byte at DBGWVR<n>+6
`0b1xxxxxxx`	Match byte at DBGWVR<n>+7

If DBGWVR<n>[2] == 1, only BAS[3:0] are used and BAS[7:4] are ignored. Arm deprecates setting DBGWVR<n>[2] == 1.

The valid values for BAS are non-zero binary numbers all of whose set bits are contiguous. All other values are reserved and must not be used by software. See *Reserved DBGWCR<n>.BAS values* on page G2-5865

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

LSC, bits [4:3]

Load/store control. This field enables watchpoint matching on the type of access being made. Possible values of this field are:

`0b01` Match instructions that load from a watchpointed address.

`0b10` Match instructions that store to a watchpointed address.

`0b11` Match instructions that load from or store to a watchpointed address.

All other values are reserved, but must behave as if the watchpoint is disabled. Software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

PAC, bits [2:1]

Privilege of access control. Determines the Exception level or levels at which a Watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the SSC and HMC fields.

For more information on the operation of the SSC, HMC, and PAC fields, see *Execution conditions for which a watchpoint generates Watchpoint exceptions* on page G2-5857.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

E, bit [0]

Enable watchpoint n. Possible values are:

`0b0` Watchpoint disabled.

`0b1` Watchpoint enabled.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the DBGWCR<n>

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	n[3:0]	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    elsif ELUsingAArch32(EL1) && DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        return DBGWCR[UInt(CRm<3:0>)];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    elsif ELUsingAArch32(EL1) && DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        return DBGWCR[UInt(CRm<3:0>)];
elsif PSTATE.EL == EL3 then
    if ELUsingAArch32(EL1) && DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        return DBGWCR[UInt(CRm<3:0>)];
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	n[3:0]	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    elsif ELUsingAArch32(EL1) && DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGWCR[UInt(CRm<3:0>)] = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    elsif ELUsingAArch32(EL1) && DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGWCR[UInt(CRm<3:0>)] = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    if ELUsingAArch32(EL1) && DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGWCR[UInt(CRm<3:0>)] = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.27 DBGWFAR, Debug Watchpoint Fault Address Register

The DBGWFAR characteristics are:

Purpose

Previously returned information about the address of the instruction that accessed a watchpointed address. Is now deprecated and RES0.

Configurations

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DBGWFAR are UNKNOWN.

Attributes

DBGWFAR is a 32-bit register.

Field descriptions

The DBGWFAR bit assignments are:

31 … 0
RES0

Bits [31:0]

Reserved, RES0.

Accessing the DBGWFAR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0110	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGWFAR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGWFAR;
elsif PSTATE.EL == EL3 then
    return DBGWFAR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0110	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        DBGWFAR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        DBGWFAR = R[t];
elsif PSTATE.EL == EL3 then
    DBGWFAR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.28 DBGWVR<n>, Debug Watchpoint Value Registers, n = 0 - 15

The DBGWVR<n> characteristics are:

Purpose

Holds a data address value for use in watchpoint matching. Forms watchpoint n together with control register DBGWCR<n>.

Configurations

AArch32 System register DBGWVR<n>[31:0] is architecturally mapped to AArch64 System register DBGWVR<n>_EL1[31:0].

AArch32 System register DBGWVR<n>[31:0] is architecturally mapped to External register DBGWVR<n>_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DBGWVR<n> are UNKNOWN.

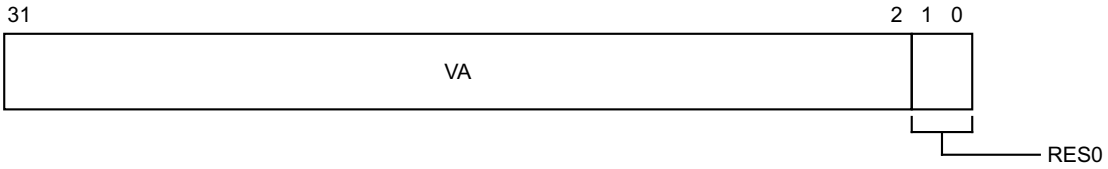
If watchpoint n is not implemented then accesses to this register are UNDEFINED.

Attributes

DBGWVR<n> is a 32-bit register.

Field descriptions

The DBGWVR<n> bit assignments are:



VA, bits [31:2]

Bits[31:2] of the address value for comparison.

Arm deprecates setting DBGWVR<n>[2] == 1.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, the value of this field is unchanged.

Bits [1:0]

Reserved, RES0.

Accessing the DBGWVR<n>

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	n[3:0]	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    elsif ELUsingAArch32(EL1) && DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        return DBGWVR[UInt(CRm<3:0>)];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    elsif ELUsingAArch32(EL1) && DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        return DBGWVR[UInt(CRm<3:0>)];
elsif PSTATE.EL == EL3 then
    if ELUsingAArch32(EL1) && DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        return DBGWVR[UInt(CRm<3:0>)];
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	n[3:0]	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    elsif ELUsingAArch32(EL1) && DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGWVR[UInt(CRm<3:0>)] = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    elsif ELUsingAArch32(EL1) && DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGWVR[UInt(CRm<3:0>)] = R[t];
elsif PSTATE.EL == EL3 then
    if ELUsingAArch32(EL1) && DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGWVR[UInt(CRm<3:0>)] = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.29 DLR, Debug Link Register

The DLR characteristics are:

Purpose

In Debug state, holds the address to restart from.

Configurations

AArch32 System register DLR[31:0] is architecturally mapped to AArch64 System register DLR_EL0[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DLR are UNKNOWN.

Attributes

DLR is a 32-bit register.

Field descriptions

The DLR bit assignments are:

31 … 0
Restart address

Bits [31:0]

Restart address.

Accessing the DLR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b011	0b0100	0b0101	0b001

```
if !Halted() then
    UNDEFINED;
else
    return DLR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b011	0b0100	0b0101	0b001

```
if !Halted() then
    UNDEFINED;
else
    DLR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.30 DSPSR, Debug Saved Program Status Register

The DSPSR characteristics are:

Purpose

Holds the saved process state for Debug state. On entering Debug state, PSTATE information is written to this register. On exiting Debug state, values are copied from this register to PSTATE.

Configurations

AArch32 System register DSPSR[31:0] is architecturally mapped to AArch64 System register DSPSR_EL0[31:0].

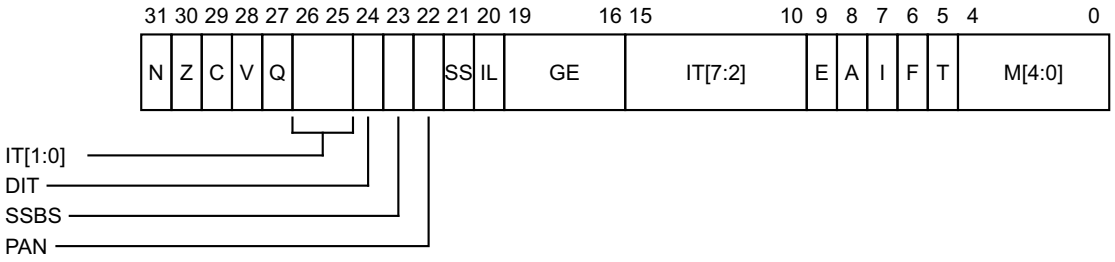
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to DSPSR are UNKNOWN.

Attributes

DSPSR is a 32-bit register.

Field descriptions

The DSPSR bit assignments are:



N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on entering Debug state, and copied to PSTATE.N on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on entering Debug state, and copied to PSTATE.Z on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on entering Debug state, and copied to PSTATE.C on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on entering Debug state, and copied to PSTATE.V on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on entering Debug state, and copied to PSTATE.Q on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

IT[1:0], bits [26:25]

If-Then. Set to the value of PSTATE.IT[1:0] on entering Debug state, and copied to PSTATE.IT[1:0] on exiting Debug state.

On exiting Debug state DSPSR.IT must contain a value that is valid for the instruction being returned to.

This field resets to an architecturally UNKNOWN value.

DIT, bit [24]

When ARMv8.4-DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on entering Debug state, and copied to PSTATE.DIT on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SSBS, bit [23]

When ARMv8.0-SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on entering Debug state, and copied to PSTATE.SSBS on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When ARMv8.1-PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on entering Debug state, and copied to PSTATE.PAN on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SS, bit [21]

Software Step. Set to the value of PSTATE.SS on entering Debug state, and conditionally copied to PSTATE.SS on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on entering Debug state, and copied to PSTATE.IL on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on entering Debug state, and copied to PSTATE.GE on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

IT[7:2], bits [15:10]

If-Then. Set to the value of PSTATE.IT[7:2] on entering Debug state, and copied to PSTATE.IT[7:2] on exiting Debug state.

DSPSR.IT must contain a value that is valid for the instruction being returned to.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

E, bit [9]

Endianness. Set to the value of PSTATE.E on entering Debug state, and copied to PSTATE.E on exiting Debug state.

If the implementation does not support big-endian operation, DSPSR.E is RES0. If the implementation does not support little-endian operation, DSPSR.E is RES1. On exiting Debug state, if the implementation does not support big-endian operation at the Exception level being returned to, DSPSR.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, DSPSR.E is RES1.

This field resets to an architecturally UNKNOWN value.

A, bit [8]

SError interrupt mask. Set to the value of PSTATE.A on entering Debug state, and copied to PSTATE.A on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on entering Debug state, and copied to PSTATE.I on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on entering Debug state, and copied to PSTATE.F on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on entering Debug state, and copied to PSTATE.T on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on entering Debug state, and copied to PSTATE.M[4:0] on exiting Debug state.

`0b10000` User.

`0b10001` FIQ.

`0b10010` IRQ.

`0b10011` Supervisor.

`0b10110` Monitor.

`0b10111` Abort.

`0b11010` Hyp.

`0b11011` Undefined.

`0b11111` System.

Other values are reserved. If DSPSR.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, exiting Debug state is an illegal return event, as described in *Illegal return events from AArch32 state* on page G1-5726.

This field resets to an architecturally UNKNOWN value.

Accessing the DSPSR

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b011	0b0100	0b0101	0b000

```
if !Halted() then
    UNDEFINED;
else
    return DSPSR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b011	0b0100	0b0101	0b000

```
if !Halted() then
    UNDEFINED;
else
    DSPSR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.31 HDCR, Hyp Debug Control Register

The HDCR characteristics are:

Purpose

Controls the trapping to Hyp mode of Non-secure accesses, at EL1 or lower, to functions provided by the debug and trace architectures and the Performance Monitors Extension.

Configurations

AArch32 System register HDCR[31:0] is architecturally mapped to AArch64 System register MDCR_EL2[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to HDCR are UNKNOWN.

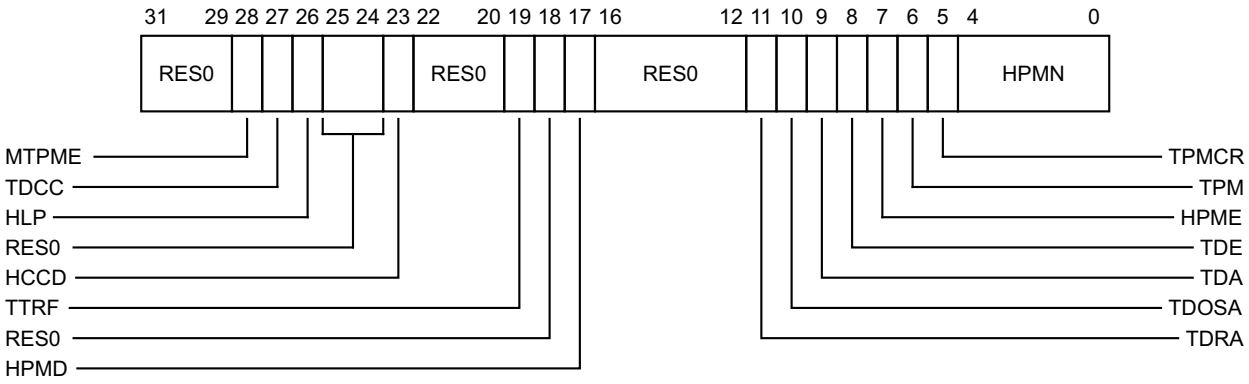
If EL2 is not implemented, this register is RES0 from EL3, and other than for a direct read of the register, the PE behaves as if HDCR.HPMN == PMCR.N.

Attributes

HDCR is a 32-bit register.

Field descriptions

The HDCR bit assignments are:



Bits [31:29]

Reserved, RES0.

MTPME, bit [28]

When ARMv8.6-MTPMU is implemented and EL3 is not implemented:

Multi-threaded PMU Enable. Enables use of the PMEVTYPER<n>.MT bits.

`0b0` ARMv8.6-MTPMU is disabled. The Effective value of PMEVTYPER<n>.MT is zero.

`0b1` PMEVTYPER<n>.MT bits not affected by this bit.

If ARMv8.6-MTPMU is disabled for any other PE in the system that has the same level 1 Affinity as the PE, it is IMPLEMENTATION DEFINED whether the PE behaves as if this bit is 0.

On a Cold reset, in a system where the PE resets into EL2 or EL3, this field resets to 1.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TDCC, bit [27]

When ARMv8.6-FGT is implemented:

Trap DCC. Traps use of the Debug Comms Channel at EL1 and EL0 to EL2.

`0b0` This control does not cause any register accesses to be trapped.

`0b1` If EL2 is implemented and enabled in the current Security state, accesses to the DCC registers at EL1 and EL0 generate a Hyp Trap exception, unless the access also generates a higher priority exception.

Traps on the DCC data transfer registers are ignored when the PE is in Debug state.

The DCC registers trapped by this control are:

- DBGDTRRXext, DBGDTRTXext, DBGDSCRint, DBGDCCINT, and, when the PE is in Non-debug state, DBGDTRRXint and DBGDTRTXint.

The traps are reported with EC syndrome value:

- `0x05` for trapped `MRC` and `MCR` accesses with `coproc` == `0b1110`.
- `0x06` for trapped `LDC` to DBGDTRTXint and `STC` from DBGDTRRXint.

When the PE is in Debug state, HDCR.TDCC does not trap any accesses to:

- DBGDTRRXint and DBGDTRTXint.

in a system where the PE resets into EL2 or EL3, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HLP, bit [26]

When ARMv8.5-PMU is implemented:

Hypervisor Long event counter enable. Determines when unsigned overflow is recorded by a counter overflow bit.

`0b0` Event counter overflow on increment that causes unsigned overflow of PMEVCNTR<n>[31:0].

`0b1` Event counter overflow on increment that causes unsigned overflow of PMEVCNTR<n>[63:0].

If the highest implemented Exception level is using AArch32, it is IMPLEMENTATION DEFINED whether this bit is read/write or RAZ/WI.

If HDCR.HPMN is less than PMCR.N, this bit affects the operation of event counters in the range [HDCR.HPMN..(PMCR.N-1)]. Otherwise this bit has no effect on the operation of the event counters.

Note

The effect of HDCR.HPMN on the operation of this bit always applies if EL2 is implemented, at all Exception levels including EL2 and EL3, and regardless of whether EL2 is enabled in the current Security state.

For more information see the description of the HDCR.HPMN field.

Note

PMEVCNTR<n>[63:32] cannot be accessed directly in AArch32 state.

On a Warm reset, in a system where the PE resets into EL2 or EL3, this field resets to `0`.

Otherwise:

Reserved, RES0.

Bits [25:24]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

HCCD, bit [23]

When ARMv8.5-PMU is implemented:

Hypervisor Cycle Counter Disable. Prohibits PMCCNTR from counting at EL2.

`0b0` Cycle counting by PMCCNTR is not affected by this bit.

`0b1` Cycle counting by PMCCNTR is prohibited at EL2.

This bit does not affect the CPU_CYCLES event or any other event that counts cycles.

On a Warm reset, in a system where the PE resets into EL2 or EL3, this field resets to `0`.

Otherwise:

Reserved, RES0.

Bits [22:20]

Reserved, RES0.

TTRF, bit [19]

When ARMv8.4-Trace is implemented:

Traps use of the Trace Filter Control registers at EL1 to EL2.

`0b0` Accesses to TRFCR at EL1 are not affected by this control bit.

`0b1` Accesses to TRFCR at EL1 generate a Hyp Trap exception.

On a Warm reset, in a system where the PE resets into EL2 or EL3, this field resets to `0`.

Otherwise:

Reserved, RES0.

Bit [18]

Reserved, RES0.

HPMD, bit [17]

When ARMv8.1-PMU is implemented:

Guest Performance Monitors Disable. This control prohibits event counting at EL2.

`0b0` Event counting allowed in Hyp mode.

`0b1` Event counting prohibited in Hyp mode.

If ARMv8.2-Debug is not implemented, event counting is prohibited unless enabled by the IMPLEMENTATION DEFINED authentication interface ExternalSecureNoninvasiveDebugEnabled().

This control applies only to:

- The event counters in the range [0..(HDCR.HPMN-1)].
- If PMCR.DP is set to 1, PMCCNTR.

The other event counters are unaffected. When PMCR.DP is set to 0, PMCCNTR is unaffected.

On a Warm reset, in a system where the PE resets into EL2 or EL3, this field resets to `0`.

Otherwise:

Reserved, RES0.

Bits [16:12]

Reserved, RES0.

TDRA, bit [11]

Trap Debug ROM Address register access. Traps Non-secure EL0 and EL1 System register accesses to the Debug ROM registers to Hyp mode.

`0b0` This control does not cause any instructions to be trapped.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` Non-secure EL0 and EL1 System register accesses to the DBGDRAR or DBGDSAR are trapped to Hyp mode, unless it is trapped by DBGDSCRext.UDCCdis.

If HCR.TGE or HDCR.TDE is 1, behavior is as if this bit is 1 other than for the purpose of a direct read.

On a Warm reset, in a system where the PE resets into EL2 or EL3, this field resets to `0`.

TDOSA, bit [10]

When ARMv8.0-DoubleLock is implemented:

Trap debug OS-related register access. Traps Non-secure EL1 System register accesses to the powerdown debug registers to Hyp mode.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Non-secure EL1 System register accesses to the powerdown debug registers are trapped to Hyp mode.

The registers for which accesses are trapped are as follows:

- DBGOSLSR, DBGOSLAR, DBGOSDLR, and DBGPRCR.
- Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this bit.

Note

These registers are not accessible at EL0.

If HCR.TGE or HDCR.TDE is 1, behavior is as if this bit is 1 other than for the purpose of a direct read.

On a Warm reset, in a system where the PE resets into EL2 or EL3, this field resets to `0`.

Otherwise:

Trap debug OS-related register access. Traps Non-secure EL1 System register accesses to the powerdown debug registers to Hyp mode.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Non-secure EL1 System register accesses to the powerdown debug registers are trapped to Hyp mode.

The registers for which accesses are trapped are as follows:

- DBGOSLSR, DBGOSLAR, and DBGPRCR.
- Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this bit.

It is IMPLEMENTATION DEFINED whether accesses to DBGOSDLR are trapped.

Note

These registers are not accessible at EL0.

If HCR.TGE or HDCR.TDE is 1, behavior is as if this bit is 1 other than for the purpose of a direct read.

On a Warm reset, in a system where the PE resets into EL2 or EL3, this field resets to `0`.

TDA, bit [9]

Trap debug access. Traps Non-secure EL0 and EL1 System register accesses to those debug System registers in the (coproc==`0b1110`) encoding space that are not trapped by either of the following:

- HDCR.TDRA.
- HDCR.TDOSA.

`0b0` This control does not cause any instructions to be trapped.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` Non-secure EL0 or EL1 System register accesses to the debug registers, other than the registers trapped by HDCR.TDRA and HDCR.TDOSA, are trapped to Hyp mode, unless it is trapped by DBGDSCRext.UDCCdis.

Traps of AArch32 accesses to DBGDTRRXint and DBGDTRTXint are ignored in Debug state.

If HCR.TGE or HDCR.TDE is 1, behavior is as if this bit is 1 other than for the purpose of a direct read.

On a Warm reset, in a system where the PE resets into EL2 or EL3, this field resets to `0`.

TDE, bit [8]

Trap Debug exceptions. The possible values of this bit are:

`0b0` This control has no effect on the routing of debug exceptions, and has no effect on Non-secure accesses to debug registers.

`0b1` Debug exceptions generated at EL1 or EL0 are routed to EL2 when enabled in the current Security state. The HDCR.{TDRA, TDOSA, TDA} fields are treated as being 1 for all purposes other than returning the result of a direct read of the register.

When HCR.TGE == 1, the PE behaves as if the value of this field is 1 for all purposes other than returning the value of a direct read of the register.

On a Warm reset, in a system where the PE resets into EL2 or EL3, this field resets to `0`.

HPME, bit [7]

When PMUv3 is implemented:

[HDCR.HPMN..(N-1)] event counters enable.

`0b0` Event counters in the range [HDCR.HPMN..(PMCR.N-1)] are disabled.

`0b1` Event counters in the range [HDCR.HPMN..(PMCR.N-1)] are enabled by PMCNTENSET.

If HDCR.HPMN is less than PMCR.N, the event counters in the range [HDCR.HPMN..(PMCR.N-1)], are enabled and disabled by this bit. Otherwise this bit has no effect on the operation of the event counters.

Note

The effect of HDCR.HPMN on the operation of this bit applies regardless of whether EL2 is enabled in the current Security state.

For more information see the description of the HPMN field.

On a Warm reset, in a system where the PE resets into EL2 or EL3, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TPM, bit [6]

When PMUv3 is implemented:

Trap Performance Monitors accesses. Traps Non-secure EL0 and EL1 accesses to all Performance Monitors registers to Hyp mode.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Non-secure EL0 and EL1 accesses to all Performance Monitors registers are trapped to Hyp mode.

Note

EL2 does not provide traps on Performance Monitor register accesses through the optional memory-mapped external debug interface.

On a Warm reset, in a system where the PE resets into EL2 or EL3, this field resets to `0`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

Reserved, RES0.

TPMCR, bit [5]

When PMUv3 is implemented:

Trap PMCR accesses. Traps Non-secure EL0 and EL1 accesses to the PMCR to Hyp mode.

`0b0` This control does not cause any instructions to be trapped.

`0b1` Non-secure EL0 and EL1 accesses to the PMCR are trapped to Hyp mode, unless it is trapped by PMUSERENR.EN.

Note

EL2 does not provide traps on Performance Monitor register accesses through the optional memory-mapped external debug interface.

On a Warm reset, in a system where the PE resets into EL2 or EL3, this field resets to `0`.

Otherwise:

Reserved, RES0.

HPMN, bits [4:0]

When PMUv3 is implemented:

Defines the number of event counters that are accessible from Non-secure EL1 modes, and from Non-secure EL0 modes if unprivileged access is enabled.

If HPMN is less than PMCR.N, HPMN divides the event counters into two ranges, [0..(HPMN-1)] and [HPMN..(PMCR.N-1)].

For an event counter in the range [0..(HPMN-1)]:

- The counter is accessible from EL1 and EL2, and from EL0 if unprivileged access to the counters is enabled.
- If ARMv8.5-PMU is implemented, PMCR.LP determines whether the counter overflows at PMEVCNTR<n>[31:0] or PMEVCNTR<n>[63:0].
- PMCR.E enables the operation of counters in this range.

Note

If HPMN is equal to PMCR.N, this applies to all event counters.

If HPMN is less than PMCR.N, for an event counter in the range [HPMN..(PMCR.N-1)]:

- The counter is accessible only from EL2 and from Secure state.
- If ARMv8.5-PMU is implemented, HDCR.HLP determines whether the counter overflows at PMEVCNTR<n>[31:0] or PMEVCNTR<n>[63:0].
- HDCR.HPME enables the operation of counters in this range.

If this field is set to 0, or to a value larger than PMCR.N, then the following CONSTRAINED UNPREDICTABLE behaviors apply:

- The value returned by a direct read of HDCR.HPMN is UNKNOWN.
- Either:
  - An UNKNOWN number of counters are reserved for EL2 use. That is, the PE behaves as if HDCR.HPMN is set to an UNKNOWN non-zero value less than or equal to PMCR.N.
  - All counters are reserved for EL2 use, meaning no counters are accessible from Non-secure EL1 and Non-secure EL0.

On a Warm reset, in a system where the PE resets into EL2 or EL3, this field resets to the value in PMCR.N.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

Reserved, RES0.

Accessing the HDCR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return HDCR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        return HDCR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        HDCR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        HDCR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.32 HTRFCR, Hyp Trace Filter Control Register

The HTRFCR characteristics are:

Purpose

Provides EL2 controls for Trace.

Configurations

AArch32 System register HTRFCR[31:0] is architecturally mapped to AArch64 System register TRFCR_EL2[31:0].

This register is present only when AArch32 is supported at any Exception level and ARMv8.4-Trace is implemented. Otherwise, direct accesses to HTRFCR are UNDEFINED.

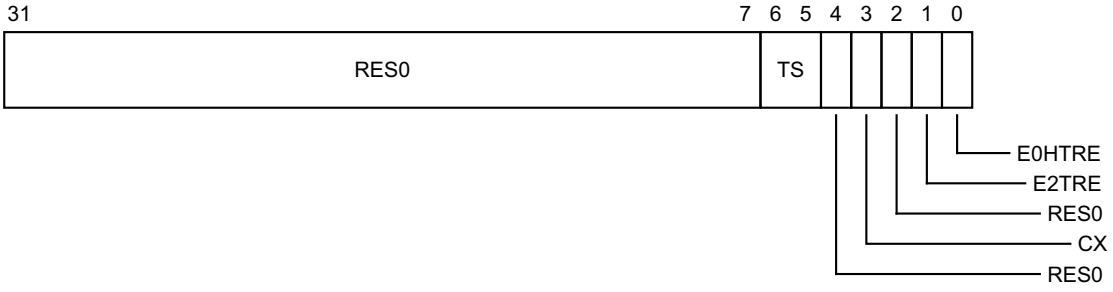
If EL2 is not implemented, this register is RES0 from Monitor mode when SCR.NS == 1.

Attributes

HTRFCR is a 32-bit register.

Field descriptions

The HTRFCR bit assignments are:



Bits [31:7]

Reserved, RES0.

TS, bits [6:5]

Timestamp Control. Controls which timebase is used for trace timestamps.

`0b00` The timestamp is controlled by TRFCR.TS.

`0b01` Virtual timestamp. The traced timestamp is the physical counter value minus the value of CNTVOFF.

`0b11` Physical timestamp. The traced timestamp is the physical counter value.

When SelfHostedTraceEnabled() == FALSE, this field is ignored.

On a Warm reset, in a system where the PE resets into EL2 or EL3, this field resets to `0`.

Bit [4]

Reserved, RES0.

CX, bit [3]

VMID Trace Enable.

`0b0` VMID tracing is not allowed.

`0b1` VMID tracing is allowed.

When SelfHostedTraceEnabled() == FALSE, this field is ignored.

On a Warm reset, in a system where the PE resets into EL2 or EL3, this field resets to `0`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bit [2]

Reserved, RES0.

E2TRE, bit [1]

EL2 Trace Enable.

`0b0` Tracing is prohibited at EL2.

`0b1` Tracing is allowed at EL2.

When SelfHostedTraceEnabled() == FALSE, this field is ignored.

On a Warm reset, in a system where the PE resets into EL2 or EL3, this field resets to `0`.

E0HTRE, bit [0]

EL0 Trace Enable.

`0b0` Tracing is prohibited at EL0 when HCR.TGE == 1.

`0b1` Tracing is allowed at EL0 when HCR.TGE == 1.

This field is ignored if any of the following are true:

- The PE is in Secure state.
- SelfHostedTraceEnabled() == FALSE.
- HCR.TGE == 0.

On a Warm reset, in a system where the PE resets into EL2 or EL3, this field resets to `0`.

Accessing the HTRFCR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TTRF == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TTRF == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return HTRFCR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        return HTRFCR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TTRF == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TTRF == '1' then
        AArch32.TakeMonitorTrapException();
    else
        HTRFCR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        HTRFCR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.33 PMMIR, Performance Monitors Machine Identification Register

The PMMIR characteristics are:

Purpose

Describes Performance Monitors parameters specific to the implementation to software.

Configurations

This register is present only when AArch32 is supported at any Exception level and ARMv8.4-PMU is implemented. Otherwise, direct accesses to PMMIR are UNDEFINED.

Attributes

PMMIR is a 32-bit register.

Field descriptions

The PMMIR bit assignments are:

31 … 8	7 … 0
RES0	SLOTS

Bits [31:8]

Reserved, RES0.

SLOTS, bits [7:0]

Operation width. The largest value by which the STALL_SLOT event might increment by in a single cycle. If the STALL_SLOT event is not implemented, this field might read as zero.

Accessing the PMMIR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1110	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMMIR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        return PMMIR;
elsif PSTATE.EL == EL3 then
    return PMMIR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.34 SDCR, Secure Debug Control Register

The SDCR characteristics are:

Purpose

Provides EL3 configuration options for self-hosted debug, trace, and the Performance Monitors Extension.

Configurations

AArch32 System register SDCR[31:0] can be mapped to AArch64 System register MDCR_EL3[31:0], but this is not architecturally mandated.

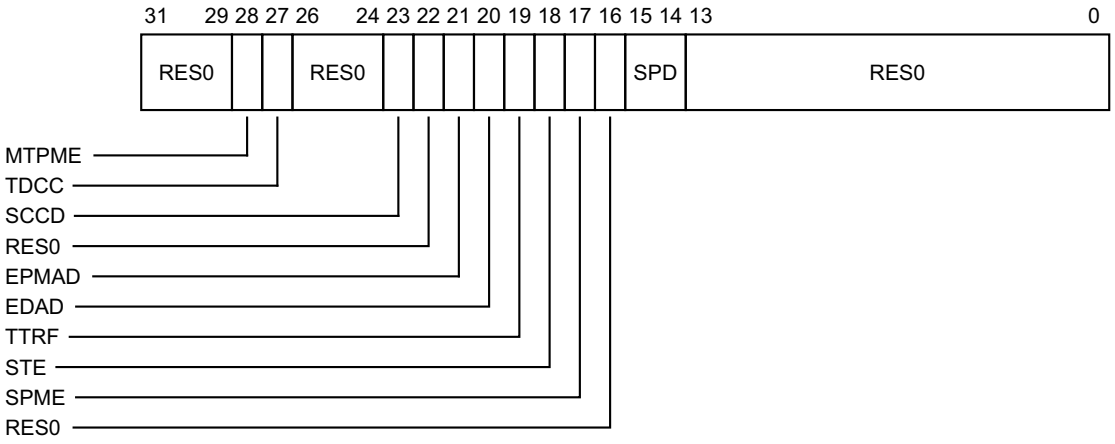
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to SDCR are UNKNOWN.

Attributes

SDCR is a 32-bit register.

Field descriptions

The SDCR bit assignments are:



Bits [31:29]

Reserved, RES0.

MTPME, bit [28]

When ARMv8.6-MTPMU is implemented:

Multi-threaded PMU Enable. Enables use of the PMEVTYPER<n>.MT bits.

`0b0` ARMv8.6-MTPMU is disabled. The Effective value of PMEVTYPER<n>.MT is zero.

`0b1` PMEVTYPER<n>.MT bits not affected by this bit.

If ARMv8.6-MTPMU is disabled for any other PE in the system that has the same level 1 Affinity as the PE, it is IMPLEMENTATION DEFINED whether the PE behaves as if this bit is 0.

On a Cold reset, in a system where the PE resets into EL3, this field resets to `1`.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TDCC, bit [27]

When ARMv8.6-FGT is implemented:

Trap DCC. Traps use of the Debug Comms Channel in modes other than Monitor mode to Monitor mode.

`0b0` This control does not cause any register accesses to be trapped.

`0b1` Accesses to the DCC registers in modes other than Monitor mode generate a Monitor Trap exception, unless the access also generates a higher priority exception.

Traps on the DCC data transfer registers are ignored when the PE is in Debug state.

The DCC registers trapped by this control are:

- DBGDTRRXext, DBGDTRTXext, DBGDSCRint, DBGDCCINT, and, when the PE is in Non-debug state, DBGDTRRXint and DBGDTRTXint.

When the PE is in Debug state, SDCR.TDCC does not trap any accesses to:

- DBGDTRRXint and DBGDTRTXint.

In a system where the PE resets into EL3, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [26:24]

Reserved, RES0.

SCCD, bit [23]

When ARMv8.5-PMU is implemented:

Secure Cycle Counter Disable. Prohibits PMCCNTR from counting in Secure state.

`0b0` Cycle counting by PMCCNTR is not affected by this bit.

`0b1` Cycle counting by PMCCNTR is prohibited in Secure state.

This bit does not affect the CPU_CYCLES event or any other event that counts cycles.

In a system where the PE resets into EL3, this field resets to `0`.

Otherwise:

Reserved, RES0.

Bit [22]

Reserved, RES0.

EPMAD, bit [21]

When ARMv8.4-Debug is implemented and PMUv3 is implemented:

External Performance Monitors Non-secure access disable. Controls Non-secure access to Performance Monitors registers by an external debugger.

`0b0` Non-secure access to the Performance Monitors registers from an external debugger is permitted.

`0b1` Non-secure access to the Performance Monitors registers from an external debugger is not permitted.

If the Performance Monitors Extension does not support external debug interface accesses this bit is RES0.

Otherwise, if EL3 is not implemented and the Effective value of SCR.NS is `0b0`, then the Effective value of this field is `0b1`.

In a system where the PE resets into EL3, this field resets to `0`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When PMUv3 is implemented:

External Performance Monitors access disable. Controls access to Performance Monitors registers by an external debugger.

`0b0` Access to Performance Monitors registers from an external debugger is permitted.

`0b1` Access to Performance Monitors registers from an external debugger is not permitted, unless overridden by the IMPLEMENTATION DEFINED authentication interface.

If the Performance Monitors Extension does not support external debug interface accesses this bit is RES0.

Otherwise, if EL3 is not implemented and the Effective value of SCR.NS is `0b0`, then the Effective value of this field is `0b1`.

In a system where the PE resets into EL3, this field resets to `0`.

Otherwise:

Reserved, RES0.

EDAD, bit [20]

When ARMv8.4-Debug is implemented:

External debug Non-secure access disable. Controls Non-secure access to breakpoint, watchpoint, and OSLAR_EL1 registers by an external debugger.

`0b0` Non-secure access to debug registers from an external debugger is permitted.

`0b1` Non-secure access to breakpoint registers, watchpoint registers, and OSLAR_EL1 from an external debugger is not permitted.

If EL3 is not implemented and the Effective value of SCR.NS is `0b0`, then the Effective value of this field is `0b1`.

In a system where the PE resets into EL3, this field resets to `0`.

When ARMv8.2-Debug is implemented:

External debug access disable. Controls access to breakpoint, watchpoint, and OSLAR_EL1 registers by an external debugger.

`0b0` Access to debug registers from an external debugger is permitted.

`0b1` Access to breakpoint registers, watchpoint registers and OSLAR_EL1 from an external debugger is not permitted, unless overridden by the IMPLEMENTATION DEFINED authentication interface.

If EL3 is not implemented and the Effective value of SCR.NS is `0b0`, then the Effective value of this field is `0b1`.

In a system where the PE resets into EL3, this field resets to `0`.

Otherwise:

External debug access disable. Controls access to breakpoint, watchpoint, and optionally OSLAR_EL1 registers by an external debugger.

`0b0` Access to debug registers from an external debugger is permitted.

`0b1` Access to breakpoint registers and watchpoint registers from an external debugger is not permitted, unless overridden by the IMPLEMENTATION DEFINED authentication interface.

It is IMPLEMENTATION DEFINED whether access to the OSLAR_EL1 register from an external debugger is permitted or not permitted.

If EL3 is not implemented and the Effective value of SCR.NS is `0b0`, then the Effective value of this field is `0b1`.

In a system where the PE resets into EL3, this field resets to `0`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TTRF, bit [19]

When ARMv8.4-Trace is implemented:

Trap Trace Filter controls. Controls whether accesses at EL2 and EL1 to the trace filter control registers are trapped to EL3.

`0b0` Accesses to HTRFCR and TRFCR registers are not affected by this control bit.

`0b1` When not in Monitor mode, accesses to HTRFCR and TRFCR registers generate a Monitor Trap exception, unless the access generates a higher priority exception.

In a system where the PE resets into EL3, this field resets to `0`.

Otherwise:

Reserved, RES0.

STE, bit [18]

When ARMv8.4-Trace is implemented:

Secure Trace Enable. This bit enables tracing in Secure state and controls the level of authentication required by an external debugger to enable external tracing.

`0b0` Trace is prohibited in Secure state unless overridden by the IMPLEMENTATION DEFINED authentication interface.

`0b1` Trace in Secure state is not affected by this bit.

This bit also controls the level of authentication required by an external debugger to enable external tracing. See *Register controls to enable self-hosted trace* on page G3-5880.

If EL3 is not implemented and the Effective value of SCR.NS is `0b0`, the PE behaves as if this bit is set to `0b1`.

In a system where the PE resets into EL3, this field resets to `0`.

Otherwise:

Reserved, RES0.

SPME, bit [17]

When ARMv8.2-Debug is implemented and PMUv3 is implemented:

Secure Performance Monitors enable. This allows event counting in Secure state.

`0b0` Event counting prohibited in Secure state.

`0b1` Event counting allowed in Secure state.

If EL3 is not implemented and the Effective value of SCR.NS is `0b0`, then the Effective value of this bit is `0b1`.

In a system where the PE resets into EL3, this field resets to `0`.

When PMUv3 is implemented:

Secure Performance Monitors enable. This allows event counting in Secure state.

`0b0` Event counting prohibited in Secure state, unless ExternalSecureNoninvasiveDebugEnabled() is TRUE.

`0b1` Event counting allowed in Secure state.

If EL3 is not implemented and the Effective value of SCR.NS is `0b0`, then the Effective value of this bit is `0b1`.

In a system where the PE resets into EL3, this field resets to `0`.

Otherwise:

Reserved, RES0.

Bit [16]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

SPD, bits [15:14]

AArch32 Secure self-hosted Privileged Debug. Enables or disables debug exceptions from EL3, other than Breakpoint Instruction exceptions

`0b00` Legacy mode. Debug exceptions from EL3 are enabled by the authentication interface.

`0b10` Secure privileged debug disabled. Debug exceptions from EL3 are disabled.

`0b11` Secure privileged debug enabled. Debug exceptions from EL3 are enabled.

Other values are reserved, and have the CONSTRAINED UNPREDICTABLE behavior that they must have the same behavior as `0b00`. Software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture.

This field has no effect on Breakpoint Instruction exceptions. These are always enabled.

This field is ignored in Non-secure state.

If debug exceptions from EL3 are enabled, then debug exceptions from Secure EL0 are also enabled.

Otherwise, debug exceptions from Secure EL0 are enabled only if the value of SDER.SUIDEN is `0b1`.

If EL3 is not implemented and the Effective value of SCR.NS is `0b0`, then the Effective value of this field is `0b11`.

In a system where the PE resets into EL3, this field resets to `0`.

Bits [13:0]

Reserved, RES0.

Accessing the SDCR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0011	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif !ELUsingAArch32(EL2) && SCR_EL3.<NS,EEL2> == '01' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    return SDCR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0011	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif !ELUsingAArch32(EL2) && SCR_EL3.<NS,EEL2> == '01' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' && CP15SDISABLE2 == HIGH then
        UNDEFINED;
    else
        SDCR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.35 SDER, Secure Debug Enable Register

The SDER characteristics are:

Purpose

Controls invasive and non-invasive debug in the Secure EL0 mode.

Configurations

AArch32 System register SDER[31:0] is architecturally mapped to AArch64 System register SDER32_EL3[31:0].

This register is present only when AArch32 is supported at any Exception level, or EL3 is implemented or the implemented Security state is Secure state. Otherwise, direct accesses to SDER are UNDEFINED.

This register is ignored by the PE when one or more of the following are true:

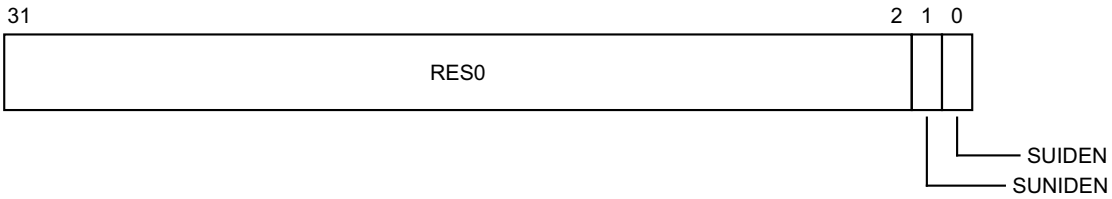
- The PE is in Non-secure state.
- EL1 is using AArch64.

Attributes

SDER is a 32-bit register.

Field descriptions

The SDER bit assignments are:



Bits [31:2]

Reserved, RES0.

SUNIDEN, bit [1]

Secure User Non-Invasive Debug Enable.

`0b0` This bit does not affect Performance Monitors event counting at Secure EL0

`0b1` If EL3 or EL1 is using AArch32, Performance Monitors event counting is allowed in Secure EL0.

On a Warm reset, this field resets to `0`.

SUIDEN, bit [0]

Secure User Invasive Debug Enable.

`0b0` This bit does not affect the generation of debug exceptions at Secure EL0.

`0b1` If EL3 or EL1 is using AArch32, debug exceptions from Secure EL0 are enabled.

On a Warm reset, this field resets to `0`.

Accessing the SDER

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif (!HaveEL(EL3) || !ELUsingAArch32(EL3)) && SCR_EL3.NS == '0' then
        return SDER;
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    return SDER;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif (!HaveEL(EL3) || !ELUsingAArch32(EL3)) && SCR_EL3.NS == '0' then
        SDER = R[t];
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' && CP15SDISABLE2 == HIGH then
        UNDEFINED;
    else
        SDER = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.3.36 TRFCR, Trace Filter Control Register

The TRFCR characteristics are:

Purpose

Provides EL1 controls for Trace.

Configurations

AArch32 System register TRFCR[31:0] is architecturally mapped to AArch64 System register TRFCR_EL1[31:0].

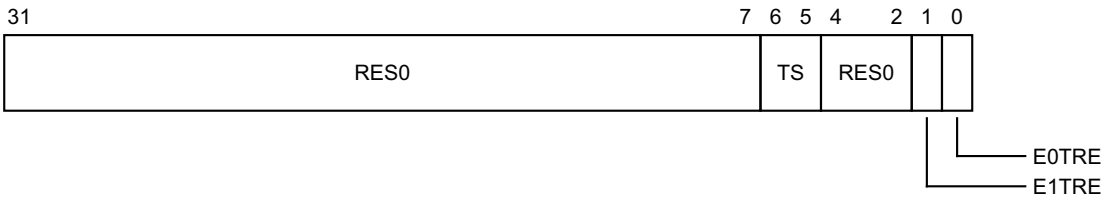
This register is present only when AArch32 is supported at any Exception level and ARMv8.4-Trace is implemented. Otherwise, direct accesses to TRFCR are UNDEFINED.

Attributes

TRFCR is a 32-bit register.

Field descriptions

The TRFCR bit assignments are:



Bits [31:7]

Reserved, RES0.

TS, bits [6:5]

Timestamp Control. Controls which timebase is used for trace timestamps.

`0b01` Virtual timestamp. The traced timestamp is the physical counter value, minus the value of CNTVOFF.

`0b10` **When ARMv8.6-ECV is implemented** Guest Physical timestamp. The traced timestamp is the physical counter value, minus the value of CNTPOFF_EL2.

`0b11` Physical timestamp. The traced timestamp is the physical counter value.

All other values are reserved.

This field is ignored if any of the following are true:

- `SelfHostedTraceEnabled()` == FALSE.
- HTRFCR.TS is not `0b00`.

If ARMv8.6-ECV is implemented, the physical counter uses a fixed physical offset of zero if any of the following are true:

- When EL2 is implemented and enabled in the current Security state and is using AArch32.
- CNTHCTL_EL2.ECV is 0.
- SCR_EL3.ECVEn is 0.
- HCR_EL2.{E2H, TGE} is {1, 1}.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [4:2]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

E1TRE, bit [1]

EL1 Trace Enable.

`0b0` Tracing is prohibited in PL1 modes.

`0b1` Tracing is allowed in PL1 modes.

This field is ignored if `SelfHostedTraceEnabled()` == FALSE.

On a Warm reset, this field resets to `0`.

E0TRE, bit [0]

EL0 Trace Enable.

`0b0` Tracing is prohibited at EL0.

`0b1` Tracing is allowed at EL0.

This field is ignored if any of the following are true:

- `SelfHostedTraceEnabled()` == FALSE.
- EL2 is implemented and enabled in the current security state and HCR.TGE == 1.

On a Warm reset, this field resets to `0`.

Accessing the TRFCR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif SCR.NS == '0' then
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TTRF == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TTRF == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TTRF == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SDCR.TTRF == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return TRFCR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TTRF == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TTRF == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return TRFCR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        return TRFCR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif SCR.NS == '0' then
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TTRF == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TTRF == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TTRF == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SDCR.TTRF == '1' then
        AArch32.TakeMonitorTrapException();
    else
        TRFCR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TTRF == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TTRF == '1' then
        AArch32.TakeMonitorTrapException();
    else
        TRFCR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        TRFCR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.4 Performance Monitors registers

This section lists the Performance Monitors registers in AArch32.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.4.1 PMCCFILTR, Performance Monitors Cycle Count Filter Register

The PMCCFILTR characteristics are:

Purpose

Determines the modes in which the Cycle Counter, PMCCNTR, increments.

Configurations

AArch32 System register PMCCFILTR[31:0] is architecturally mapped to AArch64 System register PMCCFILTR_EL0[31:0].

AArch32 System register PMCCFILTR[31:0] is architecturally mapped to External register PMCCFILTR_EL0[31:0].

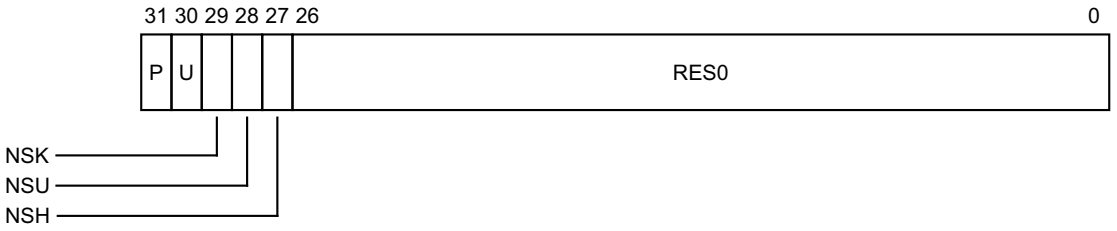
This register is present only when AArch32 is supported at any Exception level and PMUv3 is implemented. Otherwise, direct accesses to PMCCFILTR are UNDEFINED.

Attributes

PMCCFILTR is a 32-bit register.

Field descriptions

The PMCCFILTR bit assignments are:



P, bit [31]

Privileged filtering bit. Controls counting in EL1.

If EL3 is implemented, then counting in Non-secure EL1 is further controlled by the PMCCFILTR.NSK bit.

`0b0` Count cycles in EL1.

`0b1` Do not count cycles in EL1.

On a Warm reset, this field resets to `0`.

U, bit [30]

User filtering bit. Controls counting in EL0.

If EL3 is implemented, then counting in Non-secure EL0 is further controlled by the PMCCFILTR.NSU bit.

`0b0` Count cycles in EL0.

`0b1` Do not count cycles in EL0.

On a Warm reset, this field resets to `0`.

NSK, bit [29]

When EL3 is implemented:

Non-secure EL1 (kernel) modes filtering bit. Controls counting in Non-secure EL1.

If the value of this bit is equal to the value of PMCCFILTR.P, cycles in Non-secure EL1 are counted.

Otherwise, cycles in Non-secure EL1 are not counted.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

On a Warm reset, this field resets to `0`.

Otherwise:

Reserved, RES0.

NSU, bit [28]

When EL3 is implemented:

Non-secure EL0 (Unprivileged) filtering. Controls counting in Non-secure EL0.

If the value of this bit is equal to the value of PMCCFILTR.U, cycles in Non-secure EL0 are counted.

Otherwise, cycles in Non-secure EL0 are not counted.

On a Warm reset, this field resets to `0`.

Otherwise:

Reserved, RES0.

NSH, bit [27]

When EL2 is implemented:

EL2 (Hyp mode) filtering bit. Controls counting in EL2.

`0b0` Do not count cycles in EL2.

`0b1` Count cycles in EL2.

On a Warm reset, this field resets to `0`.

Otherwise:

Reserved, RES0.

Bits [26:0]

Reserved, RES0.

Accessing the PMCCFILTR

PMCCFILTR can also be accessed by using PMXEVTYPER with PMSELR.SEL set to `0b11111`.

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b1111	0b111

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && PMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMCCFILTR_EL0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMCCFILTR;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMCCFILTR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMCCFILTR;
elsif PSTATE.EL == EL3 then
    return PMCCFILTR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b1111	0b111

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && PMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMCCFILTR_EL0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMCCFILTR = R[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMCCFILTR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        else
            PMCCFILTR = R[t];
    elsif PSTATE.EL == EL3 then
        PMCCFILTR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.4.2 PMCCNTR, Performance Monitors Cycle Count Register

The PMCCNTR characteristics are:

Purpose

Holds the value of the processor Cycle Counter, CCNT, that counts processor clock cycles. See *Time as measured by the Performance Monitors cycle counter* on page D7-2702 for more information.

PMCCFILTR determines the modes and states in which the PMCCNTR can increment.

Configurations

AArch32 System register PMCCNTR[63:0] is architecturally mapped to AArch64 System register PMCCNTR_EL0[63:0].

AArch32 System register PMCCNTR[63:0] is architecturally mapped to External register PMCCNTR_EL0[63:0].

This register is present only when AArch32 is supported at any Exception level and PMUv3 is implemented. Otherwise, direct accesses to PMCCNTR are UNDEFINED.

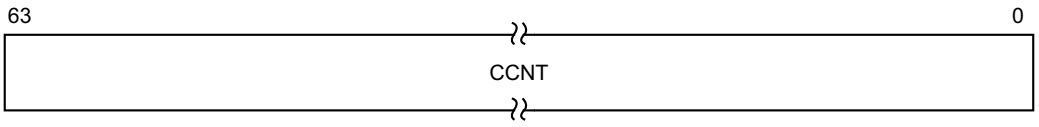
All counters are subject to any changes in clock frequency, including clock stopping caused by the WFI and WFE instructions. This means that it is CONSTRAINED UNPREDICTABLE whether or not PMCCNTR continues to increment when clocks are stopped by WFI and WFE instructions.

Attributes

PMCCNTR is a 64-bit register that can also be accessed as a 32-bit value. If it is accessed as a 32-bit register, accesses read and write bits [31:0] and do not modify bits [63:32].

Field descriptions

The PMCCNTR bit assignments are:



CCNT, bits [63:0]

Cycle count. Depending on the values of PMCR.{LC,D}, this field increments in one of the following ways:

- Every processor clock cycle.
- Every 64th processor clock cycle.

Writing 1 to PMCR.C sets this field to 0.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMCCNTR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1101	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.<CR,EN> == '00' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && PMUSERENR.<CR,EN> == '00' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMCCNTR_EL0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMCCNTR<31:0>;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMCCNTR<31:0>;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMCCNTR<31:0>;
elsif PSTATE.EL == EL3 then
    return PMCCNTR<31:0>;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1101	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && PMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMCCNTR_EL0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMCCNTR = ZeroExtend(R[t]);
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMCCNTR = ZeroExtend(R[t]);
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMCCNTR = ZeroExtend(R[t]);
elsif PSTATE.EL == EL3 then
    PMCCNTR = ZeroExtend(R[t]);
```

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1001	0b0000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.<CR,EN> == '00' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x04);
    elsif ELUsingAArch32(EL1) && PMUSERENR.<CR,EN> == '00' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMCCNTR_EL0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch32.TakeHypTrapException(0x04);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x04);
    else
        return PMCCNTR;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x04);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x04);
    else
        return PMCCNTR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x04);
    else
        return PMCCNTR;
elsif PSTATE.EL == EL3 then
    return PMCCNTR;
```

MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1001	0b0000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x04);
    elsif ELUsingAArch32(EL1) && PMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMCCNTR_EL0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x04);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x04);
    else
        PMCCNTR = R[t2]:R[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x04);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x04);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x04);
    else
        PMCCNTR = R[t2]:R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x04);
    else
        PMCCNTR = R[t2]:R[t];
elsif PSTATE.EL == EL3 then
    PMCCNTR = R[t2]:R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.4.3 PMCEID0, Performance Monitors Common Event Identification register 0

The PMCEID0 characteristics are:

Purpose

Defines which common architectural events and common microarchitectural events are implemented, or counted, using PMU events in the range `0x0000` to `0x001F`

When the value of a bit in the register is 1 the corresponding common event is implemented and counted.

Note

Arm recommends that, if a common event is never counted, the value of the corresponding register bit is 0.

For more information about the common events and the use of the PMCEIDn registers see *The PMU event number space and common events* on page D7-2721.

Configurations

AArch32 System register PMCEID0[31:0] is architecturally mapped to AArch64 System register PMCEID0_EL0[31:0].

AArch32 System register PMCEID0[31:0] is architecturally mapped to External register PMCEID0[31:0].

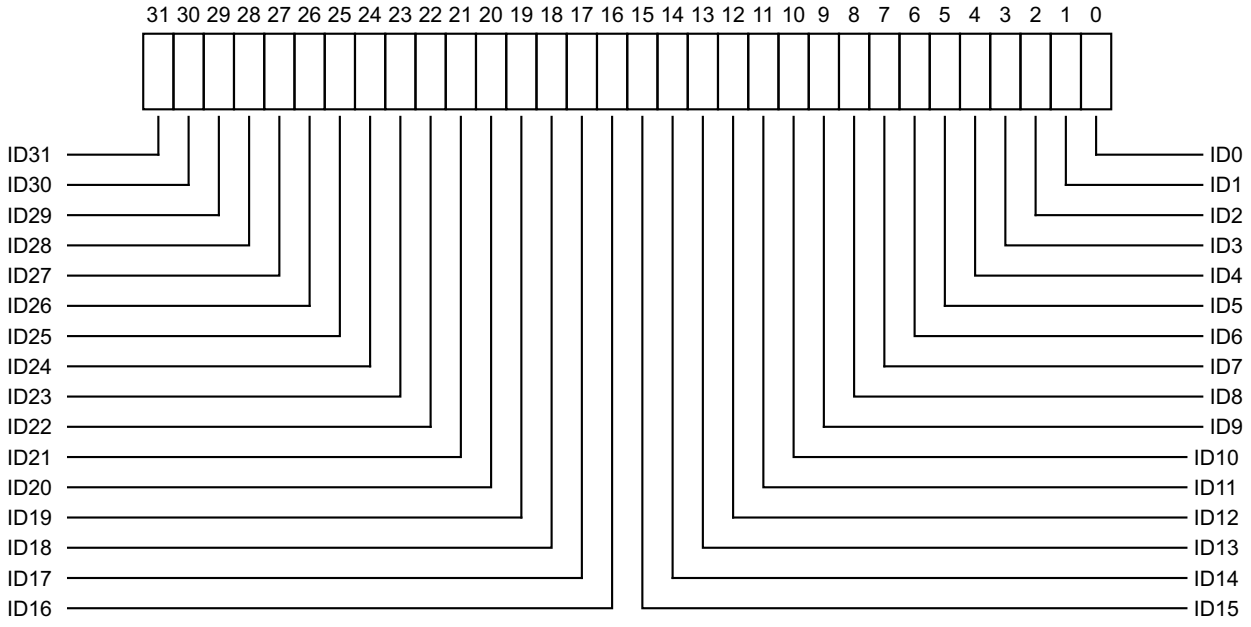
This register is present only when AArch32 is supported at any Exception level and PMUv3 is implemented. Otherwise, direct accesses to PMCEID0 are UNDEFINED.

Attributes

PMCEID0 is a 32-bit register.

Field descriptions

The PMCEID0 bit assignments are:



ID<n>, bit [n], for n = 0 to 31

ID[n] corresponds to common event n.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For each bit:

`0b0` The common event is not implemented, or not counted.

`0b1` The common event is implemented.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n> registers of that earlier version of the PMU architecture.

Accessing the PMCEID0

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b110

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && PMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMCEID0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMCEID0;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMCEID0;
elsif PSTATE.EL == EL3 then
    return PMCEID0;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.4.4 PMCEID1, Performance Monitors Common Event Identification register 1

The PMCEID1 characteristics are:

Purpose

Defines which common architectural events and common microarchitectural events are implemented, or counted, using PMU events in the range `0x0020` to `0x003F`.

When the value of a bit in the register is 1 the corresponding common event is implemented and counted.

Note

Arm recommends that, if a common event is never counted, the value of the corresponding register bit is 0.

For more information about the common events and the use of the PMCEIDn registers see *The PMU event number space and common events* on page D7-2721.

Configurations

AArch32 System register PMCEID1[31:0] is architecturally mapped to AArch64 System register PMCEID1_EL0[31:0].

AArch32 System register PMCEID1[31:0] is architecturally mapped to External register PMCEID1[31:0].

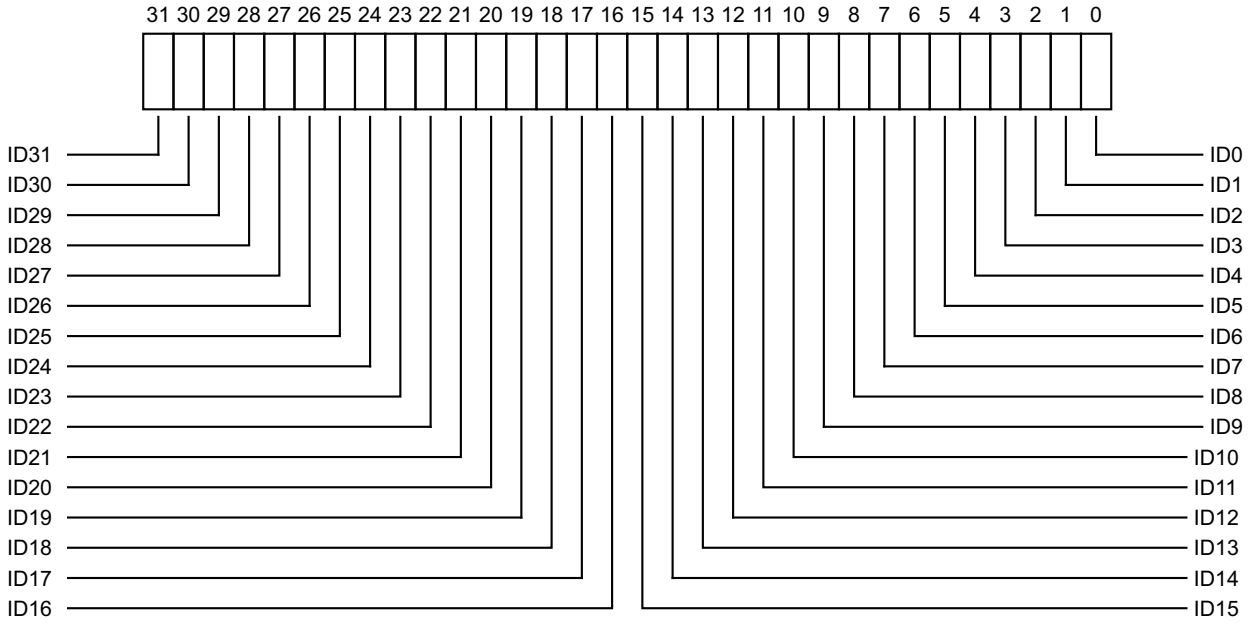
This register is present only when AArch32 is supported at any Exception level and PMUv3 is implemented. Otherwise, direct accesses to PMCEID1 are UNDEFINED.

Attributes

PMCEID1 is a 32-bit register.

Field descriptions

The PMCEID1 bit assignments are:



ID<n>, bit [n], for n = 0 to 31

ID[n] corresponds to common event (`0x0020` + n).

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For each bit:

`0b0` The common event is not implemented, or not counted.

`0b1` The common event is implemented.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n> registers of that earlier version of the PMU architecture.

Accessing the PMCEID1

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b111

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && PMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMCEID1;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMCEID1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMCEID1;
elsif PSTATE.EL == EL3 then
    return PMCEID1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.4.5 PMCEID2, Performance Monitors Common Event Identification register 2

The PMCEID2 characteristics are:

Purpose

Defines which common architectural events and common microarchitectural events are implemented, or counted, using PMU events in the range `0x4000` to `0x401F`.

When the value of a bit in the register is 1 the corresponding common event is implemented and counted.

Note

Arm recommends that, if a common event is never counted, the value of the corresponding register bit is 0.

For more information about the common events and the use of the PMCEIDn registers see *The PMU event number space and common events* on page D7-2721.

Configurations

AArch32 System register PMCEID2[31:0] is architecturally mapped to AArch64 System register PMCEID0_EL0[63:32].

AArch32 System register PMCEID2[31:0] is architecturally mapped to External register PMCEID2[63:32].

This register is present only when AArch32 is supported at any Exception level and ARMv8.1-PMU is implemented. Otherwise, direct accesses to PMCEID2 are UNDEFINED.

Attributes

PMCEID2 is a 32-bit register.

Field descriptions

The PMCEID2 bit assignments are:

31	0
IDhi<n>, bit[n], for n = 0 to 31	

IDhi<n>, bit [n], for n = 0 to 31

IDhi[n] corresponds to common event (`0x4000` + n).

For each bit:

`0b0` The common event is not implemented, or not counted.

`0b1` The common event is implemented.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n> registers of that earlier version of the PMU architecture.

Accessing the PMCEID2

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1110	0b100

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && PMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMCEID2;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMCEID2;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMCEID2;
elsif PSTATE.EL == EL3 then
    return PMCEID2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.4.6 PMCEID3, Performance Monitors Common Event Identification register 3

The PMCEID3 characteristics are:

Purpose

Defines which common architectural events and common microarchitectural events are implemented, or counted, using PMU events in the range `0x4020` to `0x403F`.

When the value of a bit in the register is 1 the corresponding common event is implemented and counted.

Note

Arm recommends that, if a common event is never counted, the value of the corresponding register bit is 0.

For more information about the common events and the use of the PMCEIDn registers see *The PMU event number space and common events* on page D7-2721.

Configurations

AArch32 System register PMCEID3[31:0] is architecturally mapped to AArch64 System register PMCEID1_EL0[63:32].

AArch32 System register PMCEID3[31:0] is architecturally mapped to External register PMCEID3[63:32].

This register is present only when AArch32 is supported at any Exception level and ARMv8.1-PMU is implemented. Otherwise, direct accesses to PMCEID3 are UNDEFINED.

Attributes

PMCEID3 is a 32-bit register.

Field descriptions

The PMCEID3 bit assignments are:

31	0
IDhi<n>, bit[n], for n = 0 to 31	

IDhi<n>, bit [n], for n = 0 to 31

IDhi[n] corresponds to common event (`0x4020` + n).

For each bit:

`0b0` The common event is not implemented, or not counted.

`0b1` The common event is implemented.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n> registers of that earlier version of the PMU architecture.

Accessing the PMCEID3

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1110	0b101

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && PMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMCEID3;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMCEID3;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMCEID3;
elsif PSTATE.EL == EL3 then
    return PMCEID3;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.4.7 PMCNTENCLR, Performance Monitors Count Enable Clear register

The PMCNTENCLR characteristics are:

Purpose

Disables the Cycle Count Register, PMCCNTR, and any implemented event counters PMEVCNTR<n>. Reading this register shows which counters are enabled.

PMCNTENCLR is used in conjunction with the PMCNTENSET register.

Configurations

AArch32 System register PMCNTENCLR[31:0] is architecturally mapped to AArch64 System register PMCNTENCLR_EL0[31:0].

AArch32 System register PMCNTENCLR[31:0] is architecturally mapped to External register PMCNTENCLR_EL0[31:0].

This register is present only when AArch32 is supported at any Exception level and PMUv3 is implemented. Otherwise, direct accesses to PMCNTENCLR are UNDEFINED.

Attributes

PMCNTENCLR is a 32-bit register.

Field descriptions

The PMCNTENCLR bit assignments are:

31	30 … 0
C	P<n>, bit [n]

C, bit [31]

PMCCNTR disable bit. Disables the cycle counter register. Possible values are:

`0b0` When read, means the cycle counter is disabled. When written, has no effect.

`0b1` When read, means the cycle counter is enabled. When written, disables the cycle counter.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

P<n>, bit [n], for n = 0 to 30

Event counter disable bit for PMEVCNTR<n>.

If N is less than 31, then bits [30:N] are RAZ/WI. When EL2 is implemented and enabled in the current Security state, in EL1 and EL0, N is the value in MDCR_EL2.HPMN if EL2 is using AArch64, or in HDCR.HPMN if EL2 is using AArch32. Otherwise, N is the value in PMCR.N.

`0b0` When read, means that PMEVCNTR<n> is disabled. When written, has no effect.

`0b1` When read, means that PMEVCNTR<n> is enabled. When written, disables PMEVCNTR<n>.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMCNTENCLR

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b010

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && PMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMCNTEN == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMCNTENCLR;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMCNTENCLR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMCNTENCLR;
elsif PSTATE.EL == EL3 then
    return PMCNTENCLR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b010

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && PMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMCNTEN == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMCNTENCLR = R[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMCNTENCLR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMCNTENCLR = R[t];
elsif PSTATE.EL == EL3 then
    PMCNTENCLR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.4.8 PMCNTENSET, Performance Monitors Count Enable Set register

The PMCNTENSET characteristics are:

Purpose

Enables the Cycle Count Register, PMCCNTR, and any implemented event counters PMEVCNTR<n>. Reading this register shows which counters are enabled.

PMCNTENSET is used in conjunction with the PMCNTENCLR register.

Configurations

AArch32 System register PMCNTENSET[31:0] is architecturally mapped to AArch64 System register PMCNTENSET_EL0[31:0].

AArch32 System register PMCNTENSET[31:0] is architecturally mapped to External register PMCNTENSET_EL0[31:0].

This register is present only when AArch32 is supported at any Exception level and PMUv3 is implemented. Otherwise, direct accesses to PMCNTENSET are UNDEFINED.

Attributes

PMCNTENSET is a 32-bit register.

Field descriptions

The PMCNTENSET bit assignments are:

31	30 … 0
C	P<n>, bit [n]

C, bit [31]

PMCCNTR enable bit. Enables the cycle counter register. Possible values are:

`0b0` When read, means the cycle counter is disabled. When written, has no effect.

`0b1` When read, means the cycle counter is enabled. When written, enables the cycle counter.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

P<n>, bit [n], for n = 0 to 30

Event counter enable bit for PMEVCNTR<n>.

If N is less than 31, then bits [30:N] are RAZ/WI. When EL2 is implemented and enabled in the current Security state, in EL1 and EL0, N is the value in MDCR_EL2.HPMN if EL2 is using AArch64, or in HDCR.HPMN if EL2 is using AArch32. Otherwise, N is the value in PMCR.N.

`0b0` When read, means that PMEVCNTR<n> is disabled. When written, has no effect.

`0b1` When read, means that PMEVCNTR<n> event counter is enabled. When written, enables PMEVCNTR<n>.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMCNTENSET

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && PMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMCNTEN == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMCNTENSET;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMCNTENSET;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMCNTENSET;
elsif PSTATE.EL == EL3 then
    return PMCNTENSET;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && PMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMCNTEN == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMCNTENSET = R[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMCNTENSET = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMCNTENSET = R[t];
elsif PSTATE.EL == EL3 then
    PMCNTENSET = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.4.9 PMCR, Performance Monitors Control Register

The PMCR characteristics are:

Purpose

Provides details of the Performance Monitors implementation, including the number of counters implemented, and configures and controls the counters.

Configurations

AArch32 System register PMCR[31:0] is architecturally mapped to AArch64 System register PMCR_EL0[31:0].

AArch32 System register PMCR[7:0] is architecturally mapped to External register PMCR_EL0[7:0].

This register is present only when AArch32 is supported at any Exception level and PMUv3 is implemented. Otherwise, direct accesses to PMCR are UNDEFINED.

Attributes

PMCR is a 32-bit register.

Field descriptions

The PMCR bit assignments are:

31 – 24	23 – 16	15 – 11	10 – 8	7	6	5	4	3	2	1	0
IMP	IDCODE	N	RES0	LP	LC	DP	X	D	C	P	E

IMP, bits [31:24]

Implementer code. This field has an IMPLEMENTATION DEFINED value.

If this field is zero, then PMCR.IDCODE is RES0 and software must use MIDR to identify the PE.

Otherwise this field and PMCR.IDCODE identify the PMU implementation to software. The implementer codes are allocated by Arm. A non-zero value has the same interpretation as MIDR.Implementer.

Access to this field is RO.

IDCODE, bits [23:16]

When PMCR.IMP != 0x00:

Identification code. This field has an IMPLEMENTATION DEFINED value.

Each implementer must maintain a list of identification codes that are specific to the implementer. A specific implementation is identified by the combination of the implementer code and the identification code.

Access to this field is RO.

Otherwise:

Reserved, RES0.

N, bits [15:11]

Indicates the number of event counters implemented. This value is in the range of `0b00000`-`0b111111`. If the value is `0b00000` then only PMCCNTR is implemented. If the value is `0b111111` PMCCNTR and 31 event counters are implemented.

In an implementation that includes EL2:

- If EL2 is using AArch32, reads of this field from Non-secure EL1 and Non-secure EL0 return the value of HDCR.HPMN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If EL2 is using AArch64 and enabled in the current Security state, reads of this field from EL1 and EL0 return the value of MDCR_EL2.HPMN.

Access to this field is RO.

Bits [10:8]

Reserved, RES0.

LP, bit [7]

When ARMv8.5-PMU is implemented:

Long event counter enable. Determines when unsigned overflow is recorded by a counter overflow bit.

`0b0` Event counter overflow on increment that causes unsigned overflow of PMEVCNTR<n>[31:0].

`0b1` Event counter overflow on increment that causes unsigned overflow of PMEVCNTR<n>[63:0].

If the highest implemented Exception level is using AArch32, it is IMPLEMENTATION DEFINED whether this bit is RW or RAZ/WI.

If EL2 is implemented and HDCR.HPMN or MDCR_EL2.HPMN is less than PMCR.N, this bit does not affect the operation of event counters in the range [HDCR.HPMN..(PMCR.N-1)] or [MDCR_EL2.HPMN..(PMCR.N-1)].

PMEVCNTR<n>[63:32] cannot be accessed directly in AArch32 state.

Note

The effect of HDCR.HPMN or MDCR_EL2.HPMN on the operation of this bit always applies if EL2 is implemented, at all Exception levels including EL2 and EL3, and regardless of whether EL2 is enabled in the current Security state. For more information, see the description of HDCR.HPMN or MDCR_EL2.HPMN.

On a Warm reset, this field resets to `0`.

Otherwise:

Reserved, RES0.

LC, bit [6]

Long cycle counter enable. Determines when unsigned overflow is recorded by the cycle counter overflow bit.

`0b0` Cycle counter overflow on increment that causes unsigned overflow of PMCCNTR[31:0].

`0b1` Cycle counter overflow on increment that causes unsigned overflow of PMCCNTR[63:0].

Arm deprecates use of PMCR.LC = 0.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

DP, bit [5]

When EL3 is implemented or (ARMv8.1-PMU is implemented and EL2 is implemented):

Disable cycle counter when event counting is prohibited.

`0b0` Cycle counting by PMCCNTR is not affected by this bit.

`0b1` When event counting for counters in the range [0..(HDCR.HPMN-1)] or [0..(MDCR_EL2.HPMN-1)] is prohibited, cycle counting by PMCCNTR is disabled.

For more information about the interaction between the Performance Monitors and EL3, see *Prohibiting event counting* on page D7-2709

On a Warm reset, this field resets to `0`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

Reserved, RES0.

X, bit [4]

When the implementation includes an PMU event export bus:

Enable export of events in an IMPLEMENTATION DEFINED PMU event export bus.

`0b0` Do not export events.

`0b1` Export events where not prohibited.

This field enables the exporting of events over an IMPLEMENTATION DEFINED PMU event export bus to another device, for example to an OPTIONAL PE trace unit.

No events are exported when counting is prohibited.

This field does not affect the generation of Performance Monitors overflow interrupt requests or signaling to a cross-trigger interface (CTI) that can be implemented as signals exported from the PE.

On a Warm reset, this field resets to `0`.

Otherwise:

Reserved, RAZ/WI.

D, bit [3]

Clock divider. The possible values of this bit are:

`0b0` When enabled, PMCCNTR counts every clock cycle.

`0b1` When enabled, PMCCNTR counts once every 64 clock cycles.

If PMCR.LC == 1, this bit is ignored and the cycle counter counts every clock cycle.

Arm deprecates use of PMCR.D = 1.

On a Warm reset, this field resets to `0`.

C, bit [2]

Cycle counter reset. The effects of writing to this bit are:

`0b0` No action.

`0b1` Reset PMCCNTR to zero.

This bit is always RAZ.

Note

Resetting PMCCNTR does not change the cycle counter overflow bit.

The value of PMCR_EL0.LC is ignored, and bits [63:0] of all affected event counters are reset.

Access to this field is WO.

P, bit [1]

Event counter reset. The effects of writing to this bit are:

`0b0` No action.

`0b1` Reset all event counters accessible in the current Exception level, not including PMCCNTR, to zero.

This bit is always RAZ.

In EL0 and EL1:

- If EL2 is implemented and enabled in the current Security state, and HDCR.HPMN or MDCR_EL2.HPMN is less than PMCR_EL0.N, a write of 1 to this bit does not reset event counters in the range [HDCR.HPMN..(PMCR.N-1)] or [MDCR_EL2.HPMN..(PMCR.N-1)].

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If EL2 is not implemented, EL2 is disabled in the current Security state, or HDCR.HPMN or MDCR_EL2.HPMN is equal to PMCR_EL0.N, a write of 1 to this bit resets all the event counters.

In EL2 and EL3, a write of 1 to this bit resets all the event counters.

Note

Resetting the event counters does not change the event counter overflow bits.

If ARMv8.5-PMU is implemented, the values of HDCR.HLP and PMCR.LP are ignored and bits [63:0] of all affected event counters are reset.

Access to this field is WO.

E, bit [0]

Enable.

`0b0` All event counters in the range [0..(PMN-1)] and PMCCNTR, are disabled.

`0b1` All event counters in the range [0..(PMN-1)] and PMCCNTR, are enabled by PMCNTENSET.

If EL2 is implemented then:

- If EL2 is using AArch32, PMN is HDCR.HPMN.
- If EL2 is using AArch64, PMN is MDCR_EL2.HPMN.
- If PMN is less than PMCR.N, this bit does not affect the operation of event counters in the range [PMN..(PMCR.N-1)].

If EL2 is not implemented, PMN is PMCR.N.

Note

The effect of MDCR_EL2.HPMN or HDCR.HPMN on the operation of this bit always applies if EL2 is implemented, at all Exception levels including EL2 and EL3, regardless of whether EL2 is enabled in the current Security state. For more information, see the description of MDCR_EL2.HPMN or HDCR.HPMN.

On a Warm reset, this field resets to `0`.

Accessing the PMCR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && PMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPMCR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPMCR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMCR;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPMCR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPMCR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMCR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMCR;
elsif PSTATE.EL == EL3 then
    return PMCR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && PMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMCR_EL0 == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPMCR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPMCR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMCR = R[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPMCR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPMCR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMCR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMCR = R[t];
elsif PSTATE.EL == EL3 then
    PMCR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.4.10 PMEVCNTR<n>, Performance Monitors Event Count Registers, n = 0 - 30

The PMEVCNTR<n> characteristics are:

Purpose

Holds event counter n, which counts events, where n is 0 to 30.

Configurations

AArch32 System register PMEVCNTR<n>[31:0] is architecturally mapped to AArch64 System register PMEVCNTR<n>_EL0[31:0].

AArch32 System register PMEVCNTR<n>[31:0] is architecturally mapped to External register PMEVCNTR<n>_EL0[31:0].

This register is present only when AArch32 is supported at any Exception level and PMUv3 is implemented. Otherwise, direct accesses to PMEVCNTR<n> are UNDEFINED.

Attributes

PMEVCNTR<n> is a 32-bit register.

Field descriptions

The PMEVCNTR<n> bit assignments are:

31	0
Event counter n	

Bits [31:0]

Event counter n. Value of event counter n, where n is the number of this register and is a number from 0 to 30.

If ARMv8.5-PMU is implemented, the event counter is 64 bits and only the least-significant part of the event counter is accessible in AArch32 state:

- Reads from PMEVCNTR<n> return bits [31:0] of the counter.
- Writes to PMEVCNTR<n> update bits [31:0] and leave bits [63:32] unchanged.
- There is no means to access bits [63:32] directly from AArch32 state.
- If the implementation does not support AArch64 at any Exception level, bits [63:32] are not required to be implemented.

If ARMv8.5-PMU is not implemented, the event counter is 32 bits.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMEVCNTR<n>

PMEVCNTR<n> can also be accessed by using PMXEVCNTR with PMSELR.SEL set to the value of <n>.

If ARMv8.6-FGT is implemented and <n> is greater than or equal to the number of accessible counters, then the behavior of permitted reads and writes of PMEVCNTR<n> is as follows:

- If <n> is an unimplemented event counter, the access is UNDEFINED.
- Otherwise, the access is trapped to EL2.

If ARMv8.6-FGT is not implemented and <n> is greater than or equal to the number of accessible counters, then reads and writes of PMEVCNTR<n> are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If <n> is greater than or equal to the number of accessible counters, then reads and writes of PMEVCNTR<n> are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP
- If EL2 is implemented and enabled in the current Security state, and <n> is less than the number of implemented counters, accesses from EL1 or permitted accesses from EL0 are trapped to EL2.

Note

In EL0, an access is permitted if it is enabled by PMUSERENR.{ER,EN} or PMUSERENR_EL0.{ER,EN}.

If EL2 is implemented and enabled in the current Security state, at EL0 and EL1:

- If EL2 is using AArch32, HDCR.HPMN identifies the number of accessible counters.
- If EL2 is using AArch64, MDCR_EL2.HPMN identifies the number of accessible counters.

Otherwise, the number of accessible counters is the number of implemented counters. See HDCR.HPMN and MDCR_EL2.HPMN for more details.

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b10:n[4:3]	n[2:0]

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.<ER,EN> == '00' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && PMUSERENR.<ER,EN> == '00' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMEVCNTRn_EL0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMEVCNTR[UInt(CRm<1:0>:opc2<2:0>)];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        return PMEVCNTR[UInt(CRm<1:0>:opc2<2:0>)];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMEVCNTR[UInt(CRm<1:0>:opc2<2:0>)];
elsif PSTATE.EL == EL3 then
    return PMEVCNTR[UInt(CRm<1:0>:opc2<2:0>)];
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b10:n[4:3]	n[2:0]

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && PMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMEVCNTRn_EL0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMEVCNTR[UInt(CRm<1:0>:opc2<2:0>)] = R[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMEVCNTR[UInt(CRm<1:0>:opc2<2:0>)] = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMEVCNTR[UInt(CRm<1:0>:opc2<2:0>)] = R[t];
elsif PSTATE.EL == EL3 then
    PMEVCNTR[UInt(CRm<1:0>:opc2<2:0>)] = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.4.11 PMEVTYPER<n>, Performance Monitors Event Type Registers, n = 0 - 30

The PMEVTYPER<n> characteristics are:

Purpose

Configures event counter n, where n is 0 to 30.

Configurations

AArch32 System register PMEVTYPER<n>[31:0] is architecturally mapped to AArch64 System register PMEVTYPER<n>_EL0[31:0].

AArch32 System register PMEVTYPER<n>[31:0] is architecturally mapped to External register PMEVTYPER<n>_EL0[31:0].

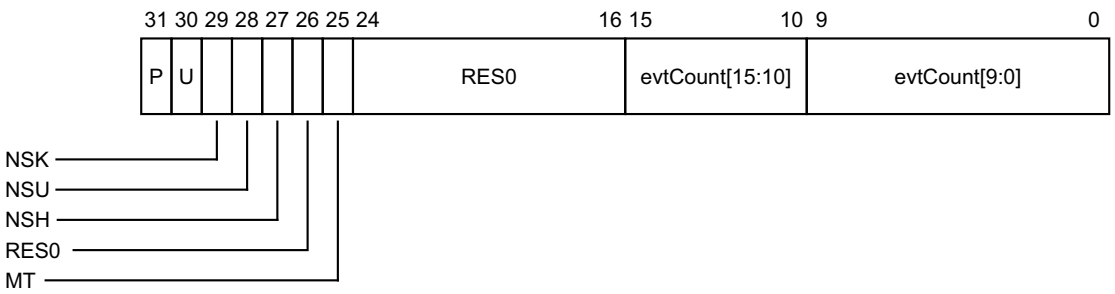
This register is present only when AArch32 is supported at any Exception level and PMUv3 is implemented. Otherwise, direct accesses to PMEVTYPER<n> are UNDEFINED.

Attributes

PMEVTYPER<n> is a 32-bit register.

Field descriptions

The PMEVTYPER<n> bit assignments are:



P, bit [31]

Privileged filtering bit. Controls counting in EL1.

If EL3 is implemented, then counting in Non-secure EL1 is further controlled by the PMEVTYPER<n>.NSK bit.

`0b0` Count events in EL1.

`0b1` Do not count events in EL1.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

U, bit [30]

User filtering bit. Controls counting in EL0.

If EL3 is implemented, then counting in Non-secure EL0 is further controlled by the PMEVTYPER<n>.NSU bit.

`0b0` Count events in EL0.

`0b1` Do not count events in EL0.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

NSK, bit [29]

When EL3 is implemented:

Non-secure EL1 (kernel) modes filtering bit. Controls counting in Non-secure EL1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the value of this bit is equal to the value of PMEVTYPER<n>.P, events in Non-secure EL1 are counted.

Otherwise, events in Non-secure EL1 are not counted.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSU, bit [28]

When EL3 is implemented:

Non-secure EL0 (Unprivileged) filtering. Controls counting in Non-secure EL0.

If the value of this bit is equal to the value of PMEVTYPER<n>.U, events in Non-secure EL0 are counted.

Otherwise, events in Non-secure EL0 are not counted.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSH, bit [27]

When EL2 is implemented:

EL2 (Hyp mode) filtering bit. Controls counting in EL2.

`0b0` Do not count events in EL2.

`0b1` Count events in EL2.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [26]

Reserved, RES0.

MT, bit [25]

When (ARMv8.6-MTPMU is implemented and enabled) or an IMPLEMENTATION DEFINED multi-threaded PMU Extension is implemented:

Multithreading.

`0b0` Count events only on controlling PE.

`0b1` Count events from any PE with the same affinity at level 1 and above as this PE.

Note

- When the lowest level of affinity consists of logical PEs that are implemented using a multi-threading type approach, an implementation is described as multi-threaded. That is, the performance of PEs at the lowest affinity level is highly interdependent.
- Events from a different thread of a multithreaded implementation are not Attributable to the thread counting the event.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [24:16]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

evtCount[15:10], bits [15:10]

When ARMv8.1-PMU is implemented:

Extension to evtCount[9:0]. See evtCount[9:0] for more details.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

evtCount[9:0], bits [9:0]

Event to count. The event number of the event that is counted by event counter PMEVCNTR<n>.

Software must program this field with an event that is supported by the PE being programmed.

The ranges of event numbers allocated to each type of event are shown in Table D7-7 on page D7-2721.

If evtCount is programmed to an event that is reserved or not supported by the PE, the behavior depends on the value written:

- For the range `0x0000` to `0x003F`, no events are counted, and the value returned by a direct or external read of the evtCount field is the value written to the field.
- If 16-bit evtCount is implemented, for the range `0x4000` to `0x403F`, no events are counted, and the value returned by a direct or external read of the evtCount field is the value written to the field.
- For IMPLEMENTATION DEFINED events, it is UNPREDICTABLE what event, if any, is counted, and the value returned by a direct or external read of the evtCount field is UNKNOWN.

Note

UNPREDICTABLE means the event must not expose privileged information.

Arm recommends that the behavior across a family of implementations is defined such that if a given implementation does not include an event from a set of common IMPLEMENTATION DEFINED events, then no event is counted and the value read back on evtCount is the value written.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMEVTYPER<n>

PMEVTYPER<n> can also be accessed by using PMXEVTYPER with PMSELR.SEL set to n.

If ARMv8.6-FGT is implemented and <n> is greater than or equal to the number of accessible counters, then the behavior of permitted reads and writes of PMEVTYPER<n> is as follows:

- If <n> is an unimplemented event counter, the access is UNDEFINED.
- Otherwise, the access is trapped to EL2.

If ARMv8.6-FGT is not implemented and <n> is greater than or equal to the number of accessible counters, then reads and writes of PMEVTYPER<n> are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

If <n> is greater or equal to the number of accessible counters, then reads and writes of PMEVTYPER<n> are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a `NOP`.
- If EL2 is implemented and enabled in the current Security state, and <n> is less than the number of implemented counters, accesses from EL1 or permitted accesses from EL0 are trapped to EL2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

In EL0, an access is permitted if it is enabled by PMUSERENR.EN or PMUSERENR_EL0.EN.

If EL2 is implemented and enabled in the current Security state, at EL0 and EL1:

- If EL2 is using AArch32, HDCR.HPMN identifies the number of accessible counters.
- If EL2 is using AArch64, MDCR_EL2.HPMN identifies the number of accessible counters.

Otherwise, the number of accessible counters is the number of implemented counters. See HDCR.HPMN and MDCR_EL2.HPMN for more details.

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b11:n[4:3]	n[2:0]

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && PMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMEVTYPERn_EL0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMEVTYPER[UInt(CRm<1:0>:opc2<2:0>)];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMEVTYPER[UInt(CRm<1:0>:opc2<2:0>)];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMEVTYPER[UInt(CRm<1:0>:opc2<2:0>)];
elsif PSTATE.EL == EL3 then
    return PMEVTYPER[UInt(CRm<1:0>:opc2<2:0>)];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b11:n[4:3]	n[2:0]

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && PMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMEVTYPERn_EL0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMEVTYPER[UInt(CRm<1:0>:opc2<2:0>)] = R[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMEVTYPER[UInt(CRm<1:0>:opc2<2:0>)] = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMEVTYPER[UInt(CRm<1:0>:opc2<2:0>)] = R[t];
elsif PSTATE.EL == EL3 then
    PMEVTYPER[UInt(CRm<1:0>:opc2<2:0>)] = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.4.12 PMINTENCLR, Performance Monitors Interrupt Enable Clear register

The PMINTENCLR characteristics are:

Purpose

Disables the generation of interrupt requests on overflows from the Cycle Count Register, PMCCNTR, and the event counters PMEVCNTR<n>. Reading the register shows which overflow interrupt requests are enabled.

PMINTENCLR is used in conjunction with the PMINTENSET register.

Configurations

AArch32 System register PMINTENCLR[31:0] is architecturally mapped to AArch64 System register PMINTENCLR_EL1[31:0].

AArch32 System register PMINTENCLR[31:0] is architecturally mapped to External register PMINTENCLR_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level and PMUv3 is implemented. Otherwise, direct accesses to PMINTENCLR are UNDEFINED.

Attributes

PMINTENCLR is a 32-bit register.

Field descriptions

The PMINTENCLR bit assignments are:

31	30 … 0
C	P<n>, bit [n]

C, bit [31]

PMCCNTR overflow interrupt request disable bit. Possible values are:

`0b0` When read, means the cycle counter overflow interrupt request is disabled. When written, has no effect.

`0b1` When read, means the cycle counter overflow interrupt request is enabled. When written, disables the cycle count overflow interrupt request.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

P<n>, bit [n], for n = 0 to 30

Event counter overflow interrupt request disable bit for PMEVCNTR<n>.

If N is less than 31, then bits [30:N] are RAZ/WI. When EL2 is implemented and enabled in the current Security state, in EL1, N is the value in MDCR_EL2.HPMN if EL2 is using AArch64, or in HDCR.HPMN if EL2 is using AArch32. Otherwise, N is the value in PMCR.N.

`0b0` When read, means that the PMEVCNTR<n> event counter interrupt request is disabled. When written, has no effect.

`0b1` When read, means that the PMEVCNTR<n> event counter interrupt request is enabled. When written, disables the PMEVCNTR<n> interrupt request.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMINTENCLR

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1110	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMINTENCLR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMINTENCLR;
elsif PSTATE.EL == EL3 then
    return PMINTENCLR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1110	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMINTENCLR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMINTENCLR = R[t];
elsif PSTATE.EL == EL3 then
    PMINTENCLR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.4.13 PMINTENSET, Performance Monitors Interrupt Enable Set register

The PMINTENSET characteristics are:

Purpose

Enables the generation of interrupt requests on overflows from the Cycle Count Register, PMCCNTR, and the event counters PMEVCNTR<n>. Reading the register shows which overflow interrupt requests are enabled.

PMINTENSET is used in conjunction with the PMINTENCLR register.

Configurations

AArch32 System register PMINTENSET[31:0] is architecturally mapped to AArch64 System register PMINTENSET_EL1[31:0].

AArch32 System register PMINTENSET[31:0] is architecturally mapped to External register PMINTENSET_EL1[31:0].

This register is present only when AArch32 is supported at any Exception level and PMUv3 is implemented. Otherwise, direct accesses to PMINTENSET are UNDEFINED.

Attributes

PMINTENSET is a 32-bit register.

Field descriptions

The PMINTENSET bit assignments are:

31	30 … 0
C	P<n>, bit [n]

C, bit [31]

PMCCNTR overflow interrupt request enable bit. Possible values are:

`0b0` When read, means the cycle counter overflow interrupt request is disabled. When written, has no effect.

`0b1` When read, means the cycle counter overflow interrupt request is enabled. When written, enables the cycle count overflow interrupt request.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

P<n>, bit [n], for n = 0 to 30

Event counter overflow interrupt request enable bit for PMEVCNTR<n>.

If N is less than 31, then bits [30:N] are RAZ/WI. When EL2 is implemented and enabled in the current Security state, in EL1, N is the value in MDCR_EL2.HPMN if EL2 is using AArch64, or in HDCR.HPMN if EL2 is using AArch32. Otherwise, N is the value in PMCR.N.

`0b0` When read, means that the PMEVCNTR<n> event counter interrupt request is disabled. When written, has no effect.

`0b1` When read, means that the PMEVCNTR<n> event counter interrupt request is enabled. When written, enables the PMEVCNTR<n> interrupt request.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMINTENSET

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1110	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMINTENSET;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMINTENSET;
elsif PSTATE.EL == EL3 then
    return PMINTENSET;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1110	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMINTENSET = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMINTENSET = R[t];
elsif PSTATE.EL == EL3 then
    PMINTENSET = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.4.14 PMOVSR, Performance Monitors Overflow Flag Status Register

The PMOVSR characteristics are:

Purpose

Contains the state of the overflow bit for the Cycle Count Register, PMCCNTR, and each of the implemented event counters PMEVCNTR<n>. Writing to this register clears these bits.

Configurations

AArch32 System register PMOVSR[31:0] is architecturally mapped to AArch64 System register PMOVSCLR_EL0[31:0].

AArch32 System register PMOVSR[31:0] is architecturally mapped to External register PMOVSCLR_EL0[31:0].

This register is present only when AArch32 is supported at any Exception level and PMUv3 is implemented. Otherwise, direct accesses to PMOVSR are UNDEFINED.

Attributes

PMOVSR is a 32-bit register.

Field descriptions

The PMOVSR bit assignments are:

31	30 … 0
C	P<n>, bit [n]

C, bit [31]

Cycle counter overflow clear bit. Possible values are:

`0b0` When read, means the cycle counter has not overflowed since this bit was last cleared. When written, has no effect.

`0b1` When read, means the cycle counter has overflowed since this bit was last cleared. When written, clears the cycle counter overflow bit to 0.

PMCR.LC controls whether an overflow is detected from unsigned overflow of PMCCNTR[31:0] or unsigned overflow of PMCCNTR[63:0].

On a Warm reset, this field resets to an architecturally UNKNOWN value.

P<n>, bit [n], for n = 0 to 30

Event counter overflow clear bit for PMEVCNTR<n>.

If N is less than 31, then bits [30:N] are RAZ/WI. When EL2 is implemented and enabled in the current Security state, in EL1 and EL0, N is the value in MDCR_EL2.HPMN if EL2 is using AArch64, or in HDCR.HPMN if EL2 is using AArch32. Otherwise, N is the value in PMCR.N.

`0b0` When read, means that PMEVCNTR<n> has not overflowed since this bit was last cleared. When written, has no effect.

`0b1` When read, means that PMEVCNTR<n> has overflowed since this bit was last cleared. When written, clears the PMEVCNTR<n> overflow bit to 0.

If ARMv8.5-PMU is implemented, MDCR_EL2.HLP, HDCR.HLP, and PMCR.LP control whether an overflow is detected from unsigned overflow of PMEVCNTR<n>[31:0] or unsigned overflow of PMEVCNTR<n>[63:0]. PMEVCNTR<n>[63:32] cannot be accessed directly in AArch32 state.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the PMOVSR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b011

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && PMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMOVS == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMOVSR;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMOVSR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMOVSR;
elsif PSTATE.EL == EL3 then
    return PMOVSR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b011

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && PMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMOVS == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMOVSR = R[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMOVSR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMOVSR = R[t];
elsif PSTATE.EL == EL3 then
    PMOVSR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.4.15 PMOVSSET, Performance Monitors Overflow Flag Status Set register

The PMOVSSET characteristics are:

Purpose

Sets the state of the overflow bit for the Cycle Count Register, PMCCNTR, and each of the implemented event counters PMEVCNTR<n>.

Configurations

AArch32 System register PMOVSSET[31:0] is architecturally mapped to AArch64 System register PMOVSSET_EL0[31:0].

AArch32 System register PMOVSSET[31:0] is architecturally mapped to External register PMOVSSET_EL0[31:0].

This register is present only when AArch32 is supported at any Exception level and PMUv3 is implemented. Otherwise, direct accesses to PMOVSSET are UNDEFINED.

Attributes

PMOVSSET is a 32-bit register.

Field descriptions

The PMOVSSET bit assignments are:

31	30 … 0
C	P<n>, bit [n]

C, bit [31]

Cycle counter overflow set bit.

`0b0` When read, means the cycle counter has not overflowed since this bit was last cleared. When written, has no effect.

`0b1` When read, means the cycle counter has overflowed since this bit was last cleared. When written, sets the cycle counter overflow bit to 1.

PMCR.LC controls whether an overflow is detected from unsigned overflow of PMCCNTR[31:0] or unsigned overflow of PMCCNTR[63:0].

On a Warm reset, this field resets to an architecturally UNKNOWN value.

P<n>, bit [n], for n = 0 to 30

Event counter overflow set bit for PMEVCNTR<n>.

If N is less than 31, then bits [30:N] are RAZ/WI. When EL2 is implemented and enabled in the current Security state, in EL1 and EL0, N is the value in MDCR_EL2.HPMN if EL2 is using AArch64, or in HDCR.HPMN if EL2 is using AArch32. Otherwise, N is the value in PMCR.N.

`0b0` When read, means that PMEVCNTR<n> has not overflowed since this bit was last cleared. When written, has no effect.

`0b1` When read, means that PMEVCNTR<n> has overflowed since this bit was last. When written, sets the PMEVCNTR<n> overflow bit to 1.

If ARMv8.5-PMU is implemented, MDCR_EL2.HLP, HDCR.HLP, and PMCR.LP control whether an overflow is detected from unsigned overflow of PMEVCNTR<n>[31:0] or unsigned overflow of PMEVCNTR<n>[63:0]. PMEVCNTR<n>[63:32] cannot be accessed directly in AArch32 state.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the PMOVSSET

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1110	0b011

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && PMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMOVS == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMOVSSET;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMOVSSET;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMOVSSET;
elsif PSTATE.EL == EL3 then
    return PMOVSSET;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1110	0b011

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && PMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMOVS == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMOVSSET = R[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMOVSSET = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMOVSSET = R[t];
elsif PSTATE.EL == EL3 then
    PMOVSSET = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.4.16 PMSELR, Performance Monitors Event Counter Selection Register

The PMSELR characteristics are:

Purpose

Selects the current event counter PMEVCNTR<n> or the cycle counter, CCNT.

PMSELR is used in conjunction with PMXEVTYPER to determine the event that increments a selected event counter, and the modes and states in which the selected counter increments.

It is also used in conjunction with PMXEVCNTR, to determine the value of a selected event counter.

Configurations

AArch32 System register PMSELR[31:0] is architecturally mapped to AArch64 System register PMSELR_EL0[31:0].

This register is present only when AArch32 is supported at any Exception level and PMUv3 is implemented. Otherwise, direct accesses to PMSELR are UNDEFINED.

Attributes

PMSELR is a 32-bit register.

Field descriptions

The PMSELR bit assignments are:

31 … 5	4 … 0
RES0	SEL

Bits [31:5]

Reserved, RES0.

SEL, bits [4:0]

Selects event counter, PMEVCNTR<n>, where n is the value held in this field. This value identifies which event counter is accessed when a subsequent access to PMXEVTYPER or PMXEVCNTR occurs.

This field can take any value from 0 (`0b00000`) to (PMCR.N)-1, or 31 (`0b11111`).

When PMSELR.SEL is `0b11111`, it selects the cycle counter and:

- A read of the PMXEVTYPER returns the value of PMCCFILTR.
- A write of the PMXEVTYPER writes to PMCCFILTR.
- A read or write of PMXEVCNTR has CONSTRAINED UNPREDICTABLE effects. See PMXEVCNTR for more details.

For details of the results of accesses to event counters, see PMXEVTYPER and PMXEVCNTR.

For information about the number of counters accessible at each Exception level, see HDCR.HPMN and MDCR_EL2.HPMN.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMSELR

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b101

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.<ER,EN> == '00' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && PMUSERENR.<ER,EN> == '00' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMSELR_EL0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMSELR;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMSELR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMSELR;
elsif PSTATE.EL == EL3 then
    return PMSELR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b101

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.<ER,EN> == '00' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && PMUSERENR.<ER,EN> == '00' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMSELR_EL0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMSELR = R[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMSELR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMSELR = R[t];
elsif PSTATE.EL == EL3 then
    PMSELR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.4.17 PMSWINC, Performance Monitors Software Increment register

The PMSWINC characteristics are:

Purpose

Increments a counter that is configured to count the Software increment event, event `0x00`. For more information, see SW_INCR.

Configurations

AArch32 System register PMSWINC[31:0] is architecturally mapped to AArch64 System register PMSWINC_EL0[31:0].

AArch32 System register PMSWINC[31:0] is architecturally mapped to External register PMSWINC_EL0[31:0].

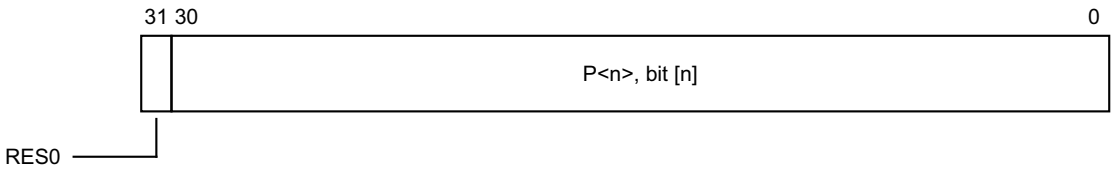
This register is present only when AArch32 is supported at any Exception level and PMUv3 is implemented. Otherwise, direct accesses to PMSWINC are UNDEFINED.

Attributes

PMSWINC is a 32-bit register.

Field descriptions

The PMSWINC bit assignments are:



Bit [31]

Reserved, RES0.

P<n>, bit [n], for n = 0 to 30

Event counter software increment bit for PMEVCNTR<n>.

If N is less than 31, then bits [30:N] are WI. When EL2 is implemented and enabled in the current Security state, in EL1 and EL0, N is the value in MDCR_EL2.HPMN if EL2 is using AArch64, or in HDCR.HPMN if EL2 is using AArch32. Otherwise, N is the value in PMCR.N.

`0b0` No action. The write to this bit is ignored.

`0b1` If PMEVCNTR<n> is enabled and configured to count the software increment event, increments PMEVCNTR<n> by 1. If PMEVCNTR<n> is disabled, or not configured to count the software increment event, the write to this bit is ignored.

Accessing the PMSWINC

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b100

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.<SW,EN> == '00' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && PMUSERENR.<SW,EN> == '00' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMSWINC_EL0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMSWINC = R[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMSWINC = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMSWINC = R[t];
elsif PSTATE.EL == EL3 then
    PMSWINC = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.4.18 PMUSERENR, Performance Monitors User Enable Register

The PMUSERENR characteristics are:

Purpose

Enables or disables User mode access to the Performance Monitors.

Configurations

AArch32 System register PMUSERENR[31:0] is architecturally mapped to AArch64 System register PMUSERENR_EL0[31:0].

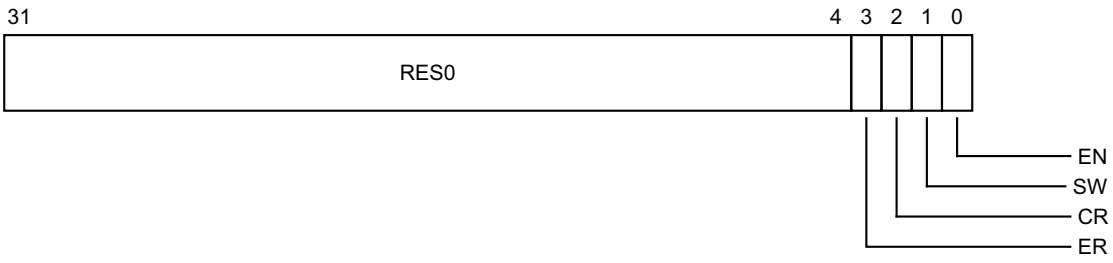
This register is present only when AArch32 is supported at any Exception level and PMUv3 is implemented. Otherwise, direct accesses to PMUSERENR are UNDEFINED.

Attributes

PMUSERENR is a 32-bit register.

Field descriptions

The PMUSERENR bit assignments are:



Bits [31:4]

Reserved, RES0.

ER, bit [3]

Event counter read trap control:

`0b0` EL0 reads of the PMXEVCNTR and PMEVCNTR<n>, and EL0 RW access to the PMSELR, are trapped to Undefined mode if PMUSERENR.EN is also 0.

`0b1` Overrides PMUSERENR.EN and enables RO access to PMXEVCNTR and PMEVCNTR<n>, and RW access to PMSELR.

On a Warm reset, this field resets to `0`.

CR, bit [2]

Cycle counter read trap control:

`0b0` EL0 reads of the PMCCNTR are trapped to Undefined mode if PMUSERENR.EN is also 0.

`0b1` Overrides PMUSERENR.EN and enables access to PMCCNTR.

On a Warm reset, this field resets to `0`.

SW, bit [1]

Software increment write trap control:

`0b0` EL0 writes to the PMSWINC are trapped to Undefined mode if PMUSERENR.EN is also 0.

`0b1` Overrides PMUSERENR.EN and enables access to PMSWINC.

On a Warm reset, this field resets to `0`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

EN, bit [0]

Traps EL0 accesses to the Performance Monitors registers to Undefined mode, as follows:

- PMCR, PMOVSR, PMSELR, PMCEID0, PMCEID1, PMCCNTR, PMXEVTYPER, PMXEVCNTR, PMCNTENSET, PMCNTENCLR, PMOVSSET, PMEVCNTR<n>, PMEVTYPER<n>, PMCCFILTR, PMSWINC.
- If ARMv8.1-PMU is implemented, PMCEID2, and PMCEID3.
- If ARMv8.4-PMU is implemented, PMMIR.

`0b0` While at EL0, accesses to the specified registers at EL0 are trapped to Undefined mode, unless overridden by one of PMUSERENR.{ER, CR, SW}.

`0b1` While at EL0, software can access all of the specified registers.

On a Warm reset, this field resets to `0`.

Accessing the PMUSERENR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1110	0b000

```
if PSTATE.EL == EL0 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMUSERENR_EL0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMUSERENR;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMUSERENR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMUSERENR;
elsif PSTATE.EL == EL3 then
    return PMUSERENR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1110	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMUSERENR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMUSERENR = R[t];
elsif PSTATE.EL == EL3 then
    PMUSERENR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.4.19 PMXEVCNTR, Performance Monitors Selected Event Count Register

The PMXEVCNTR characteristics are:

Purpose

Reads or writes the value of the selected event counter, PMEVCNTR<n>. PMSELR.SEL determines which event counter is selected.

Configurations

AArch32 System register PMXEVCNTR[31:0] is architecturally mapped to AArch64 System register PMXEVCNTR_EL0[31:0].

This register is present only when AArch32 is supported at any Exception level and PMUv3 is implemented. Otherwise, direct accesses to PMXEVCNTR are UNDEFINED.

Attributes

PMXEVCNTR is a 32-bit register.

Field descriptions

The PMXEVCNTR bit assignments are:

31 … 0
PMEVCNTR<n>

PMEVCNTR<n>, bits [31:0]

Value of the selected event counter, PMEVCNTR<n>, where n is the value stored in PMSELR.SEL.

If ARMv8.5-PMU is implemented, the event counter is 64 bits and only the least-significant part of the event counter is accessible in AArch32 state:

- Reads from PMXEVCNTR return bits [31:0] of the counter.
- Writes to PMXEVCNTR update bits [31:0] and leave bits [63:32] unchanged.
- There is no means to access bits [63:32] directly from AArch32 state.
- If the implementation does not support AArch64 at any Exception level, bits [63:32] are not required to be implemented.

If ARMv8.5-PMU is not implemented, the event counter is 32 bits.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMXEVCNTR

If ARMv8.6-FGT is implemented and PMSELR.SEL is greater than or equal to the number of accessible counters, then the behavior of permitted reads and writes of PMXEVCNTR is as follows:

- If PMSELR.SEL selects an unimplemented event counter, the access is UNDEFINED.
- Otherwise, the access is trapped to EL2.

If ARMv8.6-FGT is not implemented and PMSELR.SEL is greater than or equal to the number of accessible counters, then reads and writes of PMXEVCNTR are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Accesses to the register behave as if PMSELR.SEL has an UNKNOWN value less than the number of counters accessible at the current Exception level and Security state.
- If EL2 is implemented and enabled in the current Security state, and PMSELR.SEL is less than the number of implemented counters, accesses from EL1 or permitted accesses from EL0 are trapped to EL2.

Note

In EL0, an access is permitted if it is enabled by PMUSERENR.{ER,EN} or PMUSERENR_EL0.{ER,EN}.

If EL2 is implemented and enabled in the current Security state, at EL0 and EL1:

- If EL2 is using AArch32, HDCR.HPMN identifies the number of accessible counters.
- If EL2 is using AArch64, MDCR_EL2.HPMN identifies the number of accessible counters.

Otherwise, the number of accessible counters is the number of implemented counters. See HDCR.HPMN and MDCR_EL2.HPMN for more details.

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1101	0b010

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.<ER,EN> == '00' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && PMUSERENR.<ER,EN> == '00' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMEVCNTRn_EL0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMXEVCNTR;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMXEVCNTR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMXEVCNTR;
elsif PSTATE.EL == EL3 then
    return PMXEVCNTR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1101	0b010

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && PMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMEVCNTRn_EL0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMXEVCNTR = R[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMXEVCNTR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMXEVCNTR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    PMXEVCNTR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.4.20 PMXEVTYPER, Performance Monitors Selected Event Type Register

The PMXEVTYPER characteristics are:

Purpose

When PMSELR.SEL selects an event counter, this accesses a PMEVTYPER<n> register. When PMSELR.SEL selects the cycle counter, this accesses PMCCFILTR.

Configurations

AArch32 System register PMXEVTYPER[31:0] is architecturally mapped to AArch64 System register PMXEVTYPER_EL0[31:0].

This register is present only when AArch32 is supported at any Exception level and PMUv3 is implemented. Otherwise, direct accesses to PMXEVTYPER are UNDEFINED.

Attributes

PMXEVTYPER is a 32-bit register.

Field descriptions

The PMXEVTYPER bit assignments are:

31 … 0
Event type register or PMCCFILTR

Bits [31:0]

Event type register or PMCCFILTR.

When PMSELR.SEL == 31, this register accesses PMCCFILTR.

Otherwise, this register accesses PMEVTYPER<n> where n is the value in PMSELR.SEL.

Accessing the PMXEVTYPER

If ARMv8.6-FGT is implemented, and PMSELR.SEL is not 31 and is greater than or equal to the number of accessible counters, then the behavior of permitted reads and writes of PMXEVTYPER is as follows:

- If PMSELR.SEL selects an unimplemented event counter, the access is UNDEFINED.
- Otherwise, the access is trapped to EL2.

If ARMv8.6-FGT is not implemented, and PMSELR.SEL is not 31 and is greater than or equal to the number of accessible counters, then reads and writes of PMXEVTYPER are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a `NOP`
- Accesses to the register behave as if PMSELR.SEL has an UNKNOWN value less than the number of counters accessible at the current Exception level and Security state.
- Accesses to the register behave as if PMSELR.SEL is 31.
- If EL2 is implemented and enabled in the current Security state, and PMSELR.SEL is less than the number of implemented counters, accesses from EL1 or permitted accesses from EL0 are trapped to EL2.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

In EL0, an access is permitted if it is enabled by PMUSERENR.EN or PMUSERENR_EL0.EN.

If EL2 is implemented and enabled in the current Security state, at EL0 and EL1:

- If EL2 is using AArch32, HDCR.HPMN identifies the number of accessible counters.
- If EL2 is using AArch64, MDCR_EL2.HPMN identifies the number of accessible counters.

Otherwise, the number of accessible counters is the number of implemented counters. See HDCR.HPMN and MDCR_EL2.HPMN for more details.

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1101	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && PMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMEVTYPERn_EL0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMXEVTYPER;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return PMXEVTYPER;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        return PMXEVTYPER;
elsif PSTATE.EL == EL3 then
    return PMXEVTYPER;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1101	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && PMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMEVTYPERn_EL0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMXEVTYPER = R[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMXEVTYPER = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMXEVTYPER = R[t];
elsif PSTATE.EL == EL3 then
    PMXEVTYPER = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.5 Activity Monitors registers

This section lists the Activity Monitoring registers in AArch32.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.5.1 AMCFGR, Activity Monitors Configuration Register

The AMCFGR characteristics are:

Purpose

Global configuration register for the activity monitors.

Provides information on supported features, the number of counter groups implemented, the total number of activity monitor event counters implemented, and the size of the counters. AMCFGR is applicable to both the architected and the auxiliary counter groups.

Configurations

AArch32 System register AMCFGR[31:0] is architecturally mapped to AArch64 System register AMCFGR_EL0[31:0].

AArch32 System register AMCFGR[31:0] is architecturally mapped to External register AMCFGR[31:0].

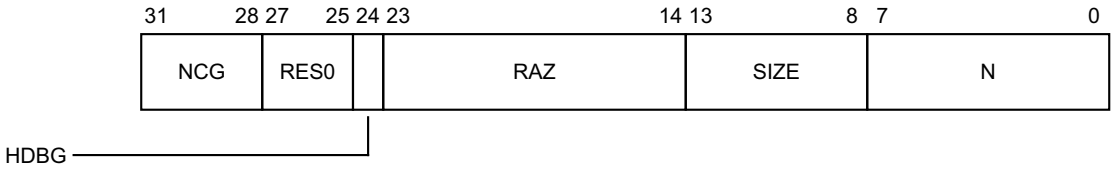
This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMCFGR are UNDEFINED.

Attributes

AMCFGR is a 32-bit register.

Field descriptions

The AMCFGR bit assignments are:



NCG, bits [31:28]

Defines the number of counter groups.

The number of implemented counter groups is defined as [AMCFGR.NCG + 1].

If the number of implemented auxiliary activity monitor event counters is zero, this field has a value of `0b0000`. Otherwise, this field has a value of `0b0001`.

Bits [27:25]

Reserved, RES0.

HDBG, bit [24]

Halt-on-debug supported.

From Armv8, this feature must be supported, and so this bit is `0b1`.

`0b0` AMCR.HDBG is RES0.

`0b1` AMCR.HDBG is read/write.

Bits [23:14]

Reserved, RAZ.

SIZE, bits [13:8]

Defines the size of activity monitor event counters.

The size of the activity monitor event counters implemented by the Activity Monitors Extension is defined as [AMCFGR.SIZE + 1].

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

From Armv8, the counters are 64-bit, and so this field is `0b111111`.

Note

Software also uses this field to determine the spacing of counters in the memory-map. From Armv8, the counters are at doubleword-aligned addresses.

N, bits [7:0]

Defines the number of activity monitor event counters.

The total number of counters implemented in all groups by the Activity Monitors Extension is defined as [AMCFGR.N + 1].

Accessing the AMCFGR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0010	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && AMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && AMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return AMCFGR;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return AMCFGR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        return AMCFGR;
elsif PSTATE.EL == EL3 then
    return AMCFGR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.5.2 AMCGCR, Activity Monitors Counter Group Configuration Register

The AMCGCR characteristics are:

Purpose

Provides information on the number of activity monitor event counters implemented within each counter group.

Configurations

AArch32 System register AMCGCR[31:0] is architecturally mapped to AArch64 System register AMCGCR_EL0[31:0].

AArch32 System register AMCGCR[31:0] is architecturally mapped to External register AMCGCR[31:0].

This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMCGCR are UNDEFINED.

Attributes

AMCGCR is a 32-bit register.

Field descriptions

The AMCGCR bit assignments are:

31 … 16	15 … 8	7 … 0
RES0	CG1NC	CG0NC

Bits [31:16]

Reserved, RES0.

CG1NC, bits [15:8]

Counter Group 1 Number of Counters. The number of counters in the auxiliary counter group.

In AMUv1, the permitted range of values is 0 to 16.

CG0NC, bits [7:0]

Counter Group 0 Number of Counters. The number of counters in the architected counter group.

In AMUv1, the value of this field is 4.

Accessing the AMCGCR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0010	0b010

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && AMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && AMUSERENR.EN == '0' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
                AArch64.AArch32SystemAccessTrap(EL2, 0x03);
            elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
                AArch32.TakeHypTrapException(0x00);
            else
                UNDEFINED;
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T13 == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
            AArch32.TakeHypTrapException(0x03);
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
            AArch32.TakeHypTrapException(0x03);
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        else
            return AMCGCR;
    elsif PSTATE.EL == EL1 then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
            AArch32.TakeHypTrapException(0x03);
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
            AArch32.TakeHypTrapException(0x03);
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        else
            return AMCGCR;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        else
            return AMCGCR;
    elsif PSTATE.EL == EL3 then
        return AMCGCR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.5.3 AMCNTENCLR0, Activity Monitors Count Enable Clear Register 0

The AMCNTENCLR0 characteristics are:

Purpose

Disable control bits for the architected activity monitors event counters, AMEVCNTR0<n>.

Configurations

AArch32 System register AMCNTENCLR0[31:0] is architecturally mapped to AArch64 System register AMCNTENCLR0_EL0[31:0].

AArch32 System register AMCNTENCLR0[31:0] is architecturally mapped to External register AMCNTENCLR0[31:0].

This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMCNTENCLR0 are UNDEFINED.

Attributes

AMCNTENCLR0 is a 32-bit register.

Field descriptions

The AMCNTENCLR0 bit assignments are:

31 … 16	15 … 0
RES0	P<n>, bit [n]

Bits [31:16]

Reserved, RES0.

P<n>, bit [n], for n = 0 to 15

Activity monitor event counter disable bit for AMEVCNTR0<n>.

Bits [31:16] are RES0. Bits [15:N] are RAZ/WI. N is the value in AMCGCR.CG0NC.

Possible values of each bit are:

`0b0` When read, means that AMEVCNTR0<n> is disabled. When written, has no effect.

`0b1` When read, means that AMEVCNTR0<n> is enabled. When written, disables AMEVCNTR0<n>.

On a Cold reset, this field resets to `0`.

Accessing the AMCNTENCLR0

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0010	0b100

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && AMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif ELUsingAArch32(EL1) && AMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HAFGRTR_EL2.AMCNTEN0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return AMCNTENCLR0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return AMCNTENCLR0;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return AMCNTENCLR0;
elsif PSTATE.EL == EL3 then
    return AMCNTENCLR0;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0010	0b100

```
if PSTATE.EL == EL1 && EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif PSTATE.EL == EL1 && EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif IsHighestEL(PSTATE.EL) then
    AMCNTENCLR0 = R[t];
else
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.5.4 AMCNTENCLR1, Activity Monitors Count Enable Clear Register 1

The AMCNTENCLR1 characteristics are:

Purpose

Disable control bits for the auxiliary activity monitors event counters, AMEVCNTR1<n>.

Configurations

AArch32 System register AMCNTENCLR1[31:0] is architecturally mapped to AArch64 System register AMCNTENCLR1_EL0[31:0].

AArch32 System register AMCNTENCLR1[31:0] is architecturally mapped to External register AMCNTENCLR1[31:0].

This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMCNTENCLR1 are UNDEFINED.

Attributes

AMCNTENCLR1 is a 32-bit register.

Field descriptions

The AMCNTENCLR1 bit assignments are:

31 … 16	15 … 0
RES0	P<n>, bit [n]

Bits [31:16]

Reserved, RES0.

P<n>, bit [n], for n = 0 to 15

Activity monitor event counter disable bit for AMEVCNTR1<n>.

Bits [31:16] are RES0. Bits [15:N] are RAZ/WI. N is the value in AMCGCR_EL0.CG1NC.

Possible values of each bit are:

`0b0` When read, means that AMEVCNTR1<n> is disabled. When written, has no effect.

`0b1` When read, means that AMEVCNTR1<n> is enabled. When written, disables AMEVCNTR1<n>.

On a Cold reset, this field resets to `0`.

Accessing the AMCNTENCLR1

If the number of auxiliary activity monitor event counters implemented is zero, reads and writes of AMCNTENCLR1 are UNDEFINED.

Note

The number of auxiliary activity monitor event counters implemented is zero exactly when AMCFGR.NCG == `0b0000`.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0011	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && AMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && AMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HAFGRTR_EL2.AMCNTEN1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return AMCNTENCLR1;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return AMCNTENCLR1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return AMCNTENCLR1;
elsif PSTATE.EL == EL3 then
    return AMCNTENCLR1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0011	0b000

```
if PSTATE.EL == EL1 && EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif PSTATE.EL == EL1 && EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif IsHighestEL(PSTATE.EL) then
    AMCNTENCLR1 = R[t];
else
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.5.5 AMCNTENSET0, Activity Monitors Count Enable Set Register 0

The AMCNTENSET0 characteristics are:

Purpose

Enable control bits for the architected activity monitors event counters, AMEVCNTR0<n>.

Configurations

AArch32 System register AMCNTENSET0[31:0] is architecturally mapped to AArch64 System register AMCNTENSET0_EL0[31:0].

AArch32 System register AMCNTENSET0[31:0] is architecturally mapped to External register AMCNTENSET0[31:0].

This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMCNTENSET0 are UNDEFINED.

Attributes

AMCNTENSET0 is a 32-bit register.

Field descriptions

The AMCNTENSET0 bit assignments are:

31 … 16	15 … 0
RES0	P<n>, bit [n]

Bits [31:16]

Reserved, RES0.

P<n>, bit [n], for n = 0 to 15

Activity monitor event counter enable bit for AMEVCNTR0<n>.

Bits [31:16] are RES0. Bits [15:N] are RAZ/WI. N is the value in AMCGCR.CG0NC.

Possible values of each bit are:

`0b0` When read, means that AMEVCNTR0<n> is disabled. When written, has no effect.

`0b1` When read, means that AMEVCNTR0<n> is enabled. When written, enables AMEVCNTR0<n>.

On a Cold reset, this field resets to `0`.

Accessing the AMCNTENSET0

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0010	0b101

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && AMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif ELUsingAArch32(EL1) && AMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HAFGRTR_EL2.AMCNTEN0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return AMCNTENSET0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return AMCNTENSET0;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return AMCNTENSET0;
elsif PSTATE.EL == EL3 then
    return AMCNTENSET0;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0010	0b101

```
if PSTATE.EL == EL1 && EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif PSTATE.EL == EL1 && EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif IsHighestEL(PSTATE.EL) then
    AMCNTENSET0 = R[t];
else
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.5.6 AMCNTENSET1, Activity Monitors Count Enable Set Register 1

The AMCNTENSET1 characteristics are:

Purpose

Enable control bits for the auxiliary activity monitors event counters, AMEVCNTR1<n>.

Configurations

AArch32 System register AMCNTENSET1[31:0] is architecturally mapped to AArch64 System register AMCNTENSET1_EL0[31:0].

AArch32 System register AMCNTENSET1[31:0] is architecturally mapped to External register AMCNTENSET1[31:0].

This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMCNTENSET1 are UNDEFINED.

Attributes

AMCNTENSET1 is a 32-bit register.

Field descriptions

The AMCNTENSET1 bit assignments are:

31 … 16	15 … 0
RES0	P<n>, bit [n]

Bits [31:16]

Reserved, RES0.

P<n>, bit [n], for n = 0 to 15

Activity monitor event counter enable bit for AMEVCNTR1<n>.

Bits [31:16] are RES0. Bits [15:N] are RAZ/WI. N is the value in AMCGCR.CG1NC.

Possible values of each bit are:

`0b0` When read, means that AMEVCNTR1<n> is disabled. When written, has no effect.

`0b1` When read, means that AMEVCNTR1<n> is enabled. When written, enables AMEVCNTR1<n>.

On a Cold reset, this field resets to `0`.

Accessing the AMCNTENSET1

If the number of auxiliary activity monitor event counters implemented is zero, reads and writes of AMCNTENSET1 are UNDEFINED.

Note

The number of auxiliary activity monitor counters implemented is zero when AMCFGR.NCG == `0b0000`.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0011	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && AMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && AMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HAFGRTR_EL2.AMCNTEN1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return AMCNTENSET1;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return AMCNTENSET1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return AMCNTENSET1;
elsif PSTATE.EL == EL3 then
    return AMCNTENSET1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0011	0b001

```
if PSTATE.EL == EL1 && EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif PSTATE.EL == EL1 && EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif IsHighestEL(PSTATE.EL) then
    AMCNTENSET1 = R[t];
else
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.5.7 AMCR, Activity Monitors Control Register

The AMCR characteristics are:

Purpose

Global control register for the activity monitors implementation. AMCR is applicable to both the architected and the auxiliary counter groups.

Configurations

AArch32 System register AMCR[31:0] is architecturally mapped to AArch64 System register AMCR_EL0[31:0].

AArch32 System register AMCR[31:0] is architecturally mapped to External register AMCR[31:0].

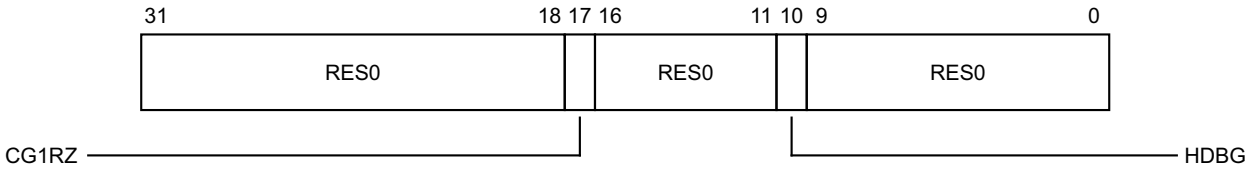
This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMCR are UNDEFINED.

Attributes

AMCR is a 32-bit register.

Field descriptions

The AMCR bit assignments are:



Bits [31:18]

Reserved, RES0.

CG1RZ, bit [17]

When ARMv8.6-AMU is implemented:

Counter Group 1 Read Zero.

`0b0` System register reads of AMEVCNTR1<n> return the event count at all implemented and enabled Exception levels.

`0b1` If the current Exception level is the highest implemented Exception level, system register reads of AMEVCNTR1<n> return the event count. Otherwise, reads of AMEVCNTR1<n> return a zero value.

Note

Reads from the memory-mapped view are unaffected by this field.

Otherwise:

Reserved, RES0.

Bits [16:11]

Reserved, RES0.

HDBG, bit [10]

This bit controls whether activity monitor counting is halted when the PE is halted in Debug state.

`0b0` Activity monitors do not halt counting when the PE is halted in Debug state.

`0b1` Activity monitors halt counting when the PE is halted in Debug state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bits [9:0]

Reserved, RES0.

Accessing the AMCR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0010	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && AMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && AMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return AMCR;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return AMCR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return AMCR;
elsif PSTATE.EL == EL3 then
    return AMCR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0010	0b000

```
if PSTATE.EL == EL1 && EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif PSTATE.EL == EL1 && EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif IsHighestEL(PSTATE.EL) then
    AMCR = R[t];
else
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.5.8 AMEVCNTR0<n>, Activity Monitors Event Counter Registers 0, n = 0 - 15

The AMEVCNTR0<n> characteristics are:

Purpose

Provides access to the architected activity monitor event counters.

Configurations

AArch32 System register AMEVCNTR0<n>[63:0] is architecturally mapped to AArch64 System register AMEVCNTR0<n>_EL0[63:0].

AArch32 System register AMEVCNTR0<n>[63:0] is architecturally mapped to External register AMEVCNTR0<n>[63:0].

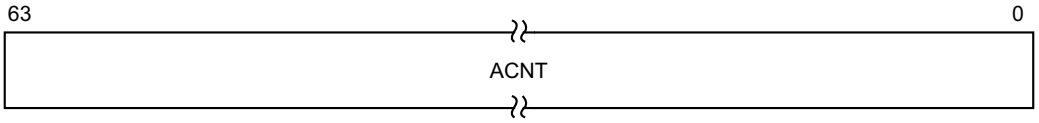
This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMEVCNTR0<n> are UNDEFINED.

Attributes

AMEVCNTR0<n> is a 64-bit register.

Field descriptions

The AMEVCNTR0<n> bit assignments are:



ACNT, bits [63:0]

Architected activity monitor event counter n.

Value of architected activity monitor event counter n, where n is the number of this register and is a number from 0 to 15.

If ARMv8.6-AMU is implemented, HCR_EL2.AMVOFFEN is 1, SCR_EL3.AMVOFFEN is 1, HCR_EL2.{E2H, TGE} is not {1,1}, and EL2 is using AArch64 and is implemented in the current Security state, access to these registers at EL0 or EL1 return (PCount<63:0> - AMEVCNTVOFF0<n>_EL2<63:0>).

PCount is the physical count returned when AMEVCNTR0<n> is read from EL2 or EL3.

If the counter is enabled, writes to this register have UNPREDICTABLE results.

On a Cold reset, this field resets to `0`.

Accessing the AMEVCNTR0<n>

If <n> is greater than or equal to the number of architected activity monitor event counters, reads and writes of AMEVCNTR0<n> are UNDEFINED.

Note

AMCGCR.CG0NC identifies the number of architected activity monitor event counters.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b000:n[3]	0b0:n[2:0]

```
if CRm == '0000' then
    if PSTATE.EL == EL0 then
        if !ELUsingAArch32(EL1) && AMUSERENR_EL0.EN == '0' then
            if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
                AArch64.AArch32SystemAccessTrap(EL2, 0x04);
            else
                AArch64.AArch32SystemAccessTrap(EL1, 0x04);
        elsif ELUsingAArch32(EL1) && AMUSERENR.EN == '0' then
            if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
                AArch64.AArch32SystemAccessTrap(EL2, 0x04);
            elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
                AArch32.TakeHypTrapException(0x00);
            else
                UNDEFINED;
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T0 == '1'
then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
            AArch32.TakeHypTrapException(0x04);
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
            AArch32.TakeHypTrapException(0x04);
        elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HAFGRTR_EL2.AMEVCNTR0<n>_EL0 == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x04);
        else
            return AMEVCNTR0[UInt(CRm<0>:opc1<2:0>)];
    elsif PSTATE.EL == EL1 then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
            AArch32.TakeHypTrapException(0x04);
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
            AArch32.TakeHypTrapException(0x04);
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x04);
        else
            return AMEVCNTR0[UInt(CRm<0>:opc1<2:0>)];
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x04);
        else
            return AMEVCNTR0[UInt(CRm<0>:opc1<2:0>)];
    elsif PSTATE.EL == EL3 then
        return AMEVCNTR0[UInt(CRm<0>:opc1<2:0>)];
else
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b000:n[3]	0b0:n[2:0]

```
if CRm == '0000' then
    if PSTATE.EL == EL1 && EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif PSTATE.EL == EL1 && EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x04);
    elsif IsHighestEL(PSTATE.EL) then
        AMEVCNTR0[UInt(CRm<0>:opc1<2:0>)] = R[t2]:R[t];
    else
        UNDEFINED;
else
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.5.9 AMEVCNTR1<n>, Activity Monitors Event Counter Registers 1, n = 0 - 15

The AMEVCNTR1<n> characteristics are:

Purpose

Provides access to the auxiliary activity monitor event counters.

Configurations

AArch32 System register AMEVCNTR1<n>[63:0] is architecturally mapped to AArch64 System register AMEVCNTR1<n>_EL0[63:0].

AArch32 System register AMEVCNTR1<n>[63:0] is architecturally mapped to External register AMEVCNTR1<n>[63:0].

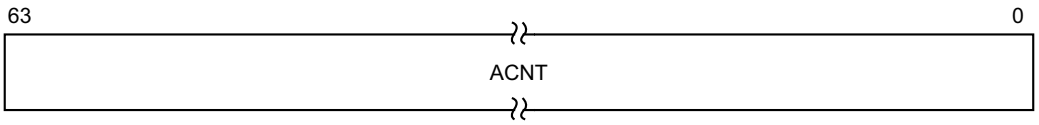
This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMEVCNTR1<n> are UNDEFINED.

Attributes

AMEVCNTR1<n> is a 64-bit register.

Field descriptions

The AMEVCNTR1<n> bit assignments are:



ACNT, bits [63:0]

Auxiliary activity monitor event counter n.

Value of auxiliary activity monitor event counter n, where n is the number of this register and is a number from 0 to 15.

If ARMv8.6-AMU is implemented, HCR_EL2.AMVOFFEN is 1, SCR_EL3.AMVOFFEN is 1, HCR_EL2.{E2H, TGE} is not {1,1}, EL2 is using AArch64 and is implemented in the current Security state, and AMCR_EL0.CG1RZ is 0, reads to these registers at EL0 or EL1 return (PCount<63:0> - AMEVCNTVOFF1<n>_EL2<63:0>).

PCount is the physical count returned when AMEVCNTR1<n> is read from EL2 or EL3.

If the counter is enabled, writes to this register have UNPREDICTABLE results.

On a Cold reset, this field resets to `0`.

Accessing the AMEVCNTR1<n>

If <n> is greater than or equal to the number of auxiliary activity monitor event counters, reads and writes of AMEVCNTR1<n> are UNDEFINED.

Note

AMCGCR.CG1NC identifies the number of auxiliary activity monitor event counters.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b010:n[3]	n[2:0]

```
if CRm == '0100' then
    if PSTATE.EL == EL0 then
        if !ELUsingAArch32(EL1) && AMUSERENR_EL0.EN == '0' then
            if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
                AArch64.AArch32SystemAccessTrap(EL2, 0x04);
            else
                AArch64.AArch32SystemAccessTrap(EL1, 0x04);
        elsif ELUsingAArch32(EL1) && AMUSERENR.EN == '0' then
            if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
                AArch64.AArch32SystemAccessTrap(EL2, 0x04);
            elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
                AArch32.TakeHypTrapException(0x00);
            else
                UNDEFINED;
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
            AArch32.TakeHypTrapException(0x04);
        elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HAFGRTR_EL2.AMEVCNTR1<n>_EL0 == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x04);
        elsif !HighestELUsingAArch32() && AMCR_EL0.CG1RZ == '1' then
            Zeros();
        elsif HighestELUsingAArch32() && AMCR.CG1RZ == '1' then
            Zeros();
        else
            return AMEVCNTR1[UInt(CRm<0>:opc1<2:0>)];
    elsif PSTATE.EL == EL1 then
        if EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
            AArch32.TakeHypTrapException(0x04);
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x04);
        elsif !IsHighestEL(PSTATE.EL) && !HighestELUsingAArch32() && AMCR_EL0.CG1RZ == '1' then
            Zeros();
        elsif !IsHighestEL(PSTATE.EL) && HighestELUsingAArch32() && AMCR.CG1RZ == '1' then
            Zeros();
        else
            return AMEVCNTR1[UInt(CRm<0>:opc1<2:0>)];
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x04);
        elsif !IsHighestEL(PSTATE.EL) && !HighestELUsingAArch32() && AMCR_EL0.CG1RZ == '1' then
            Zeros();
        elsif !IsHighestEL(PSTATE.EL) && HighestELUsingAArch32() && AMCR.CG1RZ == '1' then
            Zeros();
        else
            return AMEVCNTR1[UInt(CRm<0>:opc1<2:0>)];
    elsif PSTATE.EL == EL3 then
        return AMEVCNTR1[UInt(CRm<0>:opc1<2:0>)];
elsif CRm == '0101' then
    if PSTATE.EL == EL0 then
        if !ELUsingAArch32(EL1) && AMUSERENR_EL0.EN == '0' then
            if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
                AArch64.AArch32SystemAccessTrap(EL2, 0x04);
            else
                AArch64.AArch32SystemAccessTrap(EL1, 0x04);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        elsif ELUsingAArch32(EL1) && AMUSERENR.EN == '0' then
            if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
                AArch64.AArch32SystemAccessTrap(EL2, 0x04);
            elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
                AArch32.TakeHypTrapException(0x00);
            else
                UNDEFINED;
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T5 == '1'
then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
            AArch32.TakeHypTrapException(0x04);
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
            AArch32.TakeHypTrapException(0x04);
        elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HAFGRTR_EL2.AMEVCNTR1<n>_EL0 == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x04);
        elsif !HighestELUsingAArch32() && AMCR_EL0.CG1RZ == '1' then
            Zeros();
        elsif HighestELUsingAArch32() && AMCR.CG1RZ == '1' then
            Zeros();
        else
            return AMEVCNTR1[UInt(CRm<0>:opc1<2:0>)];
    elsif PSTATE.EL == EL1 then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
            AArch32.TakeHypTrapException(0x04);
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
            AArch32.TakeHypTrapException(0x04);
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x04);
        elsif !IsHighestEL(PSTATE.EL) && !HighestELUsingAArch32() && AMCR_EL0.CG1RZ == '1' then
            Zeros();
        elsif !IsHighestEL(PSTATE.EL) && HighestELUsingAArch32() && AMCR.CG1RZ == '1' then
            Zeros();
        else
            return AMEVCNTR1[UInt(CRm<0>:opc1<2:0>)];
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x04);
        elsif !IsHighestEL(PSTATE.EL) && !HighestELUsingAArch32() && AMCR_EL0.CG1RZ == '1' then
            Zeros();
        elsif !IsHighestEL(PSTATE.EL) && HighestELUsingAArch32() && AMCR.CG1RZ == '1' then
            Zeros();
        else
            return AMEVCNTR1[UInt(CRm<0>:opc1<2:0>)];
    elsif PSTATE.EL == EL3 then
        return AMEVCNTR1[UInt(CRm<0>:opc1<2:0>)];
else
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b010:n[3]	n[2:0]

```
if CRm == '0100' then
    if IsHighestEL(PSTATE.EL) then
        AMEVCNTR1[UInt(CRm<0>:opc1<2:0>)] = R[t2]:R[t];
    else
        UNDEFINED;
elsif CRm == '0101' then
    if PSTATE.EL == EL1 && EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif PSTATE.EL == EL1 && EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x04);
    elsif IsHighestEL(PSTATE.EL) then
        AMEVCNTR1[UInt(CRm<0>:opc1<2:0>)] = R[t2]:R[t];
    else
        UNDEFINED;
else
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.5.10 AMEVTYPER0<n>, Activity Monitors Event Type Registers 0, n = 0 - 15

The AMEVTYPER0<n> characteristics are:

Purpose

Provides information on the events that an architected activity monitor event counter AMEVCNTR0<n> counts.

Configurations

AArch32 System register AMEVTYPER0<n>[31:0] is architecturally mapped to AArch64 System register AMEVTYPER0<n>_EL0[31:0].

AArch32 System register AMEVTYPER0<n>[31:0] is architecturally mapped to External register AMEVTYPER0<n>[31:0].

This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMEVTYPER0<n> are UNDEFINED.

Attributes

AMEVTYPER0<n> is a 32-bit register.

Field descriptions

The AMEVTYPER0<n> bit assignments are:

31 … 16	15 … 0
RES0	evtCount

Bits [31:16]

Reserved, RES0.

evtCount, bits [15:0]

Event to count. The event number of the event that is counted by the architected activity monitor event counter AMEVCNTR0<n>. The value of this field is architecturally mandated for each architected counter.

The following table shows the mapping between required event numbers and the corresponding counters:

`0x0011` **When n == 0** Processor frequency cycles

`0x4004` **When n == 1** Constant frequency cycles

`0x0008` **When n == 2** Instructions retired

`0x4005` **When n == 3** Memory stall cycles

Accessing the AMEVTYPER0<n>

If <n> is greater than or equal to the number of architected activity monitor event counters, reads and writes of AMEVTYPER0<n> are NDEFINED.

Note

AMCGCR.CG0NC identifies the number of architected activity monitor event counters.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b011:n[3]	n[2:0]

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && AMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && AMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return AMEVTYPER0[UInt(CRm<0>:opc2<2:0>)];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return AMEVTYPER0[UInt(CRm<0>:opc2<2:0>)];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return AMEVTYPER0[UInt(CRm<0>:opc2<2:0>)];
elsif PSTATE.EL == EL3 then
    return AMEVTYPER0[UInt(CRm<0>:opc2<2:0>)];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.5.11 AMEVTYPER1<n>, Activity Monitors Event Type Registers 1, n = 0 - 15

The AMEVTYPER1<n> characteristics are:

Purpose

Provides information on the events that an auxiliary activity monitor event counter AMEVCNTR1<n> counts.

Configurations

AArch32 System register AMEVTYPER1<n>[31:0] is architecturally mapped to AArch64 System register AMEVTYPER1<n>_EL0[31:0].

AArch32 System register AMEVTYPER1<n>[31:0] is architecturally mapped to External register AMEVTYPER1<n>[31:0].

This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMEVTYPER1<n> are UNDEFINED.

Attributes

AMEVTYPER1<n> is a 32-bit register.

Field descriptions

The AMEVTYPER1<n> bit assignments are:

31 … 16	15 … 0
RES0	evtCount

Bits [31:16]

Reserved, RES0.

evtCount, bits [15:0]

Event to count. The event number of the event that is counted by the auxiliary activity monitor event counter AMEVCNTR1<n>.

It is IMPLEMENTATION DEFINED what values are supported by each counter.

If software writes a value to this field which is not supported by the corresponding counter AMEVCNTR1<n>, then:

- It is UNPREDICTABLE which event will be counted.
- The value read back is UNKNOWN.

The event counted by AMEVCNTR1<n> might be fixed at implementation. In this case, the field is read-only and writes are UNDEFINED.

If the corresponding counter AMEVCNTR1<n> is enabled, writes to this register have UNPREDICTABLE results.

Accessing the AMEVTYPER1<n>

If <n> is greater than or equal to the number of auxiliary activity monitor event counters, reads and writes of AMEVTYPER1<n> are UNDEFINED.

Note

AMCGCR.CG1NC identifies the number of auxiliary activity monitor event counters.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b111:n[3]	n[2:0]

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && AMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && AMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1') && HAFGRTR_EL2.AMEVTYPER1<n>_EL0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return AMEVTYPER1[UInt(CRm<0>:opc2<2:0>)];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return AMEVTYPER1[UInt(CRm<0>:opc2<2:0>)];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return AMEVTYPER1[UInt(CRm<0>:opc2<2:0>)];
elsif PSTATE.EL == EL3 then
    return AMEVTYPER1[UInt(CRm<0>:opc2<2:0>)];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b111:n[3]	n[2:0]

```
if PSTATE.EL == EL1 && EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif PSTATE.EL == EL1 && EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif IsHighestEL(PSTATE.EL) then
    AMEVTYPER1[UInt(CRm<0>:opc2<2:0>)] = R[t];
else
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.5.12 AMUSERENR, Activity Monitors User Enable Register

The AMUSERENR characteristics are:

Purpose

Global user enable register for the activity monitors. Enables or disables EL0 access to the activity monitors. AMUSERENR is applicable to both the architected and the auxiliary counter groups.

Configurations

AArch32 System register AMUSERENR[31:0] is architecturally mapped to AArch64 System register AMUSERENR_EL0[31:0].

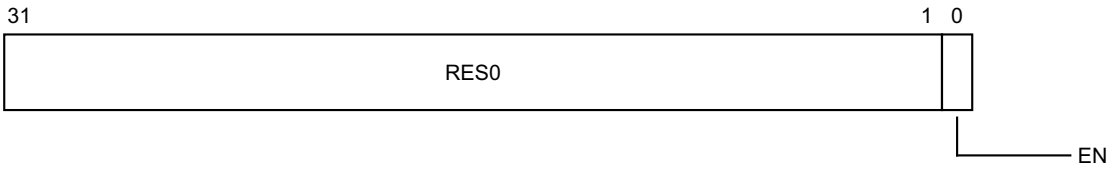
This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMUSERENR are UNDEFINED.

Attributes

AMUSERENR is a 32-bit register.

Field descriptions

The AMUSERENR bit assignments are:



Bits [31:1]

Reserved, RES0.

EN, bit [0]

Traps EL0 accesses to the activity monitors registers to EL1.

`0b0` EL0 accesses to the activity monitors registers are trapped to EL1.

`0b1` This control does not cause any instructions to be trapped. Software can access all activity monitor registers at EL0.

Note

- AMUSERENR can always be read at EL0 and is not governed by this bit.

Accessing the AMUSERENR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0010	0b011

```
if PSTATE.EL == EL0 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return AMUSERENR;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return AMUSERENR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        return AMUSERENR;
elsif PSTATE.EL == EL3 then
    return AMUSERENR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0010	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        AMUSERENR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        AMUSERENR = R[t];
elsif PSTATE.EL == EL3 then
    AMUSERENR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.6 RAS registers

This section lists the RAS Extension registers in AArch32.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.6.1 DISR, Deferred Interrupt Status Register

The DISR characteristics are:

Purpose

Records that an SError interrupt has been consumed by an ESB instruction.

Configurations

AArch32 System register DISR[31:0] is architecturally mapped to AArch64 System register DISR_EL1[31:0] when the highest implemented Exception level is using AArch64.

This register is present only when RAS is implemented. Otherwise, direct accesses to DISR are UNDEFINED.

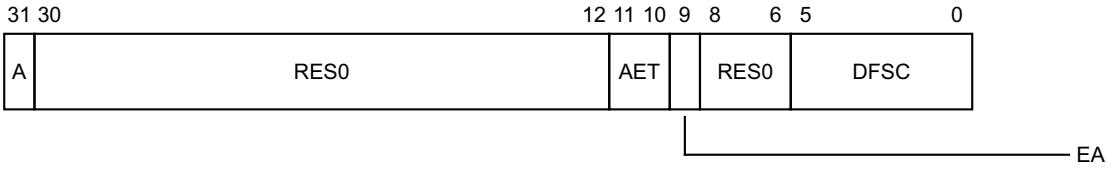
Attributes

DISR is a 32-bit register.

Field descriptions

The DISR bit assignments are:

When the ESB instruction is executed at EL2:



A, bit [31]

Set to 1 when an ESB instruction defers an asynchronous SError interrupt. If the implementation does not include any sources of SError interrupt that can be synchronized by an Error Synchronization Barrier, then this bit is RES0.

This field resets to an architecturally UNKNOWN value.

Bits [30:12]

Reserved, RES0.

AET, bits [11:10]

Asynchronous Error Type. See the description of HSR.AET for an SError interrupt.

This field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort Type. See the description of HSR.EA for an SError interrupt.

This field resets to an architecturally UNKNOWN value.

Bits [8:6]

Reserved, RES0.

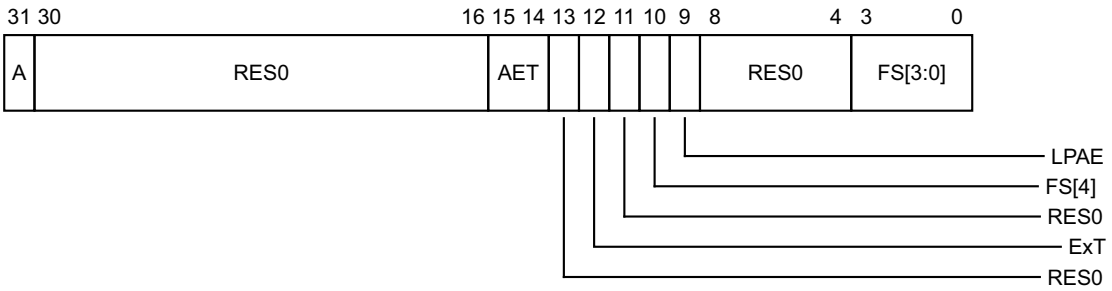
DFSC, bits [5:0]

Fault Status Code. See the description of HSR.DFSC for an SError interrupt.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When the ESB instruction is executed at EL0 or EL1 and where TTBCR.EAE == 0:



A, bit [31]

Set to 1 when an `ESB` instruction defers an asynchronous SError interrupt. If the implementation does not include any sources of SError interrupt that can be synchronized by an Error Synchronization Barrier, then this bit is RES0.

This field resets to an architecturally UNKNOWN value.

Bits [30:16]

Reserved, RES0.

AET, bits [15:14]

Asynchronous Error Type. See the description of DFSR.AET for an SError interrupt.

This field resets to an architecturally UNKNOWN value.

Bit [13]

Reserved, RES0.

ExT, bit [12]

External abort Type. See the description of DFSR.ExT for an SError interrupt.

This field resets to an architecturally UNKNOWN value.

Bit [11]

Reserved, RES0.

FS[4], bit [10]

This field is bit[4] of FS[4:0].

Fault Status Code. See the description of DFSR.FS for an SError interrupt.

The FS field is split as follows:

- FS[4] is DISR[10].
- FS[3:0] is DISR[3:0].

This field resets to an architecturally UNKNOWN value.

LPAE, bit [9]

Format.

`0b0` Using the Short-descriptor translation table format.

This field resets to an architecturally UNKNOWN value.

Bits [8:4]

Reserved, RES0.

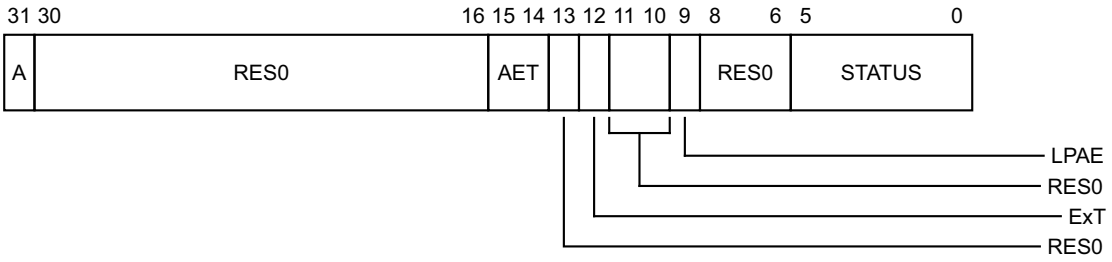
Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

FS[3:0], bits [3:0]

This field is bits[3:0] of FS[4:0].

See FS[4] for the field description.

When the ESB instruction is executed at EL0 or EL1 and where TTBCR.EAE == 1:



A, bit [31]

Set to 1 when an `ESB` instruction defers an asynchronous SError interrupt. If the implementation does not include any sources of SError interrupt that can be synchronized by an Error Synchronization Barrier, then this bit is RES0.

This field resets to an architecturally UNKNOWN value.

Bits [30:16]

Reserved, RES0.

AET, bits [15:14]

Asynchronous Error Type. See the description of DFSR.AET for an SError interrupt.

This field resets to an architecturally UNKNOWN value.

Bit [13]

Reserved, RES0.

ExT, bit [12]

External abort Type. See the description of DFSR.ExT for an SError interrupt.

This field resets to an architecturally UNKNOWN value.

Bits [11:10]

Reserved, RES0.

LPAE, bit [9]

Format.

`0b1` Using the Long-descriptor translation table format.

This field resets to an architecturally UNKNOWN value.

Bits [8:6]

Reserved, RES0.

STATUS, bits [5:0]

Fault Status Code. See the description of DFSR.FS for an SError interrupt.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the DISR

An indirect write to DISR made by an ESB instruction does not require an explicit synchronization operation for the value that is written to be observed by a direct read of DISR occurring in program order after the ESB instruction.

DISR is RAZ/WI if EL3 is implemented, the PE is in Non-debug state, and any of the following apply:

- EL3 is using AArch64, SCR_EL3.EA == 1, and any of the following apply:
  - The PE is executing at EL2.
  - The PE is executing at EL1 and ((SCR_EL3.NS == 0 && SCR_EL3.EEL2 == 0) || HCR_EL2.AMO == 0).
- EL3 is using AArch32, SCR.EA == 1, and any of the following apply:
  - The PE is executing at EL2.
  - The PE is executing at EL1 and (SCR.NS == 0 || HCR.AMO == 0).

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1100	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T12 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T12 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.AMO == '1' then
        return VDISR_EL2;
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.AMO == '1' then
        return VDISR;
    else
        return DISR;
elsif PSTATE.EL == EL2 then
    return DISR;
elsif PSTATE.EL == EL3 then
    return DISR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1100	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T12 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T12 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.AMO == '1' then
        VDISR_EL2 = R[t];
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.AMO == '1' then
        VDISR = R[t];
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        DISR = R[t];
elsif PSTATE.EL == EL2 then
    DISR = R[t];
elsif PSTATE.EL == EL3 then
    DISR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.6.2 ERRIDR, Error Record ID Register

The ERRIDR characteristics are:

Purpose

Defines the highest numbered index of the error records that can be accessed through the Error Record System registers.

Configurations

AArch32 System register ERRIDR[31:0] is architecturally mapped to AArch64 System register ERRIDR_EL1[31:0].

This register is present only when RAS is implemented. Otherwise, direct accesses to ERRIDR are UNDEFINED.

Attributes

ERRIDR is a 32-bit register.

Field descriptions

The ERRIDR bit assignments are:

31 … 16	15 … 0
RES0	NUM

Bits [31:16]

Reserved, RES0.

NUM, bits [15:0]

Highest numbered index of the records that can be accessed through the Error Record System registers plus one. Zero indicates that no records can be accessed through the Error Record System registers.

Each implemented record is owned by a node. A node might own multiple records.

Accessing the ERRIDR

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERRIDR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERRIDR;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERRIDR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.6.3 ERRSELR, Error Record Select Register

The ERRSELR characteristics are:

Purpose

Selects an error record to be accessed through the Error Record System registers.

Configurations

AArch32 System register ERRSELR[31:0] is architecturally mapped to AArch64 System register ERRSELR_EL1[31:0].

This register is present only when RAS is implemented. Otherwise, direct accesses to ERRSELR are UNDEFINED.

If ERRIDR indicates that zero error records are implemented, then it is IMPLEMENTATION DEFINED whether ERRSELR is UNDEFINED or RES0.

Attributes

ERRSELR is a 32-bit register.

Field descriptions

The ERRSELR bit assignments are:

31 … 16	15 … 0
RES0	SEL

Bits [31:16]

Reserved, RES0.

SEL, bits [15:0]

Selects the error record accessed through the ERX registers.

For example, if ERRSELR.SEL is set to `0x0004`, then direct reads and writes of ERXSTATUS access ERR4STATUS.

If ERRSELR.SEL is set to a value greater than or equal to ERRIDR.NUM, then all of the following apply:

- The value read back from ERRSELR.SEL is UNKNOWN.
- One of the following occurs:
  - An UNKNOWN error record is selected.
  - The ERX* registers are RAZ/WI.
  - ERX* register reads and writes are NOPs.
  - ERX* register reads and writes are UNDEFINED.

This field resets to an architecturally UNKNOWN value.

Accessing the ERRSELR

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0011	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERRSELR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERRSELR;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERRSELR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0011	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERRSELR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            AArch32.TakeMonitorTrapException();
        else
            ERRSELR = R[t];
    elsif PSTATE.EL == EL3 then
        if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
            AArch32.TakeMonitorTrapException();
        else
            ERRSELR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.6.4 ERXADDR, Selected Error Record Address Register

The ERXADDR characteristics are:

Purpose

Accesses bits [31:0] of ERR<n>ADDR for the error record <n> selected by ERRSELR.SEL.

For details of this, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*.

Configurations

AArch32 System register ERXADDR[31:0] is architecturally mapped to AArch64 System register ERXADDR_EL1[31:0].

This register is present only when RAS is implemented. Otherwise, direct accesses to ERXADDR are UNDEFINED.

Attributes

ERXADDR is a 32-bit register.

Field descriptions

The ERXADDR bit assignments are:

31 … 0
Bits [31:0] of ERR<n>ADDR

Bits [31:0]

ERXADDR accesses bits [31:0] of ERR<n>ADDR, where <n> is the value in ERRSELR.SEL.

Accessing the ERXADDR

If ERRIDR.NUM == `0x0000` or ERRSELR.SEL is set to a value greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXADDR is RAZ/WI.
- Direct reads and writes of ERXADDR are NOPs.
- Direct reads and writes of ERXADDR are UNDEFINED.

ERR<n>ADDR describes additional constraints that also apply when ERR<n>ADDR is accessed through ERXADDR.

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0100	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXADDR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXADDR;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXADDR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0100	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXADDR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXADDR = R[t];
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXADDR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.6.5 ERXADDR2, Selected Error Record Address Register 2

The ERXADDR2 characteristics are:

Purpose

Accesses bits [63:32] of ERR<n>ADDR for the error record <n> selected by ERRSELR.SEL.

For details of this, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*.

Configurations

AArch32 System register ERXADDR2[31:0] is architecturally mapped to AArch64 System register ERXADDR_EL1[63:32].

This register is present only when RAS is implemented. Otherwise, direct accesses to ERXADDR2 are UNDEFINED.

Attributes

ERXADDR2 is a 32-bit register.

Field descriptions

The ERXADDR2 bit assignments are:

31 … 0
Bits [63:32] of ERR<n>ADDR

Bits [31:0]

ERXADDR2 accesses bits [63:32] of ERR<n>ADDR, where <n> is the value in ERRSELR.SEL.

Accessing the ERXADDR2

If ERRIDR.NUM == `0x0000` or ERRSELR.SEL is set to a value greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXADDR2 is RAZ/WI.
- Direct reads and writes of ERXADDR2 are NOPs.
- Direct reads and writes of ERXADDR2 are UNDEFINED.

ERR<n>ADDR describes additional constraints that also apply when ERR<n>ADDR is accessed through ERXADDR2.

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0100	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXADDR2;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXADDR2;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXADDR2;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0100	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXADDR2 = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXADDR2 = R[t];
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXADDR2 = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.6.6 ERXCTLR, Selected Error Record Control Register

The ERXCTLR characteristics are:

Purpose

Accesses bits [31:0] of ERR<n>CTLR for the error record <n> selected by ERRSELR.SEL.

For details of this, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*.

Configurations

AArch32 System register ERXCTLR[31:0] is architecturally mapped to AArch64 System register ERXCTLR_EL1[31:0].

This register is present only when RAS is implemented. Otherwise, direct accesses to ERXCTLR are UNDEFINED.

Attributes

ERXCTLR is a 32-bit register.

Field descriptions

The ERXCTLR bit assignments are:

31	0
Bits [31:0] of ERR<n>CTLR	

Bits [31:0]

ERXCTLR accesses bits [31:0] of ERR<n>CTLR, where <n> is the value in ERRSELR.SEL.

Accessing the ERXCTLR

If ERRIDR.NUM == `0x0000` or ERRSELR.SEL is set to a value greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXCTLR is RAZ/WI.
- Direct reads and writes of ERXCTLR are NOPs.
- Direct reads and writes of ERXCTLR are UNDEFINED.

If ERRSELR.SEL is not the index of the first error record owned by a node, then ERR<n>CTLR[31:0] is not present, meaning reads and writes of ERXCTLR are RES0.

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0100	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXCTLR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXCTLR;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXCTLR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0100	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXCTLR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXCTLR = R[t];
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXCTLR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.6.7 ERXCTLR2, Selected Error Record Control Register 2

The ERXCTLR2 characteristics are:

Purpose

Accesses bits [63:32] of ERR<n>CTLR for the error record <n> selected by ERRSELR.SEL.

For details of this, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*.

Configurations

AArch32 System register ERXCTLR2[31:0] is architecturally mapped to AArch64 System register ERXCTLR_EL1[63:32].

This register is present only when RAS is implemented. Otherwise, direct accesses to ERXCTLR2 are UNDEFINED.

Attributes

ERXCTLR2 is a 32-bit register.

Field descriptions

The ERXCTLR2 bit assignments are:

31 … 0
Bits [63:32] of ERR<n>CTLR

Bits [31:0]

ERXCTLR2 accesses bits [63:32] of ERR<n>CTLR, where <n> is the value in ERRSELR.SEL.

Accessing the ERXCTLR2

If ERRIDR.NUM == `0x0000` or ERRSELR.SEL is set to a value greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXCTLR2 is RAZ/WI.
- Direct reads and writes of ERXCTLR2 are NOPs.
- Direct reads and writes of ERXCTLR2 are UNDEFINED.

If ERRSELR.SEL is not the index of the first error record owned by a node, then ERR<n>CTLR[63:32] is not present, meaning reads and writes of ERXCTLR2 are RES0.

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0100	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXCTLR2;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXCTLR2;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXCTLR2;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0100	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXCTLR2 = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXCTLR2 = R[t];
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXCTLR2 = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.6.8 ERXFR, Selected Error Record Feature Register

The ERXFR characteristics are:

Purpose

Accesses bits [31:0] of ERR<n>FR for the error record <n> selected by ERRSELR.SEL.

For details of this, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*.

Configurations

AArch32 System register ERXFR[31:0] is architecturally mapped to AArch64 System register ERXFR_EL1[31:0].

This register is present only when RAS is implemented. Otherwise, direct accesses to ERXFR are UNDEFINED.

Attributes

ERXFR is a 32-bit register.

Field descriptions

The ERXFR bit assignments are:

31 … 0
Bits [31:0] of ERR<n>FR

Bits [31:0]

ERXFR accesses bits [31:0] of ERR<n>FR, where <n> is the value in ERRSELR.SEL.

Accessing the ERXFR

If ERRIDR.NUM == `0x0000` or ERRSELR.SEL is set to a value greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXFR is RAZ.
- Direct reads of ERXFR are NOPs.
- Direct reads of ERXFR are UNDEFINED.

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0100	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
            AArch32.TakeHypTrapException(0x03);
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
            AArch32.TakeMonitorTrapException();
        else
            return ERXFR;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
            AArch32.TakeMonitorTrapException();
        else
            return ERXFR;
    elsif PSTATE.EL == EL3 then
        if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
            AArch32.TakeMonitorTrapException();
        else
            return ERXFR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.6.9 ERXFR2, Selected Error Record Feature Register 2

The ERXFR2 characteristics are:

Purpose

Accesses bits [63:32] of ERR<n>FR for the error record <n> selected by ERRSELR.SEL.

For details of this, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*.

Configurations

AArch32 System register ERXFR2[31:0] is architecturally mapped to AArch64 System register ERXFR_EL1[63:32].

This register is present only when RAS is implemented. Otherwise, direct accesses to ERXFR2 are UNDEFINED.

Attributes

ERXFR2 is a 32-bit register.

Field descriptions

The ERXFR2 bit assignments are:

31 … 0
Bits [63:32] of ERR<n>FR

Bits [31:0]

ERXFR2 accesses bits [63:32] of ERR<n>FR, where <n> is the value in ERRSELR.SEL.

Accessing the ERXFR2

If ERRIDR.NUM == `0x0000` or ERRSELR.SEL is set to a value greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXFR2 is RAZ.
- Direct reads of ERXFR2 are NOPs.
- Direct reads of ERXFR2 are UNDEFINED.

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0100	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
            AArch32.TakeHypTrapException(0x03);
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
            AArch32.TakeMonitorTrapException();
        else
            return ERXFR2;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
            AArch32.TakeMonitorTrapException();
        else
            return ERXFR2;
    elsif PSTATE.EL == EL3 then
        if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
            AArch32.TakeMonitorTrapException();
        else
            return ERXFR2;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.6.10 ERXMISC0, Selected Error Record Miscellaneous Register 0

The ERXMISC0 characteristics are:

Purpose

Accesses bits [31:0] of ERR<n>MISC0 for the error record <n> selected by ERRSELR.SEL.

For details of this, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*.

Configurations

AArch32 System register ERXMISC0[31:0] is architecturally mapped to AArch64 System register ERXMISC0_EL1[31:0].

This register is present only when RAS is implemented. Otherwise, direct accesses to ERXMISC0 are UNDEFINED.

Attributes

ERXMISC0 is a 32-bit register.

Field descriptions

The ERXMISC0 bit assignments are:

31	0
Bits [31:0] of ERR<n>MISC0	

Bits [31:0]

ERXMISC0 accesses bits [31:0] of ERR<n>MISC0, where <n> is the value in ERRSELR.SEL.

Accessing the ERXMISC0

If ERRIDR.NUM == `0x0000` or ERRSELR.SEL is set to a value greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC0 is RAZ/WI.
- Direct reads and writes of ERXMISC0 are NOPs.
- Direct reads and writes of ERXMISC0 are UNDEFINED.

ERR<n>MISC0 describes additional constraints that also apply when ERR<n>MISC0 is accessed through ERXMISC0.

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0101	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXMISC0;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXMISC0;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXMISC0;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0101	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXMISC0 = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXMISC0 = R[t];
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXMISC0 = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.6.11 ERXMISC1, Selected Error Record Miscellaneous Register 1

The ERXMISC1 characteristics are:

Purpose

Accesses bits [63:32] of ERR<n>MISC0 for the error record <n> selected by ERRSELR.SEL.

For details of this, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*.

Configurations

AArch32 System register ERXMISC1[31:0] is architecturally mapped to AArch64 System register ERXMISC0_EL1[63:32].

This register is present only when RAS is implemented. Otherwise, direct accesses to ERXMISC1 are UNDEFINED.

Attributes

ERXMISC1 is a 32-bit register.

Field descriptions

The ERXMISC1 bit assignments are:

31 … 0
Bits [63:32] of ERR<n>MISC0

Bits [31:0]

ERXMISC1 accesses bits [63:32] of ERR<n>MISC0, where <n> is the value in ERRSELR.SEL.

Accessing the ERXMISC1

If ERRIDR.NUM == `0x0000` or ERRSELR.SEL is set to a value greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC1 is RAZ/WI.
- Direct reads and writes of ERXMISC1 are NOPs.
- Direct reads and writes of ERXMISC1 are UNDEFINED.

ERR<n>MISC0 describes additional constraints that also apply when ERR<n>MISC0 is accessed through ERXMISC1.

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0101	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXMISC1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXMISC1;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXMISC1;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0101	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXMISC1 = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXMISC1 = R[t];
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXMISC1 = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.6.12 ERXMISC2, Selected Error Record Miscellaneous Register 2

The ERXMISC2 characteristics are:

Purpose

Accesses bits [31:0] of ERR<n>MISC1 for the error record <n> selected by ERRSELR.SEL.

For details of this, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*.

Configurations

AArch32 System register ERXMISC2[31:0] is architecturally mapped to AArch64 System register ERXMISC1_EL1[31:0].

This register is present only when RAS is implemented. Otherwise, direct accesses to ERXMISC2 are UNDEFINED.

Attributes

ERXMISC2 is a 32-bit register.

Field descriptions

The ERXMISC2 bit assignments are:

31	0
Bits [31:0] of ERR<n>MISC1	

Bits [31:0]

ERXMISC2 accesses bits [31:0] of ERR<n>MISC1, where <n> is the value in ERRSELR.SEL.

Accessing the ERXMISC2

If ERRIDR.NUM == `0x0000` or ERRSELR.SEL is set to a value greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC2 is RAZ/WI.
- Direct reads and writes of ERXMISC2 are NOPs.
- Direct reads and writes of ERXMISC2 are UNDEFINED.

ERR<n>MISC1 describes additional constraints that also apply when ERR<n>MISC1 is accessed through ERXMISC2.

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0101	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXMISC2;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXMISC2;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXMISC2;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0101	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXMISC2 = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXMISC2 = R[t];
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXMISC2 = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.6.13 ERXMISC3, Selected Error Record Miscellaneous Register 3

The ERXMISC3 characteristics are:

Purpose

Accesses bits [63:32] of ERR<n>MISC1 for the error record <n> selected by ERRSELR.SEL.

For details of this, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*.

Configurations

AArch32 System register ERXMISC3[31:0] is architecturally mapped to AArch64 System register ERXMISC1_EL1[63:32].

This register is present only when RAS is implemented. Otherwise, direct accesses to ERXMISC3 are UNDEFINED.

Attributes

ERXMISC3 is a 32-bit register.

Field descriptions

The ERXMISC3 bit assignments are:

31 … 0
Bits [63:32] of ERR<n>MISC1

Bits [31:0]

ERXMISC3 accesses bits [63:32] of ERR<n>MISC1, where <n> is the value in ERRSELR.SEL.

Accessing the ERXMISC3

If ERRIDR.NUM == `0x0000` or ERRSELR.SEL is set to a value greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC3 is RAZ/WI.
- Direct reads and writes of ERXMISC3 are NOPs.
- Direct reads and writes of ERXMISC3 are UNDEFINED.

ERR<n>MISC1 describes additional constraints that also apply when ERR<n>MISC1 is accessed through ERXMISC3.

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0101	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXMISC3;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXMISC3;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXMISC3;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0101	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXMISC3 = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXMISC3 = R[t];
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXMISC3 = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.6.14 ERXMISC4, Selected Error Record Miscellaneous Register 4

The ERXMISC4 characteristics are:

Purpose

Accesses bits [31:0] of ERR<n>MISC2 for the error record <n> selected by ERRSELR.SEL.

For details of this, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*.

Configurations

AArch32 System register ERXMISC4[31:0] is architecturally mapped to AArch64 System register ERXMISC2_EL1[31:0].

This register is present only when ARMv8.4-RAS is implemented. Otherwise, direct accesses to ERXMISC4 are UNDEFINED.

Attributes

ERXMISC4 is a 32-bit register.

Field descriptions

The ERXMISC4 bit assignments are:

31 … 0
Bits [31:0] of ERR<n>MISC2

Bits [31:0]

ERXMISC4 accesses bits [31:0] of ERR<n>MISC2, where <n> is the value in ERRSELR.SEL.

Accessing the ERXMISC4

If ERRIDR.NUM == `0x0000` or ERRSELR.SEL is set to a value greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC4 is RAZ/WI.
- Direct reads and writes of ERXMISC4 are NOPs.
- Direct reads and writes of ERXMISC4 are UNDEFINED.

ERR<n>MISC2 describes additional constraints that also apply when ERR<n>MISC2 is accessed through ERXMISC4.

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0101	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXMISC4;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXMISC4;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXMISC4;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0101	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXMISC4 = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXMISC4 = R[t];
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXMISC4 = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.6.15 ERXMISC5, Selected Error Record Miscellaneous Register 5

The ERXMISC5 characteristics are:

Purpose

Accesses bits [63:32] of ERR<n>MISC2 for the error record <n> selected by ERRSELR.SEL.

For details of this, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*.

Configurations

AArch32 System register ERXMISC5[31:0] is architecturally mapped to AArch64 System register ERXMISC2_EL1[63:32].

This register is present only when ARMv8.4-RAS is implemented. Otherwise, direct accesses to ERXMISC5 are UNDEFINED.

Attributes

ERXMISC5 is a 32-bit register.

Field descriptions

The ERXMISC5 bit assignments are:

31 … 0
Bits [63:32] of ERR<n>MISC2

Bits [31:0]

ERXMISC5 accesses bits [63:32] of ERR<n>MISC2, where <n> is the value in ERRSELR.SEL.

Accessing the ERXMISC5

If ERRIDR.NUM == `0x0000` or ERRSELR.SEL is set to a value greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC5 is RAZ/WI.
- Direct reads and writes of ERXMISC5 are NOPs.
- Direct reads and writes of ERXMISC5 are UNDEFINED.

ERR<n>MISC2 describes additional constraints that also apply when ERR<n>MISC2 is accessed through ERXMISC5.

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0101	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXMISC5;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXMISC5;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXMISC5;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0101	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXMISC5 = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXMISC5 = R[t];
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXMISC5 = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.6.16 ERXMISC6, Selected Error Record Miscellaneous Register 6

The ERXMISC6 characteristics are:

Purpose

Accesses bits [31:0] of ERR<n>MISC3 for the error record <n> selected by ERRSELR.SEL.

For details of this, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*.

Configurations

AArch32 System register ERXMISC6[31:0] is architecturally mapped to AArch64 System register ERXMISC3_EL1[31:0].

This register is present only when ARMv8.4-RAS is implemented. Otherwise, direct accesses to ERXMISC6 are UNDEFINED.

Attributes

ERXMISC6 is a 32-bit register.

Field descriptions

The ERXMISC6 bit assignments are:

31 … 0
Bits [31:0] of ERR<n>MISC3

Bits [31:0]

ERXMISC6 accesses bits [31:0] of ERR<n>MISC3, where <n> is the value in ERRSELR.SEL.

Accessing the ERXMISC6

If ERRIDR.NUM == `0x0000` or ERRSELR.SEL is set to a value greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC6 is RAZ/WI.
- Direct reads and writes of ERXMISC6 are NOPs.
- Direct reads and writes of ERXMISC6 are UNDEFINED.

ERR<n>MISC3 describes additional constraints that also apply when ERR<n>MISC3 is accessed through ERXMISC6.

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0101	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXMISC6;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXMISC6;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXMISC6;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0101	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXMISC6 = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXMISC6 = R[t];
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXMISC6 = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.6.17 ERXMISC7, Selected Error Record Miscellaneous Register 7

The ERXMISC7 characteristics are:

Purpose

Accesses bits [63:32] of ERR<n>MISC3 for the error record <n> selected by ERRSELR.SEL.

For details of this, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*.

Configurations

AArch32 System register ERXMISC7[31:0] is architecturally mapped to AArch64 System register ERXMISC3_EL1[63:32].

This register is present only when ARMv8.4-RAS is implemented. Otherwise, direct accesses to ERXMISC7 are UNDEFINED.

Attributes

ERXMISC7 is a 32-bit register.

Field descriptions

The ERXMISC7 bit assignments are:

31 … 0
Bits [63:32] of ERR<n>MISC3

Bits [31:0]

ERXMISC7 accesses bits [63:32] of ERR<n>MISC3, where <n> is the value in ERRSELR.SEL.

Accessing the ERXMISC7

If ERRIDR.NUM == `0x0000` or ERRSELR.SEL is set to a value greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC7 is RAZ/WI.
- Direct reads and writes of ERXMISC7 are NOPs.
- Direct reads and writes of ERXMISC7 are UNDEFINED.

ERR<n>MISC3 describes additional constraints that also apply when ERR<n>MISC3 is accessed through ERXMISC7.

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0101	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXMISC7;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXMISC7;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXMISC7;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0101	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXMISC7 = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXMISC7 = R[t];
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXMISC7 = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.6.18 ERXSTATUS, Selected Error Record Primary Status Register

The ERXSTATUS characteristics are:

Purpose

Accesses bits [31:0] of ERR<n>STATUS for the error record selected by ERRSELR.SEL.

For details of this, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*.

Configurations

AArch32 System register ERXSTATUS[31:0] is architecturally mapped to AArch64 System register ERXSTATUS_EL1[31:0].

This register is present only when RAS is implemented. Otherwise, direct accesses to ERXSTATUS are UNDEFINED.

Attributes

ERXSTATUS is a 32-bit register.

Field descriptions

The ERXSTATUS bit assignments are:

31 … 0
Bits [31:0] of ERR<n>STATUS

Bits [31:0]

ERXSTATUS accesses bits [31:0] of ERR<n>STATUS, where n is the value in ERRSELR.SEL.

Accessing the ERXSTATUS

If ERRIDR.NUM == 0 or ERRSELR.SEL is set to a value greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN record is selected.
- ERXSTATUS is RAZ/WI.
- Direct reads and writes of ERXSTATUS are NOPs.
- Direct reads and writes of ERXSTATUS are UNDEFINED.

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0100	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXSTATUS;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXSTATUS;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        return ERXSTATUS;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0100	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXSTATUS = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXSTATUS = R[t];
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERXSTATUS = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.6.19 VDFSR, Virtual SError Exception Syndrome Register

The VDFSR characteristics are:

Purpose

Provides the syndrome value reported to software on taking a virtual SError interrupt exception to EL1, or on executing an ESB instruction at EL1.

When a virtual SError interrupt is taken, the syndrome value is reported in DFSR.{AET, ExT} and the remainder of the DFSR is set as defined by VMSAv8-32. For more information, see Chapter G5 *The AArch32 Virtual Memory System Architecture*.

If the virtual SError interrupt is deferred by an ESB instruction, then the syndrome value is written to VDISR.

Configurations

AArch32 System register VDFSR[31:0] is architecturally mapped to AArch64 System register VSESR_EL2[31:0] when the highest implemented Exception level is using AArch64.

This register is present only when RAS is implemented. Otherwise, direct accesses to VDFSR are UNDEFINED.

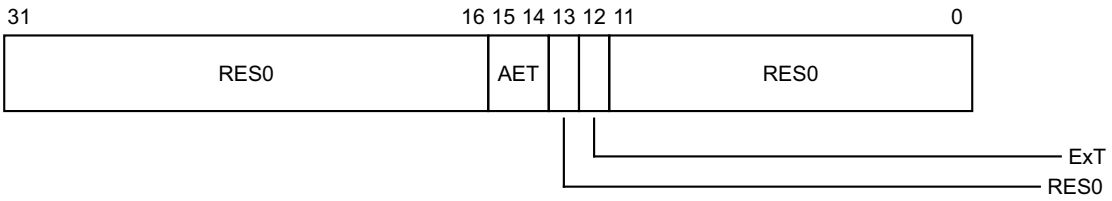
If EL2 is not implemented, then VDFSR is RES0 from Monitor mode when SCR.NS == 1.

Attributes

VDFSR is a 32-bit register.

Field descriptions

The VDFSR bit assignments are:



Bits [31:16]

Reserved, RES0.

AET, bits [15:14]

When a virtual SError interrupt is taken to EL1 using AArch32, DFSR[15:4] is set to VDFSR.AET.

When a virtual SError interrupt is deferred by an ESB instruction, VDISR[15:4] is set to VDFSR.AET.

This field resets to an architecturally UNKNOWN value.

Bit [13]

Reserved, RES0.

ExT, bit [12]

When a virtual SError interrupt is taken to EL1 using AArch32, DFSR[12] is set to VDFSR.ExT.

When a virtual SError interrupt is deferred by an ESB instruction, VDISR[12] is set to VDFSR.ExT.

This field resets to an architecturally UNKNOWN value.

Bits [11:0]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the VDFSR

Direct reads and writes of VDFSR are UNDEFINED if EL3 is implemented and using AArch32 in all Secure privileged modes other than Monitor mode.

If EL2 is not implemented, then VDFSR is RES0 from Monitor mode when SCR.NS == 1.

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0101	0b0010	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return VDFSR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        return VDFSR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0101	0b0010	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    VDFSR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        VDFSR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.6.20 VDISR, Virtual Deferred Interrupt Status Register

The VDISR characteristics are:

Purpose

Records that an SError interrupt has been consumed by an ESB instruction.

Configurations

AArch32 System register VDISR[31:0] is architecturally mapped to AArch64 System register VDISR_EL2[31:0].

This register is present only when RAS is implemented. Otherwise, direct accesses to VDISR are UNDEFINED.

If EL2 is not implemented, then VDISR is RES0 from Monitor mode when SCR.NS == 1.

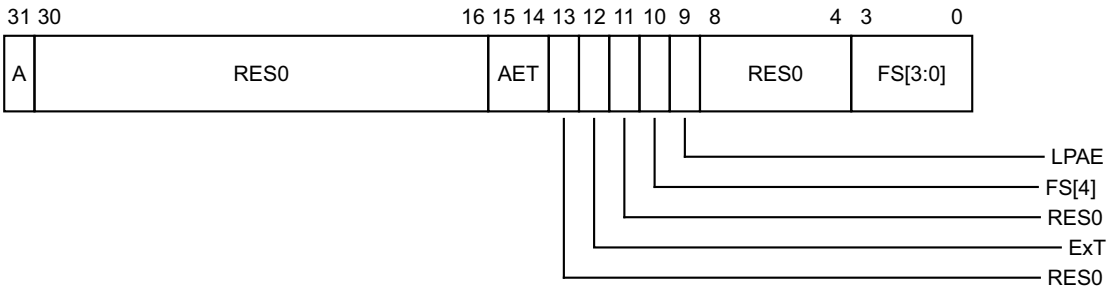
Attributes

VDISR is a 32-bit register.

Field descriptions

The VDISR bit assignments are:

When TTBCR.EAE == 0:



A, bit [31]

Set to 1 when an ESB instruction defers a virtual SError interrupt.

This field resets to an architecturally UNKNOWN value.

Bits [30:16]

Reserved, RES0.

AET, bits [15:14]

The value copied from VDFSR.AET.

This field resets to an architecturally UNKNOWN value.

Bit [13]

Reserved, RES0.

ExT, bit [12]

The value copied from VDFSR.ExT.

This field resets to an architecturally UNKNOWN value.

Bit [11]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

FS[4], bit [10]

This field is bit[4] of FS[4:0].

Fault status code. Set to `0b10110` when an `ESB` instruction defers a virtual SError interrupt.

`0b10110` Asynchronous SError interrupt.

All other values are reserved.

The FS field is split as follows:

- FS[4] is VDISR[10].
- FS[3:0] is VDISR[3:0].

This field resets to an architecturally UNKNOWN value.

LPAE, bit [9]

Format.

Set to TTBCR.EAE when an `ESB` instruction defers a virtual SError interrupt.

`0b0` Using the Short-descriptor translation table format.

This field resets to an architecturally UNKNOWN value.

Bits [8:4]

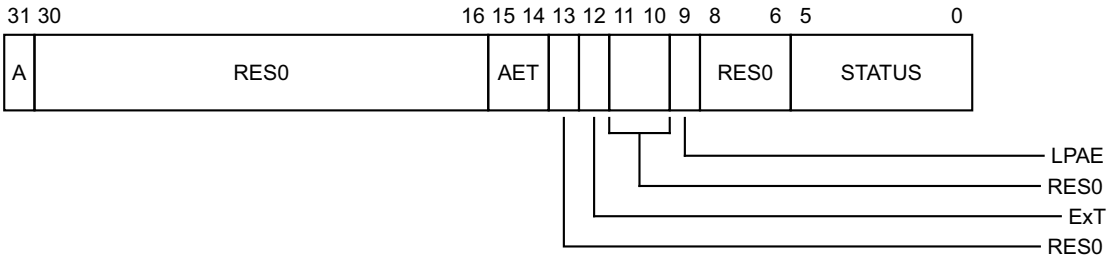
Reserved, RES0.

FS[3:0], bits [3:0]

This field is bits[3:0] of FS[4:0].

See FS[4] for the field description.

When TTBCR.EAE == 1:



A, bit [31]

Set to 1 when an `ESB` instruction defers a virtual SError interrupt.

This field resets to an architecturally UNKNOWN value.

Bits [30:16]

Reserved, RES0.

AET, bits [15:14]

The value copied from VDFSR.AET.

This field resets to an architecturally UNKNOWN value.

Bit [13]

Reserved, RES0.

ExT, bit [12]

The value copied from VDFSR.ExT.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bits [11:10]

Reserved, RES0.

LPAE, bit [9]

Format.

Set to TTBCR.EAE when an ESB instruction defers a virtual SError interrupt.

0b1 Using the Long-descriptor translation table format.

This field resets to an architecturally UNKNOWN value.

Bits [8:6]

Reserved, RES0.

STATUS, bits [5:0]

Fault status code. Set to 0b010001 when an ESB instruction defers a virtual SError interrupt.

0b010001 Asynchronous SError interrupt.

All other values are reserved.

This field resets to an architecturally UNKNOWN value.

Accessing the VDISR

Direct reads and writes of VDFSR are UNDEFINED if EL3 is implemented and using AArch32 in all Secure privileged modes other than Monitor mode.

An indirect write to VDISR made by an ESB instruction does not require an explicit synchronization operation for the value that is written to be observed by a direct read of DISR occurring in program order after the ESB instruction.

If EL2 is not implemented, then VDISR is RES0 from Monitor mode when SCR.NS == 1.

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1100	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T12 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T12 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    return VDISR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        return VDISR;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1100	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T12 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T12 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    VDISR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        VDISR = R[t];
```

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1100	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T12 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T12 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.AMO == '1' then
        return VDISR_EL2;
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.AMO == '1' then
        return VDISR;
    else
        return DISR;
elsif PSTATE.EL == EL2 then
    return DISR;
elsif PSTATE.EL == EL3 then
    return DISR;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1100	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T12 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T12 == '1' then
        AArch32.TakeHypTrapException(0x03);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.AMO == '1' then
        VDISR_EL2 = R[t];
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.AMO == '1' then
        VDISR = R[t];
    else
        DISR = R[t];
elsif PSTATE.EL == EL2 then
    DISR = R[t];
elsif PSTATE.EL == EL3 then
    DISR = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.7 Generic Timer registers

This section lists the Generic Timer registers in AArch32.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.7.1 CNTFRQ, Counter-timer Frequency register

The CNTFRQ characteristics are:

Purpose

This register is provided so that software can discover the frequency of the system counter. It must be programmed with this value as part of system initialization. The value of the register is not interpreted by hardware.

Configurations

AArch32 System register CNTFRQ[31:0] is architecturally mapped to AArch64 System register CNTFRQ_EL0[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to CNTFRQ are UNKNOWN.

Attributes

CNTFRQ is a 32-bit register.

Field descriptions

The CNTFRQ bit assignments are:

31 … 0
Clock frequency

Bits [31:0]

Clock frequency. Indicates the system counter clock frequency, in Hz.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTFRQ

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') &&
CNTKCTL_EL1.<EL0PCTEN,EL0VCTEN> == '00' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0PCTEN == '0' && CNTKCTL.PL0VCTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' &&
CNTHCTL_EL2.<EL0PCTEN,EL0VCTEN> == '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        return CNTFRQ;
elsif PSTATE.EL == EL1 then
    return CNTFRQ;
elsif PSTATE.EL == EL2 then
    return CNTFRQ;
elsif PSTATE.EL == EL3 then
    return CNTFRQ;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0000	0b000

```
if IsHighestEL(PSTATE.EL) then
    CNTFRQ = R[t];
else
    UNDEFINED;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.7.2 CNTHCTL, Counter-timer Hyp Control register

The CNTHCTL characteristics are:

Purpose

Controls the generation of an event stream from the physical counter, and access from Non-secure EL1 modes to the physical counter and the Non-secure EL1 physical timer.

Configurations

AArch32 System register CNTHCTL[31:0] is architecturally mapped to AArch64 System register CNTHCTL_EL2[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to CNTHCTL are UNKNOWN.

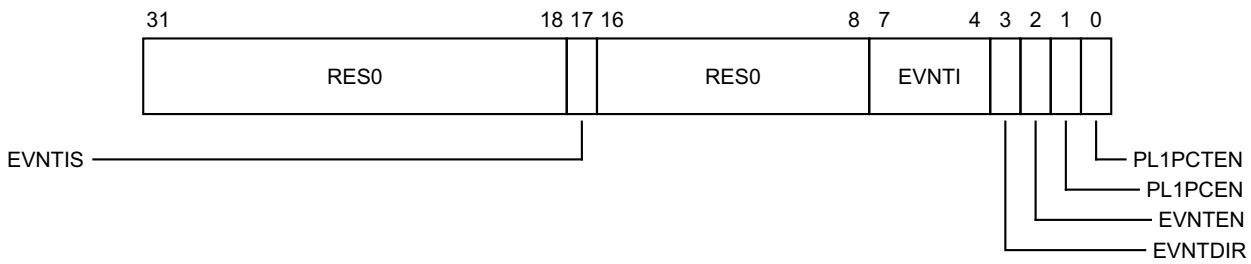
If EL2 is not implemented, this register is RES0 from EL3.

Attributes

CNTHCTL is a 32-bit register.

Field descriptions

The CNTHCTL bit assignments are:



Bits [31:18]

Reserved, RES0.

EVNTIS, bit [17]

When ARMv8.6-ECV is implemented:

Controls the scale of the generation of the event stream.

`0b0` The CNTHCTL.EVNTI field applies to CNTPCT[15:0].

`0b1` The CNTHCTL.EVNTI field applies to CNTPCT[23:8].

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [16:8]

Reserved, RES0.

EVNTI, bits [7:4]

Selects which bit (0 to 15) of the counter register CNTPCT is the trigger for the event stream generated from that counter, when that stream is enabled.

If ARMv8.6-ECV is implemented, and CNTHCTL.EVNTIS is 1, this selects which bit (8 to 23) of the counter register CNTPCT is the trigger.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

EVNTDIR, bit [3]

Controls which transition of the counter register CNTPCT trigger bit, defined by EVNTI, generates an event when the event stream is enabled:

`0b0` A 0 to 1 transition of the trigger bit triggers an event.

`0b1` A 1 to 0 transition of the trigger bit triggers an event.

This field resets to an architecturally UNKNOWN value.

EVNTEN, bit [2]

Enables the generation of an event stream from the counter register CNTPCT:

`0b0` Disables the event stream.

`0b1` Enables the event stream.

This field resets to an architecturally UNKNOWN value.

PL1PCEN, bit [1]

Traps Non-secure EL0 and EL1 accesses to the physical timer registers to Hyp mode.

`0b0` Non-secure EL0 and EL1 accesses to the CNTP_CTL, CNTP_CVAL, and CNTP_TVAL are trapped to Hyp mode, unless the it is trapped by CNTKCTL.PL0PTEN.

`0b1` This control does not cause any instructions to be trapped.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 1 other than for the purpose of a direct read.

This field resets to an architecturally UNKNOWN value.

PL1PCTEN, bit [0]

Traps Non-secure EL0 and EL1 accesses to the physical counter register to Hyp mode.

`0b0` Non-secure EL0 and EL1 accesses to the CNTPCT are trapped to Hyp mode, unless it is trapped by CNTKCTL.PL0PCTEN.

`0b1` This control does not cause any instructions to be trapped.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 1 other than for the purpose of a direct read.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTHCTL

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1110	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    return CNTHCTL;
elsif PSTATE.EL == EL3 then
    return CNTHCTL;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1110	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    CNTHCTL = R[t];
elsif PSTATE.EL == EL3 then
    CNTHCTL = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.7.3 CNTHP_CTL, Counter-timer Hyp Physical Timer Control register

The CNTHP_CTL characteristics are:

Purpose

Control register for the Hyp mode physical timer.

Configurations

AArch32 System register CNTHP_CTL[31:0] is architecturally mapped to AArch64 System register CNTHP_CTL_EL2[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to CNTHP_CTL are UNKNOWN.

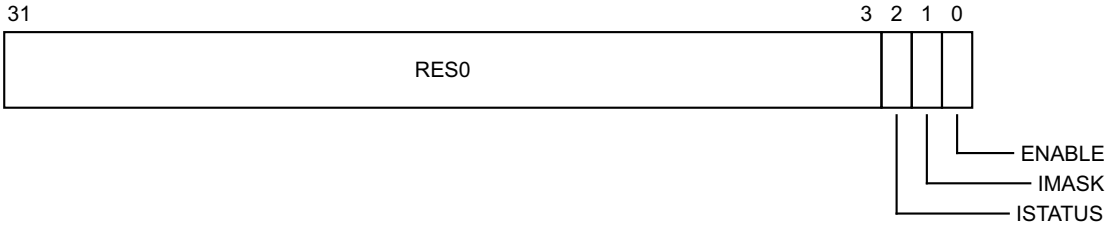
If EL2 is not implemented, this register is RES0 from EL3.

Attributes

CNTHP_CTL is a 32-bit register.

Field descriptions

The CNTHP_CTL bit assignments are:



Bits [31:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

`0b0` Timer condition is not met.

`0b1` Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

For more information see *Operation of the CompareValue views of the timers* on page D11-2847 and *Operation of the TimerValue views of the timers* on page D11-2848.

This bit is read-only.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

`0b0` Timer interrupt is not masked by the IMASK bit.

`0b1` Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

In a system where the PE resets into EL2 or EL3, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ENABLE, bit [0]

Enables the timer. Permitted values are:

`0b0` Timer disabled.

`0b1` Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTHP_TVAL continues to count down.

Note

Disabling the output signal might be a power-saving option.

In a system where the PE resets into EL2 or EL3, this field resets to `0`.

Accessing the CNTHP_CTL

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    return CNTHP_CTL;
elsif PSTATE.EL == EL3 then
    return CNTHP_CTL;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    CNTHP_CTL = R[t];
elsif PSTATE.EL == EL3 then
    CNTHP_CTL = R[t];
```

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0PTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHPS_CTL_EL2;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHP_CTL_EL2;
    else
        return CNTP_CTL;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return CNTP_CTL_NS;
    else
        return CNTP_CTL;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return CNTP_CTL_NS;
    else
        return CNTP_CTL;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return CNTP_CTL_S;
    else
        return CNTP_CTL_NS;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0PTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHPS_CTL_EL2 = R[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHP_CTL_EL2 = R[t];
    else
        CNTP_CTL = R[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        CNTP_CTL_NS = R[t];
    else
        CNTP_CTL = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        CNTP_CTL_NS = R[t];
    else
        CNTP_CTL = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        CNTP_CTL_S = R[t];
    else
        CNTP_CTL_NS = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.7.4 CNTHP_CVAL, Counter-timer Hyp Physical CompareValue register

The CNTHP_CVAL characteristics are:

Purpose

Holds the compare value for the Hyp mode physical timer.

Configurations

AArch32 System register CNTHP_CVAL[63:0] is architecturally mapped to AArch64 System register CNTHP_CVAL_EL2[63:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to CNTHP_CVAL are UNKNOWN.

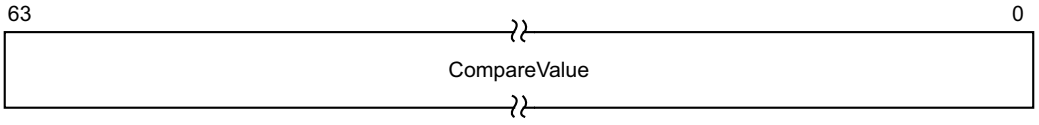
If EL2 is not implemented, this register is RES0 from EL3.

Attributes

CNTHP_CVAL is a 64-bit register.

Field descriptions

The CNTHP_CVAL bit assignments are:



CompareValue, bits [63:0]

Holds the EL2 physical timer CompareValue.

When CNTHP_CTL.ENABLE is 1, the timer condition is met when (CNTPCT - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTHP_CTL.ISTATUS is set to 1.
- If CNTHP_CTL.IMASK is 0, an interrupt is generated.

When CNTHP_CTL.ENABLE is 0, the timer condition is not met, but CNTPCT continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTHP_CVAL

Accesses to this register use the following encodings in the System instruction encoding space:

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1110	0b0110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    return CNTHP_CVAL;
elsif PSTATE.EL == EL3 then
    return CNTHP_CVAL;
```

MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1110	0b0110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    CNTHP_CVAL = R[t2]:R[t];
elsif PSTATE.EL == EL3 then
    CNTHP_CVAL = R[t2]:R[t];
```

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1110	0b0010

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x04);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0PTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHPS_CVAL_EL2;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHP_CVAL_EL2;
    else
        return CNTP_CVAL;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch32.TakeHypTrapException(0x04);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return CNTP_CVAL_NS;
    else
        return CNTP_CVAL;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return CNTP_CVAL_NS;
    else
        return CNTP_CVAL;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return CNTP_CVAL_S;
    else
        return CNTP_CVAL_NS;
```

MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1110	0b0010

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x04);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0PTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHPS_CVAL_EL2 = R[t2]:R[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHP_CVAL_EL2 = R[t2]:R[t];
    else
        CNTP_CVAL = R[t2]:R[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x04);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        CNTP_CVAL_NS = R[t2]:R[t];
    else
        CNTP_CVAL = R[t2]:R[t];
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        CNTP_CVAL_NS = R[t2]:R[t];
    else
        CNTP_CVAL = R[t2]:R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        CNTP_CVAL_S = R[t2]:R[t];
    else
        CNTP_CVAL_NS = R[t2]:R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.7.5 CNTHP_TVAL, Counter-timer Hyp Physical Timer TimerValue register

The CNTHP_TVAL characteristics are:

Purpose

Holds the timer value for the Hyp mode physical timer.

Configurations

AArch32 System register CNTHP_TVAL[31:0] is architecturally mapped to AArch64 System register CNTHP_TVAL_EL2[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to CNTHP_TVAL are UNKNOWN.

If EL2 is not implemented, this register is RES0 from EL3.

Attributes

CNTHP_TVAL is a 32-bit register.

Field descriptions

The CNTHP_TVAL bit assignments are:

31	0
TimerValue	

TimerValue, bits [31:0]

The TimerValue view of the EL2 physical timer.

On a read of this register:

- If CNTHP_CTL.ENABLE is 0, the value returned is UNKNOWN.
- If CNTHP_CTL.ENABLE is 1, the value returned is (CNTHP_CVAL - CNTPCT).

On a write of this register, CNTHP_CVAL is set to (CNTPCT + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTHP_CTL.ENABLE is 1, the timer condition is met when (CNTPCT - CNTHP_CVAL) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTHP_CTL.ISTATUS is set to 1.
- If CNTHP_CTL.IMASK is 0, an interrupt is generated.

When CNTHP_CTL.ENABLE is 0, the timer condition is not met, but CNTPCT continues to count, so the TimerValue view appears to continue to count down.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTHP_TVAL

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1110	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    return CNTHP_TVAL;
elsif PSTATE.EL == EL3 then
    return CNTHP_TVAL;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1110	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    CNTHP_TVAL = R[t];
elsif PSTATE.EL == EL3 then
    CNTHP_TVAL = R[t];
```

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0010	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0PTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x03);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHPS_TVAL_EL2;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHP_TVAL_EL2;
    else
        return CNTP_TVAL;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return CNTP_TVAL_NS;
    else
        return CNTP_TVAL;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return CNTP_TVAL_NS;
    else
        return CNTP_TVAL;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return CNTP_TVAL_S;
    else
        return CNTP_TVAL_NS;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0010	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0PTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHPS_TVAL_EL2 = R[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHP_TVAL_EL2 = R[t];
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        CNTP_TVAL = R[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        CNTP_TVAL_NS = R[t];
    else
        CNTP_TVAL = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        CNTP_TVAL_NS = R[t];
    else
        CNTP_TVAL = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        CNTP_TVAL_S = R[t];
    else
        CNTP_TVAL_NS = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.7.6 CNTHPS_CTL, Counter-timer Secure Physical Timer Control Register (EL2)

The CNTHPS_CTL characteristics are:

Purpose

Provides AArch32 access to the Secure EL2 physical timer.

Note

The Secure EL2 timer is implemented by ARMv8.4-SecEL2. It is only accessible from AArch32 state when EL2 is using AArch64 and the value of SCR_EL3.{EEL2, NS} is {1, 0}.

Configurations

AArch32 System register CNTHPS_CTL[31:0] is architecturally mapped to AArch64 System register CNTHPS_CTL_EL2[31:0].

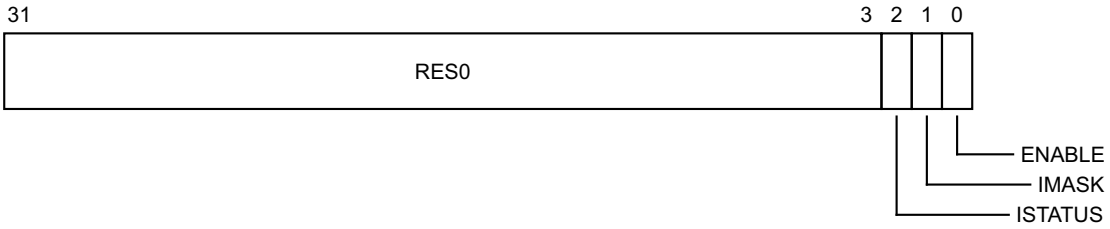
This register is present only when AArch32 is supported at any Exception level and ARMv8.4-SecEL2 is implemented. Otherwise, direct accesses to CNTHPS_CTL are UNDEFINED.

Attributes

CNTHPS_CTL is a 32-bit register.

Field descriptions

The CNTHPS_CTL bit assignments are:



Bits [31:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

`0b0` Timer condition is not met.

`0b1` Timer condition is met.

When the value of the CNTHPS_CTL.ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the CNTHPS_CTL.ENABLE bit is 0, the ISTATUS field is UNKNOWN.

For more information see *Operation of the CompareValue views of the timers* on page D11-2847 and *Operation of the TimerValue views of the timers* on page D11-2848.

This bit is read-only.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

`0b0` Timer interrupt is not masked by the IMASK bit.

`0b1` Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

ENABLE, bit [0]

Enables the timer. Permitted values are:

`0b0` Timer disabled.

`0b1` Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTHPS_TVAL_EL2 continues to count down.

Note

Disabling the output signal might be a power-saving option.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTHPS_CTL

This register is accessed using the encoding for CNTP_CTL.

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0PTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHPS_CTL_EL2;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHP_CTL_EL2;
    else
        return CNTP_CTL;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return CNTP_CTL_NS;
    else
        return CNTP_CTL;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return CNTP_CTL_NS;
    else
        return CNTP_CTL;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return CNTP_CTL_S;
    else
        return CNTP_CTL_NS;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0PTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHPS_CTL_EL2 = R[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHP_CTL_EL2 = R[t];
    else
        CNTP_CTL = R[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        CNTP_CTL_NS = R[t];
    else
        CNTP_CTL = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        CNTP_CTL_NS = R[t];
    else
        CNTP_CTL = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        CNTP_CTL_S = R[t];
    else
        CNTP_CTL_NS = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.7.7 CNTHPS_CVAL, Counter-timer Secure Physical Timer CompareValue Register (EL2)

The CNTHPS_CVAL characteristics are:

Purpose

Provides AArch32 access to the compare value for the Secure EL2 physical timer.

Note

The Secure EL2 timer is implemented by ARMv8.4-SecEL2. It is only accessible from AArch32 state when EL2 is using AArch64 and the value of SCR_EL3.{EEL2, NS} is {1, 0}.

Configurations

AArch32 System register CNTHPS_CVAL[63:0] is architecturally mapped to AArch64 System register CNTHPS_CVAL_EL2[63:0].

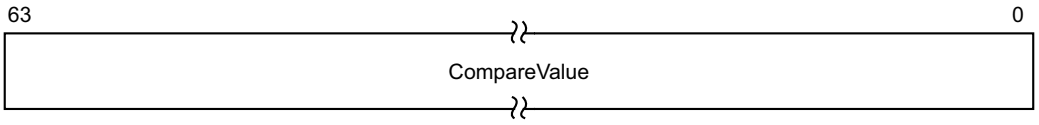
This register is present only when AArch32 is supported at any Exception level and ARMv8.4-SecEL2 is implemented. Otherwise, direct accesses to CNTHPS_CVAL are UNDEFINED.

Attributes

CNTHPS_CVAL is a 64-bit register.

Field descriptions

The CNTHPS_CVAL bit assignments are:



CompareValue, bits [63:0]

Holds the EL2 physical timer CompareValue.

When CNTHPS_CTL_EL2.ENABLE is 1, the timer condition is met when (CNTPCT_EL0 - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTHPS_CTL_EL2.ISTATUS is set to 1.
- If CNTHPS_CTL_EL2.IMASK is 0, an interrupt is generated.

When CNTHPS_CTL_EL2.ENABLE is 0, the timer condition is not met, but CNTPCT_EL0 continues to count

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTHPS_CVAL

This register is accessed using the encoding for CNTP_CVAL.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1110	0b0010

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x04);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0PTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHPS_CVAL_EL2;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHP_CVAL_EL2;
    else
        return CNTP_CVAL;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x04);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return CNTP_CVAL_NS;
    else
        return CNTP_CVAL;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return CNTP_CVAL_NS;
    else
        return CNTP_CVAL;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return CNTP_CVAL_S;
    else
        return CNTP_CVAL_NS;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1110	0b0010

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x04);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0PTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHPS_CVAL_EL2 = R[t2]:R[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHP_CVAL_EL2 = R[t2]:R[t];
    else
        CNTP_CVAL = R[t2]:R[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x04);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        CNTP_CVAL_NS = R[t2]:R[t];
    else
        CNTP_CVAL = R[t2]:R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        CNTP_CVAL_NS = R[t2]:R[t];
    else
        CNTP_CVAL = R[t2]:R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        CNTP_CVAL_S = R[t2]:R[t];
    else
        CNTP_CVAL_NS = R[t2]:R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.7.8 CNTHPS_TVAL, Counter-timer Secure Physical Timer TimerValue Register (EL2)

The CNTHPS_TVAL characteristics are:

Purpose

Provides AArch32 access to the timer value for the Secure EL2 physical timer.

Note

The Secure EL2 timer is implemented by ARMv8.4-SecEL2. It is only accessible from AArch32 state when EL2 is using AArch64 and the value of SCR_EL3.{EEL2, NS} is {1, 0}.

Configurations

AArch32 System register CNTHPS_TVAL[31:0] is architecturally mapped to AArch64 System register CNTHPS_TVAL_EL2[31:0].

This register is present only when AArch32 is supported at any Exception level and ARMv8.4-SecEL2 is implemented. Otherwise, direct accesses to CNTHPS_TVAL are UNDEFINED.

Attributes

CNTHPS_TVAL is a 32-bit register.

Field descriptions

The CNTHPS_TVAL bit assignments are:

31 … 0
TimerValue

TimerValue, bits [31:0]

The TimerValue view of the EL2 physical timer.

On a read of this register:

- If CNTHPS_CTL_EL2.ENABLE is 0, the value returned is UNKNOWN.
- If CNTHPS_CTL_EL2.ENABLE is 1, the value returned is (CNTHPS_CVAL_EL2 - CNTPCT_EL0).

On a write of this register, CNTHPS_CVAL_EL2 is set to (CNTPCT_EL0 + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTHPS_CTL_EL2.ENABLE is 1, the timer condition is met when (CNTPCT_EL0 - CNTHPS_CVAL_EL2) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTHPS_CTL_EL2.ISTATUS is set to 1.
- If CNTHPS_CTL_EL2.IMASK is 0, an interrupt is generated.

When CNTHPS_CTL_EL2.ENABLE is 0, the timer condition is not met, but CNTPCT_EL0 continues to count, so the TimerValue view appears to continue to count down.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTHPS_TVAL

This register is accessed using the encoding for CNTP_TVAL.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0010	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0PTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHPS_TVAL_EL2;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHP_TVAL_EL2;
    else
        return CNTP_TVAL;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return CNTP_TVAL_NS;
    else
        return CNTP_TVAL;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return CNTP_TVAL_NS;
    else
        return CNTP_TVAL;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return CNTP_TVAL_S;
    else
        return CNTP_TVAL_NS;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0010	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0PTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHPS_TVAL_EL2 = R[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHP_TVAL_EL2 = R[t];
    else
        CNTP_TVAL = R[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        CNTP_TVAL_NS = R[t];
    else
        CNTP_TVAL = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        CNTP_TVAL_NS = R[t];
    else
        CNTP_TVAL = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        CNTP_TVAL_S = R[t];
    else
        CNTP_TVAL_NS = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.7.9 CNTHV_CTL, Counter-timer Virtual Timer Control register (EL2)

The CNTHV_CTL characteristics are:

Purpose

Provides AArch32 access to the control register for the EL2 virtual timer.

Note

The EL2 virtual timer is implemented by ARMv8.1-VHE. It is only accessible from AArch32 state when EL0 is using AArch32, EL2 is using AArch64, and the value of HCR_EL2.{E2H, TGE} is {1, 1}.

Configurations

AArch32 System register CNTHV_CTL[31:0] is architecturally mapped to AArch64 System register CNTHV_CTL_EL2[31:0].

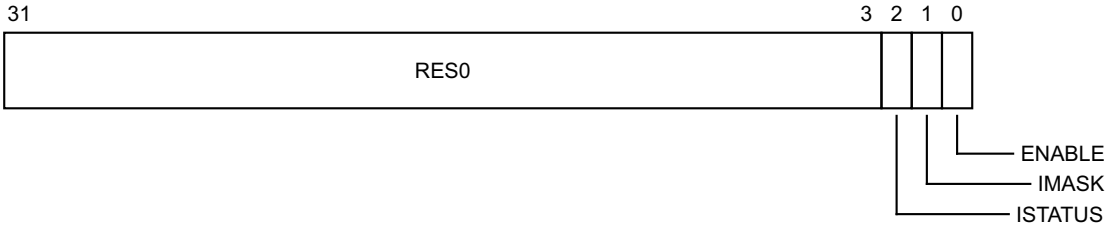
This register is present only when AArch32 is supported at any Exception level and ARMv8.1-VHE is implemented. Otherwise, direct accesses to CNTHV_CTL are UNDEFINED.

Attributes

CNTHV_CTL is a 32-bit register.

Field descriptions

The CNTHV_CTL bit assignments are:



Bits [31:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

`0b0` Timer condition is not met.

`0b1` Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

For more information see *Operation of the CompareValue views of the timers* on page D11-2847 and *Operation of the TimerValue views of the timers* on page D11-2848.

This bit is read-only.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

`0b0` Timer interrupt is not masked by the IMASK bit.

`0b1` Timer interrupt is masked by the IMASK bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For more information, see the description of the ISTATUS bit.

ENABLE, bit [0]

Enables the timer. Permitted values are:

`0b0` Timer disabled.

`0b1` Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTHV_TVAL continues to count down.

Note

Disabling the output signal might be a power-saving option.

Accessing the CNTHV_CTL

This register is accessed using the encoding for CNTV_CTL.

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0011	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0VTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHVS_CTL_EL2;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHV_CTL_EL2;
    else
        return CNTV_CTL;
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        return CNTV_CTL;
elsif PSTATE.EL == EL2 then
    return CNTV_CTL;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    return CNTV_CTL;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0011	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0VTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHVS_CTL_EL2 = R[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHV_CTL_EL2 = R[t];
    else
        CNTV_CTL = R[t];
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        CNTV_CTL = R[t];
elsif PSTATE.EL == EL2 then
    CNTV_CTL = R[t];
elsif PSTATE.EL == EL3 then
    CNTV_CTL = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.7.10 CNTHV_CVAL, Counter-timer Virtual Timer CompareValue register (EL2)

The CNTHV_CVAL characteristics are:

Purpose

Provides AArch32 access to the compare value for the EL2 virtual timer.

Note

The EL2 virtual timer is implemented by ARMv8.1-VHE. It is only accessible from AArch32 state when EL0 is using AArch32, EL2 is using AArch64, and the value of HCR_EL2.{E2H, TGE} is {1, 1}.

Configurations

AArch32 System register CNTHV_CVAL[63:0] is architecturally mapped to AArch64 System register CNTHV_CVAL_EL2[63:0].

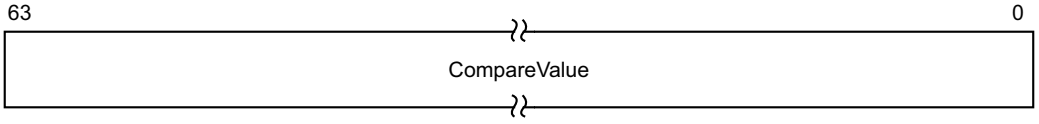
This register is present only when ARMv8.1-VHE is implemented. Otherwise, direct accesses to CNTHV_CVAL are UNDEFINED.

Attributes

CNTHV_CVAL is a 64-bit register.

Field descriptions

The CNTHV_CVAL bit assignments are:



CompareValue, bits [63:0]

Holds the EL2 virtual timer CompareValue.

When CNTHV_CTL.ENABLE is 1, the timer condition is met when (CNTVCT - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTHV_CTL.ISTATUS is set to 1.
- If CNTHV_CTL.IMASK is 0, an interrupt is generated.

When CNTHV_CTL.ENABLE is 0, the timer condition is not met, but CNTVCT continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

Accessing the CNTHV_CVAL

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1110	0b0011

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x04);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0VTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHVS_CVAL_EL2;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHV_CVAL_EL2;
    else
        return CNTV_CVAL;
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    else
        return CNTV_CVAL;
elsif PSTATE.EL == EL2 then
    return CNTV_CVAL;
elsif PSTATE.EL == EL3 then
    return CNTV_CVAL;
```

MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1110	0b0011

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x04);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0VTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHVS_CVAL_EL2 = R[t2]:R[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHV_CVAL_EL2 = R[t2]:R[t];
    else
        CNTV_CVAL = R[t2]:R[t];
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    else
        CNTV_CVAL = R[t2]:R[t];
elsif PSTATE.EL == EL2 then
    CNTV_CVAL = R[t2]:R[t];
elsif PSTATE.EL == EL3 then
    CNTV_CVAL = R[t2]:R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.7.11 CNTHV_TVAL, Counter-timer Virtual Timer TimerValue register (EL2)

The CNTHV_TVAL characteristics are:

Purpose

Provides AArch32 access to the timer value for the EL2 virtual timer.

Note

The EL2 virtual timer is implemented by ARMv8.1-VHE. It is only accessible from AArch32 state when EL0 is using AArch32, EL2 is using AArch64, and the value of HCR_EL2.{E2H, TGE} is {1, 1}.

Configurations

AArch32 System register CNTHV_TVAL[31:0] is architecturally mapped to AArch64 System register CNTHV_TVAL_EL2[31:0].

This register is present only when AArch32 is supported at any Exception level and ARMv8.1-VHE is implemented. Otherwise, direct accesses to CNTHV_TVAL are UNDEFINED.

Attributes

CNTHV_TVAL is a 32-bit register.

Field descriptions

The CNTHV_TVAL bit assignments are:

31 … 0
TimerValue

TimerValue, bits [31:0]

The TimerValue view of the EL2 virtual timer.

On a read of this register:

- If CNTHV_CTL.ENABLE is 0, the value returned is UNKNOWN.
- If CNTHV_CTL.ENABLE is 1, the value returned is (CNTHV_CVAL - CNTVCT).

On a write of this register, CNTHV_CVAL is set to (CNTVCT + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTHV_CTL.ENABLE is 1, the timer condition is met when (CNTVCT - CNTHV_CVAL) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTHV_CTL.ISTATUS is set to 1.
- If CNTHV_CTL.IMASK is 0, an interrupt is generated.

When CNTHV_CTL.ENABLE is 0, the timer condition is not met, but CNTVCT continues to count, so the TimerValue view appears to continue to count down.

Accessing the CNTHV_TVAL

This register is accessed using the encoding for CNTV_TVAL.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0011	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0VTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHVS_TVAL_EL2;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHV_TVAL_EL2;
    else
        return CNTV_TVAL;
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        return CNTV_TVAL;
elsif PSTATE.EL == EL2 then
    return CNTV_TVAL;
elsif PSTATE.EL == EL3 then
    return CNTV_TVAL;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0011	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0VTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHVS_TVAL_EL2 = R[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHV_TVAL_EL2 = R[t];
    else
        CNTV_TVAL = R[t];
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        CNTV_TVAL = R[t];
elsif PSTATE.EL == EL2 then
    CNTV_TVAL = R[t];
elsif PSTATE.EL == EL3 then
    CNTV_TVAL = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.7.12 CNTHVS_CTL, Counter-timer Secure Virtual Timer Control Register (EL2)

The CNTHVS_CTL characteristics are:

Purpose

Provides AArch32 access to the Secure EL2 virtual timer.

Note

The Secure EL2 timer is implemented by ARMv8.4-SecEL2. It is only accessible from AArch32 state when EL2 is using AArch64 and the value of SCR_EL3.{EEL2, NS} is {1, 0}.

Configurations

AArch32 System register CNTHVS_CTL[31:0] is architecturally mapped to AArch64 System register CNTHVS_CTL_EL2[31:0].

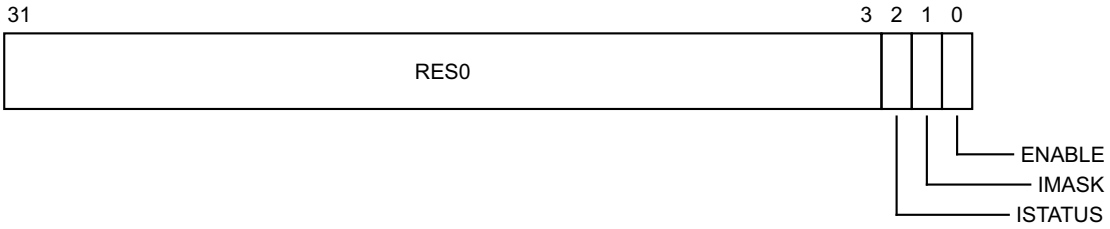
This register is present only when AArch32 is supported at any Exception level and ARMv8.4-SecEL2 is implemented. Otherwise, direct accesses to CNTHVS_CTL are UNDEFINED.

Attributes

CNTHVS_CTL is a 32-bit register.

Field descriptions

The CNTHVS_CTL bit assignments are:



Bits [31:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

`0b0` Timer condition is not met.

`0b1` Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

For more information see *Operation of the CompareValue views of the timers* on page D11-2847 and *Operation of the TimerValue views of the timers* on page D11-2848.

This bit is read-only.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

`0b0` Timer interrupt is not masked by the IMASK bit.

`0b1` Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ENABLE, bit [0]

Enables the timer. Permitted values are:

`0b0` Timer disabled.

`0b1` Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTHVS_TVAL continues to count down.

Note

Disabling the output signal might be a power-saving option.

Accessing the CNTHVS_CTL

This register is accessed using the encoding for CNTV_CTL.

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0011	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0VTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHVS_CTL_EL2;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHV_CTL_EL2;
    else
        return CNTV_CTL;
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        return CNTV_CTL;
elsif PSTATE.EL == EL2 then
    return CNTV_CTL;
elsif PSTATE.EL == EL3 then
    return CNTV_CTL;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0011	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0VTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHVS_CTL_EL2 = R[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHV_CTL_EL2 = R[t];
    else
        CNTV_CTL = R[t];
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        CNTV_CTL = R[t];
elsif PSTATE.EL == EL2 then
    CNTV_CTL = R[t];
elsif PSTATE.EL == EL3 then
    CNTV_CTL = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.7.13 CNTHVS_CVAL, Counter-timer Secure Virtual Timer CompareValue Register (EL2)

The CNTHVS_CVAL characteristics are:

Purpose

Provides AArch32 access to the compare value for the Secure EL2 virtual timer.

Note

The Secure EL2 timer is implemented by ARMv8.4-SecEL2. It is only accessible from AArch32 state when EL2 is using AArch64 and the value of SCR_EL3.{EEL2, NS} is {1, 0}.

Configurations

AArch32 System register CNTHVS_CVAL[63:0] is architecturally mapped to AArch64 System register CNTHVS_CVAL_EL2[63:0].

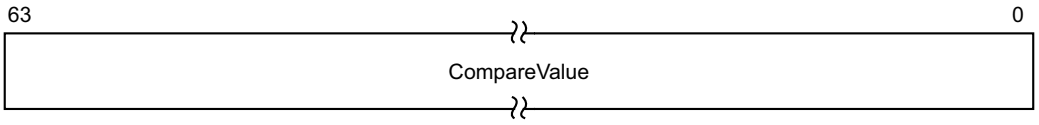
This register is present only when AArch32 is supported at any Exception level and ARMv8.4-SecEL2 is implemented. Otherwise, direct accesses to CNTHVS_CVAL are UNDEFINED.

Attributes

CNTHVS_CVAL is a 64-bit register.

Field descriptions

The CNTHVS_CVAL bit assignments are:



CompareValue, bits [63:0]

Holds the EL2 virtual timer CompareValue.

When CNTHVS_CTL.ENABLE is 1, the timer condition is met when (CNTVCT - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTHVS_CTL.ISTATUS is set to 1.
- If CNTHVS_CTL.IMASK is 0, an interrupt is generated.

When CNTHVS_CTL.ENABLE is 0, the timer condition is not met, but CNTVCT continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

Accessing the CNTHVS_CVAL

This register is accessed using the encoding for CNTV_CVAL.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1110	0b0011

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x04);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0VTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHVS_CVAL_EL2;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHV_CVAL_EL2;
    else
        return CNTV_CVAL;
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    else
        return CNTV_CVAL;
elsif PSTATE.EL == EL2 then
    return CNTV_CVAL;
elsif PSTATE.EL == EL3 then
    return CNTV_CVAL;
```

MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1110	0b0011

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x04);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0VTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHVS_CVAL_EL2 = R[t2]:R[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHV_CVAL_EL2 = R[t2]:R[t];
    else
        CNTV_CVAL = R[t2]:R[t];
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    else
        CNTV_CVAL = R[t2]:R[t];
elsif PSTATE.EL == EL2 then
    CNTV_CVAL = R[t2]:R[t];
elsif PSTATE.EL == EL3 then
    CNTV_CVAL = R[t2]:R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.7.14 CNTHVS_TVAL, Counter-timer Secure Virtual Timer TimerValue Register (EL2)

The CNTHVS_TVAL characteristics are:

Purpose

Provides AArch32 access to the timer value for the Secure EL2 virtual timer.

Note

The Secure EL2 timer is implemented by ARMv8.4-SecEL2. It is only accessible from AArch32 state when EL2 is using AArch64 and the value of SCR_EL3.{EEL2, NS} is {1, 0}.

Configurations

AArch32 System register CNTHVS_TVAL[31:0] is architecturally mapped to AArch64 System register CNTHVS_TVAL_EL2[31:0].

This register is present only when AArch32 is supported at any Exception level and ARMv8.4-SecEL2 is implemented. Otherwise, direct accesses to CNTHVS_TVAL are UNDEFINED.

Attributes

CNTHVS_TVAL is a 32-bit register.

Field descriptions

The CNTHVS_TVAL bit assignments are:

31 … 0
TimerValue

TimerValue, bits [31:0]

The TimerValue view of the EL2 virtual timer.

On a read of this register:

- If CNTHVS_CTL.ENABLE is 0, the value returned is UNKNOWN.
- If CNTHVS_CTL.ENABLE is 1, the value returned is (CNTHVS_CVAL - CNTVCT).

On a write of this register, CNTHVS_CVAL is set to (CNTVCT + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTHVS_CTL.ENABLE is 1, the timer condition is met when (CNTVCT - CNTHVS_CVAL) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTHVS_CTL.ISTATUS is set to 1.
- If CNTHVS_CTL.IMASK is 0, an interrupt is generated.

When CNTHVS_CTL.ENABLE is 0, the timer condition is not met, but CNTVCT continues to count, so the TimerValue view appears to continue to count down.

Accessing the CNTHVS_TVAL

This register is accessed using the encoding for CNTV_TVAL.

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0011	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0VTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHVS_TVAL_EL2;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHV_TVAL_EL2;
    else
        return CNTV_TVAL;
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        return CNTV_TVAL;
elsif PSTATE.EL == EL2 then
    return CNTV_TVAL;
elsif PSTATE.EL == EL3 then
    return CNTV_TVAL;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0011	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0VTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHVS_TVAL_EL2 = R[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHV_TVAL_EL2 = R[t];
    else
        CNTV_TVAL = R[t];
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        CNTV_TVAL = R[t];
elsif PSTATE.EL == EL2 then
    CNTV_TVAL = R[t];
elsif PSTATE.EL == EL3 then
    CNTV_TVAL = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.7.15 CNTKCTL, Counter-timer Kernel Control register

The CNTKCTL characteristics are:

Purpose

Controls the generation of an event stream from the virtual counter, and access from EL0 modes to the physical counter, virtual counter, EL1 physical timers, and the virtual timer.

Configurations

AArch32 System register CNTKCTL[31:0] is architecturally mapped to AArch64 System register CNTKCTL_EL1[31:0].

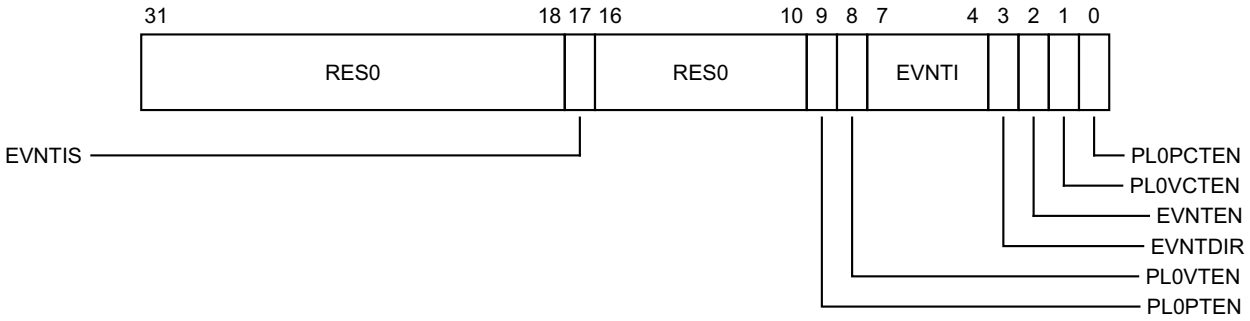
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to CNTKCTL are UNKNOWN.

Attributes

CNTKCTL is a 32-bit register.

Field descriptions

The CNTKCTL bit assignments are:



Bits [31:18]

Reserved, RES0.

EVNTIS, bit [17]

When ARMv8.6-ECV is implemented:

Controls the scale of the generation of the event stream.

`0b0` The CNTKCTL.EVNTI field applies to CNTVCT[15:0].

`0b1` The CNTKCTL.EVNTI field applies to CNTVCT[23:8].

This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [16:10]

Reserved, RES0.

PL0PTEN, bit [9]

Traps PL0 accesses to the physical timer registers to Undefined mode.

`0b0` PL0 accesses to the CNTP_CTL, CNTP_CVAL, and CNTP_TVAL registers are trapped to Undefined mode.

`0b1` This control does not cause any instructions to be trapped.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to an architecturally UNKNOWN value.

PL0VTEN, bit [8]

Traps PL0 accesses to the virtual timer registers to Undefined mode.

`0b0` PL0 accesses to the CNTV_CTL, CNTV_CVAL, and CNTV_TVAL registers are trapped to Undefined mode.

`0b1` This control does not cause any instructions to be trapped.

This field resets to an architecturally UNKNOWN value.

EVNTI, bits [7:4]

Selects which bit (0 to 15) of the counter register CNTVCT is the trigger for the event stream generated from that counter, when that stream is enabled.

If ARMv8.6-ECV is implemented, and CNTKCTL.EVNTIS is 1, this selects which bit (8 to 23) of the counter register CNTVCT is the trigger.

This field resets to an architecturally UNKNOWN value.

EVNTDIR, bit [3]

Controls which transition of the counter register CNTVCT trigger bit, defined by EVNTI, generates an event when the event stream is enabled:

`0b0` A 0 to 1 transition of the trigger bit triggers an event.

`0b1` A 1 to 0 transition of the trigger bit triggers an event.

This field resets to an architecturally UNKNOWN value.

EVNTEN, bit [2]

Enables the generation of an event stream from the counter register CNTVCT:

`0b0` Disables the event stream.

`0b1` Enables the event stream.

This field resets to an architecturally UNKNOWN value.

PL0VCTEN, bit [1]

Traps PL0 accesses to the frequency register and virtual counter register to Undefined mode.

`0b0` PL0 accesses to the CNTVCT are trapped to Undefined mode.

PL0 accesses to the CNTFRQ register are trapped to Undefined mode, if CNTKCTL.PL0PCTEN is also 0.

`0b1` This control does not cause any instructions to be trapped.

This field resets to an architecturally UNKNOWN value.

PL0PCTEN, bit [0]

Traps PL0 accesses to the frequency register and physical counter register to Undefined mode.

`0b0` PL0 accesses to the CNTPCT are trapped to Undefined mode.

PL0 accesses to the CNTFRQ register are trapped to Undefined mode, if CNTKCTL.PL0VCTEN is also 0.

`0b1` This control does not cause any instructions to be trapped.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTKCTL

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    return CNTKCTL;
elsif PSTATE.EL == EL2 then
    return CNTKCTL;
elsif PSTATE.EL == EL3 then
    return CNTKCTL;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    CNTKCTL = R[t];
elsif PSTATE.EL == EL2 then
    CNTKCTL = R[t];
elsif PSTATE.EL == EL3 then
    CNTKCTL = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.7.16 CNTP_CTL, Counter-timer Physical Timer Control register

The CNTP_CTL characteristics are:

Purpose

Control register for the EL1 physical timer.

Configurations

AArch32 System register CNTP_CTL[31:0] is architecturally mapped to AArch64 System register CNTP_CTL_EL0[31:0].

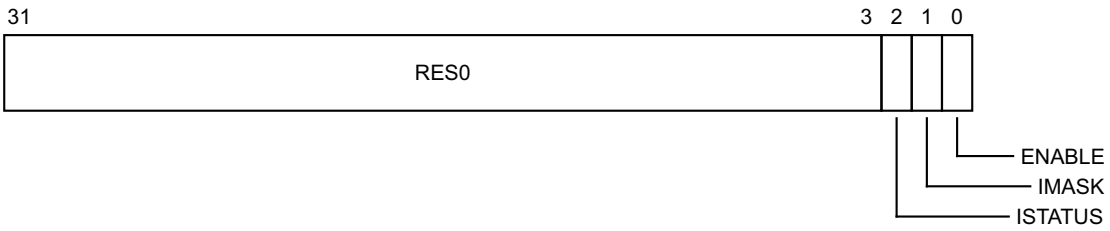
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to CNTP_CTL are UNKNOWN.

Attributes

CNTP_CTL is a 32-bit register.

Field descriptions

The CNTP_CTL bit assignments are:



Bits [31:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

`0b0` Timer condition is not met.

`0b1` Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

For more information see *Operation of the CompareValue views of the timers* on page D11-2847 and *Operation of the TimerValue views of the timers* on page D11-2848.

This bit is read-only.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

`0b0` Timer interrupt is not masked by the IMASK bit.

`0b1` Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ENABLE, bit [0]

Enables the timer. Permitted values are:

`0b0` Timer disabled.

`0b1` Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTP_TVAL continues to count down.

Note

Disabling the output signal might be a power-saving option.

This field resets to `0`.

Accessing the CNTP_CTL

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0PTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHPS_CTL_EL2;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHP_CTL_EL2;
    else
        return CNTP_CTL;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        return CNTP_CTL_NS;
    else
        return CNTP_CTL;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return CNTP_CTL_NS;
    else
        return CNTP_CTL;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return CNTP_CTL_S;
    else
        return CNTP_CTL_NS;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0PTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHPS_CTL_EL2 = R[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHP_CTL_EL2 = R[t];
    else
        CNTP_CTL = R[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        CNTP_CTL_NS = R[t];
    else
        CNTP_CTL = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        CNTP_CTL_NS = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        else
            CNTP_CTL = R[t];
    elsif PSTATE.EL == EL3 then
        if SCR.NS == '0' then
            CNTP_CTL_S = R[t];
        else
            CNTP_CTL_NS = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.7.17 CNTP_CVAL, Counter-timer Physical Timer CompareValue register

The CNTP_CVAL characteristics are:

Purpose

Holds the compare value for the EL1 physical timer.

Configurations

AArch32 System register CNTP_CVAL[63:0] is architecturally mapped to AArch64 System register CNTP_CVAL_EL0[63:0].

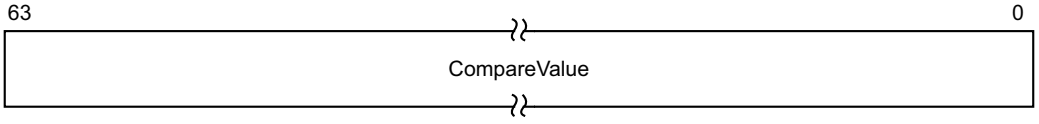
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to CNTP_CVAL are UNKNOWN.

Attributes

CNTP_CVAL is a 64-bit register.

Field descriptions

The CNTP_CVAL bit assignments are:



CompareValue, bits [63:0]

Holds the EL1 physical timer CompareValue.

When CNTP_CTL.ENABLE is 1, the timer condition is met when (CNTPCT - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTP_CTL.ISTATUS is set to 1.
- If CNTP_CTL.IMASK is 0, an interrupt is generated.

When CNTP_CTL.ENABLE is 0, the timer condition is not met, but CNTPCT continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTP_CVAL

Accesses to this register use the following encodings in the System instruction encoding space:

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1110	0b0010

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            AArch64.AArch32SystemAccessTrap(EL1, 0x04);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0PTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHPS_CVAL_EL2;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHP_CVAL_EL2;
    else
        return CNTP_CVAL;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x04);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return CNTP_CVAL_NS;
    else
        return CNTP_CVAL;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return CNTP_CVAL_NS;
    else
        return CNTP_CVAL;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return CNTP_CVAL_S;
    else
        return CNTP_CVAL_NS;
```

MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1110	0b0010

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x04);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0PTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHPS_CVAL_EL2 = R[t2]:R[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHP_CVAL_EL2 = R[t2]:R[t];
    else
        CNTP_CVAL = R[t2]:R[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x04);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        CNTP_CVAL_NS = R[t2]:R[t];
    else
        CNTP_CVAL = R[t2]:R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        CNTP_CVAL_NS = R[t2]:R[t];
    else
        CNTP_CVAL = R[t2]:R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        CNTP_CVAL_S = R[t2]:R[t];
    else
        CNTP_CVAL_NS = R[t2]:R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.7.18 CNTP_TVAL, Counter-timer Physical Timer TimerValue register

The CNTP_TVAL characteristics are:

Purpose

Holds the timer value for the EL1 physical timer.

Configurations

AArch32 System register CNTP_TVAL[31:0] is architecturally mapped to AArch64 System register CNTP_TVAL_EL0[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to CNTP_TVAL are UNKNOWN.

Attributes

CNTP_TVAL is a 32-bit register.

Field descriptions

The CNTP_TVAL bit assignments are:

31 … 0
TimerValue

TimerValue, bits [31:0]

The TimerValue view of the EL1 physical timer.

On a read of this register:

- If CNTP_CTL.ENABLE is 0, the value returned is UNKNOWN.
- If CNTP_CTL.ENABLE is 1, the value returned is (CNTP_CVAL - CNTPCT).

On a write of this register, CNTP_CVAL is set to (CNTPCT + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTP_CTL.ENABLE is 1, the timer condition is met when (CNTPCT - CNTP_CVAL) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTP_CTL.ISTATUS is set to 1.
- If CNTP_CTL.IMASK is 0, an interrupt is generated.

When CNTP_CTL.ENABLE is 0, the timer condition is not met, but CNTPCT continues to count, so the TimerValue view appears to continue to count down.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTP_TVAL

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0010	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0PTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHPS_TVAL_EL2;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHP_TVAL_EL2;
    else
        return CNTP_TVAL;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        return CNTP_TVAL_NS;
    else
        return CNTP_TVAL;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        return CNTP_TVAL_NS;
    else
        return CNTP_TVAL;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        return CNTP_TVAL_S;
    else
        return CNTP_TVAL_NS;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0010	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0PTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHPS_TVAL_EL2 = R[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHP_TVAL_EL2 = R[t];
    else
        CNTP_TVAL = R[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        CNTP_TVAL_NS = R[t];
    else
        CNTP_TVAL = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        CNTP_TVAL_NS = R[t];
    else
        CNTP_TVAL = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        CNTP_TVAL_S = R[t];
    else
        CNTP_TVAL_NS = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.7.19 CNTPCT, Counter-timer Physical Count register

The CNTPCT characteristics are:

Purpose

Holds the 64-bit physical count value.

Configurations

AArch32 System register CNTPCT[63:0] is architecturally mapped to AArch64 System register CNTPCT_EL0[63:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to CNTPCT are UNKNOWN.

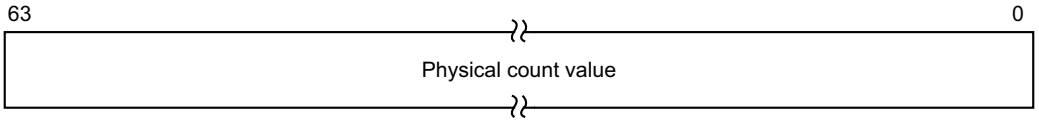
All reads to the CNTPCT occur in program order relative to reads to CNTPCTSS or CNTPCT.

Attributes

CNTPCT is a 64-bit register.

Field descriptions

The CNTPCT bit assignments are:



Bits [63:0]

Physical count value.

Accessing the CNTPCT

Accesses to this register use the following encodings in the System instruction encoding space:

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1110	0b0000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PCTEN ==
'0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x04);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0PCTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCTEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PCTEN ==
'0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PCTEN ==
'0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCTEN == '0' then
        AArch32.TakeHypTrapException(0x04);
    else
        return CNTPCT;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CNTHCTL_EL2.EL1PCTEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCTEN == '0' then
        AArch32.TakeHypTrapException(0x04);
    else
        return CNTPCT;
elsif PSTATE.EL == EL2 then
    return CNTPCT;
elsif PSTATE.EL == EL3 then
    return CNTPCT;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.7.20 CNTPCTSS, Counter-timer Self-Synchronized Physical Count register

The CNTPCTSS characteristics are:

Purpose

Holds the 64-bit physical count value.

Configurations

AArch32 System register CNTPCTSS[63:0] is architecturally mapped to AArch64 System register CNTPCTSS_EL0[63:0].

This register is present only when AArch32 is supported at any Exception level and ARMv8.6-ECV is implemented. Otherwise, direct accesses to CNTPCTSS are UNDEFINED.

All reads to the CNTPCTSS occur in program order relative to reads to CNTPCT or CNTPCTSS.

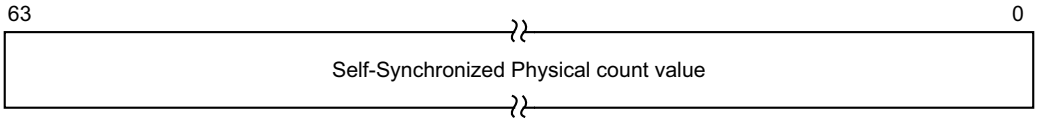
This register is a self-synchronised view of the CNTPCT counter, and cannot be read speculatively.

Attributes

CNTPCTSS is a 64-bit register.

Field descriptions

The CNTPCTSS bit assignments are:



Bits [63:0]

Self-Synchronized Physical count value.

Accessing the CNTPCTSS

Accesses to this register use the following encodings in the System instruction encoding space:

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1110	0b1000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PCTEN ==
'0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x04);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0PCTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCTEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PCTEN ==
'0' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PCTEN ==
'0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCTEN == '0' then
        AArch32.TakeHypTrapException(0x04);
    else
        return CNTPCTSS;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CNTHCTL_EL2.EL1PCTEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCTEN == '0' then
        AArch32.TakeHypTrapException(0x04);
    else
        return CNTPCTSS;
elsif PSTATE.EL == EL2 then
    return CNTPCTSS;
elsif PSTATE.EL == EL3 then
    return CNTPCTSS;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.7.21 CNTV_CTL, Counter-timer Virtual Timer Control register

The CNTV_CTL characteristics are:

Purpose

Control register for the virtual timer.

Configurations

AArch32 System register CNTV_CTL[31:0] is architecturally mapped to AArch64 System register CNTV_CTL_EL0[31:0].

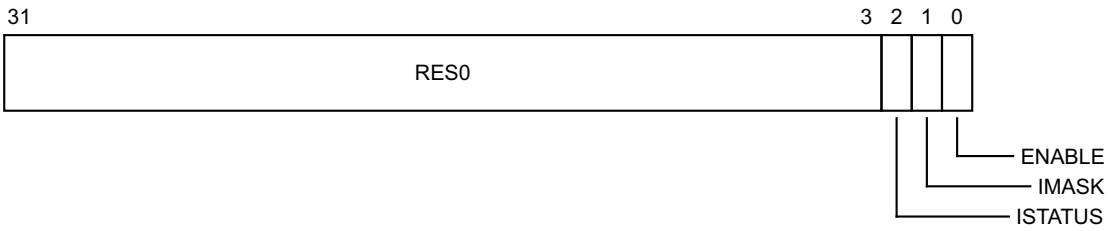
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to CNTV_CTL are UNKNOWN.

Attributes

CNTV_CTL is a 32-bit register.

Field descriptions

The CNTV_CTL bit assignments are:



Bits [31:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

`0b0` Timer condition is not met.

`0b1` Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

For more information see *Operation of the CompareValue views of the timers* on page D11-2847 and *Operation of the TimerValue views of the timers* on page D11-2848.

This bit is read-only.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

`0b0` Timer interrupt is not masked by the IMASK bit.

`0b1` Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ENABLE, bit [0]

Enables the timer. Permitted values are:

`0b0` Timer disabled.

`0b1` Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTV_TVAL continues to count down.

Note

Disabling the output signal might be a power-saving option.

This field resets to `0`.

Accessing the CNTV_CTL

Accesses to this register use the following encodings in the System instruction encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0011	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0VTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHVS_CTL_EL2;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHV_CTL_EL2;
    else
        return CNTV_CTL;
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        return CNTV_CTL;
elsif PSTATE.EL == EL2 then
    return CNTV_CTL;
elsif PSTATE.EL == EL3 then
    return CNTV_CTL;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0011	0b001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0VTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHVS_CTL_EL2 = R[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHV_CTL_EL2 = R[t];
    else
        CNTV_CTL = R[t];
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        CNTV_CTL = R[t];
elsif PSTATE.EL == EL2 then
    CNTV_CTL = R[t];
elsif PSTATE.EL == EL3 then
    CNTV_CTL = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.7.22 CNTV_CVAL, Counter-timer Virtual Timer CompareValue register

The CNTV_CVAL characteristics are:

Purpose

Holds the compare value for the virtual timer.

Configurations

AArch32 System register CNTV_CVAL[63:0] is architecturally mapped to AArch64 System register CNTV_CVAL_EL0[63:0].

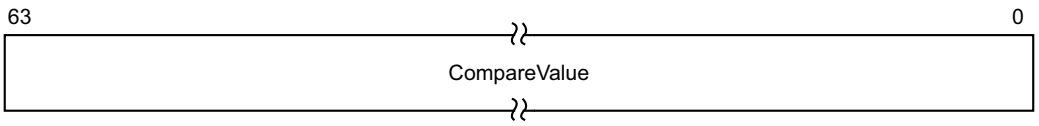
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to CNTV_CVAL are UNKNOWN.

Attributes

CNTV_CVAL is a 64-bit register.

Field descriptions

The CNTV_CVAL bit assignments are:



CompareValue, bits [63:0]

Holds the EL1 virtual timer CompareValue.

When CNTV_CTL.ENABLE is 1, the timer condition is met when (CNTVCT - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTV_CTL.ISTATUS is set to 1.
- If CNTV_CTL.IMASK is 0, an interrupt is generated.

When CNTV_CTL.ENABLE is 0, the timer condition is not met, but CNTVCT continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTV_CVAL

Accesses to this register use the following encodings in the System instruction encoding space:

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1110	0b0011

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            AArch64.AArch32SystemAccessTrap(EL1, 0x04);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0VTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHVS_CVAL_EL2;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHV_CVAL_EL2;
    else
        return CNTV_CVAL;
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    else
        return CNTV_CVAL;
elsif PSTATE.EL == EL2 then
    return CNTV_CVAL;
elsif PSTATE.EL == EL3 then
    return CNTV_CVAL;
```

MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1110	0b0011

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x04);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0VTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHVS_CVAL_EL2 = R[t2]:R[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHV_CVAL_EL2 = R[t2]:R[t];
    else
        CNTV_CVAL = R[t2]:R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    else
        CNTV_CVAL = R[t2]:R[t];
elsif PSTATE.EL == EL2 then
    CNTV_CVAL = R[t2]:R[t];
elsif PSTATE.EL == EL3 then
    CNTV_CVAL = R[t2]:R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.7.23 CNTV_TVAL, Counter-timer Virtual Timer TimerValue register

The CNTV_TVAL characteristics are:

Purpose

Holds the timer value for the virtual timer.

Configurations

AArch32 System register CNTV_TVAL[31:0] is architecturally mapped to AArch64 System register CNTV_TVAL_EL0[31:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to CNTV_TVAL are UNKNOWN.

Attributes

CNTV_TVAL is a 32-bit register.

Field descriptions

The CNTV_TVAL bit assignments are:

31 … 0
TimerValue

TimerValue, bits [31:0]

The TimerValue view of the virtual timer.

On a read of this register:

- If CNTV_CTL.ENABLE is 0, the value returned is UNKNOWN.
- If CNTV_CTL.ENABLE is 1, the value returned is (CNTV_CVAL - CNTVCT).

On a write of this register, CNTV_CVAL is set to (CNTVCT + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTP_CTL.ENABLE is 1, the timer condition is met when (CNTVCT - CNTP_CVAL) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTV_CTL.ISTATUS is set to 1.
- If CNTV_CTL.IMASK is 0, an interrupt is generated.

When CNTV_CTL.ENABLE is 0, the timer condition is not met, but CNTVCT continues to count, so the TimerValue view appears to continue to count down.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTV_TVAL

Accesses to this register use the following encodings in the System instruction encoding space:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0011	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0VTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        return CNTHVS_TVAL_EL2;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        return CNTHV_TVAL_EL2;
    else
        return CNTV_TVAL;
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        return CNTV_TVAL;
elsif PSTATE.EL == EL2 then
    return CNTV_TVAL;
elsif PSTATE.EL == EL3 then
    return CNTV_TVAL;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0011	0b000

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0VTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented("ARMv8.4-SecEL2") then
        CNTHVS_TVAL_EL2 = R[t];
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHV_TVAL_EL2 = R[t];
    else
        CNTV_TVAL = R[t];
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        CNTV_TVAL = R[t];
elsif PSTATE.EL == EL2 then
    CNTV_TVAL = R[t];
elsif PSTATE.EL == EL3 then
    CNTV_TVAL = R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.7.24 CNTVCT, Counter-timer Virtual Count register

The CNTVCT characteristics are:

Purpose

Holds the 64-bit virtual count value. The virtual count value is equal to the physical count value minus the virtual offset visible in CNTVOFF.

Configurations

AArch32 System register CNTVCT[63:0] is architecturally mapped to AArch64 System register CNTVCT_EL0[63:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to CNTVCT are UNKNOWN.

The value of this register is the same as the value of CNTPCT in the following conditions:

- When EL2 is not implemented.
- When EL2 is implemented and is using AArch64, HCR_EL2.{E2H, TGE} is {1, 1}, and this register is read from Non-secure EL0.

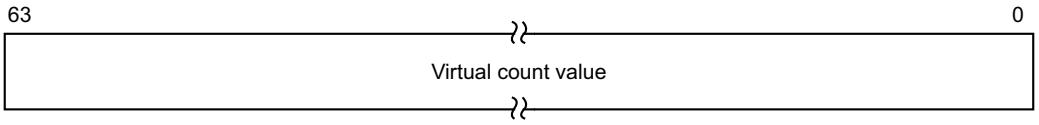
All reads to the CNTVCT occur in program order relative to reads to CNTVCTSS or CNTVCT.

Attributes

CNTVCT is a 64-bit register.

Field descriptions

The CNTVCT bit assignments are:



Bits [63:0]

Virtual count value.

Accessing the CNTVCT

Accesses to this register use the following encodings in the System instruction encoding space:

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1110	0b0001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VCTEN ==
'0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x04);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0VCTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VCTEN ==
'0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTHCTL_EL2.EL1TVCT ==
'1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    else
        return CNTVCT;
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTHCTL_EL2.EL1TVCT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    else
        return CNTVCT;
elsif PSTATE.EL == EL2 then
    return CNTVCT;
elsif PSTATE.EL == EL3 then
    return CNTVCT;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.7.25 CNTVCTSS, Counter-timer Self-Synchronized Virtual Count register

The CNTVCTSS characteristics are:

Purpose

Holds the 64-bit virtual count value. The virtual count value is equal to the physical count value visible in CNTPCT minus the virtual offset visible in CNTVOFF.

Configurations

AArch32 System register CNTVCTSS[63:0] is architecturally mapped to AArch64 System register CNTVCTSS_EL0[63:0].

This register is present only when AArch32 is supported at any Exception level and ARMv8.6-ECV is implemented. Otherwise, direct accesses to CNTVCTSS are UNDEFINED.

All reads to the CNTVCTSS occur in program order relative to reads to CNTVCT or CNTVCTSS.

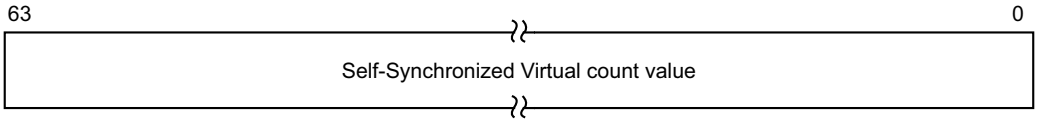
This register is a self-synchronised view of the CNTVCT counter, and cannot be read speculatively.

Attributes

CNTVCTSS is a 64-bit register.

Field descriptions

The CNTVCTSS bit assignments are:



Bits [63:0]

Self-Synchronized Virtual count value.

Accessing the CNTVCTSS

Accesses to this register use the following encodings in the System instruction encoding space:

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1110	0b1001

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VCTEN ==
'0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x04);
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0VCTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VCTEN ==
'0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTHCTL_EL2.EL1TVCT ==
'1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    else
        return CNTVCTSS;
elsif PSTATE.EL == EL1 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTHCTL_EL2.EL1TVCT == '1'
then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    else
        return CNTVCTSS;
elsif PSTATE.EL == EL2 then
    return CNTVCTSS;
elsif PSTATE.EL == EL3 then
    return CNTVCTSS;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G8.7.26 CNTVOFF, Counter-timer Virtual Offset register

The CNTVOFF characteristics are:

Purpose

Holds the 64-bit virtual offset. This is the offset between the physical count value visible in CNTPCT and the virtual count value visible in CNTVCT.

Configurations

AArch32 System register CNTVOFF[63:0] is architecturally mapped to AArch64 System register CNTVOFF_EL2[63:0].

This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to CNTVOFF are UNKNOWN.

If EL2 is not implemented, this register is RES0 from EL3 and the virtual counter uses a fixed virtual offset of zero.

Note

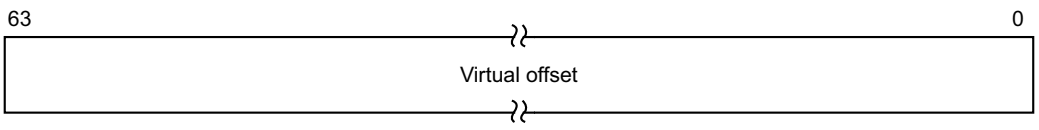
When EL2 is implemented and is using AArch64, if HCR_EL2.{E2H, TGE} is {1, 1}, the virtual counter uses a fixed virtual offset of zero when CNTVCT is read from Non-secure EL0.

Attributes

CNTVOFF is a 64-bit register.

Field descriptions

The CNTVOFF bit assignments are:



Bits [63:0]

Virtual offset.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTVOFF

Accesses to this register use the following encodings in the System instruction encoding space:

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1110	0b0100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    return CNTVOFF;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL3 then
    return CNTVOFF;
```

MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1110	0b0100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    CNTVOFF = R[t2]:R[t];
elsif PSTATE.EL == EL3 then
    CNTVOFF = R[t2]:R[t];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Part H

External Debug

Chapter H1
About External Debug

This chapter gives an overview of Armv8 external debug, and specifies the required debug authentication. It contains the following sections:

- *Introduction to external debug* on page H1-6922.
- *External debug* on page H1-6923.
- *Required debug authentication* on page H1-6924.

Note

For information about self-hosted debug, see Chapter D2 *AArch64 Self-hosted Debug* and Chapter G2 *AArch32 Self-hosted Debug*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H1.1 Introduction to external debug

Armv8 supports both:

Self-hosted debug

The PE itself hosts a debugger. That is, developers developing software to run on the PE use debugger software running on the same PE.

External debug

The debugger is external to the PE. The debugging might be either on-chip, for example in a second PE, or off-chip, for example a JTAG debugger that accesses the chip through a Debug Access Port.

External debug is particularly useful for:

- Hardware bring-up. That is, debugging during development when a system is first powered up and not all of the software functionality is available.
- PEs that are deeply embedded inside systems.

To support external debug, the Arm architecture defines required features that are called *external debug features*.

Note

An external debugger has a potentially high level of control over and visibility into the PE. The system sets this level using debug authentication. See *Required debug authentication* on page H1-6924.

If the debug authentication level is set too low, agents may be able to bypass elements of the security and privilege models. This includes both off-chip agents and on-chip agents such as unprivileged or Non-secure software.

H1.1.1 Definition and constraints of a debugger in the context of external debug

When the description of external debug in this Part of the manual describes a *debugger* as controlling external debug this debugger might be a second on-chip PE or an off-chip device such as a JTAG debugger using a Debug Access Port (DAP).

If a Debug Access Port is implemented:

- When debug is prohibited at the Debug Access Port, the port must not generate accesses to the external debug interface of the PE.
- When Secure debug is prohibited at the Debug Access Port, the port must not generate Secure accesses to the external debug interface of the PE.
- When Secure accesses are allowed at the Debug Access Port, the port must be able to generate Secure accesses.

If ARMv8.4-Debug is not implemented, accesses to the PE are controlled by the external authentication interface functions, `ExternalInvasiveDebugEnabled()`, `ExternalNoninvasiveDebugEnabled()`, `ExternalSecureNoninvasiveDebugEnabled()` and `ExternalSecureInvasiveDebugEnabled()`. The external authentication interface functions override MDCR_EL3.{EPMAD, EDAD}.

If ARMv8.4-Trace is implemented, the bus master, which may be the Debug Access Port, controls the accesses it makes to the PE and MDCR_EL3.{EPMAD, EDAD} control Non-secure access to registers.

The Debug Access Port is not required to use the same authentication interface as the PE.

Arm recommends the following authentication interface:

- When `ExternalSecureInvasiveDebugEnabled()`== FALSE at the PE, Secure debug is disabled at the DAP.
- When `ExternalInvasiveDebugEnabled()`== FALSE at the PE, all debug is prohibited at the DAP.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H1.2 External debug

Debug events allow an external debugger to halt the PE. Armv8 provides the following debug events:

- *Halting Step debug events* on page H3-6968:
  - The debugger can use this resource to make the PE step through code one line at a time.
- *Halt Instruction debug event* on page H3-6978:
  - This might occur when software executes the Halting breakpoint instruction, `HLT`.
- *Exception Catch debug event* on page H3-6979:
  - This can be programmed to occur on all entries to a given Exception level.
- *External Debug Request debug event* on page H3-6983:
  - An embedded cross-trigger can signal this debug event.
- *OS Unlock Catch debug event* on page H3-6984:
  - This might occur when the state of the OS Lock changes from locked to unlocked.
- *Reset Catch debug events* on page H3-6985:
  - This might occur when the PE exits reset state.
- *Software Access debug event* on page H3-6986:
  - This can be programmed to occur when software tries to access the Breakpoint Value registers, the Breakpoint Control registers, the Watchpoint value registers, or the Watchpoint Control registers. It caused a trap to Debug state.

Breakpoints and watchpoints can also halt the PE.

When the PE is in Debug state:

- It stops executing instructions from the location indicated by the program counter, and is instead controlled through the external debug interface.
- The *Instruction Transfer Register,* ITR, passes instructions to the PE to execute in Debug state:
  - The ITR contains a single register, EDITR, and associated flow-control flags.
- The *Debug Communications Channel*, DCC, passes data between the PE and the debugger:
  - The DCC includes the data transfer registers, DTRRX and DTRTX, and associated flow-control flags.
  - Although the DCC is an essential part of Debug state operation, it can also be used in Non-debug state.
- The PE cannot service any interrupts in Debug state.

Chapter H2 *Debug State* describes Debug state in more detail.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H1.3 Required debug authentication

Any implementation must provide the debug authentication defined in this section, that controls:

- Whether the PE can halt.
- Whether non-invasive debug is permitted.
- Some legacy aspects of the AArch32 self-hosted debug model.

The pseudocode functions shown in Table H1-1, and the conditions that follow that table, define the architectural requirements for debug authentication.

Table H1-1 Debug authentication functions

Pseudocode function	Description
`ExternalSecureNoninvasiveDebugEnabled()`	Returns TRUE if Secure non-invasive debug is enabled.
`AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()`	Returns TRUE if Secure invasive self-hosted debug is enabled in AArch32 state.
`ExternalSecureInvasiveDebugEnabled()`	Returns TRUE if Secure invasive debug is enabled.
`ExternalNoninvasiveDebugEnabled()`	Returns TRUE if Non-secure non-invasive debug is enabled.
`ExternalInvasiveDebugEnabled()`	Returns TRUE if Non-secure invasive debug is enabled.

The following conditions always apply:

- If `ExternalInvasiveDebugEnabled()` is FALSE then `ExternalSecureInvasiveDebugEnabled()` is FALSE.
- If `ExternalNoninvasiveDebugEnabled()` is FALSE then `ExternalSecureNoninvasiveDebugEnabled()` is FALSE.
- If `ExternalInvasiveDebugEnabled()` is TRUE then `ExternalNoninvasiveDebugEnabled()` is TRUE.
- If `ExternalSecureInvasiveDebugEnabled()` is TRUE then `ExternalSecureNoninvasiveDebugEnabled()` is TRUE.

If ARMv8.4-Debug is implemented:

- `ExternalNoninvasiveDebugEnabled()` always returns TRUE.
- `ExternalSecureNoninvasiveDebugEnabled()`returns the same as `ExternalSecureInvasiveDebugEnabled()`.

Arm recommends the use of the interface described in *Recommended authentication interface* on page K2-7879 to provide this debug authentication. The pseudocode functions in Chapter J1 *Armv8 Pseudocode*, that are linked to by the entries in the *Pseudocode function* column of Table H1-1, assume that this interface is implemented.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter H2
Debug State

This chapter describes Debug state. It contains the following sections:

- *About Debug state* on page H2-6926.
- *Halting the PE on debug events* on page H2-6927.
- *Entering Debug state* on page H2-6933.
- *Behavior in Debug state* on page H2-6936.
- *Exiting Debug state* on page H2-6963.

Note

Table K14-1 on page K14-8038 disambiguates the general register references used in this chapter.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H2.1 About Debug state

In external debug, debug events allow an external debugger to halt the PE. The PE then enters Debug state. When the PE is in Debug state:

- It stops executing instructions from the location indicated by the program counter, and is instead controlled through the external debug interface.
- The *Instruction Transfer Register,* ITR, passes instructions to the PE to execute in Debug state.
- The *Debug Communications Channel*, DCC, passes data between the PE and the debugger.

The PE cannot service any interrupts in Debug state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H2.2 Halting the PE on debug events

For details of debug events, see *Introduction to Halting debug events* on page H3-6966 and *Breakpoint and Watchpoint debug events* on page H2-6928.

On a debug event, the PE must do one of the following:

- Enter Debug state.
- Pend the debug event.
- Generate a debug exception.
- Ignore the debug event.

This behavior depends on both:

- Whether halting is allowed by the current state of the debug authentication interface. See *Halting allowed and halting prohibited*.
- The type of debug event and the programming of the debug control registers.
  - See *Halting debug events* for all Halting debug events.
  - See *Breakpoint and Watchpoint debug events* on page H2-6928 for Breakpoint and Watchpoint debug events.

See also *Other debug exceptions* on page H2-6928.

This means that behavior can be CONSTRAINED UNPREDICTABLE if the conditions change. See *Synchronization and Halting debug events* on page H3-6987.

Summary of debug events and possible outcomes on page H3-6966 summarizes the possible outcomes of each type of debug event.

H2.2.1 Halting allowed and halting prohibited

Halting can be either allowed or prohibited:

- Halting is always prohibited in Debug state.
- Halting is always prohibited when `DoubleLockStatus()` == TRUE.
  - This means that ARMv8.0-DoubleLock is implemented and OS Double lock is locked.
- Halting is also controlled by the IMPLEMENTATION DEFINED authentication interface, and is prohibited when either:
  - The PE is in Non-secure state and `ExternalInvasiveDebugEnabled()` == FALSE.
  - The PE is in Secure state and `ExternalSecureInvasiveDebugEnabled()` == FALSE.

    Note

    See Appendix K2 *Recommended External Debug Interface* for more information on these functions.

- Otherwise, halting is allowed.

For more information see:

- *Pseudocode description of Halting on debug events* on page H2-6932
- *Required debug authentication* on page H1-6924.

H2.2.2 Halting debug events

The Halting debug events are described in Chapter H3 *Halting Debug Events*.

When a Halting debug event is generated, it causes entry to Debug state if all of:

- Halting is allowed. See *Halting allowed and halting prohibited*.
- The Halting debug event is one of:
  - A Halt Instruction debug event and EDSCR.HDE == 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— A Software Access debug event and OSLSR_EL1.OSLK == 0, meaning that the OS Lock is unlocked.
— Neither a Halt Instruction debug event nor a Software Access debug event.

Note

— A Halt Instruction debug event is the only Halting debug event that relies on EDSCR.HDE == 1.
— Halting on Breakpoint and Watchpoint debug events is also controlled by EDSCR.HDE. See *Breakpoint and Watchpoint debug events*.
— EDSCR.HDE can be written by software when the OS Lock is locked. Privileged code can use MDCR_EL3.TDOSA and HDCR.TDOSA to trap writes to these registers.

If a Halting debug event does not generate entry to Debug state because the conditions listed in this section do not hold, then:

- If the Halting debug event is a Halt Instruction debug event, the instruction that generated the Halting debug event is treated as UNDEFINED.
- If the Halting debug event is an Exception Catch debug event or a Software Access debug event, it is ignored.

In all other cases the Halting debug event is pended, see *Pending Halting debug events* on page H3-6987.

Summary of actions from debug events on page H2-6931 summarizes the possible outcome for each type of Debug event.

Note

Halting debug events never generate debug exceptions.

H2.2.3 Breakpoint and Watchpoint debug events

A breakpoint or watchpoint generates an entry to Debug state if all of the following conditions hold:

- Halting debug is enabled, that is EDSCR.HDE == 1.
- Halting is allowed. See *Halting allowed and halting prohibited* on page H2-6927.
- The OS Lock is unlocked, that is OSLSR.OSLK == 0.

The Address Mismatch breakpoint type is reserved when all of these conditions are met.

MDSCR_EL1.MDE or DBGDSCRext.MDBGen is ignored when determining whether to enter Debug state. A breakpoint or watchpoint that generates entry to Debug state is a Breakpoint or Watchpoint debug event and does not generate a debug exception.

A breakpoint or watchpoint that does not generate an entry to Debug state either:

- Generates a Breakpoint or Watchpoint exception.
- Is ignored.

Note

EDSCR.HDE is ignored when determining whether to generate a debug exception. The debug exception is suppressed only if the PE enters Debug state. This means that the use of Halting debug mode in Non-secure state does not affect the Exception model in Secure state.

See Chapter D2 *AArch64 Self-hosted Debug*, Chapter G2 *AArch32 Self-hosted Debug*, and the Note in *Other debug exceptions*.

H2.2.4 Other debug exceptions

The following events never generate entry to Debug state:

- Breakpoint Instruction exceptions.
- Software Step exceptions.
- Vector Catch exceptions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The behavior of these events is unchanged when Halting debug mode is enabled, that is when EDSCR.HDE == 1. This means that these events can do one of the following:

- They can generate a debug exception.
- They can be ignored.

For additional information, see Chapter D2 *AArch64 Self-hosted Debug* and Chapter G2 *AArch32 Self-hosted Debug*.

H2.2.5 Debug state entry and debug event prioritization

The following are synchronous debug events:

- Breakpoint debug event.
- Watchpoint debug event.
- Halting Step debug event.
- Halt Instruction debug event.
- Exception Catch debug event.
- Software Access debug event.
- Reset Catch debug event.

Each of these synchronous debug events are treated as a synchronous exception generated by an instruction, or by the taking of an exception or reset. That is, if halting is allowed, the synchronous debug event must be taken before any subsequent instructions are executed. Reset Catch debug events must be taken before the PE executes the instruction at the reset vector.

Note

- Reset Catch and Exception Catch debug events might be generated asynchronously, because they can result from an asynchronous exception. However, if halting is allowed after the reset or asynchronous exception has been processed, the Reset Catch or Exception Catch debug event is taken synchronously.
- The Halting Step debug event is generated by the instruction after the stepped instruction. Therefore, if the stepped instruction generates any other synchronous exceptions or debug events these are taken first.

If halting is prohibited then Halting Step debug events and Reset Catch debug events might be pended and taken asynchronously. OS Unlock Catch debug events are always pended and taken asynchronously. See *Pending Halting debug events* on page H3-6987.

The architecture does not define when asynchronous debug events are taken, and therefore the prioritization of asynchronous debug events is IMPLEMENTATION DEFINED. See *Synchronization and Halting debug events* on page H3-6987.

The following list shows how the synchronous debug events are prioritized, with 1being the highest priority.

Note

The priority numbering is the same as the numbering for AArch64 synchronous exception priorities listed in *Synchronous exception types, routing and priorities* on page D1-2354, and in particular *Prioritization and recognition of interrupts* on page D1-2371. This numbering correlates with the equivalent AArch32 list in *Exception prioritization for exceptions taken to AArch32 state* on page G1-5706.

The priority for synchronous debug events is as follows:

1 Reset Catch debug event. See *Reset Catch debug events* on page H3-6985.

This debug event has a higher priority than the synchronous exceptions listed in *Synchronous exception types, routing and priorities* on page D1-2354.

2 Exception Catch debug event. See *Exception Catch debug event* on page H3-6979.

This debug event can be assigned one of two priorities. When it has a priority of 2, it has a higher priority than the synchronous exceptions listed in the Exception model. See *Exception Catch debug event* on page H3-6979.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

3 Halting Step debug event. See *Halting Step debug events* on page H3-6968.

This debug event has a higher priority than the synchronous exceptions listed in the Exception model.

4 This event is not a debug event.

5 Exception Catch debug event. See *Exception Catch debug event* on page H3-6979.

This debug event can be assigned one of two priorities, 0 or 5. See *Exception Catch debug event* on page H3-6979.

6 - 7 These events are not debug events.

8 Breakpoint exception or debug event or Address Matching Vector Catch exception. See *Breakpoint exceptions* on page D2-2439, and *Vector Catch exceptions* on page G2-5869.

These two debug events have the same priority.

9 This event is not a debug event.

10 Halt Instruction debug event. See *Halt Instruction debug event* on page H3-6978.

11 - 26 These events are not debug events.

27 Software Access debug event. See *Software Access debug event* on page H3-6986.

28 - 29 These events are not debug events.

30 Watchpoint exception or debug event. See *Watchpoint exceptions* on page D2-2457 for exceptions taken from AArch64 state, or *Watchpoint exceptions* on page G2-5855 for exceptions taken from AArch32 state.

31 This event is not a debug event.

For Reset Catch debug events and Halting Step debug events the priorities listed in this section only apply when halting is allowed at the time the event is generated. This means that the event is taken synchronously and not pended.

For more information on the prioritization of exceptions see:

- *Synchronous exception types, routing and priorities* on page D1-2354.
- *Prioritization and recognition of interrupts* on page D1-2371.
- *Exception prioritization for exceptions taken to AArch32 state* on page G1-5706. This section covers synchronous and asynchronous exceptions.

Breakpoint debug events and Vector Catch exception

An Address Matching Vector Catch exception has the same priority as a Breakpoint debug event. See *Synchronous exception prioritization for exceptions taken to AArch64 state* on page D1-2355.

The prioritization of these events is unchanged even if the breakpoint generates entry to Debug state instead of a Breakpoint exception. This means that if a single instruction generates both an Address Matching Vector Catch exception and a Breakpoint debug event, there is a CONSTRAINED UNPREDICTABLE choice of:

- The PE entering Debug state due to the Breakpoint debug event.
- A Vector Catch exception.

This only applies if all of the following are true:

- Halting debug is enabled.
- Halting is allowed.
- The OS Lock is unlocked.

An Exception Trapping Vector Catch exception must be generated immediately following the exception that generated it. This means that it does not appear in the priority table.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H2.2.6 Imprecise entry to Debug state

Entry to Debug state is normally precise, meaning that the PE cannot enter Debug state if it can neither complete nor abandon all currently executing instructions and leave the PE in a precise state.

A debugger can write a value of 1 to EDRCR.CBRRQ to allow imprecise entry to Debug state. An External Debug Request debug event must be pending before writing 1 to this bit. Support for this feature is OPTIONAL and it is IMPLEMENTATION DEFINED when it is effective at forcing entry to Debug state.

The PE ignores writes to this bit if either:

- External debugging is not enabled, meaning `ExternalInvasiveDebugEnabled()` == FALSE.
- Secure external debugging is not enabled, meaning `ExternalSecureInvasiveDebugEnabled()` == FALSE, and either:
  - — EL3 is not implemented and the implemented Security state is Secure state.
  - — EL3 is implemented.

Example H2-1 shows how entry to Debug state can be forced.

Example H2-1 Forcing entry to Debug state

The debugger pends an External Debug Request debug event through the CTI to halt a program that has stopped responding. However, the memory system is not responding and a memory access instruction cannot complete. This means that Debug state cannot be entered precisely. The debugger writes a value of 1 to EDRCR.CBRRQ. The PE cancels all outstanding memory accesses and enters Debug state. As some instructions might not have completed correctly, entry to Debug state is imprecise.

When Debug state is entered imprecisely, all memory access instructions executed through the ITR have CONSTRAINED UNPREDICTABLE behavior. The value of all registers is UNKNOWN, but might be useful for diagnostic purposes.

H2.2.7 Summary of actions from debug events

Table H2-1 on page H2-6932 shows the Software and Halting debug events. In Table H2-1 on page H2-6932 the columns have the following meaning:

Debug event type

This means the type of debug event where:

Other software Means one of:

- *Software Step exceptions* on page D2-2472.
- *Breakpoint Instruction exceptions* on page D2-2437.
- *Vector Catch exceptions* on page D2-2471 for AArch64 state or *Vector Catch exceptions* on page G2-5869 for AArch32 state.

Other Halting Means one of the following:

- *Halting Step debug events* on page H3-6968.
- *External Debug Request debug event* on page H3-6983.
- *Reset Catch debug events* on page H3-6985.
- *OS Unlock Catch debug event* on page H3-6984.

Other debug events are referred to explicitly.

Authentication

This means halting is allowed by the IMPLEMENTATION DEFINED external authentication interface. It is the result of one of the following pseudocode functions:

In Secure state `ExternalSecureInvasiveDebugEnabled()`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In Non-secure state `ExternalInvasiveDebugEnabled()`.

DLK This indicates whether ARMv8.0-DoubleLock is implemented and locked, `DoubleLockStatus()` == TRUE.

OSLK This is the value of OSLSR.OSLK. It indicates whether the OS Lock is locked.

HDE This is the value of EDSCR.HDE. It indicates whether Halting debug is enabled.

The letter X in Table H2-1 indicates that the value can be either 0 or 1.

Table H2-1 Debug authentication for external debug

Debug event type	Authentication	DLK	OSLK	HDE	Behavior
Other software	X	X	X	X	Handled by the Exception model
Breakpoint or Watchpoint debug event	X	TRUE	X	X	Handled by the Exception model (ignored)
	X	FALSE	1	X	Handled by the Exception model (ignored)
	FALSE	FALSE	0	X	Handled by the Exception model
	TRUE	FALSE	0	0	Handled by the Exception model
	TRUE	FALSE	0	1	Entry to Debug state
Halt Instruction debug event	FALSE	X	X	X	undefined
	TRUE	TRUE	X	X	undefined
	TRUE	FALSE	X	0	undefined
	TRUE	FALSE	X	1	Entry to Debug state
Exception Catch debug event	FALSE	X	X	X	Ignored
	TRUE	TRUE	X	X	Ignored
	TRUE	FALSE	X	X	Entry to Debug state
Software Access debug event	FALSE	X	X	X	Ignored
	TRUE	TRUE	X	X	Ignored
	TRUE	FALSE	1	X	Ignored
	TRUE	FALSE	0	X	Entry to Debug state
Other Halting	FALSE	X	X	X	Debug event is pended
	TRUE	TRUE	X	X	Debug event is pended
	TRUE	FALSE	X	X	Entry to Debug state

H2.2.8 Pseudocode description of Halting on debug events

The `Halted()`, `Restarting()`, `HaltingAllowed()`, and `HaltOnBreakpointOrWatchpoint()` functions are described in the Armv8 pseudocode.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H2.3 Entering Debug state

On entry to Debug state, the preferred restart address and PSTATE are saved in DLR and DSPSR. The PE remains in the mode and security state from which it entered Debug state.

If EDRCR.CBRRQ has a value of 0, entry to Debug state is precise. If EDRCR.CBRRQ has a value of 1, then imprecise entry to Debug state is permitted.

If a Watchpoint debug event causes an entry to Debug state, the address of the access that generated the Watchpoint debug event is recorded in EDWAR.

For more information see:

- *Determining the memory location that caused a Watchpoint exception* on page D2-2465 for a debug event taken from AArch64 state.
- *Determining the memory location that caused a Watchpoint exception* on page G2-5862 for a debug event taken from AArch32 state.

Other than the effect on PSTATE and EDSCR, entry to Debug state is not a *Context synchronization event*. The effects of entry to Debug state on PSTATE and EDSCR are synchronized.

H2.3.1 Entering Debug state from AArch32 state

When entering Debug state from AArch32 state, the PE remains in AArch32 state. In AArch32 Debug state the PE executes T32 instructions, regardless of the value of PSTATE.T before entering Debug state.

To allow the debugger to determine the state of the PE, the current Execution state for all four Exception levels can be read from EDSCR.RW, and the current Exception level can be read from EDSCR.EL.

The current endianness state, PSTATE.E, is unchanged on entry to Debug state.

Note

- If EL1 is using AArch32 state, the current endianness state can differ from that indicated by SCTLR.EE.
- If EL2 is using AArch32 state, the current endianness state can differ from that indicated by HSCTLR.EE.
- On entry to Debug state from AArch32 state, PSTATE.SS is copied to DSPSR.SS, even though the PE remains in AArch32 state.

See also *Effect of entering Debug state on PSTATE* on page H2-6935.

H2.3.2 Effect of Debug state entry on DLR and DSPSR

DLR is set to the preferred restart address for the debug event, that depends on the event type. The value of PSTATE is saved in DSPSR.

For entry to Debug state from AArch32 state, the values saved in DSPSR.IT are always correct for the preferred restart address.

For synchronous Halting debug events, the preferred restart address is the address of the instruction that generated the debug event. It is CONSTRAINED UNPREDICTABLE whether DSPSR_EL0.BTYPE is set to the value of PSTATE.BTYPE or 0 for synchronous debug events other than the following debug events:

- A Halting Step debug event.
- A Breakpoint debug event.
- A Halt Instruction debug event.

For asynchronous Halting debug events, including pending Halting debug events taken asynchronously, the preferred restart address is the address of the first instruction that must be executed on exit from Debug state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This means that:

- For Breakpoint and Watchpoint debug events, the preferred restart address is the same as the preferred return address for a debug exception, as described in Chapter D2 *AArch64 Self-hosted Debug* and Chapter G2 *AArch32 Self-hosted Debug*.
- For Halt Instruction debug events DLR is set to the address of the `HLT` instruction and DSPSR.IT is correct for the `HLT` instruction.
- For Software Access debug events, DLR is set to the address of the accessing instruction and DSPSR.IT is correct for this instruction.
- For Halting Step debug events taken synchronously, DLR and DSPSR are set as the ELR and SPSR would be set for a Software Step exception. This is usually the address of, and PSTATE for, the instruction after the one that was stepped.
- For Exception Catch debug events:
  - — If the debug event is generated on taking an exception to a trapped Exception level, the DLR is set to the address of the exception vector the PE would have started fetching from. This is UNKNOWN if the VBAR for the Exception level has never been initialized. The DSPSR records the value of PSTATE after taking the exception. The Exception Catch occurs after the SPSR and the Link register are set, and the debugger can use these registers to determine where in the application program the exception occurred.

    Note

    Depending on the target Exception level and Execution state for the exception, the Link register is one of ELR_EL1, ELR_EL2, ELR_EL3, ELR_hyp, or LR (R14).

  - — If the debug event is generated on an exception return to a trapped Exception level, the DLR is set to the target address of the exception return and the DSPSR records the value of PSTATE after the exception return.
- Reset Catch debug events taken synchronously behave like Exception Catch debug events.
- For Reset Catch debug events and Exception Catch debug events generated on reset to a trapped Exception level, the DLR is set to is set to the reset address and the DSPSR records the reset value of PSTATE.
- For pending Halting debug events and External Debug Request debug events, DLR is set to the address of the first instruction that must be executed on exit from Debug state and DSPSR.IT is correct for this instruction. See *Pending Halting debug events* on page H3-6987.

Normally DLR is aligned according to the instruction set state indicated in DSPSR. However, a debug event might be taken at a point where the PC is not aligned.

H2.3.3 Effect of Debug state entry on System registers, the Event register, and Exclusives monitors

Entering Debug state has no effect on System registers other than DLR and DSPSR. In particular, ESRs, FARs, and FSRs are not updated on entering Debug state. SCR is unchanged, even when entering Debug state from EL3.

Entering Debug state has no architecturally-defined effect on the Event Register and Exclusives monitors.

Note

Entry to Debug state might set the Event Register or clear the Exclusives monitors, or both. However, this is not a requirement, and debuggers must not rely on any implementation specific behavior.

Unless otherwise described in this reference manual, instructions executed in Debug state have their architecturally-defined effects on the System registers, the Event register, and Exclusives monitors.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H2.3.4 Effect of entering Debug state on PSTATE

The effect of an entry to Debug state on PSTATE is described in *Entering Debug state* on page H2-6933 and *Entering Debug state from AArch32 state* on page H2-6933.

On entry to Debug state after PSTATE is saved in DSPSR:

- PSTATE.IL is cleared to 0.
- PSTATE.TCO is set to 1.
- PSTATE.BTYPE is set to 0.
- PSTATE.{IT, T, SS, D, A, I, F, SSBS} are set to UNKNOWN values

PSTATE.{N, Z, C, V, Q, GE, E, M, nRW, EL, SP, PAN, UAO, DIT} are unchanged.

For more information see *PSTATE in Debug state* on page H2-6936.

H2.3.5 Entering Debug state during loads and stores

The PE can enter Debug state during instructions that perform a sequence of memory accesses, as opposed to a single single-copy atomic access, because of a Watchpoint debug event. The effect of entering Debug state on such an instruction is the same as taking a Data Abort exception during such an instruction.

In addition, when executing in AArch64 state, the PE can enter Debug state during instructions that perform a sequence of memory accesses because of an External Debug Request debug event. The effect of entering Debug state on such an instruction is the same as taking an interrupt exception during such an instruction.

This applies to all memory types.

H2.3.6 Entering Debug state and Software Step

When Software Step is active, a debug event that causes entry to Debug state behaves like an exception taken to an Exception level above the debug target Exception level. That is:

- If the instruction that is stepped generates a synchronous debug event that causes entry to Debug state, or an asynchronous debug event is taken before the step completes, the PE enters Debug state with DSPSR.SS set to 1.
- A pending Halting debug event or an asynchronous debug event can be taken after the step has completed. In this case the PE enters Debug state with DSPSR.SS set to 0.

In addition:

- If the instruction that is stepped generates an exception trapped by an Exception Catch debug event, the PE enters Debug state at the exception vector with DSPSR.SS set to 0. This is because PSTATE.SS is set to 0 by taking the exception.
- If the PE is reset, PSTATE.SS is reset to 0. If the following debug events are enabled, the PE enters Debug state with DSPSR.SS set to 0:
  - Reset Catch debug event at the reset Exception level.
  - Exception Catch debug event at the reset Exception level.
  - Halting Step debug event.
- If Halting Step is also active, then Halting Step and Software Step operate in parallel and can both become active-pending. In this case Halting step has a higher priority than Software step. This means that the PE enters Debug state and DSPSR.SS is set to 0.

H2.3.7 Pseudocode description of entering Debug state

The `DebugHalt` constants are described in *shared/debug/halting/DebugHalt* on page J1-7716 in the Armv8 pseudocode. The `UpdateEDSCRFields()` and `Halt()` functions are described in Chapter J1 *Armv8 Pseudocode*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H2.4 Behavior in Debug state

Instructions are executed in Debug state from the Instruction Transfer Register, ITR. The debugger controls which instructions are executed in Debug state by writing the instructions to the External Debug Instruction Transfer register, EDITR. The Execution state of the PE determines which instruction set is executed:

- If the PE is in AArch64 state it executes A64 instructions.
- If the PE is in AArch32 state it executes T32 instructions:
  - For a 32-bit T32 instruction, EDITR[15:0] specifies the first halfword and EDITR[31:16] specifies the second halfword.
  - For a 16-bit T32 instruction, EDITR[15:0] contains the instruction and EDITR[31:16] is ignored. All 16-bit T32 instructions are UNPREDICTABLE in Debug state.

The PE does not execute A32 instructions in Debug state.

Some instructions are available only in Debug state. See *Debug state operations, DCPS, DRPS, MRS, MSR* on page H2-6954. In Non-debug state these instructions are UNDEFINED.

The following sections describe behavior in Debug state:

- *PSTATE in Debug state*.
- *Executing instructions in Debug state* on page H2-6937.
- *Decode tables* on page H2-6949.
- *Security in Debug state* on page H2-6953.
- *Privilege in Debug state* on page H2-6954.
- *Debug state operations, DCPS, DRPS, MRS, MSR* on page H2-6954.
- *Exceptions in Debug state* on page H2-6957.
- *Accessing registers in Debug state* on page H2-6959.
- *Accessing memory in Debug state* on page H2-6962.

This section specifies the CONSTRAINED UNPREDICTABLE behaviors that apply in Debug state, but see *Changing the value of EDECR.SS when not in Debug state* on page H3-6975 for a change in Non-debug state that causes CONSTRAINED UNPREDICTABLE behavior.

H2.4.1 PSTATE in Debug state

PSTATE.{N, Z, C, V, Q, GE, IT, T, SS, D, A, I, F, SSBS} are all ignored in Debug state:

- There are no conditional instructions in Debug state.
- In AArch32 state, the PE only executes T32 instructions and PSTATE.IT is ignored.
- Asynchronous exceptions and debug events are ignored.
- Software step is inactive.

Instructions executed in Debug state indirectly read PSTATE.{UAO, PAN, IL, E, M, nRW, EL, SP} as they would in Non-debug state.

Note

PSTATE.DIT is not guaranteed to have any effect in Debug state.

In Debug state:

- PSTATE.PAN is set to 1 by:
  - A `DCPS` instruction to EL1 using AArch64 if SCTLR_EL1.SPAN == 0.
  - A `DCPS` instruction to EL2 using AArch64 if SCTLR_EL2.SPAN == 0.
- PSTATE.UAO is set to 0 by a `DCPS` instruction to AArch64 state.
- PSTATE.TCO is set to 1 by a `DCPS` instruction to AArch64 state.
- PSTATE can also be changed by taking exceptions in Debug state, and by the execution of `DCPS` and `DRPS` instructions.

When in Debug state, if ARMv8.0-SSBS is implemented, then hardware is permitted to load or store speculatively, regardless of the value of PSTATE.SSBS.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H2.4.2 Executing instructions in Debug state

The instructions executed in Debug state must be either A64 instructions or T32 instructions, depending on the current Execution state.

Each instruction falls into one of the following groups:

- Debug state instructions. These are instructions that are changed in Debug state. See *A64 instructions that are changed in Debug state* and *T32 instructions that are changed in Debug state* on page H2-6943.
- Instructions that are unchanged in Debug state. See *A64 instructions that are unchanged in Debug state* and *T32 instructions that are unchanged in Debug state* on page H2-6944.
- Instructions that are UNPREDICTABLE or CONSTRAINED UNPREDICTABLE in Debug state. See *A64 instructions that are CONSTRAINED UNPREDICTABLE in Debug state* on page H2-6940 and *T32 instructions that are CONSTRAINED UNPREDICTABLE in Debug state* on page H2-6946.

All T32 instructions are treated as unconditional, regardless of PSTATE.IT. See *PSTATE in Debug state* on page H2-6936.

If EDSCR.SDD == 1 then an instruction executed in Non-secure state cannot cause entry into Secure state. See *Security in Debug state* on page H2-6953

Executing A64 instructions in Debug state

The following sections describe the behavior of the A64 instructions in Debug state:

- *A64 instructions that are changed in Debug state*.
- *A64 instructions that are unchanged in Debug state*.
- *A64 instructions that are CONSTRAINED UNPREDICTABLE in Debug state* on page H2-6940.

A64 instructions that are changed in Debug state

The following A64 instructions are defined in Debug state, but are undefined in Non-debug state:

- `DCPS`.

  Note

  `DCPS` can be UNDEFINED in certain conditions in Debug state. See *DCPS<n>* on page H2-6954.

- `DRPS`.
- `MRS` (DLR_EL0), `MRS` (DSPSR_EL0), `MSR` (DLR_EL0), `MSR` (DSPSR_EL0)

For more information see *Debug state operations, DCPS, DRPS, MRS, MSR* on page H2-6954.

A64 instructions that are unchanged in Debug state

The following list shows the instructions that are unchanged in Debug state:

Any instruction that is UNDEFINED in Non-debug state

This list of instructions excludes:

- Any instruction listed in *A64 instructions that are changed in Debug state*.
- Any instruction listed in *A64 instructions that are CONSTRAINED UNPREDICTABLE in Debug state* on page H2-6940 that is UNDEFINED because an enable or disable bit is not RES0 or RES1

Instructions that move System or Special-purpose registers to or from a general-purpose register

This list of instructions:

- Includes the instructions to transfer a general-purpose register to or from the DTR, which can be executed at any Exception level.
- Excludes PSTATE access instructions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

These instructions are:

- `MRS <special_reg>`, `MSR <special_reg>`.

  Note

  This does not include NZCV, DAIF, DAIFSet, DAIFClr, SPSel, CurrentEL, PAN, UAO, DIT, and TCO.

- `MRS <system_reg>`, `MSR <system_reg>`.

Floating-point moves between a SIMD&FP register and a general-purpose register

These instructions are:

- `FMOV` (between a general-purpose register and a half-precision register).
- `FMOV` (between a general-purpose register and a single-precision register).
- `FMOV` (between a general-purpose register and a double-precision register).
- `FMOV` (between a general-purpose register and a SIMD element).

SIMD moves between a SIMD&FP register and a general-purpose register

These instructions are:

- `INS` (from a general-purpose register to a SIMD element).
- `UMOV` (from a SIMD element to a general-purpose register).

Barriers

These instructions are:

- `DMB`.
- `DSB`.
- `ESB`, when the RAS Extension is implemented.
- `ISB`.
- `PSB`, when the Statistical Profiling Extension is implemented.
- `SB`, when ARMv8.0-SB is implemented.
- `CSDB`, when ARMv8.0-SSBS is implemented.
- `SSBB`, when ARMv8.0-SSBS is implemented.
- `PSSBB`, when ARMv8.0-SSBS is implemented.

Memory access instructions at various access sizes

The following constraints apply:

- General purpose-registers only.
- One of the following addressing modes:
  - Unscaled (9-bit signed) immediate offset.
  - Immediate (9-bit signed) post-indexed.
  - Immediate (9-bit signed) pre-indexed.
  - Unprivileged (9-bit signed).
- Not literal.
- One of the following types:
  - (Single) register.
  - Exclusive.
  - Exclusive pair.
  - Acquire/Release.
  - Acquire/Release Exclusive.
  - Acquire/Release Exclusive pair.
- 32-bit and 64-bit target register variants.

These instructions are:

- `LDR`, `LDRB`, `LDRH`, `LDRSB`, `LDRSH`, `LDRSW` (immediate, not literal).
- `LDUR`, `LDURB`, `LDURH`, `LDURSB`, `LDURSH`, `LDURSW` (immediate).

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- LDTR, LDTRB, LDTRH, LDTRSB, LDTRSH, LDTRSW (immediate).
- LDAR, LDARB, LDARH, LDXR, LDXRB, LDXRH, LDAXR, LDAXRB, LDAXRH.
- LDXP, LDAXP.
- STR, STRB, STRH (immediate).
- STUR, STURB, STURH (immediate).
- STTR,STTRB, STTRH (immediate).
- STLR, STLRB, STLRH, STXR, STXRB, STXRH, STLXR, STLXRB, STLXRH.
- STXP, STLXP.
- LDLAR, LDLARB, LDLARH - Armv8.1 instructions.
- STLLR, STLLRBB, STLLRH - Armv8.1 instructions.
- CAS, CASB, CASH, CASP - Armv8.1 instructions.
- SWP, SWPB, SWPH - Armv8.1 instructions.
- LDADD, LDADDB, LDADDH - Armv8.1 instructions.
- LDCLR, LDCLB, LDCLRH - Armv8.1 instructions.
- LDEOR, LDEORB, LDEORH - Armv8.1 instructions.
- LDSET, LDSETB, LDSETH - Armv8.1 instructions.
- LDSMAX, LDSMAXB, LDSMAXH - Armv8.1 instructions.
- LDSMIN, LDSMINB, LDSMINH - Armv8.1 instructions.
- LDUMAX, LDUMAXB, LDUMAXH - Armv8.1 instructions.
- LDUMIN, LDUMINB, LDUMINH - Armv8.1 instructions.
- STADD, STADDB, STADDH - Armv8.1 instructions.
- STCLR, STCLB, STCLRH - Armv8.1 instructions.
- STEOR, STEORB, STEORH - Armv8.1 instructions.
- STSET, STSETB, STSETH - Armv8.1 instructions.
- STSMAX, STSMAXB, STSMAXH - Armv8.1 instructions.
- STSMIN, STSMINB, STSMINH - Armv8.1 instructions.
- STUMAX, STUMAXB, STUMAXH - Armv8.1 instructions.
- STUMIN, STUMINB, STUMINH - Armv8.1 instructions.
- LDAPR, LDAPRB, LDAPRH - Armv8.3 instructions.
- LDAPURH, LDAPURSH, LDAPUR, LDAPURSW, LDAPURSB, LDAPURB - Armv8.4 instructions.
- STLUR, STLURH, STLURB - Armv8.4 instructions.

Move immediate to general-purpose register

These instructions are:

- MOVZ, MOVN, MOVK (immediate).
- MOV (between a general-purpose register and the stack pointer).

System instructions, Send Event, NOP, Clear Exclusive, and Prediction

In this context, the System instructions are the Cache maintenance instructions, TLB maintenance instructions, Address translation instructions, and the prediction restriction instructions.

These instructions are:

- IC.
- DC.
- TLBI.
- AT.
- SEV, SEVL.
- NOP.
- CLREX.
- CFP.
- CPP.
- DVP.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Basic pointer authentication instructions

When ARMv8.3-PAuth is implemented, these instructions are:

- `AUTIA`, `AUTIA1716`, `AUTIASP`, `AUTIAZ`, `AUTIZA`.
- `AUTIB`, `AUTIB1716`, `AUTIBSP`, `AUTIBZ`, `AUTIZB`.
- `AUTDA`, `AUTDZA`.
- `AUTDB`, `AUTDZB`.
- `PACIA`, `PACIA1716`, `PACIASP`, `PACIAZ`, `PACIZA`.
- `PACIB`, `PACIB1716`, `PACIBSP`, `PACIBZ`, `PACIZB`.
- `PACDA`, `PACDZA`.
- `PACDB`, `PACDZB`.
- `PACGA`.
- `XPACD`, `XPACI`, `XPACLRI`.

Memory Tagging Extension Instructions

When ARMv8.5-MemTag is implemented, Loads and stores in Debug state follow the same rules for Tag Checking as Loads and stores when not in Debug state. For more information see Chapter D6 *Armv8.5 Memory Tagging Extension*.

The following instructions are unchanged in Debug state:

- `ADDG <Xd|SP>, <Xn|SP>, #<uimm6>, #<uimm4>`
- `SUBG <Xd|SP>, <Xn|SP>, #<uimm6>, #<uimm4>`
- `STG <Xt>, [<Xn|SP>{, #<simm>}]`, signed offset.
- `STZG <Xt>, [<Xn|SP>{, #<simm>}]`, signed offset.
- `ST2G <Xt>, [<Xn|SP>{, #<simm>}]`, signed offset.
- `STZ2G <Xt>, [<Xn|SP>{, #<simm>}]`, signed offset.
- `LDG <Xt>, [<Xn|SP>{, #<simm>}]`
- `LDGM <Xt>, [<Xn|SP>]`
- `STGM <Xt>, [<Xn|SP>]`
- `STZGM <Xt>, [<Xn|SP>]`
- `STGP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]`

A64 instructions that are CONSTRAINED UNPREDICTABLE in Debug state

This subsection describes all instruction not listed in either:

- *A64 instructions that are changed in Debug state* on page H2-6937.
- *A64 instructions that are unchanged in Debug state* on page H2-6937.

These instructions are CONSTRAINED UNPREDICTABLE in Debug state. In general, the permissible behaviors are:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- If the instruction reads the PC or PSTATE, it uses an UNKNOWN value.
- If the instruction modifies the PC or PSTATE, other than by advancing the PC to the sequentially next instruction, it sets DLR_EL0 and DSPSR_EL0 to UNKNOWN values.
- If the instruction is similar to a Debug state instruction, it executes as that Debug state instruction.
- The instruction has the same behavior as in Non-debug state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following list shows the permissible behaviors for A64 instruction in Debug state. An instruction might appear multiple times in the list, in which case the choice of permissible behaviors is any of those listed. An example of this is `CCMP`.

Exception-generating instructions

These instructions are:

- `SVC`.
- `HVC`.
- `SMC`.
- `BRK`.
- `HLT`.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a `NOP`.
- `SVC` behaves as `DCPS1`.
- `HVC` behaves as `DCPS2`.
- `SMC` behaves as `DCPS3`.
- They generate the exception that the instruction would generate in Non-debug state. The exception is taken as described in *Exceptions in Debug state* on page H2-6957

  Note

  `SMC` must not generate a Secure Monitor Call exception from Non-secure state if EDSCR.SDD is set to 1.

Instructions that explicitly write to the PC (branches)

These instructions are:

- `B`, `B.cond`, `BL`, `BLR`, `BR`, `CBZ`, `CBNZ`, `RET`, `TBZ`, `TBNZ`.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.
- They execute as in Non-debug state without branching and set DSPSR_EL0 and DLR_EL0 to UNKNOWN values.

Exception return and related instructions

These instructions are:

- `ERET`.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a `NOP`.
- They execute as in Non-debug state without branching. They set DSPSR_EL0 and DLR_EL0 to UNKNOWN values, and either:
  - Execute the DRPS operation instead of performing an exception return, using UNKNOWN SPSR values.
  - Not change the Exception level.

Instructions that request entry to a low-power state

These instructions are:

- `WFE`, `WFI`.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a `NOP`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- They generate a synchronous exception if the corresponding instruction would be trapped in Non-debug state. See *Configurable instruction enables and disables, and trap controls* on page D1-2373.
- A `WFE` instruction clears the Event register if it is set.

Note

This means that these instructions must not suspend execution.

Instructions that read the PC

These instructions are:

- `LDR` (literal), `LDRSW` (literal).
- `ADR`, `ADRP`.
- `PRFM` (literal).

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a `NOP`.
- They execute as in Non-debug state, using an UNKNOWN value for the PC operand.

Instructions that explicitly modify **PSTATE, other than** `DCPS` **and** `DRPS`

These instructions are:

- `ADDS`, `SUBS`, `ADCS`, `SBCS`, `ANDS`, `BICS`, `CCMN`, `CCMP`.
- `FCMP`, `FCMPE`, `FCCMP`, `FCCMPE`.
- `MSR DAIFSet` (immediate), `MSR DAIFClr` (immediate), `MSR SPSel` (immediate).
- `MSR NZCV` (register), `MSR DAIF` (register), `MSR SPSel` (register).
- When ARMv8.1-PAN is implemented, `MSR PAN` (immediate) and `MSR PAN` (register).
- When ARMv8.2-UAO is implemented, `MSR UAO` (immediate) and `MSR UAO` (register).
- When ARMv8.4-CondM is implemented, `CFINV`, `RMIF`, `SETF8`, `SETF16`.
- When ARMv8.4-DIT is implemented, `MSR DIT`.
- When ARMv8.5-CondM is implemented, `AXFLAG` and `XAFLAG`.
- When ARMv8.5-MemTag is implemented, `MSR TCO`.
- When ARMv8.5-RNG is implemented, `MRS RNDR` and `MRS RNDRRS`.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a `NOP`.
- They execute as in Non-debug state, setting DSPSR_EL0 and DLR_EL0 to UNKNOWN values.

Instructions that read PSTATE.{N, Z, C, V} or other PSTATE fields

These instructions are:

- `CSEL`, `CSINC`, `CSINV`, `CSNEG`, `CCMN`, `CCMP`, `FCSEL`, `FCCMP`, `FCCMPE`.
- `ADC`, `ADCS`,`SBC`, `SBCS`.
- `CFINV`.
- `MRS NZCV`, `MRS DAIF`, `MRS SPSel`, `MRS CurrentEL`.
- When ARMv8.1-PAN is implemented, `MRS PAN`.
- When ARMv8.2-UAO is implemented, `MRS UAO`.
- When ARMv8.4-CondM is implemented, `CFINV`.
- When ARMv8.4-DIT is implemented, `MRS DIT`.
- When ARMv8.5-MemTag is implemented `MRS TCO`.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a `NOP`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- They execute as in Non-debug state:
  - For the conditional operations and those using the PSTATE.C flag as an input, these instructions use an UNKNOWN value for the Condition flag.
  - For the MRS instruction, they return an UNKNOWN value.

Hint instructions

These instructions are:

- When ARMv8.0-DGH is implemented, DGH.

These instructions behave in one of the following ways:

- They execute as a NOP.
- They execute as in Non-debug state.

All other instructions

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.
- They execute as in Non-debug state.

Note

This includes instructions defined as UNPREDICTABLE or CONSTRAINED UNPREDICTABLE in Non-debug state. These instructions are UNPREDICTABLE or CONSTRAINED UNPREDICTABLE in Debug state.

Executing T32 instructions in Debug state

The following sections describe the behavior of the T32 instructions in Debug state:

- *T32 instructions that are changed in Debug state*.
- *T32 instructions that are unchanged in Debug state* on page H2-6944.
- *T32 instructions that are CONSTRAINED UNPREDICTABLE in Debug state* on page H2-6946.

T32 instructions that are changed in Debug state

The following T32 instructions are defined in Debug state, but are undefined in Non-debug state:

- DCPS

  Note

  DCPS can be UNDEFINED in certain conditions in Debug state. See *DCPS<n>* on page H2-6954.

- MRC p15,3,<Rt>,c4,c5,0 (DSPSR).
- MCR p15,3,<Rt>,c4,c5,0 (DSPSR).
- MRC p15,3,<Rt>,c4,c5,1 (DLR).
- MCR p15,3,<Rt>,c4,c5,1 (DLR).

In addition, ERET executes the DRPS operation in Debug state.

For more information see *Debug state operations, DCPS, DRPS, MRS, MSR* on page H2-6954.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

T32 instructions that are unchanged in Debug state

The following list shows the instructions that are unchanged in Debug state. Any T32 instruction that uses the PC or APSR.{N, Z, C, V} as the source or destination register is not included in the list. Moreover, the list only includes the 32-bit T32 encodings.

Any instruction that is UNDEFINED in Non-debug state

The list of instructions:

- Excludes any instruction listed in *T32 instructions that are changed in Debug state* on page H2-6943.
- Excludes any instruction listed in *T32 instructions that are CONSTRAINED UNPREDICTABLE in Debug state* on page H2-6946 that is UNDEFINED because an enable or disable bit is not RES0 or RES1

Instructions that move System or Special-purpose registers to or from a general-purpose register

The list of instructions:

- Includes the instructions to transfer a general-purpose register to or from the DTR, which can be executed at any Exception level.
- Excludes APSR and CPSR access instructions.
- Excludes instructions for accessing banked registers for the current mode.

These instructions are:

- `MRS <banked_reg>, MSR <banked_reg>.`

  Note

  This does not apply to cases which are UNPREDICTABLE or CONSTRAINED UNPREDICTABLE in Non-debug state in the current mode.

- `MRC, MCR.`

  Note

  This includes all allocated System registers in the (`coproc==0b111x`) encoding space other than an `MRC` move to APSR_nzcv.

- `MRS SPSR`, `MSR SPSR`.
- `VMRS <vfp_system_reg>, VMSR <vfp_system_reg>`.

  Note

  This includes all allocated Advanced SIMD and floating-point System registers, other than an a `VMRS` move to APSR_nzcv.

Floating-point moves between a SIMD&FP register and a general-purpose register

These instructions are:

- `VMOV` (between a general-purpose register and a single-precision register).
- `VMOV` (between a general-purpose register and a doubleword floating-point register).

SIMD moves between a SIMD&FP register and a general-purpose register

These instructions are:

- `VMOV` (between a general-purpose register and a scalar).

Barriers

These instructions are:

- `CSDB`, when ARMv8.0-SSBS is implemented.
- `DMB`.
- `DSB`.
- `ESB`, when the RAS Extension is implemented.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- `ISB`.
- `PSSBB`, when ARMv8.0-SSBS is implemented.
- `SB`, when ARMv8.0-SB is implemented.
- `SSBB`, when ARMv8.0-SSBS is implemented.

Memory access instructions at various access sizes

The following constraints apply:

- General purpose-registers only.
- One of the following addressing modes:
  - Immediate (8-bit or 12-bit) offset.
  - Immediate (8-bit) post-indexed.
  - Immediate (8-bit) pre-indexed.
  - Unprivileged (8-bit).
- Not literal.
- One of the following types:
  - (Single) register.
  - Dual.
  - Exclusive.
  - Exclusive doubleword.
  - Acquire/Release.
  - Acquire/Release Exclusive.
  - Acquire/Release Exclusive doubleword.

These instructions are:

- `LDR.W, LDRB.W, LDRH.W, LDRD, LDRSB.W, LDRSH.W` (immediate, not literal).
- `LDRT, LDRBT, LDRHT, LDRSBT, LDRSHT` (immediate).
- `LDREX, LDREXB, LDREXH, LDA, LDAB, LDAH, LDAEX, LDAEXB, LDAEXH`.
- `LDREXD, LDAEXD`.
- `STR.W, STRB.W, STRH.W, STRD` (immediate).
- `STRT, STRBT, STRHT` (immediate).
- `STREX, STREXB, STREXH, STL, STLB, STLH, STLEX, STLEXB, STLEXH`.
- `STREXD, STLEXD`.

Move to general-purpose register

These instructions are:

- `MOVW`, `MOVT` (immediate).

System instructions, Send Event, `NOP`, and Clear Exclusive

The System instructions are Cache maintenance instructions, TLB maintenance instructions, and Address translation instructions. These are encoded in the (`coproc==0b1111`) System register encoding space.

These instructions are:

- ICIALLU, ICIALLUIS, ICIMVAU.
- DCCIMVAC, DCCISW, DCCMVAC, DCCMVAU, DCCSW, DCIMVAC, DCISW.
- TLBIALL, TLBIALLH, TLBIALLHIS, TLBIALLIS, TLBIALLNSNH, TLBIALLNSNHIS, TLBIASID, TLBIASIDIS, TLBIIPAS2, TLBIIPAS2IS, TLBIIPAS2L, TLBIIPAS2LIS, TLBIMVA, TLBIMVAA, TLBIMVAAIS, TLBIMVAAL, TLBIMVAALIS, TLBIMVAH, TLBIMVAHIS, TLBIMVAIS, TLBIMVAL, TLBIMVALH, TLBIMVALHIS, TLBIMVALIS.
- ATS12NSOPR, ATS12NSOPW, ATS12NSOUR, ATS12NSOUW, ATS1CPR, ATS1CPW. ATS1CUR, ATS1CUW, ATS1HR, ATS1HW.
- BPIALL, BPIALLIS, BPIMVA.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- SEV.W, SEVL.W.
- NOP.W.
- CLREX.

T32 instructions that are CONSTRAINED UNPREDICTABLE in Debug state

This subsection describes all instruction not listed in either:

- *T32 instructions that are changed in Debug state* on page H2-6943.
- *T32 instructions that are unchanged in Debug state* on page H2-6944.

These instructions are CONSTRAINED UNPREDICTABLE in Debug state. In general, the permissible behaviors are:

- The instruction generates an Undefined Instruction exception.
- The instruction executes as a NOP.
- If the instruction reads the PC or PSTATE, it uses an UNKNOWN value.
- If the instruction modifies the PC or PSTATE, other than by advancing the PC to the sequentially next instruction, it sets DLR and DSPSR to UNKNOWN values.
- If the instruction is similar to a Debug state instruction, it executes as that Debug state instruction.
- The instruction has the same behavior as in Non-debug state.

The following list shows the permissible behaviors for T32 instruction in Debug state. An instruction might appear multiple times in the list, in which case the choice of permissible behaviors is any of those listed.

Exception-generating instructions

These instructions are:

- SVC.
- HVC.
- SMC.
- UDF.
- BKPT.
- HLT.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.
- SVC behaves as DCPS1.
- HVC behaves as DCPS2.
- SMC behaves as DCPS3.
- They generate the exception the instruction would generate in Non-debug state. The exception is taken as described in *Exceptions in Debug state* on page H2-6957

Note

SMC must not generate a Secure Monitor Call exception from Non-secure state if EDSCR.SDD is set to 1.

Instructions that explicitly write to the PC (branches)

These instructions are:

- B, B (conditional), CBZ, CBNZ BL.
- BX, BLX (register or immediate).
- BXJ, TBB, TBH.
- MOV pc and related instructions.
- LDR pc, LDM (with a register list includes the PC), POP (with a register list that includes the PC).

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.
- They execute as in Non-debug state without branching and set DSPSR and DLR to UNKNOWN values.

Exception return and related instructions, other than ERET

These instructions are:

- SRS, RFE, SUBS pc, lr, and related instructions.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.
- They execute as in Non-debug state without branching, setting DLR and DSPSR to UNKNOWN values, and either:
  - Execute the DRPS operation instead of performing an exception return, using UNKNOWN SPSR values.
  - Not changing Exception level or PE mode.

Instructions that request entry to a low-power state

These instructions are:

- WFE, WFI.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.
- They generate a synchronous exception if the corresponding instruction would be trapped in Non-debug state. See *Configurable instruction enables and disables, and trap controls* on page G1-5778.
- A WFE instruction is permitted to clear the Event register if it is set.

Note

This means that these instructions must not suspend execution.

Instructions that read the PC

These instructions are:

- LDR (literal), LDRB (literal), LDRH (literal), LDRSB (literal), LDRSH (literal).
- ADR, ADRL, ADRH.
- PLD (literal), PLI (literal).

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.
- They execute as in Non-debug state using an UNKNOWN value for the PC operand.

Instructions that explicitly modify **PSTATE, other than** DCPS **and** ERET

These instructions are:

- CMP, TST, TEQ, CMN.
- <opc>S.
- MRC p14,0,APSR_nzcv,c0,c1,0 (accessing DBGDSCRint).
- CPS, SETEND, IT.
- MSR CPSR (immediate), MSR CPSR (register), MSR APSR (immediate), MSR APSR (register).
- VMRS APSR_nzcv,FPSCR.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- `QADD, QDADD, QSUB, QDSUB`.
- `SMLABB`, `SMLABT`, `SMLATB`, `SMLATT`, `SMLAD`, `SMLAWB`, `SMLAWT`, `SMLSD`, `SMUAD`.
- `SSAT, SSAT16, USAT, USAT16`.
- `SADD`, `SADD8`, `SADD16`, `SASX`, `SSAX`, `SSUB`, `SSUB8`, `SSUB16`.
- `UADD`, `UADD8`, `UADD16`, `UASX`, `USAX`, `USAUB`, `USUN8`, `USUB16`.
- When ARMv8.1-PAN is implemented, `SETPAN`.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a `NOP`.
- They execute as in Non-debug state, setting DSPSR_EL0 and DLR_EL0 to UNKNOWN values.

Instructions that read PSTATE.{N, Z, C, V} or other PSTATE fields

These instructions are:

- `SEL`, `VSEL`.
- `ADC`, `SBC`, all instructions with an `RRX` shift.
- `MRS CPSR`.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a `NOP`.
- They execute as in Non-debug state:
  - For the conditional operations and those using the PSTATE.C flag as an input, these instructions use an UNKNOWN value for the Condition flag.
  - For the `MRS` instruction, they return an UNKNOWN value

All other instructions

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a `NOP`.
- They have the same behavior as in Non-debug state.

Note

This includes instructions defined as UNPREDICTABLE or CONSTRAINED UNPREDICTABLE in Non-debug state. These instructions are CONSTRAINED UNPREDICTABLE in Debug state. This includes some T32 instructions that specify R15 as a destination or source register.

Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors* describes the CONSTRAINED UNPREDICTABLE behavior for these instructions. In Debug state these CONSTRAINED UNPREDICTABLE choices are further restricted:

- Instructions that specify R15 as a destination register:
  - Are not permitted to branch, because the architecture does not define a branch operation in Debug state.
  - Might set DLR and DSPSR to UNKNOWN values.
  - Might have any of the other permitted behaviors.
- Instructions that specify R15 as a source operand:
  - Cannot use PC + offset, because there is no architecturally-defined PC in Debug state.
  - Might have any of the other permitted behaviors, including using an UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H2.4.3 Decode tables

The syntax in the tables is defined as follows:

1 The bit has a fixed value of 1.

0 The bit has a fixed value of 0.

!= The field has any value other than the value or values specified. The field might be an encoding field in the instruction whose value is supplied by the debugger.

Note

The instruction encodings in Chapter C6 *A64 Base Instruction Descriptions* and Chapter F5 *T32 and A32 Base Instruction Set Instruction Descriptions* might show these bits as (0) or (1). A debugger must set these bits to 0 or 1, as appropriate.

Any other value indicates an encoding field in the instruction whose value is supplied by the debugger. Some values might be reserved or undefined, in which case the instruction is UNDEFINED or CONSTRAINED UNPREDICTABLE in Debug state, as it is in Non-debug state.

For more information about the instruction encodings, see:

- Chapter C6 *A64 Base Instruction Descriptions*.
- Chapter F5 *T32 and A32 Base Instruction Set Instruction Descriptions*.

For information about the syntax used in Table H2-2, Table H2-3, Table H2-4 on page H2-6950, and Table H2-5 on page H2-6951, see:

- *Common syntax terms* on page C1-183.
- *Assembler symbols* on page F2-4077.

Table H2-2 shows the A64 instructions that are modified in Debug state. For details of how these are packed in the EDITR see the register description.

Table H2-2 Modified A64 instructions in Debug state

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	Description
1 1 0 1	0 1 0 0	1 0 1 imm16				0	0 0 !=00	DCPS<opt>
1 1 0 1	0 1 0 1	0 0 L 1	1 0 1 1	0 1 0 0	0 1 0 1	0 0 0 Rt		MRS|MSR accessing DSPSR_EL0
1 1 0 1	0 1 0 1	0 0 L 1	1 0 1 1	0 1 0 0	0 1 0 1	0 0 1 Rt		MRS|MSR accessing DLR_EL0
1 1 0 1	0 1 1 0	1 0 1 1	1 1 1 1	0 0 0 0	0 0 1 1	1 1 1 0	0 0 0 0	DRPS

Table H2-3 show the T32 instructions that are modified in Debug state, with the first halfword on the left side and the second halfword on the right side. For details of how these are packed in the EDITR see the register description.

Table H2-3 Modified T32 instructions in Debug state

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	Description
1 1 1 0	1 1 1 0	0 1 1 L	0 1 0 0	!=1111	1 1 1 1	0 0 0 1	0 1 0 1	MRC|MCR accessing DSPSR
1 1 1 0	1 1 1 0	0 1 1 L	0 1 0 0	!=1111	1 1 1 1	0 0 1 1	0 1 0 1	MRC|MCR accessing DLR
1 1 1 1	0 0 1 1	1 1 0 1	1 1 1 0	1 0 0 0	1 1 1 1	0 0 0 0	0 0 0 0	ERET
1 1 1 1	0 1 1 1	1 0 0 0	1 1 1 1	1 0 0 0	0 0 0 0	0 0 0 0	0 0 !=00	DCPS<opt>

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table H2-4 lists the A64 instructions that are unchanged in Debug state, other than some unallocated and UNDEFINED instructions.

Table H2-4 A64 instructions that are unchanged in Debug state

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Description
sf	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	Rd					MOV <Rn>,SP
sf	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Rn					1	1	1	1	1	MOV SP,<Rn>
1	0	0	1	1	0	1	0	1	1	0	Rm					0	0	1	1	0	0	Rn					Rd					PACGA
sf	!=01		1	0	0	1	0	1	hw		imm16																Rd					MOVN, MOVK, MOVZ
1	1	0	1	0	1	0	1	0	0	0	0	0	1	1		0	0	1	0	0	0	0	0	0	0	0	1	1	1	1	1	NOP
1	1	0	1	0	1	0	1	0	0	0	0	0	1	1		0	0	1	0	0	0	0	0	1	0	L	1	1	1	1	1	SEV, SEVL
1	1	0	1	0	1	0	1	0	0	0	0	0	1	1		0	0	1	0	0	0	0	0	1	1	1	1	1	1	1	1	XPACLRI
1	1	0	1	0	1	0	1	0	0	0	0	0	1	1		0	0	1	0	0	0	0	1	op2			1	1	1	1	1	PAC(IA|IB)1716, AUT(IA|IB)1716
1	1	0	1	0	1	0	1	0	0	0	0	0	1	1		0	0	1	0	0	0	1	0	!= 010			1	1	1	1	1	CSDB, ESB, PSB
1	1	0	1	0	1	0	1	0	0	0	0	0	1	1		0	0	1	0	0	0	1	1	op2			1	1	1	1	1	PAC(IA|IB)(Z|SP), AUT(IA|IB)(Z|SP)
1	1	0	1	0	1	0	1	0	0	0	0	0	1	1		0	0	1	1	CRm				0	1	0	1	1	1	1	1	CLREX
1	1	0	1	0	1	0	1	0	0	0	0	0	1	1		0	0	1	1	option				1	op2		1	1	1	1	1	DSB, DMB, ISB, SB, SSBB, PSSBB
1	1	0	1	0	1	0	1	0	0	0	0	1	op1			CRn				CRm				op2			Rt					IC, DC, TLBI, AT
1	1	0	1	0	1	0	1	0	0	L	1	0	op1			CRn				CRm				op2			Rt					MRS|MSR accessing System register
1	1	0	1	0	1	0	1	0	0	L	1	1	op1			!=0100				CRm				op2			Rt					MRS|MSR accessing System register
1	1	0	1	0	1	0	1	0	0	L	1	1	op1			0	1	0	0	!=0010				op2			Rt					MRS|MSR accessing Special-purpose register
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	0	0	opc			Rn					Rd					PAC(IA|IB|DA|DB), AUT(IA|IB|DA|DB)
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	0	1	opc			1	1	1	1	1	Rd					PAC(IZA|IZB|DZA|DZB), AUT(IZA|IZB|DZA|DZB)
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	1	0	0	0	op	1	1	1	1	1	Rd					XPAC(I|D)
size			0	0	1	0	0	0	o2	L	0	Rs				o0	Rt2					Rn					Rt					LD(A|LA|X|AX)R{B|H}, ST(L|LL|X|LX)R{B|H}, CAS{A|L|AL}{B|H}
size			0	0	1	0	0	0	o2	L	1	Rs				o0	Rt2					Rn					Rt					LD{A}XP, ST{L}XP, CASP{A|L|AL}
!=11			1	1	1	0	0	0	opc		0	imm9								0	0	Rn					Rt					LDUR{B|H|SB|SH|SW}, STUR{B|H}
1	1		1	1	1	0	0	0	!=10		0	imm9								0	0	Rn					Rt					LDUR, STUR
size			1	1	1	0	0	0	opc		0	imm9								1	0	Rn					Rt					LDTR{B|H|SB|SH|SW}, STTR{B|H}
size			1	1	1	0	0	0	opc		0	imm9								P	1	Rn					Rt					LDR{B|H|SB|SH|SW}, STR{B|H}
size			1	1	1	0	0	0	A	R	1	Rs				0	opc			0	0	Rn					Rt					LD<op>{A|L|AL}{B|H}, ST<op>{A|L|AL}{B|H}
size			1	1	1	0	0	0	A	R	1	Rs				1	0	0	0	0	0	Rn					Rt					SWP{A|L|AL}{B|H}
0	1	0	0	1	1	1	0	0	0	0	imm5					0	0	0	1	1	1	Rn					Rd					INS <Vd>.<Ts>[<index>],<R><n>
0	Q	0	0	1	1	1	0	0	0	0	imm5					0	0	1	1	1	1	Rn					Rd					UMOV <R><d>,<Vn>.<Ts>[<index>]
0	0	0	1	1	1	1	0	0	0	1	0	0	1	1	op	0	0	0	0	0	0	Rn					Rd					FMOV <Sd>,<Wn>, FMOV <Wd>,<Sn>
1	0	0	1	1	1	1	0	0	1	1	0	0	1	1	op	0	0	0	0	0	0	Rn					Rd					FMOV <Dd>,<Xn>, FMOV <Xd>,<Dn>
1	0	0	1	1	1	1	0	1	0	1	0	1	1	1	op	0	0	0	0	0	0	Rn					Rd					FMOV <Vd>.D[1],<Xn> FMOV <Xd>,<Vn>.D[1]
1	0	0	1	0	0	0	1	1	0	uimm6					(0)	(0)uimm4						Xn					Xd					ADDG <Xd|SP>, <XN|SP>, #<uimm6>, #<uimm4>

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table H2-4 A64 instructions that are unchanged in Debug state (continued)

31 30 29 28	27 26	25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	Description
1 1 0 1	1 0	0 1	0 0 1	imm9		1 0	Xn	Xd	STG <Xt|SP>, [<Xn|SP>{, #<simm>}]!, Signed offset
1 1 0 1	1 0	0 1	1 0 1	0 0 0 0	0 0 0 0	0 0	Xn	Xd	STGM <Xt>, [<Xt|SP>]
1 1 0 1	1 0	0 1	1 0 1	imm9		1 1	Xn	Xd	ST2G <Xt|>, [Xt|SP>{, #<simm>}] Signed offset
1 1 0 1	1 0	0 1	0 1 1	imm9		1 1	Xn	Xd	STZG <XT|SP>, [<Xn|SP{. #<simm>}]! Signed offset
1 1 0 1	1 0	0 1	1 1 1	imm9		1 0	Xn	Xd	STZ2G <XT|SP>, [<Xn|SP{. #<simm>}]! Signed offset
1 1 0 1	1 0	0 1	0 0 1	0 0 0 0	0 0 0 0	0 0	Xn	Xd	STZGM, <Xt>, [Xn|SP>]
0 1 1 0	1 0	0 1	0 0	simm7	Xt2		Xn	Xt	STGP <xt1>, <Xt2>, [<Xn|SP>{, #<imm>}] Signed offset
1 1 0 1	0 0	0 1	1 0	uimm6	op3	uimm4	Xn	Xd	SUBG <Xd|SP>, <Xn|SP>, #<uimm6>, #<uimm4>
1 1 0 1	1 0	0 1	0 1 1	imm9		0 0	Xn	Xd	LDG <Xt>, [<Xn|SP>{, #<simm>}]
1 1 0 1	1 0	0 1	1 1 1	0 0 0 0	0 0 0 0	0 0	Xn	Xd	LDGM <Xt>,[<Xn|SP>]

Table H2-5 lists the T32 instructions that are unchanged in Debug state, other than some unallocated and UNDEFINED instructions. It shows the T32 instructions with the first halfword on the left side and the second halfword on the right side.

Table H2-5 T32 instructions that are unchanged in Debug state

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	Description
1 1 1 0	1 1 0 0	0 1 0 op	!=1111	!=1111	1 0 1 1	0 0 M 1	Vm	VMOV <Dm>,<Rt>,<Rt2> VMOV <Rt>,<Rt2>,<Dm>
1 1 1 0	1 1 1 0	0 0 0 op	Vn	!=1111	1 0 1 0	N 0 0 1	0 0 0 0	VMOV <Sn>,<Rt>, VMOV <Rt>,<Sn>
1 1 1 0	1 1 1 0	0 opc 0	Vd	!=1111	1 0 1 1	D opc2 1	0 0 0 0	VMOV.<size> <Dd>[<x>],<Rt>
1 1 1 0	1 1 1 0	U opc 1	Vn	!=1111	1 0 1 1	D opc2 1	0 0 0 0	VMOV.<dt> <Rt>,<Dd>[<x>]
1 1 1 0	1 1 1 0	1 1 1 op	reg	!=1111	1 0 1 0	0 0 0 1	0 0 0 0	VMRS, VMSR
1 1 1 0	1 1 0 0	0 1 0 op	!=1111	!=1111	1 1 1 cp	opc1	CRm	MCRR|MRRC accessing System registers
1 1 1 0	1 1 1 0	opc1 op	CRn	!=1111	1 1 1 cp	opc2 1	CRm	MCR|MRC accessing System registers
1 1 1 0	1 0 0 0	0 1 0 L	!=1111	!=1111	Rd	imm8		LDREX, STREX
1 1 1 0	1 0 0 0	1 1 0 L	!=1111	!=1111	Rt2	0 1	!=10 Rd	LDREX(B|H|D), STREX(B|H|D)
1 1 1 0	1 0 0 0	1 1 0 L	!=1111	!=1111	Rt2	1	op3 Rd	LDA{EX}{B|H|D}, STL{EX}{B|H|D}
1 1 1 0	1 0 0	!=0x10 !=xx0x L	!=1111	!=1111	!=1111	imm8		LDRD, STRD
1 1 1 1	0 i 1 0	T 1 0 0	imm4	0 imm3	!=1111	imm8		MOVW, MOVT
1 1 1 1	0 0 1 1	1 0 0 R	!=1111	1 0 0 0	M1	0 0 1 M	0 0 0 0	MSR <spec_reg><mode>,<Rn>
1 1 1 1	0 0 1 1	1 0 0 1	!=1111	1 0 0 0	1 1 1 1	0 0 0 0	0 0 0 0	MSR SPSR, <Rn>
1 1 1 1	0 0 1 1	1 0 1 0	1 1 1 1	1 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	NOP.W
1 1 1 1	0 0 1 1	1 0 1 0	1 1 1 1	1 0 0 0	0 0 0 0	0 0 0 0	0 1 0 L	SEV.W, SEVL.W
1 1 1 1	0 0 1 1	1 0 1 0	1 1 1 1	1 0 0 0	0 0 0 0	0 0 0 1	0 op 0 0	ESB, CSDB
1 1 1 1	0 0 1 1	1 0 1 1	1 1 1 1	1 0 0 0	1 1 1 1	0 0 1 0	1 1 1 1	CLREX
1 1 1 1	0 0 1 1	1 0 1 1	1 1 1 1	1 0 0 0	1 1 1 1	0 1 op	option	DSB, DMB, ISB, SSBB, PSSBB, SB

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table H2-5 T32 instructions that are unchanged in Debug state (continued)

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	Description
1 1 1 1	0 0 1 1	1 1 1 R	M1	1 0 0 0	!=1111	0 0 1 M	0 0 0 0	`MRS <Rd>,<spec_reg><mode>`
1 1 1 1	0 0 1 1	1 1 1 1	1 1 1 1	1 0 0 0	!=1111	0 0 0 0	0 0 0 0	`MRS <Rd>,SPSR`
1 1 1 1	1 0 0 0	1 !=11	0 !=1111	!=1111	imm12			`STR{B|H}.W` (12-bit immediate)
1 1 1 1	1 0 0 0	0 !=11	0 !=1111	!=1111	1 !=000	imm8		`STR{B|H|}{T}` (8-bit immediate)
1 1 1 1	1 0 0 S	1 !=11	1 !=1111	!=1111	imm12			`LDR{SB|SH|B|H}.W` (12-bit immediate)
1 1 1 1	1 0 0 S	0 !=11	1 !=1111	!=1111	1 !=000	imm8		`LDR{SB|SH|B|H}{T}` (8-bit immediate)

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H2.4.4 Security in Debug state

If EL3 is implemented or the implemented Security state is Secure state, security in Debug state is governed by the Secure debug disabled flag, EDSCR.SDD.

On entry to Debug state

If entering in Secure state, EDSCR.SDD is set to 0. Otherwise EDSCR.SDD is set to the inverse of `ExternalSecureInvasiveDebugEnabled()`. That is:

- If `ExternalSecureInvasiveDebugEnabled()` == TRUE, EDSCR.SDD is set to 0.
- If `ExternalSecureInvasiveDebugEnabled()` == FALSE, EDSCR.SDD is set to 1.

Note

Normally, if `ExternalSecureInvasiveDebugEnabled()` == FALSE then halting is prohibited and it is not possible to enter Debug state from Secure state. However, because changes to the authentication signals require a *Context synchronization event* to guarantee their effect, there is a period during which the PE might halt even though the authentication signals prohibit halting.

In Debug state

The value of EDSCR.SDD does not change, even if `ExternalSecureInvasiveDebugEnabled()` changes.

Note

- DBGAUTHSTATUS_EL1.{SNID, SID, NSNID, NSID} are not frozen in Debug state.
- If EDSCR.SDD set to 1 in Debug state, then there is no means to enter Secure state from Non-secure state. In this case it is impossible for the PE to be in Secure state. This is a general principle of behavior in Debug state.

In Non-debug state

EDSCR.SDD returns the inverse of `ExternalSecureInvasiveDebugEnabled()`. If the authentication signals that control `ExternalSecureInvasiveDebugEnabled()` change, a *Context synchronization event* is required to guarantee their effect.

Note

- In Non-debug state, EDSCR.SDD is unaffected by the Security state of the PE.
- A *Context synchronization event* is also required to guarantee that changes in the authentication signals are visible in DBGAUTHSTATUS_EL1.{SNID, SID, NSNID, NSID}.

If EL3 is not implemented and the implemented Security state is Non-Secure state, EDSCR.SDD is RES1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H2.4.5 Privilege in Debug state

The only additional privileges offered to Debug state are:

- The privilege to execute *Debug state operations, DCPS, DRPS, MRS, MSR*.
- The privilege to execute DTR access instructions regardless of the Exception level and traps.

The DTR access instructions can be executed at any Exception level, including EL0, regardless of any control register settings that might force these instructions to be undefined or trapped in Non-debug state. These instruction are:

- The `MRS` and `MSR` instructions that access DBGDTR_EL0, DBGDTRTX_EL0, and DBGDTRRX_EL0 in AArch64 state.
- The `MRC` and `MCR` instructions that access DBGDTRTXint and DBGDTRRXint in AArch32 state.

All other instructions operate with the privilege determined by the current Exception level and security state. This applies to all Special-purpose and System registers accesses, memory accesses, and undefined instructions, and includes generating exceptions when the System registers trap or disable an instruction.

H2.4.6 Debug state operations, DCPS, DRPS, MRS, MSR

Armv8 defines operations to change between Exception levels in Debug state. These operations can also change the mode at the current Exception level.

DCPS<n>

Executing a `DCPS<n>` instruction in Debug state moves the PE to a higher Exception level or to a specific mode at the current Exception level.

If the `DCPS<n>` instruction is executed in AArch32 state and the target Exception level is using AArch64:

- The current instruction set switches from T32 to A64.
- The effect on registers that are not visible or only partially visible in AArch32 state is the same as for system calls in Non-debug state. See *Execution state* on page D1-2317.

Otherwise, the instruction set state does not change.

If the target Exception level is the same as the current Exception level, then the PE does not change Exception level. However, the PE might change mode.

The effect on endianness is the same as for exceptions and exception returns in Non-debug state:

- In AArch64 state the current endianness is determined by the value of SCTLR_ELx.EE for the target Exception level.
- In AArch32 state the current endianness is determined by the value of SCTLR.EE or HSCTLR.EE for the target Exception level.

The `DCPS<n>` instructions are:

In AArch64 state

- `DCPS1`
- `DCPS2`
- `DCPS3`

In AArch32 state, in the T32 instruction set only

- `DCPS1`
- `DCPS2`
- `DCPS3`

The `DCPS` instructions are undefined in Non-debug state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table H2-6 shows the target of the instruction. In Table H2-6 the entries have the following meaning:

EL1h/Svc This means that the target is:
- EL1h if EL1 is using AArch64.
- EL1 and Supervisor mode if EL1 is using AArch32.

EL2h/Hyp This means that the target is:
- EL2h if EL2 is using AArch64.
- EL2 and Hyp mode if EL2 is using AArch32.

EL3h/Monitor This means that the target is:
- EL3h if EL3 is using AArch64.
- EL3 and Monitor mode if EL3 is using AArch32.

Svc Secure Supervisor mode, in EL3 using AArch32.

Monitor Secure Monitor mode, in EL3 using AArch32.

Table H2-6 Target for DCPS instructions in Debug state

Instruction	Target when DCPS instruction executed at stated Exception level:				
	EL0	EL1	EL2	EL3 (AArch64)	EL3 (AArch32)
DCPS1	EL1h/Svc	EL1h/Svc	EL2h/Hyp	EL3h	Svc, clears SCR.NS to 0
DCPS2	EL2h/Hyp	EL2h/Hyp	EL2h/Hyp	EL3h	UNDEFINED
DCPS3	EL3h/Monitor	EL3h/Monitor	EL3h/Monitor	EL3h	Monitor, clears SCR.NS to 0

In AArch32 Monitor mode, DCPS1 and DCPS3 clear SCR.NS to 0.

Note

In AArch64 state, at EL3, DCPS<n> does not change SCR_EL3.NS.

However:

- DCPS1 is undefined at EL0 if either:
  - EL2 is implemented and enabled in the current Security state, and is using AArch64 and HCR_EL2.TGE == 1.
  - In Non-secure state, EL2 is implemented and using AArch32 and HCR.TGE == 1.
- DCPS2 is undefined at all Exception levels if EL2 is not implemented.
- DCPS2 is undefined at the following Exception levels if EL2 is implemented:
  - At EL0 and EL1 in Secure state if EL2 is disabled in the current Security state.
  - At EL3 if EL3 is using AArch32.
- DCPS3 is undefined at all Exception levels if either:
  - EDSCR.SDD == 1.
  - EL3 is not implemented.

Note

The references to DCPS1, DCPS2, and DCPS3 in this section link to the descriptions of the instructions in the A64 instruction set. The DCPS<n> instructions are also defined in the T32 instruction set, see DCPS1, DCPS2, DCPS3. These instructions are not defined in the A32 instruction set, because A32 instructions cannot be executed in Debug state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

On executing a DCPS instruction:

- If the target Exception level is using AArch64:
  - ELR_ELx of the target Exception level becomes UNKNOWN.
  - SPSR_ELx of the target Exception level becomes UNKNOWN.
  - ESR_ELx of the target Exception level becomes UNKNOWN.
  - DLR_EL0 and DSPSR_EL0 become UNKNOWN.
- If the target Exception level is using AArch32 DLR and DSPSR become UNKNOWN and:
  - If the target Exception level is EL1 or EL3, the LR and SPSR of the target mode become UNKNOWN.
  - If the target Exception level is EL2, then ELR_hyp, SPSR_hyp, and HSR become UNKNOWN.

If the target Exception level is using AArch32, and the target Exception level is EL1 or EL3, the LR and SPSR of the target mode become UNKNOWN.

If ARMv8.0-SSBS is implemented, the DCPS<n> instruction leaves the PSTATE.SSBS bit UNKNOWN.

The DCPSInstruction() function is described in Chapter J1 *Armv8 Pseudocode*.

DRPS

Executing the DRPS operation in Debug state moves the PE to a lower Exception level, or to another PE mode at the current Exception level, by copying the current SPSR to PSTATE.

If DRPS is executed in AArch64 state and the target Exception level is using AArch32:

- The current instruction set switches from A64 to T32.
- The effect on registers that are not visible or only partially visible in AArch32 state is the same as for exception returns in Non-debug state. See *Execution state* on page D1-2317.

Otherwise, the instruction set state does not change.

If the target Exception level is the same as the current Exception level, then the PE does not change Exception level. However, the PE might change mode.

The effect on endianness is the same as for exceptions and exception returns in Non-debug state:

- If targeting an Exception level using AArch64, current endianness is set according to SCTLR_ELx.EE, or SCTLR_EL1.E0E for the target Exception level.
- If targeting an Exception level using AArch32, current endianness is set by SPSR.E as appropriate.

The DRPS instructions are:

In AArch64 state

- DRPS

In AArch32 state, in the T32 instruction set only

- ERET

If the SPSR specifies an illegal exception return, then PSTATE.{M, nRW, EL, SP} are unchanged and PSTATE.IL is set to 1. For further information on illegal exception returns, see *Illegal return events from AArch64 state* on page D1-2351.

PSTATE.{N, Z, C, V, Q, GE, IT, T, SS, D, A, I, F} are ignored in Debug state. This means that the effect of the DRPS operation on these fields is to set them to an UNKNOWN value that might be the value from the SPSR. For more information see *PSTATE in Debug state* on page H2-6936.

All other PSTATE fields are copied from SPSR.

DRPS is undefined at EL0 and in Non-debug state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

Unlike an exception return, the DRPS operation has no architecturally-defined effect on the Event Register and Exclusives monitors. DRPS might set the Event Register or clear the Exclusives monitors, or both, but this is not a requirement and debuggers must not rely on any implementation specific behavior.

On executing a DRPS instruction:

- If the target Exception level is using AArch64:
  - DLR_EL0 and DSPSR_EL0 become UNKNOWN.
- If the target Exception level is using AArch32:
  - DLR and DSPSR become UNKNOWN.

If ARMv8.0-SSBS is implemented, the DRPS instruction leaves the PSTATE.SSBS bit UNKNOWN.

The DRPSInstruction() function is described in Chapter J1 *Armv8 Pseudocode*.

MRS and MSR

The other Debug state instructions are used to read or write DLR_EL0 and DSPSR_EL0.

These instructions are:

In AArch64 state

- MRS
- MSR (register)

In AArch32 state

- MRC
- MCR

```
MRS <Xt>, DLR_EL0      ; Copy DLR_EL0 to <Xt>
MRS <Xt>, DSPSR_EL0    ; Copy DSPSR_EL0 to <Xt>
MSR DLR_EL0, <Xt>      ; Copy <Xt> to DLR_EL0
MSR DSPSR_EL0, <Xt>    ; Copy <Xt> to DSPSR_EL0
```

These instructions can be executed at any Exception level when in Debug state, including EL0. They are undefined in Non-debug state.

H2.4.7 Exceptions in Debug state

The following sections describe how exceptions are handled in Debug state:

- *Generating exceptions when in Debug state*.
- *Taking exceptions when in Debug state* on page H2-6958.
- *Reset in Debug state* on page H2-6959.

Generating exceptions when in Debug state

In Debug state:

- Instruction Abort exceptions cannot happen because instructions are not fetched from memory.
- Interrupts, including SError and virtual interrupts are ignored and remain pending:
  - The pending interrupt remains visible in ISR.
- Debug exceptions and debug events are ignored.
- SCR.EA is treated as if it were set to 0, regardless of its actual state, other than for the purpose of reading the bit.
- Any attempt to execute an instruction bit pattern that is an allocated instruction at the current Exception level, but is listed in *Executing instructions in Debug state* on page H2-6937 as undefined in Debug state, generates an exception, that is taken to the current Exception level, or to EL1 if executing at EL0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

If the exception is taken to an Exception level that is using AArch32 then it is taken as an Undefined Instruction exception.

The priority and syndrome for these exceptions is the same as for executing an encoding that does not have an allocated instruction.

- Instructions executed at EL2, EL1 and EL0 that are configured by EL3 control registers to trap to EL3:
  - When the value of EDSCR.SDD is 0, generate the appropriate trap exception that is taken to EL3.
  - When the value of EDSCR.SDD is 1, are treated as UNDEFINED and generate an exception that is taken to the current Exception level, or to EL1 if the instruction is executed at EL0. If the exception is taken to an Exception level that is using AArch32 it is taken as an Undefined Instruction exception.

    If the exception is taken to an Exception level using AArch64 or to AArch32 Hyp mode, then it is reported with an EC value of `0x00`.

  Otherwise configurable traps, enables, and disables for instructions are unaffected by Debug state, and executing an affected instruction generates the appropriate exception.

Otherwise, synchronous exceptions, including Data Aborts, are generated as they would be in Non-debug state and taken to the appropriate Exception level in Debug state.

Note

If EDSCR.SDD == 1 then an exception from Non-secure state is never taken to Secure state. See *Security in Debug state* on page H2-6953.

Taking exceptions when in Debug state

When the PE is in Debug state, all exceptions are synchronous. When an exception is generated, it is taken to Debug state. This means that:

- The target Exception level is as defined for the exception in Non-debug state.
- If the target Exception level is using AArch32 then the target PE mode is as defined for the exception in Non-debug state.
- The exception syndrome is reported as defined for the exception in Non-debug state, except for the case described in *Data Aborts in Memory access mode* on page H4-6996 for which the reporting requirements are relaxed.

  The exception syndrome is reported using the syndrome register or registers for the target Exception level. In AArch64 state, these are ESR_ELx, and FAR_ELx. In AArch32 state, these are DFSR, DFAR, HSR, HDFAR, and HPFAR. For example:
  - If a Data Abort exception is taken to Abort mode at EL1 or EL3 and the exception is taken from AArch32 state and using the Short-descriptor translation table format, the DFSR reports the exception using the Short-descriptor format fault encoding. For exceptions other than Data Abort exceptions taken to Abort mode, DFSR is not updated.
  - If an instruction is trapped to an Exception level using AArch64 due to a configurable trap, disable, or enable, the exception code reported is the same as it would be in Non-debug state.

  The effect on auxiliary syndrome registers, such as AFSR, is IMPLEMENTATION DEFINED.

  Note

  Generally, the AArch32 Fault Address Registers (FARs) and Fault Status Registers (FSRs) are not described as *syndrome registers*, although the term is appropriate to their function.

- The PE remains in Debug state and changes to the target mode.
- If EL3 is using AArch32 and the exception is taken from Monitor mode, SCR.NS is cleared to 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If the exception is taken to an Exception level using AArch32, the PE continues to execute T32 instructions, regardless of the TE bit in the System register for the target Exception level.
- The endianness switches to that indicated by the EE bit of the System register for the target Exception level.
- The SPSR for the target Exception level or mode is corrupted and becomes UNKNOWN.
- If the target Exception level is using AArch64, ELR_ELx for the target Exception level becomes unknown.
- If the target Exception level is EL2 using AArch32, ELR_hyp becomes unknown.
- If the target Exception level is EL1 or EL3 using AArch32, LR_<mode> for the target mode becomes unknown.
- DLR and DSPSR become UNKNOWN.
- The cumulative error flag, EDSCR.ERR, is set to 1. See *Cumulative error flag* on page H4-7000.
- PSTATE.IL is cleared to 0.
- PSTATE.{IT, T, SS, D, A, I, F} are set to UNKNOWN values, and PSTATE.{N, Z, C, V, Q, GE} are unchanged. However, these fields are ignored and are not observable in Debug state. For more information see *PSTATE in Debug state* on page H2-6936.

The debugger must save any state that can be corrupted by an exception before executing an instruction that might generate another exception.

Pseudocode description of taking exceptions in Debug state

The pseudocode function `AArch64.TakeException()` shows the behavior when the PE takes an exception to an Exception level using AArch64 in Non-debug state. In Debug state, this is replaced with the function `AArch64.TakeExceptionInDebugState()`.

The pseudocode functions `AArch32.EnterMode()`, `AArch32.EnterHypMode()`, and `AArch32.EnterMonitorMode()` show the behavior when the PE takes an exception to an Exception level using AArch32 in Non-debug state. In Debug state:

- `AArch32.EnterMode()` is replaced with the function `AArch32.EnterModeInDebugState()`.
- `AArch32.EnterHypMode()` is replaced with the function `AArch32.EnterHypModeInDebugState()`.
- `AArch32.EnterMonitorMode()` is replaced with `AArch32.EnterMonitorModeInDebugState()`.

Reset in Debug state

If the PE is reset when in Debug state, it exits Debug state and enters Non-debug reset state. When the PE is in reset state, EDSCR.STATUS == `0b000010` and writes to EDITR are ignored.

Note

If EDECR.RCE == 1 or CTIDEVCTL.RCE ==1, meaning that a Reset Catch debug event is programmed, and if halting is allowed on exiting reset state, then on exiting reset state the PE halts and re-enters Debug state. See *Reset Catch debug events* on page H3-6985. All PE registers have taken their reset values, which might be UNKNOWN.

H2.4.8 Accessing registers in Debug state

Register accesses are unchanged in Debug state. The view of each register is determined by either the current Exception level or the mode, or both, and accesses might be disabled or trapped by controls at a higher Exception level.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

General-purpose register access, other than AArch64 state SP access

A single general-purpose register can be read by issuing an `MSR` instruction through the ITR to write DBGDTR_EL0 in AArch64 state, or an `MCR` instruction through the ITR to write DBGDTRTXint in AArch32 state. The debugger can then read the DTR register or registers through the external debug interface. The reverse sequence writes to a general-purpose register.

Figure H2-1 shows the reading and writing of general-purpose registers, other than SP, in Debug state in AArch64 state.

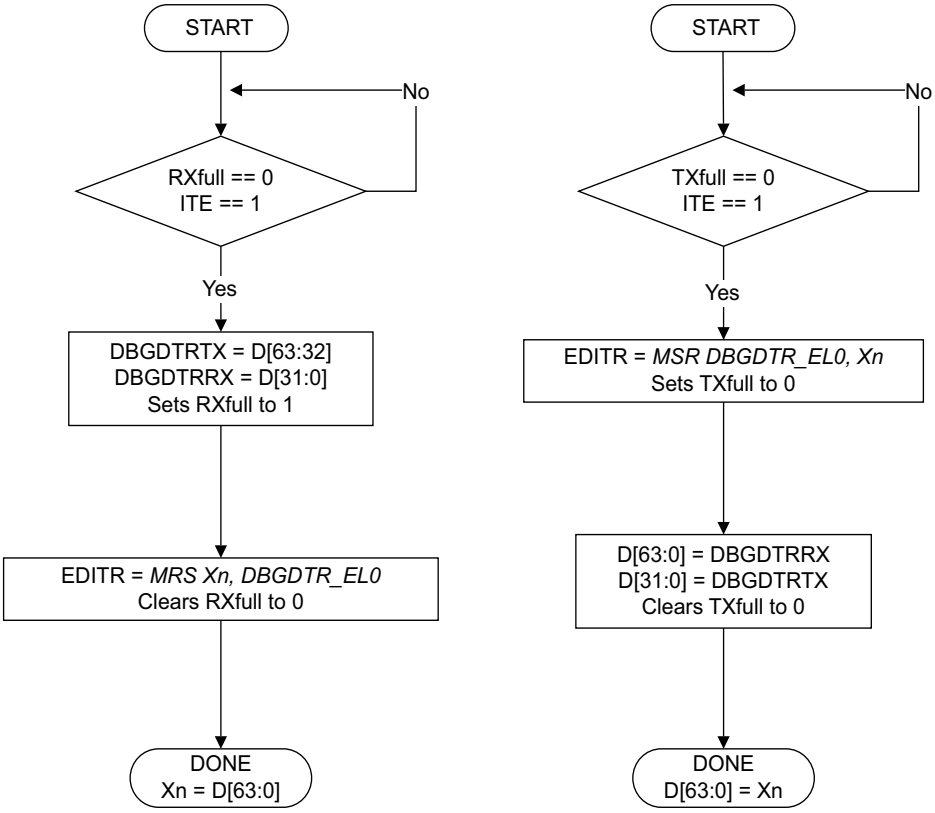


Figure H2-1 Reading and writing general-purpose registers, other than SP, in Debug state in AArch64 state

Figure H2-2 on page H2-6961 shows the reading and writing of general-purpose registers in Debug state in AArch32 state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

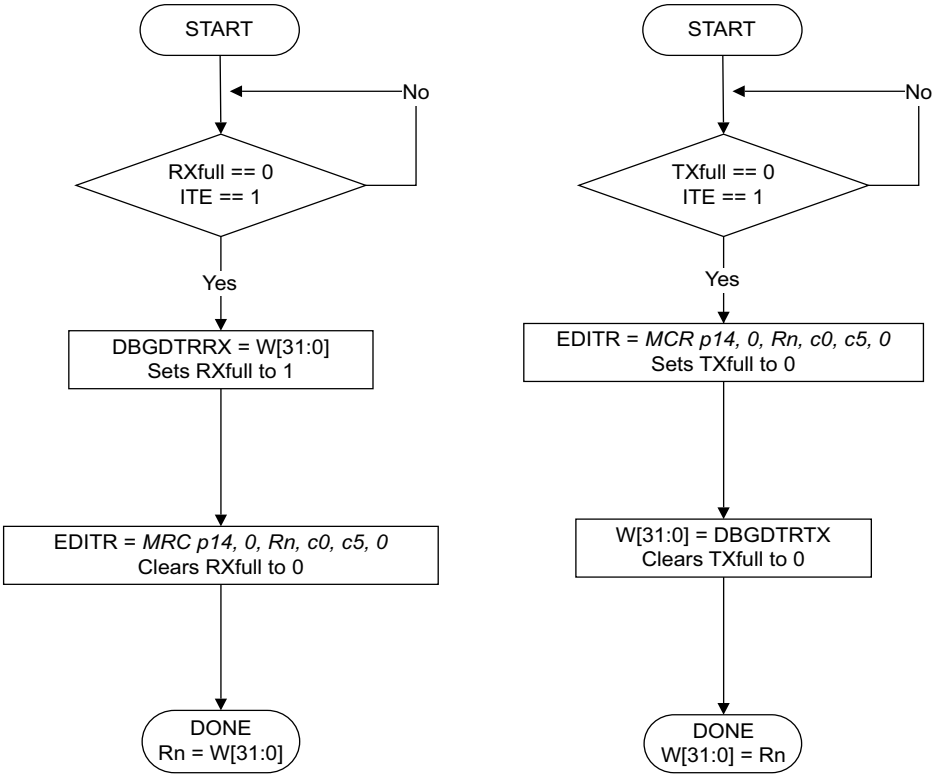


Figure H2-2 Reading and writing general-purpose registers in Debug state in AArch32 state

SIMD and floating-point register, System register, and AArch64 state SP accesses

To read a SIMD and floating-point register or a System register, the debugger must first copy the value into a general-purpose register using:

- An `FMOV` instruction in AArch64 state or a `VMOV` instruction in AArch32 state for floating-point transfers to SIMD and FP registers.
- A `UMOV` instruction in AArch64 state or a `VMOV` instruction in AArch32 state for SIMD transfers to SIMD and FP registers.
- An `MRS` instruction in AArch64 state or an `MRC` instruction in AArch32 state for System registers.
- A `MOV Xd,SP` instruction for the SP register in AArch64 state.

The debugger can then read out the particular general-purpose register. The reverse sequence writes a register.

PC and PSTATE access

The debugger reads the program counter and PSTATE of the process being debugged through the DLR_EL0 and DSPSR_EL0 System registers. The actual values of PC and PSTATE cannot be directly observed in Debug state:

- Instructions that are used for direct reads and writes of PC and PSTATE in Non-debug state are UNDEFINED in Debug state.
- On taking an exception, ELR_ELx and SPSR_ELx at the target Exception level are UNKNOWN. They do not record the PC and PSTATE.

PSTATE.{IL, E, M, nRW, EL, SP} are indirectly read by instructions executed in Debug state, but all other PSTATE fields are ignored and cannot be observed. See also:

- *PSTATE in Debug state* on page H2-6936.
- *Executing instructions in Debug state* on page H2-6937.
- *Exceptions in Debug state* on page H2-6957.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H2.4.9 Accessing memory in Debug state

How the PE accesses memory is unchanged in Debug state. This includes:

- The operation of the MMU, including address translation, tagged address handling, access permissions, memory attribute determination, and the operation of any TLBs.
- The operation of any caches and coherency mechanisms.
- Alignment support.
- Endianness support.
- The Memory order model.

Simple memory transfers

Simple memory accesses can be performed in Debug state by issuing memory access instructions through the ITR and passing data through the DTR registers. *Executing instructions in Debug state* on page H2-6937 lists the memory access instructions that are supported in Debug state.

Bulk memory transfers

Memory access mode can accelerate bulk memory transfers in Debug state. See *DCC and ITR access modes* on page H4-6994.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H2.5 Exiting Debug state

The PE exits Debug state when it receives a Restart request trigger event. If EDSCR.ITE == 0 the behavior of any instruction issued through the ITR in Normal access mode or an operation issued by a DTR access in memory access mode that has not completed execution is CONSTRAINED UNPREDICTABLE, and must do one of the following:

- It must complete execution in Debug state before the PE executes the restart sequence.
- It must complete execution in Non-debug state after the PE executes the restart sequence.
- It must be abandoned. This means that the instruction does not execute. Any registers or memory accessed by the instruction are left in an UNKNOWN state.

Note

- Implementations can set EDSCR.ITE to 1 to indicate that further instructions can be accepted by ITR before the previous instructions have completed. If any previous instruction has not completed and EDSCR.ITE == 1, then the PE must complete these instructions in Debug state before executing the restart sequence. EDSCR.ITE == 0 indicates that the PE is not ready to restart.
- A debugger must observe that any instructions issued through EDITR that might generate a synchronous exception, as complete, before issuing a restart request. It can do this by observing the completion of a later instruction, as synchronous exceptions must occur in program order. For example, a debugger can observe that an instruction that reads or writes a DTR register is complete because of its effect on the EDSCR.{TXfull, RXfull} flags.

On exiting Debug state, the PE sets the program counter to the address in DLR, where:

- If exiting to AArch32 state:
  - Bits[31:1] of the PC are set to the value of bits[31:1] of DLR.
  - Bit[0] of the PC is set to a CONSTRAINED UNPREDICTABLE choice of 0 or the value of bit[0] in DLR.
- If exiting to AArch64 state:
  - Bits[63:56] of DLR_EL0 might be ignored as part of tagged address handling. See *Address tagging in AArch64 state* on page D5-2534.
  - Otherwise the PC is set from DLR_EL0.

Note

Bits[63:32] of DLR_EL0 are ignored when exiting to AArch32 state.

Exit from Debug state can give rise to a PC alignment fault exception when the program counter is used. Unlike an exception return, this might also happen when returning to AArch32 state. For more information, see *PC alignment checking* on page D1-2333.

On exiting Debug state, PSTATE is set from DSPSR in the same way that an exception return sets PSTATE from SPSR_ELx:

- The same illegal exception return checks that apply to an exception return also apply to exiting Debug state. If the return from Debug state is an illegal exception return then the effect on PSTATE and the PC is the same as for any other illegal exception return. See *Exception return* on page D1-2350 and *Exception return to an Exception level using AArch32* on page G1-5725.
- The checks on the PSTATE.IT bits that apply to exiting Debug state into AArch32 state are the same as those that apply to an exception return. See Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.
- PSTATE.SS is copied from DSPSR.SS if all of the following hold:
  - MDSCR_EL1.SS == 1.
  - The debug target Exception level is using AArch64.
  - Software step exceptions from the restart Exception level are enabled.

  Otherwise PSTATE.SS is set to 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

Unlike a return using ERET, PSTATE.SS must be restored from DSPSR.SS because otherwise it is UNKNOWN.

However, if OSDLR.DLK == 1 and DBGPRCR.CORENPDRQ == 0, meaning ARMv8.0-DoubleLock is implemented and locked in Non-debug state and therefore Software Step exceptions are disabled, but otherwise Software Step exceptions would be enabled from the restart Exception level, it is CONSTRAINED UNPREDICTABLE whether PSTATE.SS is copied from DSPSR.SS.

- If ARMv8.0-SSBS is implemented, on exit from Debug state to AArch64 state, DSPSR_EL0.SSBS is copied to PSTATE.SSBS.
- If ARMv8.0-SSBS is implemented, on exit from Debug state to AArch32 state, DSPSR.SSBS is copied to CPSR.SSBS.
- If ARMv8.1-PAN is implemented, DSPSR_EL0.PAN is copied to PSTATE.PAN.
- If ARMv8.2-UAO is implemented, DSPSR_EL0.UAO is copied to PSTATE.UAO.
- If ARMv8.4-DIT is implemented, on exit from Debug state to AArch64 state, DSPSR_EL0.DIT is copied to PSTATE.DIT.
- If ARMv8.4-DIT is implemented, on exit from Debug state to AArch32 state, DSPSR.DIT is copied to CPSR.DIT.
- If ARMv8.5-MemTag is implemented, on exit from Debug state to AArch64 state, DSPSR_EL0.TCO is copied to PSTATE.TCO. On exit from Debug state to AArch32 state, PSTATE.TCO is not updated.
- If ARMv8.5-BTI is implemented, DSPSR_EL0.BTYPE is copied to PSTATE.BTYPE.

Note

- One important difference between Debug state exit and an exception return is that the PE can exit Debug state at EL0. Despite this, the behavior of an exit from Debug state is similar to an exception return. For example, PSTATE.{D, A, I, F} is updated regardless of the value of SCTLR_EL1.UMA.
- Exit from Debug state has no architecturally-defined effect on the Event Register and Exclusives monitors. An exit from Debug state might set the Event Register or clear the Exclusives monitors, or both, but this is not a requirement and debuggers must not rely on any implementation specific behavior.

The `ExitDebugState()` function is described in Chapter J1 *Armv8 Pseudocode*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter H3
Halting Debug Events

This chapter describes a particular class of debug events. It contains the following sections:

- *Introduction to Halting debug events* on page H3-6966.
- *Halting Step debug events* on page H3-6968.
- *Halt Instruction debug event* on page H3-6978.
- *Exception Catch debug event* on page H3-6979.
- *External Debug Request debug event* on page H3-6983.
- *OS Unlock Catch debug event* on page H3-6984.
- *Reset Catch debug events* on page H3-6985.
- *Software Access debug event* on page H3-6986.
- *Synchronization and Halting debug events* on page H3-6987.

Note

Table K14-1 on page K14-8038 disambiguates the general register references used in this chapter.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H3.1 Introduction to Halting debug events

External debug defines Halting debug events. The following Halting debug events are available in Armv8:

- *Halting Step debug events* on page H3-6968.
- *Halt Instruction debug event* on page H3-6978.
- *Exception Catch debug event* on page H3-6979.
- *External Debug Request debug event* on page H3-6983.
- *OS Unlock Catch debug event* on page H3-6984.
- *Reset Catch debug events* on page H3-6985.
- *Software Access debug event* on page H3-6986.

If halting is allowed, a Halting debug event halts the PE. The PE enters Debug state.

In addition, breakpoints and watchpoints might halt the PE if halting is allowed. See *Breakpoint and Watchpoint debug events* on page H2-6928. Because breakpoints and watchpoints can generate an exception or halt the PE, Breakpoint and Watchpoint debug events are not classified as Halting debug events.

For a definition of Debug state, see Chapter H2 *Debug State*. For a definition of halting allowed, see *Halting allowed and halting prohibited* on page H2-6927.

Debug state entry and debug event prioritization on page H2-6929 describes the behavior when multiple debug events are generated by an instruction.

See also *Synchronization and Halting debug events* on page H3-6987.

Table H3-1 shows the behavior of Breakpoint, Watchpoint, and Halting debug events.

Table H3-1 Summary of debug events and possible outcomes

Debug event type	PE behavior when halting is:	
	Allowed	Prohibited
Breakpoint and Watchpoint debug events on page H2-6928	Halt	See Table D2-1 on page D2-2426 and Table G2-1 on page G2-5817
Halt Instruction debug event on page H3-6978	Halt	UNDEFINED
Software Access debug event on page H3-6986	Halt	Ignored
Exception Catch debug event on page H3-6979	Halt	Ignored
Halting Step debug events on page H3-6968	Halt	Pended
External Debug Request debug event on page H3-6983	Halt	Pended
Reset Catch debug events on page H3-6985	Halt	Pended
OS Unlock Catch debug event on page H3-6984	Pended	Pended

Table H3-2 shows where the pseudocode for each Halting debug event type is located.

Table H3-2 Pseudocode description of Halting debug events

Halting debug event type	Pseudocode
Halt Instruction debug event on page H3-6978	*HLT* on page C6-932 for AArch64 and *HLT* on page F5-4373 for AArch32
Software Access debug event on page H3-6986	*Pseudocode description of Software Access debug event* on page H3-6986
Exception Catch debug event on page H3-6979	*Pseudocode description of Exception Catch debug events* on page H3-6982

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table H3-2 Pseudocode description of Halting debug events (continued)

Halting debug event type	Pseudocode
Halting Step debug events on page H3-6968	*Pseudocode description of Halting Step debug events* on page H3-6977
External Debug Request debug event on page H3-6983	*Pseudocode description of External Debug Request debug events* on page H3-6983
Reset Catch debug events on page H3-6985	*Pseudocode description of Reset Catch debug event* on page H3-6985
OS Unlock Catch debug event on page H3-6984	*Pseudocode description of OS Unlock Catch debug event* on page H3-6984

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H3.2 Halting Step debug events

Halting Step is a debug resource that a debugger can use to make the PE step through code one instruction at a time. This section describes the Halting Step debug events. It is divided into the following sections:

- *Overview of a Halting Step debug event*.
- *The Halting Step state machine*.
- *Using Halting Step* on page H3-6971.
- *Detailed Halting Step state machine behavior* on page H3-6971.
- *Synchronization and the Halting Step state machine* on page H3-6974.
- *Stepping T32 IT instructions* on page H3-6976.
- *Disabling interrupts while stepping* on page H3-6976.
- *Syndrome information on Halting Step* on page H3-6976.
- *Pseudocode description of Halting Step debug events* on page H3-6977.

The architecture describes the behavior as a simple Halting Step state machine. See *The Halting Step state machine*.

H3.2.1 Overview of a Halting Step debug event

The behavior of Halting Step is defined by a state machine, shown in Figure H3-1 on page H3-6970. A Halting Step debug event executes a single instruction and then returns control to the debugger. When the debugger software wants to execute a Halting Step:

1. With the PE in Debug state, the debugger activates Halting Step.
2. The debugger signals the PE to exit Debug state and return to the instruction that is to be stepped.
3. The PE executes that single instruction.
4. The PE enters Debug state before executing the next instruction.

However, an exception might be generated while the instruction is being stepped. That is either:

- A synchronous exception generated by the instruction being stepped.
- An asynchronous exception taken before or after the instruction being stepped.

Halting Step has its own enable control bit, EDECR.SS and EDESR.SS.

Note

Because the Halting Step state machine states occur as a result of normal PE operation, the states can be described as both:

- PE states.
- Halting Step states.

H3.2.2 The Halting Step state machine

The state machine states are:

Inactive

Halting Step is inactive. No Halting Step debug events can be generated, therefore execution is not affected by Halting Step. The PE is in this state whenever either of the following is true:

- Halting Step is disabled. That is, EDECR.SS is set to 0 and EDESR.SS is set to 0.
- Halting is prohibited. See *Halting the PE on debug events* on page H2-6927. In this state, if EDECR.SS is set to 1 then a Halting Step debug event is pending.

In Figure H3-1 on page H3-6970 this state is shown in red.

Active-not-pending

Halting Step is enabled and active. This is the state in which the PE steps an instruction. EDECR.SS == 1 and EDESR.SS == 0. Software must not set EDECR.SS to 1 unless the PE is in Debug state, otherwise behavior is CONSTRAINED UNPREDICTABLE, as described in *Changing the value of EDECR.SS when not in Debug state* on page H3-6975.

In Figure H3-1 on page H3-6970 this state is shown in green.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Active-pending

Halting Step is enabled and active. The step has completed, and the PE enters Debug state. EDESR.SS == 1.

In Figure H3-1 on page H3-6970 this state is shown in green.

Whenever Halting Step is enabled and active, whether the state machine is in the active-not-pending state or in the active-pending state depends on EDESR.SS. *Halting Step state machine states* on page H3-6971 shows this.

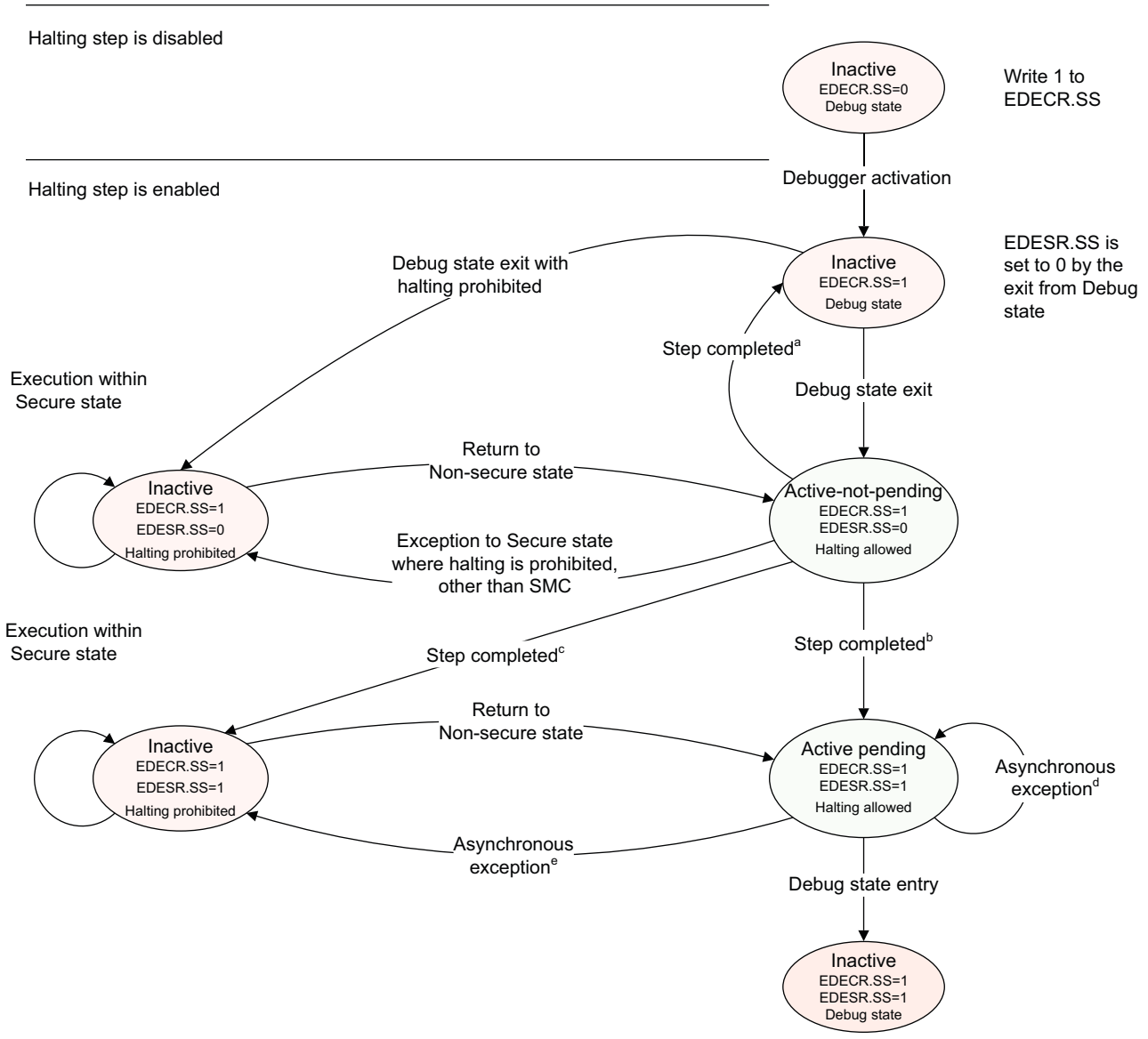
In the simple sequential execution of the program the PE executes the Halting Step state machine, as follows:

1. Initially, Halting Step is inactive.
2. After exiting Debug state, Halting Step is active-not-pending.
3. The PE executes an instruction and Halting Step is active-pending.
4. The pending Debug state entry is taken on the next instruction and the step is complete.

Exceptions and other changes to the PE context can interrupt this sequence.

Figure H3-1 on page H3-6970 shows a Halting Step state machine.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta



a. Step completed occurs when:
- A debug event, other than a Halting Step debug event, causes entry into Debug state.

b. Step completed occurs when:
- An instruction is executed without taking an exception.
- An exception is taken to a state where halting is allowed.
- A reset.

c. Step completed occurs when:
- An SMC exception is taken to Secure state where halting is prohibited.

d. An asynchronous exception taken to a state where halting is allowed.

e. An asynchronous exception taken to Secure state where halting is prohibited.

Figure H3-1 Halting Step state machine

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

Figure H3-1 on page H3-6970 only describes state transitions to and from the inactive state by exit from Debug state, executing an exception return, or taking an exception. Other changes to the PE context, including writes to registers such as EDECR and OSDLR and changes to the authentication interface can also cause changes to the Halting Step state machine. These can lead to UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior. See *Synchronization and the Halting Step state machine* on page H3-6974.

The following bits control the state machine, as shown in Table H3-3:

- EDECR.SS. This is the Halting Step enable bit.
- EDESR.SS. This is the Halting Step debug event pending bit.

Table H3-3 shows the Halting Step state machine states. The letter X in a register column means that the relevant bit can be set to either zero or one.

Table H3-3 Halting Step state machine states

Halting	EDECR.SS	EDESR.SS	Halting Step state
Prohibited	X	X	Inactive (Halting Step debug even not pending)
Prohibited	X	1	Inactive (Halting Step debug event pending).
Allowed	0	0	Inactive
Allowed	1	0	Active-not-pending
Allowed	X	1	Active-pending

H3.2.3 Using Halting Step

To step a single instruction the PE must be in Debug state:

1. The debugger sets EDECR.SS to 1 to enable Halting step.
2. The debugger signals the PE to exit Debug state with DLR set to the address of the instruction being stepped. The PE clears EDESR.SS to 0 and the Halting Step state machine enter the active-not-pending state.
3. The PE executes the instruction being stepped.

   If an exception is taken to a state where halting is prohibited, then EDESR.SS is always correct for the preferred return address of the exception.
4. The PE enters Debug state before executing the next instruction and the step is complete.

Note

- If ARMv8.3-DoPD is not implemented, EDECR.SS value is in the Debug power domain, meaning that the state machine is maintained over a powerdown of the Core power domain.
- If ARMv8.3-DoPD is implemented, the values of EDECR.SS and EDESR.SS are set to 0 on a Cold reset, and, if the PE was stepping an instruction, EDESR.SS is effectively UNKNOWN after a Warm reset. A debugger must use a Reset Catch debug event to step over a powerdown state.
- A debugger must only change the value of EDECR.SS when the PE is in Debug state, otherwise behavior is CONSTRAINED UNPREDICTABLE as described in *Changing the value of EDECR.SS when not in Debug state* on page H3-6975.

H3.2.4 Detailed Halting Step state machine behavior

The behavior of the Halting Step state machine is described in the following sections:

- *Entering the active-not-pending state* on page H3-6972.
- *PE behavior in the active-not-pending state* on page H3-6972.
- *Entering the active-pending state* on page H3-6973.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *PE behavior in the active-pending state* on page H3-6974
- *PE behavior in the inactive state when in Non-debug state* on page H3-6974.
- *PE behavior in Debug state* on page H3-6974.

Entering the active-not-pending state

The PE enters the active-not-pending state:

- By exiting Debug state with EDECR.SS == 1.
- By an exception return from a state where halting is prohibited to a state where halting is allowed with EDECR.SS == 1 and EDESR.SS == 0.
- As described in *Synchronization and the Halting Step state machine* on page H3-6974.

PE behavior in the active-not-pending state

When the PE is in the active-not-pending state it does one of the following:

- It executes one instruction and does one of the following:
  - Completes it without generating a synchronous exception.
  - Generates a synchronous exception.
  - Generates a debug event that causes entry to Debug state.
- It takes an asynchronous exception without executing any instruction.
- It takes an asynchronous debug event into Debug state.

If no exception or debug event is generated

If no exception or debug event is generated the PE sets EDESR.SS to 1. This means that the Halting Step state machine advances to the active-pending state.

If an exception or debug event is generated

The PE sets EDESR.SS according to all of the following:

- The type of exception.
- The target Exception level of the exception.
- If the exception is taken to Secure state, whether halting is prohibited in Secure state.
  - This is determined by the result of ExternalSecureInvasiveDebugEnabled().

If an exception or debug event is generated, the PE sets EDESR.SS to 1 if one of the following applies:

- A synchronous exception is generated by the instruction and one of the following applies:
  - The exception is taken to EL1 or EL2.
  - The exception is taken to EL3, it is not an SMC exception, and `ExternalSecureInvasiveDebugEnabled()` == TRUE.
  - The exception is an SMC exception.
- An asynchronous exception is generated before executing an instruction and this is either:
  - Taken to EL1 or EL2.
  - Taken to EL3 and `ExternalSecureInvasiveDebugEnabled()` == TRUE.
- A PE reset occurs.

Otherwise EDESR.SS is unchanged. This happens when:

- No instruction is executed because either:
  - An asynchronous exception is taken to EL3 and `ExternalSecureInvasiveDebugEnabled()` == FALSE.
  - An asynchronous debug event causes entry to Debug state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- An instruction is executed and either:
  - Generates a synchronous exception other than an SMC exception which is taken to EL3, and `ExternalSecureInvasiveDebugEnabled()` == FALSE.
  - Generates a synchronous debug event and causes entry to Debug state.

It is UNPREDICTABLE whether EDESR.SS is set to 1 or unchanged when an SError interrupt is taken to EL3 without executing the instruction, and `ExternalSecureInvasiveDebugEnabled()` == FALSE.

If halting is prohibited after taking the exception or debug event, then the Halting Step state machine advances to the inactive state. Otherwise the Halting Step state machine advances to the active-pending state.

Note

The underlying criteria for the value of EDESR.SS on an exception are:

- Whether halting is allowed at the target of the exception. If halting is allowed, the PE must step into the exception. If halting is prohibited, the PE must step over the exception.
- Whether the preferred return address of the exception is the instruction itself or the next instruction, if the PE steps over the exception.

Table H3-4 shows the behavior of the active-not-pending state. The letter X indicates that `ExternalSecureInvasiveDebugEnabled()` can be either TRUE or FALSE.

Table H3-4 Summary of active-not-pending state behavior

Event	Target Exception level	`ExternalSecureInvasiveDebugEnabled()`	Value written to EDESR.SS
No exception or debug event	Not applicable	X	1
SMC exception	EL3	X	1
Reset	Highest	X	1
Exception, other than SMC exception	EL1	X	1
	EL2	X	1
	EL3	TRUE	1
		FALSE	Unchanged
Debug event	Debug state	X	Unchanged

Entering the active-pending state

The PE enters the active-pending state by one of the following:

- From the active-not-pending state by:
  - Executing an instruction without taking an exception.
  - Taking an exception so that the PE remains in a state where halting is allowed.
- An exception return from a state where halting is prohibited when EDESR.SS == 1.

  Note

  That is, an exception return from Secure state with `ExternalSecureInvasiveDebugEnabled()` == FALSE to Non-secure state with `ExternalInvasiveDebugEnabled()` == TRUE.

- A reset when the value of EDECR.SS == 1, regardless of the state the PE was in before the reset occurred.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- From the active-pending state by taking an asynchronous exception to a state where halting is allowed.
- Following the description in *Synchronization and the Halting Step state machine*.

PE behavior in the active-pending state

When the PE is in the active-pending state, it enters Debug state before executing an instruction.

The entry into Debug state has higher priority than all other types of synchronous debug event and synchronous exception. However, the architecture does not define the prioritization of this Debug state entry with respect to any unmasked pending asynchronous exception. If an asynchronous exception is prioritized over the entry to Debug state, then EDESR.SS is unchanged.

For more information on the prioritization of debug events, see *Debug state entry and debug event prioritization* on page H2-6929.

PE behavior in the inactive state when in Non-debug state

EDESR.SS is not updated by the execution of an instruction or the taking of an exception when Halting Step is inactive. This means that EDESR.SS is not changed by an exception handled in a state where halting is prohibited.

On return to a state where halting is allowed, the Halting Step state machine is restored either to the active-pending state or the active-not-pending state, depending on the value of EDESR.SS. The return to a state where halting is allowed is normally by an exception return, which in some situations is a *Context synchronization event*.

See also *Synchronization and the Halting Step state machine*.

PE behavior in Debug state

Halting Step is inactive in Debug state because halting is prohibited, see *Halting allowed and halting prohibited* on page H2-6927.

Entry to Debug state does not change EDESR.SS.

EDESR.SS is cleared to 0 on exiting Debug state as the result of a restart request. If EDECR.SS == 1, Halting Step enters the active-not-pending state.

Note

This means that EDESR.SS is never cleared to 0 by the execution of an instruction in Debug state, or by taking an exception when in Debug state as described in *PE behavior in the active-not-pending state* on page H3-6972, because the Halting Step state machine is not in the active-not-pending state. EDESR.SS can be cleared by a write to EDESR, see the register description.

However, if the PE exits Debug state as the result of a PE reset and EDECR.SS == 1, then Halting Step immediately enters the active-pending state, as EDESR.SS is set to the value of EDECR.SS.

H3.2.5 Synchronization and the Halting Step state machine

The Halting Step state machine also changes state if:

- Halting becomes allowed or prohibited other than by exit from Debug state, an exception return, or taking an exception. This means that halting becomes allowed or prohibited because:
  - The security state changes without an exception return. See *State and mode changes without explicit context synchronization events* on page G2-5876.
  - The external authentication interface changes.
  - ARMv8.0-DoubleLock is implemented and the status, `DoubleLockStatus()`, changes.
- A write to EDECR when the PE is in Non-debug state changes the value of EDECR.SS.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

Behavior is CONSTRAINED UNPREDICTABLE if the value of EDECR.SS is changed when the PE is in Non-debug state, see *Changing the value of EDECR.SS when not in Debug state*.

- A write to EDESR when the PE is in Non-debug state clears EDESR.SS to 0.

These operations are guaranteed to take effect only after a *Context synchronization event*. If the instruction being stepped generates a *Context synchronization event*, then the PE might use the old or new state.

The PE must perform the required behavior of the new state before or immediately following the next *Context synchronization event*, but it is not required to do so immediately. This means that the PE can perform the required behavior of the old state before the next *Context synchronization event*. This is illustrated in Example H3-1 and Example H3-2.

Example H3-1 Synchronization requirements 1

EDECR.SS is set to 1 in Debug state, requesting the active-not-pending state on exit from Debug state. On exit from Debug state the PE immediately takes an exception to Secure state. `ExternalSecureInvasiveDebugEnabled()` == FALSE, meaning that halting is prohibited in Secure state. The PE does not step any instructions but executes the software in Secure state as normal. EDESR.SS remains set to 0. If `ExternalSecureInvasiveDebugEnabled()` subsequently becomes TRUE, meaning that halting is now allowed, the PE must perform the required behavior of the active-not-pending state before or immediately following the next *Context synchronization event*, but it is not required to do so immediately.

Example H3-2 Synchronization requirements 2

EDECR.SS is set to 1 in Debug state. On exit from Debug the PE executes an `MSR` instruction that sets OSDLR_EL1.DLK to 1 and `DoubleLockStatus()` becomes TRUE. This change requires a *Context synchronization event* to guarantee its effect, meaning it is CONSTRAINED UNPREDICTABLE whether:

- Halting is allowed:
  - The PE enters Debug state on the next instruction.
- Halting is prohibited:
  - The PE does not enter Debug state.

The value in EDESR.SS depends on whether halting was allowed or prohibited when the write to OSDLR_EL1.DLK completed, and so it might be 0 or 1. If a second `MSR` instruction clears OSDLR_EL1.DLK to 0, the PE must perform the required behavior of the state indicated by EDESR.SS before or immediately following the next *Context synchronization event*, but it is not required to do so immediately.

See also *Synchronization and Halting debug events* on page H3-6987.

Changing the value of EDECR.SS when not in Debug state

If software changes the value of EDECR.SS when the PE is not in Debug state then behavior is CONSTRAINED UNPREDICTABLE, and one or more of the following behaviors occurs:

- The value of EDECR.SS becomes UNKNOWN.
- The state of the Halting Step state machine becomes UNKNOWN.
- On a reset of the PE, the value of EDECR.SS and the state of the Halting Step state machine are UNKNOWN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H3.2.6 Stepping T32 IT instructions

In an implementation that supports the ITD control, the architecture permits a combination of one T32 IT instruction and another 16-bit T32 instruction to be treated as a single 32-bit instruction when the value of the ITD field that applies to the current Exception level is 1.

For the purpose of stepping an item, it is IMPLEMENTATION DEFINED whether:

- The PE considers such a pair of instructions to be one instruction.
- The PE considers such a pair of instructions be two instructions.

It is IMPLEMENTATION DEFINED whether this behavior depends on the value of the applicable ITD bit. For example:

- The debug logic might consider such a pair of instructions as one instruction, regardless of the state of the applicable ITD field.
- The debug logic might consider such a pair of instructions as two instructions, regardless of the state of the applicable ITD field.
- The debug logic might consider such a pair of instructions as one instruction when the value of the applicable ITD field is 1, and as two instructions when the value of the ITD field is 0.

An implementation that does not support the ITD control behaves as if the value of the ITD field is 0.

The ITD control fields are:

HSCTLR.ITD
Applies to execution at EL2 when EL2 is using AArch32.

SCTLR.ITD
Applies to execution at EL0 or EL1 when EL1 is using AArch32.

SCTLR_EL1.ITD
Applies to execution at EL0 using AArch32 when EL1 is using AArch64.

H3.2.7 Disabling interrupts while stepping

When using Halting Step, the sequence of entering Debug state, interacting with the debugger, and then exiting Debug state for each instruction reduces the rate at which the PE executes instructions. However, the rate at which certain interrupts, such as timer interrupts, are generated might be fixed by the system. This means it might be necessary to disable interrupts while using Halting Step by setting EDSCR.INTdis, to allow the code being debugged to make forward progress.

H3.2.8 Syndrome information on Halting Step

Three EDSCR.STATUS encodings record different scenarios for entering Debug state on a Halting Step debug event:

Halting Step, normal
An instruction other than a Load-Exclusive instruction was stepped.

Halting Step, exclusive
A Load-Exclusive instruction was stepped.

Halting Step, no syndrome
The syndrome data is not available.

If the PE enters Debug state due to a Halting Step debug event immediately after stepping an instruction in the active-not-pending state, EDSCR.STATUS is set to either:

- Halting Step, normal, if the stepped instruction was not a Load-Exclusive instruction.
- Halting Step, exclusive, if the stepped instruction was a Load-Exclusive instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the stepped instruction was a conditional Load-Exclusive instruction that failed its Condition code check, EDSCR.STATUS is set to a CONSTRAINED UNPREDICTABLE choice of Halting Step, normal, or Halting Step, exclusive.

Otherwise the PE enters Debug state without stepping an instruction. This means that the Halting Step state machine enters the active-pending state directly from the inactive state, without going through active-not-pending state. In this case, EDSCR.STATUS is set to Halting Step, no syndrome. This happens when:

- The PE enters directly into the active-pending state on an exception return to Non-secure state from EL3 when Halting is prohibited in Secure state.
- The active-pending state is entered for other reasons. See *Synchronization and the Halting Step state machine on page H3-6974*

In addition, EDSCR.STATUS is CONSTRAINED UNPREDICTABLE when:

- The instruction being stepped generated a Halting Step debug event before the instruction was executed.

  In this case EDSCR.STATUS is set to a CONSTRAINED UNPREDICTABLE choice of:

  — Halting Step, no syndrome, or Halting Step, normal, if the stepped instruction was not a Load-Exclusive instruction.

  — Halting Step, no syndrome, or Halting Step, exclusive, if the stepped instruction was a Load-Exclusive instruction.
- The instruction that was stepped was an Exception Return instruction or an `ISB`. As these instructions are not in the Load-Exclusive instructions, EDSCR.STATUS is set to a CONSTRAINED UNPREDICTABLE choice of Halting Step, no syndrome or Halting Step, normal.
- The PE enters directly into the active-pending state on reset because EDECR.SS is set to 1. EDSCR.STATUS is set to a CONSTRAINED UNPREDICTABLE choice of Halting Step, no syndrome or Halting Step, normal.

In all cases, if EDSCR.STATUS is not set to Halting Step, no syndrome, then it must indicate whether the stepped instruction was a Load-Exclusive instruction by setting EDSCR.STATUS to Halting Step, normal or Halting Step, exclusive.

Note

In an implementation that always sets EDSCR.STATUS to Halting Step, no syndrome is not compliant.

H3.2.9 Pseudocode description of Halting Step debug events

There are two pseudocode functions for Halting Step debug events:

- `RunHaltingStep()`. This is called after an instruction has executed and any exception generated by the instruction is taken. It is also called after taking a reset before executing any instructions. That is, reset is treated like an asynchronous exception, even if EDECR.RCE == 1 or CTIDEVCTL.RCE == 1. `RunHaltingStep()` affects the next instruction.
- `CheckHaltingStep()`. This is called before the next instruction is executed. If a step is pending, it generates the debug event.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H3.3 Halt Instruction debug event

A Halt Instruction debug event is generated when EDSCR.HDE == 1, halting is allowed, and the PE executes the Halt instruction, `HLT`.

The pseudocode for Halt Instruction debug events is described in *HLT* on page C6-932 for A64 and *HLT* on page F5-4373 for A32 and T32.

`HLT` never generates a debug exception. It is treated as UNDEFINED if EDSCR.HDE == 0, or if halting is prohibited.

Note

A debugger can replace a program instruction with a Halt instruction to generate a Halt Instruction debug event. Debuggers that use the `HLT` instruction must be aware of the Armv8-A rules for concurrent modification of executable code, CMODX. The rules for concurrent modification and execution of instructions do not allow one thread of execution or an external debugger to replace an instruction with an `HLT` instruction when these same instructions are being executed by a different thread of execution. See *Concurrent modification and execution of instructions* on page B2-120.

The T32 `HLT` instruction is unconditionally executed inside an `IT` block, even when it is treated as undefined. The A32 `HLT` instruction is CONSTRAINED UNPREDICTABLE if the Condition code field is not `0b1110`, with the set of behaviors the same as for `BKPT`. See Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Note

The `HLT` instruction is part of the external debug solution for Armv8-A. As such, the presence of the `HLT` instruction is not indicated in the ID registers. In particular, the AArch32 System register ID_ISAR0. Debug does not indicate the presence of the `HLT` instruction.

H3.3.1 HLT instructions as the first instruction in a T32 IT block

In an implementation that supports the ITD control, the architecture permits a combination of one T32 `IT` instruction and certain other 16-bit T32 instruction to be treated as a single 32-bit instruction when the value of the ITD field that applies to the current Exception level is 1.

The T32 `HLT` instruction cannot be combined with an `IT` instruction in this way. In an implementation that supports the ITD control, if the first instruction in an IT block is an `HLT` instruction, then the behavior of the instruction depends on the value of the applicable ITD field:

- If the value of the ITD field is 1, then the combination is treated as undefined and an Undefined Instruction exception is generated either by the `IT` instruction or by the `HLT` instruction.
- If the value of the ITD field is 0, then the `HLT` instruction unconditionally executed.

An implementation that does not support the ITD control behaves as if the value of the ITD field is 0.

To set an Halt Instruction debug event on the first instruction of an IT block, debuggers must replace the `IT` instruction with an `HLT` instruction to ensure consistent behavior.

The ITD control fields are:

HSCTLR.ITD
: Applies to execution at EL2 when EL2 is using AArch32.

SCTLR.ITD
: Applies to execution at EL0 or EL1 when EL1 is using AArch32.

SCTLR_EL1.ITD
: Applies to execution at EL0 using AArch32 when EL1 is using AArch64.

Note

An `HLT` instruction is always unconditional, even within an IT block.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H3.4 Exception Catch debug event

Exception Catch debug events:

- Are generated when the corresponding bit in the Exception Catch Control Register, EDECCR, is set to 1 on all entries to a given Exception level. This means:
  - Exceptions taken to the Exception level.
  - Exception returns to the Exception level.
- Are taken synchronously, after entry to the Exception level.
- Ignore the Execution state of the target Exception level.
- Are ignored if halting is prohibited.

From Armv8.2 the definition is extended to include generation of Exception Catch debug events on reset entry, see *Exception Catch debug events from Armv8.2* on page H3-6981.

The EDECCR contains two sets of fields:

- One set for Non-secure state.
- One set for Secure state.

Each set of fields control generation of Exception Catch debug events:

- On exception entry and, from Armv8.2, on reset entry.
- On exception return.

Each field within each set contains one bit for each Exception level in that state. Bits corresponding to Exception levels that are not implemented are RES0. For more information see the EDECCR description.

Note

- EDECCR does not replace DBGVCR:
  - DBGVCR is retained in AArch32 state for backwards compatibility.
  - DBGVCR is ignored in AArch64 state and never generates entries to Debug state.
  - DBGVCR cannot be accessed by the external debug interface.
- EDECCR is only visible as OSECCR_EL1 by System register instructions in AArch64 state, and as DBGOSECCR by System register access instructions in AArch32 state, when the OS Lock is locked to allow software to save and restore it over a powerdown.
- Exception Catch debug events are not disabled when the OS Lock is locked.

For an Exception Catch debug event generated after taking an exception to a trapped Exception level:

- The PE must not fetch instructions from the vector address before entering Debug state, if address translation is disabled in the translation regime at the target Exception level.
- On entering Debug state:
  - The current Exception level is the target Exception level of the exception.
  - The ELR, SPSR, ESR, and other syndrome registers contain information about the exception.
  - DLR contains the exception vector address or, from Armv8.2, the reset address.

H3.4.1 Prioritization of Exception Catch debug events

The following rules define the prioritization of Exception Catch debug events:

- It is IMPLEMENTATION DEFINED whether Exception Catch debug events are higher or lower priority than each of Software Step exceptions and Halting Step debug events.
- Exception Catch debug events are higher priority than all synchronous exceptions other than Software Step exceptions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Exception Catch debug events are lower priority than Reset Catch debug events.

Note

As described in *Synchronous exception prioritization for exceptions taken to AArch64 state* on page D1-2355, an exception trapping form of a Vector Catch debug event might generate a second debug exception as part of the exception entry, before the Exception Catch debug event is taken. See *Vector Catch exceptions* on page D2-2471 or *Vector Catch exceptions* on page G2-5869.

A second unmasked asynchronous exception can be taken before the PE enters Debug state. If this second exception does not generate an Exception Catch debug event, the exception handler executed at the higher Exception level later returns to the trapped Exception level, causing the Exception Catch debug event to be generated again.

See also *Debug state entry and debug event prioritization* on page H2-6929.

From Armv8.2, when an Exception Catch debug event is generated on exception entry, the PE must enter Debug state as part of the exception entry:

- After the exception entry has updated the program counter, PSTATE and syndrome registers for the exception.
- Before the first instruction at the handler is executed.
- Before any asynchronous or lower priority synchronous exception or debug event is taken at the first instruction in the exception handler.

From Armv8.2, when an Exception Catch debug event is generated on exception return, the PE must enter Debug state:

- After the exception return has updated the program counter and PSTATE.
- Before the execution of the first instruction at the return address is completed.

Note

From Armv8.2, there is no prioritization between asynchronous exceptions, asynchronous debug events, and an Exception Catch debug event generated on an exception return.

H3.4.2 CONSTRAINED UNPREDICTABLE generation of Exception Catch debug events

When ARMv8.2-Debug is not implemented, the PE is executing code at a given Exception level, and the corresponding EDECCR bit is 1, it is CONSTRAINED UNPREDICTABLE whether an Exception Catch debug event is generated.

Note

It is possible to generate Exception Catch debug events:

- As a trap on all instruction fetches from the trapped Exception level as part of an instruction fetch.
- On entry to the Exception level, as described in *Detailed Halting Step state machine behavior* on page H3-6971.

This is similar to the implementation options allowed for Vector Catch debug events. The architecture does not require that the event is generated following an `ISB` operation executed at the Exception level.

Examples of this are:

- If the debugger writes to EDECCR so that the current Exception level is trapped.
- If the OS restore code writes to OSECCR so that the current Exception level is trapped.
- If the code executing in AArch32 state changes the Exception level or security state other than by an exception return, and the target Exception level is trapped. See *State and mode changes without explicit context synchronization events* on page G2-5876.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When ARMv8.2-Debug is implemented, there are no CONSTRAINED UNPREDICTABLE cases that are permitted to generate Exception Catch debug events.

H3.4.3 Exception Catch debug events from Armv8.2

From Armv8.2, the definition of Exception Catch debug events is extended so that Exception Catch debug events can be generated from:

- Exception entry.
- Reset entry.
- Exception return.

It is IMPLEMENTATION DEFINED whether a reset into an Exception level generates an Exception Catch debug event.

Exception catch debug events are taken synchronously, after the exception or reset entry or the exception return has been processed by the PE.

For exception returns, the final Exception level of the exception return determines whether an Exception Catch debug event is generated. On an illegal exception return an Exception Catch debug event is generated only if EDECCR is programmed to generate an Exception Catch debug event for an exception return to the current Exception level.

The NSR and SR fields in EDECCR determine whether Exception Catch debug events can be generated from reset entry. Table H3-5 shows how exception entry and exception return Exception Catch debug events are enabled by a combination of the fields NSR, SR, NSE and SE in EDECCR.

Table H3-5 Summary of Exception Catch debug event control

(N)SR<n>	(N)SE<n>	Behavior on exception return to ELn	Behavior on exception taken to ELn
0	0	No action.	No action.
0	1	Halt if allowed.	Halt if allowed.
1	0	Halt if allowed.	No action.
1	1	No action.	Halt if allowed.

H3.4.4 Examples of Exception Catch debug events

If EDECCR == `0x20`, meaning that the Exception Catch debug event is enabled for Non-secure EL1, then the following exceptions generate Exception Catch debug events:

- An exception taken from Non-secure EL0 to Non-secure EL1.
- An exception return from EL2 to Non-secure EL1.
- An exception return from EL3 to Non-secure EL1.

For example, on taking a Data Abort exception from Non-secure EL0 to Non-secure EL1, using AArch64:

- ELR_EL1 and SPSR_EL1 are written with the preferred return address and PE state for a return to EL0.
- ESR_EL1 and FAR_EL1 are written with the syndrome information for the exception.
- DLR_EL0 is set to VBAR_EL1 + `0x400`, the synchronous exception vector.
- DSPSR_EL0 is written with the PE state for an exit to EL1.

The following do not generate Exception Catch debug events:

- An exception taken from EL0 to EL2, if enabled in the current Security state, or EL3.
- An exception return from EL2, if enabled in the current Security state, to EL0.
- An exception taken from Secure EL0 to Secure EL1.
- An exception return from EL3 to Secure EL1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H3.4.5 Pseudocode description of Exception Catch debug events

The pseudocode function `CheckExceptionCatch()` is described in Chapter J1 *Armv8 Pseudocode*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H3.5 External Debug Request debug event

External Debug Request debug events are asynchronous debug events.

An External Debug Request debug event is generated when signaled by the embedded cross-trigger. See Chapter H5 *The Embedded Cross-Trigger Interface*.

Note

Armv8-A requires the implementation of an embedded cross-trigger.

An implementation might also support IMPLEMENTATION DEFINED ways of generating an External Debug Request debug event.

H3.5.1 Synchronization and External Debug Request debug events

An External Debug Request debug event that is asserted before a *Context synchronization event* is taken and the PE enters Debug state before the first instruction following the *Context synchronization event* completes its execution, provided that halting is allowed after completion of the *Context synchronization event*.

An External Debug Request debug event that is being asserted when the PE comes out of reset is taken, and the PE enters Debug state before the first instruction after the reset completes its execution, provided that halting is allowed when the PE exits reset state.

If the first instruction after the *Context synchronization event* or after coming out of reset generates a synchronous exception then the architecture does not define the order in which the debug event and the exception or exceptions are taken.

Otherwise, when halting is allowed, External Debug Request debug events must be taken in finite time, without requiring the synchronization of any necessary change to the external authentication interface.

Note

These rules are based on the rules that apply when taking asynchronous exceptions. See *Asynchronous exception types, routing, masking and priorities* on page D1-2363.

If an unmasked External Debug Request debug event was pending but is changed to not pending before it is taken, then the architecture permits the External Debug Request debug event to be taken, but does not require this to happen. If the External Debug Request debug event is taken then it must be taken before the first *Context synchronization event* after the External Debug Request debug event was changed to not pending.

Example H3-3 shows an example of the synchronization requirements.

Example H3-3 Synchronization requirements

Secure software locks up in a tight loop, so it executes indefinitely without any synchronization operations. An External debug request must be able to break the PE out of that loop. This is a requirement even if **DBGEN** or **SPIDEN** or both are LOW on entry to the loop, meaning that halting is prohibited, and are only asserted HIGH later.

H3.5.2 Pseudocode description of External Debug Request debug events

The `ExternalDebugRequest()` function is described in Chapter J1 *Armv8 Pseudocode*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H3.6 OS Unlock Catch debug event

An OS Unlock Catch debug event is generated when enabled and the state of the OS Lock changes from locked to unlocked. When ARMv8.3-DoPD is implemented, CTIDEVCTL.OSUCE enables an OS Unlock Catch debug event, otherwise EDECR.OSUCE enables an OS Unlock Catch debug event.

When the OS Lock is unlocked, the PE sets EDESR.OSUC to 1 if the OS Unlock Catch debug event is enabled, and the PE is in Non-debug state, meaning the OS Unlock Catch debug event becomes pending. However, this is an indirect write to EDESR.OSUC, meaning the OS Unlock Catch debug event is not guaranteed to be taken before a subsequent Context synchronization event. If the PE enters Debug state or the OS Unlock Catch debug event is disabled before EDESR.OSUC becomes set to 1, then EDESR.OSUC might not be set.

OS Unlock Catch debug events are not generated if the OS Lock is unlocked when the PE is in Debug state. See also *Synchronization and Halting debug events* on page H3-6987.

EDESR.OSUC is cleared to 0 on a Warm reset and on exiting Debug state.

H3.6.1 Using the OS Unlock Catch debug event

If the debugger attempts to access a debug register when the Core power down domain is completely off or in a low-power state in which the Core power domain registers cannot be accessed, and that access returns an error, then the debugger must retry the access. However, if the Core power domain is regularly powered down, this can lead to unreliable debugger behavior.

The debugger can program a Reset Catch debug event to halt the PE when it has powered up, and can program the debug registers from Debug state. However, if the PE boot software restores the debug registers, as described in *Debug OS Save and Restore sequences* on page H6-7035, then newly written values are overwritten by the restore sequence.

The debugger can program an OS Unlock Catch debug event to halt the PE after the restore sequence has completed, and program the debug registers from Debug state.

H3.6.2 Pseudocode description of OS Unlock Catch debug event

The `CheckOSUnlockCatch()` function is called when the OS Lock is unlocked.

The `CheckPendingOSUnlockCatch()` function is called before an instruction is executed. If an OS Unlock Catch is pending, it generates the debug event.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H3.7 Reset Catch debug events

A Reset Catch debug event is generated when enabled, and the PE exits reset state. When the Reset Catch debug event is generated, it is recorded by setting EDESR.RC to 1. When ARMv8.3-DoPD is implemented, CTIDEVCTL.RCE enables a Reset Catch debug event, otherwise EDECR.RCE enables a Reset Catch debug event

If halting is allowed when the event is generated, the Reset Catch debug event is taken immediately and synchronously. On entering Debug state, DLR has the address of the reset vector. The PE must not fetch any instructions from memory.

Otherwise, the Reset Catch debug event is pended and taken when halting is allowed. See *Synchronization and Halting debug events* on page H3-6987 for more information.

This means that EDESR.RC is set to the value of EDECR.RCE or CTIDEVCTL.RCE on a Warm reset. EDESR.RC is cleared to 0 on exiting Debug state.

H3.7.1 Pseudocode description of Reset Catch debug event

The `CheckResetCatch()` function is called after reset before executing any instruction.

The `CheckPendingResetCatch()` function is called before an instruction is executed. If a Reset Catch is pending, it generates the Reset Catch debug event.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H3.8 Software Access debug event

When the value of EDSCR.TDA is 1, software access to the following AArch64 and AArch32 debug System registers generate a Software Access debug event:

- The Breakpoint Value Registers, DBGBVR.
- The Breakpoint Control Registers, DBGBCR.
- The Watchpoint Value Registers, DBGWVR.
- The Watchpoint Control Registers, DBGWCR.

However, EDSCR.TDA is ignored if any of the following applies:

- The value of OSLSR.OSLK == 1, meaning that the OS Lock is locked.
- Halting is prohibited. See *Halting allowed and halting prohibited* on page H2-6927.
- The register access generates an exception.

Note

- The only accesses to the specified registers that generate a Software Access debug event are:
  - Accesses to System registers in AArch64 state.
  - Accesses to System registers in the (`coproc==0b1110`) encoding space in AArch32 state.
- Accesses by a PE using the external debug interface never generate a Software Access debug event.

H3.8.1 Pseudocode description of Software Access debug event

The `CheckSoftwareAccessToDebugRegisters()` function is described in Chapter J1 *Armv8 Pseudocode*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H3.9 Synchronization and Halting debug events

The behavior of external debug depends on:

- Indirect reads of:
  - External debug registers.
  - System registers, including system debug registers.
  - Special-purpose registers.
- The state of the external authentication interface.

For some registers, all read and write accesses that update the register occur in program order, without any additional synchronization, but others require an explicit *Context synchronization event*. For more information on the synchronization of register updates see:

- *Synchronization requirements for AArch64 System registers* on page D13-2869.
- *Synchronization of changes to the external debug registers* on page H8-7050.
- *State and mode changes without explicit context synchronization events* on page G2-5876.

Changes to the external authentication interface do not require explicit synchronization to affect External Debug Request debug events. See *Synchronization and External Debug Request debug events* on page H3-6983.

For changes that require explicit synchronization, it is CONSTRAINED UNPREDICTABLE whether instructions between the change and the *Context synchronization event* observe the old state or the new state.

This means that any change to these registers or the external authentication interface requires explicit synchronization by a *Context synchronization event* before the change takes effect. This ensures that for instructions appearing in program order after the change, the change affects the following:

- The generation and behavior of Breakpoint and Watchpoint debug events. See *Synchronization and debug exceptions* on page D2-2485 for exceptions taken from AArch64 state, or *Synchronization and debug exceptions* on page G2-5876 for exceptions taken from AArch32 state.
- The generation of all Halting debug events by instructions.
- Taking a pending Halting debug event or other asynchronous Debug event. See:
  - *Pending Halting debug events*.
  - *Synchronization and External Debug Request debug events* on page H3-6983.
- The behavior of the Halting Step state machine. See *Synchronization and the Halting Step state machine* on page H3-6974.

H3.9.1 Pending Halting debug events

A Halting debug event might be pending:

1. If Halting Step of an instruction sets EDESR.SS is set to 1, and halting is prohibited following the step, then the Halting Step state machine is inactive but a Halting Step debug event is pending.

2. If a Reset Catch debug event sets EDESR.RC to 1, and halting is prohibited following reset, then a Reset Catch debug event is pending.

3. If an OS Unlock Catch debug event sets EDESR.OSUC to 1, then an OS Unlock Catch debug event is pending.

Pending Halting debug events are taken asynchronously when halting is allowed.

Pending Halting debug events are discarded by a Cold reset. The debugger can also force a pending event to be dropped by writing to EDESR.

Any Halting debug event that is observed as pending in the EDESR before a *Context synchronization event* is taken and the PE enters Debug state before the first instruction following the *Context synchronization event* completes its execution. This is only possible if halting is allowed after completion of the *Context synchronization event*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the first instruction after the *Context synchronization event* generates a synchronous exception then the architecture does not define the order in which the debug event and the exception or exceptions are taken, unless both:

- A Halting Step debug event is pending. EDESR.SS == 1.
- The *Context synchronization event* is an exception return from a state where halting is prohibited to a state where halting is allowed.

  Note

  This applies to an exception return from Secure state with `ExternalSecureInvasiveDebugEnabled()` == FALSE to Non-secure state with `ExternalInvasiveDebugEnabled()` == TRUE.

In this case the order in which the debug events are handled is specified to avoid a double-step. See *Entering the active-pending state* on page H3-6973.

If an asynchronous exception is also pending after the *Context synchronization event* then the architecture does not define the order in which the debug event and the exception or exceptions are taken.

Note

These rules are based on the rules that apply to taking asynchronous exceptions. See *Asynchronous exception types, routing, masking and priorities* on page D1-2363.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter H4
The Debug Communication Channel and Instruction Transfer Register

This chapter describes communication between a debugger and the implemented debug logic, using the *Debug Communications Channel* (DCC) and the *Instruction Transfer Register* (ITR), and associated control flags. It contains the following sections:

- *Introduction* on page H4-6990.
- *DCC and ITR registers* on page H4-6991.
- *DCC and ITR access modes* on page H4-6994.
- *Flow control of the DCC and ITR registers* on page H4-6998.
- *Synchronization of DCC and ITR accesses* on page H4-7002.
- *Interrupt-driven use of the DCC* on page H4-7008.
- *Pseudocode description of the operation of the DCC and ITR registers* on page H4-7009.

Note

Where necessary Table K14-1 on page K14-8038 disambiguates the general register references used in this chapter.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H4.1 Introduction

The *Debug Communications Channe*l, DCC, is a channel for passing data between the PE and an external agent, such as a debugger. The DCC provides a communications channel between:

- An external debugger, described as the *debug host*.
- The debug implementation on the PE, described as the *debug target*.

The DCC can be used:

- As a 32-bit full-duplex channel.
- As a 64-bit half-duplex channel.

The DCC is an essential part of Debug state operation and can also be used in Non-debug state.

The *Instruction Transfer Register*, ITR, passes instructions to the PE to execute in Debug state.

The PE includes flow-control mechanisms for both the DCC and ITR.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H4.2 DCC and ITR registers

The DCC comprises *data transfer registers*, the DTRs, and associated flow-control flags. The data transfer registers are DTRRX and DTRTX.

The ITR comprises a single register, EDITR, and associated flow-control flags.

In AArch64 state, software can access the data transfer registers as:

- A receive and transmit pair for 32-bit full duplex operation:
  - The write-only DBGDTRTX_EL0 register to transmit data.
  - The read-only DBGDTRRX_EL0 register to receive data.
- A single 64-bit read/write register, DBGDTR_EL0, for 64-bit half-duplex operation.
- The read/write OSDTRTX_EL1 and OSDTRRX_EL1 registers for save and restore.

In AArch32 state, software can only access the data transfer registers as:

- A receive and transmit pair, for 32-bit full duplex operation:
  - The write-only DBGDTRTXint register to transmit data.
  - The read-only DBGDTRRXint register to receive data.
- The read/write DBGDTRTXext and DBGDTRRXext registers for save and restore.

The data transfer registers are also accessible by the external debug interface as a pair of 32-bit registers, DBGDTRRX_EL0 and DBGDTRTX_EL0. Both registers are read/write, allowing both 32-bit full-duplex and 64-bit half-duplex operation.

The DCC flow-control flags are EDSCR.{RXfull, TXfull, RXO, TXU}:

- The RXfull and TXfull ready flags are used for flow-control and are visible to software in the Debug system registers in DCCSR.
- The RX overrun flag, RXO, and the TX underrun flag, TXU, report flow-control errors.
- The flow-control flags are also accessible by software as simple read/write bits for saving and restoring over a powerdown when the OS Lock is locked in DSCR.
- The flow-control flags are accessible from the external debug interface in EDSCR.

Figure H4-1 on page H4-6992 shows the System register and external debug interface views of the EDSCR and DTR registers in both AArch64 state and AArch32 state. These figures do not include the save and restore views.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

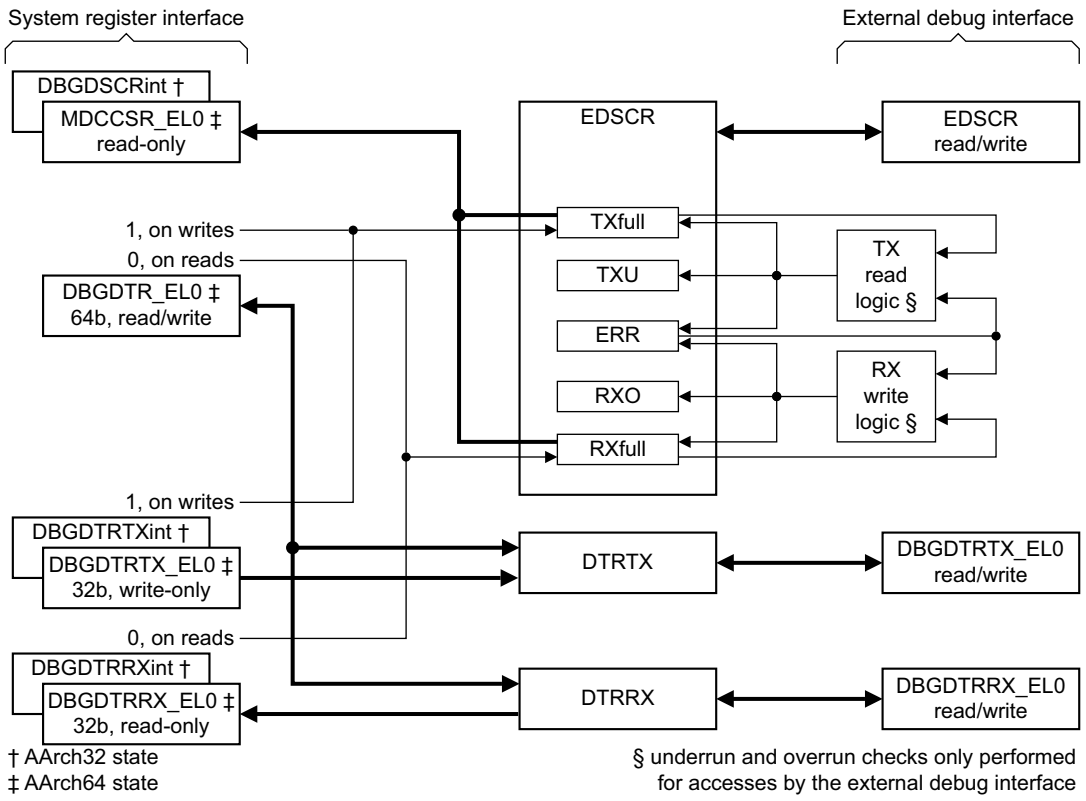


Figure H4-1 System register and external debug interface views of EDSCR and DTR registers, Normal access mode

EDITR and the ITR flow-control flags, EDSCR.{ITE, ITO} are accessible only by the external debug interface:

- The EDITR specifies an instruction to execute in Debug state.
- The ITR empty flag, ITE, is used for flow-control.
- The ITR overrun flag, ITO, reports flow-control errors.

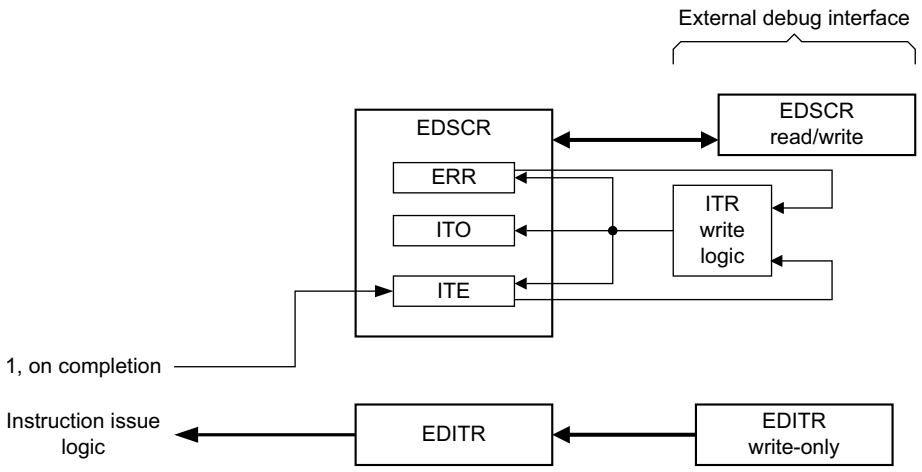


Figure H4-2 External debug interface views of EDSCR and EDITR registers, Normal access mode

The sticky overflow flag, EDSCR.ERR, is used by both the DCC and ITR to report flow-control errors.

To save and restore the DCC registers for an external debugger over powerdown, software uses:

- The MDSCR_EL1, OSDTRTX_EL1, and OSDTRRX_EL1 registers in AArch64 state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The DBGDSCRext, DBGDTRTXext, and DBGDTRRXext registers in AArch32 state.

Note

There is no save and restore mechanism for the ITR registers as the ITR is only used in Debug state.

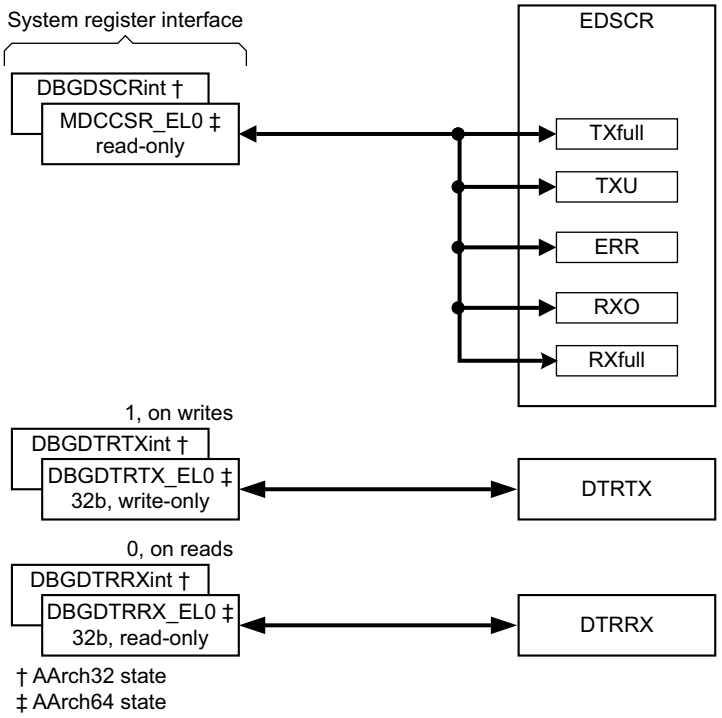


Figure H4-3 System register views of EDSCR and DTR registers for save and restore

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H4.3 DCC and ITR access modes

The DCC and ITR support two access modes:

DCC and ITR access mode, links to description	Applies when:
Normal access mode	EDSCR.MA == 0 or the PE is in Non-debug state
Memory access mode on page H4-6995	EDSCR.MA == 1 and the PE is in Debug state

H4.3.1 Normal access mode

The Normal access mode allows use of the DCC as a communications channel between target and host. It also allows the use of the ITR for issuing instructions to the PE in Debug state.

In Normal access mode, if there is no overrun or underrun, the following occurs:

For accesses by software:

- Direct writes to DBGDTRTX update the value in DTRTX and indirectly write 1 to TXfull.
- Direct reads from DBGDTRRX return the value in DTRRX and indirectly write 0 to RXfull.
- In AArch64 state, direct writes to DBGDTR_EL0 update both DTRTX and DTRRX, indirectly write 1 to TXfull, and do not change RXfull:
  - DTRTX is set from bits[31:0] of the transfer register.
  - DTRRX is set from bits[63:32] of the transfer register.
- In AArch64 state, direct reads from DBGDTR_EL0 return the concatenation of DTRRX and DTRTX, indirectly write 0 to RXfull, and do not change TXfull:
  - Bits[31:0] of the transfer register are set from DTRRX.
  - Bits[63:32] of the transfer register are set from DTRTX.

Note

For DBGDTR_EL0, the word order is reversed for reads with respect to writes.

Software reads TXfull and RXfull using DCCSR.

For accesses by the external debug interface:

- Writes to EDITR trigger the instruction to be executed if the PE is in Debug state:
  - If the PE is in AArch64 state, this is an A64 instruction.
  - If the PE is in AArch32 state, this is a T32 instruction. The T32 instruction is a pair of halfwords where the first halfword is taken from the lower 16-bits, and the second halfword is taken from the upper 16-bits.
- Reads of DBGDTRTX_EL0 return the value in DTRTX and indirectly write 0 to TXfull.
- Writes to DBGDTRTX_EL0 update the value in DTRTX and do not change TXfull.
- Reads of DBGDTRRX_EL0 return the value in DTRRX and do not change RXfull.
- Writes to DBGDTRRX_EL0 update the value in DTRRX and indirectly write 1 to RXfull.

TXfull and RXfull are visible to the external debug interface in EDSCR.

The PE detects overrun and underrun by the external debug interface, and records errors in EDSCR.{TXU, RXO, ITO, ERR}. See *Flow control of the DCC and ITR registers* on page H4-6998.

See also *Synchronization of DCC and ITR accesses* on page H4-7002.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H4.3.2 Memory access mode

When the PE is in Debug state, *Memory access mode* can be selected to accelerate word-aligned block reads or writes of memory by an external debugger. Memory access mode can only be enabled in Debug state, and no instructions can be issued directly by the debugger when in Memory access mode.

If there is no overrun or underrun when in Memory access mode, an access by the external debug interface results in the following:

- External reads from DBGDTRTX_EL0 cause:
  1. The existing value in DTRTX to be returned. This clears EDSCR.TXfull to 0.
  2. The equivalent of `LDR W1,[X0],#4`, if in AArch64 state, or `LDR R1,[R0],#4`, if in AArch32 state, to be executed.
  3. The equivalent of the `MSR DBGDTRTX_EL0,X1` instruction, if in AArch64 state, or the `MCR p14,0,R1,c0,c5,0` instruction, if in AArch32 state, to be executed.
  4. EDSCR.{TXfull, ITE} to be set to {1,1}, and X1 or R1 to be set to an UNKNOWN value.
- External writes to DBGDTRRX_EL0 cause:
  1. The value in DTRRX to be updated. This sets EDSCR.RXfull to 1.
  2. The equivalent of the instruction `MRS X1,DBGDTRRX_EL0`, if in AArch64 state, or `MRC p14,0,R1,c0,c5,0` if in AArch32 state, to be executed.
  3. The equivalent of the instruction `STR W1,[X0],#4`, if in AArch64 state, or `STR R1,[R0],#4`, if in AArch32 state, to be executed.
  4. EDSCR.{RXfull, ITE} to be set to {0,1}, and X1 or R1 to be set to an UNKNOWN value.
- External reads from DBGDTRRX_EL0 return the last value written to DTRRX.
- External writes to EDITR generate an overrun error.

During these accesses, EDSCR.{TXfull, RXfull, ITE} are used for flow control.

Note

An overrun or underrun might result in EDSCR.ERR being set to 1 asynchronously to the sequence of operations that are outlined in this section. As this is timing-dependent, it is UNPREDICTABLE when the EDSCR.ERR flag affects the instructions and therefore whether neither instruction, only the first instruction, or both instructions are executed. If the second instruction is executed, then the first instruction must have been executed. However, in each case X1 or R1 is set to an UNKNOWN value. This means that:

- In both cases, if the memory access instruction is not executed, then the base register X0 or R0 is not updated, meaning the debugger can determine the last accessed location.
- In the list describing External reads from DBGDTRTX_EL0, DTRTX and EDSCR.TXfull get set to UNKNOWN values. If the load was executed, then the value that was read by the PE is lost. This means the operation might need to be repeated by the debugger, and it is not advisable to use Memory access mode to read from read-sensitive locations using the underrun and overrun detection for flow control.
- In the list describing External writes to DBGDTRRX_EL0, EDSCR.RXfull is set to an UNKNOWN value.

A Data Abort from the memory access can also set EDSCR.ERR to 1. See *Data Aborts in Memory access mode* on page H4-6996.

The architecture does not require precisely when these flags are set or cleared by the sequence of operations outlined in this section. For example, in the case of an external write to DBGDTRRX_EL0, in AArch64 state, RXfull might be cleared after step 2, or it might not be cleared until after step 3, as an implementation is free to fuse these steps into a single operation. The architecture does require that the flags are set as at step 4 when the PE is ready to accept a further read or write without causing an overrun error or an underrun error.

The process outlined in this section represents a simple sequential execution model of Memory access mode. An implementation is free to pipeline, buffer, and re-order instructions and transactions, as long as the following remain true:

- Data items are transferred into and out of the DTR in order and without loss of data, other than as a result of an overrun or an underrun.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Data Aborts occur in order.
- The constraints of the memory type are met.
- In the list describing *External reads from DBGDTRTX_EL0* on page H4-6995:
  - The MSR equivalent operation at step 3 of the sequence reads the value loaded by step 2.
  - If the list is performed in a loop, for all but the first iteration of this list, the value read by step 1 returns the values written by the MSR equivalent operation at the previous iteration of step 3.
- In the list describing *External writes to DBGDTRRX_EL0* on page H4-6995:
  - The MRS equivalent operation at step 2 of the sequence returns the value written at step 1.
  - The STR equivalent at step 3 of the sequence writes the value read at step 2.
- If the PE cannot accept a read or write, as applicable, during the sequence, then the flags are updated to indicate an overrun or underrun.

See *Flow control of the DCC and ITR registers* on page H4-6998 for more information on overrun and underrun.

Ordering, access sizes and effect on Exclusives monitors

For the purposes of memory ordering, access sizes, and effect on the Exclusives monitor, accesses in Memory access mode are consistent with Load/Store word instructions executed by the PE.

The simple sequential access model of Memory-access mode, as stated in *Memory access mode* on page H4-6995, must also be ordered with respect to instructions executed as a result of explicit writes to EDITR in Normal mode both before and after accesses to the DTR registers in Memory-access mode.

Data Aborts in Memory access mode

If a memory access generates a Data Abort, then:

- The Data Abort exception is taken. See *Exceptions in Debug state* on page H2-6957:
  - This means EDSCR.ERR is set to 1, see *Cumulative error flag* on page H4-7000.
  - If the Data Abort occurs on stage 2 of an address translation, then the values returned in the ISV field and in bits[23:14] of the ISS are UNKNOWN.

    If this Data Abort is taken to EL2 using AArch64, the ISS is returned by ESR_EL2. *ISS encoding for an exception from a Data Abort* on page D13-2992 describes the usual encoding of this ISS.

    If EL2 is using AArch32 and this Data Abort is taken to Hyp mode, the ISS is returned by HSR. *ISS encoding for an exception from a Data Abort* on page G8-6303 describes the usual encoding of this ISS.
- Register R0 retains the address that generated the abort.
- Register R1 is set to an UNKNOWN value.
- EDSCR.TXfull, for a load, or EDSCR.RXfull, for a store, is set to an UNKNOWN value.
- DTRTX, for a load, or DTRRX, for a store, is set to an UNKNOWN value.
- EDSCR.ITE is set to 1.

Illegal Execution state exception

If PSTATE.IL is set to 1 when EDSCR.MA == 1, then on an external write access to DBGDTRRX_EL0 or an external read from DBGDTRTX_EL0, it is CONSTRAINED UNPREDICTABLE whether the PE:

- Takes an Illegal Execution state exception without performing any operations. In this case:
  - EDSCR.ERR is set to 1, see *Cumulative error flag* on page H4-7000.
  - Register R0 is unchanged.
  - Register R1 is set to an UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— EDSCR.TXfull or EDSCR.RXfull, as applicable, is set to an UNKNOWN value.
— DTRTX or DTRRX, as applicable, is set an UNKNOWN value.
— EDSCR.ITE is set to 1.

See also *Exceptions in Debug state* on page H2-6957.

- Ignores PSTATE.IL.

Note

The typical usage model for Memory access mode involves executing instructions in Normal access mode to set up X0 before setting EDSCR.MA to 1. These instructions generate an Illegal state exception if PSTATE.IL is set to 1.

Alignment constraints

If the address in R0 is not aligned to a multiple of four, the behavior is as follows:

- For each external DTR access a CONSTRAINED UNPREDICTABLE choice of:
  1. The PE makes an unaligned memory access to R0. If alignment checking is enabled for the memory access, this generates an Alignment fault.
  2. The PE makes a memory access to `Align(X[0],4)` in AArch64 state, or `Align(R[0],4)` in AArch32 state.
  3. The PE generates an Alignment fault, regardless of whether alignment checking is enabled.
  4. The PE does nothing.
- Following each memory access, if there is no Data Abort, R0 is updated with an UNKNOWN value.
- For external writes to DBGDTRRX_EL0, if the PE writes to memory, an UNKNOWN value is written.
- For external reads of DBGDTRTX_EL0 an UNKNOWN value is returned.
- The RXfull and TXfull flags are left in an UNKNOWN state, meaning that a DBGDTRTX_EL0 read can trigger a TX underrun, and a DBGDTRTX_EL0 write can trigger an RX overrun.

H4.3.3 Memory-mapped accesses to the DCC and ITR

Writes to the flags in EDSCR by external debug interface accesses to the DCC and the ITR registers are indirect writes, because they are a side-effect of the access. The indirect write might not occur for a memory-mapped access to the external debug interface. For more information, see *Register access permissions for memory-mapped accesses* on page H8-7054.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H4.4 Flow control of the DCC and ITR registers

- *Ready flags*.
- *Buffering writes to EDITR*.
- *Overrun and underrun flags*.
- *Cumulative error flag* on page H4-7000.

H4.4.1 Ready flags

In Normal access mode:

- For the DTR registers there are two ready flags:
  - EDSCR.RXfull == 1 indicates that DBGDTRRX_EL0 contains a valid value that has been written by the external debugger and not yet read by software running on the target.
  - EDSCR.TXfull == 1 indicates that DBGDTRTX_EL0 contains a valid value that has been written by software running on the target and not yet read by an external debugger.
- For the ITR register there is a single ready flag:
  - EDSCR.ITE == 1 indicates that the PE is ready to accept an instruction to the ITR.

    Note

    The architecture permits a PE to continue to accept and buffer instructions when previous instructions have not completed their architecturally defined behavior, as long as those instructions are discarded if EDSCR.ERR is set, either by an underrun or overrun or by any of the other error conditions described in this architecture, such as an instruction generating an abort.

In Memory access mode:

- EDSCR.{RXfull, ITE} == {0,1} indicates that DBGDTRRX_EL0 is empty and the PE is ready to accept a word external write to DBGDTRRX_EL0.
- EDSCR.{TXfull, ITE} == {1,1} indicates that DBGDTRTX_EL0 is full and the PE is ready to accept a word external read from DBGDTRTX_EL0.

All other values indicate that the PE is not ready, and result in a DTR overrun or underrun error, an ITR overrun error, or both, as defined in *Overrun and underrun flags*.

EDSCR.{ITE, RXfull, TXfull} shows the status of the ITR and DCC registers. It ignores the question of whether a read or write cannot be accepted because, for example, EDSCR.ERR is set or the OPTIONAL Software Lock is locked for memory-mapped accesses (EDLSR.SLK == 1).

H4.4.2 Buffering writes to EDITR

The architecture permits a processor to continue to accept and buffer instructions when previous instructions have not completed their architecturally defined behavior, provided that:

- Those instructions are discarded if EDSCR.ERR is set to 1, either by an underrun or an overrun, or by any other error conditions described in this architecture, such as an instruction generating an abort.
- The PE maintains the simple sequential execution model with the order of instructions determined by the order in which the PE accepts the EDITR writes. In particular, the buffered instructions must be executed in the Execution state consistent with a simple sequential execution of the instructions, even if one of the previous instructions is a state changing operation, such as `DCPS` or `DRPS`.

H4.4.3 Overrun and underrun flags

Each of the ready flags has a corresponding overrun or a corresponding underrun flag. These are sticky status flags that are set if the register is accessed using the external debug interface when the corresponding ready flag is not in the ready state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the PE is in Debug state and Memory access mode, the corresponding error flag is also set if the PE is not ready to accept an operation because a previous load or store is still in progress. The sticky status flag remains set until cleared by writing 1 to EDRCR.CSE.

Note

The architecture permits a PE to continue to accept and buffer data to write to memory in Memory access mode.

Table H4-1 shows DCC and ITR ready flags and the overrun and underrun flags associated with them.

Table H4-1 DCC and ITR ready flags and the associated overrun/underrun flags

External debug interface access	Overrun/Underrun condition	EDSCR flag
Write DBGDTRRX_EL0	`EDSCR.RXfull == '1' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0')`	RXO
Read DBGDTRTX_EL0	`EDSCR.TXfull == '0' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0')`	TXU
Write EDITR	`Halted() && (EDSCR.ITE == '0' || EDSCR.MA == '1')`	ITO

When an overrun or underrun flag is set to 1, the cumulative error flag, EDSCR.ERR, described in *Cumulative error flag* on page H4-7000, is also set to 1.

In the event of an external write to DBGDTRRX_EL0 or EDITR generating an overrun, or an external read from DBGDTRTX_EL0 generating an underrun:

- For a write, the written value is ignored.
- For a read, an UNKNOWN value is returned.
- EDSCR.TXfull, EDSCR.RXfull or EDSCR.ITE, as applicable, are not updated.

There is no overrun or underrun detection on external reads of DBGDTRRX_EL0 or external writes of DBGDTRTX_EL0.

There is no overrun or underrun detection of direct reads and direct writes of the DTR System registers by software:

- If RXfull == 0, a direct read of DBGDTRRX or DBGDTR_EL0 returns UNKNOWN.
- If TXfull == 1, a direct write of:
  - DBGDTRTX sets DTRTX to UNKNOWN.
  - DBGDTR_EL0 sets DTRRX and DTRTX to UNKNOWN.

See *DCC accesses in Non-debug state* on page H4-7003 for more information.

Accessing 64-bit data

In AArch64 state, a software access to the DBGDTR_EL0 register and an external debugger access to both DBGDTRRX_EL0 and DBGDTRTX_EL0 can perform a 64-bit half-duplex operation.

However, there is only overrun and underrun detection on one of the external debug registers. That is:

- If software directly writes a 64-bit value to DBGDTR_EL0, only TXfull is set to 1, meaning:
  - A subsequent external write to DBGDTRRX_EL0 would not be detected as an overrun.
  - If the external debugger reads DBGDTRTX_EL0 first, software might observe MDCCSR_EL0.TXfull == 0 and send a second value before the external debugger reads DBGDTRRX_EL0, leading to an undetected overrun.
- On external writes to both DBGDTRRX_EL0 and DBGDTRTX_EL0 only RXfull is set to 1, meaning:
  - A subsequent direct write of DBGDTRTX_EL0 would not be detected as an overrun.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— If the external debugger writes to DBGDTRRX_EL0 first, software might observe MDCCSR_EL0.RXfull == 1 and read a full 64-bit value, before the external debugger writes to DBGDTRTX_EL0, leading to an undetected underrun.

To avoid this, debuggers need to be aware of the data size used by software for transfers and ensure that 64-bit data is read or written in the correct order. If the PE is in Non-debug state, this order is as follows:

- The external debugger must check EDSCR.{RXfull, TXfull} before each transfer.
- To receive a 64-bit value from the target, the external debugger must read DBGDTRRX_EL0 before reading DBGDTRTX_EL0.
- To send a 64-bit value to the target, the external debugger must write to DBGDTRTX_EL0 before writing DBGDTRRX_EL0.

Because three accesses are required to transfer 64 bits of data, 64-bit transfers are not recommended for regular communication between host and target. The use of underrun and overrun detection means that only one access is required for 32 bits of data when using 32-bit transfers.

In Debug state, the debugger controls the instructions executed by the PE, so these limitations do not apply. 64-bit transfers provide a means to transfer a 64-bit general register between the host and the target in Debug state.

H4.4.4 Cumulative error flag

The cumulative error flag, EDSCR.ERR, is set to 1:

- On taking an exception from Debug state.
- On any signalled overrun or underrun in the DCC or ITR.

When EDSCR.ERR == 1:

- External reads of DBGDTRTX_EL0 do not have any side-effects.
- External writes to DBGDTRRX_EL0 are ignored.
- External writes to EDITR are ignored.
- No further instructions can be issued in Debug state. This includes any instructions previously accepted as external writes to EDITR that occur in program order after the instruction or access that caused the error.

This allows a debugger to stream data, or, in Debug state, instructions, to the target without having to:

- Check EDSCR.{RXfull, TXfull, ITE} before each access.
- Check EDSCR.{ITO, RXO, TXU} following each access, for overrun or underrun.
- Check PSTATE or other syndrome registers, or both, for an exception following each instruction executed in Debug state that might generate a synchronous exception.

The cumulative error flag remains set until cleared to 0 by writing 1 to EDRCR.CSE. However, the effect of writing 1 to EDRCR.CSE to clear EDSCR.ERR is CONSTRAINED UNPREDICTABLE when both of the following apply:

- The PE is in Debug state.
- The value of EDSCR.ITE is 0.

When these conditions apply and a value of 1 is written to EDRCR.CSE, either or both of the following might occur:

- EDSCR.ERR is not cleared to 0.
- Any instructions in EDITR that have not been executed might be executed subsequently, rather than being ignored.

Note

This means that a debugger must poll EDSCR.ITE until it has the value 1, indicating that EDITR is empty, before writing to EDRCR.CSE to clear the EDSCR.ERR flag to 0.

For overruns and underruns, EDSCR.{ITO, RXO, TXU} record the error type.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Pseudocode description of clearing the error flag

The `ClearStickyErrors()` pseudocode function is described in Chapter J1 *Armv8 Pseudocode*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H4.5 Synchronization of DCC and ITR accesses

In addition to the standard synchronization requirements for register accesses, the following subsections describe additional requirements that apply for the DCC and ITR registers:

- *Summary of System register accesses to the DCC*.
- *DCC accesses in Non-debug state* on page H4-7003.
- *Synchronization of DCC interrupt request signals* on page H4-7006.
- *DCC and ITR access in Debug state* on page H4-7006.

In these sections, accesses by the external debug interface are referred to as external reads and external writes. Accesses to System registers are referred to as direct reads, direct writes, indirect reads, and indirect writes.

Note

In *Synchronization requirements for AArch64 System registers* on page D13-2869 external reads and external writes are described as forms of indirect access. This whole section uses more explicit terminology.

The DTR registers and the DCC flags, TXfull and RXfull, form a communication channel, with one end operating asynchronously to the other. Implementations must respect the ordering of accesses to these registers in order to maintain the correct behavior of the channel.

External reads of, and external writes to DBGDTRRX_EL0 and DBGDTRTX_EL0 are asynchronous to direct reads of, and direct writes to, DBGDTRRX, DBGDTRTX, and in AArch64 state DBGDTR_EL0, made by software using System register access instructions. The direct reads and direct writes indirectly write to the DCC flags. The external reads and external writes indirectly read the DCC flags to check for underrun and overrun.

Throughout this section:

DCC flags Means any or all of the following:
- The EDSCR.{RXfull.TXfull} ready flags.
- The EDSCR.RXO overrun flag.
- The EDSCR.TXU underrun flag.
- The EDSCR.ERR cumulative error flag.

ITR flags Means any or all of the following:
- The EDSCR.ITE ready flag.
- The EDSCR.ITO overrun flag.
- The EDSCR.ERR cumulative error flag.

H4.5.1 Summary of System register accesses to the DCC

System register accesses to the DTR registers are direct reads and writes of those registers, as shown in Table H4-2 on page H4-7003. Several of these instructions access the same registers using different encodings.

DBGDTRRX_EL0 and DBGDTRTX_EL0 are encoded as MRS and MSR accesses respectively to the same System register, even though they access different underlying register values. DBGDTRRX and DBGDTRTX are similarly encoded as MRC and MCR accesses respectively to the same System register. The encoding means that direct reads and writes using these encodings must be ordered with respect to each other. For more information, see *Synchronization requirements for AArch64 System registers* on page D13-2869 and *Synchronization of changes to AArch32 System registers* on page G8-6100.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table H4-2 shows a summary of System register accesses to the DCC.

Table H4-2 Summary of System register accesses to the DCC

Operation	OS Lock	AArch64 (MRS/MSR)	AArch32 (MRC/MCR)	Description
Read	-	DBGDTRRX_EL0	DBGDTRRXint	Direct read of DTRRX Indirect write to the DCC flags An STC instruction that reads DBGDTRRXint makes an indirect write to DBGDSCRint.RXfull
Write	-	DBGDTRTX_EL0	DBGDTRTXint	Direct read of DTRTX Indirect write to the DCC flags An LDC instruction that writes to DBGDTRTXint using a value read from memory is a direct write to DBGDTRTXint
Read/write	-	DBGDTR_EL0	-	Direct read/write of both DTRRX and DTRTX Indirect write to the DCC flags
Read	-	MDCCSR_EL0	DBGDSCRint	Direct read of the DCC flags
Read/write	-	OSDTRRX_EL1	DBGDTRRXext	Direct read/write of DTRRX
Read/write	-	OSDTRTX_EL1	DBGDTRTXext	Direct read/write of DTRTX
Read	Unlocked	MDSCR_EL1	DBGDSCRext	Direct read of DCC flags
Read/write	Locked	MDSCR_EL1	DBGDSCRext	Direct read/write of DCC flags

H4.5.2 DCC accesses in Non-debug state

In Non-debug state DCC accesses are as described in *Normal access mode* on page H4-6994:

- If a direct read of DCCSR returns RXfull == 1, then a following direct read of DBGDTRRX, or in AArch64 state of DBGDTR_EL0, returns valid data and indirectly writes 0 to DCCSR.RXfull as a side-effect.
- If a direct read of DCCSR returns TXfull == 0, then a following direct write to DBGDTRTX, or in AArch64 state to DBGDTR_EL0, writes the intended value, and indirectly writes 1 to DCCSR.TXfull as a side-effect.

No *Context synchronization event* is required between these two instructions. Overrun and underrun detection prevents intervening external reads and external writes affecting the outcome of the second instruction.

The indirect write to the DCC flags as part of the DTR access instruction is made atomically with the DTR access.

Because a direct read of DBGDTRRX is an indirect write to DCCSR.RXfull, it must occur in program order with respect to the direct read of DCCSR, meaning it must not return a speculative value for DTTRX that predates the RXfull flag returned by the read of DCCSR. The direct write to DBGDTRTX must not be executed speculatively.

Direct reads of DBGDTRRX, or in AArch64 state DBGDTR_EL0, and DCCSR, must occur in program order with respect to other direct reads of the same register using the same encoding.

The following accesses have an implied order within the atomic access:

- In the simple sequential execution of the program the indirect write of the DCC flags occurs immediately after the direct DTR access.

  Note

  For an access to DBGDTR_EL0, this means the indirect write happens after both DBGDTRRX_EL0 and DBGDTRTX_EL0 have been accessed.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- In the simple sequential execution model, for an external read of DBGDTRTX_EL0 or an external write of DBGDTRRX_EL0:
  - The check of the DCC flags for overrun or underrun occurs immediately before the access.
  - If there is no underrun or overrun, the update of the DCC flags occurs immediately after the access.
  - If there is underrun or overrun, the update of the DCC underrun or overrun flags occurs immediately after the access.

All observers must observe the same order for accesses.

Note

These requirements do not create order where order does not otherwise exist. It applies only for ordered accesses.

Without explicit synchronization following external writes and external reads:

- The value written by the external write to DBGDTRRX_EL0 that does not overrun, must be observable to direct reads of DBGDTRRX and DBGDTR_EL0 in finite time.
- The DCC flags that are updated as a side-effect of the external write or external read must be observable:
  - To subsequent external reads of EDSCR.
  - To subsequent external reads of DBGDTRRX_EL0 when checking for underrun.
  - To subsequent external writes to DBGDTRTX_EL0 when checking for overrun.
  - To direct reads of DCCSR in finite time.

However, explicit synchronization is required to guarantee that a direct read of DCCSR returns up-to-date DCC flags. This means that if a signal is received from another agent that indicates that DCCSR must be read, an `ISB` is required to ensure that the direct read of DCCSR occurs after the signal has been received. This also synchronizes the value in DBGDTRRX, if applicable. However, if that signal is an interrupt exception triggered by **COMMIRQ**, **COMMTX**, or **COMMRX**, the exception entry is sufficient synchronization. See *Synchronization of DCC interrupt request signals* on page H4-7006.

Explicit synchronization is required following a direct read or direct write:

- To ensure that a value directly written to DBGDTRTX is observable to external reads of DBGDTRTX_EL0.
- To ensure that a value directly written to DBGDTR_EL0 is observable to external reads of DBGDTRTX_EL0 and DBGDTRRX_EL0.
- To guarantee that the indirect writes to the DCC flags that were a side-effect of the direct read or direct write have occurred, and therefore that the updated values are:
  - Observable to external reads of EDSCR.
  - Observable to external reads of DBGDTRRX_EL0 when checking for underrun.
  - Observable to external writes of DBGDTRTX_EL0 when checking for overrun.
  - Returned by a following direct read of DCCSR.

See also *Memory-mapped accesses to the DCC and ITR* on page H4-6997 and *Synchronization of changes to the external debug registers* on page H8-7050.

Note

These ordering rules mean that software:

- Must not read DBGDTRRX without first checking DCCSR.RXfull or if the previously-read value of DCCSR.RXfull is 0.

  It is not sufficient to read both registers and then later decide whether to discard the read value, as there might be an intervening write from the external debug interface.

- Must not write DBGDTRTX without first checking DCCSR.TXfull or if the previously-read value of DCCSR.TXfull is 1.

  The write to DBGDTRTX overwrites the value in DTRTX, and the external debugger might or might not have read this value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Must ensure there is an explicit *Context synchronization event* following a DTR access, even if not immediately returning to read DCCSR again. This synchronization operation can be an exception return.

Derived requirements

The rules for DCC accesses in Non-debug state are as follows:

- Following a direct read of DBGDTRRX when RXfull is 1:
  - — If an external write to DBGDTRRX checks the RXfull flag for overrun and observes that the value of RXfull is 0, the value returned by the previous direct read must not be affected by the external write.
  - — If an external read of EDSCR returns a RXfull value of 0, then the value returned by the previous direct read must not be affected by a following external write to DBGDTRRX, and the following external write does not overrun.
- Following a direct read of DBGDTR_EL0, when RXfull is 1:
  - — If an external write to DBGDTRRX checks the RXfull flag for overrun and observes that the value of RXfull is 0, the value returned by the previous direct read must not be affected by the external write nor by a following direct write to DBGDTRTX.
  - — If an external read of EDSCR returns a RXfull value of 0, then the value returned by the previous direct read must not be affected by subsequent external writes to DBGDTRRX and DBGDTRTX in any order, and the following external write of DBGDTRRX will not overrun.
- Following a direct write to DBGDTRTX, when TXfull is 0:
  - — If an external read of DBGDTRTX checks the TXfull flag for underrun and observes that the value of TXfull is 1, the value returned by the external read must be the value written by the previous direct write.
  - — If an external read of EDSCR returns a TXfull value of 1, then the value returned by a following external read of DBGDTRRX must be the value written by the previous direct read, and the subsequent external read will not underrun.
- Following a direct write to DBGDTR_EL0, when TXfull is 0:
  - — If an external read of DBGDTRTX checks the TXfull flag for underrun and observes that the value of TXfull is 1, the values returned by the external read and by a subsequent external read of DBGDTRRX must be the value written by the previous direct write.
  - — If an external read of EDSCR returns a TXfull value of 1, then the value returned by subsequent external reads of DBGDTRRX and DBGDTRTX, in any order, must be the value written by the previous direct read, and the subsequent external read of DBGDTRTX does not underrun.
- Following an external read of DBGDTRTX that does not underrun, if a direct read of DCCSR returns a TXfull value of 0, then the value returned by the external read must not be affected by a following direct write to DBGDTRTX.
- Following a first external read DBGDTRRX and a following second external read of DBGDTRTX that does not underrun, if a direct read of DCCSR returns a TXfull value of 0, then the values returned by the external reads must not be affected by a following direct write to DBGDTR_EL0.
- Following an external write to DBGDTRRX that does not overrun, if a direct read of DCCSR returns an RXfull value of 1, then the value returned by a following direct read of DBGDTRRX or DBGDTR_EL0 must be the value written by the previous external write.
- Following a first external write to DBGDTRTX and a following second external write to DBGDTRRX that does not overrun, if a direct read of DCCSR returns an RXfull value of 1, then the value returned by a subsequent direct read of DBGDTR_EL0 must return the values written by the previous external writes.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H4.5.3 Synchronization of DCC interrupt request signals

Following an external read or external write access to the DTR registers, the interrupt request signals, **COMMIRQ**, **COMMTX**, and **COMMRX**, must be updated in finite time without explicit synchronization.

The updated values must be observable to a direct read of DCCSR or DBGDTRRX, or a direct write of DBGDTRTX executed after taking an interrupt exception generated by the interrupt request. The updated values must also be observable to a direct write of DBGDTRTX executed after taking an interrupt exception generated by the interrupt request.

Note

The requirement that indirect writes to registers are observable to direct reads in finite time does not imply that all observers will observe the indirect write at the same time. For more information, see *Synchronization requirements for AArch64 System registers* on page D13-2869 and *Synchronization of changes to AArch32 System registers* on page G8-6100.

Following a direct read of DBGDTRRX or a direct write to DBGDTRRX, software must execute a *Context synchronization event* to guarantee the interrupt request signals have been updated in finite time. This synchronization operation can be an exception return.

H4.5.4 DCC and ITR access in Debug state

In Debug state, stricter observability rules apply for instructions issued through the ITR, to maintain communication between a debugger and the PE, without requiring excessive explicit synchronization.

In Normal access mode, without explicit synchronization:

- A direct read or direct write of the DTR registers by an instruction written to EDITR must be observable to an external write or an external read in finite time:
  - A direct read of DBGDTRRX must be observable to an external write of DBGDTRRX_EL0.
  - A direct read of DBGDTR_EL0 must be observable to an external write of DBGDTRRX_EL0 and DBGDTRTX_EL0.
  - A direct write of DBGDTRTX must be observable to an external read of DBGDTRTX_EL0.
  - A direct write of DBGDTR_EL0 must be observable to an external read of DBGDTRRX_EL0 and DBGDTRTX_EL0.

  This includes the indirect write to the DCC flags that occurs atomically with the access as described in *DCC accesses in Non-debug state* on page H4-7003.

  The subsequent external write or external read must observe either the old or the new values of both the DTR contents and DCC flags. If the old values are observed, this typically results in overrun or underrun, assuming the old values of the DCC flags indicate an overrun or underrun condition, as would normally be the case.

  This means the debugger can observe the direct read or direct write without explicit synchronization and without explicitly testing the DCC flags in EDSCR, because it can rely on overrun and underrun tests.
- External reads of DBGDTRTX_EL0 that do not underrun and external writes to DBGDTRRX_EL0 that do not overrun must be observable to an instruction subsequently written to EDITR on completion of the first external access. This includes the indirect write to the DCC flags.

  This means that without explicit synchronization and without the need to first check the DCC flags in DCCSR:
  - If the instruction is a direct read of DBGDTRRX, it observes the external write.
  - If the instruction is a direct write of DBGDTRTX, it observes the external read.
- Writes to EDITR that do not overrun commit an instruction for execution immediately. The instruction must complete execution in finite time without requiring any further operation by the debugger.
- After an external write to the EDITR, the ITR flags that are updated as a side effect of that write must be observable by:
  - An external read of the EDSCR that follows the external write to the EDITR.
  - When checking for overrun, another external write to the EDITR that follows the original external write to the EDITR.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In Memory access mode, these requirements shift to the instructions implicitly executed by external reads and external writes of the DTR registers, as described in *Memory access mode* on page H4-6995.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H4.6 Interrupt-driven use of the DCC

Arm recommends implementations provide a level-sensitive DCC interrupt request through the IMPLEMENTATION DEFINED interrupt controller as a private peripheral interrupt for the originating PE.

Note

- In addition to connection to the interrupt controller Arm also recommends **COMMIRQ**, **COMMTX**, and **COMMRX** signals that might be implemented for use by any legacy system peripherals.
- GICv3 reserves a private peripheral interrupt number for the **COMMIRQ** interrupt.

The DCCINT register provides a first level of interrupt masking within the PE, meaning only a single interrupt source, **COMMIRQ**, is needed at the interrupt controller.

See also *Synchronization of DCC interrupt request signals* on page H4-7006.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H4.7 Pseudocode description of the operation of the DCC and ITR registers

The basic operation of the DCC and ITR registers is shown by the following pseudocode functions. These functions do not cover the behavior when OSLSR.OSLK == 1, meaning that the OS Lock is locked:

- `DBGDTR_EL0[]`.
- `DBGDTRRX_EL0[]`.
- `DBGDTRTX_EL0[]`.
- `EDITR[]`.
- `CheckForDCCInterrupts()`.

For the definition of the DTR Registers, see *shared/debug/dccanditr/DTR* on page J1-7713.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter H5
The Embedded Cross-Trigger Interface

This chapter describes the embedded cross-trigger interface. It contains the following sections:

- *About the Embedded Cross-Trigger (ECT)* on page H5-7012.
- *Basic operation on the ECT* on page H5-7014.
- *Cross-triggers on a PE in an Armv8 implementation* on page H5-7018.
- *Description and allocation of CTI triggers* on page H5-7019.
- *CTI registers programmers' model* on page H5-7023.
- *Examples* on page H5-7024.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H5.1 About the Embedded Cross-Trigger (ECT)

The *Embedded Cross-Trigger*, ECT, allows a debugger to:

- Send trigger events to a PE. For example, this might be done to halt the PE.
- Send a trigger event to one or more PEs, or other system components, when a trigger event occurs on another PE or system component. For example, this might be done to halt all PEs when one individual PE halts.

Figure H5-1 shows the logical structure of an ECT.

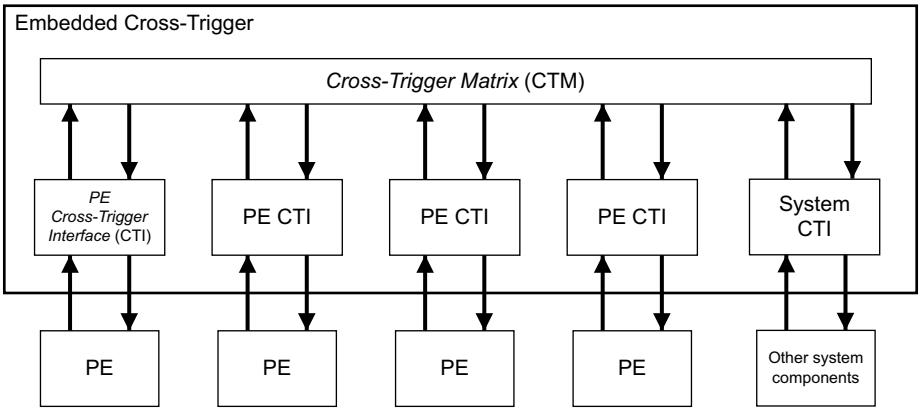


Figure H5-1 Structure of an embedded cross-trigger

The ECT can deliver many types of trigger events, which are described in the following sections:

- *Debug request trigger event* on page H5-7019.
- *Restart request trigger event* on page H5-7020.
- *Cross-halt trigger event* on page H5-7020.
- *Performance Monitors overflow trigger event* on page H5-7020.
- *Generic trace external input trigger events* on page H5-7021.
- *Generic trace external output trigger events* on page H5-7021.
- *Generic CTI interrupt trigger event* on page H5-7021.

An Armv8-A implementation must:

- Include a cross-trigger interface, CTI.
- Implement at least the input and output triggers defined in this architecture.

In addition, see *Cross-triggers on a PE in an Armv8 implementation* on page H5-7018.

Arm recommends that this cross-trigger interface includes:

- The ability to route trigger events between Trace Units, which typically have advanced event triggering logic.
- An output trigger to the interrupt controller.

Also, Arm recommends that the Embedded Cross-Trigger includes the capability to send and receive IMPLEMENTATION DEFINED system trigger events to and from other system components, including a system counter, using a system CTI. See *Halt-on-debug* on page I2-7234.

Note

The ECT and CTI must only signal trigger events for external debugging. They must not route software events, such as interrupts. For example, the Performance Monitors overflow input trigger is provided to allow entry to Debug state on a counter overflow, and the output trigger to the interrupt controller is provided to generally allow events from the external debug sub-system to be routed to a software agent. However, the combination of the two must not be used as a mechanism to route Performance Monitors overflows to an interrupt controller.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

CTI version 1 (CTIv1) is defined by the *CoreSight™ SoC Technical Reference Manual*. CTIv2 extends CTIv1 with the addition of the input channel gate see *Implementation with CTIv2*

H5.1.1 Implementation with a CoreSight CTI

For details of the recommended connections in an Armv8-A implementation, see Appendix K2 *Recommended External Debug Interface*. See also *CoreSight™ SoC Technical Reference Manual.*

H5.1.2 Implementation with CTIv2

If the CTI implemented is CTIv2 then:

- The CTIDEVARCH, CTIDEVAFF0, and CTIDEVAFF1 registers must be implemented.
- If the channel gate function is implemented, it applies to both input and output channels.
- The input channel gate function must be implemented if either of the following is true:
  - The CTM is implemented and the architecture variant is Armv8.5 or higher.
  - The CTIDEVARCH,REVISION field reads as `0b0001`or higher.

Implementation of CTIv2 features in architecture variants below Armv8.5 is OPTIONAL, but Arm recommends that CTIv2 is implemented, CTIv2 must be implemented from Armv8.5.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H5.2 Basic operation on the ECT

The ECT comprises a Cross-Trigger Matrix, CTM, and one Cross-Trigger Interface, CTI, for each PE. The ECT might also include other CTIs for other system components. The CTM passes events between the CTI blocks over channels. The CTM can have a maximum of 32 channels.

The main interfaces of the cross-trigger interface, CTI, are:

- The input triggers:
  - These are trigger event inputs from the PE to the CTI.
- The output triggers:
  - These are trigger event outputs from the CTI to the PE.
- The input channels:
  - These are channel event inputs from the cross-trigger matrix, CTM, to the CTI.
- The output channels:
  - These are channel event outputs from the CTI to the CTM.

Each CTI block has:

- Up to 32 input triggers that come from the PE:
  - The input triggers are numbered 0-31.
- Up to 32 output triggers that go to the PE:
  - The output triggers are numbered 0-31.

If the CTI is not powered up when the Core power domain is powered up, the CTI ignores all input triggers and input channel events, and does not generate any output triggers or output channel events.

Figure H5-2 on page H5-7015 shows the logical internal structure of a CTI.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

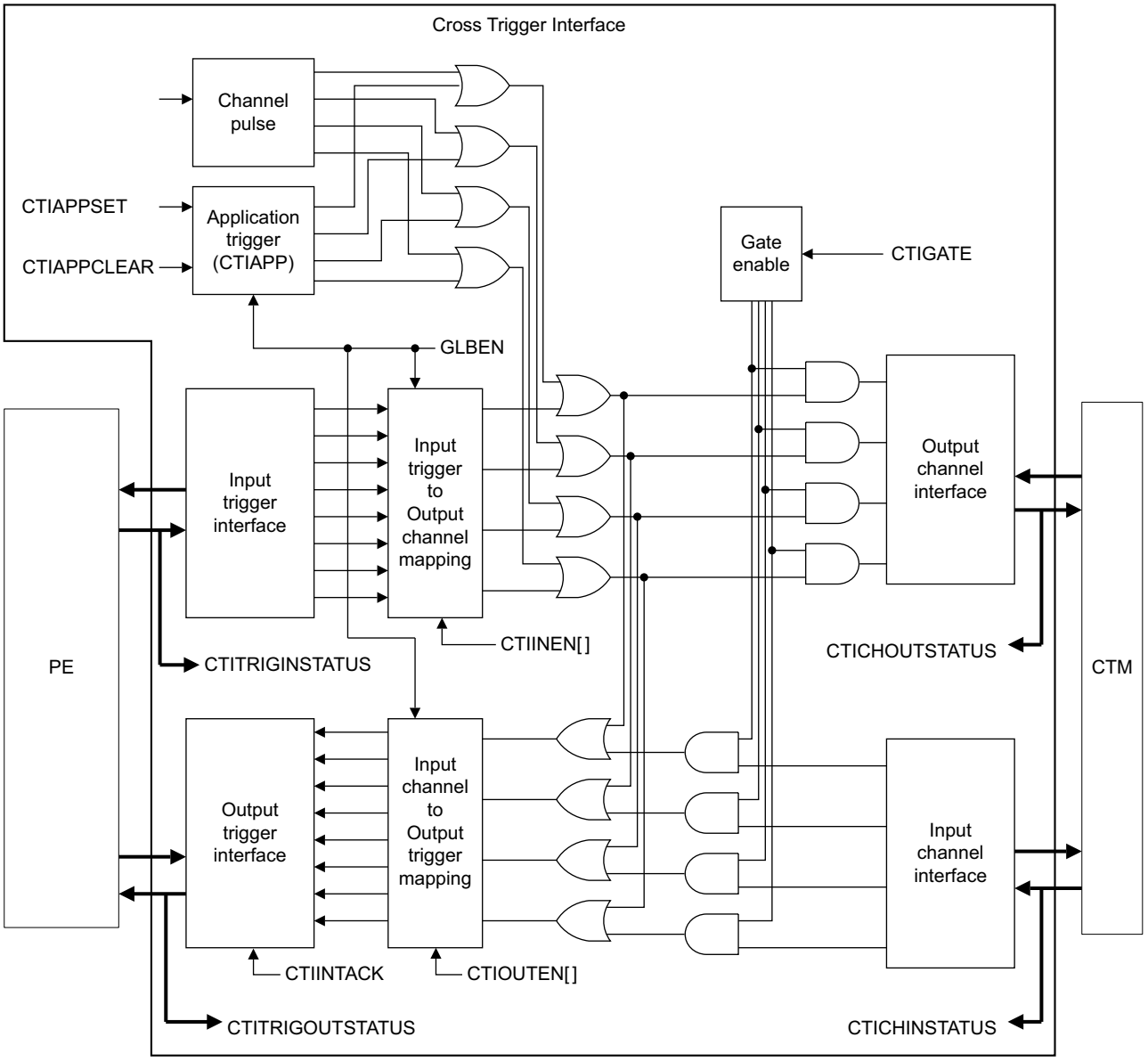


Figure H5-2 Structure of a cross-trigger interface

Note

- The number of triggers in IMPLEMENTATION DEFINED. Figure H5-2 shows eight input and eight output triggers.
- The number of channels is IMPLEMENTATION DEFINED. Figure H5-2 shows four channels.
- In Figure H5-2 the input channel gate function is a CTIv2 feature.

When the CTI receives an input trigger event, this generates channel events on one or more internal channels, according to the *mapping function* defined by the *Input trigger→output channel mapping registers*, CTIINEN<n>.

The CTI also contains an *application trigger* and *channel pulse* to allow a debugger to create channel events directly on internal channels by writing to the CTI control registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Channel events on each internal channel are passed to a corresponding output channel that is controlled by a *channel gate*. The channel gate can block propagation of channel events from an internal channel to an output channel.

Note

If the CTM is implemented:

- The gate function must be implemented.
- If the CTI is CTIv1, the gate function applies to output triggers only.

The output channels from a CTI are combined, using a logical OR function, with the output channels from all other CTIs to form the input channels on other CTIs. The input channels of this CTI are the logical OR of the output channels on all other CTIs. This is the *cross-trigger matrix*, CTM. Therefore, the number of input channels must equal the number of output channels.

Note

The number of input triggers and output triggers is not required to be the same.

The internal channels form an internal cross-trigger matrix within the CTI. This delivers events directly from the input triggers to the output triggers. Therefore the number of internal channels is the same as the number of input and output channels on the external CTM, and there is a direct mapping between the two.

Channel events received on each input channel are passed to the corresponding internal channel. It is IMPLEMENTATION DEFINED whether the cross-trigger gate also blocks propagation of channel events from input channels to internal channels.

Note

If CTIv2 is implemented, the cross-trigger gate also blocks propagation of channel events from input channels to internal channels.

When the CTI receives a channel event on an internal channel this generates trigger events on one or more output triggers, according to the mapping function defined by the *Input channel → output trigger mapping registers*, CTIOUTEN<n>.

The CTI contains the input and output trigger interfaces to the PE and the interface of the cross-trigger matrix. The architecture does not define the signal protocol used on the trigger interfaces, and:

- It is IMPLEMENTATION DEFINED whether the CTI supports multicycle input trigger events.
- It is IMPLEMENTATION DEFINED whether the CTM supports multicycle channel events.

See *Multicycle events* on page H5-7017.

However, an output trigger is asserted until acknowledged. The output trigger can be:

- Self-acknowledging. This means that no further action is required from the debugger.
- Acknowledged by the debugger writing 1 to the corresponding bit of CTIINTACK.

The time taken to propagate a trigger event from the first PE, through its CTI, across the CTM to another CTI, and thereby to a second PE is IMPLEMENTATION DEFINED.

Note

Arm recommends that this path is not longer than the shortest software communication path between those PEs. This is because if the first PE halts, the Cross-halt trigger event can propagate through the ECT and halt the second PE without causing software on the second PE to malfunction because the first PE is in Debug state and is not responding.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H5.2.1 Multicycle events

A multicycle event is one with a continuous state that might persist over many cycles, as opposed to a discrete event. A typical implementation of a multicycle event is a level-based signal interface, whereas a discrete event might be implemented as a pulse signal or message.

CTI support for multicycle trigger events is IMPLEMENTATION DEFINED. Use of multicycle trigger events is deprecated. Of the architecturally defined input trigger events, the Performance Monitors overflow trigger event and Generic trace external output trigger events can be multicycle input triggers.

CTM support for multicycle channel events is IMPLEMENTATION DEFINED. A CTM that does not support multicycle channel events cannot propagate a multicycle trigger event between CTIs.

Note

A full ECT might comprise a mix of CTIs, some of which can support multicycle trigger events. In bridging these components, multicycle channel events become single channel events at the boundary between the CTIs.

An ECT that supports multicycle trigger events

When an ECT supports multicycle trigger events, an input trigger event to the CTI continuously asserts channel events on all output channels mapped to it until either:

- The input trigger event is removed.
- The channel mapping function is disabled.

This means that an input trigger that is asserted for multiple cycles causes any channels that are mapped to it to become active for multiple cycles. Consequently, any output triggers mapped from that channel are asserted for multiple cycles.

Note

The output trigger remains asserted for at least as long as the channel remains active. This means that even if the output trigger is acknowledged, it remains asserted until the channel deactivates.

The CTI does not guarantee that these events have precisely the same duration, as the triggers and channels can cross between clock domains.

CTIAPPSET and CTIAPPCLEAR can set a channel active for multiple cycles. CTIAPPPULSE generates a single channel event. CTICHINSTATUS and CTICHOUTSTATUS can report whether a channel is active.

An ECT that does not support multicycle trigger events

When an ECT does not support multicycle trigger events, an input trigger event to the CTI generates a single channel event on all output channels mapped to it, regardless of how long the input trigger event is asserted.

This means that an input trigger event that is asserted for multiple cycles generates a single channel event on any channels mapped to it. Consequently any self-acknowledging output triggers mapped from those channels are single trigger events.

Note

A single event is typically a single cycle, but there is no guarantee that this is always the case.

CTIAPPSET and CTIAPPCLEAR can only generate a single channel event. CTIAPPPULSE generates a single channel event. If the ECT does not support multicycle channel events, use of CTIAPPSET and CTIAPPCLEAR is deprecated, and the debugger must only use CTIAPPPULSE. CTICHINSTATUS and CTICHOUTSTATUS must be treated as UNKNOWN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H5.3 Cross-triggers on a PE in an Armv8 implementation

An Armv8 PE must include a cross-trigger interface, and the implementation must include at least the input and output triggers defined in this architecture. The number of channels in the cross-trigger matrix is IMPLEMENTATION DEFINED, but there must be a minimum of three. Software can read CTIDEVID.NUMCHAN to discover the number of implemented channels.

The CTM must connect to all PEs in the same Inner Shareability domain as the Armv8-A PE, but can also connect to additional PEs. Arm strongly recommends that the CTM connects all PEs implementing a CTI in the system. This includes Armv7-A PEs and other PEs that can be connected using a CoreSight CTI module.

Note

In a uniprocessor system the CTM is OPTIONAL. In a multiprocessor system the CTM is required. The CTM might be connected other CTI modules for non-PEs, such as triggers for system visibility components. Arm recommends that the CTM is implemented.

Any CTI connected to a PE that is not an Armv8-A PE must implement at least:

- The Debug request trigger event.
- The Restart trigger event.
- The Cross-halt trigger event.

For more information about the CTI, see the *CoreSight ™ SoC Technical Reference Manual*. Armv8-A refines the generic CTI by defining roles for each of the implemented input and output triggers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H5.4 Description and allocation of CTI triggers

Table H5-1 shows the output trigger events defined by the architecture and the related trigger numbers.

Table H5-1 Allocation of CTI output trigger events

Number	Source	Destination	Event description
0	CTI	PE	*Debug request trigger event*
1	CTI	PE	*Restart request trigger event* on page H5-7020
2	CTI	IRQ controller	*Generic CTI interrupt trigger event* on page H5-7021
3	-	-	Reserved
4 - 7	CTI	PE Trace Unit	OPTIONAL *Generic trace external input trigger events* on page H5-7021

Note

Output triggers from the CTI are inputs to other blocks.

Table H5-2 shows the input trigger events defined by the architecture and the related trigger numbers.

Table H5-2 Allocation of CTI input trigger events

Number	Source	Destination	Event description
0	PE	CTI	*Cross-halt trigger event* on page H5-7020
1	PE	CTI	*Performance Monitors overflow trigger event* on page H5-7020
2	PE	CTI	*Statistical Profiling Extension sample trigger event* on page H5-7021
3	-	-	Reserved
4 - 7	PE Trace Unit	CTI	OPTIONAL *Generic trace external output trigger events* on page H5-7021

Note

Input triggers to the CTI are outputs from other blocks.

Table H5-1 and Table H5-2 show the minimum set of trigger events defined by the architecture. However:

- The Generic trace external input and output trigger events are only required if the OPTIONAL PE Trace Unit is implemented. If the OPTIONAL PE Trace Unit is not implemented, these trigger events are reserved.
- Support for the generic CTI interrupt trigger event is IMPLEMENTATION DEFINED because details of interrupt handling in the system, including any interrupt controllers, are IMPLEMENTATION DEFINED. Details regarding how the CTI interrupt is connected to an interrupt controller and its allocated interrupt number lie outside the scope of the architecture. Arm strongly recommends that implementations provide a means to generate interrupts based on external debug events.
- The other trigger events are required by the architecture.

An Armv8-A implementation can extend the CTI with additional triggers. These start with the number eight.

H5.4.1 Debug request trigger event

This is an output trigger event from the CTI, and an input trigger event to the PE, asserted by the CTI to force the PE into Debug state. The trigger event is asserted until acknowledged by the debugger. The debugger acknowledges the trigger event by writing 1 to CTIINTACK[0].

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

A debugger must poll CTITRIGOUTSTATUS[0] until it reads as 0, to confirm that the output trigger has been deasserted before generating any event that must be ordered after the write to CTIINTACK, such as a write to CTIAPPPULSE to activate another trigger.

If the PE is already in Debug state, the PE ignores the trigger event, but the CTI continues to assert it until it is removed by the debugger. See also *External Debug Request debug event* on page H3-6983.

H5.4.2 Restart request trigger event

This is an output trigger event from the CTI, and an input trigger event to the PE, asserted by the CTI to request the PE to exit Debug state. If the PE is in Non-debug state, the request is ignored by the PE.

If a Restart request trigger event is received at or about the same time as the PE enters Debug state, it is CONSTRAINED UNPREDICTABLE whether:

- The request is ignored by the PE. In this case the PE enters Debug state and remains in Debug state.
- The PE enters Debug state and then immediately restarts.

Debuggers must program the CTI to send Restart request trigger events only to PEs that are halted. To enable the PE to disambiguate discrete Restart request trigger events, after sending a Restart request trigger event, the debugger must confirm that the PE has restarted and halted before sending another Restart request trigger event. Debuggers can use EDPRSR.{SDR, HALTED} to determine the Execution state of the PE.

Note

Before generating a Restart request trigger event for a PE, a debugger must ensure any Debug request trigger event targeting that PE is cleared. *Debug request trigger event* on page H5-7019 describes how to do this.

The trigger event is self-acknowledging, meaning that the debugger requires no further action to remove the trigger event. The trigger event is acknowledged even if the request is ignored by the PE. See also *Exiting Debug state* on page H2-6963.

H5.4.3 Cross-halt trigger event

This is an input trigger event to the CTI, and an output trigger event from the PE, asserted by a PE when it is entering Debug state.

Note

To reduce the latency of halting, Arm recommends that an implementation issues the Cross-halt trigger event early in the committed process of entering Debug state. This means that there is no requirement to wait until all aspects of entry to Debug state have completed before issuing the trigger event. Speculative emission of Cross-halt trigger events is not allowed. The Cross-halt trigger event must not be issued early enough for a subsequent Debug request trigger event, that might be derived from the Cross-halt trigger event, to be recorded in the EDSCR.STATUS field. This applies to Debug request trigger events that are acting as inputs to the PE.

H5.4.4 Performance Monitors overflow trigger event

This is an input trigger event to the CTI, and an output trigger event from the PE, asserted each time the PE asserts a new Performance Monitors counter overflow interrupt request. See Chapter D7 *The Performance Monitors Extension*.

If the CTI supports multicycle trigger events, then the trigger event remains asserted until the overflow is cleared by a write to PMOVSCLR_EL0. Otherwise, the trigger event is asserted when the value of PMOVSCLR_EL0 changes from zero to a non-zero value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

- This does not replace the recommended connection of Performance Monitors overflow trigger event to an interrupt controller. Software must be able to program an interrupt on Performance Monitors overflow without programming the CTI.
- Events can be counted when `ExternalNoninvasiveDebugEnabled()`==FALSE, and, in Secure state, when `ExternalSecureNoninvasiveDebugEnabled()`==FALSE. Secure software must be aware that overflow trigger events are nevertheless visible to the CTI.

H5.4.5 Statistical Profiling Extension sample trigger event

If the Statistical Profiling Extension is implemented, and a sample record is written to memory, CTI input trigger 2 is asserted. This trigger might also be directly connected to other IMPLEMENTATION DEFINED debug features.

For more information see Chapter D9 *The Statistical Profiling Extension*.

H5.4.6 Generic trace external input trigger events

These are output trigger events from the CTI, and input trigger events to the OPTIONAL PE Trace Unit, that are used in conjunction with the Generic trace external output trigger events to pass trigger events between:

- The PE and the OPTIONAL PE Trace Unit.
- The OPTIONAL PE Trace Unit and any other component attached to the CTM, including other Trace Units.

There are four Generic trace external input trigger events.

The trigger events are self-acknowledging. This means that the debugger does not have to take any further action to remove the events.

H5.4.7 Generic trace external output trigger events

These are input trigger events to the CTI, and output trigger events from the OPTIONAL PE Trace Unit, used in conjunction with the Generic trace external input trigger events to pass trigger events between:

- The PE and the OPTIONAL PE Trace Unit.
- The OPTIONAL PE Trace Unit and any other component attached to the CTM, including other Trace Units.

There are four Generic trace external output trigger events.

H5.4.8 Generic CTI interrupt trigger event

This is an output trigger event from the CTI, and an input to an IMPLEMENTATION DEFINED interrupt controller, and can transfer trigger events from the PE, PE Trace Units, or any other component attached to the CTI and CTM to software as an interrupt. The Generic CTI interrupt trigger event must be connected to the interrupt controller as an interrupt that can target the originating PE.

Note

- Arm recommends that the Generic CTI interrupt trigger event is a private peripheral interrupt, but implementations might instead make this trigger event available as a shared peripheral interrupt or a local peripheral interrupt.
- GICv3 reserves a private peripheral interrupt number for this interrupt.

It is IMPLEMENTATION DEFINED whether this trigger event is:

- Self-acknowledging. This means that the debugger is not required to take any further action, and that the interrupt controller must treat the trigger event as a pulse or edge-sensitive interrupt.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Acknowledged by the debugger. The debugger acknowledges the trigger event by writing 1 to CTIINTACK[2]. This means that the interrupt controller must treat the trigger event as a level-sensitive interrupt.

Arm recommends that the Generic CTI interrupt trigger event is a self-acknowledging trigger event.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H5.5 CTI registers programmers' model

The CTI registers programmers' model is described in Chapter H8 *About the External Debug Registers*. The following sections contain information specific to the CTI:

- *External debug register resets* on page H8-7070.
- *External debug interface register access permissions* on page H8-7057.
- *Cross-trigger interface registers* on page H8-7068.
- The individual register descriptions in *Cross-Trigger Interface registers* on page H9-7177.

See also *Memory-mapped accesses to the external debug interface* on page H8-7054.

H5.5.1 CTI reset

An External Debug reset resets the CTI. See *External debug register resets* on page H8-7070 for details of CTI register resets. All CTI output triggers and output channels are deasserted on an External Debug reset.

Note

An indirect read of an output trigger might not observe the deasserted state until the processor is Cold reset. For more information, see *Synchronization of changes to the external debug registers* on page H8-7050.

H5.5.2 CTI authentication

The CTI ignores the state of the IMPLEMENTATION DEFINED authentication interface. This means that:

- CTITRIGINSTATUS shows the status of the input triggers and CTICHINSTATUS shows the status of the input channels, regardless of the value of `ExternalNoninvasiveDebugEnabled()`.

  Note

  The PE does not generate the Cross-halt trigger event and the PE Trace Unit does not generate Generic trace external output trigger events when `ExternalNoninvasiveDebugEnabled()`==FALSE. However, the PE can generate Performance Monitors overflow trigger events.

- The CTI can generate external triggers regardless of the value of `ExternalInvasiveDebugEnabled()`.

  Note

  The PE ignores Debug request and Restart request trigger events when `ExternalInvasiveDebugEnabled()`==FALSE. The PE Trace Unit ignores Generic trace external input trigger events when `ExternalNoninvasiveDebugEnabled()`==FALSE. The behavior of Generic CTI interrupt requests is part of the IMPLEMENTATION DEFINED handling of these interrupts, but it is permissible for an interrupt controller to receive these requests even when `ExternalInvasiveDebugEnabled()`==FALSE.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H5.6 Examples

The CTI is fully programmable and allows for flexible cross-triggering of events within a PE and between PEs in a multiprocessor system. For example:

- The Cross-halt trigger event and the Debug request trigger event can be used for cross-triggering in a multiprocessor system.
- The Cross-halt trigger event and the Generic interrupt trigger event can be used for event-driven debugging in a multiprocessor system.
- The Performance Monitors overflow trigger event and the Debug request trigger event can force entry to Debug state on overflow of a Performance Monitors event counter, for event-driven profiling.

  Note

  This does not replace the recommended connection of Performance Monitors overflow trigger events to an interrupt controller. Software must be able to program an interrupt on Performance Monitors overflow without programming the CTI. Arm recommends that the Performance Monitors overflow signal is directly available as a local interrupt source.

- The Generic trace external input and Generic trace external output trigger events can pass trace events into and out of the event logic of the PE Trace Unit. They can do this:
  - To pass trace events between Trace Units.
  - In conjunction with the Performance Monitors overflow trigger event, to couple the Performance Monitors to the PE Trace Unit.
  - In conjunction with the Debug request trigger event, to trigger entry to Debug state on a trace event.
  - In conjunction with other CTIs, to signal a trace trigger event onto a CoreSight trace interconnect.

The following sections describe some examples in more detail:

- *Halting a single PE*.
- *Halting all PEs in a group when any one PE halts* on page H5-7025.
- *Synchronously restarting a group of PEs* on page H5-7025.
- *Halting a single PE on Performance Monitors overflow* on page H5-7025.

Example H5-1 Halting a single PE

To halt a single PE, set:

1. CTIGATE[0] to 0, so that the CTI does not pass channel events on internal channel 0 to the CTM.
2. CTIOUTEN0[0] to 1, so that the CTI generates a Debug request trigger event in response to a channel event on channel 0.

   Note

   The Cross-halt trigger event is input trigger 0, meaning it is controlled by the instance of CTIOUTEN<n> for which <n> is 0.

3. CTIAPPPULSE[0] to 1, to generate a channel event on channel 0.

When the PE has entered Debug state, clear the Debug request trigger event by writing 1 to CTIINTACK[0], before restarting the PE.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Example H5-2 Halting all PEs in a group when any one PE halts

To program a group of PEs so that when one PE in the group halts, all of the PEs in that group halt, set the following registers for each PE in the group:

1. CTIGATE[2] to 1, so that each CTI passes channel events on internal channel 2 to the CTM.
2. CTIINEN0[2] to 1, so that each CTI generates a channel event on channel 2 in response to a Cross-halt trigger event.
3. CTIOUTEN0[2] to 1, so that each CTI generates a Debug request trigger event in response to a channel event on channel 2.

   Note

   The Cross-halt trigger event is input trigger 0, meaning it is controlled by the instances of CTIINEN<n> and CTIOUTEN<n> for which <n> is 0.

When a PE has halted, clear the Debug request trigger event by writing a value of 1to CTIINTACK[0], before restarting the PE.

Example H5-3 Synchronously restarting a group of PEs

To restart a group of PEs, for each PE in the group:

1. If the PE was halted because of a Debug request trigger event, the debugger must ensure the trigger event is deasserted. It can do this by:
   a. Writing 1 to CTIINTACK[0] to clear the Debug request trigger event.
   b. Polling CTITRIGOUTSTATUS[0], until it reads as 0, to confirm that the trigger event has been deasserted.
2. Set CTIGATE[1] to 1, so that each CTI passes channel events on internal channel 1 to the CTM.
3. Set CTIOUTEN1[1] to 1, so that each CTI generates a Restart request trigger event in response to a channel event on channel 1.

   Note

   This example must use the instance of CTIOUTEN<n> for which <n> is 1.

4. Set CTIAPPPULSE[1] to 1 on any one PE in the group, to generate a channel event on channel 1.

Example H5-4 Halting a single PE on Performance Monitors overflow

To halt a single PE on a Performance Monitors overflow set:

1. CTIGATE[3] to 0, so that the CTI does not pass channel events on internal channel 3 to the CTM.
2. CTIINEN1[3] to 1, so that the CTI generates a channel event on channel 3 in response to a Performance Monitors overflow trigger event.

   Note

   This step of this example must use the instance of CTIINEN<n> for which <n> is 1.

3. CTIOUTEN0[3] to 1, so that the CTI generates a Debug request trigger event in response to a channel event on channel 3.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

This step of this example must use the instance of CTIOUTEN<n> for which <n> is 0.

When the PE has entered Debug state, clear the Debug request trigger event by writing 1 to CTIINTACK[0], before restarting the PE. Clear the overflow status by writing to PMOVSCLR_EL0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter H6
Debug Reset and Powerdown Support

This chapter describes the reset and powerdown support in the Debug architecture. It contains the following sections:

- *About Debug over powerdown* on page H6-7028.
- *Power domains and debug* on page H6-7029.
- *Core power domain power states* on page H6-7030.
- *Emulating low-power states* on page H6-7033.
- *Powerup request mechanism* on page H6-7032.
- *Debug OS Save and Restore sequences* on page H6-7035.
- *Reset and debug* on page H6-7041.

Note

Where necessary, Table K14-1 on page K14-8038 disambiguates the general register references used in this chapter.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H6.1 About Debug over powerdown

Armv8 external debug defines a logical model for the hardware on which a PE executes. This hardware is logically split into the *Core power domain* and the *Debug power domain*, and the model contains descriptions of the states of those domains. See:

- *Power domains and debug* on page H6-7029.
- *Core power domain power states* on page H6-7030.

An implementation may allow power domains to be powered up and down independently. Debug over powerdown provides:

- A facility for software executing on the PE to save and restore the PE state on behalf of a self-hosted or external debugger or both. See *Debug OS Save and Restore sequences* on page H6-7035.
- A facility for an external debugger to request power up of the Core power domain. See *Powerup request mechanism* on page H6-7032.
- A facility for an external debugger, or software executing on the PE, to request emulation of powerdown of the Core power domain. See *Emulating low-power states* on page H6-7033.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H6.2 Power domains and debug

Armv8 external debug has two logical power domains, each with its own reset:

- The Debug power domain contains the interface between the PE and the external debugger, and is powered up whenever an external debugger is connected to the SoC. It remains powered up while the external debugger is connected. Registers in this domain are reset by an External Debug reset.
- The Core power domain contains the rest of the PE, and might be allowed to power up and power down independently of the Debug power domain.

Note

- The model of two logical power domains has an impact on the reset and access permission requirements of the debug programmers' model.
- The power domains are described as logical because the architecture defines the requirements but does not require two physical power domains. Any power domain split that meets the requirements of the programmers' model is a valid implementation.

The Core power domain contains several types of registers:

- Non-debug logic refers to all registers and logic that are not associated with debug.
- Self-hosted debug logic refers to registers and logic associated solely with the self-hosted debug aspects of the architecture.
- Shared debug logic refers to registers and logic associated with both the self-hosted and external debug aspects of the architecture.
- External debug logic refers to registers and logic associated solely with the external debug aspects of the architecture.

For information about which groups of registers and components are in each power domain, and which registers change power domain if ARMv8.3-DoPD is implemented, see:

- *Access permissions for the External debug interface registers* on page H8-7063.
- *Cross-trigger interface registers* on page H8-7068.
- *Management register access permissions* on page K2-7882.
- *Access permissions for external views of the Performance Monitors* on page I3-7243.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H6.3 Core power domain power states

The Arm architecture does not define the power states of the PE as these are not normally visible to software. However, they are visible to the external debugger. Armv8 external debug uses a four logical power states model for the Core power domain. The four logical power states are as follows:

Normal The Core power domain is fully powered up and the debug registers are accessible.

Standby The Core power domain is on, but there are measures to reduce energy consumption. In a typical implementation, the PE enters standby by executing a `WFI` or `WFE` instruction, and exits on a wake-up event. There can be other IMPLEMENTATION DEFINED measures the OS can take to enter standby.

The PE preserves the PE state, including the debug logic state. Changing from standby to normal operation does not involve a reset of the PE.

Standby is the least invasive OS energy saving state. Standby implies only that the PE is unavailable and does not clear any debug settings. For standby, the Debug architecture requires only the following:

- An External Debug Request debug event is a wake-up event when halting is allowed. This means that the PE must exit standby to handle the debug event. If the PE executed a `WFE` or a `WFI` instruction to enter standby, then it retires that instruction,
- If the external debug interface is accessed, the PE must respond to that access. Arm recommends that, if the PE executed a `WFI` or `WFE` instruction to enter standby, then it does not retire that instruction.

Standby is transparent, meaning that to software and to an external debugger it is indistinguishable from normal operation.

Retention The OS takes some measures, including IMPLEMENTATION DEFINED code sequences and registers, to reduce energy consumption. The PE state, including debug settings, is preserved in low-power structures, allowing the Core power domain to be at least partially turned off.

Changing from low-power retention to normal operation does not involve a reset of the PE. The saved PE state is restored on changing from low-power retention state to normal operation. If software has to use an IMPLEMENTATION DEFINED code sequence before entering, or after leaving, a retention state, this is referred to as a *software-visible retention state*. It is IMPLEMENTATION DEFINED whether the value of DBGPRCR.CORENPDRQ is set to the value of EDPRCR.COREPURQ on leaving the software-visible retention state.

External Debug Request debug events stay pending and registers in the Core power domain cannot be accessed.

Note

- This model of retention does not include implementations where the PE exits the state in response to a debug register access. From the Debug architecture perspective, implementations like this are forms of standby.

Powerdown The OS takes some measures to reduce energy consumption by turning the Core power domain off. These measures must include the OS saving any PE state, including the debug settings, that must be preserved over powerdown.

If ARMv8.0-DoubleLock is implemented, it is used during powerdown.

Changing from powerdown to normal operation must include:

- A Cold reset of the PE after the power level has been restored.
- The OS restoring the saved PE state.

External Debug Request debug events stay pending and debug registers in the Core power domain cannot be accessed.

An implementation might support enabling and disabling threads, either dynamically or once at reset time. Threads that are disabled in this way must appear to the external debugger as either:

- Powered off, meaning they are either:
  - In a powerdown state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

  — In a retention state.
- Held in reset state.

Armv8 external debug uses a simpler two states model for the Debug power domain. The two states are:

Off The Debug power domain is turned off.

On The Debug power domain is turned on.

The available power states, including the cross-product of Core power domain and Debug power domain power states is IMPLEMENTATION DEFINED. Implementations are not required to implement all of these states and might include additional states. These additional states must appear to the debugger as one of the logical power states defined by this model. The control of power states is IMPLEMENTATION DEFINED.

Note

As a result, it is IMPLEMENTATION DEFINED whether it is possible for the Debug power domain to be on when the Core power domain is off.

If the Debug power domain is implemented but is not powered up when the Core power domain is powered up, the Reset Catch debug event and the OS Unlock Catch debug event are disabled.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H6.4 Powerup request mechanism

If a powerup request mechanism is implemented, asserting the powerup request requests the power controller to power up the Core power domain, and to emulate any subsequent powerdown requests, until the powerup request mechanism is deasserted.

DBGPRCR.CORENPDRQ is set to the value of the powerup request at reset.

H6.4.1 Powerup request mechanism if ARMv8.3-DoPD is implemented

If ARMv8.3-DoPD is implemented, the external debug component implements an OPTIONAL powerup request mechanism.

If the powerup request mechanism is implemented, the powerup request must be a CoreSight Class 0x9 ROM table block that contains both:

- A parent entry for the debug registers of the PE.
- A parent entry for the PMU registers of the PE, if the OPTIONAL PMU with an external debug interface is implemented.

A parent entry of a component is an entry in a ROM table that either locates the component, or locates another ROM table that contains the parent entry for the component.

Note

The ROM table and any descendants might describe other debug components, including debug components for other PEs.

The ROM table might have a parent entry in a second ROM table and that parent entry might also have a powerup request mechanism in the second ROM table. This applies recursively.

The parent entries for the debug components have the following properties:

For the debug registers and Performance Monitors registers:

These components are in the Core power domain.

The POWERIDVALID bit is 1.

All parent entries must have the same IMPLEMENTATION DEFINED POWERID value.

Note

The IMPLEMENTATION DEFINED POWERID value does not need to be unique for each PE.

For the CTI registers:

This component is in the Debug power domain.

The POWERIDVALID bit is IMPLEMENTATION DEFINED.

If the POWERIDVALID bit is 1, the entries must have a valid POWERID value.

For more information about Coresight Class 0x9 ROM Tables, see *ARM® CoreSight™ Architecture Specification*.

H6.4.2 Powerup request mechanism if ARMv8.3-DoPD is not implemented

If ARMv8.3-DoPD is not implemented, the bit EDPRCR.COREPURQ is the powerup request mechanism.

The control registers DBGPRCR.CORENPDRQ and EDPRCR.COREPURQ provide an interface between the power controller and the PE. They typically map directly to signals in the recommended external debug interface.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H6.5 Emulating low-power states

DBGPRCR.CORENPDRQ and the powerup request mechanism can request the power controller to emulate states where the Core power domain is completely off or in a low-power state where the Core power domain registers cannot be accessed. This simplifies the requirements on software by sacrificing entirely realistic behavior.

If ARMv8.3-DoPD is not implemented, EDPRSR.{SPD, PU} indicates the Core power domain power state. For more information see:

- The DBGPRCR_EL1 and DBGPRCR System register descriptions.
- The EDPRCR and EDPRSR external debug register descriptions.
- Appendix K2 *Recommended External Debug Interface*.

The measures to emulate powerdown are IMPLEMENTATION DEFINED. The ability of the debugger to access the state of the PE and the system might be limited as a result of the measures adopted.

In an emulated powerdown state, the debugger must be able to access all debug, PMU, CTI, and trace unit registers that are accessible on the external debug interface and are in one of:

- The Debug power domain.
- The Core power domain.
- When a trace unit with a separate trace unit Core power domain is implemented, and the trace unit Core power domain is powered on, the trace unit Core power domain.

That is, the debugger must be able to read and write to such registers without receiving errors. This allows an external debugger to debug the powerup sequence.

Arm recommends that any IMPLEMENTATION DEFINED registers that are on the external debug interface and in either the Core power domain or the Debug power domain are also accessible in an emulated powerdown state.

If ARMv8.0-DoubleLock is implemented, `DoubleLockStatus() == FALSE` when DBGPRCR.CORENPDRQ == 1.

Otherwise, the behavior of the PE in emulated powerdown must be similar to that in a real powerdown state. In particular, the PE must not respond to other system stimuli, such as interrupts.

Example H6-1 and Example H6-2 are examples of two approaches to emulating powerdown.

Example H6-1 An example of emulating powerdown

The PE is held in Standby state, isolated from any system stimuli. It is IMPLEMENTATION DEFINED whether the PE can respond to debug stimuli such as an External Debug Request debug event.

If the PE can enter Debug state, then the external debugger is able to use the ITR to execute instructions, such as loads and stores. This causes the external debugger to interact with the system. If the external debugger restarts the PE, the PE leaves Standby state and restarts fetching instructions from memory.

Example H6-2 Another example of emulating powerdown

The PE is held in Warm reset. This limits the ability of an external debugger to access the resources of the PE. For example, the PE cannot be put into Debug state.

On exit from emulated powerdown the PE is reset. However, the debug registers that are only reset by a Cold reset must not be reset. Typically this means that a Warm reset is substituted for the Cold reset. As such, the effect of accessing any register that is reset by a Warm reset while the PE is in the emulated powerdown state will have an IMPLEMENTATION DEFINED effect on that register.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

- Warm reset and Cold reset have different effects apart from resetting the debug registers. In particular, RMR_ELx is reset by a Cold reset and controls the reset state on a Warm reset. This means that if a Cold reset is substituted by a Warm reset, the behavior of the reset code might be different.
- The timing effects of powering down are typically not factored in the powerdown emulation. Examples of these timing effects are clock and voltage stabilization.
- Emulation does not model the state lost during powerdown, meaning that it might mask errors in the state storage and recovery routines.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H6.6 Debug OS Save and Restore sequences

In Armv8-A, the following registers provide the OS Save and Restore mechanism:

- The *OS Lock Access Register*, OSLAR, locks the OS Lock to restrict access to debug registers before starting an OS Save sequence, and unlocks the OS Lock after an OS Restore sequence.
- The *OS Lock Status Register*, OSLSR, shows the status of the OS Lock.
- The PE can be configured to generate an *OS Unlock Catch debug event* on page H3-6984 when the OS Lock is unlocked.
- If ARMv8.0-DoubleLock is implemented, the OS Double Lock locks out an external debug interface entirely. This is only used immediately before a powerdown sequence.

See also:

- *ARMv8.0-DoubleLock* on page A2-66
- *Reset and debug* on page H6-7041
- Appendix K8 *Example OS Save and Restore Sequences*

H6.6.1 EDPRSR.{DLK, SPD, PU} and the Core power domain

If ARMv8.3-DoPD is not implemented, a debugger uses EDPRSR.{DLK, SPD, PU} to determine whether registers in the Core power domain can be accessed, and whether their state has been lost since the last time the register was read.

Table H6-1 Interpretation of the EDPRSR.{DLK, SPD, PU} bits

EDPRSR			Core power domain			Notes
DLK	SPD	PU	Power	Accesses	State lost	
0	0	1	On	OK	No	-
0	1	1	On	OK	Yes	SPD is cleared to 0 following the read.
1	X	1	On	Error	Not known	ARMv8.0-DoubleLock is implemented and `DoubleLockStatus()` == TRUE. Software locks the OS Double Lock before removing power.
X	1	0	Off	Error	Yes	A Cold reset will be asserted on exiting powerdown state, but not on exiting low-power retention state.
X	0	0	Not known	Error	Not known	

If ARMv8.3-DoPD is implemented, accesses to EDPRSR return an error when the Core power domain is off or in a retention state, meaning successful reads of EDPRSR always return 1 for EDPRSR.PU.

When ARMv8.4-Debug is implemented, and whenever ARMv8.0-DoubleLock is not implemented, EDPRSR.DLK is always 0.

If ARMv8.0-DoubleLock is not implemented, `DoubleLockStatus()` always returns FALSE.

If the Core power domain is powered up and `DoubleLockStatus()` == TRUE, then:

- When ARMv8.2-Debug is not implemented, EDPRSR.{DLK, SPD, PU} can read either {1, UNKNOWN, 1} or {UNKNOWN, 0, 0}.
- When ARMv8.2-Debug is implemented, and ARMv8.4-Debug is not implemented, EDPRSR.{DLK, SPD, PU} can only read {UNKNOWN, 0, 0}.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H6.6.2 EDPRSR.SPD when the Core domain is in either retention or powerdown state

If ARMv8.3-DoPD is not implemented, when the Core power domain is in either the retention or powerdown state, EDPRSR.SPD is not cleared following a read of EDPRSR and it is IMPLEMENTATION DEFINED whether:

- EDPRSR.SPD shows whether the state of the debug registers in the Core power domain has been lost since the last time that EDPRSR was read. This means that:
  - When the Core power domain is in the powerdown state, EDPRSR.SPD is RAO, this indicates that the state of the debug registers has been lost.
  - When the Core power domain is in the retention state, EDPRSR.SPD indicates whether the state of the debug registers was lost before the Core power domain entered retention state.
- EDPRSR.SPD is RAZ, and:
  - On leaving the powerdown state, EDPRSR.SPD is set to 1 which indicates that the state of the debug registers has been lost.
  - On leaving the retention state, EDPRSR.SPD reverts the value it had on entering the retention state.

Note

If ARMv8.3-DoPD is implemented, accesses to EDPRSR return an error when the Core power domain is off or in a retention state.

H6.6.3 EDPRSR.{DLK, R} and reset state

If ARMv8.3-DoPD is implemented, accesses to EDPRSR return an error when the Core power domain is off or in a retention state, meaning successful reads of EDPRSR always return 1 for EDPRSR.PU.

When ARMv8.4-Debug is implemented, and whenever ARMv8.0-DoubleLock is not implemented, EDPRSR.DLK is always 0.

If ARMv8.0-DoubleLock is not implemented, `DoubleLockStatus()` always returns FALSE.

If ARMv8.0-DoubleLock is implemented and enabled, the behavior of all registers and fields except EDPRSR.DLK is the same as their behavior if ARMv8.4-Debug is not implemented.

If ARMv8.4-Debug is implemented EDPRSR.DLK is always 0 and does not give any information about the OS Double Lock.

EDPRSR.R is UNKNOWN when `DoubleLockStatus()` == TRUE. OSDLR_EL1.DLK is cleared to 0 by a reset. If the Core power domain is powered up and entered reset state with the OS Double Lock locked, it is CONSTRAINED UNPREDICTABLE whether a read of EDPRSR while the PE is in reset state returns:

- EDPRSR.{DLK, R, PU} == {1, UNKNOWN, 1} indicating that the OS Double Lock is locked. This is not permitted from Armv8.2.
- EDPRSR.{DLK, R, PU} == {0, 1, 1} indicating that the PE is in reset state.
- EDPRSR.{DLK, R, PU} == {UNKNOWN, UNKNOWN, 0} indicating that the registers in the Core power domain cannot be accessed because the OS Double Lock is locked.

If the PE was powered up and the OS Double Lock was unlocked when the PE was reset, then EDPRSR.{DLK, R, PU} reads as {0, 1, 1} while the PE is in reset state.

On leaving reset state, EDPRSR.{DLK, R} reads as {0, 0}.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H6.6.4 Debug registers to save over powerdown

Table H6-2 shows the different requirements for self-hosted debug over powerdown and external debug over powerdown:

- The column labeled Self-hosted lists registers that software must preserve over powerdown so that it can support self-hosted debug over powerdown. This does not require use of the OS Save and Restore mechanism.
- The column labeled External lists registers that software must preserve over powerdown so that it can support external debug over powerdown. This requires use of the OS Save and Restore mechanism:
  - Some external debug registers are not normally accessible to software executing on the PE. Additional debug registers are provided that give software the required access to save and restore these external debug registers when OSLSR.OSLK is locked. These registers include OSECCR, OSDTRRX, and OSDTRTX.
- Some registers might only present in some implementations, or might not be accessible at all Exception levels or in Non-secure state. DBGVCR32_EL2 and SDER32_EL3 are only required to support AArch32.

Table H6-2 does not include registers for the OPTIONAL Trace and Performance Monitor extensions.

Table H6-2 Debug registers to save over powerdown

Register in AArch64 state	Register in AArch32 state	Self-hosted	External
MDSCR_EL1	DBGDSCRext	Yes	Yes[a]
DBGBVR<n>_EL1	DBGBVR<n>	Yes	Yes
DBGBCR<n>_EL1	DBGBCR<n>	Yes	Yes
DBGWVR<n>_EL1	DBGWVR<n>	Yes	Yes
DBGWCR<n>_EL1	DBGWCR<n>	Yes	Yes
DBGVCR32_EL2	DBGVCR	Yes	-
MDCR_EL2	HDCR	Yes	-
SDER32_EL3	SDER	Yes	-
MDCR_EL3	SDCR	Yes[b]	-
MDCCINT_EL1	DBGDCCINT	-	Yes[b]
DBGCLAIMSET_EL1 DBGCLAIMCLR_EL1	DBGCLAIMSET, DBGCLAIMCLR	-	Yes[c]
OSECCR_EL1	DBGOSECCR	-	Yes[ab]
OSDTRRX_EL1 OSDTRTX_EL1	DBGDTRRXext DBGDTRTXext	-	Yes

a. The OS Lock must be locked to save and restore for external debug. When the OS Lock is locked, DSCR is part of the software save and restore mechanism for external debug. It provides a mechanism for an operating system to access some fields of EDSCR that are otherwise read-only or not visible to software. This allows the operating system to save and restore these settings over a powerdown for the external debugger.

b. This register is new in Armv8-A. Sequences written for Armv7 do not preserve the register over powerdown.

c. Read DBGCLAIMCLR to save, write DBGCLAIMSET to restore.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H6.6.5 OS Save sequence

To preserve the debug logic state over a powerdown, the state must be saved to nonvolatile storage. This means the OS Save sequence must:

1. Lock the OS Lock by:
   - Writing the key value `0xC5ACCE55` to the DBGOSLAR in AArch32 state.
   - Writing 1 to OSLAR_EL1.OSLK in AArch64 state.
2. Execute an `ISB` instruction.
3. Walk through the debug registers listed in *Debug registers to save over powerdown* on page H6-7037 and save the values to the nonvolatile storage.

If the ARMv8.0-DoubleLock is implemented, before removing power from the Core power domain, software must:

1. Lock the OS Double Lock by:
   - Writing 1 to DBGOSDLR.DLK in AArch32 state.
   - Writing 1 to OSDLR_EL1.DLK in AArch64 state.

   If ARMv8.0-DoubleLock is not implemented, OSDLR_EL1 and DBGOSDLR ignore writes.
2. Execute a *Context synchronization event*.

H6.6.6 OS Restore sequence

After a powerdown, the OS Restore sequence must perform the following steps to restore the debug logic state from the non-volatile storage:

1. Lock the OS Lock, as described in *OS Save sequence*. The OS Lock is generally locked by the Cold reset, but this step ensures that it is locked.
2. Execute an `ISB` instruction.
3. To ensure that, if an external debugger clears the OS Lock before the end of this sequence, no debug exceptions are generated:
   - Write 0 to MDSCR_EL1 if executing in AArch64 state.
   - Write 0 to DBGDSCRext if executing in AArch32 state.
4. Walk through the debug registers listed in *Debug registers to save over powerdown* on page H6-7037, and restore the values from the nonvolatile storage. The last register to be restored must be:
   - MDSCR_EL1 if executing in AArch64 state.
   - DBGDSCRext if executing in AArch32 state.
5. Execute an `ISB` instruction.
6. Unlock the OS Lock by:
   - Writing any non-key value to DBGOSLAR if executing in AArch32 state.
   - Writing 0 to OSLAR_EL1.OSLK if executing in AArch64 state.
7. Execute a *Context synchronization event*.

Note

The OS Restore sequence overwrites the debug registers with the values that were saved. If there are valid values in these registers immediately before the restore sequence, then those values are lost.

H6.6.7 Debug behavior when the OS Lock is locked

The main purpose of the OS Lock is to prevent updates to debug registers during an OS Save or OS Restore operation. The OS Lock is locked on a Cold reset.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When the OS Lock is locked:

- Access to debug registers through the System register interface is mainly unchanged except that:
  - Certain registers are read and written without side-effects.
  - Fields in DSCR and OSECCR that are normally read-only become read/write.

  This allows the state to be saved or restored. For more information, see the relevant register description in Chapter H9 *External Debug Register Descriptions*.
- Access to debug registers by the external debug interface is restricted to prevent an external debugger modifying the registers that are being saved or restored. For more information see *External debug interface register access permissions summary* on page H8-7059.
- Debug exceptions, other than Breakpoint Instruction exceptions are not generated.
- Breakpoint and Watchpoint debug events are not generated. The OS Lock has no effect on Breakpoint Instruction exceptions and other debug events.

H6.6.8 Debug behavior when the OS Lock is unlocked

When the OS Lock is unlocked, the PE sets EDESR.OSUC to 1 if the OS Unlock Catch debug event is enabled and the PE is in Non-debug state, meaning the OS Unlock Catch debug event becomes pending. See *OS Unlock Catch debug event* on page H3-6984.

H6.6.9 Debug behavior when the OS Double Lock is locked

If the ARMv8.0-DoubleLock is implemented, software locks the OS Double Lock immediately before a powerdown sequence.

The OS Double Lock ensures that it is safe to remove core power by forcing the debug interfaces to be quiescent.

When `DoubleLockStatus()` == TRUE:

- The external debug interface only has restricted access to the debug registers, so that it is quiescent before removing power. See *External debug interface register access permissions summary* on page H8-7059.
- Debug exceptions, other than Breakpoint Instruction exceptions, are not generated.
- Halting is prohibited. See *Halting allowed and halting prohibited* on page H2-6927.

  Note

  Pending Halting debug events might be lost when core power is removed.

- No asynchronous debug events are `WFI` or `WFE` wake-up events.

If the ARMv8.0-DoubleLock is not implemented, the PE ensures these conditions are met before allowing power to be removed.

Software must synchronize the update to OSDLR before it indicates to the system that core power can be removed. The interface between the PE and its power controller is IMPLEMENTATION DEFINED.

Typically software indicates that core power can be removed by entering the Wait For Interrupt state. This means that software must explicitly synchronize the OSDLR update before issuing the `WFI` instruction.

OSDLR.DLK is ignored and `DoubleLockStatus()` == FALSE if either:

- The PE is in Debug state.
- DBGPRCR.CORENPDRQ is set to 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

It is possible to enter Debug state with OSDLR.DLK set to 1. This is because a *Context synchronization event* is required to ensure the OS Double Lock is locked, meaning that Debug state might be entered before the OSDLR update is synchronized.

Because OSDLR.DLK is ignored when DBGPRCR.CORENPDRQ is set to 1, an external debugger can write to DBGPRCR.CORENPDRQ, and the ARMv8.0-DoubleLock is not always implemented, software must not rely on using the OS Double Lock to disable debug exceptions or to prohibit halting, or both. Arm deprecates use of the OS Double Lock for these purposes, and instead recommends that software:

- Uses the OS Lock to disable debug exceptions during save or restore sequences.
- Uses the debug authentication interface to prohibit halting and external debug access to debug registers at times other than immediately prior to removing power.

As the purpose of the OS Double Lock is to ensure that it is safe to remove core power, if the ARMv8.0-DoubleLock is implemented, it is important to avoid race conditions that defeat this purpose. Arm recommends that:

- Once the write to OSDLR.DLK has been synchronized by a *Context synchronization event* and `DoubleLockStatus()` == TRUE, a PE must:
  - Not allow a debug event generated before the *Context synchronization event* to cause an entry to Debug state or act as a wake-up event for a `WFI` or `WFE` instruction after the *Context synchronization event* has completed.
  - Complete any external debug access started before the *Context synchronization event* by the time the *Context synchronization event* completes.

    Note

    A debug register access might be in progress when software sets OSDLR.DLK to 1. An implementation must not permit the synchronization of locking the OS Double Lock to stall indefinitely while waiting for that access to complete. This means that any debug register access that is in progress when software sets OSDLR.DLK to 1 must complete or return an error in finite time.

- If a write to DBGPRCR or EDPRCR made when OSDLR.DLK == 1 changes DBGPRCR.CORENPDRQ or EDPRCR.CORENPDRQ from 1 to 0, meaning `DoubleLockStatus()` changes from FALSE to TRUE, then before signaling to the system that the CORENPDRQ field has been cleared and emulation of powerdown is no longer requested, meaning the system can remove core power, the PE must ensure that all the requirements for `DoubleLockStatus()` == TRUE listed in this section are met.

In a standard OS Save sequence, the OS Lock is locked before the OS Double Lock is locked. This means that writes to CORENPDRQ are ignored by the time the OS Double Lock is locked. However, if `DoubleLockStatus()` == FALSE, an external debugger can clear the OS Lock at any time, and then write to EDPRCR.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H6.7 Reset and debug

All registers in the Core power domain are either:

- Reset by both a Cold and a Warm reset.
- Reset only by a Cold reset and are not changed by a Warm reset.

For more information, see *Reset* on page D1-2335.

All registers in the Debug power domain are reset by an External Debug reset.

Figure H6-1 shows this reset scheme. The following three reset signals are an example implementation of the reset scheme:

- **CORERESET**, which must be asserted for a Warm reset.
- **CPUPORESET**, which must be asserted for a Cold reset.
- **PRESETDBG**, which must be asserted for an External Debug reset.

As shown in the figure, the external debug logic is split between the Debug power domain and the Core power domain.

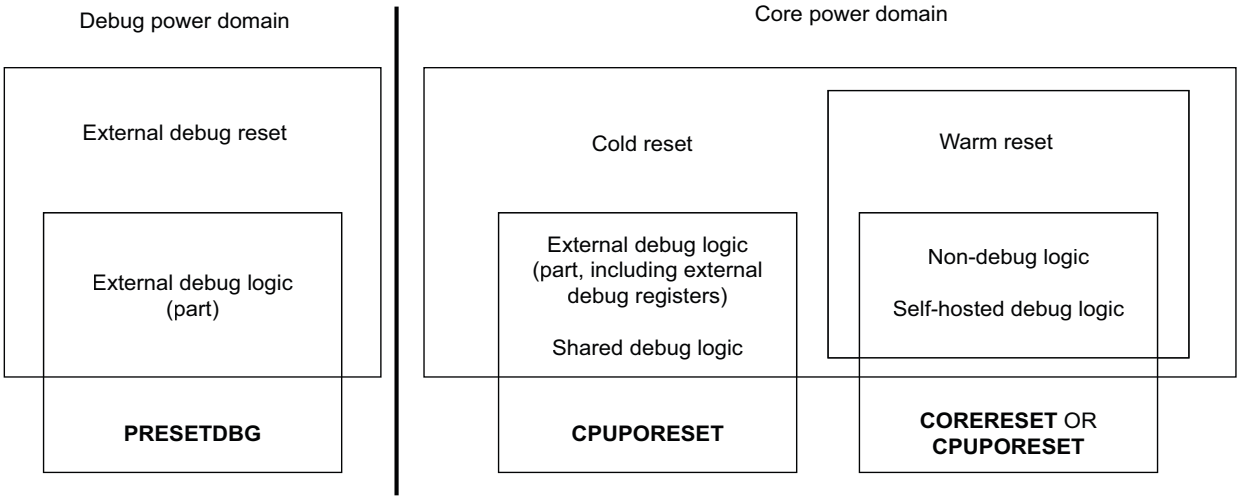


Figure H6-1 Power and reset domains

For more information about power domains and power states, see *Power domains and debug* on page H6-7029.

When power is first applied to the Debug power domain, **PRESETDBG** must be asserted.

When power is first applied to the Core power domain, **CPUPORESET** must be asserted.

Note

In this scheme, logic in the Warm reset domain is reset by asserting either **CORERESET** or **CPUPORESET**. This implies a particular implementation style that permits these approaches.

CPUPORESET is not normally asserted on moving from a low-power state, where power has not been removed, to a full-power state. This can occur, for example, on exiting a low-power retention state. See also *Emulating low-power states* on page H6-7033 and the EDPRSR register description.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H6.7.1 External debug interface accesses to registers in reset

If a reset signal is asserted and the external debug interface:

- Writes a register, or indirectly writes a register or register field as a side-effect of an access:
  - Then, if the register or register field is reset by that reset signal, it is CONSTRAINED UNPREDICTABLE whether the register or register field takes the reset value or the value written. The reset value might be UNKNOWN.
  - Otherwise the register or register field takes the value that is written.
- Reads a register, or indirectly reads a register or register field, as part of an access:
  - Then, if the register or register field is reset by that reset signal, the value returned in UNKNOWN.
  - Otherwise, the value of the register or register field is returned.

It is IMPLEMENTATION DEFINED whether any register can be accessed when External Debug reset is being asserted. The result of these accesses is IMPLEMENTATION DEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter H7
The PC Sample-based Profiling Extension

This chapter describes the OPTIONAL PC Sample-based Profiling Extension that provides a non-invasive external debug component.

It contains the following section:

- *About the PC Sample-based Profiling Extension* on page H7-7044.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H7.1 About the PC Sample-based Profiling Extension

The PC Sample-based Profiling Extension is an OPTIONAL extension that provides coarse-grained, non-invasive profiling by an external debugger. See also *Non-invasive behavior* on page D7-2703.

PC Sample-based Profiling creates samples so that tools can populate a statistical model of the performance of software executing on the PE.

Note

Data returned by periodic sampling of PC Sample-based Profiling registers is sufficient to allow tools to estimate the distribution of time spent executing software on the PE.

The delay between an instruction being executed by the PE and its address appearing in the PC Sample Register is not defined, and Armv8 does not require that the sampled instruction was recently executed. For example, if a piece of software executes a load instruction that reads the PC Sample Register of the PE it is running on, there is no guaranteed relationship between the address of the load instruction and the value read. The PC Sample Register is intended only for use by an external agent to provide statistical information for software profiling.

It must be possible to sample references to branch targets. It is IMPLEMENTATION DEFINED whether references to other instructions can be sampled. The branch target for a conditional branch instruction that fails its condition check is the instruction that follows the conditional branch instruction. The branch target for an exception is the exception vector address.

To keep the implementation and validation cost low, a reasonable degree of inaccuracy in the sampled data is acceptable. Arm does not define *a reasonable degree of inaccuracy* but recommends the following guidelines:

- In exceptional circumstances, such as a change in Security state or other boundary condition, it is acceptable for the sample to represent an instruction that was not committed for execution.
- Under unusual non-repeating pathological cases, the sample can represent an instruction that was not committed for execution. These cases are likely to occur as a result of asynchronous exceptions, such as interrupts, where the chance of a systematic error in sampling is very unlikely.
- Under normal operating conditions, the sample must reference an instruction that was committed for execution, including its context, and must not reference instructions that are fetched but not committed for execution.

Note

In the Armv7 PC Sample-based Profiling Extension, an offset was applied to the sampled program counter value and this offset and the instruction set state indicated in bits [1:0] of the sampled value. In the Armv8 PC Sample-based Profiling Extension, the sampled value is the address of an instruction that has executed, with no offset and no indication of the instruction set state.

- *Controlling the PC Sample-based Profiling Extension*.
- *Registers implemented by the PC Sample-based Profiling Extension* on page H7-7045.
- *Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN* on page H7-7046.
- *Pseudocode description of PC Sample-based Profiling* on page H7-7046.

H7.1.1 Controlling the PC Sample-based Profiling Extension

PC Sample-based Profiling is controlled by the IMPLEMENTATION DEFINED authentication interface `ExternalNoninvasiveDebugEnabled()`.

PC Sample-based Profiling is prohibited unless both:

- It is allowed by the IMPLEMENTATION DEFINED authentication interface `ExternalNoninvasiveDebugEnabled()`.
- At least one of the following applies:
  - The PE is executing in Non-secure state.
  - EL3 is not implemented.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— EL3 is implemented, the PE is executing in Secure state, and non-invasive debug is allowed by the IMPLEMENTATION DEFINED authentication interface `ExternalSecureNoninvasiveDebugEnabled()`.

— EL3 is implemented, EL3 or EL1 is using AArch32, the PE is executing at EL0 in Secure state, and the value of SDER.SUNIDEN is 1.

The state of the IMPLEMENTATION DEFINED authentication interface is visible through DBGAUTHSTATUS_EL1. See *Recommended authentication interface* on page K2-7879.

H7.1.2 Registers implemented by the PC Sample-based Profiling Extension

The options for implementing the PC Sample-based Profiling extension are:

- The extension is implemented in the external debug register space. EDDEVID.PCSample and identifies the implemented level of profiling, and EDDEVID1.PCSROffset also indicates that this option is implemented.

  From Armv8.2 this option is not permitted.

- ARMv8.2-PCSample is implemented, meaning the PC Sample-based Profiling extension is implemented in the Performance Monitors memory-mapped register space. PMDEVID.PCSample identifies the implemented level of profiling.

If PC Sample-based Profiling is implemented in the external debug register space:

- The following external debug registers can be implemented:
  — EDCIDSR.
  — EDPCSR.
  — EDVIDSR.

  See *External debug interface register map* on page H8-7061.
- If ARMv8.1-VHE is implemented, EDSCR.SC2 controls what PC Sample-based Profiling samples.

If ARMv8.2-PCSample is implemented, the following registers can be implemented in the Performance Monitors memory-mapped register space:

- PMCID1SR and PMCID2SR.
- PMPCSR.
- PMVIDSR.

See *Performance Monitors external register views* on page I5-7252.

If the PC Sample-based Profiling Extension is implemented with ARMv8.2-PCSample but the Performance Monitors Extension is not implemented, then the PC Sample-based Profiling Extension is implemented in its own memory-mapped register space, within the area that is reserved for the Performance Monitors, see Table H7-1. If CoreSight compliance is required:

- The management registers are defined as in Table K2-3 on page K2-7881.
- The support for PC Sample-based profiling is defined in the following registers:
  — PMDEVTYPE.MAJOR has the value `0x0`.
  — PMDEVARCH.ARCHID has the value `0x0A10`.

Table H7-1 PC Sample-based Profiling register map without the Performance Monitors Extension

Offset	Description
`0x200`	PMPCSR[31:0]
`0x204`	PMPCSR[63:32]
`0x208`	PMCID1SR
`0x20C`	PMVIDSR
`0x220`	PMPCSR[31:0] (alias)
`0x224`	PMPCSR[63:32] (alias)

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table H7-1 PC Sample-based Profiling register map without the Performance Monitors Extension

Offset	Description
`0x228`	PMCID1SR (alias)
`0x22C`	PMCID2SR
`0x600-0x6FC`	IMPLEMENTATION DEFINED
`0xE80-0xEFC`	IMPLEMENTATION DEFINED for CoreSight compliance
`0xFF0-0xFFc`	Management and CoreSight compliance registers

H7.1.3 Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN

The architecture permits IMPLEMENTATION DEFINED extensions to external debug to define mechanisms that make the values of the PC Sample-based profiling registers UNKNOWN. However, it requires that any such mechanism is disabled by default. This means that powerup or a hard reset of the PE must leave the PE in a state where the PC Sample-based Profiling Extension, if implemented, exhibits its architecturally-defined behavior.

Note

A mechanism that, when enabled, makes the PC Sample-based profiling registers UNKNOWN might use other sample-based profiling events that are appropriate for a use that is independent of PC Sample-based Profiling.

If no instruction has been retired since the PE left Debug state, Reset state, or a state where PC Sample-based profiling is prohibited, the sampled value is UNKNOWN. If an instruction has been retired but this is the first time the PMPCSR or EDPCSR is read since the PE left Reset state, the sampled value is permitted but not required to return the value `0xFFFFFFF`.

H7.1.4 Pseudocode description of PC Sample-based Profiling

When PC Sample-based Profiling is implemented but not with ARMv8.2-PCSample, the functionality is described by the pseudocode functions:

- `CreatePCSample()`, which populates a variable of type `PCSample`.
- `EDPCSRlo[]`, which writes a PC sample to the EDPCSR and associated registers.

When ARMv8.2-PCSample is implemented, the functionality is described by the pseudocode functions:

- `CreatePCSample()`, which populates a variable of type `PCSample`.
- `PMPCSR[]`, which writes a PC Sample to the PMPCSR and associated registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter H8
About the External Debug Registers

This chapter provides some additional information about the external debug registers. It contains the following sections:

- *Relationship between external debug and System registers* on page H8-7048.
- *Endianness and supported access sizes* on page H8-7049.
- *Synchronization of changes to the external debug registers* on page H8-7050.
- *Memory-mapped accesses to the external debug interface* on page H8-7054.
- *External debug interface register access permissions* on page H8-7057.
- *External debug interface registers* on page H8-7061.
- *Cross-trigger interface registers* on page H8-7068.
- *External debug register resets* on page H8-7070.

Note

Where necessary Table K14-1 on page K14-8038 disambiguates the general register references used in this chapter.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H8.1 Relationship between external debug and System registers

Table H8-1 shows the relationship between external debug registers and System registers. Where no relationship exists, the registers are not listed.

Table H8-1 Equivalence between external debug and System registers

	System register		
External debug register	**AArch64**	**AArch32**	**Notes**
DBGDTRRX_EL0	DBGDTRRX_EL0	DBGDTRRXint	See also *Summary of System register accesses to the DCC* on page H4-7002
DBGDTRTX_EL0	DBGDTRTX_EL0	DBGDTRTXint	
OSLAR_EL1	OSLAR_EL1	DBGOSLAR	-
DBGBVR<n>_EL1[31:0] DBGBVR<n>_EL1[63:32]	DBGBVR<n>_EL1[31:0] DBGBVR<n>_EL1[63:32]	DBGBVR<n> DBGBXVR<n>	-
DBGBCR<n>_EL1	DBGBCR<n>_EL1	DBGBCR<n>	-
DBGWVR<n>_EL1[31:0] DBGWVR<n>_EL1[63:32]	DBGWVR<n>_EL1[31:0] DBGWVR<n>_EL1[63:32]	DBGWVR<n>	-
DBGWCR<n>_EL1	DBGWCR<n>_EL1	DBGWCR<n>	-
DBGCLAIMSET_EL1	DBGCLAIMSET_EL1	DBGCLAIMSET	-
DBGCLAIMCLR_EL1	DBGCLAIMCLR_EL1	DBGCLAIMCLR	-
DBGAUTHSTATUS_EL1	DBGAUTHSTATUS_EL1	DBGAUTHSTATUS	Read-only
EDSCR	MDSCR_EL1	DBGDSCRext	Only some fields map
EDECCR	OSECCR_EL1	DBGOSECCR	Applies when the OS Lock is locked.
MIDR_EL1	MIDR_EL1	MIDR	Read-only copies of Processor ID Registers
EDDEVAFF0 EDDEVAFF1	MPIDR_EL1[31:0][a] MPIDR_EL1[63:32][a]	MPIDR	Read-only copies of system ID registers

a. This is a word of a 64-bit register.

In addition:

- EDSCR.{TXfull, RXfull} are read-only aliases for DCCSR.{TXfull, RXfull}.
- EDPRCR.CORENPDRQ is a read/write alias for DBGPRCR.CORENPDRQ.
- EDPRSR.OSLK is a read-only alias for OSLSR.OSLK.
- If the ARMv8.0-DoubleLock is implemented, EDPRSR.DLK is a read-only function of OSDLR.DLK.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H8.2 Endianness and supported access sizes

The debug registers, Performance Monitors registers, and CTI registers are implemented as memory-mapped peripherals. The Arm architecture requires memory-mapped peripherals to be little-endian.

The memory access sizes supported by any peripheral is IMPLEMENTATION DEFINED by the peripheral. For accesses to the debug registers, Performance Monitors registers, and CTI registers, implementations must:

- Comply with the requirements of *Supported access sizes* on page I1-7224.
- Support word-aligned 32-bit accesses to access 32-bit registers or either half of a 64-bit register mapped to a doubleword-aligned pair of adjacent 32-bit locations, even if no PE in the system implements AArch32.

Note

These requirements mean that a system implementing the debug registers using a 32-bit bus, such as a AMBA APB3, with a wider system interconnect must implement a bridge between the system and the debug bus that can split 64-bit accesses.

For accesses from the external debug interface, the size of an access is determined by the interface. For an access from an ADIv5-compliant Memory Access Port, MEM-AP, this is specified by the MEM-AP CSW register.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H8.3 Synchronization of changes to the external debug registers

This section describes the synchronization requirements for the external debug interface.

For more information on how these requirements affect debug, see:

- *Synchronization and debug exceptions* on page D2-2485 for exceptions taken from AArch64 state.
- *Synchronization and debug exceptions* on page G2-5876 for exceptions taken from AArch32 state.
- *Synchronization and Halting debug events* on page H3-6987.
- *Synchronization of DCC and ITR accesses* on page H4-7002.

This section refers to accesses from the external debug interface as external reads and external writes. It refers to accesses to System registers as direct reads, direct writes, indirect reads, and indirect writes.

Note

Synchronization requirements for AArch64 System registers on page D13-2869 and *Synchronization of changes to AArch32 System registers* on page G8-6100 define direct read, direct write, indirect read, and indirect write, and classifies external reads as indirect reads, and external writes as indirect writes.

For general information about synchronization, access completion, ordering, and observability, see *Synchronization of memory-mapped registers* on page I1-7226.

Writes to the same register are serialized, meaning they are observed in the same order by all observers, although some observers might not observe all of the writes. With the exception of DBGBCR<n>_EL1, DBGBVR<n>_EL1, DBGWCR<n>_EL1, and DBGWVR<n>_EL1, external writes to different registers are not necessarily observed in the same order by all observers as the order in which they complete.

Synchronization of DCC and ITR accesses on page H4-7002 describes the synchronization requirements for the DCC and ITR.

Changes to the IMPLEMENTATION DEFINED authentication interface are external writes to the authentication status registers by the master of the authentication interface. See *Synchronization and the authentication interface* on page H8-7051.

The external agent must be able to guarantee completion of a write. For example by:

- Marking the memory as Device-nGnRnE and executing a `DSB` barrier, if the system supports this property.
- Reading back the value written.
- Some guaranteed property of the connection between the PE and the external agent.

Note

For an external Debug Access Port, access completion is an IMPLEMENTATION DEFINED property. For a CoreSight system using APB-AP to access a debug APB, accesses complete in order.

However, the external agent cannot force synchronization of completed writes without halting the PE. Executing an `ISB` instruction, either in Debug state or in Non-debug state, and exiting from Debug state forces synchronization. If the PE is in Debug state, executing an `ISB` instruction is guaranteed to explicitly synchronize any external reads, external writes, and changes to the authentication interface that are ordered before the external write to EDITR.

For any given observer, external writes to the following register groups are guaranteed to be observable in the same order in which they complete:

- The breakpoint registers, DBGBCR<n>_EL1 and DBGBVR<n>_EL1.
- The watchpoint registers, DBGWCR<n>_EL1 and DBGWVR<n>_EL1.

This guarantee only applies to external writes to registers within one of these groups. There is no guarantee regarding the ordering of the observability of external writes within these groups with respect to external writes to registers, for example EDSCR, or between breakpoints and watchpoints, including watchpoints linked to context matching breakpoints.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

This means that a debugger can rely on the external writes to be observed in the same order in which they complete. It does not mean that a debugger can rely on the external writes being observed in finite time.

In a simple sequential execution an indirect write that occurs as a side-effect of an access happens atomically with the access, meaning no other accesses are allowed between the register access and its side-effect.

If two or more interfaces simultaneously access a register, the behavior must be as if the accesses occurred atomically and in any order. This is described in *Examples of the synchronization of changes to the external debug registers*.

Some registers have the property that for certain bits a write of 0 is ignored and a write of 1 has an effect. This means that simultaneous writes must be merged. Registers that have this property and support both external debug and System register access include DBGCLAIMSET_EL1, DBGCLAIMCLR_EL1, PMCR_EL0.{C,P}, PMOVSSET_EL0, PMOVSCLR_EL0, PMCNTENSET_EL0, PMCNTENCLR_EL0, PMINTENSET_EL1, PMINTENCLR_EL1, and PMSWINC_EL0. This last register is OPTIONAL and deprecated in the external debug interface.

H8.3.1 Synchronization and the authentication interface

Changes to the authentication interface are indirect writes to the state of the PE by the master of the authentication interface.

For an external debug interface read of any Authentication Status register, or an indirect read of the authentication interface made in determining the response to a subsequent external debug interface access, a change on the authentication interface must be observable following a subsequent explicit *Context synchronization event*, and:

- It is IMPLEMENTATION DEFINED whether a change is guaranteed to be observable in finite time.
- It is IMPLEMENTATION DEFINED whether a change is guaranteed to be observable following an entry to Debug state.

For a System register read of DBGAUTHSTATUS_EL1, a change on the authentication interface is guaranteed to be observable only after a *Context synchronization event*.

Note

- In some systems the authentication interface is fixed by configuration, or changed under the control of software. These systems can require explicit synchronization for any change to the authentication interface.
- In other systems, the authentication interface is controlled dynamically by an external agent. In these systems, it is desirable that changes to the authentication interface do not require explicit synchronization by software executing on the PE to be observable by subsequent external debug interface accesses, and are either observable in finite time or are synchronized by entry to Debug state. Otherwise there are scenarios where a debugger is not able to halt and debug the system.

H8.3.2 Examples of the synchronization of changes to the external debug registers

Example H8-1, Example H8-2 on page H8-7052, and Example H8-3 on page H8-7052 show the synchronization of changes to the external debug registers.

Example H8-1 Order of synchronization of Breakpoint and Watchpoint register writes

Initially DBGBVR<n>_EL1 is `0x8000` and DBGBCR<n>_EL1 is `0x0181`. This means that a breakpoint is enabled on the halfword T32 instruction at address `0x8000`.

A sequence of external writes occurs in the following order:

1. `0x0000` is written to DBGBCR<n>_EL1, disabling the breakpoint.
2. `0x9000` is written to DBGBVR<n>_EL1[31:0].

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

3. `0x0061` is written to DBGBCR<n>_EL1, enabling a breakpoint on the halfword at address `0x9002`.

The external writes must be observable to indirect reads in the same order as the external writes complete. This means that at no point is there a breakpoint enabled on either of the halfwords at address `0x8002` and `0x9000`.

Similarly a breakpoint or watchpoint must be disabled:

- If both halves of a 64-bit address have to be updated.
- If any of the DBGBCR<n>_EL1 or DBGWCR<n>_EL1 fields are modified at the same time as updating the address.

Example H8-2 Simultaneous accesses to DTR registers

Initially EDSCR.{TXfull, TXU, ERR} are 0. Then:

- `0x0DCCDA7A` is directly written to DBGDTRTX_EL0 by an `MSR` instruction.
- DBGDTRTX_EL0 is indirectly read by the external debug interface.

These accesses might happen at the same time and in any order.

If the direct write of `0x0DCCDA7A` to DBGDTRTX_EL0 is handled first, then:

- The external debug interface read of DBGDTRTX_EL0 clears EDSCR.TXfull to 0.
- EDSCR.{TXU, ERR} are unchanged.
- The external debug interface read returns `0x0DCCDA7A`.

If the indirect read of DBGDTRTX_EL0 by the external debug interface is handled first, then:

- The external debug interface read of DBGDTRTX_EL0 causes an underrun and as a result EDSCR.{TXU, ERR} are both set to 1.
- The external debug interface returns an UNKNOWN value.
- Writing `0x0DCCDA7A` to DBGDTRTX_EL0 sets DTRTX to `0x0DCCDA7A` and EDSCR.TXfull to 1.

Example H8-3 Simultaneous writes to CLAIM registers

Initially all CLAIM tag bits are 0. Then:

- `0x01` is written to DBGCLAIMSET_EL1 by a direct write, followed by an explicit *Context synchronization event*.
- `0x02` is written to DBGCLAIMSET_EL1 by an external write.

These events might happen at the same time and in either order.

After this:

- DBGCLAIMCLR_EL1 is read by a direct read.
- DBGCLAIMCLR_EL1 is read by an external read.

In this case, a direct read can return either `0x01` or `0x03`, and the external read can return either `0x02` or `0x03`.

The only permitted final result for the CLAIM tags is the value `0x03`, because this would be the result regardless of whether `0x01` or `0x02` is written first. This is because the external write is guaranteed to be observable to a direct read in finite time. See *Synchronization requirements for AArch64 System registers* on page D13-2869.

It is not possible for a direct read to return `0x01` and the external read to return `0x02`, because the writes to DBGCLAIMCLR_EL1 are serialized.

In the following scenario, there is only one permitted result. Both observers observe the value `0x03`, and then, at the same time, two writes occur:

- `0x04` is written to DBGCLAIMSET_EL1 by a direct write, followed by an explicit *Context synchronization event*.
- `0x01` is written to DBGCLAIMCLR_EL1 by an external write.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In this case only permitted final result for the CLAIM tags is the value `0x06`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H8.4 Memory-mapped accesses to the external debug interface

Support for memory-mapped access to the external debug interface is OPTIONAL. When memory-mapped access to the external debug interface is supported, the external debug interface is accessed as a little-endian memory-mapped peripheral.

If the external debug interface is CoreSight compliant, then an OPTIONAL Software Lock can be implemented for memory-mapped accesses to each component.

The Software Lock is OPTIONAL and deprecated. If ARMv8.4-Debug is implemented, the Software Lock is not implemented. If it is not implemented, the behavior is as if it is unlocked. The Software Locks are controlled by EDLSR and EDLAR, PMLSR and PMLAR, and CTILSR and CTILAR. See *Management registers and CoreSight compliance* on page K2-7881.

If ARMv8.3-DoPD is implemented, Software Lock is not implemented by the architecturally-defined debug components in the Core power domain.

With the exception of these registers and the effect of the Software Lock, the behavior of the memory-mapped accesses is the same as for other accesses to the external debug interface.

Note

The recommended memory-mapped accesses to the external debug interface are not compatible with the memory-mapped interface defined in Armv7. In particular:

- The memory map is different.
- Memory-mapped accesses do not behave differently to Debug Access Port accesses when OSLSR.OSLK == 1, meaning that the OS Lock is locked.

The following sections give more information about these memory-mapped accesses:

- *Register access permissions for memory-mapped accesses*.
- *Synchronization of memory-mapped accesses to external debug registers* on page H8-7056.

See also *Supported access sizes* on page I1-7224.

H8.4.1 Register access permissions for memory-mapped accesses

It is IMPLEMENTATION DEFINED whether unprivileged memory-mapped accesses are allowed. Privileged software is responsible for controlling memory-mapped accesses using the MMU.

If ARMv8.4-Debug is implemented, the Secure view of a debug component is mapped into Secure physical memory and the Non-secure view is mapped into Non-secure physical memory.

If ARMv8.4-Debug is implemented, the access permissions are different in each Security state, but Secure and Non-secure views of the debug components are identical. Arm recommends the views are located at the same address in the Secure and Non-secure physical address maps.

If memory-mapped accesses are made through an ADIv5 interface, the Debug Access Port can block the access using **DBGSWENABLE**. This is outside the scope of the Armv8-A architecture. See *Arm® Debug Interface Architecture Specification ADIv5.0 to ADIv5.2*.

Effect of the OPTIONAL Software Lock on memory-mapped access

For memory-mapped accesses, if other controls permit access to a register, the OPTIONAL Software Lock is implemented, and EDLSR.SLK, PMLSR.SLK, or CTILSR.SLK is set to 1, meaning the Software Lock is locked, then with the exception of the LAR itself:

- If other controls permit access to a register, then writes are ignored. That is:
  - Read/write (RW) registers become read-only, writes ignored (RO/WI).
  - Write-only (WO) registers become writes ignored (WI).

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Reads and writes have no side-effects. A side-effect is where a direct read or a direct write of a register creates an indirect write of the same or another register. When the Software Lock is locked, the indirect write does not occur.
- Writes to EDLAR, PMLAR, and CTILAR are unaffected.

This behavior must also apply to all IMPLEMENTATION DEFINED registers.

For example, if EDLSR.SLK is set to 1:

- EDSCR.{TXfull, TXU, ERR} are unchanged by a memory-mapped read from DBGDTRTX_EL0.
- EDSCR.{RXfull, RXO, ERR} are unchanged by a memory-mapped write to DBGDTRRX_EL0 that is ignored.
- EDSCR.{ITE, ITO, ERR} are unchanged by a memory-mapped write to EDITR that is ignored.
- OSLSR.OSLK is unchanged by a memory-mapped write to OSLAR_EL1 that is ignored.
- EDPCSR[63:32], EDCIDSR, and EDVIDSR are unchanged by a memory-mapped read from EDPCSR[31:0].

  Note

  Updating EDVIDSR, EDCIDSR, and EDPCSRhi are side-effects of reading EDPCSRlo, such that these registers contain the matching context for EDPCSRlo. The process that updates EDPCSRlo with PC samples is not a side-effect of the access. Reads of EDPCSRlo made when the Software Lock is locked can be used to profile software.

- PMPCSR[63:32], PMCID1SR/PMCID2SR, and PMVIDSR are unchanged by a memory-mapped read from PMPCSR[31:0].

  Note

  Updating PMVIDSR, PMCID1SR/PMCID2SR, and PMPCSR[31:0] are side-effects of reading PMPCSR[63:32], such that these registers contain the matching context for PMPCSR[63:32]. The process that updates PMPCSR[63:32] with PC samples is not a side-effect of the access. Reads of PMPCSR[63:32] made when the Software Lock is locked can be used to profile software.

- EDPRSR.{SDR, SPMAD, SDAD, SR, SPD} are unchanged by a memory-mapped read from EDPRSR.
- EDPRSR.SDAD is not set if an error response is returned due to a memory-mapped read or write of any debug register as the result of the value of the EDAD field.
- The CLAIM tags are unchanged by memory-mapped writes to DBGCLAIMSET_EL1 and DBGCLAIMCLR_EL1 which are ignored.

Similarly, if PMLSR.SLK is set to 1, then EDPRSR.SPMAD is not set if an error response is returned to a memory-mapped read or write of any Performance Monitors register due to the value of the EPMAD field.

Behavior of a not permitted memory-mapped access

Where the architecture requires that an external debug interface access generates an error response, a memory-mapped access must also generate an error response. However, it is IMPLEMENTATION DEFINED how the error response is handled, as this depends on the system.

Arm recommends that the error is returned as either:

- A synchronous external Data Abort.
- An SError interrupt.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H8.4.2 Synchronization of memory-mapped accesses to external debug registers

The synchronization requirements for memory-mapped accesses to the external debug interface is described in *Synchronization of changes to the external debug registers* on page H8-7050 and *Synchronization of memory-mapped registers* on page I1-7226.

The synchronization requirements between different routes to the external debug interface, that is, between Debug Access Port accesses and memory-mapped accesses are IMPLEMENTATION DEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H8.5 External debug interface register access permissions

Some external accesses to debug registers and Performance Monitor registers are not permitted and return an error response if:

- The Core power domain is powered down or is in low-power state where the registers cannot be accessed.
- OSLSR.OSLK == 1. The OS Lock is locked.
- ARMv8.0-DoubleLock is implemented and `DoubleLockStatus()` == TRUE. The OS Double Lock is locked.
- The access is disabled by either the authentication interface or secure monitor.

Not all registers are affected in all of these cases. For details, see *External debug interface register access permissions summary* on page H8-7059.

H8.5.1 External debug over powerdown and locks

Accessing registers using the external debug interface is not possible when the Debug power domain is off. In this case all accesses return an error.

External accesses to debug and Performance Monitors registers in the Core power domain are not permitted and return an error response if:

- The Core power domain is off or in low-power state where the registers cannot be accessed.
- OSLSR.OSLK == 1, meaning that the OS Lock is locked. This allows software to prevent external debugger modification of the registers while it saves and restores them over powerdown.
- ARMv8.0-DoubleLock is implemented and `DoubleLockStatus()` == TRUE. This means that the OS Double Lock is locked. The OS Double Lock ensures that it is safe to remove Core power by forcing the debug interface to be quiescent.

If ARMv8.0-DoubleLock is not implemented, the hardware must provide another method to safely remove Core power.

The OS Lock condition does not apply to the following debug registers:

- OSLAR_EL1. This means that an external debugger can override this lock.
- EDESR. This means that an external debugger can program a debug event for when software unlocks the OS Lock. See *OS Unlock Catch debug event* on page H3-6984.
- The ID registers that describe the PE to the debugger.

See also *Debug registers to save over powerdown* on page H6-7037.

H8.5.2 External access disabled

Accesses are further controlled by the external authentication interface. An untrusted external debugger cannot program the breakpoint and watchpoint registers to generate spurious debug exceptions. If external invasive debugging is not enabled, these external accesses to the registers are disabled. If EL3 is implemented, then SDCR provides additional external access controls for those registers.

The disable applies to:

- The DBGBVR<n>_EL1, DBGBCR<n>_EL1, DBGWVR<n>_EL1, and DBGWCR<n>_EL1 registers.
- From Armv8.2, the OSLAR_EL1 register.

If ARMv8.2-Debug is not implemented, it is IMPLEMENTATION DEFINED whether the disable applies to OSLAR_EL1.

If ARMv8.4-Debug is not implemented, the external debug interface cannot access these registers if any of the following are true:

- `ExternalInvasiveDebugEnabled()` == FALSE.
- `ExternalSecureInvasiveDebugEnabled()` == FALSE, EL3 is not implemented, and the PE behaves as if the Security state is Secure.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- `ExternalSecureInvasiveDebugEnabled()` == FALSE, EL3 is implemented and SDCR.EDAD == 1.

If ARMv8.4-Debug is implemented, Non-secure accesses from the external debug interface to these registers are not permitted if any of the following are true:

- EL3 is not implemented and the PE behaves as if the Security state is Secure.
- EL3 is implemented and SDCR.EDAD == 1.

The `AllowExternalDebugAccess()` pseudocode function describes these accessibility rules.

PEs might also provide an OPTIONAL external debug interface to the Performance Monitor registers. The authentication interface and SDCR provide similar external access disable controls for those registers.

If ARMv8.4-Debug is not implemented, the external debug interface cannot access the Performance Monitor registers if any of the following are true:

- `ExternalNoninvasiveDebugEnabled()` == FALSE.
- `ExternalSecureNoninvasiveDebugEnabled()` == FALSE, EL3 is not implemented and the PE behaves as if the Security state is Secure.
- `ExternalSecureNoninvasiveDebugEnabled()` == FALSE, EL3 is implemented and SDCR.EPMAD == 1.

Note

- Arm recommends that secure software that is not making use of debug hardware does not lock out the external debug interface.
- Armv8-A does not provide the equivalent control over access to Trace extension registers, which means if ARMv8.4-Debug is implemented, the Non-secure and Secure views are identical.

If ARMv8.4-Debug is implemented, Non-secure accesses from the external debug interface to these registers are not permitted if any of the following are true:

- EL3 is not implemented and the PE behaves as if the Security state is Secure.
- EL3 is implemented and SDCR.EPMAD == 1.

The `AllowExternalPMUAccess()` pseudocode function describes these accessibility rules.

H8.5.3 Behavior of a not permitted access

For an external debug interface access by a Debug Access Port, the Debug Access Port receives the error response and must signal this to the external debugger. For an ADIv5 implementation of a Debug Access Port, the error sets a sticky error flag in the Debug Access Port that the debugger can poll, and that suppresses further accesses until it is explicitly cleared.

When an error is returned because external access is disabled, and this is the highest priority error condition, a sticky error flag in EDPRSR is indirectly written to 1 as a side-effect of the access:

- For a debug register access when `AllowExternalDebugAccess`() == FALSE, EDPRSR.SDAD is indirectly written to 1.
- For Performance Monitor register access when `AllowExternalPMUAccess`s() == FALSE, EDPRSR.SPMAD is indirectly written to 1.

The indirect write might not occur for a memory-mapped access to the external debug interface. For more information, see *Register access permissions for memory-mapped accesses* on page H8-7054.

If no error is returned, or the error is returned because of a higher priority error condition, the flag in EDPRSR is unchanged.

See also *Behavior of a not permitted memory-mapped access* on page H8-7055.

For more information, see *Arm® Debug Interface Architecture Specification*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H8.5.4 External debug interface register access permissions summary

For accesses to:

- IMPLEMENTATION DEFINED registers, see *IMPLEMENTATION DEFINED registers*.
- OPTIONAL registers for CoreSight compliance, see *Management registers and CoreSight compliance* on page K2-7881.
- Reserved, unallocated, or unimplemented registers, writes to read-only registers, and reads of write-only registers, see *Reserved and unallocated registers*.

For all other external debug interface, CTI, and Performance Monitor registers, Table H8-3 on page H8-7064, Table H8-4 on page H8-7065, Table H8-6 on page H8-7069 and Table I3-1 on page I3-7244, show the response of the PE to accesses by the external debug interface.

H8.5.5 IMPLEMENTATION DEFINED registers

For debug registers, Performance Monitors registers, CTI registers, register access permissions for IMPLEMENTATION DEFINED registers are IMPLEMENTATION DEFINED.

If OPTIONAL memory-mapped access to the external debug interface is supported, there are additional constraints on memory-mapped accesses to registers. These constraints must also apply to IMPLEMENTATION DEFINED registers. For more information see *Register access permissions for memory-mapped accesses* on page H8-7054.

If ARMv8.3-DoPD is not implemented, the power domain of these registers is in which these registers are implemented is also IMPLEMENTATION DEFINED. The registers must apply the constraint that if the OPTIONAL Software Lock is locked, writes are ignored and accesses have no side-effects.

If ARMv8.3-DoPD is implemented, then:

- For debug registers and Performance Monitors registers, IMPLEMENTATION DEFINED registers are implemented in the Core power domain. Accesses return an error when the Core power domain is off or in a low-power state.
- For CTI registers, IMPLEMENTATION DEFINED registers are implemented in the Debug power domain.

H8.5.6 Reserved and unallocated registers

The default access requirements for reserved and unallocated registers are described in *Access requirements for reserved and unallocated registers* on page I1-7228.

Note

Reads of WO and writes to RO refers to the default access permissions for a register. For example, when the SLK field is set, meaning that the relevant registers become RO, a memory-mapped write to a RW register is ignored, and not treated as a reserved access.

The following reserved registers are RES0 in all conditions, other than when debug power is off:

- All reserved CTI registers.
- For the debug registers, and Performance Monitors registers, if the implementation is CoreSight architecture compliant, and either ARMv8.3-DoPD is not implemented or the Core power domain is on, all reserved registers in the range `0xF00 - 0xFFC`. See *Management register access permissions* on page K2-7882.

Otherwise, the architecture defines that:

1. If debug power is off, all register accesses, including reserved accesses, return an error.
2. For reserved debug registers and Performance Monitors registers, if ARMv8.3-DoPD is implemented, and the Core power domain is off or in a low-power state, the response is an error. Otherwise the response is a CONSTRAINED UNPREDICTABLE choice of error or RES0, when any of the following hold:

   Off The Core power domain is either completely off or in a low-power state in which the Core power domain registers cannot be accessed.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

DLK ARMv8.0-DoubleLock is implemented and `DoubleLockStatus()` == TRUE. The OS Double Lock is locked.

OSLK OSLSR.OSLK == 1. The OS Lock is locked.

3. In addition, for reserved debug registers in the address ranges `0x400 - 0x4FC` and `0x800 - 0x8FC`, the response is a CONSTRAINED UNPREDICTABLE choice of error or RES0 when conditions 1 or 2 do not apply and:

   EDAD `AllowExternalDebugAccess()` == FALSE. External debug is disabled.

   Note

   See also *Behavior of a not permitted access* on page H8-7058.

4. In addition, for reserved Performance Monitors registers in the address ranges `0x000 - 0xEFC`, the response is a CONSTRAINED UNPREDICTABLE choice of error or RES0 when conditions 1 or 2 do not apply and:

   EPMAD `AllowExternalPMUAccess()` == FALSE. External Performance Monitor access is disabled.

   Note

   See also *Behavior of a not permitted access* on page H8-7058.

5. For reads of WO locations, the response is a CONSTRAINED UNPREDICTABLE choice of error or RES0 when the architecture permits or requires a write to the location to return an error.

6. For writes of RO locations, the response is a CONSTRAINED UNPREDICTABLE choice of error or RES0 when the architecture permits or requires a read to the location to return an error.

7. For reads and writes of locations for features that are not implemented, the response is a CONSTRAINED UNPREDICTABLE choice of error or RES0 when the architecture permits or requires an access to the location to return an error if the feature is implemented.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H8.6 External debug interface registers

The external debug interface register map is described by:

- *Performance Monitors external register views* on page I5-7252.
- *Cross-trigger interface registers* on page H8-7068.
- *External debug interface register map*.

Table H8-2 External debug interface register map

Offset	Mnemonic	Register, or additional information
0x020	EDESR	*EDESR, External Debug Event Status Register* on page H9-7127
0x024	EDECR	*EDECR, External Debug Execution Control Register* on page H9-7125
0x030 0x034	EDWAR[31:0] EDWAR[63:32]	*EDWAR, External Debug Watchpoint Address Register* on page H9-7171
0x080	DBGDTRRX_EL0	Chapter H4 *The Debug Communication Channel and Instruction Transfer Register*
0x084	EDITR	*EDITR, External Debug Instruction Transfer Register* on page H9-7130
0x088	EDSCR	*EDSCR, External Debug Status and Control Register* on page H9-7162
0x08C	DBGDTRTX_EL0	Chapter H4 *The Debug Communication Channel and Instruction Transfer Register*
0x090	EDRCR	*EDRCR, External Debug Reserve Control Register* on page H9-7160
0x094	EDACR	*EDACR, External Debug Auxiliary Control Register* on page H9-7102
0x098	EDECCR	*EDECCR, External Debug Exception Catch Control Register* on page H9-7121
0x0A0	EDPCSRlo[a]	*EDPCSR, External Debug Program Counter Sample Register* on page H9-7136
0x0A4	EDCIDSR	*EDCIDSR, External Debug Context ID Sample Register* on page H9-7107
0x0A8	EDVIDSR	*EDVIDSR, External Debug Virtual Context Sample Register* on page H9-7168
0x0AC	EDPCSRhi	*EDPCSR, External Debug Program Counter Sample Register* on page H9-7136
0x0300	OSLAR_EL1	*OSLAR_EL1, OS Lock Access Register* on page H9-7175
0x0310	EDPRCR	*EDPRCR, External Debug Power/Reset Control Register* on page H9-7148
0x0314	EDPRSR	*EDPRSR, External Debug Processor Status Register* on page H9-7152
0x0400+16×n 0x0404+16×n	DBGBVR<n>_EL1[31:0][bc] DBGBVR<n>_EL1[63:32][bc]	*DBGBVR<n>_EL1, Debug Breakpoint Value Registers, n = 0 - 15* on page H9-7082
0x0408+16×n	DBGBCR<n>_EL1	*DBGBCR<n>_EL1, Debug Breakpoint Control Registers, n = 0 - 15* on page H9-7078
0x800+16 0x804+16×n	DBGWVR<n>_EL1[31:0][bc] DBGWVR<n>_EL1[63:32][bc]	*DBGWVR<n>_EL1, Debug Watchpoint Value Registers, n = 0 - 15* on page H9-7098
0x808+16×n	DBGWCR<n>_EL1[c]	*DBGWCR<n>_EL1, Debug Watchpoint Control Registers, n = 0 - 15* on page H9-7095
0xC00-0xCFC	IMPLEMENTATION DEFINED	-
0xD00	MIDR_EL1	Main ID register
0xD04-0xD1C	-	Reserved, RES0

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table H8-2 External debug interface register map (continued)

Offset	Mnemonic	Register, or additional information
0xD20	EDPFR[31:0]	External Debug Processor Feature Register 0
0xD24	EDPFR[63:32]	
0xD28	EDDFR[31:0]	External Debug Feature Register 0
0xD2C	EDDFR[63:32]	
0xD30	Reserved, see next column	Previously defined as Instruction Set Attribute Register 0 bits[31:0]. Behavior is: **Bits[31:20]** RES0. **Bits[19:4]** UNKNOWN. **Bits[3:0]** RES0.
0xD34	RES0	Previously defined as Instruction Set Attribute Register 0 bits[63:32]
0xD38	UNKNOWN	Previously defined as Memory Model Feature Register 0
0xD3C	RES0	
0xD40-0xDFC	RES0	Reserved, RES0
0xD60	EDAA32PFR[31:0]	External Debug AArch32 Processor Feature Register
0xD64	EDAA32PFR[63:32]	External Debug AArch32 Processor Feature Register
0xE80-EFC	IMPLEMENTATION DEFINED	-
0xF00-E8C	Management registers	*Management registers and CoreSight compliance* on page K2-7881
0xFA0	DBGCLAIMSET_EL1	*DBGCLAIMSET_EL1, Debug CLAIM Tag Set register* on page H9-7089
0xFA4	DBGCLAIMCLR_EL1	*DBGCLAIMCLR_EL1, Debug CLAIM Tag Clear register* on page H9-7087
0xFA8	EDDEVAFF0	*EDDEVAFF0, External Debug Device Affinity register 0* on page H9-7109
0xFAC	EDDEVAFF1	*EDDEVAFF1, External Debug Device Affinity register 1* on page H9-7110
0xFB0-FB4	Management registers	*Management registers and CoreSight compliance* on page K2-7881
0xFB8	DBGAUTHSTATUS_EL1	*DBGAUTHSTATUS_EL1, Debug Authentication Status register* on page H9-7076
0xFC0	EDDEVID2	*EDDEVID, External Debug Device ID register 0* on page H9-7113
0xFC4	EDDEVID1	*EDDEVID1, External Debug Device ID register 1* on page H9-7115
0xFC8	EDDEVID	*EDDEVID2, External Debug Device ID register 2* on page H9-7116
0xFD0-FFC	Management registers	*Management registers and CoreSight compliance* on page K2-7881

a. Supported only if the OPTIONAL PC Sample-Based Profiling is implemented but ARMv8.2-PCSample is not implemented. See Chapter H7 *The PC Sample-based Profiling Extension*.

Implemented only if the OPTIONAL PC Sample-based Profiling Extension is implemented.

b. A 64-bit register mapped to a pair of 32-bit locations. Doubleword accesses to this register are not guaranteed to be 64-bit single copy atomic. See *Endianness and supported access sizes* on page H8-7049. Software must ensure a breakpoint or watchpoint is disabled before altering the value register.

c. Implemented breakpoints and watchpoints only. *n* is the breakpoint or the watchpoint number.

Note

All other locations are reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H8.6.1 Access permissions for the External debug interface registers

Table H8-3 on page H8-7064 and Table H8-4 on page H8-7065 show the access permissions for the external debug interface registers in an Armv8-A Debug implementation. The terms are defined as follows:

Domain This describes the power domain in which the register is logically implemented. Registers described as implemented in the Core power domain might be implemented in the Debug power domain, as long as they exhibit the required behavior.

If ARMv8.3-DoPD is implemented, most External debug interface registers are in the Core power domain, as shown in Table H8-3 on page H8-7064.

If ARMv8.3-DoPD is not implemented, most of the registers are in the Debug Power Domain, as shown in Table H8-4 on page H8-7065.

Conditions This lists the conditions under which the access is attempted.

To determine the access permissions for a register, read these columns from left to right, and stop at first column that lists the condition as being true.

The conditions are:

Off The Core power domain is completely off, or in low-power state. In these cases the Core power domain registers cannot be accessed, and if ARMv8.3-DoPD is not implemented, EDPRSR.PU will read as 0.

Note

When the Debug power domain is off, all accesses to the registers in the external Debug power domain return an error.

DLK ARMv8.0-DoubleLock is implemented and `DoubleLockStatus()` == TRUE. The OS Double Lock is locked. If ARMv8.3-DoPD is implemented, ARMv8.0-DoubleLock is not implemented and so Table H8-3 on page H8-7064 does not include this column.

OSLK OSLSR.OSLK == 1. The OS Lock is locked.

EDAD `AllowExternalDebugAccess()` == FALSE. External debug access is disabled for the access. If ARMv8.4-Debug is implemented, this applies only for Non-secure accesses to the register. See also *Behavior of a not permitted access* on page H8-7058.

EPMAD `AllowExternalPMUAccess()` == FALSE. Access to the external Performance Monitors is disabled for the access. If ARMv8.4-Debug is implemented, this applies only for Non-secure accesses to the register. See also *Behavior of a not permitted access* on page H8-7058.

SLK The Software Lock is implemented and `SoftwareLockStatus()` == TRUE. This provides the modified default access permissions for OPTIONAL memory-mapped accesses to the external debug interface if the OPTIONAL Software Lock is locked. See *Register access permissions for memory-mapped accesses* on page H8-7054. If ARMv8.3-DoPD is implemented, the Software Lock is not locked or not implemented, this column is ignored.

Default This provides the default access permissions, if there are no conditions that prevent access to the register.

The access permissions are:

- This means that the default access permission applies. See the Default column, or the SLK column, if applicable.

RO This means that the register or field is read-only, and:

- Unless the register description states otherwise, a RO field in an RW register ignores writes.
- Where the **SLK** control makes a RW register RO, the register ignores writes.

RW This means that the register or field is read/write. Individual fields within the register might be RO or WO. See the relevant register description for details.

RC This means that a read of the register bit clears the field to 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

WO This means that the register or field is write-only. Unless the register description states otherwise, a WO field in a RW register returns an UNKNOWN value on a read of the register.

WI This means that the register or field ignores writes.

IMP DEF This means that the access permissions are IMPLEMENTATION DEFINED.

If OPTIONAL memory-mapped access to the external debug interface is supported, there might be additional constraints on memory-mapped accesses. See *Register access permissions for memory-mapped accesses* on page H8-7054.

For the reset values for the external debug interface registers, see Table H8-7 on page H8-7071.

Table H8-3 Access permissions for the external debug interface registers if ARMv8.3-DoPD is implemented

Offset	Register	Domain	Conditions (priority from left to right)				
			Off	DLK	OSLK	EDAD	Default
0x020	EDESR	Core	Error	-	-	-	RW
0x024	EDECR	Core	Error	-	-	-	RW
0x030 0x034	EDWAR[31:0] EDWAR[63:32]	Core	Error	-	Error	-	RO
0x080	DBGDTRRX_EL0	Core	Error	-	Error	-	RW
0x084	EDITR	Core	Error	-	Error	-	WO
0x088	EDSCR	Core	Error	-	Error	-	RW
0x08C	DBGDTRTX_EL0	Core	Error	-	Error	-	RW
0x090	EDRCR	Core	Error	-	Error	-	WO
0x094	EDACR	Core	Error	-	IMP DEF	-	RW
0x098	EDECCR	Core	Error	-	Error	-	RW
0x0A0	EDPCSR[31:0][a]	Core	Error	-	Error	-	RO
0x0A4	EDCIDSR[a]	Core	Error	-	Error	-	RO
0x0A8	EDVIDSR[a]	Core	Error	-	Error	-	RO
0x0AC	EDPCSR[63:32][a]	Core	Error	-	Error	-	RO
0x0300	OSLAR_EL1	Core	Error	-	-	Error	WO
0x0310	EDPRCR	Core	Error	-	Error	-	RW
0x0314	EDPRSR	Core	Error	-	Error	-	RO
0x0400+16×n 0x0404+16×n	DBGBVR<n>_EL1[31:0][b] DBGBVR<n>_EL1[63:32][b]	Core Core	Error Error	-	Error Error	Error Error	RW RW
0x0408+16×n	DBGBCR<n>_EL1[b]	Core	Error	-	Error	Error	RW
0x800+16×n 0x804+16×n	DBGWVR<n>_EL1[31:0][b] DBGWVR<n>_EL1[63:32][b]	Core Core	Error Error	-	Error Error	Error Error	RW RW
0x808+16×n	DBGWCR<n>_EL1[b]	Core	Error	-	Error	Error	RW

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table H8-3 Access permissions for the external debug interface registers if ARMv8.3-DoPD is implemented

Offset	Register	Domain	Conditions (priority from left to right)				
			Off	DLK	OSLK	EDAD	Default
0xD00	MIDR_EL1	Core	Error	-	-	-	RO
0xD20	EDPFR[31:0]	Core	Error	-	-	-	RO
0xD24	EDPFR[63:32]	Core	Error	-	-	-	RO
0xD28	EDDFR[31:0]	Core	Error	-	-	-	RO
0xD2C	EDDFR[63:32]	Core	Error	-	-	-	RO
0xD60	EDAA32PFR[31:0]	Core	Error	-	-	-	RO
0xD64	EDAA32PFR[63:32]	Core	Error	-	-	-	RO
0xFA0	DBGCLAIMSET_EL1	Core	Error	-	Error	-	RW
0xFA4	DBGCLAIMCLR_EL1	Core	Error	-	Error	-	RW
0xFA8	EDDEVAFF0	Core	Error	-	-	-	RO
0xFAC	EDDEVAFF1	Core	Error	-	-	-	RO
0xFB8	DBGAUTHSTATUS_EL1	Core	Error	-	-	-	RO
0xFC0	EDDEVID2	Core	Error	-	-	-	RO
0xFC4	EDDEVID1	Core	Error	-	-	-	RO
0xFC8	EDDEVID	Core	Error	-	-	-	RO

a. Implemented only if the PC Sample-based profiling Extension is implemented and ARMv8.2-PCSample is not implemented.

b. Implemented breakpoints and watchpoints only. *n* is the breakpoint or watchpoint number.

Table H8-4 Access permissions for the external debug interface registers if ARMv8.3-DoPD is not implemented

Offset	Register	Domain	Conditions (priority from left to right)					
			Off	DLK	OSLK	EDAD	Default	SLK
0x020	EDESR	Core	Error	Error	-	-	RW	RO
0x024	EDECR	Debug	-	-	-	-	RW	RO
0x030 0x034	EDWAR[31:0] EDWAR[63:32]	Core	Error	Error	Error	-	RO	-
0x080	DBGDTRRX_EL0	Core	Error	Error	Error	-	RW	RO
0x084	EDITR	Core	Error	Error	Error	-	WO	WI
0x088	EDSCR	Core	Error	Error	Error	-	RW	RO
0x08C	DBGDTRTX_EL0	Core	Error	Error	Error	-	RW	RO
0x090	EDRCR	Core	Error	Error	Error	-	WO	WI

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table H8-4 Access permissions for the external debug interface registers if ARMv8.3-DoPD is not implemented

			Conditions (priority from left to right)					
Offset	**Register**	**Domain**	**Off**	**DLK**	**OSLK**	**EDAD**	**Default**	**SLK**
0x094	EDACR	IMP DEF	IMP DEF	IMP DEF	IMP DEF	-	RW	RO
0x098	EDECCR	Core	Error	Error	Error	-	RW	RO
0x0A0	EDPCSR[31:0][a]	Core	Error	Error	Error	-	RO	RO
0x0A4	EDCIDSR[a]	Core	Error	Error	Error	-	RO	RO
0x0A8	EDVIDSR[a]	Core	Error	Error	Error	-	RO	RO
0x0AC	EDPCSR[63:32]	Core	Error	Error	Error	-	RO	RO
0x0300	OSLAR_EL1	Core	Error	Error	-	IMP DEF[b]	WO	WI
0x0310	EDPRCR	Core and Debug [c]	Error	Error	Error	-	RW	RO
0x0314	EDPRSR	Core and Debug [c]	Error	Error	Error	-	RO	RO
0x0400+16×n	DBGBVR<n>_EL1[31:0][d]	Core	Error	Error	Error	Error	RW	RO
0x0404+16×n	DBGBVR<n>_EL1[63:32][d]	Core	Error	Error	Error	Error	RW	RO
0x0408+16×n	DBGBCR<n>_EL1[d]	Core	Error	Error	Error	Error	RW	RO
0x800+16×n	DBGWVR<n>_EL1[31:0][d]	Core	Error	Error	Error	Error	RW	RO
0x804+16×n	DBGWVR<n>_EL1[63:32][d]	Core	Error	Error	Error	Error	RW	RO
0x808+16×n	DBGWCR<n>_EL1[d]	Core	Error	Error	Error	Error	RW	RO
0xD00	MIDR_EL1	IMP DEF	IMP DEF[e]	IMP DEF[e]	-	-	RO	RO
0xD20	EDPFR[31:0]	IMP DEF	IMP DEF[e]	IMP DEF[e]	-	-	RO	RO
0xD24	EDPFR[63:32]	IMP DEF	IMP DEF[e]	IMP DEF[e]	-	-	RO	RO
0xD28	EDDFR[31:0]	IMP DEF	IMP DEF[e]	IMP DEF[e]	-	-	RO	RO
0xD2C	EDDFR[63:32]	IMP DEF	IMP DEF[e]	IMP DEF[e]	-	-	RO	RO
0xD60	EDAA32PFR[31:0]	IMP DEF	IMP DEF[e]	IMP DEF[e]	-	-	RO	RO
0xD64	EDAA32PFR[63:32]	IMP DEF	IMP DEF[e]	IMP DEF[e]	-	-	RO	RO
0xFA0	DBGCLAIMSET_EL1	Core	Error	Error	Error	-	RW	RO
0xFA4	DBGCLAIMCLR_EL1	Core	Error	Error	Error	-	RW	RO
0xFA8	EDDEVAFF0	Debug	Error	-	-	-	RO	RO
0xFAC	EDDEVAFF1	Debug	Error	-	-	-	RO	RO
0xFB8	DBGAUTHSTATUS_EL1	Debug	Error	-	-	-	RO	RO
0xFC0	EDDEVID2	Debug	Error	-	-	-	RO	RO
0xFC4	EDDEVID1	Debug	Error	-	-	-	RO	RO
0xFC8	EDDEVID	Debug	Error	-	-	-	RO	RO

a. Implemented only if the PC Sample-based profiling Extension is implemented.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

b. If ARMv8.2-Debug is not implemented, it is IMPLEMENTATION DEFINED whether an error is returned. See *External access disabled* on page H8-7057. If no error is returned, the access is permitted.
c. Some bits are in the Debug power domain and some bits are in the Core power domain. See register field descriptions for information.
d. Implemented breakpoints and watchpoints only. *n* is the breakpoint or watchpoint number.
e. It is IMPLEMENTATION DEFINED whether an error is returned. See *External debug over powerdown and locks* on page H8-7057. If no error is returned, the access is permitted.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H8.7 Cross-trigger interface registers

The embedded Cross-trigger Interface, CTI, is located within its own block of the external debug memory map. There must be one such block for each PE.

If the CTI of a PE does not implement the CTIDEVAFF0 or CTIDEVAFF1 registers it must be located 64KB above the debug registers in the external debug interface.

When ARMv8.4-Debug is implemented, each debug component has a Secure and Non-secure view. The Secure view of a debug component is mapped into Secure physical memory and the Non-secure view of a debug component is mapped into Non-secure memory. Apart from access conditions, the Non-secure and Secure views of the debug components are identical.

Table H8-5 shows the CTI register map.

Table H8-5 Cross-trigger interface map

Offset	Mnemonic	Location of further details
0x000	CTICONTROL	*CTICONTROL, CTI Control register* on page H9-7191
0x010	CTIINTACK	*CTIINTACK, CTI Output Trigger Acknowledge register* on page H9-7205
0x014	CTIAPPSET	*CTIAPPSET, CTI Application Trigger Set register* on page H9-7181
0x018	CTIAPPCLEAR	*CTIAPPCLEAR, CTI Application Trigger Clear register* on page H9-7179
0x01C	CTIAPPPULSE	*CTIAPPPULSE, CTI Application Pulse register* on page H9-7180
0x020+4×n	CTIINEN<n>[a]	*CTIINEN<n>, CTI Input Trigger to Output Channel Enable registers, n = 0 - 31* on page H9-7204
0x0A0+4×n	CTIOUTEN<n>[a]	*CTIOUTEN<n>, CTI Input Channel to Output Trigger Enable registers, n = 0 - 31* on page H9-7213
0x130	CTITRIGINSTATUS	*CTITRIGINSTATUS, CTI Trigger In Status register* on page H9-7219
0x134	CTITRIGOUTSTATUS	*CTITRIGOUTSTATUS, CTI Trigger Out Status register* on page H9-7220
0x138	CTICHINSTATUS	*CTICHINSTATUS, CTI Channel In Status register* on page H9-7183
0x13C	CTICHOUTSTATUS	*CTICHOUTSTATUS, CTI Channel Out Status register* on page H9-7184
0x140	CTIGATE	*CTIGATE, CTI Channel Gate Enable register* on page H9-7203
0x144	ASICCTL	*ASICCTL, CTI External Multiplexer Control register* on page H9-7178
0xE80 - 0xEFC	IMPLEMENTATION DEFINED	IMPLEMENTATION DEFINED. See *Management registers and CoreSight compliance* on page K2-7881
0xF00 - 0xFBC	Management registers	*Management registers and CoreSight compliance* on page K2-7881
0xFC0	CTIDEVID2	*CTIDEVID2, CTI Device ID register 2* on page H9-7201
0xFC4	CTIDEVID1	*CTIDEVID1, CTI Device ID register 1* on page H9-7200
0xFC8	CTIDEVID	*CTIDEVID, CTI Device ID register 0* on page H9-7198
0xFD0 - 0xFFC	Management registers	*Management registers and CoreSight compliance* on page K2-7881

a. Implemented triggers, including triggers that are not connected, only. *n* is the trigger number.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table H8-6 shows the access permissions for the CTI registers in an Armv8-A Debug implementation. For a definition of the terms used, see *External debug interface registers* on page H8-7061.

Table H8-6 Access permissions for the CTI registers

			Conditions (priority from left to right)					
Offset	**Register**	**Domain**	**Off**	**DLK**	**OSLK**	**EDAD**	**Default**	**SLK**
0x000	CTICONTROL	Debug	-	-	-	-	RW	RO
0x010	CTIINTACK	Debug	-	-	-	-	WO	WI
0x014	CTIAPPSET	Debug	-	-	-	-	RW	RO
0x018	CTIAPPCLEAR	Debug	-	-	-	-	WO	WI
0x01C	CTIAPPPULSE	Debug	-	-	-	-	WO	WI
0x020+4×n	CTIINEN<n>[a]	Debug	-	-	-	-	RW	RO
0x0A0+4×n	CTIOUTEN<n>	Debug	-	-	-	-	RW	RO
0x130	CTITRIGINSTATUS	Debug	-	-	-	-	RO	RO
0x134	CTITRIGOUTSTATUS	Debug	-	-	-	-	RO	RO
0x138	CTICHINSTATUS	Debug	-	-	-	-	RO	RO
0x13C	CTICHOUTSTATUS	Debug	-	-	-	-	RO	RO
0x140	CTIGATE	Debug	-	-	-	-	RW	RO
0xFC0	CTIDEVID2	Debug	-	-	-	-	RO	RO
0xFC4	CTIDEVID1	Debug	-	-	-	-	RO	RO
0xFC8	CTIDEVID	Debug	-	-	-	-	RO	RO

a. Implemented triggers only (including triggers that are not connected). *n* is the trigger number.

For the reset values of the CTI registers, see Table H8-8 on page H8-7072.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H8.8 External debug register resets

Each register or field has a defined reset domain:

- Registers and fields in the Warm reset domain are also reset by a Cold reset and unchanged by an External Debug reset that is not coincident with a Cold reset or a Warm reset.
- Registers and fields in the Cold reset domain are unchanged by a Warm reset or an External Debug reset that is not coincident with a Cold reset.
- Registers and fields in the External Debug reset domain are unchanged by a Cold reset or a Warm reset that is not coincident with an External Debug reset.

A reset might change the value of a register. Specific rules apply to the observability of registers in the External Debug reset domain by indirect reads from the Core power domain when an External Debug reset is asserted without a coincident Cold reset. For more information, see *Synchronization of changes to the external debug registers* on page H8-7050.

Table H8-7 on page H8-7071 and Table H8-8 on page H8-7072 show the external debug register and CTI register resets. For other debug registers and Performance Monitors registers, see *Management register resets* on page K2-7888 and *Power domains and Performance Monitors registers reset* on page I3-7245.

Note

By reference to Figure H6-1 on page H6-7041 the power domain can be deduced from the reset domain. Table K2-9 on page K2-7888 also shows reset power domains.

Table H8-7 on page H8-7071 and Table H8-8 on page H8-7072 do not include:

- Read-only identification registers, such as Processor ID Registers and PMCFGR, that have a fixed value from reset.
- Read-only status registers, such as EDSCR.RW, that are evaluated each time the register is read and that have no meaningful reset value.
- Write-only registers, such as EDRCR, that only have an effect on writes, and have no meaningful reset value.
- Read/write registers, such as breakpoint and watchpoint registers, and EDPRCR.CORENPDRQ, that alias other registers. The reset values are described by the descriptions of those other registers.
- IMPLEMENTATION DEFINED registers. The reset values and reset domains of these registers are also IMPLEMENTATION DEFINED and might be UNKNOWN.

All other fields in the registers are set to an IMPLEMENTATION DEFINED value, that can be UNKNOWN. The register is in the specified reset domain.

Note

An IMPLEMENTATION DEFINED reset value, which can be UNKNOWN, means that hardware is not required to reset the register on the specified reset, but software must not rely on the register being preserved over reset.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table H8-7 Summary of external debug register resets, debug registers

Register	Reset domain	Field	Value	Description
DBGPRCR_EL1	Cold into AArch64 state	CORENPRDRQ	The value of the powerup request [a]	Debug Power Control Register.
DBGPRCR	Cold into AArch32 state	CORENPRDRQ	The value of the powerup request [a]	Debug Power Control Register.
EDESR	Warm	SS	EDECR.SS	Halting Step debug event pending
		RC	EDECR.RCE	Reset Catch debug event pending
		OSUC	0	OS Unlock Catch debug event pending
EDESR if ARMv8.3-DoPD is implemented	Cold	SS	0	Halting Step debug event pending
	Warm	RC	CTIDEVCTL .RCE	Reset Catch debug event pending
EDECR if ARMv8.3-DoPD is implemented	Cold	SS	0	Halting Step debug event enable
EDECR if ARMv8.3-DoPD is not implemented	External debug	SS	0	Halting Step debug event enable
		RCE	0	Reset Catch debug event enable
		OSUCE	0	OS Unlock Catch debug event enable
EDWAR	Cold	-	-	All fields
EDSCR	Cold	RXfull	0	DTRRX register full
		TXfull	0	DTRTX register full
		RXO	0	DTRRX overrun
		TXU	0	DTRTX underrun
		INTdis	0	Interrupt disable
		TDA	0	Trap debug register accesses to Debug state
		MA	0	Memory access mode in Debug state
		HDE	0	Halting debug mode enable
		ERR	0	Cumulative error flag
EDECCR	Cold	NSE[2:1]	`0b00`	Coarse-grained Non-secure Exception Catch
		SE[3,1]	`0b00`	Coarse-grained Secure Exception Catch
EDPCSR	Cold	-	-	All fields
EDCIDSR	Cold	-	-	All fields
EDVIDSR	Cold	-	-	All fields

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table H8-7 Summary of external debug register resets, debug registers (continued)

Register	Reset domain	Field	Value	Description
EDPRCR if ARMv8.3-DoPD is implemented	Cold	-	-	-
EDPRCR if ARMv8.3-DoPD is not implemented	External debug	COREPURQ[b]	-	Core powerup request
EDPRSR	Warm	SDR	-	Sticky debug restart
	Cold	SPMAD	0	Sticky EPMAD error
		SDAD	0	Sticky EDAD error
	Warm	SR	1	Sticky reset status
	Cold	SPD	1	Sticky powerdown status

a. If ARMv8.3-DoPD is not implemented, the powerup request is the EDPRCR.COREPURQ control bit.

b. If ARMv8.3-DoPD is not implemented, on a Cold reset into AArch64 state, DBGPRCR_EL1.CORENPRDRQ resets to the value of EDPRCR.COREPURQ. On a Cold reset into AArch32 state, DBGPRCR.CORENPRDRQ resets to the value of EDPRCR.COREPURQ. If an External Debug reset and a Cold reset coincide, both EDPRCR.COREPURQ and the CORENPRDRQ field of the appropriate System register are reset to 0.

Table H8-8 shows the reset values for the CTI registers

Table H8-8 Summary of external debug register resets, CTI registers

Register	Reset domain	Field	Value	Description
CTICONTROL	External debug	GLBEN	0	CTI global enable
CTIDEVCTL	External debug	RCE	0	If ARMv8.3-DoPD is implemented, Reset Catch debug event enable
		OSUCE	0	If ARMv8.3-DoPD is implemented, OS Unlock Catch debug event enable
CTIAPPSET	External debug	-	-	All fields
CTIINEN<n>	External debug	-	-	All fields
CTIOUTEN<n>	External debug	-	-	All fields
CTIGATE	External debug	-	-	All fields
ASICCTL	IMPLEMENTATION DEFINED	-	IMPLEMENTATION DEFINED	All of register

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter H9
External Debug Register Descriptions

This chapter provides a description of the external debug registers.

It contains the following sections:

- *About the debug registers* on page H9-7074.
- *External debug registers* on page H9-7075.
- *Cross-Trigger Interface registers* on page H9-7177.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.1 About the debug registers

The following sections describe the registers that are accessible through the external debug interface:

- *External debug registers* on page H9-7075.
- *Cross-Trigger Interface registers* on page H9-7177.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2 External debug registers

This section describes the debug registers that are accessible through the external debug interface and are used for external debug.

This section lists the registers that are accessible through the external debug interface.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.1 DBGAUTHSTATUS_EL1, Debug Authentication Status register

The DBGAUTHSTATUS_EL1 characteristics are:

Purpose

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for debug.

Configurations

External register DBGAUTHSTATUS_EL1[31:0] is architecturally mapped to AArch64 System register DBGAUTHSTATUS_EL1[31:0].

External register DBGAUTHSTATUS_EL1[31:0] is architecturally mapped to AArch32 System register DBGAUTHSTATUS[31:0].

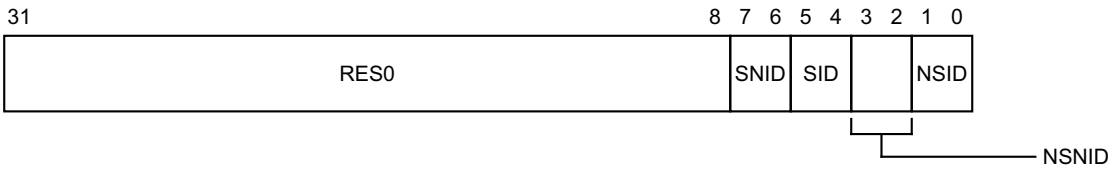
If ARMv8.3-DoPD is implemented, this register is in the Core power domain. If ARMv8.3-DoPD is not implemented, this register is in the Debug power domain.

Attributes

DBGAUTHSTATUS_EL1 is a 32-bit register.

Field descriptions

The DBGAUTHSTATUS_EL1 bit assignments are:



Bits [31:8]

Reserved, RES0.

SNID, bits [7:6]

When ARMv8.4-Debug is implemented:

Secure non-invasive debug.

ExternalSecureNoninvasiveDebugEnabled() == ExternalSecureInvasiveDebugEnabled().

This field has the same value as DBGAUTHSTATUS_EL1.SID.

Otherwise:

Secure non-invasive debug.

`0b00` Not implemented. EL3 is not implemented and the Effective value of SCR_EL3.NS is 1.

`0b10` Implemented and disabled. ExternalSecureNoninvasiveDebugEnabled() == FALSE.

`0b11` Implemented and enabled. ExternalSecureNoninvasiveDebugEnabled() == TRUE.

All other values are reserved.

SID, bits [5:4]

Secure invasive debug.

`0b00` Not implemented. EL3 is not implemented and the Effective value of SCR_EL3.NS is 1.

`0b10` Implemented and disabled. ExternalSecureInvasiveDebugEnabled() == FALSE.

`0b11` Implemented and enabled. ExternalSecureInvasiveDebugEnabled() == TRUE.

All other values are reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

NSNID, bits [3:2]

When ARMv8.4-Debug is implemented:

Non-secure non-invasive debug.

`0b00` Not implemented. EL3 is not implemented and the Effective value of SCR_EL3.NS is 0.

`0b11` Implemented and enabled. ExternalNoninvasiveDebugEnabled() == TRUE.

If the Effective value of SCR_EL3.NS is 1, or if EL3 is implemented and EL2 is not implemented, this field reads as `0b11`.

All other values are reserved.

Otherwise:

Non-secure non-invasive debug.

`0b00` Not implemented. EL3 is not implemented and the Effective value of SCR_EL3.NS is 0.

`0b10` Implemented and disabled. ExternalNoninvasiveDebugEnabled() == FALSE.

`0b11` Implemented and enabled. ExternalNoninvasiveDebugEnabled() == TRUE.

All other values are reserved.

NSID, bits [1:0]

Non-secure invasive debug.

`0b00` Not implemented. EL3 is not implemented and the Effective value of SCR_EL3.NS is 0.

`0b10` Implemented and disabled. ExternalInvasiveDebugEnabled() == FALSE.

`0b11` Implemented and enabled. ExternalInvasiveDebugEnabled() == TRUE.

All other values are reserved.

Accessing the DBGAUTHSTATUS_EL1:

DBGAUTHSTATUS_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFB8`	DBGAUTHSTATUS_EL1

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.2 DBGBCR<n>_EL1, Debug Breakpoint Control Registers, n = 0 - 15

The DBGBCR<n>_EL1 characteristics are:

Purpose

Holds control information for a breakpoint. Forms breakpoint n together with value register DBGBVR<n>_EL1.

Configurations

External register DBGBCR<n>_EL1[31:0] is architecturally mapped to AArch64 System register DBGBCR<n>_EL1[31:0].

External register DBGBCR<n>_EL1[31:0] is architecturally mapped to AArch32 System register DBGBCR<n>[31:0].

DBGBCR<n>_EL1 is in the Core power domain.

If breakpoint n is not implemented then accesses to this register are:

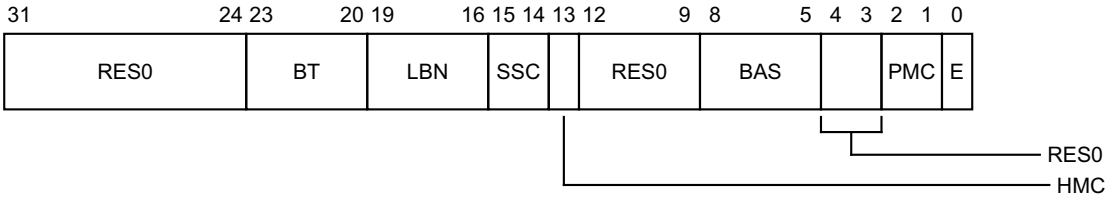
- RES0 when IsCorePowered() && !DoubleLockStatus() && !OSLockStatus() && AllowExternalDebugAccess().
- A CONSTRAINED UNPREDICTABLE choice of RES0 or ERROR otherwise.

Attributes

DBGBCR<n>_EL1 is a 32-bit register.

Field descriptions

The DBGBCR<n>_EL1 bit assignments are:



When the E field is zero, all the other fields in the register are ignored.

Bits [31:24]

Reserved, RES0.

BT, bits [23:20]

Breakpoint Type. Possible values are:

`0b0000` Unlinked instruction address match. DBGBVR<n>_EL1 is the address of an instruction.

`0b0001` As `0b0000` but linked to a Context matching breakpoint.

`0b0010` Unlinked Context ID match. When ARMv8.1-VHE is implemented, EL2 is using AArch64, and the Effective value of HCR_EL2.E2H is 1, if either the PE is executing at EL0 with HCR_EL2.TGE set to 1 or the PE is executing at EL2, then DBGBVR<n>_EL1.ContextID must match the CONTEXTIDR_EL2 value. Otherwise, DBGBVR<n>_EL1.ContextID must match the CONTEXTIDR_EL1 value.

`0b0011` As `0b0010`, with linking enabled.

`0b0100` Unlinked instruction address mismatch. DBGBVR<n>_EL1 is the address of an instruction to be stepped.

`0b0101` As `0b0100`, with linking enabled.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b0110` Unlinked CONTEXTIDR_EL1 match. DBGBVR<n>_EL1.ContextID is a Context ID compared against CONTEXTIDR_EL1.

`0b0111` As `0b0110`, with linking enabled.

`0b1000` Unlinked VMID match. DBGBVR<n>_EL1.VMID is a VMID compared against VTTBR_EL2.VMID.

`0b1001` As `0b1000`, with linking enabled.

`0b1010` Unlinked VMID and Context ID match. DBGBVR<n>_EL1.ContextID is a Context ID compared against CONTEXTIDR_EL1, and DBGBVR<n>_EL1.VMID is a VMID compared against VTTBR_EL2.VMID.

`0b1011` As `0b1010`, with linking enabled.

`0b1100` Unlinked CONTEXTIDR_EL2 match. DBGBVR<n>_EL1.ContextID2 is a Context ID compared against CONTEXTIDR_EL2.

`0b1101` As `0b1100`, with linking enabled.

`0b1110` Unlinked Full Context ID match. DBGBVR<n>_EL1.ContextID is compared against CONTEXTIDR_EL1, and DBGBVR<n>_EL1.ContextID2 is compared against CONTEXTIDR_EL2.

`0b1111` As `0b1110`, with linking enabled.

Constraints on breakpoint programming mean some values are reserved under certain conditions.

For more information on the operation of the SSC, HMC, and PMC fields, and on the effect of programming this field to a reserved value, see *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page D2-2448 and *Reserved DBGBCR<n>_EL1.BT values* on page D2-2453.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

LBN, bits [19:16]

Linked breakpoint number. For Linked address matching breakpoints, this specifies the index of the Context-matching breakpoint linked to.

For all other breakpoint types this field is ignored and reads of the register return an UNKNOWN value.

This field is ignored when the value of DBGBCR<n>_EL1.E is 0.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

SSC, bits [15:14]

Security state control. Determines the Security states under which a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the HMC and PMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information, including the effect of programming the fields to a reserved set of values, see *Reserved DBGBCR<n>_EL1.{SSC, HMC, PMC} values* on page D2-2454.

For more information on the operation of the SSC, HMC, and PMC fields, see *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page D2-2448.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

HMC, bit [13]

Higher mode control. Determines the debug perspective for deciding when a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the SSC and PMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information see DBGBCR<n>_EL1.SSC description.

For more information on the operation of the SSC, HMC, and PMC fields, see *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page D2-2448.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bits [12:9]

Reserved, RES0.

BAS, bits [8:5]

When AArch32 is supported at any Exception level:

Byte address select. Defines which half-words an address-matching breakpoint matches, regardless of the instruction set and Execution state.

The permitted values depend on the breakpoint type.

For Address match breakpoints in either AArch32 or AArch64 state, the permitted values are:

BAS	Match instruction at	Constraint for debuggers
`0b0011`	DBGBVR<n>_EL1	Use for T32 instructions
`0b1100`	DBGBVR<n>_EL1 + 2	Use for T32 instructions
`0b1111`	DBGBVR<n>_EL1	Use for A64 and A32 instructions

All other values are reserved.

For more information, see *Using the BAS field in Address Match breakpoints* on page G2-5843.

For Address mismatch breakpoints in an AArch32 stage 1 translation regime, the permitted values are:

BAS	Match instruction at	Constraint for debuggers
`0b0000`	-	Use for a match anywhere breakpoint
`0b0011`	DBGBVR<n>_EL1	Use for stepping T32 instructions
`0b1100`	DBGBVR<n>_EL1 + 2	Use for stepping T32 instructions
`0b1111`	DBGBVR<n>_EL1	Use for stepping A64 and A32 instructions

For more information, see *Using the BAS field in Address Match breakpoints* on page G2-5843.

For Context matching breakpoints, this field is RES1 and ignored.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

Bits [4:3]

Reserved, RES0.

PMC, bits [2:1]

Privilege mode control. Determines the Exception level or levels at which a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the SSC and HMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information see the DBGBCR<n>_EL1.SSC description.

For more information on the operation of the SSC, HMC, and PMC fields, see *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page D2-2448.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

E, bit [0]

Enable breakpoint DBGBVR<n>_EL1. Possible values are:

`0b0` Breakpoint disabled.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` Breakpoint enabled.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing the DBGBCR<n>_EL1:

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalDebugAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

DBGBCR<n>_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x408` + 16n	DBGBCR<n>_EL1

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalDebugAccess() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalDebugAccess() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.3 DBGBVR<n>_EL1, Debug Breakpoint Value Registers, n = 0 - 15

The DBGBVR<n>_EL1 characteristics are:

Purpose

Holds a virtual address, or a VMID and/or a context ID, for use in breakpoint matching. Forms breakpoint n together with control register DBGBCR<n>_EL1.

Configurations

External register DBGBVR<n>_EL1[63:0] is architecturally mapped to AArch64 System register DBGBVR<n>_EL1[63:0].

External register DBGBVR<n>_EL1[31:0] is architecturally mapped to AArch32 System register DBGBVR<n>[31:0].

If the breakpoint is context-aware and EL2 is implemented then External register DBGBVR<n>_EL1[63:32] is architecturally mapped to AArch32 System register DBGBXVR<n>. Otherwise there is no External register access to DBGBVR<n>_EL1[63:32] from AArch32 state.

DBGBVR<n>_EL1 is in the Core power domain.

If breakpoint n is not implemented then accesses to this register are:

- RES0 when IsCorePowered() && !DoubleLockStatus() && !OSLockStatus() && AllowExternalDebugAccess().
- A CONSTRAINED UNPREDICTABLE choice of RES0 or ERROR otherwise.

Attributes

How this register is interpreted depends on the value of DBGBCR<n>_EL1.BT.

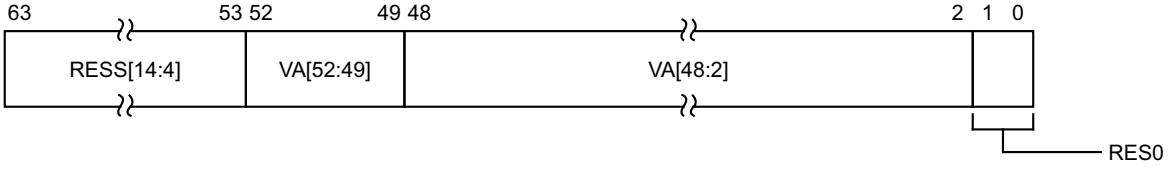
- When DBGBCR<n>_EL1.BT is `0b0x0x`, this register holds a virtual address.
- When DBGBCR<n>_EL1.BT is `0b001x`, `0b011x`, or `0b110x`, this register holds a Context ID.
- When DBGBCR<n>_EL1.BT is `0b100x`, this register holds a VMID.
- When DBGBCR<n>_EL1.BT is `0b101x`, this register holds a VMID and a Context ID.
- When DBGBCR<n>_EL1.BT is `0b111x`, this register holds two Context ID values.

For other values of DBGBCR<n>_EL1.BT, this register is RES0.

Field descriptions

The DBGBVR<n>_EL1 bit assignments are:

When DBGBCR<n>_EL1.BT == 0b0x0x:



RESS[14:4], bits [63:53]

Reserved, Sign extended. Software must treat this field as RES0 if the most significant bit of VA is 0 or RES0, and as RES1 if the most significant bit of VA is 1.

Hardware always ignores the value of these bits and it is IMPLEMENTATION DEFINED whether:

- The bits are hardwired to a copy of the most significant bit of VA, meaning writes to these bits are ignored, and reads to the bits always return the hardwired value.
- The value in those bits can be written, and reads will return the last value written. The value held in those bits is ignored by hardware.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

VA[52:49], bits [52:49]

When ARMv8.2-LVA is implemented:

Extension to VA[48:2]. See VA[48:2] for more details.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Extension to RESS[14:4]. See RESS[14:4] for more details.

VA[48:2], bits [48:2]

If the address is being matched in an AArch64 stage 1 translation regime:

- This field contains bits[48:2] of the address for comparison.
- When ARMv8.2-LVA is implemented, VA[52:49] forms the upper part of the address value. Otherwise, VA[52:49] are RESS.

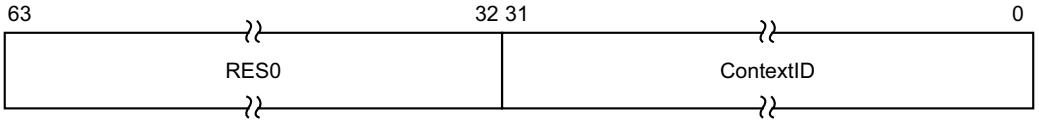
If the address is being matched in an AArch32 stage 1 translation regime, the first 20 bits of this field are RES0, and the rest of the field contains bits[31:2] of the address for comparison.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [1:0]

Reserved, RES0.

When DBGBCR<n>_EL1.BT == 0b001x:



Bits [63:32]

Reserved, RES0.

ContextID, bits [31:0]

Context ID value for comparison.

The value is compared against CONTEXTIDR_EL2 when ARMv8.1-VHE is implemented, EL2 is using AArch64, HCR_EL2.E2H is 1, and either:

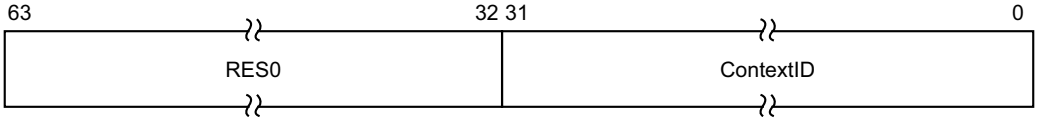
- The PE is executing at EL2.
- HCR_EL2.TGE is 1, the PE is executing at EL0, and EL2 is enabled in the current Security state.

Otherwise, the value is compared against the following:

- CONTEXTIDR when the PE is executing at AArch32
- CONTEXTIDR_EL1 when the PE is executing at AArch64.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

When DBGBCR<n>_EL1.BT == 0b011x, EL2 is implemented and (ARMv8.1-VHE is implemented or ARMv8.2-Debug is implemented):



Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bits [63:32]

Reserved, RES0.

ContextID, bits [31:0]

Context ID value for comparison against CONTEXTIDR_EL1.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

When DBGBCR<n>_EL1.BT == 0b100x and EL2 is implemented:

Bits [63:48]

Reserved, RES0.

VMID[15:8], bits [47:40]

When ARMv8.1-VHE is implemented and VTCR_EL2.VS == 1:

Extension to VMID[7:0]. See DBGBVR<n>_EL1.VMID[7:0] for more details.

If EL2 is using AArch32, this field is RES0.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

VMID[7:0], bits [39:32]

VMID value for comparison.

The VMID is 8 bits when any of the following are true:

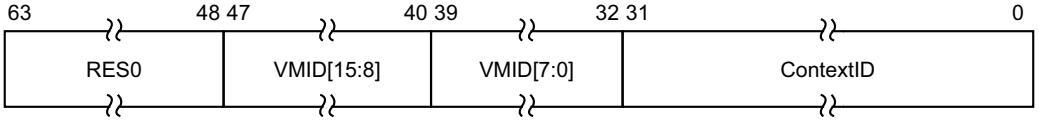
- EL2 is using AArch32.
- VTCR_EL2.VS is 0.
- ARMv8.1-VMID16 is not implemented.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [31:0]

Reserved, RES0.

When DBGBCR<n>_EL1.BT == 0b101x and EL2 is implemented:



Bits [63:48]

Reserved, RES0.

VMID[15:8], bits [47:40]

When ARMv8.1-VMID16 is implemented and VTCR_EL2.VS == 1:

Extension to VMID[7:0]. See DBGBVR<n>_EL1.VMID[7:0] for more details.

If EL2 is using AArch32, or if the implementation has an 8-bit VMID, this field is RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

VMID[7:0], bits [39:32]

VMID value for comparison.

The VMID is 8 bits when any of the following are true:

- EL2 is using AArch32.
- VTCR_EL2.VS is 0.
- ARMv8.1-VMID16 is not implemented.

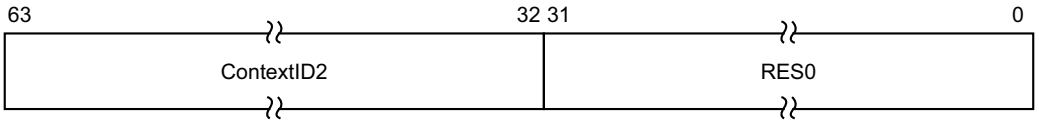
On a Cold reset, this field resets to an architecturally UNKNOWN value.

ContextID, bits [31:0]

Context ID value for comparison against CONTEXTIDR_EL1.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

When DBGBCR<n>_EL1.BT == 0b110x, EL2 is implemented and (ARMv8.1-VHE is implemented or ARMv8.2-Debug is implemented):



ContextID2, bits [63:32]

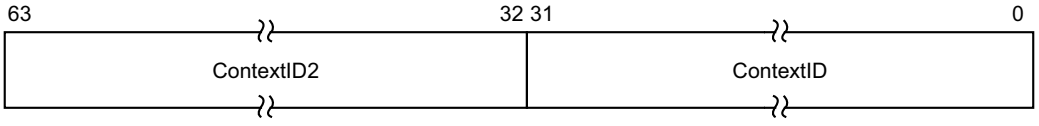
Context ID value for comparison against CONTEXTIDR_EL2.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [31:0]

Reserved, RES0.

When DBGBCR<n>_EL1.BT == 0b111x, EL2 is implemented and (ARMv8.1-VHE is implemented or ARMv8.2-Debug is implemented):



ContextID2, bits [63:32]

Context ID value for comparison against CONTEXTIDR_EL2.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

ContextID, bits [31:0]

Context ID value for comparison against CONTEXTIDR_EL1.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the DBGBVR<n>_EL1:

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalDebugAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

DBGBVR<n>_EL1[63:0] can be accessed through the external debug interface:

Component	Offset	Instance	Range
Debug	`0x400` + 16n	DBGBVR<n>_EL1	63:0

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalDebugAccess() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalDebugAccess() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.4 DBGCLAIMCLR_EL1, Debug CLAIM Tag Clear register

The DBGCLAIMCLR_EL1 characteristics are:

Purpose

Used by software to read the values of the CLAIM tag bits, and to clear CLAIM tag bits to 0.

The architecture does not define any functionality for the CLAIM tag bits.

Note

CLAIM tags are typically used for communication between the debugger and target software.

Used in conjunction with the DBGCLAIMSET_EL1 register.

Configurations

External register DBGCLAIMCLR_EL1[31:0] is architecturally mapped to AArch64 System register DBGCLAIMCLR_EL1[31:0].

External register DBGCLAIMCLR_EL1[31:0] is architecturally mapped to AArch32 System register DBGCLAIMCLR[31:0].

DBGCLAIMCLR_EL1 is in the Core power domain.

An implementation must include eight CLAIM tag bits.

Attributes

DBGCLAIMCLR_EL1 is a 32-bit register.

Field descriptions

The DBGCLAIMCLR_EL1 bit assignments are:

31 … 8	7 … 0
RAZ/SBZ	CLAIM

Bits [31:8]

Reserved, RAZ/SBZ. Software can rely on these bits reading as zero, and must use a should-be-zero policy on writes. Implementations must ignore writes.

CLAIM, bits [7:0]

Read or clear CLAIM tag bits. Reading this field returns the current value of the CLAIM tag bits.

Writing a 1 to one of these bits clears the corresponding CLAIM tag bit to 0. This is an indirect write to the CLAIM tag bits. A single write operation can clear multiple CLAIM tag bits to 0.

Writing 0 to one of these bits has no effect.

On a Cold reset, this field resets to `0`.

Accessing the DBGCLAIMCLR_EL1:

DBGCLAIMCLR_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFA4`	DBGCLAIMCLR_EL1

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.5 DBGCLAIMSET_EL1, Debug CLAIM Tag Set register

The DBGCLAIMSET_EL1 characteristics are:

Purpose

Used by software to set the CLAIM tag bits to 1.

The architecture does not define any functionality for the CLAIM tag bits.

Note

CLAIM tags are typically used for communication between the debugger and target software.

Used in conjunction with the DBGCLAIMCLR_EL1 register.

Configurations

External register DBGCLAIMSET_EL1[31:0] is architecturally mapped to AArch64 System register DBGCLAIMSET_EL1[31:0].

External register DBGCLAIMSET_EL1[31:0] is architecturally mapped to AArch32 System register DBGCLAIMSET[31:0].

DBGCLAIMSET_EL1 is in the Core power domain.

An implementation must include eight CLAIM tag bits.

Attributes

DBGCLAIMSET_EL1 is a 32-bit register.

Field descriptions

The DBGCLAIMSET_EL1 bit assignments are:

31 … 8	7 … 0
RAZ/SBZ	CLAIM

Bits [31:8]

Reserved, RAZ/SBZ. Software can rely on these bits reading as zero, and must use a should-be-zero policy on writes. Implementations must ignore writes.

CLAIM, bits [7:0]

Set CLAIM tag bits.

This field is RAO.

Writing a 1 to one of these bits sets the corresponding CLAIM tag bit to 1. This is an indirect write to the CLAIM tag bits. A single write operation can set multiple CLAIM tag bits to 1.

Writing 0 to one of these bits has no effect.

Accessing the DBGCLAIMSET_EL1:

DBGCLAIMSET_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFA0`	DBGCLAIMSET_EL1

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.6 DBGDTRRX_EL0, Debug Data Transfer Register, Receive

The DBGDTRRX_EL0 characteristics are:

Purpose

Transfers data from an external debugger to the PE. For example, it is used by a debugger transferring commands and data to a debug target. See DBGDTR_EL0 for additional architectural mappings. It is a component of the Debug Communications Channel.

Configurations

External register DBGDTRRX_EL0[31:0] is architecturally mapped to AArch64 System register DBGDTRRX_EL0[31:0].

External register DBGDTRRX_EL0[31:0] is architecturally mapped to AArch32 System register DBGDTRRXint[31:0].

DBGDTRRX_EL0 is in the Core power domain.

Attributes

DBGDTRRX_EL0 is a 32-bit register.

Field descriptions

The DBGDTRRX_EL0 bit assignments are:

31 … 0
Update DTRRX

Bits [31:0]

Update DTRRX.

Writes to this register:

- If RXfull is set to 1, set DTRRX to UNKNOWN.
- If RXfull is set to 0, update the value in DTRRX.

After the write, RXfull is set to 1.

Reads of this register:

- If RXfull is set to 1, return the last value written to DTRRX.
- If RXfull is set to 0, return an UNKNOWN value.

After the read, RXfull remains unchanged.

For the full behavior of the Debug Communications Channel, see Chapter H4 *The Debug Communication Channel and Instruction Transfer Register*.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing the DBGDTRRX_EL0:

If EDSCR.ITE == 0 when the PE exits Debug state on receiving a Restart request trigger event, the behavior of any operation issued by a DTR access in memory access mode that has not completed execution is CONSTRAINED UNPREDICTABLE, and must do one of the following:

- It must complete execution in Debug state before the PE executes the restart sequence.
- It must complete execution in Non-debug state before the PE executes the restart sequence.
- It must be abandoned. This means that the instruction does not execute. Any registers or memory accessed by the instruction are left in an UNKNOWN state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

DBGDTRRX_EL0 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x080`	DBGDTRRX_EL0

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.7 DBGDTRTX_EL0, Debug Data Transfer Register, Transmit

The DBGDTRTX_EL0 characteristics are:

Purpose

Transfers data from the PE to an external debugger. For example, it is used by a debug target to transfer data to the debugger. See DBGDTR_EL0 for additional architectural mappings. It is a component of the Debug Communication Channel.

Configurations

External register DBGDTRTX_EL0[31:0] is architecturally mapped to AArch64 System register DBGDTRTX_EL0[31:0].

External register DBGDTRTX_EL0[31:0] is architecturally mapped to AArch32 System register DBGDTRTXint[31:0].

DBGDTRTX_EL0 is in the Core power domain.

Attributes

DBGDTRTX_EL0 is a 32-bit register.

Field descriptions

The DBGDTRTX_EL0 bit assignments are:

31 … 0
Return DTRTX

Bits [31:0]

Return DTRTX.

Reads of this register:

- If TXfull is set to 1, return the last value written to DTRTX.
- If TXfull is set to 0, return an UNKNOWN value.

After the read, TXfull is cleared to 0.

Writes to this register:

- If TXfull is set to 1, set DTRTX to UNKNOWN.
- If TXfull is set to 0, update the value in DTRTX.

After the write, TXfull remains unchanged.

For the full behavior of the Debug Communications Channel, see Chapter H4 *The Debug Communication Channel and Instruction Transfer Register*.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing the DBGDTRTX_EL0:

If EDSCR.ITE == 0 when the PE exits Debug state on receiving a Restart request trigger event, the behavior of any operation issued by a DTR access in memory access mode that has not completed execution is CONSTRAINED UNPREDICTABLE, and must do one of the following:

- It must complete execution in Debug state before the PE executes the restart sequence.
- It must complete execution in Non-debug state before the PE executes the restart sequence.
- It must be abandoned. This means that the instruction does not execute. Any registers or memory accessed by the instruction are left in an UNKNOWN state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

DBGDTRTX_EL0 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x08C`	DBGDTRTX_EL0

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.8 DBGWCR<n>_EL1, Debug Watchpoint Control Registers, n = 0 - 15

The DBGWCR<n>_EL1 characteristics are:

Purpose

Holds control information for a watchpoint. Forms watchpoint n together with value register DBGWVR<n>_EL1.

Configurations

External register DBGWCR<n>_EL1[31:0] is architecturally mapped to AArch64 System register DBGWCR<n>_EL1[31:0].

External register DBGWCR<n>_EL1[31:0] is architecturally mapped to AArch32 System register DBGWCR<n>[31:0].

DBGWCR<n>_EL1 is in the Core power domain.

If watchpoint n is not implemented then accesses to this register are:

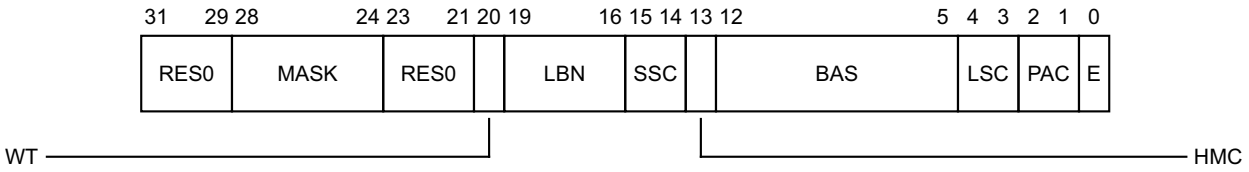
- When IsCorePowered() && !DoubleLockStatus() && !OSLockStatus() && AllowExternalDebugAccess(), RES0.
- Otherwise, a CONSTRAINED UNPREDICTABLE choice of RES0 or ERROR.

Attributes

DBGWCR<n>_EL1 is a 32-bit register.

Field descriptions

The DBGWCR<n>_EL1 bit assignments are:



When the E field is zero, all the other fields in the register are ignored.

Bits [31:29]

Reserved, RES0.

MASK, bits [28:24]

Address mask. Only objects up to 2GB can be watched using a single mask.

`0b00000` No mask.

`0b00001` Reserved.

`0b00010` Reserved.

If programmed with a reserved value, a watchpoint must behave as if either:

- MASK has been programmed with a defined value, which might be 0 (no mask), other than for a direct read of DBGWCRn_EL1.
- The watchpoint is disabled.

Software must not rely on this property because the behavior of reserved values might change in a future revision of the architecture.

Other values mask the corresponding number of address bits, from `0b00011` masking 3 address bits (`0x00000007` mask for address) to `0b11111` masking 31 address bits (`0x7FFFFFFF` mask for address).

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bits [23:21]

Reserved, RES0.

WT, bit [20]

Watchpoint type. Possible values are:

`0b0` Unlinked data address match.

`0b1` Linked data address match.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

LBN, bits [19:16]

Linked breakpoint number. For Linked data address watchpoints, this specifies the index of the Context-matching breakpoint linked to.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

SSC, bits [15:14]

Security state control. Determines the Security states under which a Watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the HMC and PAC fields.

For more information on the operation of the SSC, HMC, and PAC fields, see *Execution conditions for which a watchpoint generates Watchpoint exceptions* on page D2-2459.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

HMC, bit [13]

Higher mode control. Determines the debug perspective for deciding when a Watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the SSC and PAC fields.

For more information on the operation of the SSC, HMC, and PAC fields, see *Execution conditions for which a watchpoint generates Watchpoint exceptions* on page D2-2459.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

BAS, bits [12:5]

Byte address select. Each bit of this field selects whether a byte from within the word or double-word addressed by DBGWVR<n>_EL1 is being watched.

BAS	Description
xxxxxxx1	Match byte at DBGWVR<n>_EL1
xxxxxx1x	Match byte at DBGWVR<n>_EL1 + 1
xxxxx1xx	Match byte at DBGWVR<n>_EL1 + 2
xxxx1xxx	Match byte at DBGWVR<n>_EL1 + 3

In cases where DBGWVR<n>_EL1 addresses a double-word:

BAS	Description, if DBGWVR<n>_EL1[2] == 0
xxx1xxxx	Match byte at DBGWVR<n>_EL1 + 4
xx1xxxxx	Match byte at DBGWVR<n>_EL1 + 5
x1xxxxxx	Match byte at DBGWVR<n>_EL1 + 6
1xxxxxxx	Match byte at DBGWVR<n>_EL1 + 7

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If DBGWVR<n>_EL1[2] == 1, only BAS[3:0] is used. Arm deprecates setting DBGWVR<n>_EL1[2] == 1.

The valid values for BAS are non-zero binary number all of whose set bits are contiguous. All other values are reserved and must not be used by software. See *Reserved DBGWCR<n>.BAS values* on page G2-5865.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

LSC, bits [4:3]

Load/store control. This field enables watchpoint matching on the type of access being made. Possible values of this field are:

`0b01` Match instructions that load from a watchpointed address.

`0b10` Match instructions that store to a watchpointed address.

`0b11` Match instructions that load from or store to a watchpointed address.

All other values are reserved, but must behave as if the watchpoint is disabled. Software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

PAC, bits [2:1]

Privilege of access control. Determines the Exception level or levels at which a Watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the SSC and HMC fields.

For more information on the operation of the SSC, HMC, and PAC fields, see *Execution conditions for which a watchpoint generates Watchpoint exceptions* on page D2-2459.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

E, bit [0]

Enable watchpoint n. Possible values are:

`0b0` Watchpoint disabled.

`0b1` Watchpoint enabled.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing the DBGWCR<n>_EL1:

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalDebugAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

DBGWCR<n>_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x808` + 16n	DBGWCR<n>_EL1

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalDebugAccess() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalDebugAccess() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.9 DBGWVR<n>_EL1, Debug Watchpoint Value Registers, n = 0 - 15

The DBGWVR<n>_EL1 characteristics are:

Purpose

Holds a data address value for use in watchpoint matching. Forms watchpoint n together with control register DBGWCR<n>_EL1.

Configurations

External register DBGWVR<n>_EL1[63:0] is architecturally mapped to AArch64 System register DBGWVR<n>_EL1[63:0].

External register DBGWVR<n>_EL1[31:0] is architecturally mapped to AArch32 System register DBGWVR<n>[31:0].

DBGWVR<n>_EL1 is in the Core power domain.

If watchpoint n is not implemented then accesses to this register are:

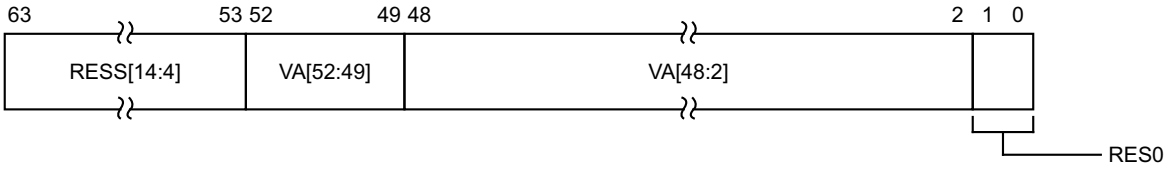
- When IsCorePowered() && !DoubleLockStatus() && !OSLockStatus() && AllowExternalDebugAccess(), RES0.
- Otherwise, a CONSTRAINED UNPREDICTABLE choice of RES0 or ERROR.

Attributes

DBGWVR<n>_EL1 is a 64-bit register.

Field descriptions

The DBGWVR<n>_EL1 bit assignments are:



RESS[14:4], bits [63:53]

Reserved, Sign extended. Hardware and software must treat this field as RES0 if the most significant bit of VA is 0 or RES0, and as RES1 if the most significant bit of VA is 1.

Hardware always ignores the value of these bits and it is IMPLEMENTATION DEFINED whether:

- The bits are hardwired to a copy of the most significant bit of VA, meaning writes to these bits are ignored, and reads to the bits always return the hardwired value.
- The value in those bits can be written, and reads will return the last value written. The value held in those bits is ignored by hardware.

VA[52:49], bits [52:49]

When ARMv8.2-LVA is implemented:

Extension to VA[48:2]. See VA[48:2] for more details.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Extension to RESS[14:4]. See RESS[14:4] for more details.

VA[48:2], bits [48:2]

Bits[48:2] of the address value for comparison.

When ARMv8.2-LVA is implemented, VA[52:49] forms the upper part of the address value. Otherwise, VA[52:49] are RESS.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Arm deprecates setting DBGWVR<n>_EL1[2] == 1.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [1:0]

Reserved, RES0.

Accessing the DBGWVR<n>_EL1:

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalDebugAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

DBGWVR<n>_EL1[63:0] can be accessed through the external debug interface:

Component	Offset	Instance	Range
Debug	`0x800` + 16n	DBGWVR<n>_EL1	63:0

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalDebugAccess() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalDebugAccess() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.10 EDAA32PFR, External Debug AArch32 Processor Feature Register

The EDAA32PFR characteristics are:

Purpose

Provides information about implemented PE features.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

It is IMPLEMENTATION DEFINED whether EDAA32PFR is implemented in the Core power domain or in the Debug power domain.

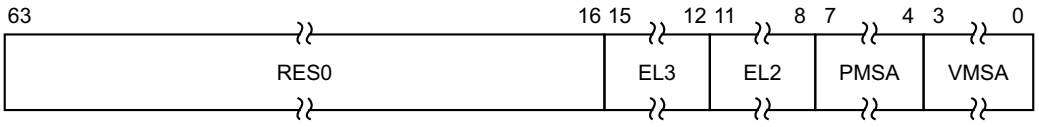
This register is present only when AArch32 is supported at any Exception level. Otherwise, direct accesses to EDAA32PFR are RES0.

Attributes

EDAA32PFR is a 64-bit register.

Field descriptions

The EDAA32PFR bit assignments are:



Bits [63:16]

Reserved, RES0.

EL3, bits [15:12]

AArch32 EL3 Exception level handling. Defined values are:

`0b0000` EL3 is not implemented or can be executed in AArch64 state.

`0b0001` EL3 can be executed in AArch32 state only.

When the value of EDPFR.EL3 is non-zero, this field must be `0b0000`.

All other values are reserved.

Note

EDPFR.{EL1, EL0} indicate whether EL1 and EL0 can only be executed in AArch32 state.

EL2, bits [11:8]

AArch32 EL2 Exception level handling. Defined values are:

`0b0000` EL2 is not implemented or can be executed in AArch64 state.

`0b0001` EL2 can be executed in AArch32 state only.

When the value of EDPFR.EL2 is non-zero, this field must be `0b0000`.

All other values are reserved.

Note

EDPFR.{EL1, EL0} indicate whether EL1 and EL0 can only be executed in AArch32 state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

PMSA, bits [7:4]

Indicates support for a PMSA. Defined values are:

`0b0000` PMSA not supported.

`0b0100` Support for an Armv8-R PMSAv8-32.

All other values are reserved. In Armv8-A, the only permitted value is `0b0000`.

VMSA, bits [3:0]

Indicates support for a VMSA. When the PMSA field is nonzero, determines support for a VMSA. When the PMSA field is `0b0000`, VMSA is supported. Defined values are:

`0b0000` VMSA not supported.

All other values are reserved. In Armv8-A, the only permitted value is `0b0000`.

Accessing the EDAA32PFR:

EDAA32PFR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xD60`	EDAA32PFR

This interface is accessible as follows:

- When IsCorePowered() and !DoubleLockStatus() accesses to this register are RO.
- Otherwise accesses to this register are IMPDEF.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.11 EDACR, External Debug Auxiliary Control Register

The EDACR characteristics are:

Purpose

Allows implementations to support IMPLEMENTATION DEFINED controls.

Configurations

It is IMPLEMENTATION DEFINED whether EDACR is implemented in the Core power domain or in the Debug power domain.

Changing this register from its reset value causes IMPLEMENTATION DEFINED behavior, including possible deviation from the architecturally-defined behavior.

If the EDACR contains any control bits that must be preserved over power down, then these bits must be accessible by the external debug interface when the OS Lock is locked, OSLSR_EL1.OSLK == 1, and when the Core is powered off.

Attributes

EDACR is a 32-bit register.

Field descriptions

The EDACR bit assignments are:

31 … 0
IMPLEMENTATION DEFINED

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

Accessing the EDACR:

EDACR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x094`	EDACR

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register are IMPDEF.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.12 EDCIDR0, External Debug Component Identification Register 0

The EDCIDR0 characteristics are:

Purpose

Provides information to identify an external debug component.

For more information see *About the Component Identification scheme* on page K2-7892.

Configurations

Implementation of this register is OPTIONAL.

If ARMv8.3-DoPD is implemented, this register is in the Core power domain. If ARMv8.3-DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

Attributes

EDCIDR0 is a 32-bit register.

Field descriptions

The EDCIDR0 bit assignments are:

31 … 8	7 … 0
RES0	PRMBL_0

Bits [31:8]

Reserved, RES0.

PRMBL_0, bits [7:0]

Preamble.

Reads as 0x0D.

Accessing the EDCIDR0:

EDCIDR0 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFF0`	EDCIDR0

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.13 EDCIDR1, External Debug Component Identification Register 1

The EDCIDR1 characteristics are:

Purpose

Provides information to identify an external debug component.

For more information see *About the Component Identification scheme* on page K2-7892.

Configurations

Implementation of this register is OPTIONAL.

If ARMv8.3-DoPD is implemented, this register is in the Core power domain. If ARMv8.3-DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

Attributes

EDCIDR1 is a 32-bit register.

Field descriptions

The EDCIDR1 bit assignments are:

31 … 8	7 … 4	3 … 0
RES0	CLASS	PRMBL_1

Bits [31:8]

Reserved, RES0.

CLASS, bits [7:4]

Component class. Debug component.

Reads as 0b1001.

PRMBL_1, bits [3:0]

Preamble.

Reads as 0b0000.

Accessing the EDCIDR1:

EDCIDR1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFF4`	EDCIDR1

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.14 EDCIDR2, External Debug Component Identification Register 2

The EDCIDR2 characteristics are:

Purpose

Provides information to identify an external debug component.

For more information see *About the Component Identification scheme* on page K2-7892.

Configurations

Implementation of this register is OPTIONAL.

If ARMv8.3-DoPD is implemented, this register is in the Core power domain. If ARMv8.3-DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

Attributes

EDCIDR2 is a 32-bit register.

Field descriptions

The EDCIDR2 bit assignments are:

31 … 8	7 … 0
RES0	PRMBL_2

Bits [31:8]

Reserved, RES0.

PRMBL_2, bits [7:0]

Preamble.

Reads as 0x05.

Accessing the EDCIDR2:

EDCIDR2 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFF8`	EDCIDR2

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.15 EDCIDR3, External Debug Component Identification Register 3

The EDCIDR3 characteristics are:

Purpose

Provides information to identify an external debug component.

For more information see *About the Component Identification scheme* on page K2-7892.

Configurations

Implementation of this register is OPTIONAL.

If ARMv8.3-DoPD is implemented, this register is in the Core power domain. If ARMv8.3-DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

Attributes

EDCIDR3 is a 32-bit register.

Field descriptions

The EDCIDR3 bit assignments are:

31 … 8	7 … 0
RES0	PRMBL_3

Bits [31:8]

Reserved, RES0.

PRMBL_3, bits [7:0]

Preamble.

Reads as 0xB1.

Accessing the EDCIDR3:

EDCIDR3 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFFC`	EDCIDR3

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.16 EDCIDSR, External Debug Context ID Sample Register

The EDCIDSR characteristics are:

Purpose

Contains the sampled value of the Context ID, captured on reading EDPCSR[31:0].

Configurations

EDCIDSR is in the Core power domain.

This register is present only when ARMv8.0-PCSample is implemented and ARMv8.2-PCSample is not implemented. Otherwise, direct accesses to EDCIDSR are RES0.

Implemented only if the OPTIONAL PC Sample-based Profiling Extension is implemented in the external debug registers space.

Note

ARMv8.2-PCSample implements the PC Sample-based Profiling Extension in the Performance Monitors registers space.

Attributes

EDCIDSR is a 32-bit register.

Field descriptions

The EDCIDSR bit assignments are:

31	0
CONTEXTIDR	

CONTEXTIDR, bits [31:0]

Context ID. The value of CONTEXTIDR that is associated with the most recent EDPCSR sample. When the most recent EDPCSR sample was generated:

- If EL1 is using AArch64, then the Context ID is sampled from CONTEXTIDR_EL1.
- If EL1 is using AArch32, then the Context ID is sampled from CONTEXTIDR.
- If EL3 is implemented and is using AArch32, then CONTEXTIDR is a banked register, and EDCIDSR samples the current banked copy of CONTEXTIDR for the Security state that is associated with the most recent EDPCSR sample.

Because the value written to EDCIDSR is an indirect read of CONTEXTIDR, it is CONSTRAINED UNPREDICTABLE whether EDCIDSR is set to the original or new value if EDPCSR samples:

- An instruction that writes to CONTEXTIDR.
- The next Context synchronization event.
- Any instruction executed between these two instructions.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing the EDCIDSR:

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register UNKNOWN, see *Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN* on page H7-7046.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

EDCIDSR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x0A4`	EDCIDSR

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus() and !OSLockStatus() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.17 EDDEVAFF0, External Debug Device Affinity register 0

The EDDEVAFF0 characteristics are:

Purpose

Copy of the low half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the external debug component relates to.

Configurations

If ARMv8.3-DoPD is implemented, this register is in the Core power domain. If ARMv8.3-DoPD is not implemented, this register is in the Debug power domain.

Attributes

EDDEVAFF0 is a 32-bit register.

Field descriptions

The EDDEVAFF0 bit assignments are:

31	0
MPIDR_EL1lo	

MPIDR_EL1lo, bits [31:0]

MPIDR_EL1 low half. Read-only copy of the low half of MPIDR_EL1, as seen from the highest implemented Exception level.

Accessing the EDDEVAFF0:

EDDEVAFF0 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFA8`	EDDEVAFF0

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.18 EDDEVAFF1, External Debug Device Affinity register 1

The EDDEVAFF1 characteristics are:

Purpose

Copy of the high half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the external debug component relates to.

Configurations

If ARMv8.3-DoPD is implemented, this register is in the Core power domain. If ARMv8.3-DoPD is not implemented, this register is in the Debug power domain.

Attributes

EDDEVAFF1 is a 32-bit register.

Field descriptions

The EDDEVAFF1 bit assignments are:

31	0
MPIDR_EL1hi	

MPIDR_EL1hi, bits [31:0]

MPIDR_EL1 high half. Read-only copy of the high half of MPIDR_EL1, as seen from the highest implemented Exception level.

Accessing the EDDEVAFF1:

EDDEVAFF1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFAC`	EDDEVAFF1

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.19 EDDEVARCH, External Debug Device Architecture register

The EDDEVARCH characteristics are:

Purpose

Identifies the programmers' model architecture of the external debug component.

Configurations

Implementation of this register is OPTIONAL.

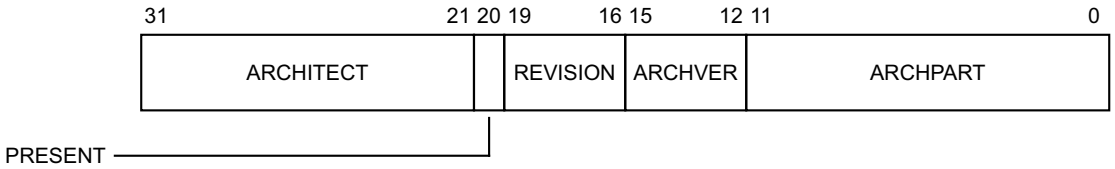
If ARMv8.3-DoPD is implemented, this register is in the Core power domain. If ARMv8.3-DoPD is not implemented, this register is in the Debug power domain.

Attributes

EDDEVARCH is a 32-bit register.

Field descriptions

The EDDEVARCH bit assignments are:



ARCHITECT, bits [31:21]

Defines the architecture of the component. For debug, this is Arm Limited.

Bits [31:28] are the JEP106 continuation code, `0x4`.

Bits [27:21] are the JEP106 ID code, `0x3B`.

PRESENT, bit [20]

When set to 1, indicates that the DEVARCH is present.

This field is 1 in Armv8.

REVISION, bits [19:16]

Defines the architecture revision. For architectures defined by Arm this is the minor revision.

For debug, the revision defined by Armv8-A is `0x0`.

All other values are reserved.

ARCHVER, bits [15:12]

Defines the architecture version of the component. This is the same value as ID_AA64DFR0_EL1.DebugVer and DBGDIDR.Version. The defined values of this field are:

`0b0110` Armv8.0 Debug architecture.

`0b0111` Armv8.0 Debug architecture with Virtualization Host Extensions.

`0b1000` Armv8.2 Debug architecture.

`0b1001` Armv8.4 Debug architecture.

ARMv8.4-Debug adds the functionality indicated by the value `0b1001`.ARMv8.2-Debug adds the functionality indicated by the value `0b1000`. If ARMv8.1-VHE is not implemented, the only permitted value is `0b0110`.

The fields ARCHVER and ARCHPART together form the field ARCHID, so that ARCHVER is ARCHID[15:12].

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ARCHPART, bits [11:0]

`0xA15` The part number of the Armv8-A debug component.

The fields ARCHVER and ARCHPART together form the field ARCHID, so that ARCHPART is ARCHID[11:0].

Accessing the EDDEVARCH:

EDDEVARCH can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFBC`	EDDEVARCH

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.20 EDDEVID, External Debug Device ID register 0

The EDDEVID characteristics are:

Purpose

Provides extra information for external debuggers about features of the debug implementation.

Configurations

If ARMv8.3-DoPD is implemented, this register is in the Core power domain. If ARMv8.3-DoPD is not implemented, this register is in the Debug power domain.

Attributes

EDDEVID is a 32-bit register.

Field descriptions

The EDDEVID bit assignments are:

31 28	27 24	23 8	7 4	3 0
RES0	AuxRegs	RES0	DebugPower	PCSample

Bits [31:28]

Reserved, RES0.

AuxRegs, bits [27:24]

Indicates support for Auxiliary registers. Permitted values for this field are:

`0b0000` None supported.

`0b0001` Support for External Debug Auxiliary Control Register, EDACR.

All other values are reserved.

Bits [23:8]

Reserved, RES0.

DebugPower, bits [7:4]

From ARMv8.3:

Indicates support for the ARMv8.3-DoPD feature. Defined values of this field are:

`0b0000` ARMv8.3-DoPD not implemented. Registers in the external debug interface register map are implemented in a mix of the Debug and Core power domains.

`0b0001` ARMv8.3-DoPD implemented. All registers in the external debug interface register map are implemented in the Core power domain.

ARMv8.3-DoPD implements the functionality added by the value `0b0001`.

All other values are reserved.

Otherwise:

Reserved, RES0.

PCSample, bits [3:0]

Indicates the level of PC Sample-based Profiling support using external debug registers. Permitted values of this field are:

`0b0000` PC Sample-based Profiling Extension is not implemented in the external debug registers space.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b0010` Only EDPCSR and EDCIDSR are implemented. This option is only permitted if EL3 and EL2 are not implemented.

`0b0011` EDPCSR, EDCIDSR, and EDVIDSR are implemented.

All other values are reserved.

When ARMv8.2-PCSample is implemented, the only permitted value is `0b0000`.

Note

ARMv8.2-PCSample implements the PC Sample-based Profiling Extension in the Performance Monitors register space, as indicated by the value of PMDEVID.PCSample.

Accessing the EDDEVID:

EDDEVID can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFC8`	EDDEVID

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.21 EDDEVID1, External Debug Device ID register 1

The EDDEVID1 characteristics are:

Purpose

Provides extra information for external debuggers about features of the debug implementation.

Configurations

If ARMv8.3-DoPD is implemented, this register is in the Core power domain. If ARMv8.3-DoPD is not implemented, this register is in the Debug power domain.

Attributes

EDDEVID1 is a 32-bit register.

Field descriptions

The EDDEVID1 bit assignments are:

31 … 4	3 … 0
RES0	PCSROffset

Bits [31:4]

Reserved, RES0.

PCSROffset, bits [3:0]

This field indicates the offset applied to PC samples returned by reads of EDPCSR. Permitted values of this field in Armv8 are:

`0b0000` EDPCSR not implemented.

`0b0010` EDPCSR implemented, and samples have no offset applied and do not sample the instruction set state in AArch32 state.

When ARMv8.2-PCSample is implemented, the only permitted value is `0b0000`.

Note

ARMv8.2-PCSample implements the PC Sample-based Profiling Extension in the Performance Monitors register space, as indicated by the value of PMDEVID.PCSample.

Accessing the EDDEVID1:

EDDEVID1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFC4`	EDDEVID1

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.22 EDDEVID2, External Debug Device ID register 2

The EDDEVID2 characteristics are:

Purpose

Reserved for future descriptions of features of the debug implementation.

Configurations

If ARMv8.3-DoPD is implemented, this register is in the Core power domain. If ARMv8.3-DoPD is not implemented, this register is in the Debug power domain.

Attributes

EDDEVID2 is a 32-bit register.

Field descriptions

The EDDEVID2 bit assignments are:

31 … 0
RES0

Bits [31:0]

Reserved, RES0.

Accessing the EDDEVID2:

EDDEVID2 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFC0`	EDDEVID2

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.23 EDDEVTYPE, External Debug Device Type register

The EDDEVTYPE characteristics are:

Purpose

Indicates to a debugger that this component is part of a PEs debug logic.

Configurations

Implementation of this register is OPTIONAL.

If ARMv8.3-DoPD is implemented, this register is in the Core power domain. If ARMv8.3-DoPD is not implemented, this register is in the Debug power domain.

Attributes

EDDEVTYPE is a 32-bit register.

Field descriptions

The EDDEVTYPE bit assignments are:

31 … 8	7 … 4	3 … 0
RES0	SUB	MAJOR

Bits [31:8]

Reserved, RES0.

SUB, bits [7:4]

Subtype. Must read as `0x1` to indicate this is a component within a PE.

MAJOR, bits [3:0]

Major type. Must read as `0x5` to indicate this is a debug logic component.

Accessing the EDDEVTYPE:

EDDEVTYPE can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFCC`	EDDEVTYPE

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.24 EDDFR, External Debug Feature Register

The EDDFR characteristics are:

Purpose

Provides top level information about the debug system.

Note

Debuggers must use EDDEVARCH to determine the Debug architecture version.

For general information about the interpretation of the ID registers, see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

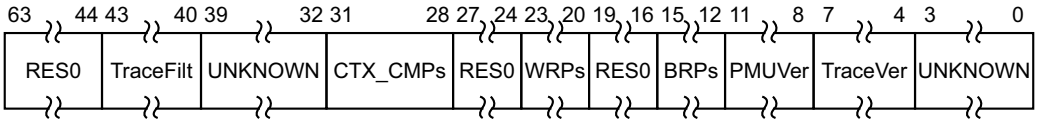
It is IMPLEMENTATION DEFINED whether EDDFR is implemented in the Core power domain or in the Debug power domain.

Attributes

EDDFR is a 64-bit register.

Field descriptions

The EDDFR bit assignments are:



Bits [63:44]

Reserved, RES0.

TraceFilt, bits [43:40]

Armv8.4 Self-hosted Trace Extension version. Defined values are:

`0b0000` Armv8.4 Self-hosted Trace Extension is not implemented.

`0b0001` Armv8.4 Self-hosted Trace Extension is implemented.

All other values are reserved.

ARMv8.4-Trace implements the functionality added by `0b0001`.

From Armv8.4, the permitted values are `0b0000` and `0b0001`.

UNKNOWN, bits [39:32]

Reserved, UNKNOWN.

CTX_CMPs, bits [31:28]

Number of breakpoints that are context-aware, minus 1. These are the highest numbered breakpoints.

In an Armv8-A implementation that supports AArch64 state in at least one Exception level, this field returns the value of ID_AA64DFR0_EL1.CTX_CMPs.

Bits [27:24]

Reserved, RES0.

WRPs, bits [23:20]

Number of watchpoints, minus 1. The value of `0b0000` is reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In an Armv8-A implementation that supports AArch64 state in at least one Exception level, this field returns the value of ID_AA64DFR0_EL1.WRPs.

Bits [19:16]

Reserved, RES0.

BRPs, bits [15:12]

Number of breakpoints, minus 1. The value of `0b0000` is reserved.

In an Armv8-A implementation that supports AArch64 state in at least one Exception level, this field returns the value of ID_AA64DFR0_EL1.BRPs.

PMUVer, bits [11:8]

Performance Monitors Extension version.

This field does not follow the standard ID scheme, but uses the Alternative ID scheme described in *Alternative ID scheme used for the Performance Monitors Extension version* on page D13-2875.

Defined values are:

`0b0000` Performance Monitors Extension not implemented.

`0b0001` Performance Monitors Extension implemented, PMUv3.

`0b0100` PMUv3 for Armv8.1. As `0b0001`, and also includes support for:

- Extended 16-bit PMEVTYPER<n>_EL0.evtCount field.
- If EL2 is implemented, the MDCR_EL2.HPMD control bit.

`0b0101` PMUv3 for Armv8.4. As `0b0100`, and also includes support for the PMMIR register.

`0b0110` PMUv3 for Armv8.5. As `0b0101`, and also includes support for:

- 64-bit event counters.
- If EL2 is implemented, the MDCR_EL2.HCCD control bit.
- If EL3 is implemented, the MDCR_EL3.SCCD control bit.

`0b1111` IMPLEMENTATION DEFINED form of performance monitors supported, PMUv3 not supported. Arm does not recommend this value in new implementations.

ARMv8.1-PMU implements the functionality added by the value `0b0100`.

ARMv8.4-PMU implements the functionality added by the value `0b0101`.

ARMv8.5-PMU implements the functionality added by the value `0b0110`.

All other values are reserved.

From Armv8.1, the value `0b0001` is not permitted.

From Armv8.4, the value `0b0100` is not permitted.

From Armv8.5, the value `0b0101` is not permitted.

In an Armv8-A implementation that supports AArch64 state in at least one Exception level, this field returns the value of ID_AA64DFR0_EL1.PMUVer.

TraceVer, bits [7:4]

Trace support. Indicates whether System register interface to a PE trace unit is implemented. Defined values are:

`0b0000` PE trace unit System registers not implemented.

`0b0001` PE trace unit System registers implemented.

All other values are reserved.

A value of `0b0000` only indicates that no System register interface to a PE trace unit is implemented. A PE trace unit might nevertheless be implemented without a System register interface.

In an Armv8-A implementation that supports AArch64 state in at least one Exception level, this field returns the value of ID_AA64DFR0_EL1.TraceVer.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

UNKNOWN, bits [3:0]

Reserved, UNKNOWN.

Accessing the EDDFR:

EDDFR[31:0] can be accessed through the external debug interface:

Component	Offset	Instance	Range
Debug	`0xD28`	EDDFR	31:0

This interface is accessible as follows:

- When IsCorePowered() and !DoubleLockStatus() accesses to EDDFR[31:0] are RO.
- Otherwise accesses to EDDFR[31:0] are IMPDEF.

EDDFR[63:32] can be accessed through the external debug interface:

Component	Offset	Instance	Range
Debug	`0xD2C`	EDDFR	63:32

This interface is accessible as follows:

- When IsCorePowered() and !DoubleLockStatus() accesses to EDDFR[63:32] are RO.
- Otherwise accesses to EDDFR[63:32] are IMPDEF.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.25 EDECCR, External Debug Exception Catch Control Register

The EDECCR characteristics are:

Purpose

Controls Exception Catch debug events.

Configurations

External register EDECCR[31:0] is architecturally mapped to AArch64 System register OSECCR_EL1[31:0].

External register EDECCR[31:0] is architecturally mapped to AArch32 System register DBGOSECCR[31:0].

EDECCR is in the Core power domain.

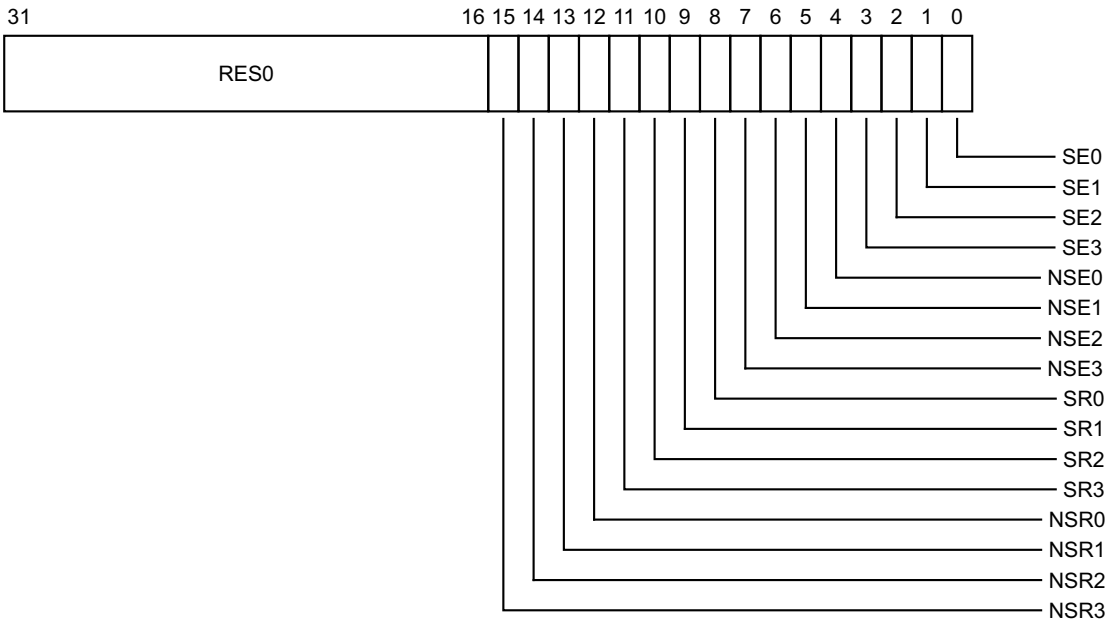
Attributes

EDECCR is a 32-bit register.

Field descriptions

The EDECCR bit assignments are:

When ARMv8.2-Debug is implemented:



Bits [31:16]

Reserved, RES0.

NSR<n>, bit [n+12], for n = 0 to 3

Controls Non-secure exception catch on exception return to EL<n> in conjunction with NSE<n>. See the Table H3-5 on page H3-6981 for information.

`0b0` If the corresponding NSE<n> bit is 0, then Exception Catch debug events are disabled for Non-secure Exception level <n>.

If the corresponding NSE<n> bit is 1, then Exception Catch debug events are enabled for exception entry, reset entry and exception return to Non-secure Exception level <n>.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` If the corresponding NSE<n> bit is 0, then Exception Catch debug events are enabled for exception returns to Non-secure Exception level <n>.

If the corresponding NSE<n> bit is 1, then Exception Catch debug events are enabled for exception entry and reset entry to Non-secure Exception level <n>.

If EL3 is not implemented and the PE behaves as if SCR_EL3.NS is set to 0, this field is reserved, RES0.

Note

It is IMPLEMENTATION DEFINED whether a reset entry to an Exception level is permitted to generate an Exception Catch debug event.

A value of the NSR field that enables an Exception Catch debug event for an Exception level that is not implemented is reserved. If the NSR field is programmed with a reserved value then:

- The PE behaves as if it is programmed with a defined value, other than for a read of EDECCR.
- The value returned for NSR by a read of EDECCR is UNKNOWN.

On a Cold reset, this field resets to `0`.

SR<n>, bit [n+8], for n = 0 to 3

Controls Secure exception catch on exception return to EL<n> in conjunction with SE<n>. See the Table H3-5 on page H3-6981 for information.

`0b0` If the corresponding SE<n> bit is 0, then Exception Catch debug events are disabled for Secure Exception level <n>.

If the corresponding SE<n> bit is 1, then Exception Catch debug events are enabled for exception entry, reset entry and exception return to Secure Exception level <n>.

`0b1` If the corresponding SE<n> bit is 0, then Exception Catch debug events are enabled for exception returns to Secure Exception level <n>.

If the corresponding SE<n> bit is 1, then Exception Catch debug events are enabled for exception entry and reset entry to Secure Exception level <n>.

If EL3 is not implemented and the PE behaves as if SCR_EL3.NS is set to 1, this field is reserved, RES0.

Note

It is IMPLEMENTATION DEFINED whether a reset entry to an Exception level is permitted to generate an Exception Catch debug event.

A value of the SR field that enables an Exception Catch debug event for an Exception level that is not implemented is reserved. If the SR field is programmed with a reserved value then:

- The PE behaves as if it is programmed with a defined value, other than for a read of EDECCR.
- The value returned for SR by a read of EDECCR is UNKNOWN.

On a Cold reset, this field resets to `0`.

NSE<n>, bit [n+4], for n = 0 to 3

Coarse-grained Non-secure exception catch for EL<n>. This controls whether Exception Catch debug events are enabled for Non-secure EL<n>. This also controls:

- The behavior of exception catch on exception entry to EL<n>.
- The behavior of exception catch on exception return to EL<n> in conjunction with NSR<n>.

`0b0` If the corresponding NSR<n> bit is 0, then Exception Catch debug events are disabled for Non-secure Exception level <n>.

If the corresponding NSR<n> bit is 1, then Exception Catch debug events are enabled for exception returns to Non-secure Exception level <n>.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` If the corresponding NSR<n> bit is 0, then Exception Catch debug events are enabled for exception entry, reset entry and exception return to Non-secure Exception level <n>.

If the corresponding NSR<n> bit is 1, then Exception Catch debug events are enabled for exception entry and reset entry to Non-secure Exception level <n>.

If EL3 is not implemented and the PE behaves as if SCR_EL3.NS is set to 0, this field is reserved, RES0.

A value of the NSE field that enables an Exception Catch debug event for an Exception level that is not implemented is reserved. If the NSE field is programmed with a reserved value then:

- The PE behaves as if it is programmed with a defined value, other than for a read of EDECCR.
- The value returned for NSE by a read of EDECCR is UNKNOWN.

On a Cold reset, this field resets to `0`.

SE<n>, bit [n], for n = 0 to 3

Coarse-grained Secure exception catch for EL<n>. This field controls whether Exception Catch debug events are enabled for Secure EL<n>.

- The behavior of exception catch on exception entry to EL<n>.
- The behavior of exception catch on exception return to EL<n> in conjunction with SR<n>.

`0b0` If the corresponding SR<n> bit is 0, then Exception Catch debug events are disabled for Secure Exception level <n>.

If the corresponding SR<n> bit is 1, then Exception Catch debug events are enabled for exception returns to Secure Exception level <n>.

`0b1` If the corresponding SR<n> bit is 0, then Exception Catch debug events are enabled for exception entry, reset entry and exception return to Secure Exception level <n>.

If the corresponding SR<n> bit is 1, then Exception Catch debug events are enabled for exception entry and reset entry to Secure Exception level <n>.

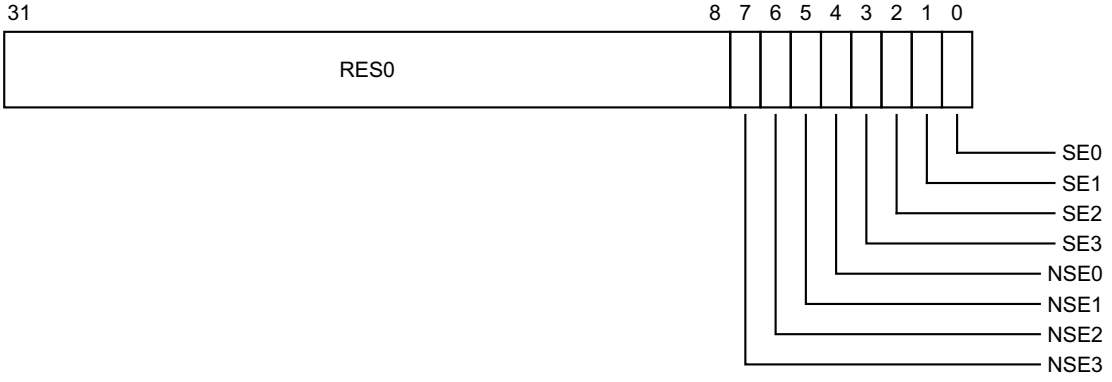
If EL3 is not implemented and the PE behaves as if SCR_EL3.NS is set to 1, this field is reserved, RES0.

A value of the SE field that enables an Exception Catch debug event for an Exception level that is not implemented is reserved. If the SE field is programmed with a reserved value then:

- The PE behaves as if it is programmed with a defined value, other than for a read of EDECCR.
- The value returned for SE by a read of EDECCR is UNKNOWN.

On a Cold reset, this field resets to `0`.

Otherwise:



Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bits [31:8]

Reserved, RES0.

NSE<n>, bit [n+4], for n = 0 to 3

Coarse-grained Non-secure exception catch. If EL3 and EL2 are not implemented and the PE behaves as if SCR_EL3.NS is set to 0, this field is reserved, RES0. Otherwise, possible values for this field are:

`0b0` Exception Catch debug events are disabled for Non-secure Exception level <n>.

`0b1` Exception Catch debug events are enabled for Non-secure Exception level <n>.

A value of the NSE field that enables an Exception Catch debug event for an Exception level that is not implemented is reserved. If the NSE field is programmed with a reserved value then:

- The PE behaves as if it is programmed with a defined value, other than for a read of EDECCR.
- The value returned for NSE by a read of EDECCR is UNKNOWN.

SE<n>, bit [n], for n = 0 to 3

Coarse-grained Secure exception catch.

`0b0` Exception Catch debug events are disabled for Secure Exception level <n>.

`0b1` Exception Catch debug events are enabled for Secure Exception level <n>.

If EL3 is not implemented and the PE behaves as if SCR_EL3.NS is set to 1, this field is reserved, RES0.

A value of the SE field that enables an Exception Catch debug event for an Exception level that is not implemented is reserved. If the SE field is programmed with a reserved value then:

- The PE behaves as if it is programmed with a defined value, other than for a read of EDECCR.
- The value returned for SE by a read of EDECCR is UNKNOWN.

Accessing the EDECCR:

EDECCR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x098`	EDECCR

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.26 EDECR, External Debug Execution Control Register

The EDECR characteristics are:

Purpose

Controls Halting debug events.

Configurations

If ARMv8.3-DoPD is implemented, this register is in the Core power domain.

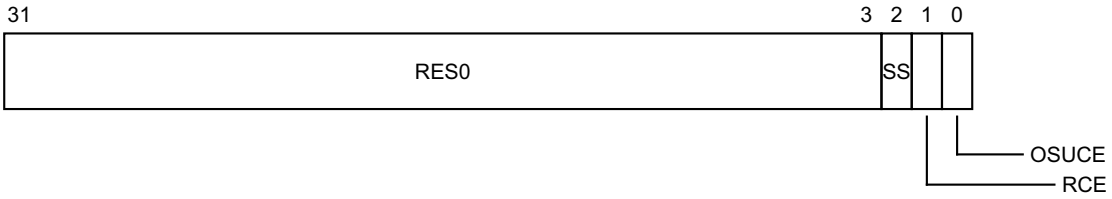
If ARMv8.3-DoPD is not implemented, this register is in the Debug power domain.

Attributes

EDECR is a 32-bit register.

Field descriptions

The EDECR bit assignments are:



Bits [31:3]

Reserved, RES0.

SS, bit [2]

Halting step enable. Possible values of this field are:

`0b0` Halting step debug event disabled.

`0b1` Halting step debug event enabled.

If the value of EDECR.SS is changed when the PE is in Non-debug state, behavior is CONSTRAINED UNPREDICTABLE as described in *Changing the value of EDECR.SS when not in Debug state* on page H3-6975.

The following resets apply:

- On a Cold reset, this field resets to:
  - If ARMv8.3-DoPD is implemented, `0`.
- On a External debug reset, this field resets to:
  - If ARMv8.3-DoPD is not implemented, `0`.
- On a Warm reset, this field resets to:
  - If ARMv8.3-DoPD is implemented, an unchanged value.

RCE, bit [1]

When ARMv8.3-DoPD is not implemented:

Reset Catch Enable.

`0b0` Reset Catch debug event disabled.

`0b1` Reset Catch debug event enabled.

The following resets apply:

- On a External debug reset, this field resets to `0`.
- On a Warm reset, the value of this field is unchanged.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

Reserved, RES0.

OSUCE, bit [0]

When ARMv8.3-DoPD is not implemented:

OS Unlock Catch Enable.

`0b0` OS Unlock Catch debug event disabled.

`0b1` OS Unlock Catch debug event enabled.

The following resets apply:

- On a External debug reset, this field resets to `0`.
- On a Warm reset, the value of this field is unchanged.

Otherwise:

Reserved, RES0.

Accessing the EDECR:

EDECR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x024`	EDECR

This interface is accessible as follows:

- When (ARMv8.3-DoPD is not implemented or IsCorePowered()) and SoftwareLockStatus() accesses to this register are RO.
- When (ARMv8.3-DoPD is not implemented or IsCorePowered()) and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.27 EDESR, External Debug Event Status Register

The EDESR characteristics are:

Purpose

Indicates the status of internally pending Halting debug events.

Configurations

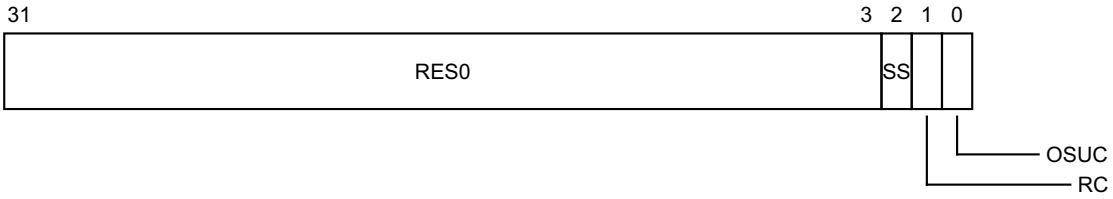
EDESR is in the Core power domain.

Attributes

EDESR is a 32-bit register.

Field descriptions

The EDESR bit assignments are:



Bits [31:3]

Reserved, RES0.

SS, bit [2]

When ARMv8.3-DoPD is implemented:

Halting step debug event pending. Possible values of this field are:

`0b0` Reading this means that a Halting step debug event is not pending. Writing this means no action.

`0b1` Reading this means that a Halting step debug event is pending. Writing this clears the pending Halting step debug event.

On a Cold reset, this field resets to `0`.

Otherwise:

Halting step debug event pending. Possible values of this field are:

`0b0` Reading this means that a Halting step debug event is not pending. Writing this means no action.

`0b1` Reading this means that a Halting step debug event is pending. Writing this clears the pending Halting step debug event.

On a Warm reset, this field resets to the value in EDECR.SS.

RC, bit [1]

Reset Catch debug event pending. Possible values of this field are:

`0b0` Reading this means that a Reset Catch debug event is not pending. Writing this means no action.

`0b1` Reading this means that a Reset Catch debug event is pending. Writing this clears the pending Reset Catch debug event.

On a Warm reset, this field resets to:

- If ARMv8.3-DoPD is implemented, the value in CTIDEVCTL.RCE.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If ARMv8.3-DoPD is not implemented, the value in EDECR.RCE.

OSUC, bit [0]

OS Unlock Catch debug event pending. Possible values of this field are:

`0b0` Reading this means that an OS Unlock Catch debug event is not pending. Writing this means no action.

`0b1` Reading this means that an OS Unlock Catch debug event is pending. Writing this clears the pending OS Unlock Catch debug event.

On a Warm reset, this field resets to `0`.

Accessing the EDESR:

If a request to clear a pending Halting debug event is received at or about the time when halting becomes allowed, it is CONSTRAINED UNPREDICTABLE whether the event is taken.

If Core power is removed while a Halting debug event is pending, it is lost. However, it might become pending again when the Core is powered back on and Cold reset.

EDESR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x020`	EDESR

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.28 EDITCTRL, External Debug Integration mode Control register

The EDITCTRL characteristics are:

Purpose

Enables the external debug to switch from its default mode into integration mode, where test software can control directly the inputs and outputs of the PE, for integration testing or topology detection.

Configurations

It is IMPLEMENTATION DEFINED whether EDITCTRL is implemented in the Core power domain or in the Debug power domain.

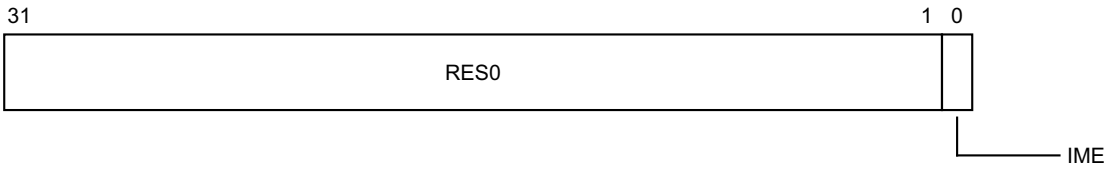
Implementation of this register is OPTIONAL.

Attributes

EDITCTRL is a 32-bit register.

Field descriptions

The EDITCTRL bit assignments are:



Bits [31:1]

Reserved, RES0.

IME, bit [0]

Integration mode enable. When IME == 1, the device reverts to an integration mode to enable integration testing or topology detection. The integration mode behavior is IMPLEMENTATION DEFINED.

`0b0` Normal operation.

`0b1` Integration mode enabled.

On a Implementation reset, this field resets to `0`.

Accessing the EDITCTRL:

EDITCTRL can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xF00`	EDITCTRL

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register are IMPDEF.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.29 EDITR, External Debug Instruction Transfer Register

The EDITR characteristics are:

Purpose

Used in Debug state for passing instructions to the PE for execution.

Configurations

EDITR is in the Core power domain.

Attributes

EDITR is a 32-bit register.

Field descriptions

The EDITR bit assignments are:

When AArch32 is supported at any Exception level and in AArch32 state:

31 … 16	15 … 0
T32Second	T32First

T32Second, bits [31:16]

Second halfword of the T32 instruction to be executed on the PE. When EDITR contains a 16-bit T32 instruction, this field is ignored. For more information see *Behavior in Debug state* on page H2-6936

T32First, bits [15:0]

First halfword of the T32 instruction to be executed on the PE.

When AArch64 is supported at any Exception level and in AArch64 state:

31 … 0
A64 instruction to be executed on the PE

Bits [31:0]

A64 instruction to be executed on the PE.

Accessing the EDITR:

If EDSCR.ITE == 0 when the PE exits Debug state on receiving a Restart request trigger event, the behavior of any instruction issued through the ITR in Normal access mode that has not completed execution is CONSTRAINED UNPREDICTABLE, and must do one of the following:

- It must complete execution in Debug state before the PE executes the restart sequence.
- It must complete execution in Non-debug state before the PE executes the restart sequence.
- It must be abandoned. This means that the instruction does not execute. Any registers or memory accessed by the instruction are left in an UNKNOWN state.

EDITR ignores writes if the PE is in Non-debug state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

EDITR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x084`	EDITR

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and SoftwareLockStatus() accesses to this register are WI.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and !SoftwareLockStatus() accesses to this register are WO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.30 EDLAR, External Debug Lock Access Register

The EDLAR characteristics are:

Purpose

Allows or disallows access to the external debug registers through a memory-mapped interface.

The optional Software Lock provides a lock to prevent memory-mapped writes to the debug registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the debug registers. It does not, and cannot, prevent all accidental or malicious damage.

Configurations

If ARMv8.3-DoPD is implemented, then ARMv8.0-SoftwareLock is not implemented by the architecturally-defined debug components of the PE in the Core power domain.

If ARMv8.3-DoPD is not implemented, then this register is in the Debug power domain.

Software uses EDLAR to set or clear the lock, and EDLSR to check the current status of the lock.

Attributes

EDLAR is a 32-bit register.

Field descriptions

The EDLAR bit assignments are:

When the Software Lock is implemented.:

31	0
KEY	

KEY, bits [31:0]

Lock Access control. Writing the key value `0xC5ACCE55` to this field unlocks the lock, enabling write accesses to this component's registers through a memory-mapped interface.

Writing any other value to this register locks the lock, disabling write accesses to this component's registers through a memory mapped interface.

Otherwise:

31	0
RES0	

Otherwise

Bits [31:0]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the EDLAR:

EDLAR can be accessed through its memory-mapped interface:

Component	Offset	Instance
Debug	`0xFB0`	EDLAR

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are WO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.31 EDLSR, External Debug Lock Status Register

The EDLSR characteristics are:

Purpose

Indicates the current status of the software lock for external debug registers.

The optional Software Lock provides a lock to prevent memory-mapped writes to the debug registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the debug registers. It does not, and cannot, prevent all accidental or malicious damage.

Configurations

If ARMv8.3-DoPD is implemented, then ARMv8.0-SoftwareLock is not implemented by the architecturally-defined debug components of the PE in the Core power domain.

If ARMv8.3-DoPD is not implemented, then this register is in the Debug power domain.

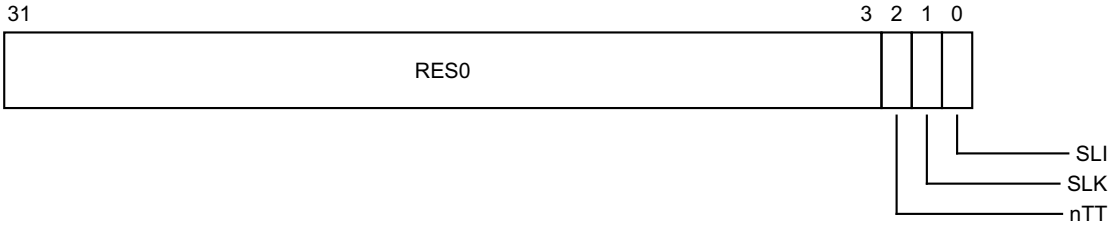
Software uses EDLAR to set or clear the lock, and EDLSR to check the current status of the lock.

Attributes

EDLSR is a 32-bit register.

Field descriptions

The EDLSR bit assignments are:



Bits [31:3]

Reserved, RES0.

nTT, bit [2]

Not thirty-two bit access required. RAZ.

SLK, bit [1]

When the Software Lock is implemented.:

Software Lock status for this component. For an access to LSR that is not a memory-mapped access, or when the Software Lock is not implemented, this field is RES0.

For memory-mapped accesses when the Software Lock is implemented, possible values of this field are:

`0b0` Lock clear. Writes are permitted to this component's registers.

`0b1` Lock set. Writes to this component's registers are ignored, and reads have no side effects.

The following resets apply:

- If Armv8.3-DoPD is implemented, this register is reset by Cold reset and not affected by External debug reset. If Armv8.3-DoPD is not implemented, this register is reset by External debug reset and not affected by Cold reset.
- This field resets to 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

Reserved, RAZ.

SLI, bit [0]

Software Lock implemented. For an access to LSR that is not a memory-mapped access, this field is RAZ. For memory-mapped accesses, the value of this field is IMPLEMENTATION DEFINED. Permitted values are:

`0b0` Software Lock not implemented or not memory-mapped access.

`0b1` Software Lock implemented and memory-mapped access.

Accessing the EDLSR:

EDLSR can be accessed through its memory-mapped interface:

Component	Offset	Instance
Debug	`0xFB4`	EDLSR

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.32 EDPCSR, External Debug Program Counter Sample Register

The EDPCSR characteristics are:

Purpose

Holds a sampled instruction address value.

Configurations

EDPCSR is in the Core power domain.

This register is present only when ARMv8.0-PCSample is implemented and ARMv8.2-PCSample is not implemented. Otherwise, direct accesses to EDPCSR are RES0.

Implemented only if the OPTIONAL PC Sample-based Profiling Extension is implemented in the external debug registers space.

Note

ARMv8.2-PCSample implements the PC Sample-based Profiling Extension in the Performance Monitors registers space.

Attributes

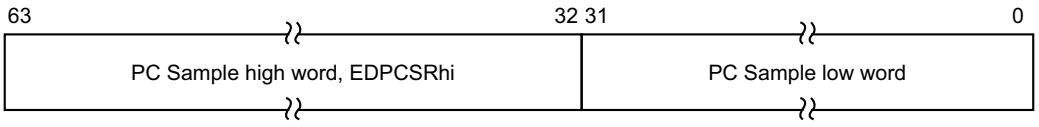
EDPCSR is a pair of 32-bit registers.

If ARMv8.1-VHE is implemented, the format of this register differs depending on the value of EDSCR.SC2.

Field descriptions

The EDPCSR bit assignments are:

When ARMv8.1-VHE is not implemented or EDSCR.SC2 == 0:



Bits [63:32]

PC Sample high word, EDPCSRhi. If EDVIDSR.HV == 0 then this field is RAZ, otherwise bits [63:32] of the sampled instruction address value. The translation regime that EDPCSR samples can be determined from EDVIDSR.{NS,E2,E3}.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [31:0]

PC Sample low word. EDPCSRlo, bits[31:0] of the sampled instruction address value.

EDPCSRlo reads as `0xFFFFFFFF` when any of the following are true:

- The PE is in Debug state.
- PC Sample-based profiling is prohibited.

If an instruction has retired since the PE left Reset state, then the first read of EDPCSR[31:0] is permitted but not required to return `0xFFFFFFFF`.

EDPCSRlo reads as an UNKNOWN value when any of the following are true:

- The PE is in Reset state.
- No instruction has retired since the PE left Reset state, Debug state, or a state where PC Sample-based Profiling is prohibited.
- No instruction has retired since the last read of EDPCSR[31:0].

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the cases where a read of EDPCSR[31:0] returns `0xFFFFFFFF` or an UNKNOWN value, the read has the side-effect of setting EDPCSRhi, EDCIDSR, and EDVIDSR to UNKNOWN values.

Otherwise, a read of EDPCSR[31:0] returns bits [31:0] of the sampled instruction address value and has the side-effect of indirectly writing to EDPCSRhi, EDCIDSR, and EDVIDSR. The translation regime that EDPCSR samples can be determined from EDVIDSR.{NS,E2,E3}.

For a read of EDPCSR[31:0] from the memory-mapped interface, if EDLSR.SLK == 1, meaning the OPTIONAL Software Lock is locked, then the side-effect of the access does not occur and EDPCSRhi, EDCIDSR, and EDVIDSR are unchanged.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

When ARMv8.1-VHE is implemented and EDSCR.SC2 == 1:

NS, bit [63]

Non-secure state sample. Indicates the Security state that is associated with the most recent EDPCSR sample or, when it is read as a single atomic 64-bit read, the current EDPCSR sample. The translation regime that EDPCSR samples can be determined from EDPCSR.{NS,EL}.

If EL3 is not implemented, this bit indicates the Effective value of SCR.NS.

`0b0` Sample is from Secure state.

`0b1` Sample is from Non-secure state.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

EL, bits [62:61]

Exception level status sample. Indicates the Exception level that is associated with the most recent EDPCSR sample or, when it is read as a single atomic 64-bit read, the current EDPCSR sample. The translation regime that EDPCSR samples can be determined from EDPCSR.{NS,EL}.

`0b00` Sample is from EL0.

`0b01` Sample is from EL1.

`0b10` Sample is from EL2.

`0b11` Sample is from EL3.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [60:56]

Reserved, RES0.

Bits [55:32]

PC Sample high word, EDPCSRhi. Bits [55:32] of the sampled instruction address value. The translation regime that EDPCSR samples can be determined from EDPCSR.{NS,EL}.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [31:0]

PC Sample low word. EDPCSRlo, bits[31:0] of the sampled instruction address value.

EDPCSRlo reads as `0xFFFFFFFF` when any of the following are true:

- The PE is in Debug state.
- PC Sample-based profiling is prohibited.

If an instruction has retired since the PE left Reset state, then the first read of EDPCSR[31:0] is permitted but not required to return `0xFFFFFFFF`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

EDPCSRlo reads as an UNKNOWN value when any of the following are true:

- The PE is in Reset state.
- No instruction has retired since the PE left Reset state, Debug state, or a state where PC Sample-based Profiling is prohibited.
- No instruction has retired since the last read of EDPCSR[31:0].

For the cases where a read of EDPCSR[31:0] returns `0xFFFFFFFF` or an UNKNOWN value, the read has the side-effect of setting EDPCSRhi, EDCIDSR, and EDVIDSR to UNKNOWN values.

Otherwise, a read of EDPCSR[31:0] returns bits [31:0] of the sampled instruction address value and has the side-effect of indirectly writing to EDPCSRhi, EDCIDSR, and EDVIDSR. The translation regime that EDPCSR samples can be determined from EDPCSR.{NS,EL}.

For a read of EDPCSR[31:0] from the memory-mapped interface, if EDLSR.SLK == 1, meaning the OPTIONAL Software Lock is locked, then the side-effect of the access does not occur and EDPCSRhi, EDCIDSR, and EDVIDSR are unchanged.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing the EDPCSR:

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register UNKNOWN, see *Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN* on page H7-7046

EDPCSR[31:0] can be accessed through its memory-mapped interface:

Component	Offset	Instance	Range
Debug	`0x0A0`	EDPCSR	31:0

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus() and !OSLockStatus() accesses to EDPCSR[31:0] are RO.
- Otherwise accesses to EDPCSR[31:0] generate an error response.

EDPCSR[63:32] can be accessed through the external debug interface:

Component	Offset	Instance	Range
Debug	`0x0AC`	EDPCSR	63:32

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus() and !OSLockStatus() accesses to EDPCSR[63:32] are RO.
- Otherwise accesses to EDPCSR[63:32] generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.33 EDPFR, External Debug Processor Feature Register

The EDPFR characteristics are:

Purpose

Provides information about implemented PE features.

For general information about the interpretation of the ID registers, see *Principles of the ID scheme for fields in ID registers* on page D13-2873.

Configurations

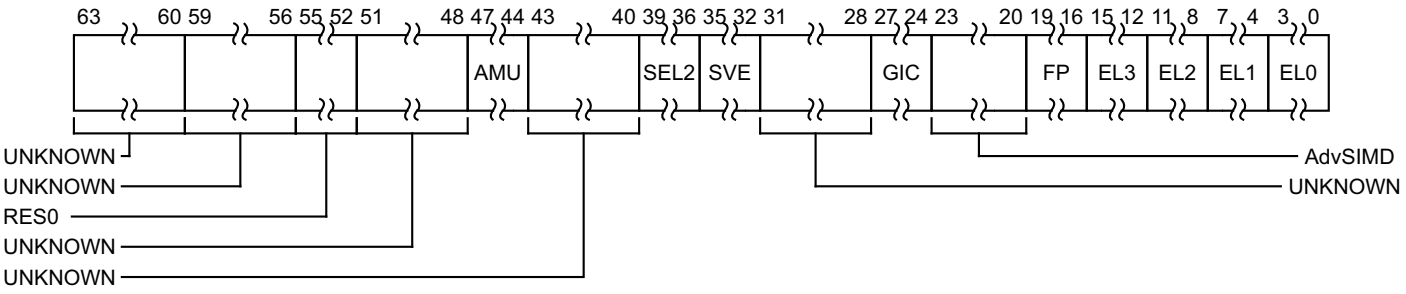
It is IMPLEMENTATION DEFINED whether EDPFR is implemented in the Core power domain or in the Debug power domain.

Attributes

EDPFR is a 64-bit register.

Field descriptions

The EDPFR bit assignments are:



UNKNOWN, bits [63:60]

From ARMv8.5:

Reserved, UNKNOWN.

Otherwise:

Reserved, RES0.

UNKNOWN, bits [59:56]

From ARMv8.5:

Reserved, UNKNOWN.

Otherwise:

Reserved, RES0.

Bits [55:52]

Reserved, RES0.

UNKNOWN, bits [51:48]

From ARMv8.4:

Reserved, UNKNOWN.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

AMU, bits [47:44]

Activity Monitors Extension. Defined values are:

`0b0000` Activity Monitors Extension is not implemented.

`0b0001` Activity Monitors Extension version 1 is implemented.

All other values are reserved.

The Activity Monitors Extension on page A2-100 implements the functionality identified by the value `0b0001`.

In Armv8.0, Armv8.1, Armv8.2, and Armv8.3, the only permitted value is `0b0000`.

From Armv8.4, the permitted values are `0b0000` and `0b0001`.

UNKNOWN, bits [43:40]

From ARMv8.2:

Reserved, UNKNOWN.

Otherwise:

Reserved, RES0.

SEL2, bits [39:36]

Secure EL2. Defined values are:

`0b0000` Secure EL2 is not implemented.

`0b0001` Secure EL2 is implemented.

All other values are reserved.

SVE, bits [35:32]

Scalable Vector Extension. Defined values are:

`0b0000` SVE is not implemented.

`0b0001` SVE is implemented.

All other values are reserved.

UNKNOWN, bits [31:28]

From ARMv8.2:

Reserved, UNKNOWN.

Otherwise:

Reserved, RES0.

GIC, bits [27:24]

System register GIC interface support. Defined values are:

`0b0000` GIC CPU interface system registers not implemented.

`0b0001` System register interface to versions 3.0 and 4.0 of the GIC CPU interface is supported.

`0b0011` System register interface to version 4.1 of the GIC CPU interface is supported.

All other values are reserved.

In an Armv8-A implementation that supports AArch64 state in at least one Exception level, this field returns the value of ID_AA64PFR0_EL1.GIC.

AdvSIMD, bits [23:20]

Advanced SIMD. Defined values are:

`0b0000` Advanced SIMD is implemented, including support for the following SISD and SIMD operations:

- Integer byte, halfword, word and doubleword element operations.
- Single-precision and double-precision floating-point arithmetic.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Conversions between single-precision and half-precision data types, and double-precision and half-precision data types.

`0b0001` As for `0b0000`, and also includes support for half-precision floating-point arithmetic.

`0b1111` Advanced SIMD is not implemented.

All other values are reserved.

This field must have the same value as the FP field.

The permitted values are:

- `0b0000` in an implementation with Advanced SIMD support, that does not include the ARMv8.2-FP16 extension.
- `0b0001` in an implementation with Advanced SIMD support, that includes the ARMv8.2-FP16 extension.
- `0b1111` in an implementation without Advanced SIMD support.

In an Armv8-A implementation that supports AArch64 state in at least one Exception level, this field returns the value of ID_AA64PFR0_EL1.AdvSIMD.

FP, bits [19:16]

Floating-point. Defined values are:

`0b0000` Floating-point is implemented, and includes support for:

- Single-precision and double-precision floating-point types.
- Conversions between single-precision and half-precision data types, and double-precision and half-precision data types.

`0b0001` As for `0b0000`, and also includes support for half-precision floating-point arithmetic.

`0b1111` Floating-point is not implemented.

All other values are reserved.

This field must have the same value as the AdvSIMD field.

The permitted values are:

- `0b0000` in an implementation with floating-point support, that does not include the ARMv8.2-FP16 extension.
- `0b0001` in an implementation with floating-point support, that includes the ARMv8.2-FP16 extension.
- `0b1111` in an implementation without floating-point support.

In an Armv8-A implementation that supports AArch64 state in at least one Exception level, this field returns the value of ID_AA64PFR0_EL1.FP.

EL3, bits [15:12]

AArch64 EL3 Exception level handling. Defined values are:

`0b0000` EL3 is not implemented or cannot be executed in AArch64 state.

`0b0001` EL3 can be executed in AArch64 state only.

`0b0010` EL3 can be executed in both Execution states.

When the value of EDAA32PFR.EL3 is non-zero, this field must be `0b0000`.

All other values are reserved.

In an Armv8-A implementation that supports AArch64 state in at least one Exception level, this field returns the value of ID_AA64PFR0_EL1.EL3.

EL2, bits [11:8]

AArch64 EL2 Exception level handling. Defined values are:

`0b0000` EL2 is not implemented or cannot be executed in AArch64 state.

`0b0001` EL2 can be executed in AArch64 state only.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b0010` EL2 can be executed in both Execution states.

When the value of EDAA32PFR.EL2 is non-zero, this field must be `0b0000`.

All other values are reserved.

In an Armv8-A implementation that supports AArch64 state in at least one Exception level, this field returns the value of ID_AA64PFR0_EL1.EL2.

EL1, bits [7:4]

AArch64 EL1 Exception level handling. Defined values are:

`0b0000` EL1 cannot be executed in AArch64 state.
EL1 can be executed in AArch32 state only.

`0b0001` EL1 can be executed in AArch64 state only.

`0b0010` EL1 can be executed in both Execution states.

All other values are reserved.

In an Armv8-A implementation that supports AArch64 state in at least one Exception level, this field returns the value of ID_AA64PFR0_EL1.EL1.

EL0, bits [3:0]

AArch64 EL0 Exception level handling. Defined values are:

`0b0000` EL0 cannot be executed in AArch64 state.
EL0 can be executed in AArch32 state only.

`0b0001` EL0 can be executed in AArch64 state only.

`0b0010` EL0 can be executed in both Execution states.

All other values are reserved.

In an Armv8-A implementation that supports AArch64 state in at least one Exception level, this field returns the value of ID_AA64PFR0_EL1.EL0.

Accessing the EDPFR:

EDPFR[31:0] can be accessed through the external debug interface:

Component	Offset	Instance	Range
Debug	`0xD20`	EDPFR	31:0

This interface is accessible as follows:

- When IsCorePowered() and !DoubleLockStatus() accesses to EDPFR[31:0] are RO.
- Otherwise accesses to EDPFR[31:0] are IMPDEF.

EDPFR[63:32] can be accessed through the external debug interface:

Component	Offset	Instance	Range
Debug	`0xD24`	EDPFR	63:32

This interface is accessible as follows:

- When IsCorePowered() and !DoubleLockStatus() accesses to EDPFR[63:32] are RO.
- Otherwise accesses to EDPFR[63:32] are IMPDEF.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.34 EDPIDR0, External Debug Peripheral Identification Register 0

The EDPIDR0 characteristics are:

Purpose

Provides information to identify an external debug component.

For more information see *About the Peripheral identification scheme* on page K2-7889.

Configurations

Implementation of this register is OPTIONAL.

If ARMv8.3-DoPD is implemented, this register is in the Core power domain. If ARMv8.3-DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

Attributes

EDPIDR0 is a 32-bit register.

Field descriptions

The EDPIDR0 bit assignments are:

31 … 8	7 … 0
RES0	PART_0

Bits [31:8]

Reserved, RES0.

PART_0, bits [7:0]

Part number, least significant byte.

Accessing the EDPIDR0:

EDPIDR0 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFE0`	EDPIDR0

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.35 EDPIDR1, External Debug Peripheral Identification Register 1

The EDPIDR1 characteristics are:

Purpose

Provides information to identify an external debug component.

For more information see *About the Peripheral identification scheme* on page K2-7889.

Configurations

Implementation of this register is OPTIONAL.

If ARMv8.3-DoPD is implemented, this register is in the Core power domain. If ARMv8.3-DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

Attributes

EDPIDR1 is a 32-bit register.

Field descriptions

The EDPIDR1 bit assignments are:

31 … 8	7 … 4	3 … 0
RES0	DES_0	PART_1

Bits [31:8]

Reserved, RES0.

DES_0, bits [7:4]

Designer, least significant nibble of JEP106 ID code. For Arm Limited, this field is `0b1011`.

PART_1, bits [3:0]

Part number, most significant nibble.

Accessing the EDPIDR1:

EDPIDR1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFE4`	EDPIDR1

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.36 EDPIDR2, External Debug Peripheral Identification Register 2

The EDPIDR2 characteristics are:

Purpose

Provides information to identify an external debug component.

For more information see *About the Peripheral identification scheme* on page K2-7889.

Configurations

Implementation of this register is OPTIONAL.

If ARMv8.3-DoPD is implemented, this register is in the Core power domain. If ARMv8.3-DoPD is not implemented, this register is in the Debug power domain.

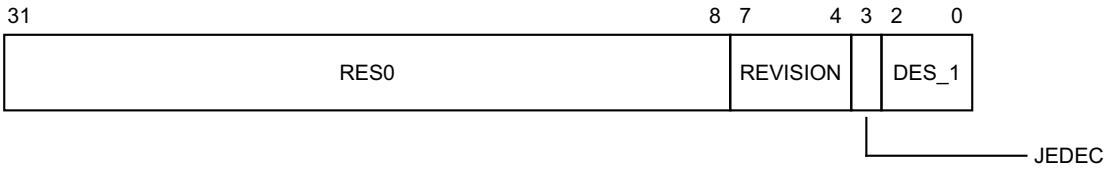
This register is required for CoreSight compliance.

Attributes

EDPIDR2 is a 32-bit register.

Field descriptions

The EDPIDR2 bit assignments are:



Bits [31:8]

Reserved, RES0.

REVISION, bits [7:4]

Part major revision. Parts can also use this field to extend Part number to 16-bits.

JEDEC, bit [3]

RAO. Indicates a JEP106 identity code is used.

DES_1, bits [2:0]

Designer, most significant bits of JEP106 ID code. For Arm Limited, this field is `0b011`.

Accessing the EDPIDR2:

EDPIDR2 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFE8`	EDPIDR2

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.37 EDPIDR3, External Debug Peripheral Identification Register 3

The EDPIDR3 characteristics are:

Purpose

Provides information to identify an external debug component.

For more information see *About the Peripheral identification scheme* on page K2-7889.

Configurations

Implementation of this register is OPTIONAL.

If ARMv8.3-DoPD is implemented, this register is in the Core power domain. If ARMv8.3-DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

Attributes

EDPIDR3 is a 32-bit register.

Field descriptions

The EDPIDR3 bit assignments are:

31 … 8	7 … 4	3 … 0
RES0	REVAND	CMOD

Bits [31:8]

Reserved, RES0.

REVAND, bits [7:4]

Part minor revision. Parts using EDPIDR2.REVISION as an extension to the Part number must use this field as a major revision number.

CMOD, bits [3:0]

Customer modified. Indicates someone other than the Designer has modified the component.

Accessing the EDPIDR3:

EDPIDR3 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFEC`	EDPIDR3

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.38 EDPIDR4, External Debug Peripheral Identification Register 4

The EDPIDR4 characteristics are:

Purpose

Provides information to identify an external debug component.

For more information see *About the Peripheral identification scheme* on page K2-7889.

Configurations

Implementation of this register is OPTIONAL.

If ARMv8.3-DoPD is implemented, this register is in the Core power domain. If ARMv8.3-DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

Attributes

EDPIDR4 is a 32-bit register.

Field descriptions

The EDPIDR4 bit assignments are:

31 … 8	7 … 4	3 … 0
RES0	SIZE	DES_2

Bits [31:8]

Reserved, RES0.

SIZE, bits [7:4]

Size of the component. RAZ. Log_2 of the number of 4KB pages from the start of the component to the end of the component ID registers.

DES_2, bits [3:0]

Designer, JEP106 continuation code, least significant nibble. For Arm Limited, this field is `0b0100`.

Accessing the EDPIDR4:

EDPIDR4 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFD0`	EDPIDR4

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.39 EDPRCR, External Debug Power/Reset Control Register

The EDPRCR characteristics are:

Purpose

Controls the PE functionality related to powerup, reset, and powerdown.

Configurations

EDPRCR contains fields that are in the Core power domain and fields that are in the Debug power domain.

If ARMv8.3-DoPD is implemented then all fields in this register are in the Core power domain.

CORENPDRQ is the only field that is mapped between the EDPRCR and DBGPRCR and DBGPRCR_EL1.

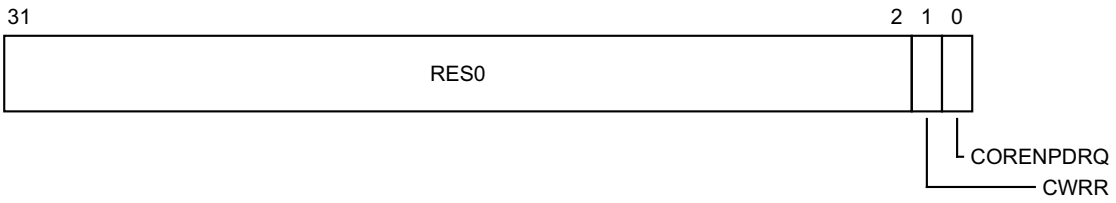
Attributes

EDPRCR is a 32-bit register.

Field descriptions

The EDPRCR bit assignments are:

When ARMv8.3-DoPD is implemented:



Bits [31:2]

Reserved, RES0.

CWRR, bit [1]

Warm reset request. Write-only bit that reads as zero.

The extent of the reset is IMPLEMENTATION DEFINED, but must be one of:

- The request is ignored.
- Only this PE is Warm reset.
- This PE and other components of the system, possibly including other PEs, are Warm reset.

Arm deprecates use of this bit, and recommends that implementations ignore the request.

`0b0` No action.

`0b1` Request Warm reset.

This field is in the Core power domain

The PE ignores writes to this bit if any of the following are true:

- ExternalInvasiveDebugEnabled() == FALSE, EL3 is not implemented, and the implemented Security state is Non-secure state.
- ExternalSecureInvasiveDebugEnabled() == FALSE, EL3 is not implemented, and the implemented Security state is Secure state.
- ExternalSecureInvasiveDebugEnabled() == FALSE and EL3 is implemented.

In an implementation that includes the recommended external debug interface, this bit drives the DBGRSTREQ signal.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

On a Warm reset, this field resets to `0`.

Accessing this field has the following behavior:

- When !IsCorePowered(), or DoubleLockStatus(), or OSLockStatus() or SoftwareLockStatus(), access to this field is WI.
- Otherwise, access to this field is WO.

CORENPDRQ, bit [0]

Core no powerdown request. Requests emulation of powerdown.

This request is typically passed to an external power controller. This means that whether a request causes power up is dependent on the IMPLEMENTATION DEFINED nature of the system. The power controller must not allow the Core power domain to switch off while this bit is 1.

`0b0` If the system responds to a powerdown request, it powers down Core power domain.

`0b1` If the system responds to a powerdown request, it does not powerdown the Core power domain, but instead emulates a powerdown of that domain.

When this bit reads as UNKNOWN, the PE ignores writes to this bit.

This field is in the Core power domain, and permitted accesses to this field map to the DBGPRCR.CORENPDRQ and DBGPRCR_EL1.CORENPDRQ fields.

In an implementation that includes the recommended external debug interface, this bit drives the DBGNOPWRDWN signal.

It is IMPLEMENTATION DEFINED whether this bit is reset the Cold reset value on exit from an IMPLEMENTATION DEFINED software-visible retention state. For more information about retention states see *Core power domain power states* on page H6-7030.

Note

Writes to this bit are not prohibited by the IMPLEMENTATION DEFINED authentication interface. This means that a debugger can request emulation of powerdown regardless of whether invasive debug is permitted.

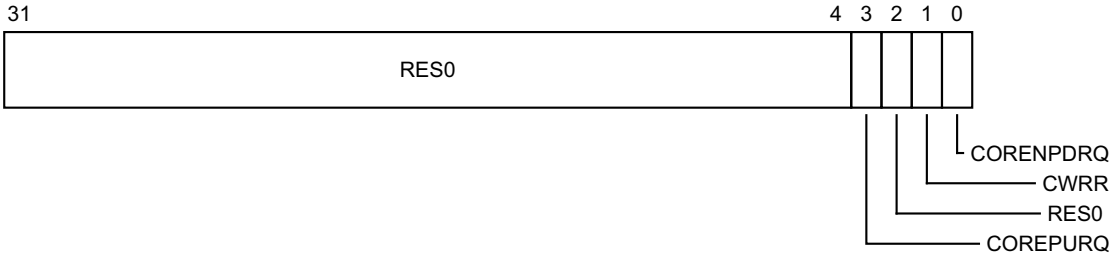
The following resets apply:

- On a Cold reset, this field is set to 1 if the powerup request is implemented and the powerup request has been asserted, and is set to 0 otherwise.
- On a Warm reset, the value of this field is unchanged.

Accessing this field has the following behavior:

- When !IsCorePowered(), or DoubleLockStatus() or OSLockStatus(), access to this field is UNKNOWN.
- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is RW.

Otherwise:



Bits [31:4]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

COREPURQ, bit [3]

Core powerup request. Allows a debugger to request that the power controller power up the core, enabling access to the debug register in the Core power domain, and that the power controller emulates powerdown.

This request is typically passed to an external power controller. This means that whether a request causes power up is dependent on the IMPLEMENTATION DEFINED nature of the system. The power controller must not allow the Core power domain to switch off while this bit is 1.

`0b0` Do not request power up of the Core power domain.

`0b1` Request power up of the Core power domain, and emulation of powerdown.

In an implementation that includes the recommended external debug interface, this bit drives the DBGPWRUPREQ signal.

This field is in the Debug power domain and can be read and written when the Core power domain is powered off.

Note

Writes to this bit are not prohibited by the IMPLEMENTATION DEFINED authentication interface. This means that a debugger can request emulation of powerdown regardless of whether invasive debug is permitted.

On a External debug reset, this field resets to `0`.

Accessing this field has the following behavior:

- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is RW.

Bit [2]

Reserved, RES0.

CWRR, bit [1]

Warm reset request. Write-only bit that reads as zero.

The extent of the reset is IMPLEMENTATION DEFINED, but must be one of:

- The request is ignored.
- Only this PE is Warm reset.
- This PE and other components of the system, possibly including other PEs, are Warm reset.

Arm deprecates use of this bit, and recommends that implementations ignore the request.

`0b0` No action.

`0b1` Request Warm reset.

This field is in the Core power domain

The PE ignores writes to this bit if any of the following are true:

- ExternalInvasiveDebugEnabled() == FALSE, EL3 is not implemented, and the implemented Security state is Non-secure state.
- ExternalSecureInvasiveDebugEnabled() == FALSE, EL3 is not implemented, and the implemented Security state is Secure state.
- ExternalSecureInvasiveDebugEnabled() == FALSE and EL3 is implemented.

In an implementation that includes the recommended external debug interface, this bit drives the DBGRSTREQ signal.

On a Warm reset, this field resets to `0`.

Accessing this field has the following behavior:

- When !IsCorePowered(), or DoubleLockStatus(), or OSLockStatus() or SoftwareLockStatus(), access to this field is WI.
- Otherwise, access to this field is WO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CORENPDRQ, bit [0]

Core no powerdown request. Requests emulation of powerdown.

This request is typically passed to an external power controller. This means that whether a request causes power up is dependent on the IMPLEMENTATION DEFINED nature of the system. The power controller must not allow the Core power domain to switch off while this bit is 1.

`0b0` If the system responds to a powerdown request, it powers down Core power domain.

`0b1` If the system responds to a powerdown request, it does not powerdown the Core power domain, but instead emulates a powerdown of that domain.

When this bit reads as UNKNOWN, the PE ignores writes to this bit.

This field is in the Core power domain, and permitted accesses to this field map to the DBGPRCR.CORENPDRQ and DBGPRCR_EL1.CORENPDRQ fields.

In an implementation that includes the recommended external debug interface, this bit drives the DBGNOPWRDWN signal.

It is IMPLEMENTATION DEFINED whether this bit is reset to the value of EDPRCR.COREPURQ on exit from an IMPLEMENTATION DEFINED software-visible retention state. For more information about retention states see *Core power domain power states* on page H6-7030.

Note

Writes to this bit are not prohibited by the IMPLEMENTATION DEFINED authentication interface. This means that a debugger can request emulation of powerdown regardless of whether invasive debug is permitted.

The following resets apply:

- On a Cold reset, this field resets to the value in EDPRCR.COREPURQ.
- On a Warm reset, the value of this field is unchanged.

Accessing this field has the following behavior:

- When !IsCorePowered(), or DoubleLockStatus() or OSLockStatus(), access to this field is UNKNOWN.
- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is RW.

Accessing the EDPRCR:

On permitted accesses to the register, other access controls affect the behavior of some fields. See the field descriptions for more information.

EDPRCR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x310`	EDPRCR

This interface is accessible as follows:

- When (ARMv8.3-DoPD is not implemented or IsCorePowered()) and SoftwareLockStatus() accesses to this register are RO.
- When (ARMv8.3-DoPD is not implemented or IsCorePowered()) and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.40 EDPRSR, External Debug Processor Status Register

The EDPRSR characteristics are:

Purpose

Holds information about the reset and powerdown state of the PE.

Configurations

EDPRSR contains fields that are in the Core power domain and fields that are in the Debug power domain.

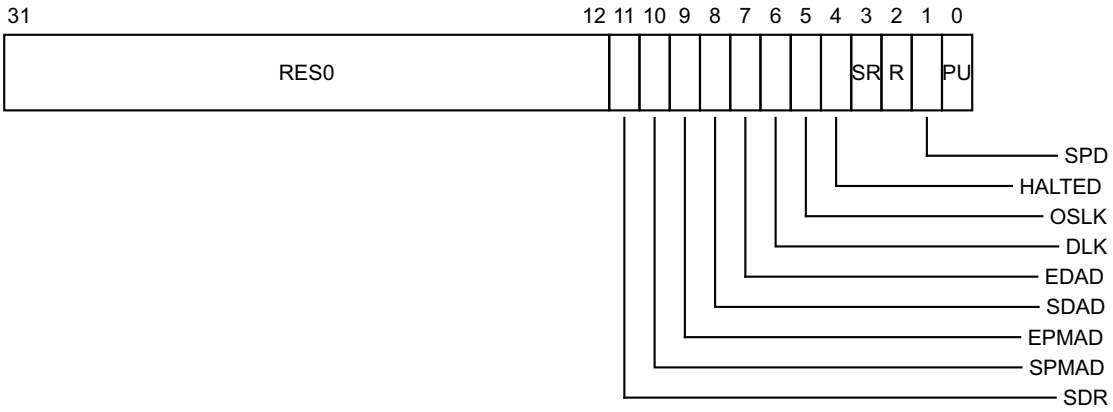
If ARMv8.3-DoPD is implemented then all fields in this register are in the Core power domain.

Attributes

EDPRSR is a 32-bit register.

Field descriptions

The EDPRSR bit assignments are:



Bits [31:12]

Reserved, RES0.

SDR, bit [11]

Sticky Debug Restart. Set to 1 when the PE exits Debug state.

Permitted values are:

`0b0` The PE has not restarted since EDPRSR was last read.

`0b1` The PE has restarted since EDPRSR was last read.

Note

If a reset occurs when the PE is in Debug state, the PE exits Debug state. SDR is UNKNOWN on Warm reset, meaning a debugger must also use the SR bit to determine whether the PE has left Debug state.

If The Core power domain is powered up, then following a read of EDPRSR:

- If ARMv8.0-DoubleLock is not implemented or `DoubleLockStatus()` == FALSE this bit clears to 0.
- If ARMv8.0-DoubleLock is implemented and `DoubleLockStatus()` == TRUE, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

This field is in the Core power domain and the Warm reset domain.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing this field has the following behavior:

- When !IsCorePowered(), or DoubleLockStatus() or EDPRSR.R == 1, access to this field is UNKNOWN.
- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is RC.

SPMAD, bit [10]

When ARMv8.4-Debug is implemented:

Sticky EPMAD error. Set to 1 if an external debug interface access to a Performance Monitors register returns an error because `AllowExternalPMUAccess()` == FALSE.

Permitted values are:

`0b0` No Non-secure external debug interface accesses to the external Performance Monitors registers have failed because `AllowExternalPMUAccess()` == FALSE for the access since EDPRSR was last read.

`0b1` At least one Non-secure external debug interface access to the external Performance Monitors register has failed and returned an error because `AllowExternalPMUAccess()` == FALSE for the access since EDPRSR was last read.

If the Core power domain is powered up, then, following a read of EDPRSR:

- If ARMv8.0-DoubleLock is not implemented or `DoubleLockStatus()` == FALSE, this bit clears to 0.
- If ARMv8.0-DoubleLock is implemented, and `DoubleLockStatus()` == TRUE, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

This field is in the Core power domain.

On a Cold reset, this field resets to `0`.

Accessing this field has the following behavior:

- When !IsCorePowered(), or DoubleLockStatus() or EDPRSR.R == 1, access to this field is UNKNOWN.
- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is RC.

Otherwise:

Sticky EPMAD error.

`0b0` No external debug interface accesses to the Performance Monitors registers have failed because `AllowExternalPMUAccess()` == FALSE since EDPRSR was last read.

`0b1` At least one external debug interface access to the Performance Monitors registers has failed and returned an error because `AllowExternalPMUAccess()` == FALSE since EDPRSR was last read.

If the Core power domain is powered up, then, following a read of EDPRSR:

- If ARMv8.0-DoubleLock is not implemented or `DoubleLockStatus()` == FALSE, this bit clears to 0.
- If ARMv8.0-DoubleLock is implemented, and `DoubleLockStatus()` == TRUE, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

This field is in the Core power domain.

On a Cold reset, this field resets to `0`.

Accessing this field has the following behavior:

- When !IsCorePowered(), or OSLockStatus(), or DoubleLockStatus() or EDPRSR.R == 1, access to this field is UNKNOWN.
- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is RC.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

EPMAD, bit [9]

When ARMv8.4-Debug is implemented:

External Performance Monitors Access Disable status.

`0b0` External Non-secure Performance Monitors access enabled. `AllowExternalPMUAccess()` == TRUE for a Non-secure access.

`0b1` External Non-secure Performance Monitors access disabled. `AllowExternalPMUAccess()` == FALSE for a Non-secure access.

This field is in the Core power domain.

Accessing this field has the following behavior:

- When !IsCorePowered(), or DoubleLockStatus() or EDPRSR.R == 1, access to this field is UNKNOWN.
- Otherwise, access to this field is RO.

Otherwise:

External Performance Monitors access disable status.

`0b0` External Performance Monitors access enabled. `AllowExternalPMUAccess()` == TRUE.

`0b1` External Performance Monitors access disabled. `AllowExternalPMUAccess()` == FALSE.

This field is in the Core power domain.

Accessing this field has the following behavior:

- When !IsCorePowered(), or OSLockStatus(), or DoubleLockStatus() or EDPRSR.R == 1, access to this field is UNKNOWN.
- Otherwise, access to this field is RO.

SDAD, bit [8]

When ARMv8.4-Debug is implemented:

Sticky EDAD error. Set to 1 if an external debug interface access to a debug register returns an error because `AllowExternalDebugAccess()` == FALSE.

`0b0` No Non-secure external debug interface accesses to the debug registers have failed because `AllowExternalDebugAccess()` == FALSE for the access since EDPRSR was last read.

`0b1` At least one Non-secure external debug interface access to the debug registers has failed and returned an error because `AllowExternalDebugAccess()` == FALSE for the access since EDPRSR was last read.

If the Core power domain is powered up, then, following a read of EDPRSR:

- If ARMv8.0-DoubleLock is not implemented or `DoubleLockStatus()` == FALSE this bit clears to 0.
- If ARMv8.0-DoubleLock is implemented and `DoubleLockStatus()` == TRUE, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

This field is in the Core power domain.

On a Cold reset, this field resets to `0`.

Accessing this field has the following behavior:

- When !IsCorePowered(), or DoubleLockStatus() or EDPRSR.R == 1, access to this field is UNKNOWN.
- Otherwise, access to this field is RO.

Otherwise:

Sticky EDAD error. Set to 1 if an external debug interface access to a debug register returns an error because `AllowExternalDebugAccess()` == FALSE.

`0b0` No external debug interface accesses to the debug registers have failed because `AllowExternalDebugAccess()` == FALSE since EDPRSR was last read.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` At least one external debug interface access to the debug registers has failed and returned an error because `AllowExternalDebugAccess()` == FALSE since EDPRSR was last read.

If the Core power domain is powered up, then, following a read of EDPRSR:

- If ARMv8.0-DoubleLock is not implemented or `DoubleLockStatus()` == FALSE this bit clears to 0.
- If ARMv8.0-DoubleLock is implemented and `DoubleLockStatus()` == TRUE, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

This bit is UNKNOWN on reads if `OSLockStatus()` == TRUE and external debug writes to OSLAR_EL1 do not return an error when `AllowExternalDebugAccess()` == FALSE.

This field is in the Core power domain.

On a Cold reset, this field resets to `0`.

Accessing this field has the following behavior:

- When !IsCorePowered(), or DoubleLockStatus() or EDPRSR.R == 1, access to this field is UNKNOWN.
- Otherwise, access to this field is RO.

EDAD, bit [7]

When ARMv8.4-Debug is implemented:

External Debug Access Disable status.

`0b0` External Non-secure access to breakpoint registers, watchpoint registers, and OSLAR_EL1 enabled. `AllowExternalDebugAccess()` == TRUE for a Non-secure access.

`0b1` External Non-secure access to breakpoint registers, watchpoint registers, and OSLAR_EL1 disabled. `AllowExternalDebugAccess()` == FALSE for a Non-secure access.

This field is in the Core power domain.

Accessing this field has the following behavior:

- When !IsCorePowered(), or DoubleLockStatus() or EDPRSR.R == 1, access to this field is UNKNOWN.
- Otherwise, access to this field is RO.

When ARMv8.2-Debug is implemented:

External Debug Access Disable status.

`0b0` External access to breakpoint registers, watchpoint registers, and OSLAR_EL1 enabled. `AllowExternalDebugAccess()` == TRUE.

`0b1` External access to breakpoint registers, watchpoint registers, and OSLAR_EL1 disabled. `AllowExternalDebugAccess()` == FALSE.

This bit is not valid and reads UNKNOWN if `OSLockStatus()` == TRUE and external debug writes to OSLAR_EL1 do not return an error when `AllowExternalDebugAccess()` == FALSE.

This field is in the Core power domain.

Accessing this field has the following behavior:

- When !IsCorePowered(), or DoubleLockStatus() or EDPRSR.R == 1, access to this field is UNKNOWN.
- Otherwise, access to this field is RO.

Otherwise:

External Debug Access Disable status.

`0b0` External access to breakpoint registers, watchpoint registers, and OSLAR_EL1 enabled. `AllowExternalDebugAccess()` == TRUE.

`0b1` External access to breakpoint registers, watchpoint registers disabled. It is IMPLEMENTATION DEFINED whether accesses to OSLAR_EL1 are enabled or disabled. `AllowExternalDebugAccess()` == FALSE.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field is in the Core power domain.

Accessing this field has the following behavior:

- When !IsCorePowered(), or DoubleLockStatus() or EDPRSR.R == 1, access to this field is UNKNOWN.
- Otherwise, access to this field is RO.

DLK, bit [6]

When ARMv8.4-Debug is implemented:

This field is RES0.

When ARMv8.2-Debug is implemented and ARMv8.0-DoubleLock is implemented:

Double Lock.

From Armv8.2, this field is deprecated.

This field is in the Core power domain.

Accessing this field has the following behavior:

- When IsCorePowered() and !DoubleLockStatus(), access to this field is RAZ.
- Otherwise, access to this field is UNKNOWN.

When ARMv8.0-DoubleLock is implemented:

Double Lock.

This field returns the result of the pseudocode function `DoubleLockStatus()`.

If the Core power domain is powered up and `DoubleLockStatus()` == TRUE, it is IMPLEMENTATION DEFINED whether:

- EDPRSR.PU reads as 1, EDPRSR.DLK reads as 1, and EDPRSR.SPD is UNKNOWN.
- EDPRSR.PU reads as 0, EDPRSR.DLK is UNKNOWN, and EDPRSR.SPD reads as 0.

This field is in the Core power domain.

`0b0` `DoubleLockStatus()` returns FALSE.

`0b1` `DoubleLockStatus()` returns TRUE and the Core power domain is powered up.

Accessing this field has the following behavior:

- When !IsCorePowered(), access to this field is UNKNOWN.
- Otherwise, access to this field is RO.

Otherwise:

Reserved, RES0.

OSLK, bit [5]

OS Lock status bit.

A read of this bit returns the value of OSLSR_EL1.OSLK.

This field is in the Core power domain.

Accessing this field has the following behavior:

- When !IsCorePowered(), DoubleLockStatus() and EDPRSR.R == 1, access to this field is UNKNOWN.
- Otherwise, access to this field is RO.

HALTED, bit [4]

Halted status bit.

`0b0` PE is in Non-debug state.

`0b1` PE is in Debug state.

This field is in the Core power domain.

Accessing this field has the following behavior:

- When !IsCorePowered(), access to this field is UNKNOWN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Otherwise, access to this field is RO.

SR, bit [3]

Sticky core Reset status bit.

Permitted values are:

`0b0` The non-debug logic of the PE is not in reset state and has not been reset since the last time EDPRSR was read.

`0b1` The non-debug logic of the PE is in reset state or has been reset since the last time EDPRSR was read.

If EDPRSR.PU reads as 1 and EDPRSR.R reads as 0, which means that the Core power domain is in a powerup state and that the non-debug logic of the PE is not in reset state, then following a read of EDPRSR:

- If ARMv8.0-DoubleLock is not implemented or `DoubleLockStatus()` == FALSE this bit clears to 0.
- If ARMv8.0-DoubleLock is implemented and `DoubleLockStatus()` == TRUE, it is UNPREDICTABLE whether this bit clears to 0 or is unchanged.

This field is in the Core power domain and the Warm reset domain.

This field resets to `1`.

Accessing this field has the following behavior:

- When !IsCorePowered() or DoubleLockStatus(), access to this field is UNKNOWN.
- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is RC.

R, bit [2]

PE Reset status bit.

Permitted values are:

`0b0` The non-debug logic of the PE is not in reset state.

`0b1` The non-debug logic of the PE is in reset state.

If ARMv8.0-DoubleLock is implemented, the PE is in reset state, and the PE entered reset state with the OS Double Lock locked this bit has a CONSTRAINED UNPREDICTABLE value. For more information see *EDPRSR.{DLK, R} and reset state* on page H6-7036

This field is in the Core power domain.

Accessing this field has the following behavior:

- When !IsCorePowered() or DoubleLockStatus(), access to this field is UNKNOWN.
- Otherwise, access to this field is RO.

SPD, bit [1]

Sticky core Powerdown status bit.

If ARMv8.0-DoubleLock is implemented and `DoubleLockStatus()` == TRUE, then:

- If ARMv8.2-Debug is implemented, this bit reads as 0.
- If ARMv8.2-Debug is not implemented, this bit might read as 0 or 1.

For more information, see *EDPRSR.{DLK, SPD, PU} and the Core power domain* on page H6-7035.

`0b0` If EDPRSR.PU is 0, it is not known whether the state of the debug registers in the Core power domain is lost.

If EDPRSR.PU is 1, the state of the debug registers in the Core power domain has not been lost.

`0b1` The state of the debug registers in the Core power domain has been lost.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the Core power domain is powered up, then, following a read of EDPRSR:

- If ARMv8.0-DoubleLock is not implemented or `DoubleLockStatus()` == FALSE this bit clears to 0.
- If ARMv8.0-DoubleLock is implemented and `DoubleLockStatus()` == TRUE, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

When the Core power domain is in either retention or powerdown state, the value of EDPRSR.SPD is IMPLEMENTATION DEFINED. For more information, see *EDPRSR.SPD when the Core domain is in either retention or powerdown state* on page H6-7036.

EDPRSR.{DLK, SPD, PU} describe whether registers in the Core power domain can be accessed, and whether their state has been lost since the last time the register was read. For more information, see *EDPRSR.{DLK, SPD, PU} and the Core power domain* on page H6-7035.

This field is in the Core power domain and the Cold reset domain.

On a Cold reset, this field resets to `1`.

Accessing this field has the following behavior:

- When !IsCorePowered(), access to this field is RAZ.
- When IsCorePowered() and DoubleLockStatus(), access to this field is UNKNOWN.
- Otherwise, access to this field is RO.

PU, bit [0]

When ARMv8.3-DoPD is implemented:

Core powerup status bit.

Access to this field is RAO.

When ARMv8.2-Debug is implemented:

Core Powerup status bit. Indicates whether the debug registers in the Core power domain can be accessed.

`0b0` Either the Core power domain is in a low-power or powerdown state, or ARMv8.0-DoubleLock is implemented and `DoubleLockStatus()` == TRUE, meaning the debug registers in the Core power domain cannot be accessed.

`0b1` The Core power domain is in a powerup state, and either ARMv8.0-DoubleLock is not implemented or `DoubleLockStatus()` == FALSE, meaning the debug registers in the Core power domain can be accessed.

If ARMv8.0-DoubleLock is implemented, the PE is in reset state, and the PE entered reset state with the OS Double Lock locked this bit has a CONSTRAINED UNPREDICTABLE value. For more information see *EDPRSR.{DLK, R} and reset state* on page H6-7036

EDPRSR.{DLK, SPD, PU} describe whether registers in the Core power domain can be accessed, and whether their state has been lost since the last time the register was read. For more information, see *EDPRSR.{DLK, SPD, PU} and the Core power domain* on page H6-7035

Access to this field is RO.

Otherwise:

Core Powerup status bit. Indicates whether the debug registers in the Core power domain can be accessed.

When the Core power domain is powered-up and `DoubleLockStatus()` == TRUE, then the value of EDPRSR.PU is IMPLEMENTATION DEFINED. See the description of the DLK bit for more information.

Otherwise, permitted values are:

`0b0` Core power domain is in a low-power or powerdown state where the debug registers in the Core power domain cannot be accessed.

`0b1` Core power domain is in a powerup state where the debug registers in the Core power domain can be accessed.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If ARMv8.0-DoubleLock is implemented, the Core power domain is powered up, and `DoubleLockStatus()` == TRUE, it is IMPLEMENTATION DEFINED whether this bit reads as 0 or 1.

If ARMv8.0-DoubleLock is implemented, the PE is in reset state, and the PE entered reset state with the OS Double Lock locked this bit has a CONSTRAINED UNPREDICTABLE value. For more information see *EDPRSR.{DLK, R} and reset state* on page H6-7036

EDPRSR.{DLK, SPD, PU} describe whether registers in the Core power domain can be accessed, and whether their state has been lost since the last time the register was read. For more information, see *EDPRSR.{DLK, SPD, PU} and the Core power domain* on page H6-7035

Access to this field is RO.

Accessing the EDPRSR:

On permitted accesses to the register, other access controls affect the behavior of some fields. See the field descriptions for more information.

If the Core power domain is powered up (EDPRSR.PU == 1), then following a read of EDPRSR:

- If ARMv8.0-DoubleLock is not implemented or `DoubleLockStatus()` == FALSE, then:
  - EDPRSR.{SDR, SPMAD, SDAD, SPD} are cleared to 0.
  - EDPRSR.SR is cleared to 0 if the non-debug logic of the PE is not in reset state (EDPRSR.R == 0).
- Otherwise it is CONSTRAINED UNPREDICTABLE whether or not this clearing occurs.

If the Core power domain is powered down (EDPRSR.PU == 0), then:

- EDPRSR.{SDR, SPMAD, SDAD, SR} are all UNKNOWN, and are either reset or restored on being powered up.
- EDPRSR.SPD is not cleared following a read of EDPRSR. See the SPD bit description for more information.

The clearing of bits is an indirect write to EDPRSR.

EDPRSR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x314`	EDPRSR

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.41 EDRCR, External Debug Reserve Control Register

The EDRCR characteristics are:

Purpose

This register is used to allow imprecise entry to Debug state and clear sticky bits in EDSCR.

Configurations

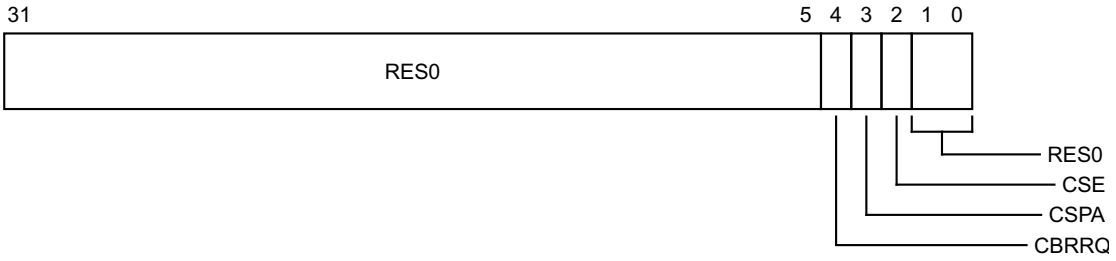
EDRCR is in the Core power domain.

Attributes

EDRCR is a 32-bit register.

Field descriptions

The EDRCR bit assignments are:



Bits [31:5]

Reserved, RES0.

CBRRQ, bit [4]

Allow imprecise entry to Debug state. The actions on writing to this bit are:

`0b0` No action.

`0b1` Allow imprecise entry to Debug state, for example by canceling pending bus accesses.

Setting this bit to 1 allows a debugger to request imprecise entry to Debug state. An External Debug Request debug event must be pending before the debugger sets this bit to 1.

This feature is optional. If this feature is not implemented, writes to this bit are ignored.

CSPA, bit [3]

Clear Sticky Pipeline Advance. This bit is used to clear the EDSCR.PipeAdv bit to 0. The actions on writing to this bit are:

`0b0` No action.

`0b1` Clear the EDSCR.PipeAdv bit to 0.

CSE, bit [2]

Clear Sticky Error. Used to clear the EDSCR cumulative error bits to 0. The actions on writing to this bit are:

`0b0` No action.

`0b1` Clear the EDSCR.{TXU, RXO, ERR} bits, and, if the PE is in Debug state, the EDSCR.ITO bit, to 0.

Bits [1:0]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the EDRCR:

EDRCR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x090`	EDRCR

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and SoftwareLockStatus() accesses to this register are WI.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and !SoftwareLockStatus() accesses to this register are WO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.42 EDSCR, External Debug Status and Control Register

The EDSCR characteristics are:

Purpose

Main control register for the debug implementation.

Configurations

External register EDSCR[30:29] is architecturally mapped to AArch64 System register MDCCSR_EL0[30:29].

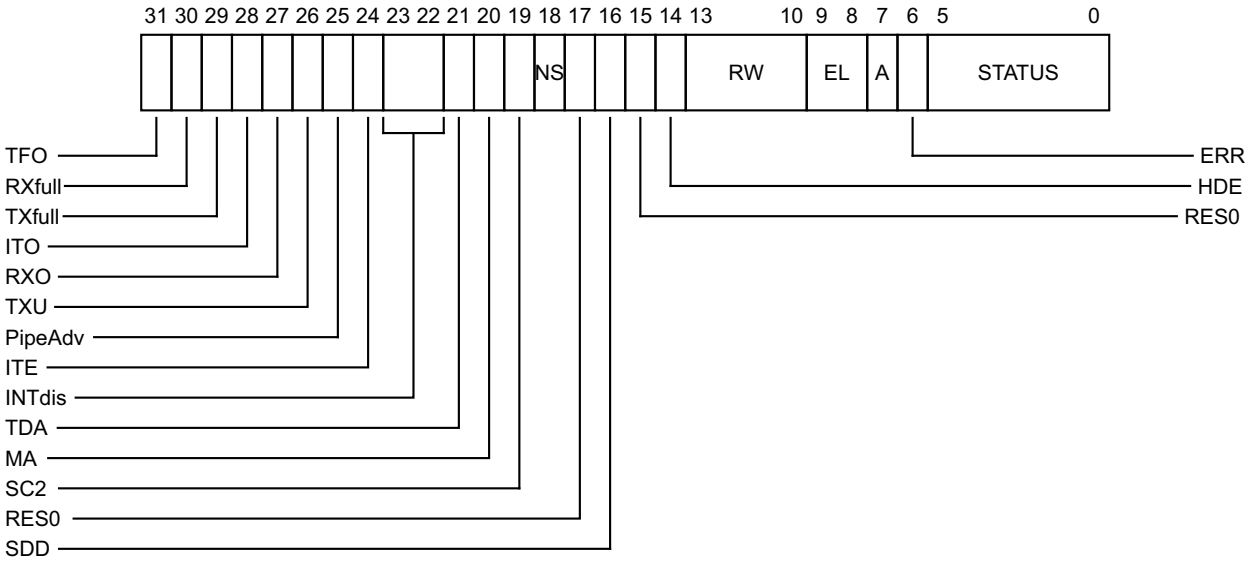
EDSCR is in the Core power domain.

Attributes

EDSCR is a 32-bit register.

Field descriptions

The EDSCR bit assignments are:



TFO, bit [31]

When ARMv8.4-Trace is implemented:

Trace Filter Override. Overrides the Trace Filter controls allowing the external debugger to trace any visible Exception level.

`0b0` Trace Filter controls are not affected.

`0b1` Trace Filter controls in TRFCR_EL1, TRFCR_EL2 are ignored.
Trace Filter controls TRFCR and HTRFCR are ignored.

When OSLSR_EL1.OSLK == 1, this bit can be indirectly read and written through the MDSCR_EL1 and DBGDSCRext System registers.

This bit is ignored by the PE when ExternalSecureNoninvasiveDebugEnabled() == FALSE and the Effective value of MDCR_EL3.STE == 1.

On a Cold reset, this field resets to `0`.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

RXfull, bit [30]

DTRRX full.

On a Cold reset, this field resets to `0`.

Access to this field is RO.

TXfull, bit [29]

DTRTX full.

On a Cold reset, this field resets to `0`.

Access to this field is RO.

ITO, bit [28]

ITR overrun.

If the PE is in Non-debug state, this bit is UNKNOWN. ITO is set to 0 on entry to Debug state.

Access to this field is RO.

RXO, bit [27]

DTRRX overrun.

On a Cold reset, this field resets to `0`.

Access to this field is RO.

TXU, bit [26]

DTRTX underrun.

On a Cold reset, this field resets to `0`.

Access to this field is RO.

PipeAdv, bit [25]

Pipeline advance. Set to 1 every time the PE pipeline retires one or more instructions. Cleared to 0 by a write to EDRCR.CSPA.

The architecture does not define precisely when this bit is set to 1. It requires only that this happen periodically in Non-debug state to indicate that software execution is progressing.

Access to this field is RO.

ITE, bit [24]

ITR empty.

If the PE is in Non-debug state, this bit is UNKNOWN. It is always valid in Debug state.

Access to this field is RO.

INTdis, bits [23:22]

When ARMv8.4-Debug is implemented:

Interrupt disable. Disables taking interrupts in Non-Debug state.

`0b0` Masking of interrupts is controlled by PSTATE and interrupt routing controls.

`0b1` If ExternalSecureDebugEnabled() == TRUE, then all interrupts, including virtual and SError interrupts, are masked.

If ExternalSecureDebugEnabled() == FALSE, then all interrupts targeting Non-secure state are masked.

When OSLSR_EL1.OSLK == 1, this field can be indirectly read and written through the MDSCR_EL1 and DBGDSCRext System registers.

This field is ignored by the PE and treated as zero when ExternalDebugEnabled() == FALSE.

On a Cold reset, this field resets to `0`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

Interrupt disable.

When OSLSR_EL1.OSLK == 1, this field can be indirectly read and written through the MDSCR_EL1 and DBGDSCRext System registers.

`0b00` Do not disable interrupts.

`0b01` Disable interrupts taken to Non-secure EL1.

`0b10` Disable interrupts taken only to Non-secure EL1 and Non-secure EL2. If ExternalSecureInvasiveDebugEnabled() == TRUE, also disable interrupts taken to Secure EL1.

`0b11` Disable interrupts taken only to Non-secure EL1 and Non-secure EL2. If ExternalSecureInvasiveDebugEnabled() == TRUE, also disable all other interrupts.

On a Cold reset, this field resets to `0`.

TDA, bit [21]

Traps accesses to the following debug System registers:

- AArch64: DBGBCR<n>_EL1, DBGBVR<n>_EL1, DBGWCR<n>_EL1, DBGWVR<n>_EL1.
- AArch32: DBGBCR<n>, DBGBVR<n>, DBGBXVR<n>, DBGWCR<n>, DBGWVR<n>.

The possible values of this field are:

`0b0` Accesses to debug System registers do not generate a Software Access Debug event.

`0b1` Accesses to debug System registers generate a Software Access Debug event, if OSLSR_EL1.OSLK is 0 and if halting is allowed.

On a Cold reset, this field resets to `0`.

MA, bit [20]

Memory access mode. Controls the use of memory-access mode for accessing ITR and the DCC. This bit is ignored if in Non-debug state and set to zero on entry to Debug state.

Possible values of this field are:

`0b0` Normal access mode.

`0b1` Memory access mode.

On a Cold reset, this field resets to `0`.

SC2, bit [19]

When ARMv8.0-PCSample is implemented, (ARMv8.1-VHE is implemented or ARMv8.2-Debug is implemented) and ARMv8.2-PCSample is not implemented:

Sample CONTEXTIDR_EL2. Controls whether the PC Sample-based Profiling Extension samples CONTEXTIDR_EL2 or VTTBR_EL2.VMID.

`0b0` Sample VTTBR_EL2.VMID.

`0b1` Sample CONTEXTIDR_EL2.

On a Cold reset, this field resets to `0`.

Otherwise:

Reserved, RES0.

NS, bit [18]

Non-secure status. When in Debug state, gives the current Security state:

`0b0` Secure state, IsSecure() == TRUE.

`0b1` Non-secure state, IsSecure() == FALSE.

In Non-debug state, this bit is UNKNOWN.

Access to this field is RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bit [17]

Reserved, RES0.

SDD, bit [16]

Secure debug disabled.

On entry to Debug state:

- If entering in Secure state, SDD is set to 0.
- If entering in Non-secure state, SDD is set to the inverse of ExternalSecureInvasiveDebugEnabled().

In Debug state, the value of the SDD bit does not change, even if ExternalSecureInvasiveDebugEnabled() changes.

In Non-debug state:

- SDD returns the inverse of ExternalSecureInvasiveDebugEnabled(). If the authentication signals that control ExternalSecureInvasiveDebugEnabled() change, a context synchronization event is required to guarantee their effect.
- This bit is unaffected by the Security state of the PE.

If EL3 is not implemented and the implementation is Non-secure, this bit is RES1.

Access to this field is RO.

Bit [15]

Reserved, RES0.

HDE, bit [14]

Halting debug enable. The possible values of this field are:

`0b0` Halting disabled for Breakpoint, Watchpoint and Halt Instruction debug events.

`0b1` Halting enabled for Breakpoint, Watchpoint and Halt Instruction debug events.

On a Cold reset, this field resets to `0`.

RW, bits [13:10]

Exception level Execution state status. In Debug state, each bit gives the current Execution state of each Exception level.

`0b1111` All Exception levels are using AArch64 or the PE is in Non-debug state.

`0b1110` **When AArch32 is supported at any Exception level** The PE is in Debug state. EL0 is using AArch32. All other Exception levels are using AArch64. Only permitted if the PE is executing at EL0.

`0b110x` **When AArch32 is supported at any Exception level** The PE is in Debug state. EL0 and EL1 are using AArch32. EL2 and EL3 are using AArch64. Only permitted if EL2 is implemented and enabled in the current Security state.

`0b10xx` **When AArch32 is supported at any Exception level and EL3 is implemented** The PE is in Debug state. EL0, EL1, and, if implemented in the current Security state, EL2 are using AArch32. EL3 is using AArch64.

`0b0xxx` The PE is in Debug state. All Exception levels are using AArch32.

In Non-debug state, this field is RAO.

Access to this field is RO.

EL, bits [9:8]

Exception level. In Debug state, this gives the current Exception level of the PE.

In Non-debug state, this field is RAZ.

Access to this field is RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A, bit [7]

SError interrupt pending. In Debug state, indicates whether an SError interrupt is pending:

- If HCR_EL2.{AMO, TGE} = {1, 0}, EL2 is enabled in the current Security state, and the PE is executing at EL0 or EL1, a virtual SError interrupt.
- Otherwise, a physical SError interrupt.

`0b0` No SError interrupt pending.

`0b1` SError interrupt pending.

A debugger can read EDSCR to check whether an SError interrupt is pending without having to execute further instructions. A pending SError might indicate data from target memory is corrupted.

UNKNOWN in Non-debug state.

Access to this field is RO.

ERR, bit [6]

Cumulative error flag. This bit is set to 1 following exceptions in Debug state and on any signaled overrun or underrun on the DTR or EDITR.

On a Cold reset, this field resets to `0`.

Access to this field is RO.

STATUS, bits [5:0]

Debug status flags.

`0b000001` PE is restarting, exiting Debug state.

`0b000010` PE is in Non-debug state.

`0b000111` Breakpoint.

`0b010011` External debug request.

`0b011011` Halting step, normal.

`0b011111` Halting step, exclusive.

`0b100011` OS Unlock Catch.

`0b100111` Reset Catch.

`0b101011` Watchpoint.

`0b101111` HLT instruction.

`0b110011` Software access to debug register.

`0b110111` Exception Catch.

`0b111011` Halting step, no syndrome.

All other values of STATUS are reserved.

Access to this field is RO.

Accessing the EDSCR:

EDSCR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x088`	EDSCR

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and SoftwareLockStatus() accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.43 EDVIDSR, External Debug Virtual Context Sample Register

The EDVIDSR characteristics are:

Purpose

Contains sampled values captured on reading EDPCSR[31:0].

Configurations

EDVIDSR is in the Core power domain.

This register is present only when ARMv8.0-PCSample is implemented and ARMv8.2-PCSample is not implemented. Otherwise, direct accesses to EDVIDSR are RES0.

Implemented only if the OPTIONAL PC Sample-based Profiling Extension is implemented in the external debug registers space.

When the PC Sample-based Profiling Extension is implemented in the external debug registers space, if EL2 is not implemented and EL3 is not implemented, it is IMPLEMENTATION DEFINED whether EDVIDSR is implemented.

Note

ARMv8.2-PCSample implements the PC Sample-based Profiling Extension in the Performance Monitors registers space.

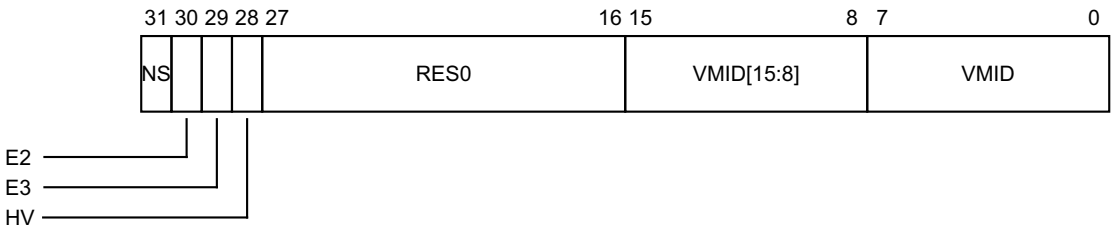
Attributes

If ARMv8.1-VHE is implemented, the format of this register differs depending on the value of EDSCR.SC2.

Field descriptions

The EDVIDSR bit assignments are:

When ARMv8.1-VHE is not implemented or EDSCR.SC2 == 0:



This format applies in all Armv8.0 implementations.

NS, bit [31]

Non-secure state sample. Indicates the Security state associated with the most recent EDPCSR sample.

If EL3 is not implemented, this bit indicates the Effective value of SCR.NS.

`0b0` Sample is from Secure state.

`0b1` Sample is from Non-secure state.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

E2, bit [30]

When EL2 is implemented:

Exception level 2 status sample. Indicates whether the most recent EDPCSR sample was associated with EL2.

`0b0` Sample is not from EL2.

`0b1` Sample is from EL2.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

E3, bit [29]

When EL3 is implemented and the highest implemented Exception level is using AArch64 state:

Exception level 3 status sample. Indicates whether the most recent EDPCSR sample was associated with EL3 using AArch64.

`0b0` Sample is not from EL3 using AArch64.

`0b1` Sample is from EL3 using AArch64.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HV, bit [28]

EDPCSRhi (EDPCSR[63:32]) valid. Indicates whether bits [63:32] of the most recent EDPCSR sample might be nonzero:

`0b0` Bits[63:32] of the most recent EDPCSR sample are zero.

`0b1` Bits[63:32] of the most recent EDPCSR sample might be nonzero.

An EDVIDSR.HV value of 1 does not mean that the value of EDPCSRhi is nonzero. An EDVIDSR.HV value of 0 is a hint that EDPCSRhi (EDPCSR[63:32]) does not need to be read.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [27:16]

Reserved, RES0.

VMID[15:8], bits [15:8]

When ARMv8.1-VMID16 is implemented and EL2 is implemented:

Extension to VMID[7:0]. See VMID[7:0] for more details.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

VMID, bits [7:0]

When EL2 is implemented:

VMID sample. The VMID associated with the most recent EDPCSRlo (EDPCSR[31:0]) sample. When the most recent EDPCSR sample was generated:

- This field is RES0 if any of the following apply:
  - The PE is executing in Secure state.
  - The PE is executing at EL2.
- Otherwise:
  - If EL2 is using AArch64 and either ARMv8.1-VMID16 is not implemented or VTCR_EL2.VS is 1, this field is set to VTTBR_EL2.VMID.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— If EL2 is using AArch64, ARMv8.1-VMID16 is implemented, and VTCR_EL2.VS is 0, PMVIDSR.VMID[7:0] is set to VTTBR_EL2.VMID[7:0] and PMVIDSR.VMID[15:8] is RES0.

— If EL2 is using AArch32, this field is set to VTTBR.VMID.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

When (ARMv8.1-VHE is implemented or ARMv8.2-Debug is implemented) and EDSCR.SC2 == 1:

31	0
CONTEXTIDR_EL2	

CONTEXTIDR_EL2, bits [31:0]

Context ID. The value of CONTEXTIDR_EL2 that is associated with the most recent EDPCSR sample. When the most recent EDPCSR sample was generated:

- If EL2 was using AArch64 and the PE was executing in Non-secure state, then this field is set to the Context ID sampled from CONTEXTIDR_EL2.
- If EL2 was using AArch32 or the PE was executing in Secure state, then this field is set to an UNKNOWN value.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing the EDVIDSR:

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register UNKNOWN, see *Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN* on page H7-7046

EDVIDSR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x0A8`	EDVIDSR

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus() and !OSLockStatus() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.44 EDWAR, External Debug Watchpoint Address Register

The EDWAR characteristics are:

Purpose

Returns the virtual data address being accessed when a Watchpoint Debug Event was triggered.

Configurations

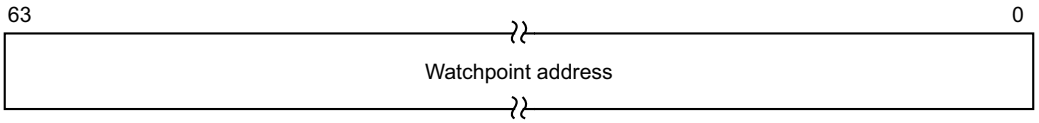
EDWAR is in the Core power domain.

Attributes

EDWAR is a 64-bit register.

Field descriptions

The EDWAR bit assignments are:



Bits [63:0]

Watchpoint address. The data virtual address being accessed when a Watchpoint Debug Event was triggered and caused entry to Debug state. This address must be within a naturally-aligned block of memory of power-of-two size no larger than the DC ZVA block size.

The value of this register is UNKNOWN if the PE is in Non-debug state, or if Debug state was entered other than for a Watchpoint debug event.

The value of EDWAR[63:32] is UNKNOWN if Debug state was entered for a Watchpoint debug event taken from AArch32 state.

The EDWAR is subject to the same alignment rules as the reporting of a watchpointed address in the FAR. See *Determining the memory location that caused a Watchpoint exception* on page D2-2465

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing the EDWAR:

EDWAR[31:0] can be accessed through the external debug interface:

Component	Offset	Instance	Range
Debug	`0x030`	EDWAR	31:0

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus() and !OSLockStatus() accesses to EDWAR[31:0] are RO.
- Otherwise accesses to EDWAR[31:0] generate an error response.

EDWAR[63:32] can be accessed through the external debug interface:

Component	Offset	Instance	Range
Debug	`0x034`	EDWAR	63:32

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus() and !OSLockStatus() accesses to EDWAR[63:32] are RO.
- Otherwise accesses to EDWAR[63:32] generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.45 MIDR_EL1, Main ID Register

The MIDR_EL1 characteristics are:

Purpose

Provides identification information for the PE, including an implementer code for the device and a device ID number.

Configurations

External register MIDR_EL1[31:0] is architecturally mapped to AArch64 System register MIDR_EL1[31:0].

External register MIDR_EL1[31:0] is architecturally mapped to AArch32 System register MIDR[31:0].

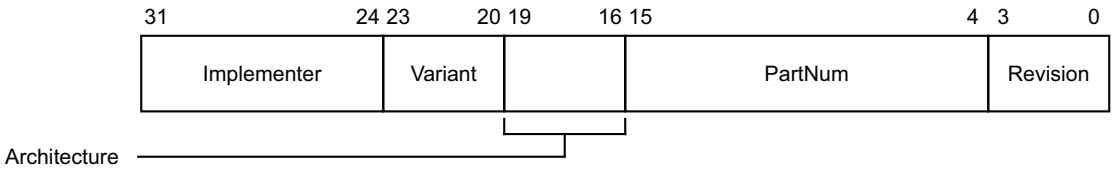
It is IMPLEMENTATION DEFINED whether MIDR_EL1 is implemented in the Core power domain or in the Debug power domain.

Attributes

MIDR_EL1 is a 32-bit register.

Field descriptions

The MIDR_EL1 bit assignments are:



Implementer, bits [31:24]

The Implementer code. This field must hold an implementer code that has been assigned by Arm. Assigned codes include the following:

Hex representation	Implementer
`0x00`	Reserved for software use
`0xC0`	Ampere Computing
`0x41`	Arm Limited
`0x42`	Broadcom Corporation
`0x43`	Cavium Inc.
`0x44`	Digital Equipment Corporation
`0x46`	Fujitsu Ltd.
`0x49`	Infineon Technologies AG
`0x4D`	Motorola or Freescale Semiconductor Inc.
`0x4E`	NVIDIA Corporation
`0x50`	Applied Micro Circuits Corporation

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Hex representation	Implementer
0x51	Qualcomm Inc.
0x56	Marvell International Ltd.
0x69	Intel Corporation

Arm can assign codes that are not published in this manual. All values not assigned by Arm are reserved and must not be used.

Variant, bits [23:20]

An IMPLEMENTATION DEFINED variant number. Typically, this field is used to distinguish between different product variants, or major revisions of a product.

Architecture, bits [19:16]

The permitted values of this field are:

0b0001 Armv4.

0b0010 Armv4T.

0b0011 Armv5 (obsolete).

0b0100 Armv5T.

0b0101 Armv5TE.

0b0110 Armv5TEJ.

0b0111 Armv6.

0b1111 Architectural features are individually identified in the ID_* registers, see Table K14-9 on page K14-8060

All other values are reserved.

PartNum, bits [15:4]

An IMPLEMENTATION DEFINED primary part number for the device.

On processors implemented by Arm, if the top four bits of the primary part number are 0x0 or 0x7, the variant and architecture are encoded differently.

Revision, bits [3:0]

An IMPLEMENTATION DEFINED revision number for the device.

Accessing the MIDR_EL1:

MIDR_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xD00	MIDR_EL1

This interface is accessible as follows:

- When IsCorePowered() and !DoubleLockStatus() accesses to this register are RO.
- Otherwise accesses to this register are IMPDEF.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.2.46 OSLAR_EL1, OS Lock Access Register

The OSLAR_EL1 characteristics are:

Purpose

Used to lock or unlock the OS lock.

Configurations

External register OSLAR_EL1[31:0] is architecturally mapped to AArch64 System register OSLAR_EL1[31:0].

External register OSLAR_EL1[31:0] is architecturally mapped to AArch32 System register DBGOSLAR[31:0].

OSLAR_EL1 is in the Core power domain.

If ARMv8.2-Debug is not implemented, it is IMPLEMENTATION DEFINED whether external debug accesses to OSLAR_EL1 are ignored and return an error when AllowExternalDebugAccess() returns FALSE for the access.

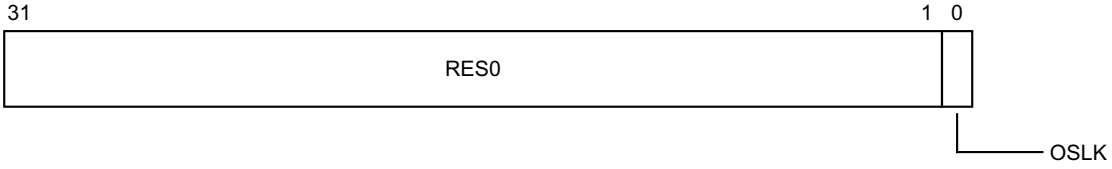
If ARMv8.2-Debug is implemented, external debug accesses to OSLAR_EL1 are ignored and return an error when AllowExternalDebugAccess() returns FALSE for the access.

Attributes

OSLAR_EL1 is a 32-bit register.

Field descriptions

The OSLAR_EL1 bit assignments are:



Bits [31:1]

Reserved, RES0.

OSLK, bit [0]

On writes to OSLAR_EL1, bit[0] is copied to the OS lock.

Use EDPRSR.OSLK to check the current status of the lock.

Accessing the OSLAR_EL1:

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalDebugAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

OSLAR_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x300`	OSLAR_EL1

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), AllowExternalDebugAccess() and SoftwareLockStatus() accesses to this register are WI.
- When IsCorePowered(), !DoubleLockStatus(), AllowExternalDebugAccess() and !SoftwareLockStatus() accesses to this register are WO.
- When IsCorePowered(), !DoubleLockStatus(), !AllowExternalDebugAccess() and ARMv8.2-Debug is not implemented accesses to this register are IMPDEF.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3 Cross-Trigger Interface registers

This section lists the Cross-Trigger Interface registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.1 ASICCTL, CTI External Multiplexer Control register

The ASICCTL characteristics are:

Purpose

Can be used to provide IMPLEMENTATION DEFINED controls for the CTI. For example, the register might be used to control multiplexors for additional IMPLEMENTATION DEFINED triggers. The IMPLEMENTATION DEFINED controls provided by this register might modify the architecturally defined behavior of the CTI.

Note

The architecturally-defined triggers must not be multiplexed.

Configurations

It is IMPLEMENTATION DEFINED whether ASICCTL is implemented in the Core power domain or in the Debug power domain.

If it is implemented in the Core power domain then it is IMPLEMENTATION DEFINED whether it is in the Cold reset domain or the Warm reset domain.

This register must reset to a value that supports the architecturally-defined behavior of the CTI. Changing the value of the register from its reset value causes IMPLEMENTATION DEFINED behavior that might differ from the architecturally-defined behavior of the CTI.

Other than the requirements listed in this register description, all aspects of the reset behavior of the ASICCTL are IMPLEMENTATION DEFINED.

Attributes

ASICCTL is a 32-bit register.

Field descriptions

The ASICCTL bit assignments are:

31 … 0
IMPLEMENTATION DEFINED

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

Accessing the ASICCTL:

ASICCTL can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x144`	ASICCTL

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalDebugAccess() and SoftwareLockStatus() accesses to this register are RO.
- Otherwise accesses to this register are IMPDEF.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.2 CTIAPPCLEAR, CTI Application Trigger Clear register

The CTIAPPCLEAR characteristics are:

Purpose

Clears bits of the Application Trigger register.

Configurations

CTIAPPCLEAR is in the Debug power domain.

Attributes

CTIAPPCLEAR is a 32-bit register.

Field descriptions

The CTIAPPCLEAR bit assignments are:

31	0
APPCLEAR<x>, bit [x]	

APPCLEAR<x>, bit [x], for x = 0 to 31

Application trigger <x> disable.

Bits [31:N] are RAZ/WI. N is the number of ECT channels implemented as defined by the CTIDEVID.NUMCHAN field.

Writing to this bit has the following effect:

`0b0` No effect.

`0b1` Clear corresponding bit in CTIAPPTRIG to 0 and clear the corresponding channel event.

If the ECT does not support multicycle channel events, use of CTIAPPCLEAR is deprecated and the debugger must only use CTIAPPPULSE.

Accessing the CTIAPPCLEAR:

CTIAPPCLEAR can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x018`	CTIAPPCLEAR

This interface is accessible as follows:

- When SoftwareLockStatus() accesses to this register are WI.
- When !SoftwareLockStatus() accesses to this register are WO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.3 CTIAPPPULSE, CTI Application Pulse register

The CTIAPPPULSE characteristics are:

Purpose

Causes event pulses to be generated on ECT channels.

Configurations

CTIAPPPULSE is in the Debug power domain.

Attributes

CTIAPPPULSE is a 32-bit register.

Field descriptions

The CTIAPPPULSE bit assignments are:

31 … 0
APPPULSE<x>, bit [x]

APPPULSE<x>, bit [x], for x = 0 to 31

Generate event pulse on ECT channel <x>.

Bits [31:N] are RAZ/WI. N is the number of ECT channels implemented as defined by the CTIDEVID.NUMCHAN field.

Writing to this bit has the following effect:

`0b0` No effect.

`0b1` Channel <x> event pulse generated.

Note

- The CTIAPPPULSE operation does not affect the state of the Application Trigger register, CTIAPPTRIG. If the channel is active, either because of an earlier event or from the application trigger, then the value written to CTIAPPPULSE might have no effect.
- Multiple pulse events that occur close together might be merged into a single pulse event.

Accessing the CTIAPPPULSE:

It is CONSTRAINED UNPREDICTABLE whether a write to CTIAPPPULSE generates an event on a channel if CTICONTROL.GLBEN is 0.

CTIAPPPULSE can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x01C`	CTIAPPPULSE

This interface is accessible as follows:

- When SoftwareLockStatus() accesses to this register are WI.
- When !SoftwareLockStatus() accesses to this register are WO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.4 CTIAPPSET, CTI Application Trigger Set register

The CTIAPPSET characteristics are:

Purpose

Sets bits of the Application Trigger register.

Configurations

CTIAPPSET is in the Debug power domain.

Attributes

CTIAPPSET is a 32-bit register.

Field descriptions

The CTIAPPSET bit assignments are:

31	0
APPSET<x>, bit [x]	

APPSET<x>, bit [x], for x = 0 to 31

Application trigger <x> enable.

Bits [31:N] are RAZ/WI. N is the number of ECT channels implemented as defined by the CTIDEVID.NUMCHAN field.

Possible values of this bit are:

`0b0` Reading this means the application trigger is inactive. Writing this has no effect.

`0b1` Reading this means the application trigger is active. Writing this sets the corresponding bit in CTIAPPTRIG to 1 and generates a channel event.

If the ECT does not support multicycle channel events, use of CTIAPPSET is deprecated and the debugger must only use CTIAPPPULSE.

On a External debug reset, this field resets to an architecturally UNKNOWN value.

Accessing the CTIAPPSET:

CTIAPPSET can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x014`	CTIAPPSET

This interface is accessible as follows:

- When SoftwareLockStatus() accesses to this register are RO.
- When !SoftwareLockStatus() accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.5 CTIAUTHSTATUS, CTI Authentication Status register

The CTIAUTHSTATUS characteristics are:

Purpose

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for CTI.

Configurations

CTIAUTHSTATUS is in the Debug power domain.

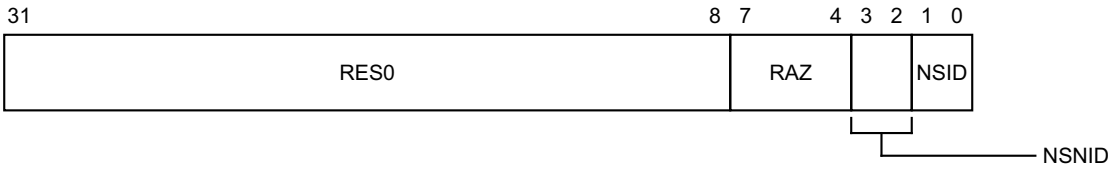
This register is OPTIONAL, and is required for CoreSight compliance.

Attributes

CTIAUTHSTATUS is a 32-bit register.

Field descriptions

The CTIAUTHSTATUS bit assignments are:



Bits [31:8]

Reserved, RES0.

Bits [7:4]

Reserved, RAZ.

NSNID, bits [3:2]

If EL3 is implemented, this field holds the same value as DBGAUTHSTATUS_EL1.NSNID.

If EL3 is not implemented and the implemented Security state is Secure state, this field holds the same value as DBGAUTHSTATUS_EL1.SNID.

NSID, bits [1:0]

If EL3 is implemented, this field holds the same value as DBGAUTHSTATUS_EL1.NSID.

If EL3 is not implemented and the implemented Security state is Secure state, this field holds the same value as DBGAUTHSTATUS_EL1.SID.

Accessing the CTIAUTHSTATUS:

CTIAUTHSTATUS can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFB8`	CTIAUTHSTATUS

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.6 CTICHINSTATUS, CTI Channel In Status register

The CTICHINSTATUS characteristics are:

Purpose

Provides the raw status of the ECT channel inputs to the CTI.

Configurations

CTICHINSTATUS is in the Debug power domain.

Attributes

CTICHINSTATUS is a 32-bit register.

Field descriptions

The CTICHINSTATUS bit assignments are:

31 … 0
CHIN<n>, bit [n]

CHIN<n>, bit [n], for n = 0 to 31

Input channel <n> status.

Bits [31:N] are RAZ. N is the number of ECT channels implemented as defined by the CTIDEVID.NUMCHAN field.

Possible values of this bit are:

`0b0` Input channel <n> is inactive.

`0b1` Input channel <n> is active.

If the ECT channels do not support multicycle events then it is IMPLEMENTATION DEFINED whether an input channel can be observed as active.

Accessing the CTICHINSTATUS:

CTICHINSTATUS can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x138`	CTICHINSTATUS

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.7 CTICHOUTSTATUS, CTI Channel Out Status register

The CTICHOUTSTATUS characteristics are:

Purpose

Provides the status of the ECT channel outputs from the CTI.

Configurations

CTICHOUTSTATUS is in the Debug power domain.

Attributes

CTICHOUTSTATUS is a 32-bit register.

Field descriptions

The CTICHOUTSTATUS bit assignments are:

31	0
CHOUT<n>, bit [n]	

CHOUT<n>, bit [n], for n = 0 to 31

Output channel <n> status.

Bits [31:N] are RAZ. N is the number of ECT channels implemented as defined by the CTIDEVID.NUMCHAN field.

Possible values of this bit are:

`0b0` Output channel <n> is inactive.

`0b1` Output channel <n> is active.

If the ECT channels do not support multicycle events then it is IMPLEMENTATION DEFINED whether an output channel can be observed as active.

Note

The value in CTICHOUTSTATUS is after gating by the channel gate. For more information, see CTIGATE.

Accessing the CTICHOUTSTATUS:

CTICHOUTSTATUS can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x13C`	CTICHOUTSTATUS

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.8 CTICIDR0, CTI Component Identification Register 0

The CTICIDR0 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information see *About the Component Identification scheme* on page K2-7892.

Configurations

CTICIDR0 is in the Debug power domain.

Implementation of this register is OPTIONAL.

This register is required for CoreSight compliance.

Attributes

CTICIDR0 is a 32-bit register.

Field descriptions

The CTICIDR0 bit assignments are:

31 … 8	7 … 0
RES0	PRMBL_0

Bits [31:8]

Reserved, RES0.

PRMBL_0, bits [7:0]

Preamble. Must read as `0x0D`.

Accessing the CTICIDR0:

CTICIDR0 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFF0`	CTICIDR0

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.9 CTICIDR1, CTI Component Identification Register 1

The CTICIDR1 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information see *About the Component Identification scheme* on page K2-7892.

Configurations

CTICIDR1 is in the Debug power domain.

Implementation of this register is OPTIONAL.

This register is required for CoreSight compliance.

Attributes

CTICIDR1 is a 32-bit register.

Field descriptions

The CTICIDR1 bit assignments are:

31 … 8	7 … 4	3 … 0
RES0	CLASS	PRMBL_1

Bits [31:8]

Reserved, RES0.

CLASS, bits [7:4]

Component class. Reads as `0x9`, debug component.

PRMBL_1, bits [3:0]

Preamble. RAZ.

Accessing the CTICIDR1:

CTICIDR1 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFF4`	CTICIDR1

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.10 CTICIDR2, CTI Component Identification Register 2

The CTICIDR2 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information see *About the Component Identification scheme* on page K2-7892.

Configurations

CTICIDR2 is in the Debug power domain.

Implementation of this register is OPTIONAL.

This register is required for CoreSight compliance.

Attributes

CTICIDR2 is a 32-bit register.

Field descriptions

The CTICIDR2 bit assignments are:

31 … 8	7 … 0
RES0	PRMBL_2

Bits [31:8]

Reserved, RES0.

PRMBL_2, bits [7:0]

Preamble. Must read as `0x05`.

Accessing the CTICIDR2:

CTICIDR2 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFF8`	CTICIDR2

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.11 CTICIDR3, CTI Component Identification Register 3

The CTICIDR3 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information see *About the Component Identification scheme* on page K2-7892.

Configurations

CTICIDR3 is in the Debug power domain.

Implementation of this register is OPTIONAL.

This register is required for CoreSight compliance.

Attributes

CTICIDR3 is a 32-bit register.

Field descriptions

The CTICIDR3 bit assignments are:

31 … 8	7 … 0
RES0	PRMBL_3

Bits [31:8]

Reserved, RES0.

PRMBL_3, bits [7:0]

Preamble. Must read as `0xB1`.

Accessing the CTICIDR3:

CTICIDR3 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFFC`	CTICIDR3

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.12 CTICLAIMCLR, CTI CLAIM Tag Clear register

The CTICLAIMCLR characteristics are:

Purpose

Used by software to read the values of the CLAIM bits, and to clear CLAIM tag bits to 0.

Configurations

CTICLAIMCLR is in the Debug power domain.

Implementation of this register is OPTIONAL.

Attributes

CTICLAIMCLR is a 32-bit register.

Field descriptions

The CTICLAIMCLR bit assignments are:

31	0
CLAIM[<x>], bit [x]	

CLAIM[<x>], bit [x], for x = 0 to 31

CLAIM tag clear bit.

For values of x greater than or equal to the IMPLEMENTATION DEFINED number of CLAIM tags, this bit is RAZ/SBZ. Software can rely on these bits reading as zero, and must use a Should-Be-Zero policy on writes. Implementations must ignore writes.

For other values of x, reads return the value of CLAIM[x] and the behavior on writes is:

`0b0` No action.

`0b1` Indirectly clear CLAIM[x] to 0.

A single write to CTICLAIMCLR can clear multiple tags to 0.

An External Debug reset clears the CLAIM tag bits to 0.

Accessing the CTICLAIMCLR:

CTICLAIMCLR can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFA4`	CTICLAIMCLR

This interface is accessible as follows:

- When SoftwareLockStatus() accesses to this register are RO.
- When !SoftwareLockStatus() accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.13 CTICLAIMSET, CTI CLAIM Tag Set register

The CTICLAIMSET characteristics are:

Purpose

Used by software to set CLAIM bits to 1.

Configurations

CTICLAIMSET is in the Debug power domain.

Implementation of this register is OPTIONAL.

Attributes

CTICLAIMSET is a 32-bit register.

Field descriptions

The CTICLAIMSET bit assignments are:

31 … 0
CLAIM[<x>], bit [x]

CLAIM[<x>], bit [x], for x = 0 to 31

CLAIM tag set bit.

For values of x greater than or equal to the IMPLEMENTATION DEFINED number of CLAIM tags, this bit is RAZ/SBZ. Software can rely on these bits reading as zero, and must use a Should-Be-Zero policy on writes. Implementations must ignore writes.

For other values of x, the bit is RAO and the behavior on writes is:

`0b0` No action.

`0b1` Indirectly set CLAIM[x] tag to 1.

A single write to CTICLAIMSET can set multiple tags to 1.

An External Debug reset clears the CLAIM tag bits to 0.

Accessing the CTICLAIMSET:

CTICLAIMSET can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFA0`	CTICLAIMSET

This interface is accessible as follows:

- When SoftwareLockStatus() accesses to this register are RO.
- When !SoftwareLockStatus() accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.14 CTICONTROL, CTI Control register

The CTICONTROL characteristics are:

Purpose

Controls whether the CTI is enabled.

Configurations

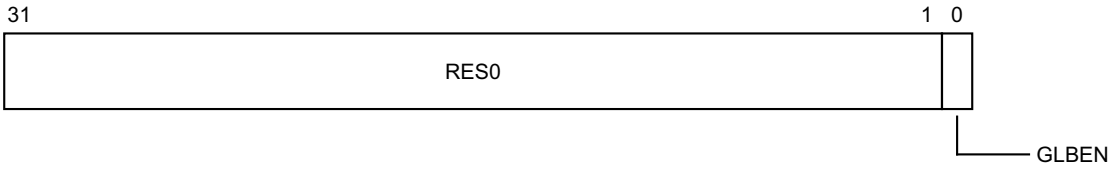
CTICONTROL is in the Debug power domain.

Attributes

CTICONTROL is a 32-bit register.

Field descriptions

The CTICONTROL bit assignments are:



Bits [31:1]

Reserved, RES0.

GLBEN, bit [0]

Enables or disables the CTI mapping functions. Possible values of this field are:

`0b0` CTI mapping functions and application trigger disabled.

`0b1` CTI mapping functions and application trigger enabled.

When GLBEN is 0, the input channel to output trigger, input trigger to output channel, and application trigger functions are disabled and do not signal new events on either output triggers or output channels. If a previously asserted output trigger has not been acknowledged, it remains asserted after the mapping functions are disabled. All output triggers are disabled by CTI reset.

If the ECT supports multicycle channel events any existing output channel events will be terminated.

The following resets apply:

- On a Cold reset, the value of this field is unchanged.
- On a External debug reset, this field resets to `0`.
- On a Warm reset, the value of this field is unchanged.

Accessing the CTICONTROL:

CTICONTROL can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x000`	CTICONTROL

This interface is accessible as follows:

- When SoftwareLockStatus() accesses to this register are RO.
- When !SoftwareLockStatus() accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.15 CTIDEVAFF0, CTI Device Affinity register 0

The CTIDEVAFF0 characteristics are:

Purpose

Copy of the low half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the CTI component relates to.

If the CTI is CTIv1, this register is OPTIONAL. If the CTI is CTIv2, this register is mandatory.

Arm recommends that the CTI is CTIv2.

In an Armv8.5 compliant implementation the CTI must be CTIv2.

If this register is implemented, then CTIDEVAFF1 must also be implemented. If the CTI of a PE does not implement the CTI Device Affinity registers, the CTI block of the external debug memory map must be located 64KB above the debug registers in the external debug interface.

Configurations

CTIDEVAFF0 is in the Debug power domain.

Implementation of this register is OPTIONAL.

Attributes

CTIDEVAFF0 is a 32-bit register.

Field descriptions

The CTIDEVAFF0 bit assignments are:

31	0
MPIDR_EL1lo	

MPIDR_EL1lo, bits [31:0]

MPIDR_EL1 low half. Read-only copy of the low half of MPIDR_EL1, as seen from the highest implemented Exception level.

Accessing the CTIDEVAFF0:

CTIDEVAFF0 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFA8`	CTIDEVAFF0

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.16 CTIDEVAFF1, CTI Device Affinity register 1

The CTIDEVAFF1 characteristics are:

Purpose

Copy of the high half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the CTI component relates to.

If the CTI is CTIv1, this register is OPTIONAL. If the CTI is CTIv2, this register is mandatory.

Arm recommends that the CTI is CTIv2.

In an Armv8.5 compliant implementation the CTI must be CTIv2.

If this register is implemented, then CTIDEVAFF0 must also be implemented. If the CTI of a PE does not implement the CTI Device Affinity registers, the CTI block of the external debug memory map must be located 64KB above the debug registers in the external debug interface.

Configurations

CTIDEVAFF1 is in the Debug power domain.

Implementation of this register is OPTIONAL.

Attributes

CTIDEVAFF1 is a 32-bit register.

Field descriptions

The CTIDEVAFF1 bit assignments are:

31	0
MPIDR_EL1hi	

MPIDR_EL1hi, bits [31:0]

MPIDR_EL1 high half. Read-only copy of the high half of MPIDR_EL1, as seen from the highest implemented Exception level.

Accessing the CTIDEVAFF1:

CTIDEVAFF1 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFAC`	CTIDEVAFF1

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.17 CTIDEVARCH, CTI Device Architecture register

The CTIDEVARCH characteristics are:

Purpose

Identifies the programmers' model architecture of the CTI component.

If the CTI is CTIv1, this register is OPTIONAL. If the CTI is CTIv2, this register is mandatory.

Arm recommends that the CTI is CTIv2.

In an Armv8.5 compliant implementation the CTI must be CTIv2.

If this register is not implemented, CTIDEVAFF0 and CTIDEVAFF1 are also not implemented.

Configurations

CTIDEVARCH is in the Debug power domain.

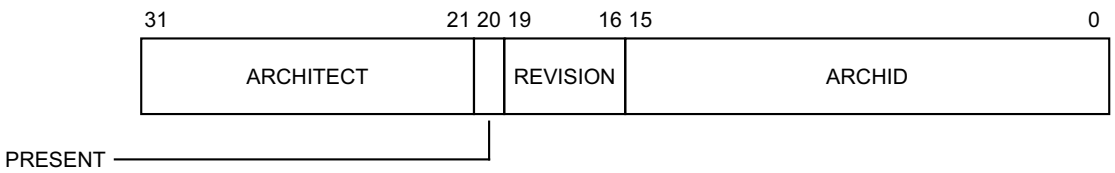
Implementation of this register is OPTIONAL.

Attributes

CTIDEVARCH is a 32-bit register.

Field descriptions

The CTIDEVARCH bit assignments are:



ARCHITECT, bits [31:21]

Defines the architecture of the component. For CTI, this is Arm Limited.

Bits [31:28] are the JEP106 continuation code, `0x4`.

Bits [27:21] are the JEP106 ID code, `0x3B`.

PRESENT, bit [20]

When set to 1, indicates that the DEVARCH is present.

This field is 1 in Armv8.

REVISION, bits [19:16]

Revision.

Defines the architecture revision of the component.

`0b0000` First revision.

`0b0001` **When ARMv8.3-DoPD is implemented** As `0b0000`, and also adds support for CTIDEVCTL.

All other values are reserved.

ARCHID, bits [15:0]

Defines this part to be an Armv8 debug component. For architectures defined by Arm this is further subdivided.

For CTI:

- Bits [15:12] are the architecture version, `0x1`.
- Bits [11:0] are the architecture part number, `0xA14`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This corresponds to CTI architecture version CTIv2.

Accessing the CTIDEVARCH:

CTIDEVARCH can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFBC`	CTIDEVARCH

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.18 CTIDEVCTL, CTI Device Control register

The CTIDEVCTL characteristics are:

Purpose

Provides target-specific device controls

Configurations

CTIDEVCTL is in the Debug power domain.

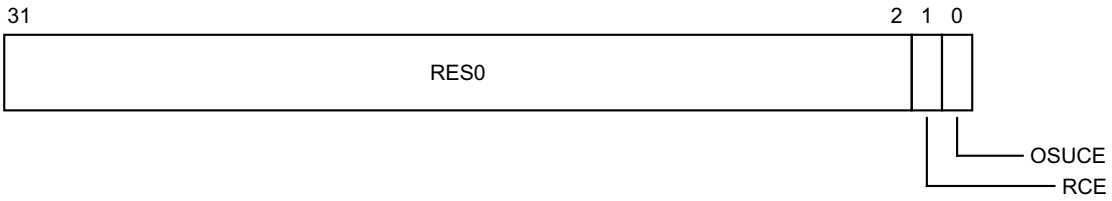
This register is present only when ARMv8.3-DoPD is implemented. Otherwise, direct accesses to CTIDEVCTL are RES0.

Attributes

CTIDEVCTL is a 32-bit register.

Field descriptions

The CTIDEVCTL bit assignments are:



Bits [31:2]

Reserved, RES0.

RCE, bit [1]

Reset Catch Enable.

`0b0` Reset Catch debug event disabled.

`0b1` Reset Catch debug event enabled.

The following resets apply:

- On a Cold reset, the value of this field is unchanged.
- On a External debug reset, this field resets to `0`.
- On a Warm reset, the value of this field is unchanged.

OSUCE, bit [0]

OS Unlock Catch Enable

`0b0` OS Unlock Catch debug event disabled.

`0b1` OS Unlock Catch debug event enabled.

The following resets apply:

- On a Cold reset, the value of this field is unchanged.
- On a External debug reset, this field resets to `0`.
- On a Warm reset, the value of this field is unchanged.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the CTIDEVCTL:

CTIDEVCTL can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x150`	CTIDEVCTL

This interface is accessible as follows:

- When SoftwareLockStatus() accesses to this register are RO.
- When !SoftwareLockStatus() accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.19 CTIDEVID, CTI Device ID register 0

The CTIDEVID characteristics are:

Purpose

Describes the CTI component to the debugger.

Configurations

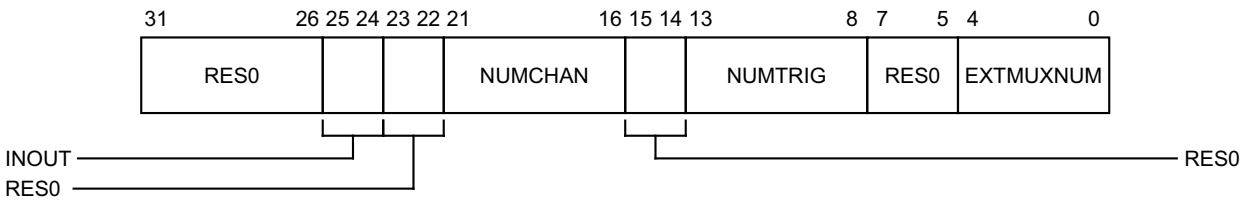
CTIDEVID is in the Debug power domain.

Attributes

CTIDEVID is a 32-bit register.

Field descriptions

The CTIDEVID bit assignments are:



Bits [31:26]

Reserved, RES0.

INOUT, bits [25:24]

Input/output options. Indicates presence of the input gate. If the CTM is not implemented or CTIv2 is not implemented, this field is RAZ.

`0b00` CTIGATE does not mask propagation of input events from external channels.

`0b01` CTIGATE masks propagation of input events from external channels.

All other values are reserved.

Bits [23:22]

Reserved, RES0.

NUMCHAN, bits [21:16]

Number of ECT channels implemented. IMPLEMENTATION DEFINED. For Armv8, valid values are:

`0b000011` 3 channels (0..2) implemented.

`0b000100` 4 channels (0..3) implemented.

`0b000101` 5 channels (0..4) implemented.

`0b000110` 6 channels (0..5) implemented.

and so on up to `0b100000`, 32 channels (0..31) implemented.

All other values are reserved.

Bits [15:14]

Reserved, RES0.

NUMTRIG, bits [13:8]

Number of triggers implemented. IMPLEMENTATION DEFINED. This is one more than the index of the largest trigger, rather than the actual number of triggers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For Armv8, valid values are:

`0b000011` Up to 3 triggers (0..2) implemented.

`0b001000` Up to 8 triggers (0..7) implemented.

`0b001001` Up to 9 triggers (0..8) implemented.

`0b001010` Up to 10 triggers (0..9) implemented.

and so on up to `0b100000`, 32 triggers (0..31) implemented.

All other values are reserved. If the PE contains a Trace extension, this field must be at least `0b001000`. There is no guarantee that any of the implemented triggers, including the highest numbered, are connected to any components.

Bits [7:5]

Reserved, RES0.

EXTMUXNUM, bits [4:0]

Number of multiplexors available on triggers. This value is used in conjunction with External Control register, ASICCTL.

Accessing the CTIDEVID:

CTIDEVID can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFC8`	CTIDEVID

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.20 CTIDEVID1, CTI Device ID register 1

The CTIDEVID1 characteristics are:

Purpose

Reserved for future information about the CTI component to the debugger.

Configurations

CTIDEVID1 is in the Debug power domain.

Attributes

CTIDEVID1 is a 32-bit register.

Field descriptions

The CTIDEVID1 bit assignments are:

31	0
RES0	

Bits [31:0]

Reserved, RES0.

Accessing the CTIDEVID1:

CTIDEVID1 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFC4`	CTIDEVID1

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.21 CTIDEVID2, CTI Device ID register 2

The CTIDEVID2 characteristics are:

Purpose

Reserved for future information about the CTI component to the debugger.

Configurations

CTIDEVID2 is in the Debug power domain.

Attributes

CTIDEVID2 is a 32-bit register.

Field descriptions

The CTIDEVID2 bit assignments are:

31	0
RES0	

Bits [31:0]

Reserved, RES0.

Accessing the CTIDEVID2:

CTIDEVID2 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFC0`	CTIDEVID2

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.22 CTIDEVTYPE, CTI Device Type register

The CTIDEVTYPE characteristics are:

Purpose

Indicates to a debugger that this component is part of a PEs cross-trigger interface.

Configurations

CTIDEVTYPE is in the Debug power domain.

Implementation of this register is OPTIONAL.

Attributes

CTIDEVTYPE is a 32-bit register.

Field descriptions

The CTIDEVTYPE bit assignments are:

31 … 8	7 … 4	3 … 0
RES0	SUB	MAJOR

Bits [31:8]

Reserved, RES0.

SUB, bits [7:4]

Subtype. Must read as `0x1` to indicate this is a component within a PE.

MAJOR, bits [3:0]

Major type. Must read as `0x4` to indicate this is a cross-trigger component.

Accessing the CTIDEVTYPE:

CTIDEVTYPE can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFCC`	CTIDEVTYPE

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.23 CTIGATE, CTI Channel Gate Enable register

The CTIGATE characteristics are:

Purpose

Determines whether events on channels propagate through the CTM to other ECT components, or from the CTM into the CTI.

Configurations

CTIGATE is in the Debug power domain.

Attributes

CTIGATE is a 32-bit register.

Field descriptions

The CTIGATE bit assignments are:

31	0
GATE<x>, bit [x]	

GATE<x>, bit [x], for x = 0 to 31

Channel <x> gate enable.

Bits [31:N] are RAZ/WI. N is the number of ECT channels implemented as defined by the CTIDEVID.NUMCHAN field.

Possible values of this bit are:

`0b0` Disable output and, if CTIDEVID.INOUT == `0b01`, input channel <x> propagation.

`0b1` Enable output and, if CTIDEVID.INOUT == `0b01`, input channel <x> propagation.

If GATE[x] is set to 0, no new events will be propagated to the ECT, and if the ECT supports multicycle channel events any existing output channel events will be terminated.

On a External debug reset, this field resets to an architecturally UNKNOWN value.

Accessing the CTIGATE:

CTIGATE can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x140`	CTIGATE

This interface is accessible as follows:

- When SoftwareLockStatus() accesses to this register are RO.
- When !SoftwareLockStatus() accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.24 CTIINEN<n>, CTI Input Trigger to Output Channel Enable registers, n = 0 - 31

The CTIINEN<n> characteristics are:

Purpose

Enables the signaling of an event on output channels when input trigger event n is received by the CTI.

Configurations

CTIINEN<n> is in the Debug power domain.

If input trigger n is not connected, the behavior of CTIINEN<n> is IMPLEMENTATION DEFINED.

Attributes

CTIINEN<n> is a 32-bit register.

Field descriptions

The CTIINEN<n> bit assignments are:

31	0
INEN<x>, bit [x]	

INEN<x>, bit [x], for x = 0 to 31

Input trigger <n> to output channel <x> enable.

Bits [31:N] are RAZ/WI. N is the number of ECT channels implemented as defined by the CTIDEVID.NUMCHAN field.

Possible values of this bit are:

`0b0` Input trigger <n> will not generate an event on output channel <x>.

`0b1` Input trigger <n> will generate an event on output channel <x>.

On a External debug reset, this field resets to an architecturally UNKNOWN value.

Accessing the CTIINEN<n>:

CTIINEN<n> can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x020` + 4n	CTIINEN<n>

This interface is accessible as follows:

- When SoftwareLockStatus() accesses to this register are RO.
- When !SoftwareLockStatus() accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.25 CTIINTACK, CTI Output Trigger Acknowledge register

The CTIINTACK characteristics are:

Purpose

Can be used to deactivate the output triggers.

Configurations

CTIINTACK is in the Debug power domain.

Attributes

CTIINTACK is a 32-bit register.

Field descriptions

The CTIINTACK bit assignments are:

31	0
ACK<n>, bit [n]	

ACK<n>, bit [n], for n = 0 to 31

Acknowledge for output trigger <n>.

Bits [31:N] are RAZ/WI. N is the number of CTI triggers implemented as defined by the CTIDEVID.NUMTRIG field.

If any of the following is true, writes to ACK<n> are ignored:

- n >= CTIDEVID.NUMTRIG, the number of implemented triggers.
- Output trigger n is not active.
- The channel mapping function output, as controlled by CTIOUTEN<n>, is still active.

Otherwise, if any of the following are true, it is IMPLEMENTATION DEFINED whether writes to ACK<n> are ignored:

- Output trigger n is not implemented.
- Output trigger n is not connected.
- Output trigger n is self-acknowledging and does not require software acknowledge.

Otherwise, the behavior on writes to ACK<n> is as follows:

`0b0` No effect

`0b1` Deactivate the trigger.

Accessing the CTIINTACK:

A debugger must read CTITRIGOUTSTATUS to confirm that the output trigger has been acknowledged before generating any event that must be ordered after the write to CTIINTACK, such as a write to CTIAPPPULSE to activate another trigger.

CTIINTACK can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x010`	CTIINTACK

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This interface is accessible as follows:

- When SoftwareLockStatus() accesses to this register are WI.
- When !SoftwareLockStatus() accesses to this register are WO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.26 CTIITCTRL, CTI Integration mode Control register

The CTIITCTRL characteristics are:

Purpose

Enables the CTI to switch from its default mode into integration mode, where test software can control directly the inputs and outputs of the PE, for integration testing or topology detection.

Configurations

It is IMPLEMENTATION DEFINED whether CTIITCTRL is implemented in the Core power domain or in the Debug power domain.

Implementation of this register is OPTIONAL.

Some or all RW fields of this register have defined reset values, and:

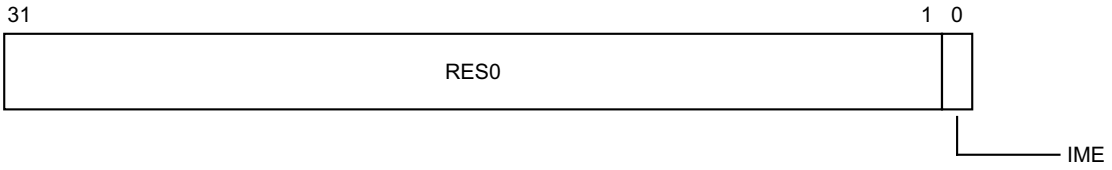
- The register is not affected by a Warm reset.
- If the register is implemented in the Core power domain, the reset values apply on a Cold reset, and the register is not affected by an External debug reset.
- If the register is implemented in the Debug power domain, the reset values apply on an External debug reset, and the register is not affected by a Cold reset.

Attributes

CTIITCTRL is a 32-bit register.

Field descriptions

The CTIITCTRL bit assignments are:



Bits [31:1]

Reserved, RES0.

IME, bit [0]

Integration mode enable. When IME == 1, the device reverts to an integration mode to enable integration testing or topology detection. The integration mode behavior is IMPLEMENTATION DEFINED.

`0b0` Normal operation.

`0b1` Integration mode enabled.

On a Implementation reset, this field resets to `0`.

Accessing the CTIITCTRL:

CTIITCTRL can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xF00`	CTIITCTRL

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register are IMPDEF.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.27 CTILAR, CTI Lock Access Register

The CTILAR characteristics are:

Purpose

Allows or disallows access to the CTI registers through a memory-mapped interface.

The optional Software Lock provides a lock to prevent memory-mapped writes to the Cross-Trigger Interface registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the Cross-Trigger Interface registers. It does not, and cannot, prevent all accidental or malicious damage.

Configurations

CTILAR is in the Debug power domain.

If ARMv8.4-Debug is implemented, the Software Lock is not implemented.

Software uses CTILAR to set or clear the lock, and CTILSR to check the current status of the lock.

Attributes

CTILAR is a 32-bit register.

Field descriptions

The CTILAR bit assignments are:

When the Software Lock is implemented.:

31	0
KEY	

KEY, bits [31:0]

Lock Access control. Writing the key value `0xC5ACCE55` to this field unlocks the lock, enabling write accesses to this component's registers through a memory-mapped interface.

Writing any other value to this register locks the lock, disabling write accesses to this component's registers through a memory mapped interface.

Otherwise:

31	0
RES0	

Otherwise

Bits [31:0]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the CTILAR:

CTILAR can be accessed through a memory-mapped interface access to the external debug interface:

Component	Offset	Instance
CTI	`0xFB0`	CTILAR

This interface is accessible as follows:

- Accesses to this register are WO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.28 CTILSR, CTI Lock Status Register

The CTILSR characteristics are:

Purpose

Indicates the current status of the Software Lock for CTI registers.

The optional Software Lock provides a lock to prevent memory-mapped writes to the Cross-Trigger Interface registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the Cross-Trigger Interface registers. It does not, and cannot, prevent all accidental or malicious damage.

Configurations

CTILSR is in the Debug power domain.

If ARMv8.4-Debug is implemented, the Software Lock is not implemented.

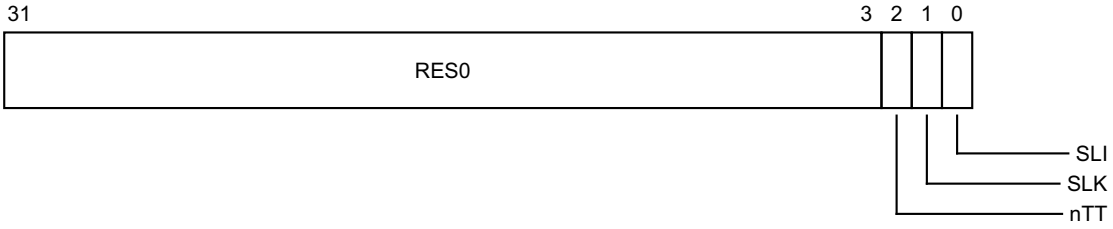
Software uses CTILAR to set or clear the lock, and CTILSR to check the current status of the lock.

Attributes

CTILSR is a 32-bit register.

Field descriptions

The CTILSR bit assignments are:



Bits [31:3]

Reserved, RES0.

nTT, bit [2]

Not thirty-two bit access required. RAZ.

SLK, bit [1]

When the Software Lock is implemented.:

Software Lock status for this component. For an access to LSR that is not a memory-mapped access, or when the Software Lock is not implemented, this field is RES0.

For memory-mapped accesses when the Software Lock is implemented, possible values of this field are:

`0b0` Lock clear. Writes are permitted to this component's registers.

`0b1` Lock set. Writes to this component's registers are ignored, and reads have no side effects.

On a External debug reset, this field resets to `1`.

Otherwise:

Reserved, RAZ.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

SLI, bit [0]

Software Lock implemented. For an access to LSR that is not a memory-mapped access, this field is RAZ. For memory-mapped accesses, the value of this field is IMPLEMENTATION DEFINED. Permitted values are:

`0b0` Software Lock not implemented or not memory-mapped access.

`0b1` Software Lock implemented and memory-mapped access.

Accessing the CTILSR:

CTILSR can be accessed through a memory-mapped interface access to the external debug interface:

Component	Offset	Instance
CTI	`0xFB4`	CTILSR

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.29 CTIOUTEN<n>, CTI Input Channel to Output Trigger Enable registers, n = 0 - 31

The CTIOUTEN<n> characteristics are:

Purpose

Defines which input channels generate output trigger n.

Configurations

CTIOUTEN<n> is in the Debug power domain.

If output trigger n is not connected, the behavior of CTIOUTEN<n> is IMPLEMENTATION DEFINED.

Attributes

CTIOUTEN<n> is a 32-bit register.

Field descriptions

The CTIOUTEN<n> bit assignments are:

31	0
OUTEN<x>, bit [x]	

OUTEN<x>, bit [x], for x = 0 to 31

Input channel <x> to output trigger <n> enable.

Bits [31:N] are RAZ/WI. N is the number of ECT channels implemented as defined by the CTIDEVID.NUMCHAN field.

Possible values of this bit are:

`0b0` An event on input channel <x> will not cause output trigger <n> to be asserted.

`0b1` An event on input channel <x> will cause output trigger <n> to be asserted.

On a External debug reset, this field resets to an architecturally UNKNOWN value.

Accessing the CTIOUTEN<n>:

CTIOUTEN<n> can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x0A0` + 4n	CTIOUTEN<n>

This interface is accessible as follows:

- When SoftwareLockStatus() accesses to this register are RO.
- When !SoftwareLockStatus() accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.30 CTIPIDR0, CTI Peripheral Identification Register 0

The CTIPIDR0 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information see *About the Peripheral identification scheme* on page K2-7889.

Configurations

CTIPIDR0 is in the Debug power domain.

Implementation of this register is OPTIONAL.

This register is required for CoreSight compliance.

Attributes

CTIPIDR0 is a 32-bit register.

Field descriptions

The CTIPIDR0 bit assignments are:

31 … 8	7 … 0
RES0	PART_0

Bits [31:8]

Reserved, RES0.

PART_0, bits [7:0]

Part number, least significant byte.

Accessing the CTIPIDR0:

CTIPIDR0 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFE0`	CTIPIDR0

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.31 CTIPIDR1, CTI Peripheral Identification Register 1

The CTIPIDR1 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information see *About the Peripheral identification scheme* on page K2-7889.

Configurations

CTIPIDR1 is in the Debug power domain.

Implementation of this register is OPTIONAL.

This register is required for CoreSight compliance.

Attributes

CTIPIDR1 is a 32-bit register.

Field descriptions

The CTIPIDR1 bit assignments are:

31 … 8	7 … 4	3 … 0
RES0	DES_0	PART_1

Bits [31:8]

Reserved, RES0.

DES_0, bits [7:4]

Designer, least significant nibble of JEP106 ID code. For Arm Limited, this field is `0b1011`.

PART_1, bits [3:0]

Part number, most significant nibble.

Accessing the CTIPIDR1:

CTIPIDR1 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFE4`	CTIPIDR1

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.32 CTIPIDR2, CTI Peripheral Identification Register 2

The CTIPIDR2 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information see *About the Peripheral identification scheme* on page K2-7889.

Configurations

CTIPIDR2 is in the Debug power domain.

Implementation of this register is OPTIONAL.

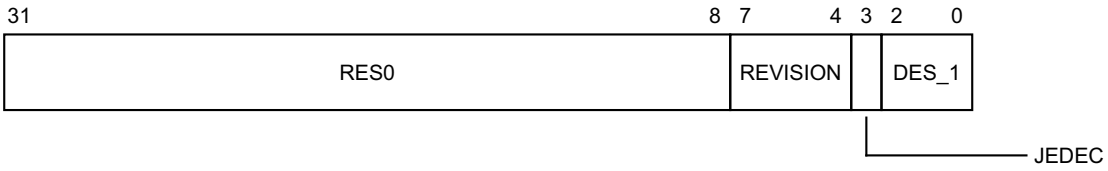
This register is required for CoreSight compliance.

Attributes

CTIPIDR2 is a 32-bit register.

Field descriptions

The CTIPIDR2 bit assignments are:



Bits [31:8]

Reserved, RES0.

REVISION, bits [7:4]

Part major revision. Parts can also use this field to extend Part number to 16-bits.

JEDEC, bit [3]

RAO. Indicates a JEP106 identity code is used.

DES_1, bits [2:0]

Designer, most significant bits of JEP106 ID code. For Arm Limited, this field is `0b011`.

Accessing the CTIPIDR2:

CTIPIDR2 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFE8`	CTIPIDR2

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.33 CTIPIDR3, CTI Peripheral Identification Register 3

The CTIPIDR3 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information see *About the Peripheral identification scheme* on page K2-7889.

Configurations

CTIPIDR3 is in the Debug power domain.

Implementation of this register is OPTIONAL.

This register is required for CoreSight compliance.

Attributes

CTIPIDR3 is a 32-bit register.

Field descriptions

The CTIPIDR3 bit assignments are:

31 … 8	7 … 4	3 … 0
RES0	REVAND	CMOD

Bits [31:8]

Reserved, RES0.

REVAND, bits [7:4]

Part minor revision. Parts using CTIPIDR2.REVISION as an extension to the Part number must use this field as a major revision number.

CMOD, bits [3:0]

Customer modified. Indicates someone other than the Designer has modified the component.

Accessing the CTIPIDR3:

CTIPIDR3 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFEC`	CTIPIDR3

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.34 CTIPIDR4, CTI Peripheral Identification Register 4

The CTIPIDR4 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information see *About the Peripheral identification scheme* on page K2-7889.

Configurations

CTIPIDR4 is in the Debug power domain.

Implementation of this register is OPTIONAL.

This register is required for CoreSight compliance.

Attributes

CTIPIDR4 is a 32-bit register.

Field descriptions

The CTIPIDR4 bit assignments are:

31 … 8	7 … 4	3 … 0
RES0	SIZE	DES_2

Bits [31:8]

Reserved, RES0.

SIZE, bits [7:4]

Size of the component. RAZ. Log_2 of the number of 4KB pages from the start of the component to the end of the component ID registers.

DES_2, bits [3:0]

Designer, JEP106 continuation code, least significant nibble. For Arm Limited, this field is `0b0100`.

Accessing the CTIPIDR4:

CTIPIDR4 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFD0`	CTIPIDR4

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.35 CTITRIGINSTATUS, CTI Trigger In Status register

The CTITRIGINSTATUS characteristics are:

Purpose

Provides the status of the trigger inputs.

Configurations

CTITRIGINSTATUS is in the Debug power domain.

Attributes

CTITRIGINSTATUS is a 32-bit register.

Field descriptions

The CTITRIGINSTATUS bit assignments are:

31	0
TRIN<n>, bit [n]	

TRIN<n>, bit [n], for n = 0 to 31

Trigger input <n> status.

Bits [31:N] are RAZ. N is the number of CTI triggers implemented as defined by the CTIDEVID.NUMTRIG field.

Possible values of this bit are:

`0b0` Input trigger n is inactive.

`0b1` Input trigger n is active.

Not implemented and not-connected input triggers are always inactive.

It is IMPLEMENTATION DEFINED whether an input trigger that does not support multicycle events can be observed as active.

Accessing the CTITRIGINSTATUS:

CTITRIGINSTATUS can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x130`	CTITRIGINSTATUS

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

H9.3.36 CTITRIGOUTSTATUS, CTI Trigger Out Status register

The CTITRIGOUTSTATUS characteristics are:

Purpose

Provides the raw status of the trigger outputs, after processing by any IMPLEMENTATION DEFINED trigger interface logic. For output triggers that are self-acknowledging, this is only meaningful if the CTI implements multicycle channel events.

Configurations

CTITRIGOUTSTATUS is in the Debug power domain.

Attributes

CTITRIGOUTSTATUS is a 32-bit register.

Field descriptions

The CTITRIGOUTSTATUS bit assignments are:

31	0
TROUT<n>, bit [n]	

TROUT<n>, bit [n], for n = 0 to 31

Trigger output <n> status.

Bits [31:N] are RAZ. N is the value in CTIDEVID.NUMTRIG.

If n < N, and output trigger <n> is implemented and connected, and either the trigger is not self-acknowledging or the CTI implements multicycle channel events, then permitted values for TROUT<n> are:

`0b0` Output trigger n is inactive.

`0b1` Output trigger n is active.

Otherwise when n < N it is IMPLEMENTATION DEFINED whether TROUT<n> behaves as described here or is RAZ.

Accessing the CTITRIGOUTSTATUS:

CTITRIGOUTSTATUS can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x134`	CTITRIGOUTSTATUS

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Part I

Memory-mapped Components of the Armv8 Architecture

Chapter I1
Requirements for Memory-mapped Components

This chapter provides some additional information about memory-mapped components. It contains the following sections:

- *Supported access sizes* on page I1-7224.
- *Synchronization of memory-mapped registers* on page I1-7226.
- *Access requirements for reserved and unallocated registers* on page I1-7228.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I1.1 Supported access sizes

The information in this section applies to all accesses to memory-mapped components of the Armv8 architecture, unless a register or component description explicitly states otherwise.

The memory access sizes that are supported by any peripheral are IMPLEMENTATION DEFINED by the peripheral.

When `HaveSecureExtDebugView`() == TRUE, each debug component has a Secure and Non-secure view. The Secure view of a debug component is mapped into Secure physical memory and the Non-secure view of a debug component is mapped into Non-secure memory. Apart from access conditions, the Non-secure and Secure views of the debug components are identical.

An implementation of a memory-mapped component that is compatible with the Armv8 architecture must support the following:

- Word-aligned 32-bit accesses to access 32-bit registers.
- If any PE in the system implements AArch32, word-aligned 32-bit accesses to either half of a 64-bit register that is mapped to a doubleword-aligned pair of adjacent 32-bit locations.

  Note

  Some memory-mapped components of the Armv8 architecture require support for word-aligned 32-bit accesses to either half of a 64-bit memory mapped register regardless of whether any PE in the system implements AArch32. These include:

  — The memory-mapped interface to the external debug and CTI registers that are described in Chapter H9 *External Debug Register Descriptions*.

  — The memory-mapped interfaces to the Generic Timer registers that are described in Chapter I2 *System Level Implementation of the Generic Timer*.

  — The memory-mapped interfaces to the Performance Monitors registers that are described in Chapter I3 *Recommended External Interface to the Performance Monitors*.

- Doubleword-aligned 64-bit accesses to access 64-bit registers that are mapped to a doubleword-aligned pair of adjacent 32-bit locations.

All registers are only single-copy atomic at word granularity. This means that for 64-bit accesses to a 64-bit register, the system might generate a pair of 32-bit accesses. The order in which the two halves are accessed is not specified.

The following accesses are not supported:

- Byte accesses.
- Halfword accesses.
- Unaligned word accesses. These accesses are not word single-copy atomic.
- Unaligned doubleword accesses. These accesses are not doubleword single-copy atomic.
- Doubleword accesses to a pair of 32-bit locations that are not a doubleword-aligned pair that forms a 64-bit register.
- Quadword accesses or higher accesses.
- Exclusive accesses.

For unsupported accesses, it is CONSTRAINED UNPREDICTABLE whether:

- The access generates an External abort or not.
- The defined side-effects of a read occur or not. A read returns UNKNOWN values.
- A write is ignored or sets the accessed register or registers to UNKNOWN.
- The access generates a fault handling interrupt or not. A read returns UNKNOWN data.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For memory-mapped accesses from a PE that complies with an Arm architecture, the single-copy atomicity rules for the instruction, the type of instruction, and the type of memory that is accessed, determine the size of the access that is made by an instruction. Example I1-1 shows this.

Example I1-1 Access sizes for memory-mapped accesses

Two Load Doubleword instructions that are made to consecutive doubleword-aligned locations generate a pair of single-copy atomic doubleword reads. However, if the accesses are made to Normal memory or Device-GRE memory they might appear as a single quadword access that is not supported by the peripheral.

The Armv8 architecture does not require the size of each element that is accessed by a multi-register load or store instruction to be identifiable by the memory system beyond the PE. Unless otherwise specified by the component, any access to a memory-mapped component of the Armv8 architecture is defined to be beyond the PE.

Software must use a Device-nGRE or stronger memory type, and only single register load and store instructions, to create memory accesses that are supported by the peripheral. For more information, see *Memory types and attributes* on page B2-153.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I1.2 Synchronization of memory-mapped registers

This section describes the synchronization requirements for the memory-mapped accesses to System registers.

This section refers to accesses to external system control registers as external reads and external writes. It refers to accesses to System registers as direct reads, direct writes, indirect reads and indirect writes.

Note

Synchronization requirements for AArch64 System registers on page D13-2869 and *Synchronization of changes to AArch32 System registers* on page G8-6100 define direct read, direct write, indirect read, and indirect write, and classifies external reads as indirect reads and external writes as indirect writes.

Writes to the same register are serialized, meaning they are observed in the same order by all observers, although some observers might not observe all of the writes. Unless otherwise stated, external writes to different registers are not necessarily observed in the same order by all observers as the order in which they complete.

Explicit synchronization is not required for an external read or an external write by an external agent to be observable to a following external read or external write by that agent to the same register using the same address, and so is never required for registers that are accessible as external system control registers.

Unless required to be observable to all observers in finite time, without explicit synchronization, explicit synchronization is normally required following an external write to any register for that write to be observable by:

- A direct access.
- An indirect read by an instruction.
- An external read of the register using a different address.

This means that an external write by an external agent is guaranteed to have an effect on subsequent instructions executed by the PE only if all of the following are true:

- The write has completed.
- The PE has executed a Context synchronization event.
- The Context synchronization event was executed after the write completed.

The order and synchronization of direct reads and direct writes of System registers is defined by:

- *Synchronization requirements for AArch64 System registers* on page D13-2869
- *Synchronization of changes to AArch32 System registers* on page G8-6100

The external agent must be able to guarantee completion of a write. For example, the agent can:

- Mark the memory as Device-nGnRnE and executing a DSB barrier, if the system supports this property.
- If the register is read/write and reads are not destructive, read back the value written.
- Use some guaranteed property of the connection between the PE and the external agent.

The external agent and PE can guarantee ordering by, for example, passing messages in an ordered way with respect to the external write and the Context synchronization event, and relying on the memory ordering rules provided by the memory model.

External reads and external write complete in the order in which they arrive at the PE. For accesses to different register locations the external agent must create this order. The agent can:

- Mark the memory as Device-nGnRnE or Device-nGnRE.
- Use the appropriate memory barriers.
- Rely on some guaranteed property of the connection between the PE and the external agent.

However, the external agent cannot force the synchronization of completed writes.

In a simple sequential execution, an indirect write that occurs as a side-effect of an access happens atomically with the access, meaning no other accesses are allowed between the register access and its side-effect.

Without explicit synchronization to guarantee the order of the accesses, where the same register is accessed by two or more of a System register access instruction, and external agent, and autonomous asynchronous event, or as a result of a memory-mapped access, the behavior must be as if the accesses occurred atomically and in any order. This applies even if the accesses occur simultaneously.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For example, some registers have the property that for certain bits a write of 0 is ignored and a write of 1 has an effect. This means the simultaneous writes must be merged.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I1.3 Access requirements for reserved and unallocated registers

This section describes the access requirements for reserved and unallocated memory-mapped components.

The following information relates to certain types of reserved accesses:

- Reads and writes of unallocated locations. These accesses are reserved for the architecture.
- Reads and writes of locations for features that are not implemented, including:
  - OPTIONAL features that are not implemented.
  - Breakpoints and watchpoints that are not implemented.
  - Performance Monitors counters that are not implemented.
  - CTI triggers that are not implemented.
  - Error records that are not implemented.

  These accesses are reserved.
- Reads of WO locations. These accesses are reserved for the architecture.
- Writes to RO locations. These accesses are reserved for the architecture.

Reserved accesses are normally RAZ/WI. However, software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture. Software must treat reserved accesses as RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter I2
System Level Implementation of the Generic Timer

This chapter defines the system level implementation of the Generic Timer. It contains the following sections:

- *About the Generic Timer specification* on page I2-7230.
- *Memory-mapped counter module* on page I2-7232.
- *Memory-mapped timer components* on page I2-7236.

Note

- *Generic Timer memory-mapped register descriptions* on page I5-7366 describes the System level Generic Timer registers. These registers are memory-mapped.
- Appendix K5 *Additional Information for Implementations of the Generic Timer* gives additional information, that does not form part of the architectural definition of a system level implementation of the Generic Timer,
- Chapter D11 *The Generic Timer in AArch64 state* gives a general description of the AArch64 state view of the Generic Timer, and describes the AArch64 System register interface to the Generic Timer.
- Chapter G6 *The Generic Timer in AArch32 state* gives a general description of the AArch32 state view of the Generic Timer, and describes the AArch32 System register interface to the Generic Timer.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I2.1 About the Generic Timer specification

Chapter D11 *The Generic Timer in AArch64 state* describes the Arm Generic Timer and its implementation as seen from AArch64 state. Chapter G6 *The Generic Timer in AArch32 state* describes the Arm Generic Timer and its implementation as seen from AArch32 state. These chapters include the definition of the low-latency System register interface to the Generic Timer. However, the Arm Generic Timer architecture also defines a memory-mapped component, that comprises:

- A memory-mapped counter module, that controls the generation of the Count value used by the Generic Timer.

  This memory-mapped counter module is required in any Arm Generic Timer implementation that requires software control of the Count value of the Generic Timer.

- Optional memory-mapped timer modules. These give a standardized way of providing timers for programmable system components other than PEs that implement the Arm architecture.

The full set of Generic Timer components on page D11-2839 summarizes these components as seen from AArch64 state, and *The full set of Generic Timer components* on page G6-6061 summarizes them as seen from AArch32 state. *The system level components of the Generic Timer* on page I2-7231 summarizes the system level components.

I2.1.1 Registers in the system level implementation of the Generic Timer

Registers that control components of the system level implementation of the Generic Timer are grouped into *frames*. This specification defines the registers in each frame, and their offsets within the frame. The system defines the position of each frame in the memory map. This means the base addresses for each frame is IMPLEMENTATION DEFINED.

Note

The final 12 words of the first or only 4KB block of a register memory frame is an ID block.

Each frame must be in its own memory page, or memory protection region, and must be aligned to the size of the translation granule or protection granule.

Note

When a system level implementation of the Generic Timer is accessed by a PE:

- Using a VMSA, each frame is in its own memory page, aligned to the size of the translation granule.
- Using a PMSA, each frame is in its own memory protection region, aligned to the size of the memory protection granule.

The following sections give more information about the requirements for the system level Generic Timer component:

- *Endianness and supported access sizes*.
- *Power and reset domains for the system level implementation of the Generic Timer* on page I2-7231.

Endianness and supported access sizes

All memory-mapped peripherals defined in the Arm architecture must be little-endian. This means the system-level Generic Timer registers, and the register frames, are little-endian.

The memory access sizes supported by any peripheral is IMPLEMENTATION DEFINED by the peripheral. For accesses to the memory-mapped Generic Timer registers implementations must:

- Comply with the requirements of *Supported access sizes* on page I1-7224.
- Support word-aligned 32-bit accesses to access 32-bit registers or either half of a 64-bit register mapped to a doubleword-aligned pair of adjacent 32-bit locations, even if no PE in the system implements AArch32.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Power and reset domains for the system level implementation of the Generic Timer

The power and reset domains of the system level implementation of the Generic Timer are IMPLEMENTATION DEFINED as part of the system implementation. These domains can be outside the PE power and reset domains defined by the remainder of this manual.

The Arm architecture requires that the CNTCR.{FCREQ, EN} and CNTSR.FCACK fields reset to 0. These reset values apply only on powerup of the power domain in which the registers are implemented or a reset of the reset domain in which they are implemented.

Every other register, or register field, of a system level implementation of the Generic Timer resets to a value that is architecturally UNKNOWN if it has a meaningful reset value. This applies on powerup of the power domain in which the register is implemented, and on a reset of the reset domain in which it is implemented.

I2.1.2 The system level components of the Generic Timer

Each system level component has one or two register frames. The possible system level components are:

The memory-mapped counter module, required

This module controls the system counter. It has two frames:

- A control frame, CNTControlBase.
- A status frame, CNTReadBase.

Memory-mapped counter module on page I2-7232 describes this component.

The memory-mapped timer control module, required

The system level implementation of the Generic Timer can provide up to eight timers, and the memory-mapped timer control module identifies:

- Which timers are implemented.
- The features of each implemented timer.

This module has a single frame, CNTCTLBase.

The CNTCTLBase frame on page I2-7237 describes this frame.

Memory-mapped timers, optional

An implemented memory-mapped timer:

- Must provide a privileged view of the timer, in the CNTBase*N* frame.
- Optionally provides an unprivileged view of the timer in the CNTEL0Base*N* frame.

N is the timer number, and the corresponding frame number, in the range 0-7.

The CNTBaseN and CNTEL0BaseN frames on page I2-7238 describes these frames.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I2.2 Memory-mapped counter module

The memory-mapped counter module provides top-level control of the system counter. The CNTControlBase frame holds the registers for the memory-mapped counter, and provides:

- An RW control register CNTCR, that provides:
  - An enable bit for the system counter.
  - An enable bit for Halt-on-debug. For more information, see *Halt-on-debug* on page I2-7234.
  - A field that can be written to request a change to the update frequency of the system counter, with a corresponding change to the increment made at each update. This mechanism means that, for example, if the update frequency is halved, the increment at each update is doubled.

    For more information, see *Control of counter operating frequency and increment* on page I2-7233.

  Writes to this register are rare. In a system that supports two Security states, this register is writable only by Secure writes.

- A RO status register, CNTSR, that provides:
  - A bit that indicates whether the system counter is halted because of an asserted Halt-on-debug signal.
  - A field that indicates the current update frequency of the system counter. This field can be polled to determine when a requested change to the update frequency has been made.

- Two contiguous 32-bit RW registers that hold the current system counter value, CNTCV. If the system supports 64-bit atomic accesses, these two registers must be accessible by such accesses.

  The system counter must be disabled before writing to these registers, otherwise the effect of the write is UNPREDICTABLE.

  Writes to these registers are rare. In a system that supports two Security states, these registers are writable only by Secure writes.

- A *Frequency modes table* of one or more 32-bit entries, where:
  - The first entry in the table defines the *base frequency* of the system counter. This is the maximum frequency at which the counter updates.
  - Each subsequent entry in the table defines an alternative frequency of the system counter, that must be an exact divisor of the base frequency.

  A 32-bit zero entry immediately follows the last table entry.

  This table can be RO or RW. For more information, see *The Frequency modes table* on page I2-7233.

In addition, the CNTReadBase frame includes a read-only copy of the system counter value, CNTCV, as two contiguous 32-bit RO registers. If the system supports 64-bit atomic accesses, these two registers must be accessible by such accesses.

Counter module control and status register summary on page I2-7234 describes CNTReadBase and CNTControlBase memory maps, and the registers in each frame.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I2.2.1 Control of counter operating frequency and increment

The system counter has a fixed *base frequency*, and must maintain the required counter accuracy, meaning Arm recommends that it does not gain or lose more than ten seconds in a 24-hour period, see *The system counter* on page D11-2840. However, the counter can increment at a lower frequency than the base frequency, using a correspondingly larger increment. For example, it can increment by four at a quarter of the base frequency. Any lower-frequency operation, and any switching between operating frequencies, must not reduce the accuracy of the counter.

Control of the system counter frequency and increment is provided only through the memory-mapped counter module. The following sections describe this control:

- *The Frequency modes table*.
- *Changing the system counter and increment*.

The Frequency modes table

The Frequency modes table starts at offset `0x20` in the CNTControlBase frame.

Table entries are 32-bits, and each entry specifies a system counter update frequency, in Hz.

The first entry in the table specifies the base frequency of the system counter.

When the system timer is operating at a lower frequency than the base frequency, the increment applied at each counter update is given by:

increment = (base_frequency) / (selected_frequency)

A 32-bit word of zero value marks the end of the table. That is, the word of memory immediately after the last entry in the table must be zero.

The only required entry in the table is the entry for the base frequency.

Typically, the Frequency modes table is in RO memory. However, a system implementation might use RW memory for the table, and initialize the table entries as part of its startup sequence. Therefore, the CNTControlBase memory map shows the table region as RO or RW.

Arm strongly recommends that the Frequency modes table is not updated once the system is running.

The architecture can support up to 1004 entries in the Frequency modes table, including the zero-word end marker, and the number of entries is IMPLEMENTATION DEFINED, up to this limit.

Note

- Arm considers it likely that implementations will require significantly fewer entries than the architectural limit.
- In the CNTControlBase frame, the offset range `0x0C0-0x0FC` can be used for IMPLEMENTATION DEFINED registers. If any registers are defined in this space, then the Frequency modes table cannot extend beyond offset `0x0B8`, with a zero word at offset `0x0BC`. This means that if any IMPLEMENTATION DEFINED registers are defined the maximum number of entries in the table is 40, including the zero-word end marker.

Changing the system counter and increment

The value of the CNTCR.FCREQ field specifies which entry in the Frequency modes table specifies the system counter update frequency.

Changing the value of CNTCR.FCREQ requests a change to the system counter update frequency. To ensure the frequency change does not affect the overall accuracy of the counter, a change is made as follows:

- When changing from a higher frequency to a lower frequency, the counter:
  1. Continues running at the higher frequency until the count reaches an integer multiple of the required lower frequency.
  2. Switches to operating at the lower frequency.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- When changing from a lower frequency to a higher frequency, the counter:
  1. Waits until the end of the current lower-frequency cycle.
  2. Makes the counter increment required for operation at that lower frequency.
  3. Switches to operating at the higher frequency.

When the frequency has changed, CNTSR is updated to indicate the new frequency. Therefore, a system component that is waiting for a frequency change can poll CNTSR to detect the change.

I2.2.2 Halt-on-debug

The CNTCR register provides an enable bit for an OPTIONAL Halt-on-debug signal.

When the CNTCR.HDBG bit is set to 1, and the Halt-on-debug signal is implemented and asserted, the system counter is halted. Otherwise, the system counter ignores the state of this signal.

Arm recommends that a system counter implements a Halt-on-debug signal that can be controlled by a debugger using the Embedded Cross-Trigger (ECT) using a system-level cross-trigger interface that includes:

- A debug request output trigger event that asserts the Halt-on-debug signal.
- A restart request output trigger event that deasserts the Halt-on-debug signal.

For more information, see *About the Embedded Cross-Trigger (ECT)* on page H5-7012.

Note

Software must use the Halt-on-debug enable bit to ensure that the timers cannot be halted maliciously in an attempt to prohibit progress.

For more information about Halt-on-debug, contact Arm.

I2.2.3 Counter module control and status register summary

The Counter module control and status registers are memory-mapped registers in the following register memory *frames*:

- A control frame, with base address CNTControlBase.
- A status frame, with base address CNTReadBase.

Each of these register memory frames is in its own memory page or memory protection region, and the frame base address points to the start of this region. Each base address must be aligned to the size of the translation granule or protection granule.

Note

Each frame of a memory-mapped Generic Timer takes the name of its base address.

In each register memory frame, the memory at offset `0xFD0-0xFFF` is reserved for twelve 32-bit IMPLEMENTATION DEFINED ID registers, see the CounterID<n> register descriptions for more information.

Note

The Arm architecture requires memory-mapped peripherals to be little-endian, and therefore the counter is little-endian.

In an implementation that supports Secure and Non-secure memory maps, CNTControlBase is accessible only by Secure accesses.

Table I2-1 on page I2-7235 shows the CNTControlBase control registers, in order of their offsets from the CNTControlBase base address, for an implementation that includes registers in the IMPLEMENTATION DEFINED register space `0x0C0-0x0FC`, and also has fewer than 39 CNTFID<n> registers. *The Frequency modes table* on page I2-7233 describes how this memory map differs if more CNTFID<n> registers are implemented.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Generic Timer memory-mapped register descriptions on page I5-7366 describes each of these registers.

Table I2-1 CNTControlBase memory map

Offset	Name	Type	Description
0x000	CNTCR	RW	Counter Control Register.
0x004	CNTSR	RO	Counter Status Register.
0x008	CNTCV[31:0]	RW	Counter Count Value register.
0x00C	CNTCV[63:32]	RW	
0x010	CNTSCR[a]	RW	Counter Scale Register.
0x014-0x018	-	RES0	Reserved.
0x01C	CNTID[a]	RO	Counter Identification Register.
0x020	CNTFID0	RO or RW	Frequency modes table, and end marker. For more information, see *The Frequency modes table* on page I2-7233.
0x020+4*n*	CNTFID<n>	RO or RW	
0x024+4*n*	-	RO or RW, RAZ	
(0x028+4*n*)-0x0BC	-	RO, RES0	Reserved.
0x0C0-0x0FC	-	IMPLEMENTATION DEFINED	Reserved for IMPLEMENTATION DEFINED registers.
0x100-0xFCC	-	RO, RES0	Reserved.
0xFD0-0xFFC	CounterID<n>	RO	Counter ID registers 0-11.

a. Implemented only if ARMv8.4-CNTSC is implemented.

Table I2-2 shows the CNTReadBase control registers, in order of their offsets from the CNTReadBase base address. *Generic Timer memory-mapped register descriptions* on page I5-7366 describes each of these registers.

Table I2-2 CNTReadBase memory map

Offset	Name	Type	Description
0x000	CNTCV[31:0]	RO	Counter Count Value register
0x004	CNTCV[63:32]	RO	
0x008-0xFCC	-	RES0	Reserved
0xFD0-0xFFC	CounterID<n>	RO	Counter ID registers 0-11

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I2.3 Memory-mapped timer components

This part of the Arm Generic Timer specification defines an optional memory-mapped timer component. This can be implemented as part of any programmable system component that does not incorporate a System register mapped Arm Generic Timer, to provide that system component with the timer functionality of an Arm Generic Timer.

The memory map consists of up to eight timer *frames*. The base address of a frame is CNTBaseN, where *N* numbers from 0 up to a maximum permitted value of 7.

Each CNTBaseN timer frame:

- Provides its own set of timers and associated interrupts.
- Is implemented in its own memory page or memory protection region.
- Is implemented at a base address, identified as CNTBaseN, that is aligned to the size of the translation granule or memory protection region.

For each implemented CNTBaseN frame the system can optionally provide an unprivileged view of the frame, described as the EL0 view of the frame. The base address of this second view of the CNTBaseN frame is CNTEL0BaseN.

Note

In the naming of the registers associated with a CNTBaseNor CNTEL0BaseN frame, the value of *N* is represented as <n>, for example CNTACR<n>.

If a CNTEL0BaseN frame is implemented:

- Is implemented in its own memory page or memory protection region and is aligned to the size of the translation granule or memory protection region.
- All registers visible in CNTBaseN, except for CNTVOFF and CNTEL0ACR, can be visible in CNTEL0BaseN.
  - Control fields in CNTEL0ACR determine whether each register is visible.
- The offsets of all visible registers are the same as their offsets in the CNTBaseN frame.

In addition to the implemented CNTBaseN and CNTEL0BaseN frames, the system must provide a single control frame at base address CNTCTLBase. CNTCTLBase must be implemented in its own memory page or memory protection region and is aligned to the size of the translation granule or memory protection region.

The system defines the position of each frame in the memory map. This means the values of each of the CNTBaseN, CNTEL0BaseN, and CNTCTLBase base addresses is IMPLEMENTATION DEFINED.

Note

The Arm architecture requires memory-mapped peripherals to be little-endian, and therefore the memory-mapped timers are little-endian.

The following sections describe the implementation of a memory-mapped view of the counter and timer:

- *The CNTCTLBase frame* on page I2-7237.
- *The CNTBaseN and CNTEL0BaseN frames* on page I2-7238.

Note

Providing a complete set of features in a system level implementation on page K5-7914 gives an implementation example for a system level Generic Timer implementation that provides equivalent features to a System registers Generic Timer implementation in a PE that includes all of the Exception levels.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I2.3.1 The CNTCTLBase frame

The CNTCTLBase frame contains:

- An identification register for the features of the memory-mapped counter and timer implementation.
- Access controls for each CNTBaseN frame.
- A virtual offset register for frames that implement a virtual timer.

Table I2-3 shows the CNTCTLBase registers, in order of their offsets from the CNTCTLBase base address.

Note

CNTFRQ and CNTVOFF registers are also implemented in a System register interface to the Generic Timer.

Generic Timer memory-mapped register descriptions on page I5-7366 describes each of these registers.

Table I2-3 CNTCTLBase memory map

Offset	Register	Type	Security[a]	Description
0x000	CNTFRQ[b]	RW	Secure only	Counter Frequency register.
0x004	CNTNSAR	RW	Secure only	Counter Non-Secure Access register.
0x008	CNTTIDR	RO	Both	Counter Timer ID register.
0x00C- 0x03F	-	RES0	-	Reserved.
0x040+4*N*[c]	CNTACR<n>	RW	Configurable[d]	Counter Access Control register *N*.
0x060- 0x07F	-	RES0	-	Reserved.
0x080+8*N*[c]	CNTVOFF<n>[31:0][b]	RW[e]	Configurable[d]	Virtual Offset register *N*. If the CNTBaseN frame has virtual timer capability then CNTVOFFis implemented as an RW register, otherwise its location is RAZ/WI.
0x084+8*N*[c]	CNTVOFF<n>[63:32][b]	RW[e]		
0x0C0-0x0FC	-	RES0	-	Reserved.
0x100-0x7FC	-	-	-	IMPLEMENTATION DEFINED.
0x800-0xFBC	-	RES0	-	Reserved.
0xFC0-0xFCF	-	-	-	IMPLEMENTATION DEFINED.
0xFD0- 0xFFC	CounterID<n>	RO	Both	Counter ID registers 0-11.

a. Access security requirement in an implementation that supports two Security states. In an implementation that does not support multiple Security states all registers are accessible as shown in the *Type* column.

b. These registers are also defined in the System register interface to the Generic Timer, and therefore are also described in *Generic Timer registers* on page D13-3839 and *Generic Timer registers* on page G8-6840. The bit assignments of the registers are identical in the System register interface and in the memory-mapped system level interface.

c. Implemented for each value of *N* from 0 to 7 for which a CNTBaseN frame is implemented.

d. The CNTNSAR determines the Non-secure accessibility of the CNTACR<n>s and the CNTVOFF<n> in the CNTCTLBase frame. For more information, see the register descriptions.

e. Address is reserved, RAZ/WI if register not implemented.

All implementations of the Generic Timer include the virtual counter. Therefore, conceptually, all implementations include the CNTVOFF register that defines the virtual offset between the physical count and the virtual count. If a memory-mapped Generic Timer component does not distinguish between real time and virtual time, then it can implement CNTVOFF as RAZ/WI. Otherwise CNTVOFF is an RW register, and Arm strongly recommends that the system only permits access to CNTVOFF from EL2 or higher.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I2.3.2 The CNTBaseN and CNTEL0BaseN frames

Each CNTBaseN frame, or {CNTBaseN, CNTEL0BaseN} pair of frames, provides a memory-mapped counter and timer, see:

- *The CNTBaseN frame*.
- *The CNTEL0BaseN frame* on page I2-7239.
- *CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames* on page I2-7239.

The CNTBase*N* frame

Table I2-4 shows the CNTBaseN registers, in order of their offsets from the CNTBaseN base address. Whether a frame includes a virtual timer is IMPLEMENTATION DEFINED. If it does not, then memory at offsets `0x030-0x03C` is RAZ/WI. Except for CNTEL0ACR and the CounterID<n> registers, equivalent registers are also implemented in a System register interface to the timer component of a Generic Timer.

Generic Timer memory-mapped register descriptions on page I5-7366 describes each of these registers.

Table I2-4 CNTBase*N* memory map

Offset	Register	Type	Description
`0x000`	CNTPCT[31:0][a]	RO	Physical Count register.
`0x004`	CNTPCT[63:32][a]	RO	
`0x008`	CNTVCT[31:0][a]	RO	Virtual Count register.
`0x00C`	CNTVCT[63:32][a]	RO	
`0x010`	CNTFRQ[a]	RO[c]	Counter Frequency register.
`0x014`	CNTEL0ACR	RW[b]	Counter EL0 Access Control Register, optional in the CNTBaseN memory map.
`0x018`	CNTVOFF[31:0][a]	RO[c]	Virtual Offset register. If CNTVOFFin the CNTCTLBase frame is an RW register, a read of this register returns the value of that register. Otherwise is RAZ.
`0x01C`	CNTVOFF[63:32][a]	RO[c]	
`0x020`	CNTP_CVAL[31:0][a]	RW	Physical Timer CompareValue register.
`0x024`	CNTP_CVAL[63:32][a]	RW	
`0x028`	CNTP_TVAL[a]	RW	Physical TimerValue register.
`0x02C`	CNTP_CTL[a]	RW	Physical Timer Control register.
`0x030`	CNTV_CVAL[31:0][a]	RW[b]	Virtual Timer CompareValue register, optional in the CNTBaseN memory map.
`0x034`	CNTV_CVAL[63:32][a]	RW[b]	
`0x038`	CNTV_TVAL[a]	RW[b]	Virtual TimerValue register, optional in the CNTBaseN memory map.
`0x03C`	CNTV_CTL[a]	RW[b]	Virtual Timer Control register, optional in the CNTBaseN memory map.
`0x040-0xFCF`	-	RES0	Reserved.
`0xFD0-0xFFC`	CounterID<n>	RO	Counter ID registers 0-11.

a. These registers are also defined in the System register interface to the Generic Timer, and therefore are also described in *Generic Timer registers* on page D13-3839 and *Generic Timer registers* on page G8-6840. The bit assignments of the registers are identical in the System register interface and in the memory-mapped system level interface.

b. Address is reserved, RAZ/WI if register not implemented.

c. The CNTCTLBase frame includes an RW view of this register.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The CNTEL0Base*N* frame

For any value of *N*, the layout of the registers in the CNTEL0Base*N* frame is identical to the CNTBaseN frame, except that, in the CNTEL0Base*N* frame:

- CNTVOFF is never visible, and the memory at `0x018-0x01C` is RAZ/WI.
- CNTEL0ACR is never visible, and the memory at `0x014` is RAZ/WI.
- If implemented in the CNTBaseN frame, CNTEL0ACR controls whether CNTPCT, CNTVCT, CNTFRQ, the Physical Timer, and the Virtual Timer registers are visible in the CNTEL0BaseN frame.

  If CNTEL0ACR is not implemented then these registers are not visible in the CNTEL0BaseN frame, and their addresses in that frame are RAZ/WI.

If an implementation supports 64-bit atomic accesses, then CNTPCT, CNTVCT, CNTVOFF, CNTP_CVAL, and CNTV_CVAL must be accessible as atomic 64-bit values.

CNTCTLBase status and control fields for the CNTBase*N* and CNTEL0Base*N* frames

In the CNTCTLBase frame:

CNTTIDR controls:

- Whether each CNTBaseN frame is implemented.
- If a CNTBaseN frame is implemented, whether:
  - That CNTBaseN frame has virtual timer capability.
  - A corresponding CNTEL0BaseN frame is implemented.

CNTNSAR controls:

In an implementation that recognizes two Security states, determines whether each implemented CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses.

This control also determines whether, in the CNTCTLBase frame, the CNTACR<n> and CNTVOFF<n> registers are accessible by Non-secure accesses.

The CNTACR<n> registers control:

For each implemented CNTBaseN frame, the accessibility of the following registers in that frame:

- CNTP_CTL, CNTP_CVAL, and CNTP_TVAL.
- CNTV_CTL, CNTV_CVAL, and CNTV_TVAL.
- CNTVOFF.
- CNTFRQ.
- CNTPCT.
- CNTVCT.

For CNTACR<n>, the value of <n> corresponds to the value of *N* for the controlled CNTBaseN frame.

The CNTVOFF<n> registers provide:

For each implemented CNTBaseN frame that has virtual capability, the RW copy of the CNTVOFF register for that frame.

Note

In a CNTBaseN frame that has virtual timer capability the CNTVOFF register is RO.

For CNTVOFF<n>, the value of <n> corresponds to the value of *N* for the controlled CNTBaseN frame.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter I3
Recommended External Interface to the Performance Monitors

This chapter describes the recommended external interface to the Performance Monitors. It contains the following section:

- *About the external interface to the Performance Monitors registers* on page I3-7242.

Note

Performance Monitors external register descriptions on page I5-7255 describes the external view of the Performance Monitors registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I3.1 About the external interface to the Performance Monitors registers

Arm recommends that:

- An implementation provides the OPTIONAL external debug interface to the Performance Monitors registers.

  Note

  A debugger can use this interface to access counters in the Performance Monitors.

- The implementation includes the OPTIONAL support for memory-mapped access to the External debug interface.

  Note

  — Software running on any PE in a system can use this interface to access counters in the Performance Monitors.

  — Privileged software should use the MMU to control access to this interface.

- The external debug interface is implemented as defined in Appendix K2 *Recommended External Debug Interface*.

The following sections describe the memory-mapped views of the Performance Monitors registers:

- *Differences in the external views of the Performance Monitors registers*.
- *Synchronization of changes to the memory-mapped views* on page I3-7243.
- *Access permissions for external views of the Performance Monitors* on page I3-7243.

In this section, unless the context explicitly indicates otherwise, any reference to *a memory-mapped view* applies equally to a register view using:

- An access through an external debug interface.
- A memory-mapped access.

I3.1.1 Endianness and supported access sizes

When an implementation supports memory-mapped access to the external debug interface the interface is accessed as a little-endian memory-mapped peripheral. *External Performance Monitors registers summary* on page I5-7252 gives the memory map of these registers.

The memory access sizes supported by any peripheral is IMPLEMENTATION DEFINED by the peripheral. For accesses to the external interface to the Performance Monitors registers implementations must:

- Comply with the requirements of *Supported access sizes* on page I1-7224.
- Support word-aligned 32-bit accesses to access 32-bit registers or either half of a 64-bit register mapped to a doubleword-aligned pair of adjacent 32-bit locations, even if no PE in the system implements AArch32.

I3.1.2 Differences in the external views of the Performance Monitors registers

An external view of the Performance Monitors registers accesses the same registers as the System registers interface described in *Performance Monitors Extension registers* on page D7-2772, except that:

1. The PMSELR is accessible only in the System registers interface.
2. The following registers are accessible only in external views:
   - PMCFGR
   - PMDEVAFF0
   - PMDEVAFF1
   - PMLAR
   - PMLSR
   - PMAUTHSTATUS

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- PMDEVARCH
- PMDEVTYPE
- PMPIDR0
- PMPIDR1
- PMPIDR2
- PMPIDR3
- PMPIDR4
- PMCIDR0
- PMCIDR1
- PMCIDR2
- PMCIDR3

Performance Monitors external register descriptions on page I5-7255 describes these registers.

3. The following controls do not affect the external views:
   - PMSELR.
   - PMUSERENR.
   - HDCR.{TPM, TPMCR, HPMN}.

   Instead, see the register descriptions in Chapter I5 *External System Control Register Descriptions*.

I3.1.3 Synchronization of changes to the memory-mapped views

Synchronization must comply with *Synchronization of memory-mapped registers* on page I1-7226.

In particular, if a Performance Monitor is visible in both System register and an external view, and is accessed simultaneously through these two mechanisms, the behavior must be as if the access occurred atomically in any order. For more information, see *Synchronization of changes to the external debug registers* on page H8-7050.

I3.1.4 Access permissions for external views of the Performance Monitors

For more information, see *External debug interface register access permissions* on page H8-7057.

Table I3-1 on page I3-7244 shows the access permissions for the Performance Monitors registers in a v8 Debug implementation. This table uses the following terms:

DLK When ARMv8.0-DoubleLock is implemented and locked, `DoubleLockStatus()` == TRUE, accesses to some registers produce an error. Applies to both interfaces.

EPMAD When `AllowExternalPMUAccess()` == FALSE, external debug access is disabled for the access. If ARMv8.4-Debug is implemented, this applies only for Non-secure access to the register. See also *Behavior of a not permitted memory-mapped access* on page H8-7055.

Error Indicates that the access gives an error response.

Default This shows the default access permissions, if none of the conditions in this list prevent access to the register.

Off The Core power domain is completely off, or in a low-power state where the Core power domain registers cannot be accessed, and EDPRSR.PU will read as zero.

Note

If debug power is off, then all external debug interface accesses return an error.

OSLK When the OS Lock is locked, OSLAR_EL1.OSLK == 1, accesses to some registers produces an error. This column shows the effect of this control on accesses using the external debug interface.

SLK This indicates the modified default access permissions for OPTIONAL memory-mapped accesses to the external debug interface if the optional Software Lock is locked. See *Register access permissions for memory-mapped accesses* on page H8-7054.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For all other accesses, this column is ignored.

- Indicates that the control has no effect on the behavior of the access:
  - If no other control affects the behavior, the Default access behavior applies.
  - However, another control might determine the behavior.

Table I3-1 Access permissions for the Performance Monitors registers

Offset	Register	Domain	Off	DLK	OSLK	EPMAD	Default	SLK
0x000+8xn	PMEVCNTR<n>_EL0[a]	Core	Error	Error	Error	Error	RW	RO
0x0F8	PMCCNTR_EL0[31:0]	Core	Error	Error	Error	Error	RW	RO
0x0FC	PMCCNTR_EL0[63:32]	Core	Error	Error	Error	Error	RW	RO
0x200	PMPCSR[31:0][b]	Core	Error	Error	Error	-	RO	RO
0x204	PMPCSR[63:32][b]	Core	Error	Error	Error	-	RO	RO
0x208	PMCID1SR[b]	Core	Error	Error	Error	-	RO	RO
0x20C	PMVIDSR[b]	Core	Error	Error	Error	-	RO	RO
0x220	PMPCSR[31:0][b]	Core	Error	Error	Error	-	RO	RO
0x224	PMPCSR[63:32][b]	Core	Error	Error	Error	-	RO	RO
0x228	PMCID1SR[b]	Core	Error	Error	Error	-	RO	RO
0x22C	PMCID2SR[b]	Core	Error	Error	Error	-	RO	RO
0x400+4xn	PMEVTYPER<n>_EL0[a]	Core	Error	Error	Error	Error	RW	RO
0x47C	PMCCFILTR_EL0	Core	Error	Error	Error	Error	RW	RO
0x600-0x6FC	-	-	Access is IMPLEMENTATION DEFINED					
0xA00-0xBFC	-	-	Access is IMPLEMENTATION DEFINED					
0xC00	PMCNTENSET_EL0	Core	Error	Error	Error	Error	RW	RO
0xC20	PMCNTENCLR_EL0	Core	Error	Error	Error	Error	RW	RO
0xC40	PMINTENSET_EL1	Core	Error	Error	Error	Error	RW	RO
0xC60	PMINTENCLR_EL1	Core	Error	Error	Error	Error	RW	RO
0xC80	PMOVSCLR_EL0	Core	Error	Error	Error	Error	RW	RO
0xCA0	PMSWINC_EL0[c]	Core	Error	Error	Error	Error	WO	WI
0xCC0	PMOVSSET_EL0	Core	Error	Error	Error	Error	RW	RO
0xD80-0xDFC	-	-	Access is IMPLEMENTATION DEFINED					
0xE00	PMCFGR	Core	Error	Error	Error	Error	RO	RO
0xE04	PMCR_EL0	Core	Error	Error	Error	Error	RW	RO
0xE20	PMCEID0	Core	Error	Error	Error	Error	RO	RO
0xE24	PMCEID1	Core	Error	Error	Error	Error	RO	RO

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table I3-1 Access permissions for the Performance Monitors registers (continued)

Offset	Register	Domain	Off	DLK	OSLK	EPMAD	Default	SLK
0xE28	PMCEID2	Core	Error	Error	Error	Error	RO	RO
0xE2C	PMCEID3	Core	Error	Error	Error	Error	RO	RO
0xE40	PMMIR	Core	Error	Error	Error	Error	RO	RO
0xE80-0xEFC	Integration registers	-	Access is IMPLEMENTATION DEFINED					
0xF00-0xFFC	*Management registers and CoreSight compliance* on page K2-7881							

a. Implemented counters only. *n* is the counter number.

b. Implemented only if ARMv8.2-PCSample is implemented. See Chapter H7 *The PC Sample-based Profiling Extension*.

c. Only if the OPTIONAL PMSWINC_EL0 register is implemented in the external debug interface.

I3.1.5 Power domains and Performance Monitors registers reset

For Armv8-A implementations, Arm recommends that Performance Monitors are implemented as part of the Core power domain, not as part of a separate Debug power domain. There is no interface to access the Performance Monitors registers when the Core power domain is powered down.

A Warm or Cold reset sets the Performance Monitors registers to their reset values. An External Debug reset does not change the values of the Performance Monitors registers.

For more information about the reset scheme recommended for a v8 Debug implementation see Chapter H6 *Debug Reset and Powerdown Support*.

Table I3-2 on page I3-7246 shows the Performance Monitors register resets for writable register fields. The column headings use the following terms:

64 This is the architectural reset value when resetting into AArch64 state.

32 This is the architectural reset value when resetting into AArch32 state.

- This indicates an IMPLEMENTATION DEFINED reset value on the specified reset. This might be UNKNOWN.

Note

This table does not include:

- Read-only identification registers and fields that have a fixed value. In this case, the reset value is that fixed value. An example of this is PMCR_EL0.N.
- Write-only registers and fields that only have an effect on writes. These do not have a reset value. An example of this is PMSWINC_EL0.
- IMPLEMENTATION DEFINED registers. In this case, the reset domains are IMPLEMENTATION DEFINED. The reset values are IMPLEMENTATION DEFINED and might be UNKNOWN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table I3-2 Performance Monitors System register resets

Register	Domain	Field	64	32	Description
PMCR_EL0	Warm	DP	-	0	Disable PMCCNTR_EL0 when prohibited
		X	-	0	Export enable
		D	-	0	Clock divider
		E	0	0	Performance Monitors enable
PMCNTENSET_EL0 PMCNTENCLR_EL0	Warm	-	-	-	All fields in register
PMOVSSET_EL0 PMOVSCLR_EL0	Warm	-	-	-	All fields in register
PMSELR_EL0	Warm	SEL	-	-	Selected event counter
PMCCNTR_EL0	Warm	-	-	-	All fields in register
PMEVTYPER<n>_EL0	Warm	-	-	-	All fields in register
PMCCFILTR_EL0	Warm	[31:26]	-	`0x00`	PMCCNTR_EL0 filtering controls
PMEVCNTR<n>_EL0	Warm	-	-	-	All fields in register
PMUSERENR_EL0	Warm	ER	-	0	Enable counter read access in EL0
		CR	-	0	Enable PMCCNTR_EL0 read access in EL0
		SW	-	0	Enable PMSWINC_EL0 write access in EL0
		EN	-	0	Enable Performance Monitors access in EL0
PMINTENSET_EL1 PMINTENCLR_EL1	Warm	-	-	-	All fields in register

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter I4
Recommended External Interface to the Activity Monitors

This chapter describes the optional external interface to the Activity Monitors Extension registers. It contains the following section:

- *About the external interface to the Activity Monitors Extension registers* on page I4-7248

Note

Activity Monitors external register descriptions on page I5-7330 describes the external view of the Activity Monitors Extension registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I4.1 About the external interface to the Activity Monitors Extension registers

If an implementation supports the Activity Monitors Extension, it may optionally support an external memory-mapped interface to the Activity Monitors Extension, and, if so, may further optionally support CoreSight device registers and ID registers.

The memory access sizes supported by the memory-mapped interface are IMPLEMENTATION DEFINED and comply with the requirements of *Supported access sizes* on page I1-7224.

The base address of the memory-mapped view is aligned to a 4KB boundary, but is otherwise IMPLEMENTATION DEFINED. The address offsets for the memory-mapped view are given in Table I5-2 on page I5-7328.

I4.1.1 Differences in the external views of the Activity Monitors Extension registers

The external memory-mapped interface view of the Activity Monitors Extension registers accesses the same registers as the System registers interface to the registers, except that:

- AMUSERENR is accessible only in the System registers interface.
- If implemented, the following registers are accessible only in the memory-mapped view:
  - AMIIDR
  - AMDEVAFF0
  - AMDEVAFF1
  - AMDEVARCH
  - AMDEVTYPE
  - AMPIDR0
  - AMPIDR1
  - AMPIDR2
  - AMPIDR3
  - AMPIDR4
  - AMCIDR0
  - AMCIDR1
  - AMCIDR2
  - AMCIDR3

  Activity Monitors external register descriptions on page I5-7330 describes these registers.
- If ARMv8.6-AMU virtualization of the activity monitors is enabled, the memory-mapped view of the registers presents the physical view of the counter without any offset. Virtualization of the Activity Monitors does not affect the memory-mapped view of the registers.

  Note

  The memory mapped view of the activity monitors is unaffected byAMCR_EL0.CG1RZ and AMCR.CG1RZ.

I4.1.2 Access during reset and power transitions

As described in *Power and reset domains* on page D8-2775, the power and reset domains of the activity monitoring unit are IMPLEMENTATION DEFINED, and when a Cold reset of the power domain of the activity monitoring unit occurs, the activity monitoring unit is reset and the counters are reset to zero.

If the power domain of the activity monitoring unit is an always-on power domain, while the PE is reset or powered down counter values may be preserved and might be accessible by memory-mapped access.

If the power domain of the activity monitoring unit is the Core power domain, while the PE is reset or powered down and when a memory-mapped access occurs, the access reads as zero and the bus access completes without an error.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Chapter I5
External System Control Register Descriptions

This chapter describes the external system control registers. It excludes the External debug registers that are described in Chapter H9 *External Debug Register Descriptions*. It contains the following sections:

- *About the external system control register descriptions* on page I5-7250.
- *External Performance Monitors registers summary* on page I5-7252.
- *Performance Monitors external register descriptions* on page I5-7255.
- *External Activity Monitors Extension registers summary* on page I5-7328.
- *Activity Monitors external register descriptions* on page I5-7330.
- *Generic Timer memory-mapped registers overview* on page I5-7365.
- *Generic Timer memory-mapped register descriptions* on page I5-7366.
- *RAS Registers* on page I5-7410.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.1 About the external system control register descriptions

This chapter describes the external system control registers other than the external debug registers. That is, it describes:

An external view of the Performance Monitors registers

Arm recommends that implementations provide access to the Performance Monitors registers through the OPTIONAL External debug interface, and provide the OPTIONAL memory-mapped interface to this interface:

- *External Performance Monitors registers summary* on page I5-7252 lists the registers that are accessible in this view of the Performance Monitors, and describes their memory map.
- *Performance Monitors external register descriptions* on page I5-7255 describes each of the memory-mapped registers.

Chapter I3 *Recommended External Interface to the Performance Monitors* describes the recommended interface to these registers.

Note

Chapter D7 *The Performance Monitors Extension* describes the Performance Monitors. The following sections describe the System register interfaces to the Performance Monitors:

- *Performance Monitors registers* on page D13-3663, for accesses from an Exception level that is using AArch64.
- *Performance Monitors registers* on page G8-6694, for accesses from an Exception level that is using AArch32.

An external view of the Activity Monitors Extension registers

An implementation which supports the Activity Monitors Extension may support an optional external memory-mapped interface to the Activity Monitors Extension registers.

- *External Activity Monitors Extension registers summary* on page I5-7328 lists the registers that are accessible in this view of the Performance Monitors, and describes their memory map.
- *Activity Monitors external register descriptions* on page I5-7330 describes each of the memory-mapped registers.

Chapter I3 *Recommended External Interface to the Performance Monitors* describes the recommended interface to these registers.

Note

Chapter D8 *The Activity Monitors Extension* describes the Activity Monitors. The following sections describe the System register interfaces to the Activity Monitors:

- *Activity Monitors registers* on page D13-3724, for accesses from an Exception level that is using AArch64.
- *Activity Monitors registers* on page G8-6759, for accesses from an Exception level that is using AArch32.

The registers for the system level Generic Timer component

Any implementation that includes the Generic Timer must include the memory-mapped system level component described in Chapter I2 *System Level Implementation of the Generic Timer*. In this chapter:

- *Generic Timer memory-mapped registers overview* on page I5-7365 gives an overview of the registers, referring to Chapter I2 for more information.
- *Generic Timer memory-mapped register descriptions* on page I5-7366 describes each of the memory-mapped registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

Chapter D11 *The Generic Timer in AArch64 state* describes the Generic Timer component that is accessible using the System registers. The following sections describe the System register interfaces to that component:

- *Generic Timer registers* on page D13-3839, for accesses from an Exception level that is using AArch64.
- *Generic Timer registers* on page G8-6840, for accesses from an Exception level that is using AArch32.

Note

Chapter H9 *External Debug Register Descriptions* describes the external debug registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.2 External Performance Monitors registers summary

When an implementation provides access to the Performance Monitors registers through the External debug interface, that interface provides access to:

- Performance Monitors System registers.
- A read-only configuration register, PMCFGR.
- The OPTIONAL CoreSight registers for the Performance Monitors, if they are implemented.

The locations of the registers are defined as offsets from a system-defined base address. *Performance Monitors external register views* defines this memory map.

I5.2.1 Performance Monitors external register views

Table I5-1 shows the external view of the Performance Monitors registers. All other entries are reserved.

Note

- Counters that are reserved because HDCR.HPMN has been changed from its reset value remain visible in any external view.
- The registers that relate to an implemented event counter, PMN*x*, are PMEVCNTR<n> and PMEVTYPER<n>.
- The mapping of the *Performance Monitors Event Counter Registers*, at offsets `0x000-0x0F4`, has changed compared to the mappings of the equivalent registers in Armv7.

Each entry in the Name column links to the register description in *Performance Monitors external register descriptions* on page I5-7255, and:

- If the *System register?* column of the table shows that the register is a System register, the memory-mapped interface provides a view of the System register described in:
  - *Performance Monitors registers* on page D13-3663, for the AArch64 System register.
  - *Performance Monitors registers* on page G8-6694, for the AArch32 System register.
- Otherwise, the register is accessible only using the external interface.

Table I5-1 Performance Monitors external register views

Name	Type	Description	System register?	Offset
PMEVCNTR<n>_EL0	RW	Performance Monitors Event Counter Register.	Yes	`0x000`+8n
PMCCNTR_EL0[31:0] PMCCNTR_EL0[63:32]	RW	Performance Monitors Cycle Counter Register [a]	Yes	`0x0F8` `0x0FC`
PMPCSR[31:0] [b]	RW	Program Counter Sample Register, bits[31:0]	No	`0x200`
PMPCSR[63:32] [b]	RW	Program Counter Sample Register, bits[63:32]	No	`0x204`
PMCID1SR [b]	RW	CONTEXTIDR_EL1 Sample Register	No	`0x208`
PMVIDSR [b]	RW	VMID Sample Register	No	`0x20C`
PMPCSR[31:0] [b]	RW	Program Counter Sample Register, bits[31:0], alias	No	`0x220`
PMPCSR[63:32] [b]	RW	Program Counter Sample Register, bits[63:32], alias	No	`0x224`
PMCID1SR [b]	RW	CONTEXTIDR_EL1 Sample Register (alias)	No	`0x248`
PMCID2SR [b]	RW	CONTEXTIDR_EL2 Sample Register	No	`0x22C`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table I5-1 Performance Monitors external register views (continued)

Name	Type	Description	System register?	Offset
PMEVTYPER<n>_EL0	RW	Performance Monitors Event Type and Filter Register.	Yes	0x400+4n
PMCCFILTR_EL0	RW	Performance Monitors Cycle Counter Filter Register	Yes	0x47C
-	-	IMPLEMENTATION DEFINED	-	0x600-0x6FC
-	-	IMPLEMENTATION DEFINED	-	0xA00-0xBFC
PMCNTENSET_EL0	RW	Performance Monitors Count Enable Set register	Yes	0xC00
PMCNTENCLR_EL0	RW	Performance Monitors Count Enable Clear register	Yes	0xC20
PMINTENSET_EL1	RW	Performance Monitors Interrupt Enable Set register	Yes	0xC40
PMINTENCLR_EL1	RW	Performance Monitors Interrupt Enable Clear register	Yes	0xC60
PMOVSCLR_EL0	RW	Performance Monitors Overflow Flag Status Clear register	Yes	0xC80
PMSWINC_EL0	WO	Performance Monitors Software Increment register	Yes	0xCA0
PMOVSSET_EL0	RW	Performance Monitors Overflow Flag Status Set register	Yes	0xCC0
-	-	IMPLEMENTATION DEFINED	-	0xD80-0xDFC
PMCFGR	RO	Performance Monitors Configuration Register	No	0xE00
PMCR_EL0	RW	Performance Monitors Control Register	Yes	0xE04
PMCEID0	RO	Performance Monitors Common Event Identification register 0	Yes	0xE20
PMCEID1	RO	Performance Monitors Common Event Identification register 1	Yes	0xE24
PMCEID2	RO	Performance Monitors Common Event Identification register 2	Yes	0xE28
PMCEID3	RO	Performance Monitors Common Event Identification register 3	Yes	0xE2C
-	-	IMPLEMENTATION DEFINED	-	0xE80-0xEFC
PMITCTRL[c]	RW	Integration Model Control registers	No	0xF00
PMDEVAFF0[c]	RO	Device Affinity registers	No	0xFA8
PMDEVAFF1[c]	RO			0xFAC
PMLAR[c, d]	WO	Lock Access register	No	0xFB0
PMLSR[c, d]	RO	Lock Status register	No	0xFB4
PMAUTHSTATUS[c]	RO	Authentication Status register	No	0XFB8
PMDEVARCH[c]	RO	Device Architecture register	No	0xFBC
PMDEVID[b]	RO	Performance Monitors Device ID register	No	0xFC8
PMDEVTYPE[c]	RO	Device Type register	No	0xFCC

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table I5-1 Performance Monitors external register views (continued)

Name	Type	Description	System register?	Offset
PMPIDR4[c]	RO	Peripheral ID registers	No	0xFD0
PMPIDR0[c]	RO			0xFE0
PMPIDR1[c]	RO			0xFE4
PMPIDR2[c]	RO			0xFE8
PMPIDR3[c]	RO			0xFEC
PMCIDR0[c]	RO	Component ID registers	No	0xFF0
PMCIDR1[c]	RO			0xFF4
PMCIDR2[c]	RO			0xFF8
PMCIDR3[c]	RO			0xFFC

a. The interface must support at least single-copy atomic 32-bit accesses. If single-copy atomic 64-bit access to the registers is not possible, software must use a high-low-high read access to read the counter value if the counter is enabled.

b. PC Sample-based Profiling Extension registers. Implemented only when ARMv8.2-PCSample is implemented, except that from Armv8.2 PMDEVIDt is required regardless of whether ARMv8.2-PCSample is implemented.

Before Armv8.2. the PC Sample-based Profiling Extension can, instead, be implemented in the memory-mapped debug registers space, see Chapter H7 *The PC Sample-based Profiling Extension*.

c. CoreSight interface registers, see *Management registers and CoreSight compliance* on page K2-7881.

d. The Software lock registers are defined as part of CoreSight compliance, but their contents depend on the type of access that is made and whether the OPTIONAL Software lock is implemented. See the register description for details.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3 Performance Monitors external register descriptions

This section describes the external view of the Performance Monitors registers. *External Performance Monitors registers summary* on page I5-7252 lists these registers in offset order.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.1 PMAUTHSTATUS, Performance Monitors Authentication Status register

The PMAUTHSTATUS characteristics are:

Purpose

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for Performance Monitors.

Configurations

If ARMv8.3-DoPD is implemented, this register is in the Core power domain. If ARMv8.3-DoPD is not implemented, this register is in the Debug power domain.

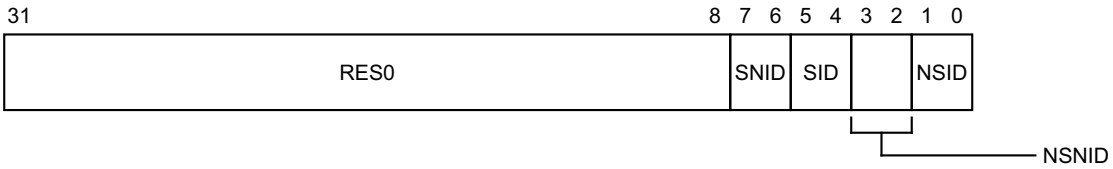
This register is OPTIONAL, and is required for CoreSight compliance. Arm recommends that this register is implemented.

Attributes

PMAUTHSTATUS is a 32-bit register.

Field descriptions

The PMAUTHSTATUS bit assignments are:



Bits [31:8]

Reserved, RES0.

SNID, bits [7:6]

Holds the same value as DBGAUTHSTATUS_EL1.SNID.

SID, bits [5:4]

Secure invasive debug. Possible values of this field are:

`0b00` Not implemented.

All other values are reserved.

NSNID, bits [3:2]

Holds the same value as DBGAUTHSTATUS_EL1.NSNID.

NSID, bits [1:0]

Non-secure invasive debug. Possible values of this field are:

`0b00` Not implemented.

All other values are reserved.

Accessing the PMAUTHSTATUS:

PMAUTHSTATUS can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0xFB8`	PMAUTHSTATUS

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.2 PMCCFILTR_EL0, Performance Monitors Cycle Counter Filter Register

The PMCCFILTR_EL0 characteristics are:

Purpose

Determines the modes in which the Cycle Counter, PMCCNTR_EL0, increments.

Configurations

External register PMCCFILTR_EL0[31:0] is architecturally mapped to AArch64 System register PMCCFILTR_EL0[31:0].

External register PMCCFILTR_EL0[31:0] is architecturally mapped to AArch32 System register PMCCFILTR[31:0].

PMCCFILTR_EL0 is in the Core power domain.

On a Warm or Cold reset, RW fields in this register reset:

- To architecturally UNKNOWN values if the reset is to an Exception level that is using AArch64.
- To 0 if the reset is to an Exception level that is using AArch32.

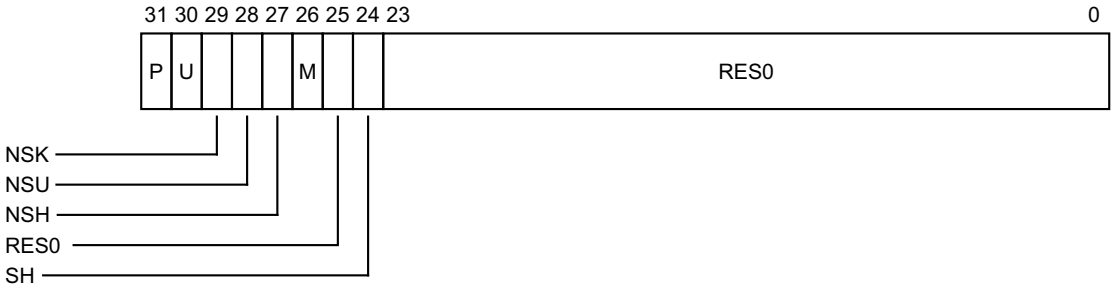
The register is not affected by an External debug reset.

Attributes

PMCCFILTR_EL0 is a 32-bit register.

Field descriptions

The PMCCFILTR_EL0 bit assignments are:



P, bit [31]

Privileged filtering bit. Controls counting in EL1.

If EL3 is implemented, then counting in Non-secure EL1 is further controlled by the PMCCFILTR_EL0.NSK bit.

`0b0` Count cycles in EL1.

`0b1` Do not count cycles in EL1.

U, bit [30]

User filtering bit. Controls counting in EL0.

If EL3 is implemented, then counting in Non-secure EL0 is further controlled by the PMCCFILTR_EL0.NSU bit.

`0b0` Count cycles in EL0.

`0b1` Do not count cycles in EL0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

NSK, bit [29]

When EL3 is implemented:

Non-secure EL1 (kernel) modes filtering bit. Controls counting in Non-secure EL1.

If the value of this bit is equal to the value of the PMCCFILTR_EL0.P bit, cycles in Non-secure EL1 are counted.

Otherwise, cycles in Non-secure EL1 are not counted.

Otherwise:

Reserved, RES0.

NSU, bit [28]

When EL3 is implemented:

Non-secure EL0 (Unprivileged) filtering bit. Controls counting in Non-secure EL0.

If the value of this bit is equal to the value of the PMCCFILTR_EL0.U bit, cycles in Non-secure EL0 are counted.

Otherwise, cycles in Non-secure EL0 are not counted.

Otherwise:

Reserved, RES0.

NSH, bit [27]

When EL2 is implemented:

EL2 (Hypervisor) filtering bit. Controls counting in EL2.

If Secure EL2 is implemented, counting in Secure EL2 is further controlled by the PMCCFILTR_EL0.SH bit.

`0b0` Do not count cycles in EL2.

`0b1` Count cycles in EL2.

Otherwise:

Reserved, RES0.

M, bit [26]

When EL3 is implemented:

Secure EL3 filtering bit.

If the value of this bit is equal to the value of the PMCCFILTR_EL0.P bit, cycles in Secure EL3 are counted.

Otherwise, cycles in Secure EL3 are not counted.

Most applications can ignore this field and set its value to 0.

Note

This field is not visible in the AArch32 PMCCFILTR System register.

Otherwise:

Reserved, RES0.

Bit [25]

Reserved, RES0.

SH, bit [24]

When ARMv8.4-SecEL2 is implemented:

Secure EL2 filtering.

If the value of this bit is not equal to the value of the PMCCFILTR_EL0.NSH bit, cycles in Secure EL2 are counted.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise, cycles in Secure EL2 are not counted.

If Secure EL2 is not implemented or is disabled, this field is RES0.

Note

This field is not visible in the AArch32 PMCCFILTR System register.

Otherwise:

Reserved, RES0.

Bits [23:0]

Reserved, RES0.

Accessing the PMCCFILTR_EL0:

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

PMCCFILTR_EL0 can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0x47C`	PMCCFILTR_EL0

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.3 PMCCNTR_EL0, Performance Monitors Cycle Counter

The PMCCNTR_EL0 characteristics are:

Purpose

Holds the value of the processor Cycle Counter, CCNT, that counts processor clock cycles. See *Time as measured by the Performance Monitors cycle counter* on page D7-2702 for more information.

PMCCFILTR_EL0 determines the modes and states in which the PMCCNTR_EL0 can increment.

Configurations

External register PMCCNTR_EL0[63:0] is architecturally mapped to AArch64 System register PMCCNTR_EL0[63:0].

External register PMCCNTR_EL0[63:0] is architecturally mapped to AArch32 System register PMCCNTR[63:0].

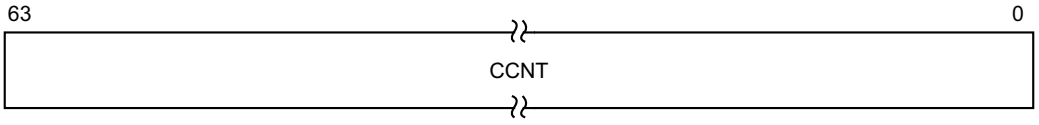
PMCCNTR_EL0 is in the Core power domain.

Attributes

PMCCNTR_EL0 is a 64-bit register.

Field descriptions

The PMCCNTR_EL0 bit assignments are:



CCNT, bits [63:0]

Cycle count. Depending on the values of PMCR_EL0.{LC,D}, the cycle count increments in one of the following ways:

- Every processor clock cycle.
- Every 64th processor clock cycle.

Writing 1 to PMCR_EL0.C sets this field to 0.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMCCNTR_EL0:

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

PMCCNTR_EL0[31:0] can be accessed through the external debug interface:

Component	Offset	Instance	Range
PMU	`0x0F8`	PMCCNTR_EL0	31:0

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and SoftwareLockStatus() accesses to PMCCNTR_EL0[31:0] are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and !SoftwareLockStatus() accesses to PMCCNTR_EL0[31:0] are RW.
- Otherwise accesses to PMCCNTR_EL0[31:0] generate an error response.

PMCCNTR_EL0[63:32] can be accessed through the external debug interface:

Component	Offset	Instance	Range
PMU	`0x0FC`	PMCCNTR_EL0	63:32

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and SoftwareLockStatus() accesses to PMCCNTR_EL0[63:32] are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and !SoftwareLockStatus() accesses to PMCCNTR_EL0[63:32] are RW.
- Otherwise accesses to PMCCNTR_EL0[63:32] generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.4 PMCEID0, Performance Monitors Common Event Identification register 0

The PMCEID0 characteristics are:

Purpose

Defines which common architectural events and common microarchitectural events are implemented, or counted, using PMU events in the range `0x0000` to `0x001F`

When the value of a bit in the register is 1 the corresponding common event is implemented and counted.

For more information about the common events and the use of the PMCEIDn registers see *The PMU event number space and common events* on page D7-2721.

Note

- Arm recommends that, if a common event is never counted, the value of the corresponding register bit is 0.
- This view of the register was previously called PMCEID0_EL0.

Configurations

External register PMCEID0[31:0] is architecturally mapped to AArch64 System register PMCEID0_EL0[31:0].

External register PMCEID0[31:0] is architecturally mapped to AArch32 System register PMCEID0[31:0].

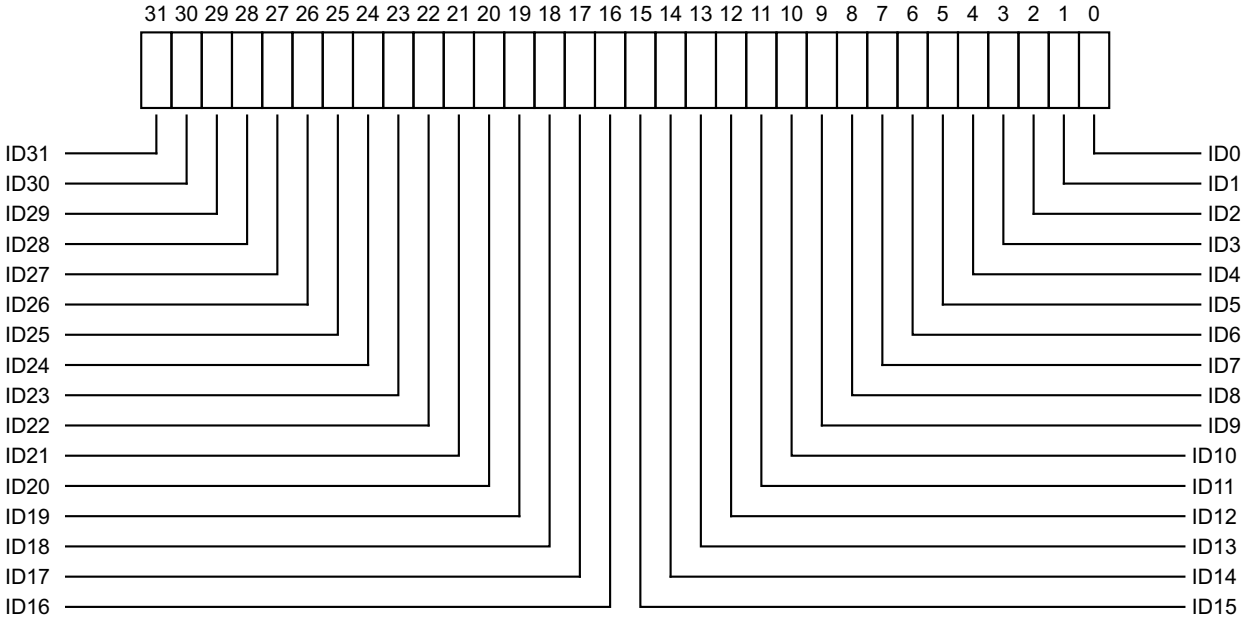
PMCEID0 is in the Core power domain.

Attributes

PMCEID0 is a 32-bit register.

Field descriptions

The PMCEID0 bit assignments are:



Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ID<n>, bit [n], for n = 0 to 31

ID[n] corresponds to common event n.

For each bit:

`0b0` The common event is not implemented, or not counted.

`0b1` The common event is implemented.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n> registers of that earlier version of the PMU architecture.

Accessing the PMCEID0:

Note

AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

PMCEID0 can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0xE20`	PMCEID0

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and AllowExternalPMUAccess() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.5 PMCEID1, Performance Monitors Common Event Identification register 1

The PMCEID1 characteristics are:

Purpose

Defines which common architectural events and common microarchitectural events are implemented, or counted, using PMU events in the range `0x020` to `0x03F`.

When the value of a bit in the register is 1 the corresponding common event is implemented and counted.

For more information about the common events and the use of the PMCEIDn registers see *The PMU event number space and common events* on page D7-2721.

Note

- Arm recommends that, if a common event is never counted, the value of the corresponding register bit is 0.
- This view of the register was previously called PMCEID1_EL0.

Configurations

External register PMCEID1[31:0] is architecturally mapped to AArch64 System register PMCEID1_EL0[31:0].

External register PMCEID1[31:0] is architecturally mapped to AArch32 System register PMCEID1[31:0].

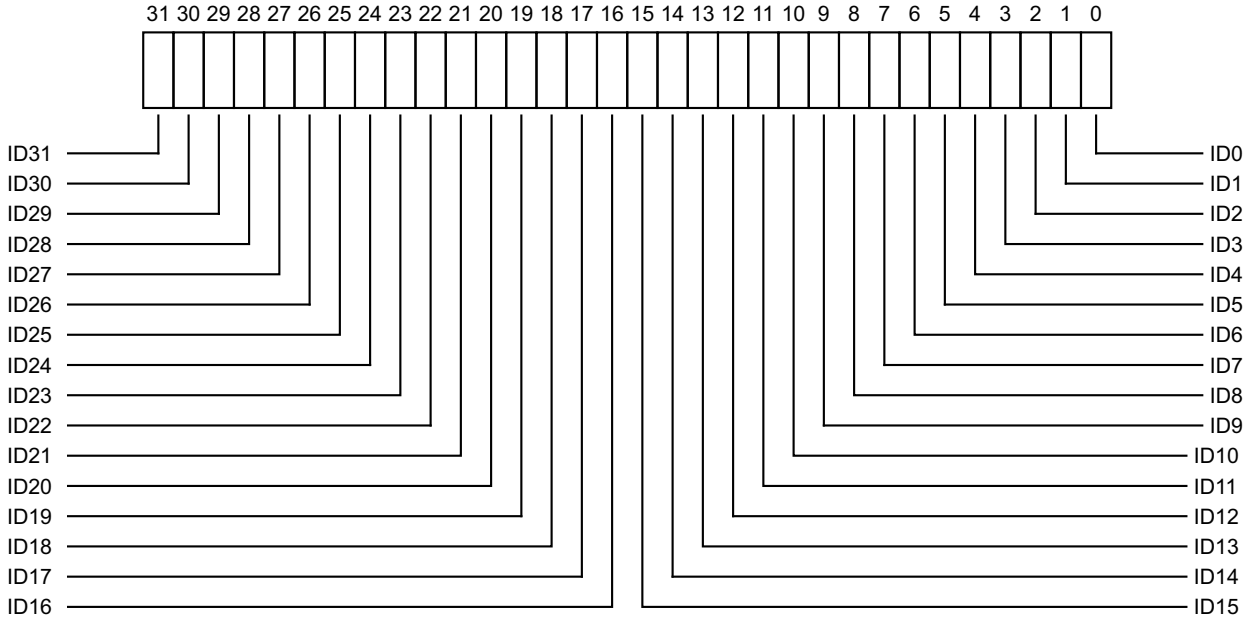
PMCEID1 is in the Core power domain.

Attributes

PMCEID1 is a 32-bit register.

Field descriptions

The PMCEID1 bit assignments are:



Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ID<n>, bit [n], for n = 0 to 31

ID[n] corresponds to common event (`0x0020` + n).

For each bit:

`0b0` The common event is not implemented, or not counted.

`0b1` The common event is implemented.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n> registers of that earlier version of the PMU architecture.

Accessing the PMCEID1:

Note

AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

PMCEID1 can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0xE24`	PMCEID1

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and AllowExternalPMUAccess() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.6 PMCEID2, Performance Monitors Common Event Identification register 2

The PMCEID2 characteristics are:

Purpose

Defines which common architectural events and common microarchitectural events are implemented, or counted, using PMU events in the range `0x4000` to `0x401F`.

When the value of a bit in the register is 1 the corresponding common event is implemented and counted.

Note

Arm recommends that, if a common event is never counted, the value of the corresponding register bit is 0.

For more information about the common events and the use of the PMCEIDn registers see *The PMU event number space and common events* on page D7-2721.

Configurations

External register PMCEID2[31:0] is architecturally mapped to AArch64 System register PMCEID0_EL0[63:32].

External register PMCEID2[63:32] is architecturally mapped to AArch32 System register PMCEID2[31:0].

PMCEID2 is in the Core power domain.

This register is present only when ARMv8.1-PMU is implemented. Otherwise, direct accesses to PMCEID2 are RES0.

Attributes

PMCEID2 is a 32-bit register.

Field descriptions

The PMCEID2 bit assignments are:

31 … 0
IDhi<n>, bit[n], for n = 0 to 31

IDhi<n>, bit [n], for n = 0 to 31

IDhi[n] corresponds to common event (`0x4000` + n).

For each bit:

`0b0` The common event is not implemented, or not counted.

`0b1` The common event is implemented.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n> registers of that earlier version of the PMU architecture.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the PMCEID2:

Note

AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

PMCEID2 can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0xE28`	PMCEID2

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and AllowExternalPMUAccess() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.7 PMCEID3, Performance Monitors Common Event Identification register 3

The PMCEID3 characteristics are:

Purpose

Defines which common architectural events and common microarchitectural events are implemented, or counted, using PMU events in the range `0x4020` to `0x403F`.

When the value of a bit in the register is 1 the corresponding common event is implemented and counted.

Note

Arm recommends that, if a common event is never counted, the value of the corresponding register bit is 0.

For more information about the common events and the use of the PMCEIDn registers see *The PMU event number space and common events* on page D7-2721.

Configurations

External register PMCEID3[31:0] is architecturally mapped to AArch64 System register PMCEID1_EL0[63:32].

External register PMCEID3[63:32] is architecturally mapped to AArch32 System register PMCEID3[31:0].

PMCEID3 is in the Core power domain.

This register is present only when ARMv8.1-PMU is implemented. Otherwise, direct accesses to PMCEID3 are RES0.

Attributes

PMCEID3 is a 32-bit register.

Field descriptions

The PMCEID3 bit assignments are:

31 … 0
IDhi<n>, bit[n], for n = 0 to 31

IDhi<n>, bit [n], for n = 0 to 31

IDhi[n] corresponds to common event (`0x4020` + n).

For each bit:

`0b0` The common event is not implemented, or not counted.

`0b1` The common event is implemented.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n> registers of that earlier version of the PMU architecture.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the PMCEID3:

Note

AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

PMCEID3 can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0xE2C`	PMCEID3

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and AllowExternalPMUAccess() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.8 PMCFGR, Performance Monitors Configuration Register

The PMCFGR characteristics are:

Purpose

Contains PMU-specific configuration data.

Configurations

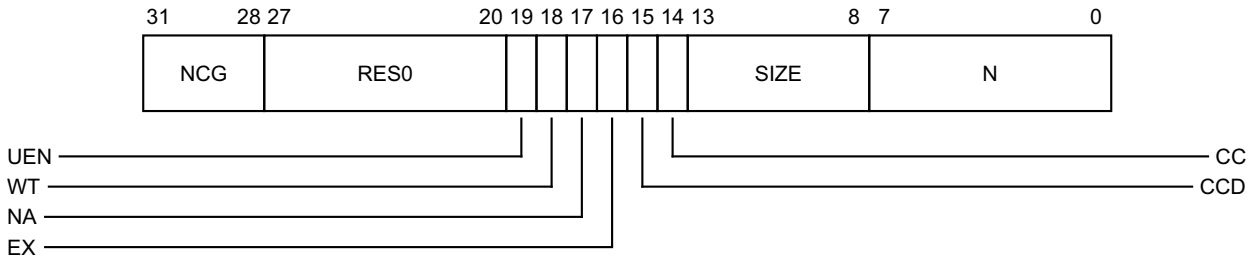
PMCFGR is in the Core power domain.

Attributes

PMCFGR is a 32-bit register.

Field descriptions

The PMCFGR bit assignments are:



NCG, bits [31:28]

This feature is not supported, so this field is RAZ.

Bits [27:20]

Reserved, RES0.

UEN, bit [19]

User-mode Enable Register supported. PMUSERENR_EL0 is not visible in the external debug interface, so this bit is RAZ.

WT, bit [18]

This feature is not supported, so this bit is RAZ.

NA, bit [17]

This feature is not supported, so this bit is RAZ.

EX, bit [16]

Export supported. Value is IMPLEMENTATION DEFINED.

`0b0` PMCR_EL0.X is RES0.

`0b1` PMCR_EL0.X is read/write.

CCD, bit [15]

Cycle counter has prescale.

This is RES1 if AArch32 is supported at any Exception level, and RAZ otherwise.

`0b0` PMCR_EL0.D is RES0.

`0b1` PMCR_EL0.D is read/write.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CC, bit [14]

Dedicated cycle counter (counter 31) supported. This bit is RAO.

SIZE, bits [13:8]

Size of counters, minus one. This field defines the size of the largest counter implemented by the Performance Monitors Unit.

From Armv8, the largest counter is 64-bits, so the value of this field is `0b111111`.

This field is used by software to determine the spacing of the counters in the memory-map. From Armv8, the counters are a doubleword-aligned addresses.

N, bits [7:0]

Number of counters implemented in addition to the cycle counter, PMCCNTR_EL0. The maximum number of event counters is 31.

`0x00` Only PMCCNTR_EL0 implemented.

`0x01` PMCCNTR_EL0 plus one event counter implemented.

and so on up to `0b00011111`, which indicates PMCCNTR_EL0 and 31 event counters implemented.

Accessing the PMCFGR:

Note

AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

PMCFGR can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0xE00`	PMCFGR

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and AllowExternalPMUAccess() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.9 PMCIDR0, Performance Monitors Component Identification Register 0

The PMCIDR0 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information see *About the Component Identification scheme* on page K2-7892.

Configurations

Implementation of this register is OPTIONAL.

If ARMv8.3-DoPD is implemented, this register is in the Core power domain. If ARMv8.3-DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

Attributes

PMCIDR0 is a 32-bit register.

Field descriptions

The PMCIDR0 bit assignments are:

31 … 8	7 … 0
RES0	PRMBL_0

Bits [31:8]

Reserved, RES0.

PRMBL_0, bits [7:0]

Preamble. Must read as `0x0D`.

Accessing the PMCIDR0:

PMCIDR0 can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0xFF0`	PMCIDR0

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.10 PMCIDR1, Performance Monitors Component Identification Register 1

The PMCIDR1 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information see *About the Component Identification scheme* on page K2-7892.

Configurations

Implementation of this register is OPTIONAL.

If ARMv8.3-DoPD is implemented, this register is in the Core power domain. If ARMv8.3-DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

Attributes

PMCIDR1 is a 32-bit register.

Field descriptions

The PMCIDR1 bit assignments are:

31 … 8	7 … 4	3 … 0
RES0	CLASS	PRMBL_1

Bits [31:8]

Reserved, RES0.

CLASS, bits [7:4]

Component class. Reads as `0x9`, debug component.

PRMBL_1, bits [3:0]

Preamble. RAZ.

Accessing the PMCIDR1:

PMCIDR1 can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0xFF4`	PMCIDR1

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.11 PMCIDR2, Performance Monitors Component Identification Register 2

The PMCIDR2 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information see *About the Component Identification scheme* on page K2-7892.

Configurations

Implementation of this register is OPTIONAL.

If ARMv8.3-DoPD is implemented, this register is in the Core power domain. If ARMv8.3-DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

Attributes

PMCIDR2 is a 32-bit register.

Field descriptions

The PMCIDR2 bit assignments are:

31 … 8	7 … 0
RES0	PRMBL_2

Bits [31:8]

Reserved, RES0.

PRMBL_2, bits [7:0]

Preamble. Must read as `0x05`.

Accessing the PMCIDR2:

PMCIDR2 can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0xFF8`	PMCIDR2

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.12 PMCIDR3, Performance Monitors Component Identification Register 3

The PMCIDR3 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information see *About the Component Identification scheme* on page K2-7892.

Configurations

Implementation of this register is OPTIONAL.

If ARMv8.3-DoPD is implemented, this register is in the Core power domain. If ARMv8.3-DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

Attributes

PMCIDR3 is a 32-bit register.

Field descriptions

The PMCIDR3 bit assignments are:

31 … 8	7 … 0
RES0	PRMBL_3

Bits [31:8]

Reserved, RES0.

PRMBL_3, bits [7:0]

Preamble. Must read as `0xB1`.

Accessing the PMCIDR3:

PMCIDR3 can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0xFFC`	PMCIDR3

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.13 PMCID1SR, CONTEXTIDR_EL1 Sample Register

The PMCID1SR characteristics are:

Purpose

Contains the sampled value of CONTEXTIDR_EL1, captured on reading PMPCSR[31:0].

Configurations

PMCID1SR is in the Core power domain.

This register is present only when ARMv8.2-PCSample is implemented. Otherwise, direct accesses to PMCID1SR are RES0.

Note

Before Armv8.2, the PC Sample-based Profiling Extension can be implemented in the external debug register space, as indicated by the value of EDDEVID.PCSample.

Attributes

PMCID1SR is a 32-bit register.

Field descriptions

The PMCID1SR bit assignments are:

31 … 0
CONTEXTIDR_EL1

CONTEXTIDR_EL1, bits [31:0]

Context ID. The value of CONTEXTIDR that is associated with the most recent PMPCSR sample. When the most recent PMPCSR sample was generated:

- If EL1 is using AArch64, then the Context ID is sampled from CONTEXTIDR_EL1.
- If EL1 is using AArch32, then the Context ID is sampled from CONTEXTIDR.
- If EL3 is implemented and is using AArch32, then CONTEXTIDR is a banked register and PMCID1SR samples the current banked copy of CONTEXTIDR for the Security state that is associated with the most recent PMPCSR sample.

Because the value written to PMCID1SR is an indirect read of CONTEXTIDR, it is CONSTRAINED UNPREDICTABLE whether PMCID1SR is set to the original or new value if PMPCSR samples:

- An instruction that writes to CONTEXTIDR.
- The next Context synchronization event.
- Any instruction executed between these two instructions.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a External debug reset, the value of this field is unchanged.
- On a Warm reset, the value of this field is unchanged.

Accessing the PMCID1SR:

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register UNKNOWN, see *Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN* on page H7-7046

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

PMCID1SR can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0x208`	PMCID1SR

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus() and !OSLockStatus() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

PMCID1SR can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0x228`	PMCID1SR

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus() and !OSLockStatus() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.14 PMCID2SR, CONTEXTIDR_EL2 Sample Register

The PMCID2SR characteristics are:

Purpose

Contains the sampled value of CONTEXTIDR_EL2, captured on reading PMPCSR[31:0].

Configurations

PMCID2SR is in the Core power domain.

This register is present only when ARMv8.2-PCSample is implemented and EL2 is implemented. Otherwise, direct accesses to PMCID2SR are RES0.

Note

If ARMv8.2-PCSample is not implemented, the PC Sample-based Profiling Extension can be implemented in the external debug register space, as indicated by the value of EDDEVID.PCSample.

Attributes

PMCID2SR is a 32-bit register.

Field descriptions

The PMCID2SR bit assignments are:

31 … 0
CONTEXTIDR_EL2

CONTEXTIDR_EL2, bits [31:0]

Context ID. The value of CONTEXTIDR_EL2 that is associated with the most recent PMPCSR sample. When the most recent PMPCSR sample was generated:

- If EL2 is using AArch64, then this field is set to the Context ID sampled from CONTEXTIDR_EL2.
- If EL2 is using AArch32, then this field is set to an UNKNOWN value.

Because the value written to PMCID2SR is an indirect read of CONTEXTIDR_EL2, it is CONSTRAINED UNPREDICTABLE whether PMCID2SR is set to the original or new value if PMPCSR samples:

- An instruction that writes to CONTEXTIDR_EL2.
- The next Context synchronization event.
- Any instruction executed between these two instructions.

The following resets apply:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
- On a External debug reset, the value of this field is unchanged.
- On a Warm reset, the value of this field is unchanged.

Accessing the PMCID2SR:

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register UNKNOWN, see *Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN* on page H7-7046

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

PMCID2SR can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0x22C`	PMCID2SR

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus() and !OSLockStatus() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.15 PMCNTENCLR_EL0, Performance Monitors Count Enable Clear register

The PMCNTENCLR_EL0 characteristics are:

Purpose

Disables the Cycle Count Register, PMCCNTR_EL0, and any implemented event counters PMEVCNTR<n>. Reading this register shows which counters are enabled.

Configurations

External register PMCNTENCLR_EL0[31:0] is architecturally mapped to AArch64 System register PMCNTENCLR_EL0[31:0].

External register PMCNTENCLR_EL0[31:0] is architecturally mapped to AArch32 System register PMCNTENCLR[31:0].

PMCNTENCLR_EL0 is in the Core power domain.

Attributes

PMCNTENCLR_EL0 is a 32-bit register.

Field descriptions

The PMCNTENCLR_EL0 bit assignments are:

31	30 … 0
C	P<n>, bit [n]

C, bit [31]

PMCCNTR_EL0 disable bit. Disables the cycle counter register. Possible values are:

`0b0` When read, means the cycle counter is disabled. When written, has no effect.

`0b1` When read, means the cycle counter is enabled. When written, disables the cycle counter.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

P<n>, bit [n], for n = 0 to 30

Event counter disable bit for PMEVCNTR<n>_EL0.

If PMCFGR.N is less than 31, bits [30:PMCFGR.N] are RAZ/WI.

`0b0` When read, means that PMEVCNTR<n>_EL0 is disabled. When written, has no effect.

`0b1` When read, means that PMEVCNTR<n>_EL0 is enabled. When written, disables PMEVCNTR<n>_EL0.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMCNTENCLR_EL0:

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

PMCNTENCLR_EL0 can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0xC20`	PMCNTENCLR_EL0

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.16 PMCNTENSET_EL0, Performance Monitors Count Enable Set register

The PMCNTENSET_EL0 characteristics are:

Purpose

Enables the Cycle Count Register, PMCCNTR_EL0, and any implemented event counters PMEVCNTR<n>. Reading this register shows which counters are enabled.

Configurations

External register PMCNTENSET_EL0[31:0] is architecturally mapped to AArch64 System register PMCNTENSET_EL0[31:0].

External register PMCNTENSET_EL0[31:0] is architecturally mapped to AArch32 System register PMCNTENSET[31:0].

PMCNTENSET_EL0 is in the Core power domain.

Attributes

PMCNTENSET_EL0 is a 32-bit register.

Field descriptions

The PMCNTENSET_EL0 bit assignments are:

31	30 … 0
C	P<n>, bit [n]

C, bit [31]

PMCCNTR_EL0 enable bit. Enables the cycle counter register. Possible values are:

`0b0` When read, means the cycle counter is disabled. When written, has no effect.

`0b1` When read, means the cycle counter is enabled. When written, enables the cycle counter.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

P<n>, bit [n], for n = 0 to 30

Event counter enable bit for PMEVCNTR<n>_EL0.

If PMCFGR.N is less than 31, bits [30:PMCFGR.N] are RAZ/WI.

`0b0` When read, means that PMEVCNTR<n>_EL0 is disabled. When written, has no effect.

`0b1` When read, means that PMEVCNTR<n>_EL0 event counter is enabled. When written, enables PMEVCNTR<n>_EL0.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMCNTENSET_EL0:

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

PMCNTENSET_EL0 can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0xC00`	PMCNTENSET_EL0

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.17 PMCR_EL0, Performance Monitors Control Register

The PMCR_EL0 characteristics are:

Purpose

Provides details of the Performance Monitors implementation, including the number of counters implemented, and configures and controls the counters.

Configurations

External register PMCR_EL0[7:0] is architecturally mapped to AArch32 System register PMCR[7:0].

External register PMCR_EL0[7:0] is architecturally mapped to AArch64 System register PMCR_EL0[7:0].

PMCR_EL0 is in the Core power domain.

This register is only partially mapped to the internal PMCR System register. An external agent must use other means to discover the information held in PMCR[31:11], such as accessing PMCFGR and the ID registers.

Attributes

PMCR_EL0 is a 32-bit register.

Field descriptions

The PMCR_EL0 bit assignments are:

31 … 11	10 … 8	7	6	5	4	3	2	1	0
RAZ/WI	RES0	LP	LC	DP	X	D	C	P	E

Bits [31:11]

Reserved, RAZ/WI.

Hardware must implement this field as RAZ/WI. Software must not rely on the register reading as zero, and must use a read-modify-write sequence to write to the register.

Bits [10:8]

Reserved, RES0.

LP, bit [7]

When ARMv8.5-PMU is implemented:

Long event counter enable. Determines when unsigned overflow is recorded by a counter overflow bit.

`0b0` Event counter overflow on increment that causes unsigned overflow of PMEVCNTR<n>_EL0[31:0].

`0b1` Event counter overflow on increment that causes unsigned overflow of PMEVCNTR<n>_EL0[63:0].

If EL2 is implemented and MDCR_EL2.HPMN is less than PMCR_EL0.N, this bit does not affect the operation of event counters in the range [MDCR_EL2.HPMN:(PMCR_EL0.N-1)].

If EL2 is implemented and HDCR.HPMN is less than PMCR_EL0.N, this bit does not affect the operation of event counters in the range [HDCR.HPMN..(PMCR_EL0.N-1)].

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

The effect of MDCR_EL2.HPMN or HDCR.HPMN on the operation of this bit always applies if EL2 is implemented, at all Exception levels including EL2 and EL3, and regardless of whether EL2 is enabled in the current Security state. For more information, see the description of MDCR_EL2.HPMN or HDCR.HPMN.

If the highest implemented Exception level is using AArch32, it is IMPLEMENTATION DEFINED whether this bit is RW or RAZ/WI.

Otherwise:

Reserved, RES0.

LC, bit [6]

When AArch32 is supported at any Exception level:

Long cycle counter enable. Determines when unsigned overflow is recorded by the cycle counter overflow bit.

`0b0` Cycle counter overflow on increment that causes unsigned overflow of PMCCNTR_EL0[31:0].

`0b1` Cycle counter overflow on increment that causes unsigned overflow of PMCCNTR_EL0[63:0].

Arm deprecates use of PMCR_EL0.LC = 0.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

DP, bit [5]

When EL3 is implemented or (ARMv8.1-PMU is implemented and EL2 is implemented):

Disable cycle counter when event counting is prohibited. The possible values of this bit are:

`0b0` Cycle counting by PMCCNTR_EL0 is not affected by this bit.

`0b1` When event counting for counters in the range [0..(MDCR_EL2.HPMN-1)] is prohibited, cycle counting by PMCCNTR_EL0 is disabled.

For more information about the interaction between the Performance Monitors and EL3, see *Prohibiting event counting* on page D7-2709.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field it resets to:

- A value that is architecturally UNKNOWN if the reset is into an Exception level that is using AArch64.
- 0 if the reset is into an Exception level that is using AArch32.

Otherwise:

Reserved, RES0.

X, bit [4]

When the implementation includes a PMU event export bus:

Enable export of events in an IMPLEMENTATION DEFINED PMU event export bus.

`0b0` Do not export events.

`0b1` Export events where not prohibited.

This field enables the exporting of events over an IMPLEMENTATION DEFINED PMU event export bus to another device, for example to an OPTIONAL PE trace unit.

No events are exported when counting is prohibited.

This field does not affect the generation of Performance Monitors overflow interrupt requests or signaling to a cross-trigger interface (CTI) that can be implemented as signals exported from the PE.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When this register has an architecturally-defined reset value, if this field is implemented as an RW field it resets to:

- A value that is architecturally UNKNOWN if the reset is into an Exception level that is using AArch64.
- 0 if the reset is into an Exception level that is using AArch32.

Otherwise:

Reserved, RAZ/WI.

D, bit [3]

When AArch32 is supported at any Exception level:

Clock divider.

`0b0` When enabled, PMCCNTR_EL0 counts every clock cycle.

`0b1` When enabled, PMCCNTR_EL0 counts once every 64 clock cycles.

If PMCR_EL0.LC == 1, this bit is ignored and the cycle counter counts every clock cycle.

Arm deprecates use of PMCR_EL0.D = 1.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field it resets to:

- A value that is architecturally UNKNOWN if the reset is into an Exception level that is using AArch64.
- 0 if the reset is into an Exception level that is using AArch32.

Otherwise:

Reserved, RES0.

C, bit [2]

Cycle counter reset. The effects of writing to this bit are:

`0b0` No action.

`0b1` Reset PMCCNTR_EL0 to zero.

This bit is always RAZ.

Note

Resetting PMCCNTR_EL0 does not change the cycle counter overflow bit.

Access to this field is WO.

P, bit [1]

Event counter reset. The effects of writing to this bit are:

`0b0` No action.

`0b1` Reset all event counters, not including PMCCNTR_EL0, to zero.

This bit is always RAZ.

Note

Resetting the event counters does not change the event counter overflow bits.

If ARMv8.5-PMU is implemented, the value of MDCR_EL2.HLP, or PMCR_EL0.LP is ignored and bits [63:0] of all event counters are reset.

Access to this field is WO.

E, bit [0]

Enable.

`0b0` All event counters in the range [0..(PMN-1)] and PMCCNTR_EL0, are disabled.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` All event counters in the range [0..(PMN-1)] and PMCCNTR_EL0, are enabled by PMCNTENSET_EL0.

If EL2 is implemented then:

- If EL2 is using AArch32, PMN is HDCR.HPMN.
- If EL2 is using AArch64, PMN is MDCR_EL2.HPMN.
- If PMN is less than PMCR_EL0.N, this bit does not affect the operation of event counters in the range [PMN..(PMCR_EL0.N-1)].

If EL2 is not implemented, PMN is PMCR_EL0.N.

Note

The effect of the following fields on the operation of this bit applies if EL2 is implemented regardless of whether EL2 is enabled in the current Security state:

- HDCR.HPMN. See the description of HDCR.HPMN for more information.
- MDCR_EL2.HPMN. See the description of MDCR_EL2.HPMN for more information.

On a Warm reset, this field resets to `0`.

Accessing the PMCR_EL0:

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

PMCR_EL0 can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0xE04`	PMCR_EL0

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.18 PMDEVAFF0, Performance Monitors Device Affinity register 0

The PMDEVAFF0 characteristics are:

Purpose

Copy of the low half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the Performance Monitor component relates to.

Configurations

If ARMv8.3-DoPD is implemented, this register is in the Core power domain. If ARMv8.3-DoPD is not implemented, this register is in the Debug power domain.

This register is required if the external interface to the PMU is implemented.

Attributes

PMDEVAFF0 is a 32-bit register.

Field descriptions

The PMDEVAFF0 bit assignments are:

31 … 0
MPIDR_EL1lo

MPIDR_EL1lo, bits [31:0]

MPIDR_EL1 low half. Read-only copy of the low half of MPIDR_EL1, as seen from the highest implemented Exception level.

Accessing the PMDEVAFF0:

PMDEVAFF0 can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0xFA8`	PMDEVAFF0

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.19 PMDEVAFF1, Performance Monitors Device Affinity register 1

The PMDEVAFF1 characteristics are:

Purpose

Copy of the high half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the Performance Monitor component relates to.

Configurations

If ARMv8.3-DoPD is implemented, this register is in the Core power domain. If ARMv8.3-DoPD is not implemented, this register is in the Debug power domain.

This register is required if the external interface to the PMU is implemented.

Attributes

PMDEVAFF1 is a 32-bit register.

Field descriptions

The PMDEVAFF1 bit assignments are:

31	0
MPIDR_EL1hi	

MPIDR_EL1hi, bits [31:0]

MPIDR_EL1 high half. Read-only copy of the high half of MPIDR_EL1, as seen from the highest implemented Exception level.

Accessing the PMDEVAFF1:

PMDEVAFF1 can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0xFAC`	PMDEVAFF1

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.20 PMDEVARCH, Performance Monitors Device Architecture register

The PMDEVARCH characteristics are:

Purpose

Identifies the programmers' model architecture of the Performance Monitor component.

Configurations

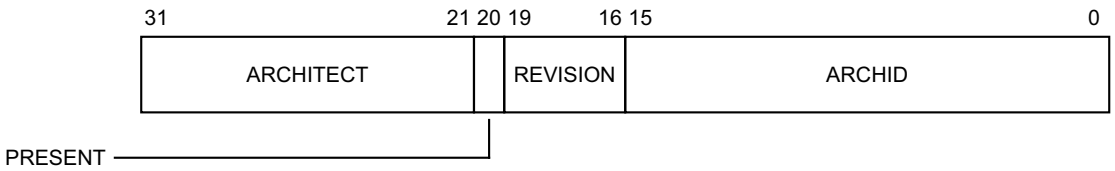
If ARMv8.3-DoPD is implemented, this register is in the Core power domain. If ARMv8.3-DoPD is not implemented, this register is in the Debug power domain.

Attributes

PMDEVARCH is a 32-bit register.

Field descriptions

The PMDEVARCH bit assignments are:



ARCHITECT, bits [31:21]

Defines the architecture of the component. For Performance Monitors, this is Arm Limited.

Bits [31:28] are the JEP106 continuation code, `0x4`.

Bits [27:21] are the JEP106 ID code, `0x3B`.

PRESENT, bit [20]

When set to 1, indicates that the DEVARCH is present.

This field is 1 in Armv8.

REVISION, bits [19:16]

Defines the architecture revision. For architectures defined by Arm this is the minor revision.

For Performance Monitors, the revision defined by Armv8 is `0x0`.

All other values are reserved.

ARCHID, bits [15:0]

Defines this part to be an Armv8 debug component. For architectures defined by Arm this is further subdivided.

For Performance Monitors:

- Bits [15:12] are the architecture version, `0x2`.
- Bits [11:0] are the architecture part number, `0xA16`.

This corresponds to Performance Monitors architecture version PMUv3.

Note

The PMUv3 memory-mapped programmers' model can be used by devices other than Armv8 processors. Software must determine whether the PMU is attached to an Armv8 processor by using the PMDEVAFF0 and PMDEVAFF1 registers to discover the affinity of the PMU to any Armv8 processors.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the PMDEVARCH:

PMDEVARCH can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0xFBC`	PMDEVARCH

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.21 PMDEVID, Performance Monitors Device ID register

The PMDEVID characteristics are:

Purpose

Provides information about features of the Performance Monitors implementation.

Configurations

If ARMv8.3-DoPD is implemented, this register is in the Core power domain. If ARMv8.3-DoPD is not implemented, this register is in the Debug power domain.

This register is required from Armv8.2 and in any implementation that includes ARMv8.2-PCSample. Otherwise, its location is RES0.

Note

Before Armv8.2, the PC Sample-based Profiling Extension can be implemented in the external debug register space, as indicated by the value of EDDEVID.PCSample.

Attributes

PMDEVID is a 32-bit register.

Field descriptions

The PMDEVID bit assignments are:

31 … 4	3 … 0
RES0	PCSample

Bits [31:4]

Reserved, RES0.

PCSample, bits [3:0]

Indicates the level of PC Sample-based Profiling support using Performance Monitors registers. Permitted values of this field are:

`0b0000` PC Sample-based Profiling Extension is not implemented in the Performance Monitors register space.

`0b0001` PC Sample-based Profiling Extension is implemented in the Performance Monitors register space.

All other values are reserved.

ARMv8.2-PCSample implements the functionality identified by the value `0b0001`.

Accessing the PMDEVID:

PMDEVID can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0xFC8`	PMDEVID

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.22 PMDEVTYPE, Performance Monitors Device Type register

The PMDEVTYPE characteristics are:

Purpose

Indicates to a debugger that this component is part of a PEs performance monitor interface.

Configurations

Implementation of this register is OPTIONAL.

If ARMv8.3-DoPD is implemented, this register is in the Core power domain. If ARMv8.3-DoPD is not implemented, this register is in the Debug power domain.

Attributes

PMDEVTYPE is a 32-bit register.

Field descriptions

The PMDEVTYPE bit assignments are:

31 … 8	7 … 4	3 … 0
RES0	SUB	MAJOR

Bits [31:8]

Reserved, RES0.

SUB, bits [7:4]

Subtype. Must read as `0x1` to indicate this is a component within a PE.

MAJOR, bits [3:0]

Major type. Must read as `0x6` to indicate this is a performance monitor component.

Accessing the PMDEVTYPE:

PMDEVTYPE can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0xFCC`	PMDEVTYPE

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.23 PMEVCNTR<n>_EL0, Performance Monitors Event Count Registers, n = 0 - 30

The PMEVCNTR<n>_EL0 characteristics are:

Purpose

Holds event counter n, which counts events, where n is 0 to 30.

Configurations

External register PMEVCNTR<n>_EL0[31:0] is architecturally mapped to AArch64 System register PMEVCNTR<n>_EL0[31:0].

External register PMEVCNTR<n>_EL0[31:0] is architecturally mapped to AArch32 System register PMEVCNTR<n>[31:0].

PMEVCNTR<n>_EL0 is in the Core power domain.

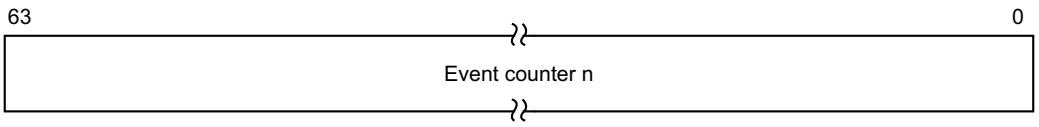
Attributes

PMEVCNTR<n>_EL0 is a 64-bit register.

Field descriptions

The PMEVCNTR<n>_EL0 bit assignments are:

When ARMv8.5-PMU is implemented:



Bits [63:0]

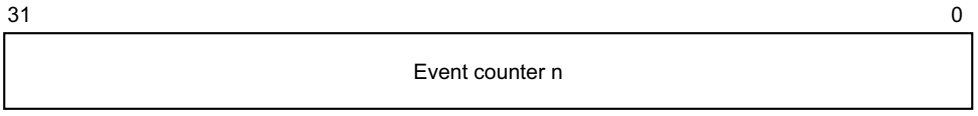
Event counter n. Value of event counter n, where n is the number of this register and is a number from 0 to 30.

If the highest implemented Exception level is using AArch32, the optional external interface to the performance monitors is implemented, and the PMCR.LP and HDCR.HLP bits are RAZ/WI, then locations in the external interface to the performance monitors that map to PMEVCNTR<n>_EL0[63:32] return UNKNOWN values on reads.

If the implementation does not support AArch64 at any Exception level, bits [63:32] of the event counters are not required to be implemented.

This field resets to an architecturally UNKNOWN value.

Otherwise:



Bits [31:0]

Event counter n. Value of event counter n, where n is the number of this register and is a number from 0 to 30.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the PMEVCNTR<n>_EL0:

External accesses to the performance monitors ignore PMUSERENR_EL0 and, if implemented, MDCR_EL2.{TPM, TPMCR, HPMN} and MDCR_EL3.TPM. This means that all counters are accessible regardless of the current Exception level or privilege of the access.

If ARMv8.5-PMU is not implemented, when IsCorePowered(), DoubleLockStatus(), OSLockStatus() or !AllowExternalPMUAccess(), 32-bit accesses to `0x004`+8×n have a CONSTRAINED UNPREDICTABLE behavior.

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

PMEVCNTR<n>_EL0 can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0x000` + 8n	PMEVCNTR<n>_EL0

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.24 PMEVTYPER<n>_EL0, Performance Monitors Event Type Registers, n = 0 - 30

The PMEVTYPER<n>_EL0 characteristics are:

Purpose

Configures event counter n, where n is 0 to 30.

Configurations

External register PMEVTYPER<n>_EL0[31:0] is architecturally mapped to AArch64 System register PMEVTYPER<n>_EL0[31:0].

External register PMEVTYPER<n>_EL0[31:0] is architecturally mapped to AArch32 System register PMEVTYPER<n>[31:0].

PMEVTYPER<n>_EL0 is in the Core power domain.

If event counter n is not implemented then accesses to this register are:

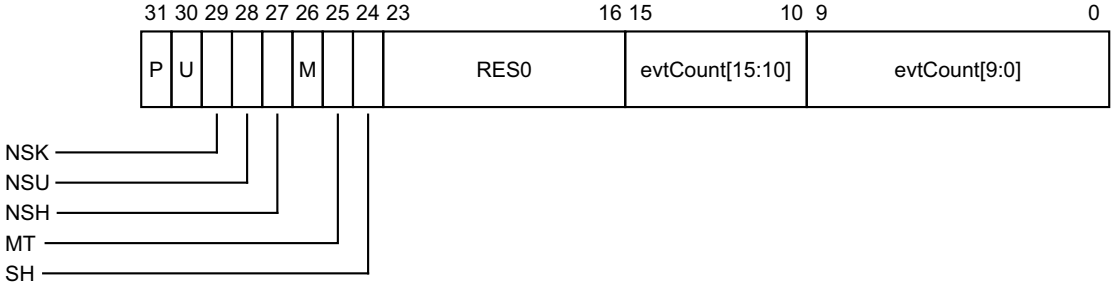
- RES0 when IsCorePowered() && !DoubleLockStatus() && !OSLockStatus() && AllowExternalPMUAccess().
- A CONSTRAINED UNPREDICTABLE choice of RES0 or ERROR otherwise.

Attributes

PMEVTYPER<n>_EL0 is a 32-bit register.

Field descriptions

The PMEVTYPER<n>_EL0 bit assignments are:



P, bit [31]

Privileged filtering bit. Controls counting in EL1.

If EL3 is implemented, then counting in Non-secure EL1 is further controlled by the PMEVTYPER<n>_EL0.NSK bit.

`0b0` Count events in EL1.

`0b1` Do not count events in EL1.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

U, bit [30]

User filtering bit. Controls counting in EL0.

If EL3 is implemented, then counting in Non-secure EL0 is further controlled by the PMEVTYPER<n>_EL0.NSU bit.

`0b0` Count events in EL0.

`0b1` Do not count events in EL0.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

NSK, bit [29]

When EL3 is implemented:

Non-secure EL1 (kernel) modes filtering bit. Controls counting in Non-secure EL1.

If the value of this bit is equal to the value of the PMEVTYPER<n>_EL0.P bit, events in Non-secure EL1 are counted.

Otherwise, events in Non-secure EL1 are not counted.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSU, bit [28]

When EL3 is implemented:

Non-secure EL0 (Unprivileged) filtering bit. Controls counting in Non-secure EL0.

If the value of this bit is equal to the value of the PMEVTYPER<n>_EL0.U bit, events in Non-secure EL0 are counted.

Otherwise, events in Non-secure EL0 are not counted.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSH, bit [27]

When EL2 is implemented:

EL2 (Hypervisor) filtering bit. Controls counting in EL2.

If Secure EL2 is implemented, counting in Secure EL2 is further controlled by the PMEVTYPER<n>_EL0.SH bit.

`0b0` Do not count events in EL2.

`0b1` Count events in EL2.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

M, bit [26]

When EL3 is implemented:

Secure EL3 filtering bit.

If the value of this bit is equal to the value of the PMEVTYPER<n>_EL0.P bit, events in Secure EL3 are counted.

Otherwise, events in Secure EL3 are not counted.

Most applications can ignore this field and set its value to `0b0`.

Note

This field is not visible in the AArch32 PMEVTYPER<n> System register.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

MT, bit [25]

When (ARMv8.6-MTPMU is implemented and enabled) or an IMPLEMENTATION DEFINED multi-threaded PMU Extension is implemented:

Multithreading.

`0b0` Count events only on controlling PE.

`0b1` Count events from any PE with the same affinity at level 1 and above as this PE.

Note

- When the lowest level of affinity consists of logical PEs that are implemented using a multi-threading type approach, an implementation is described as multi-threaded. That is, the performance of PEs at the lowest affinity level is highly interdependent.
- Events from a different thread of a multithreaded implementation are not Attributable to the thread counting the event.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SH, bit [24]

When ARMv8.4-SecEL2 is implemented:

Secure EL2 filtering.

If the value of this bit is not equal to the value of the PMEVTYPER<n>_EL0.NSH bit, events in Secure EL2 are counted.

Otherwise, events in Secure EL2 are not counted.

Note

This field is not visible in the AArch32 PMEVTYPER<n> System register.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [23:16]

Reserved, RES0.

evtCount[15:10], bits [15:10]

When ARMv8.1-PMU is implemented:

Extension to evtCount[9:0]. See evtCount[9:0] for more details.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

evtCount[9:0], bits [9:0]

Event to count. The event number of the event that is counted by event counter PMEVCNTR<n>_EL0.

Software must program this field with an event that is supported by the PE being programmed.

The ranges of event numbers allocated to each type of event are shown in Table D7-7 on page D7-2721.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If evtCount is programmed to an event that is reserved or not supported by the PE, the behavior depends on the value written:

- For the range `0x0000` to `0x003F`, no events are counted, and the value returned by a direct or external read of the evtCount field is the value written to the field.
- If 16-bit evtCount is implemented, for the range `0x4000` to `0x403F`, no events are counted, and the value returned by a direct or external read of the evtCount field is the value written to the field.
- For IMPLEMENTATION DEFINED events, it is UNPREDICTABLE what event, if any, is counted, and the value returned by a direct or external read of the evtCount field is UNKNOWN.

Note

UNPREDICTABLE means the event must not expose privileged information.

Arm recommends that the behavior across a family of implementations is defined such that if a given implementation does not include an event from a set of common IMPLEMENTATION DEFINED events, then no event is counted and the value read back on evtCount is the value written.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMEVTYPER<n>_EL0:

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

PMEVTYPER<n>_EL0 can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0x400` + 4n	PMEVTYPER<n>_EL0

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.25 PMINTENCLR_EL1, Performance Monitors Interrupt Enable Clear register

The PMINTENCLR_EL1 characteristics are:

Purpose

Disables the generation of interrupt requests on overflows from the Cycle Count Register, PMCCNTR_EL0, and the event counters PMEVCNTR<n>_EL0. Reading the register shows which overflow interrupt requests are enabled.

Configurations

External register PMINTENCLR_EL1[31:0] is architecturally mapped to AArch64 System register PMINTENCLR_EL1[31:0].

External register PMINTENCLR_EL1[31:0] is architecturally mapped to AArch32 System register PMINTENCLR[31:0].

PMINTENCLR_EL1 is in the Core power domain.

Attributes

PMINTENCLR_EL1 is a 32-bit register.

Field descriptions

The PMINTENCLR_EL1 bit assignments are:

31	30 … 0
C	P<n>, bit [n]

C, bit [31]

PMCCNTR_EL0 overflow interrupt request disable bit. Possible values are:

`0b0` When read, means the cycle counter overflow interrupt request is disabled. When written, has no effect.

`0b1` When read, means the cycle counter overflow interrupt request is enabled. When written, disables the cycle count overflow interrupt request.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

P<n>, bit [n], for n = 0 to 30

Event counter overflow interrupt request disable bit for PMEVCNTR<n>_EL0.

If PMCFGR.N is less than 31, bits [30:PMCFGR.N] are RAZ/WI.

`0b0` When read, means that the PMEVCNTR<n>_EL0 event counter interrupt request is disabled. When written, has no effect.

`0b1` When read, means that the PMEVCNTR<n>_EL0 event counter interrupt request is enabled. When written, disables the PMEVCNTR<n>_EL0 interrupt request.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMINTENCLR_EL1:

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

PMINTENCLR_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0xC60`	PMINTENCLR_EL1

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.26 PMINTENSET_EL1, Performance Monitors Interrupt Enable Set register

The PMINTENSET_EL1 characteristics are:

Purpose

Enables the generation of interrupt requests on overflows from the Cycle Count Register, PMCCNTR_EL0, and the event counters PMEVCNTR<n>_EL0. Reading the register shows which overflow interrupt requests are enabled.

Configurations

External register PMINTENSET_EL1[31:0] is architecturally mapped to AArch64 System register PMINTENSET_EL1[31:0].

External register PMINTENSET_EL1[31:0] is architecturally mapped to AArch32 System register PMINTENSET[31:0].

PMINTENSET_EL1 is in the Core power domain.

Attributes

PMINTENSET_EL1 is a 32-bit register.

Field descriptions

The PMINTENSET_EL1 bit assignments are:

31	30 … 0
C	P<n>, bit [n]

C, bit [31]

PMCCNTR_EL0 overflow interrupt request enable bit. Possible values are:

`0b0` When read, means the cycle counter overflow interrupt request is disabled. When written, has no effect.

`0b1` When read, means the cycle counter overflow interrupt request is enabled. When written, enables the cycle count overflow interrupt request.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

P<n>, bit [n], for n = 0 to 30

Event counter overflow interrupt request enable bit for PMEVCNTR<n>_EL0.

If PMCFGR.N is less than 31, bits [30:PMCFGR.N] are RAZ/WI.

`0b0` When read, means that the PMEVCNTR<n>_EL0 event counter interrupt request is disabled. When written, has no effect.

`0b1` When read, means that the PMEVCNTR<n>_EL0 event counter interrupt request is enabled. When written, enables the PMEVCNTR<n>_EL0 interrupt request.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMINTENSET_EL1:

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

PMINTENSET_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0xC40`	PMINTENSET_EL1

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.27 PMITCTRL, Performance Monitors Integration mode Control register

The PMITCTRL characteristics are:

Purpose

Enables the Performance Monitors to switch from default mode into integration mode, where test software can control directly the inputs and outputs of the PE, for integration testing or topology detection.

Configurations

It is IMPLEMENTATION DEFINED whether PMITCTRL is implemented in the Core power domain or in the Debug power domain.

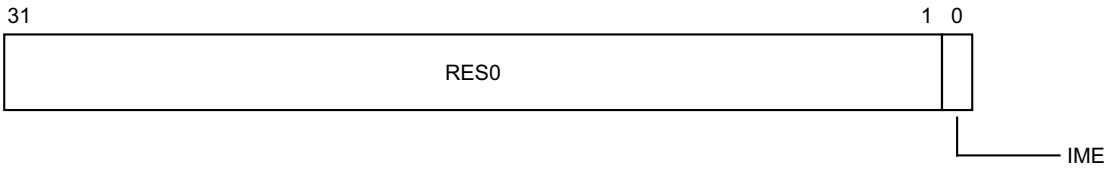
Implementation of this register is OPTIONAL.

Attributes

PMITCTRL is a 32-bit register.

Field descriptions

The PMITCTRL bit assignments are:



Bits [31:1]

Reserved, RES0.

IME, bit [0]

Integration mode enable. When IME == 1, the device reverts to an integration mode to enable integration testing or topology detection. The integration mode behavior is IMPLEMENTATION DEFINED.

`0b0` Normal operation.

`0b1` Integration mode enabled.

On a Implementation reset, this field resets to `0`.

Accessing the PMITCTRL:

PMITCTRL can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0xF00`	PMITCTRL

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register are IMPDEF.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.28 PMLAR, Performance Monitors Lock Access Register

The PMLAR characteristics are:

Purpose

Allows or disallows access to the Performance Monitors registers through a memory-mapped interface.

The optional Software Lock provides a lock to prevent memory-mapped writes to the Performance Monitors registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the Performance Monitors registers. It does not, and cannot, prevent all accidental or malicious damage.

Configurations

If ARMv8.3-DoPD is implemented, then ARMv8.0-SoftwareLock is not implemented by the architecturally-defined debug components of the PE in the Core power domain.

If ARMv8.3-DoPD is not implemented, then this register is in the Debug power domain.

Software uses PMLAR to set or clear the lock, and PMLSR to check the current status of the lock.

Attributes

PMLAR is a 32-bit register.

Field descriptions

The PMLAR bit assignments are:

When the Software Lock is implemented.:

31	0
KEY	

KEY, bits [31:0]

Lock Access control. Writing the key value `0xC5ACCE55` to this field unlocks the lock, enabling write accesses to this component's registers through a memory-mapped interface.

Writing any other value to this register locks the lock, disabling write accesses to this component's registers through a memory mapped interface.

Otherwise:

31	0
RES0	

Otherwise

Bits [31:0]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the PMLAR:

PMLAR can be accessed through its memory-mapped interface:

Component	Offset	Instance
PMU	`0xFB0`	PMLAR

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are WO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.29 PMLSR, Performance Monitors Lock Status Register

The PMLSR characteristics are:

Purpose

Indicates the current status of the software lock for Performance Monitors registers.

The optional Software Lock provides a lock to prevent memory-mapped writes to the Performance Monitors registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the Performance Monitors registers. It does not, and cannot, prevent all accidental or malicious damage.

Configurations

If ARMv8.3-DoPD is implemented, then ARMv8.0-SoftwareLock is not implemented by the architecturally-defined debug components of the PE in the Core power domain.

If ARMv8.3-DoPD is not implemented, then this register is in the Debug power domain.

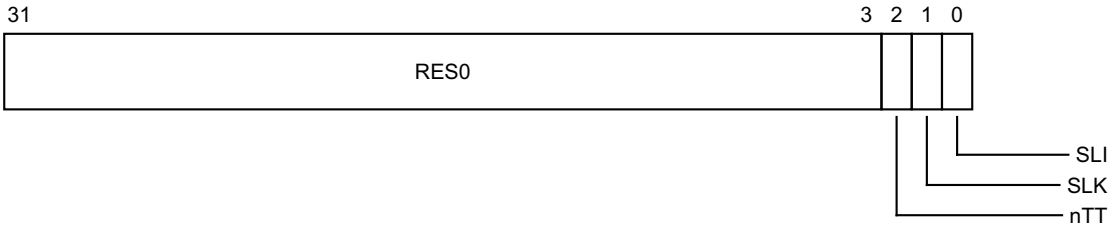
Software uses PMLAR to set or clear the lock, and PMLSR to check the current status of the lock.

Attributes

PMLSR is a 32-bit register.

Field descriptions

The PMLSR bit assignments are:



Bits [31:3]

Reserved, RES0.

nTT, bit [2]

Not thirty-two bit access required. RAZ.

SLK, bit [1]

When the Software Lock is implemented.:

Software Lock status for this component. For an access to LSR that is not a memory-mapped access, or when the Software Lock is not implemented, this field is RES0.

For memory-mapped accesses when the software lock is implemented, possible values of this field are:

`0b0` Lock clear. Writes are permitted to this component's registers.

`0b1` Lock set. Writes to this component's registers are ignored, and reads have no side effects.

The following resets apply:

- If Armv8.3-DoPD is implemented, this register is reset by Cold reset and not affected by External debug reset. If Armv8.3-DoPD is not implemented, this register is reset by External debug reset and not affected by Cold reset.
- On a Cold reset, this field resets to 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

Reserved, RAZ.

SLI, bit [0]

Software Lock implemented. For an access to LSR that is not a memory-mapped access, this field is RAZ. For memory-mapped accesses, the value of this field is IMPLEMENTATION DEFINED. Permitted values are:

`0b0` Software Lock not implemented or not memory-mapped access.

`0b1` Software Lock implemented and memory-mapped access.

Accessing the PMLSR:

PMLSR can be accessed through its memory-mapped interface:

Component	Offset	Instance
PMU	`0xFB4`	PMLSR

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.30 PMMIR, Performance Monitors Machine Identification Register

The PMMIR characteristics are:

Purpose

Describes Performance Monitors parameters specific to the implementation.

Configurations

PMMIR is in the Core power domain.

This register is present only when ARMv8.4-PMU is implemented. Otherwise, direct accesses to PMMIR are RES0.

Attributes

PMMIR is a 32-bit register.

Field descriptions

The PMMIR bit assignments are:

31 … 8	7 … 0
RES0	SLOTS

Bits [31:8]

Reserved, RES0.

SLOTS, bits [7:0]

Operation width. The largest value by which the STALL_SLOT event might increment by in a single cycle. If the STALL_SLOT event is implemented, this field must not be zero.

Accessing the PMMIR:

If the Core power domain is off or in a low-power state, access on this interface returns an Error.

PMMIR can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0xE40`	PMMIR

This interface is accessible as follows:

- When !IsCorePowered(), or DoubleLockStatus(), or OSLockStatus() or !AllowExternalPMUAccess() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.31 PMOVSCLR_EL0, Performance Monitors Overflow Flag Status Clear register

The PMOVSCLR_EL0 characteristics are:

Purpose

Contains the state of the overflow bit for the Cycle Count Register, PMCCNTR_EL0, and each of the implemented event counters PMEVCNTR<n>. Writing to this register clears these bits.

Configurations

External register PMOVSCLR_EL0[31:0] is architecturally mapped to AArch64 System register PMOVSCLR_EL0[31:0].

External register PMOVSCLR_EL0[31:0] is architecturally mapped to AArch32 System register PMOVSR[31:0].

PMOVSCLR_EL0 is in the Core power domain.

Attributes

PMOVSCLR_EL0 is a 32-bit register.

Field descriptions

The PMOVSCLR_EL0 bit assignments are:

31	30 … 0
C	P<n>, bit [n]

C, bit [31]

Cycle counter overflow clear bit.

`0b0` When read, means the cycle counter has not overflowed since this bit was last cleared. When written, has no effect.

`0b1` When read, means the cycle counter has overflowed since this bit was last cleared. When written, clears the cycle counter overflow bit to 0.

PMCR_EL0.LC controls whether an overflow is detected from unsigned overflow of PMCCNTR_EL0[31:0] or unsigned overflow of PMCCNTR_EL0[63:0].

On a Warm reset, this field resets to an architecturally UNKNOWN value.

P<n>, bit [n], for n = 0 to 30

Event counter overflow clear bit for PMEVCNTR<n>_EL0.

If PMCFGR.N is less than 31, bits [30:PMCFGR.N] are RAZ/WI.

`0b0` When read, means that PMEVCNTR<n>_EL0 has not overflowed since this bit was last cleared. When written, has no effect.

`0b1` When read, means that PMEVCNTR<n>_EL0 has overflowed since this bit was last cleared. When written, clears the PMEVCNTR<n>_EL0 overflow bit to 0.

If ARMv8.5-PMU is implemented, MDCR_EL2.HLP and PMCR_EL0.LP control whether an overflow is detected from unsigned overflow of PMEVCNTR<n>_EL0[31:0] or unsigned overflow of PMEVCNTR<n>_EL0[63:0].

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the PMOVSCLR_EL0:

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

PMOVSCLR_EL0 can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0xC80`	PMOVSCLR_EL0

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.32 PMOVSSET_EL0, Performance Monitors Overflow Flag Status Set register

The PMOVSSET_EL0 characteristics are:

Purpose

Sets the state of the overflow bit for the Cycle Count Register, PMCCNTR_EL0, and each of the implemented event counters PMEVCNTR<n>.

Configurations

External register PMOVSSET_EL0[31:0] is architecturally mapped to AArch64 System register PMOVSSET_EL0[31:0].

External register PMOVSSET_EL0[31:0] is architecturally mapped to AArch32 System register PMOVSSET[31:0].

PMOVSSET_EL0 is in the Core power domain.

Attributes

PMOVSSET_EL0 is a 32-bit register.

Field descriptions

The PMOVSSET_EL0 bit assignments are:

31	30 … 0
C	P<n>, bit [n]

C, bit [31]

Cycle counter overflow set bit.

`0b0` When read, means the cycle counter has not overflowed since this bit was last cleared. When written, has no effect.

`0b1` When read, means the cycle counter has overflowed since this bit was last cleared. When written, sets the cycle counter overflow bit to 1.

PMCR_EL0.LC controls whether an overflow is detected from unsigned overflow of PMCCNTR_EL0[31:0] or unsigned overflow of PMCCNTR_EL0[63:0].

On a Warm reset, this field resets to an architecturally UNKNOWN value.

P<n>, bit [n], for n = 0 to 30

Event counter overflow set bit for PMEVCNTR<n>_EL0.

If PMCFGR.N is less than 31, bits [30:PMCFGR.N] are RAZ/WI.

`0b0` When read, means that PMEVCNTR<n>_EL0 has not overflowed since this bit was last cleared. When written, has no effect.

`0b1` When read, means that PMEVCNTR<n>_EL0 has overflowed since this bit was last cleared. When written, sets the PMEVCNTR<n>_EL0 overflow bit to 1.

If ARMv8.5-PMU is implemented, MDCR_EL2.HLP and PMCR_EL0.LP control whether an overflow is detected from unsigned overflow of PMEVCNTR<n>_EL0[31:0] or unsigned overflow of PMEVCNTR<n>_EL0[63:0].

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the PMOVSSET_EL0:

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

PMOVSSET_EL0 can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0xCC0`	PMOVSSET_EL0

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.33 PMPCSR, Program Counter Sample Register

The PMPCSR characteristics are:

Purpose

Holds a sampled instruction address value.

Configurations

PMPCSR is in the Core power domain.

This register is present only when ARMv8.2-PCSample is implemented. Otherwise, direct accesses to PMPCSR are RES0.

Note

Before Armv8.2, the PC Sample-based Profiling Extension can be implemented in the external debug register space, as indicated by the value of EDDEVID.PCSample.

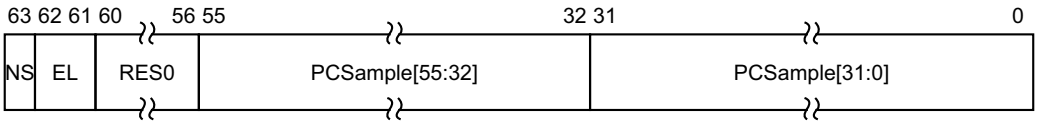
Support for 64-bit atomic reads is IMPLEMENTATION DEFINED. If 64-bit atomic reads are implemented, a 64-bit read of PMPCSR has the same side-effect as a 32-bit read of PMCSR[31:0] followed by a 32-bit read of PMPCSR[63:32], returning the combined value. For example, if the PE is in Debug state then a 64-bit atomic read returns bits[31:0] == `0xFFFFFFFF` and bits[63:32] UNKNOWN.

Attributes

PMPCSR is a 64-bit register.

Field descriptions

The PMPCSR bit assignments are:



NS, bit [63]

Non-secure state sample. Indicates the Security state that is associated with the most recent PMPCSR sample or, when it is read as a single atomic 64-bit read, the current PMPCSR sample.

If EL3 is not implemented, this bit indicates the Effective value of SCR.NS.

`0b0` Sample is from Secure state.

`0b1` Sample is from Non-secure state.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

EL, bits [62:61]

Exception level status sample. Indicates the Exception level that is associated with the most recent PMPCSR sample or, when it is read as a single atomic 64-bit read, the current PMPCSR sample.

`0b00` Sample is from EL0.

`0b01` Sample is from EL1.

`0b10` Sample is from EL2.

`0b11` Sample is from EL3.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [60:56]

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

PCSample[55:32], bits [55:32]

Bits[55:32] of the sampled instruction address value. The translation regime that PMPCSR samples can be determined from PMPCSR.{NS,EL}.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

PCSample[31:0], bits [31:0]

Bits[31:0] of the sampled instruction address value.

PMPCSR[31:0] reads as `0xFFFFFFFF` when any of the following are true:

- The PE is in Debug state.
- PC Sample-based profiling is prohibited.

If an instruction has retired since the PE left Reset state, then the first read of PMPCSR[31:0] is permitted but not required to return `0xFFFFFFFF`.

PMPCSR[31:0] reads as an UNKNOWN value when any of the following are true:

- The PE is in Reset state.
- No instruction has retired since the PE left Reset state, Debug state, or a state where PC Sample-based Profiling is prohibited.
- No instruction has retired since the last read of PMPCSR[31:0].

For the cases where a read of PMPCSR[31:0] returns `0xFFFFFFFF` or an UNKNOWN value, the read has the side-effect of setting PMPCSR[63:32], PMCID1SR, PMCID2SR, and PMVIDSR to UNKNOWN values.

Otherwise, a read of PMPCSR[31:0] returns bits [31:0] of the sampled instruction address value and has the side-effect of indirectly writing to PMPCSR[63:32], PMCID1SR, PMCID2SR, and PMVIDSR. The translation regime that PMPCSR samples can be determined from PMPCSR.{NS,EL}.

For a read of PMPCSR[31:0] from the memory-mapped interface, if PMLSR.SLK == 1, meaning the OPTIONAL Software Lock is locked, then the side-effect of the access does not occur and PMPCSR[63:32], PMCID1SR, PMCID2SR, and PMVIDSR are unchanged.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMPCSR:

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register UNKNOWN, see *Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN* on page H7-7046

PMPCSR[31:0] can be accessed through the external debug interface:

Component	Offset	Instance	Range
PMU	`0x200`	PMPCSR	31:0

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus() and !OSLockStatus() accesses to PMPCSR[31:0] are RO.
- Otherwise accesses to PMPCSR[31:0] generate an error response.

PMPCSR[31:0] can be accessed through the external debug interface:

Component	Offset	Instance	Range
PMU	`0x220`	PMPCSR	31:0

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus() and !OSLockStatus() accesses to PMPCSR[31:0] are RO.
- Otherwise accesses to PMPCSR[31:0] generate an error response.

PMPCSR[63:32] can be accessed through the external debug interface:

Component	Offset	Instance	Range
PMU	`0x204`	PMPCSR	63:32

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus() and !OSLockStatus() accesses to PMPCSR[63:32] are RO.
- Otherwise accesses to PMPCSR[63:32] generate an error response.

PMPCSR[63:32] can be accessed through the external debug interface:

Component	Offset	Instance	Range
PMU	`0x224`	PMPCSR	63:32

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus() and !OSLockStatus() accesses to PMPCSR[63:32] are RO.
- Otherwise accesses to PMPCSR[63:32] generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.34 PMPIDR0, Performance Monitors Peripheral Identification Register 0

The PMPIDR0 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information see *About the Peripheral identification scheme* on page K2-7889.

Configurations

Implementation of this register is OPTIONAL.

If ARMv8.3-DoPD is implemented, this register is in the Core power domain. If ARMv8.3-DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

Attributes

PMPIDR0 is a 32-bit register.

Field descriptions

The PMPIDR0 bit assignments are:

31 … 8	7 … 0
RES0	PART_0

Bits [31:8]

Reserved, RES0.

PART_0, bits [7:0]

Part number, least significant byte.

Accessing the PMPIDR0:

PMPIDR0 can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0xFE0`	PMPIDR0

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.35 PMPIDR1, Performance Monitors Peripheral Identification Register 1

The PMPIDR1 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information see *About the Peripheral identification scheme* on page K2-7889.

Configurations

Implementation of this register is OPTIONAL.

If ARMv8.3-DoPD is implemented, this register is in the Core power domain. If ARMv8.3-DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

Attributes

PMPIDR1 is a 32-bit register.

Field descriptions

The PMPIDR1 bit assignments are:

31 … 8	7 … 4	3 … 0
RES0	DES_0	PART_1

Bits [31:8]

Reserved, RES0.

DES_0, bits [7:4]

Designer, least significant nibble of JEP106 ID code. For Arm Limited, this field is `0b1011`.

PART_1, bits [3:0]

Part number, most significant nibble.

Accessing the PMPIDR1:

PMPIDR1 can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0xFE4`	PMPIDR1

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.36 PMPIDR2, Performance Monitors Peripheral Identification Register 2

The PMPIDR2 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information see *About the Peripheral identification scheme* on page K2-7889.

Configurations

Implementation of this register is OPTIONAL.

If ARMv8.3-DoPD is implemented, this register is in the Core power domain. If ARMv8.3-DoPD is not implemented, this register is in the Debug power domain.

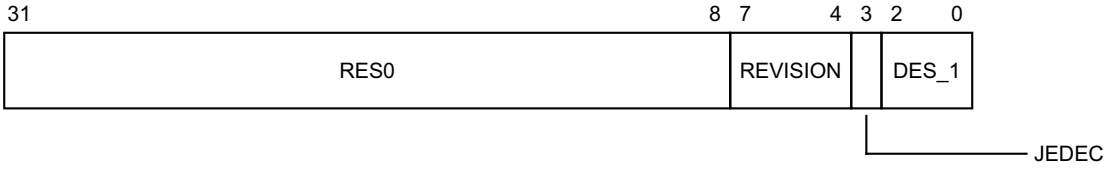
This register is required for CoreSight compliance.

Attributes

PMPIDR2 is a 32-bit register.

Field descriptions

The PMPIDR2 bit assignments are:



Bits [31:8]

Reserved, RES0.

REVISION, bits [7:4]

Part major revision. Parts can also use this field to extend Part number to 16-bits.

JEDEC, bit [3]

RAO. Indicates a JEP106 identity code is used.

DES_1, bits [2:0]

Designer, most significant bits of JEP106 ID code. For Arm Limited, this field is `0b011`.

Accessing the PMPIDR2:

PMPIDR2 can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0xFE8`	PMPIDR2

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.37 PMPIDR3, Performance Monitors Peripheral Identification Register 3

The PMPIDR3 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information see *About the Peripheral identification scheme* on page K2-7889.

Configurations

Implementation of this register is OPTIONAL.

If ARMv8.3-DoPD is implemented, this register is in the Core power domain. If ARMv8.3-DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

Attributes

PMPIDR3 is a 32-bit register.

Field descriptions

The PMPIDR3 bit assignments are:

31 … 8	7 … 4	3 … 0
RES0	REVAND	CMOD

Bits [31:8]

Reserved, RES0.

REVAND, bits [7:4]

Part minor revision. Parts using PMPIDR2.REVISION as an extension to the Part number must use this field as a major revision number.

CMOD, bits [3:0]

Customer modified. Indicates someone other than the Designer has modified the component.

Accessing the PMPIDR3:

PMPIDR3 can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0xFEC`	PMPIDR3

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.38 PMPIDR4, Performance Monitors Peripheral Identification Register 4

The PMPIDR4 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information see *About the Peripheral identification scheme* on page K2-7889.

Configurations

Implementation of this register is OPTIONAL.

If ARMv8.3-DoPD is implemented, this register is in the Core power domain. If ARMv8.3-DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

Attributes

PMPIDR4 is a 32-bit register.

Field descriptions

The PMPIDR4 bit assignments are:

31 … 8	7 … 4	3 … 0
RES0	SIZE	DES_2

Bits [31:8]

Reserved, RES0.

SIZE, bits [7:4]

Size of the component. RAZ. Log_2 of the number of 4KB pages from the start of the component to the end of the component ID registers.

DES_2, bits [3:0]

Designer, JEP106 continuation code, least significant nibble. For Arm Limited, this field is `0b0100`.

Accessing the PMPIDR4:

PMPIDR4 can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0xFD0`	PMPIDR4

This interface is accessible as follows:

- When ARMv8.3-DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.39 PMSWINC_EL0, Performance Monitors Software Increment register

The PMSWINC_EL0 characteristics are:

Purpose

Increments a counter that is configured to count the Software increment event, event `0x00`. For more information, see SW_INCR.

Configurations

External register PMSWINC_EL0[31:0] is architecturally mapped to AArch64 System register PMSWINC_EL0[31:0].

External register PMSWINC_EL0[31:0] is architecturally mapped to AArch32 System register PMSWINC[31:0].

PMSWINC_EL0 is in the Core power domain.

Implementation of this register is OPTIONAL.

If this register is implemented, use of it is deprecated.

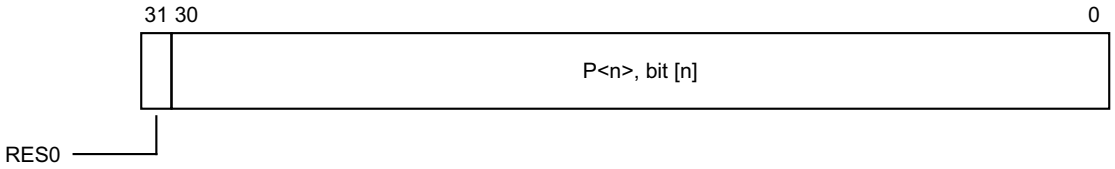
If 1 is written to bit [n] from the external debug interface, it is CONSTRAINED UNPREDICTABLE whether or not a SW_INCR event is created for counter n. This is consistent with not implementing the register in the external debug interface.

Attributes

PMSWINC_EL0 is a 32-bit register.

Field descriptions

The PMSWINC_EL0 bit assignments are:



Bit [31]

Reserved, RES0.

P<n>, bit [n], for n = 0 to 30

Event counter software increment bit for PMEVCNTR<n>_EL0.

If PMCFGR.N is less than 31, bits [30:PMCFGR.N] are WI.

`0b0` No action. The write to this bit is ignored.

`0b1` It is CONSTRAINED UNPREDICTABLE whether a SW_INCR event is generated for event counter n.

Accessing the PMSWINC_EL0:

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

PMSWINC_EL0 can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0xCA0`	PMSWINC_EL0

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and SoftwareLockStatus() accesses to this register are WI.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and !SoftwareLockStatus() accesses to this register are WO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.3.40 PMVIDSR, VMID Sample Register

The PMVIDSR characteristics are:

Purpose

Contains the sampled VMID value that is captured on reading PMPCSR[31:0].

Configurations

PMVIDSR is in the Core power domain.

This register is present only when ARMv8.2-PCSample is implemented and EL2 is implemented. Otherwise, direct accesses to PMVIDSR are RES0.

Note

Before Armv8.2, the PC Sample-based Profiling Extension can be implemented in the external debug register space, as indicated by the value of EDDEVID.PCSample.

Attributes

PMVIDSR is a 32-bit register.

Field descriptions

The PMVIDSR bit assignments are:

31 … 16	15 … 8	7 … 0
RES0	VMID[15:8]	VMID

Bits [31:16]

Reserved, RES0.

VMID[15:8], bits [15:8]

When ARMv8.1-VMID16 is implemented:

Extension to VMID[7:0]. See VMID[7:0] for more details.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

VMID, bits [7:0]

VMID sample. The VMID associated with the most recent PMPCSR sample. When the most recent PMPCSR sample was generated:

- This field is set to an UNKNOWN value if any of the following apply:
  - EL2 is disabled in the current Security state.
  - The PE is executing at EL2.
  - EL2 is enabled in the current Security state, the PE is executing at EL0, EL2 is using AArch64, HCR_EL2.E2H == 1, and HCR_EL2.TGE == 1.
- Otherwise:
  - If EL2 is using AArch64 and either ARMv8.1-VMID16 is not implemented or VTCR_EL2.VS is 1, this field is set to VTTBR_EL2.VMID.
  - If EL2 is using AArch64, ARMv8.1-VMID16 is implemented, and VTCR_EL2.VS is 0, PMVIDSR.VMID[7:0] is set to VTTBR_EL2.VMID[7:0] and PMVIDSR.VMID[15:8] is RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— If EL2 is using AArch32, this field is set to VTTBR.VMID.

Because the value written to PMVIDR is an indirect read of the VMID value, it is CONSTRAINED UNPREDICTABLE whether PMVIDSR is set to the original or new value if PMPCSR samples:

- An instruction that writes to the VMID value.
- The next Context synchronization event.
- Any instruction executed between these two instructions.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMVIDSR:

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register UNKNOWN, see *Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN* on page H7-7046

PMVIDSR can be accessed through the external debug interface:

Component	Offset	Instance
PMU	`0x20C`	PMVIDSR

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus() and !OSLockStatus() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.4 External Activity Monitors Extension registers summary

The memory-mapped interface to the Activity Monitors Extension registers provides read-only access to:

- Read-only copies of the Activity Monitors Extension System registers, with the exception of AMUSERENR.
- An implementation identification register, AMIIDR.
- If they are implemented, the OPTIONAL Activity Monitors CoreSight and ID registers.

The locations of the registers are defined as offsets from a base address. The base address of the memory-mapped view must be aligned to a 4KB boundary, but is otherwise IMPLEMENTATION DEFINED. *Activity Monitors external register views* defines this memory map.

I5.4.1 Activity Monitors external register views

Table I5-2 shows the external view of the Activity Monitors registers. All implemented registers are RO. Offsets within the 4KB region not defined in this table are *RAZ/WI*.

Each entry in the Name column links to the register description in *Activity Monitors external register descriptions* on page I5-7330, and:

- If the *System register?* on page I5-7252 column of the table shows that the register is a System register, the memory-mapped interface provides a view of the System register described in:
  - *Activity Monitors registers* on page D13-3724, for the AArch64 System register.
  - *Activity Monitors registers* on page G8-6759, for the AArch32 System register.
- Otherwise, the register is accessible only using the external memory-mapped interface.

Table I5-2 Activity Monitors external register views

Name	Description	System register?	Offset
AMEVCNTR0<n>[31:0] AMEVCNTR0<n>[63:32]	Activity Monitor Event Counter registers 0	Yes	`0x000+8n` `0x004+8n`
AMEVCNTR1<n>[31:0] AMEVCNTR1<n>[63:32]	Activity Monitor Event Counter registers 1	Yes	`0x100+8n` `0x104+8n`
AMEVTYPER0<n>	Activity Monitor Event Type registers 0	Yes	`0x400+4n`
AMEVTYPER1<n>	Activity Monitor Event Type registers 1	Yes	`0x480+4n`
AMCNTENSET0	Activity Monitors Counter Enable Set register 0	Yes	`0xC00`
AMCNTENSET1	Activity Monitors Counter Enable Set register 1	Yes	`0xC04`
AMCNTENCLR0	Activity Monitors Counter Enable Clear register 0	Yes	`0xC20`
AMCNTENCLR1	Activity Monitors Counter Enable Clear register 1	Yes	`0xC24`
AMCGCR	Activity Monitors Counter Group Configuration Register	Yes	`0xCE0`
AMCFGR	Activity Monitors Configuration Register	Yes	`0xE00`
AMCR	Activity Monitors Control Register	Yes	`0xE04`
AMIIDR	Activity Monitors Implementation Identification Register	No	`0xE08`
AMDEVAFF0[a]	Device Affinity registers	No	`0xFA8`
AMDEVAFF1[a]		No	`0xFAC`
AMDEVARCH[a]	Device Architecture register	No	`0xFBC`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table I5-2 Activity Monitors external register views (continued)

Name	Description	System register?	Offset
AMDEVTYPE[a]	Device Type register	No	0xFCC
AMPIDR4[a]	Peripheral ID registers	No	0xFD0
AMPIDR0[a]		No	0xFE0
AMPIDR1[a]		No	0xFE4
AMPIDR2[a]		No	0xFE8
AMPIDR3[a]		No	0xFEC
AMCIDR0[a]	Component ID registers	No	0xFF0
AMCIDR1[a]		No	0xFF4
AMCIDR2[a]		No	0xFF8
AMCIDR3[a]		No	0xFFC

a. CoreSight interface registers, see *Management registers and CoreSight compliance* on page K2-7881.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.5 Activity Monitors external register descriptions

This section lists the external Activity Monitors registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.5.1 AMCFGR, Activity Monitors Configuration Register

The AMCFGR characteristics are:

Purpose

Global configuration register for the activity monitors.

Provides information on supported features, the number of counter groups implemented, the total number of activity monitor event counters implemented, and the size of the counters. AMCFGR is applicable to both the architected and the auxiliary counter groups.

Configurations

External register AMCFGR[31:0] is architecturally mapped to AArch64 System register AMCFGR_EL0[31:0].

External register AMCFGR[31:0] is architecturally mapped to AArch32 System register AMCFGR[31:0].

The power domain of AMCFGR is IMPLEMENTATION DEFINED.

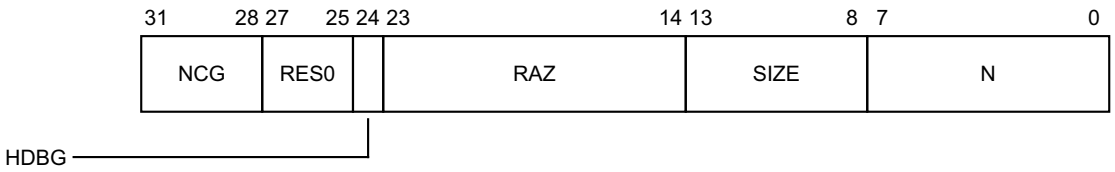
This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMCFGR are RES0.

Attributes

AMCFGR is a 32-bit register.

Field descriptions

The AMCFGR bit assignments are:



NCG, bits [31:28]

Defines the number of counter groups.

The number of implemented counter groups is defined as [AMCFGR.NCG + 1].

If the number of implemented auxiliary activity monitor event counters is zero, this field has a value of `0b0000`. Otherwise, this field has a value of `0b0001`.

Bits [27:25]

Reserved, RES0.

HDBG, bit [24]

Halt-on-debug supported.

From Armv8, this feature must be supported, and so this bit is `0b1`.

`0b0` AMCR.HDBG is RES0.

`0b1` AMCR.HDBG is read/write.

Bits [23:14]

Reserved, RAZ.

SIZE, bits [13:8]

Defines the size of activity monitor event counters.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The size of the activity monitor event counters implemented by the Activity Monitors Extension is defined as [AMCFGR.SIZE + 1].

From Armv8, the counters are 64-bit, and so this field is `0b111111`.

Note

Software also uses this field to determine the spacing of counters in the memory-map. From Armv8, the counters are at doubleword-aligned addresses.

N, bits [7:0]

Defines the number of activity monitor event counters.

The total number of counters implemented in all groups by the Activity Monitors Extension is defined as [AMCFGR.N + 1].

Accessing the AMCFGR:

AMCFGR can be accessed through its memory-mapped interface:

Component	Offset	Instance
AMU	`0xE00`	AMCFGR

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.5.2 AMCGCR, Activity Monitors Counter Group Configuration Register

The AMCGCR characteristics are:

Purpose

Provides information on the number of activity monitor event counters implemented within each counter group.

Configurations

External register AMCGCR[31:0] is architecturally mapped to AArch64 System register AMCGCR_EL0[31:0].

External register AMCGCR[31:0] is architecturally mapped to AArch32 System register AMCGCR[31:0].

The power domain of AMCGCR is IMPLEMENTATION DEFINED.

This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMCGCR are RES0.

Attributes

AMCGCR is a 32-bit register.

Field descriptions

The AMCGCR bit assignments are:

31 … 16	15 … 8	7 … 0
RES0	CG1NC	CG0NC

Bits [31:16]

Reserved, RES0.

CG1NC, bits [15:8]

Counter Group 1 Number of Counters. The number of counters in the auxiliary counter group.

In AMUv1, the permitted range of values is 0 to 16.

CG0NC, bits [7:0]

Counter Group 0 Number of Counters. The number of counters in the architected counter group.

In AMUv1, the value of this field is 4.

Accessing the AMCGCR:

AMCGCR can be accessed through its memory-mapped interface:

Component	Offset	Instance
AMU	`0xCE0`	AMCGCR

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.5.3 AMCIDR0, Activity Monitors Component Identification Register 0

The AMCIDR0 characteristics are:

Purpose

Provides information to identify an activity monitors component.

For more information, see *About the Component Identification scheme* on page K2-7892 in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

Configurations

The power domain of AMCIDR0 is IMPLEMENTATION DEFINED.

Implementation of this register is OPTIONAL.

This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMCIDR0 are RES0.

Attributes

AMCIDR0 is a 32-bit register.

Field descriptions

The AMCIDR0 bit assignments are:

31 … 8	7 … 0
RES0	PRMBL_0

Bits [31:8]

Reserved, RES0.

PRMBL_0, bits [7:0]

Preamble. Must read as `0x0D`.

Accessing the AMCIDR0:

AMCIDR0 can be accessed through its memory-mapped interface:

Component	Offset	Instance
AMU	`0xFF0`	AMCIDR0

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.5.4 AMCIDR1, Activity Monitors Component Identification Register 1

The AMCIDR1 characteristics are:

Purpose

Provides information to identify an activity monitors component.

For more information, see *About the Component Identification scheme* on page K2-7892 in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

Configurations

The power domain of AMCIDR1 is IMPLEMENTATION DEFINED.

Implementation of this register is OPTIONAL.

This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMCIDR1 are RES0.

Attributes

AMCIDR1 is a 32-bit register.

Field descriptions

The AMCIDR1 bit assignments are:

31 … 8	7 … 4	3 … 0
RES0	CLASS	PRMBL_1

Bits [31:8]

Reserved, RES0.

CLASS, bits [7:4]

Component class. Reads as `0x9`, CoreSight component.

PRMBL_1, bits [3:0]

Preamble. Reads as `0x0`.

Accessing the AMCIDR1:

AMCIDR1 can be accessed through its memory-mapped interface:

Component	Offset	Instance
AMU	`0xFF4`	AMCIDR1

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.5.5 AMCIDR2, Activity Monitors Component Identification Register 2

The AMCIDR2 characteristics are:

Purpose

Provides information to identify an activity monitors component.

For more information, see *About the Component Identification scheme* on page K2-7892 in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

Configurations

The power domain of AMCIDR2 is IMPLEMENTATION DEFINED.

Implementation of this register is OPTIONAL.

This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMCIDR2 are RES0.

Attributes

AMCIDR2 is a 32-bit register.

Field descriptions

The AMCIDR2 bit assignments are:

31 … 8	7 … 0
RES0	PRMBL_2

Bits [31:8]

Reserved, RES0.

PRMBL_2, bits [7:0]

Preamble. Reads as `0x05`.

Accessing the AMCIDR2:

AMCIDR2 can be accessed through its memory-mapped interface:

Component	Offset	Instance
AMU	`0xFF8`	AMCIDR2

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.5.6 AMCIDR3, Activity Monitors Component Identification Register 3

The AMCIDR3 characteristics are:

Purpose

Provides information to identify an activity monitors component.

For more information, see *About the Component Identification scheme* on page K2-7892 in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

Configurations

The power domain of AMCIDR3 is IMPLEMENTATION DEFINED.

Implementation of this register is OPTIONAL.

This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMCIDR3 are RES0.

Attributes

AMCIDR3 is a 32-bit register.

Field descriptions

The AMCIDR3 bit assignments are:

31 … 8	7 … 0
RES0	PRMBL_3

Bits [31:8]

Reserved, RES0.

PRMBL_3, bits [7:0]

Preamble. Reads as `0xB1`.

Accessing the AMCIDR3:

AMCIDR3 can be accessed through its memory-mapped interface:

Component	Offset	Instance
AMU	`0xFFC`	AMCIDR3

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.5.7 AMCNTENCLR0, Activity Monitors Count Enable Clear Register 0

The AMCNTENCLR0 characteristics are:

Purpose

Disable control bits for the architected s event counters, AMEVCNTR0<n>.

Configurations

External register AMCNTENCLR0[31:0] is architecturally mapped to AArch64 System register AMCNTENCLR0_EL0[31:0].

External register AMCNTENCLR0[31:0] is architecturally mapped to AArch32 System register AMCNTENCLR0[31:0].

The power domain of AMCNTENCLR0 is IMPLEMENTATION DEFINED.

This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMCNTENCLR0 are RES0.

Attributes

AMCNTENCLR0 is a 32-bit register.

Field descriptions

The AMCNTENCLR0 bit assignments are:

31	0
P<n>, bit [n]	

P<n>, bit [n], for n = 0 to 31

Activity monitor event counter disable bit for AMEVCNTR0<n>.

Bits [31:N] are RAZ/WI. N is the value in AMCGCR.CG0NC.

Possible values of each bit are:

`0b0` When read, means that AMEVCNTR0<n> is disabled. When written, has no effect.

`0b1` When read, means that AMEVCNTR0<n> is enabled. When written, disables AMEVCNTR0<n>.

On a Cold reset, this field resets to `0`.

Accessing the AMCNTENCLR0:

AMCNTENCLR0 can be accessed through its memory-mapped interface:

Component	Offset	Instance
AMU	`0xC20`	AMCNTENCLR0

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.5.8 AMCNTENCLR1, Activity Monitors Count Enable Clear Register 1

The AMCNTENCLR1 characteristics are:

Purpose

Disable control bits for the auxiliary activity monitors event counters, AMEVCNTR1<n>.

Configurations

External register AMCNTENCLR1[31:0] is architecturally mapped to AArch64 System register AMCNTENCLR1_EL0[31:0].

External register AMCNTENCLR1[31:0] is architecturally mapped to AArch32 System register AMCNTENCLR1[31:0].

The power domain of AMCNTENCLR1 is IMPLEMENTATION DEFINED.

This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMCNTENCLR1 are RES0.

Attributes

AMCNTENCLR1 is a 32-bit register.

Field descriptions

The AMCNTENCLR1 bit assignments are:

31	0
P<n>, bit [n]	

P<n>, bit [n], for n = 0 to 31

Activity monitor event counter disable bit for AMEVCNTR1<n>.

Bits [31:N] are RAZ/WI. N is the value in AMCGCR.CG1NC.

Possible values of each bit are:

`0b0` When read, means that AMEVCNTR1<n> is disabled. When written, has no effect.

`0b1` When read, means that AMEVCNTR1<n> is enabled. When written, disables AMEVCNTR1<n>.

On a Cold reset, this field resets to `0`.

Accessing the AMCNTENCLR1:

If the number of auxiliary activity monitor event counters implemented is zero, reads and writes of AMCNTENCLR1 are CONSTRAINED UNPREDICTABLE, and accesses to the register behave as RAZ/WI.

Note

The number of auxiliary activity monitor event counters implemented is zero exactly when AMCFGR.NCG == `0b0000`.

AMCNTENCLR1 can be accessed through its memory-mapped interface:

Component	Offset	Instance
AMU	`0xC24`	AMCNTENCLR1

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.5.9 AMCNTENSET0, Activity Monitors Count Enable Set Register 0

The AMCNTENSET0 characteristics are:

Purpose

Enable control bits for the architected activity monitors event counters, AMEVCNTR0<n>.

Configurations

External register AMCNTENSET0[31:0] is architecturally mapped to AArch64 System register AMCNTENSET0_EL0[31:0].

External register AMCNTENSET0[31:0] is architecturally mapped to AArch32 System register AMCNTENSET0[31:0].

The power domain of AMCNTENSET0 is IMPLEMENTATION DEFINED.

This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMCNTENSET0 are RES0.

Attributes

AMCNTENSET0 is a 32-bit register.

Field descriptions

The AMCNTENSET0 bit assignments are:

31	0
P<n>, bit [n]	

P<n>, bit [n], for n = 0 to 31

Activity monitor event counter enable bit for AMEVCNTR0<n>.

Bits [31:N] are RAZ/WI. N is the value in AMCGCR.CG0NC.

Possible values of each bit are:

`0b0` When read, means that AMEVCNTR0<n> is disabled. When written, has no effect.

`0b1` When read, means that AMEVCNTR0<n> is enabled. When written, enables AMEVCNTR0<n>.

On a Cold reset, this field resets to `0`.

Accessing the AMCNTENSET0:

AMCNTENSET0 can be accessed through its memory-mapped interface:

Component	Offset	Instance
AMU	`0xC00`	AMCNTENSET0

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.5.10 AMCNTENSET1, Activity Monitors Count Enable Set Register 1

The AMCNTENSET1 characteristics are:

Purpose

Enable control bits for the auxiliary activity monitors event counters, AMEVCNTR1<n>.

Configurations

External register AMCNTENSET1[31:0] is architecturally mapped to AArch64 System register AMCNTENSET1_EL0[31:0].

External register AMCNTENSET1[31:0] is architecturally mapped to AArch32 System register AMCNTENSET1[31:0].

The power domain of AMCNTENSET1 is IMPLEMENTATION DEFINED.

This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMCNTENSET1 are RES0.

Attributes

AMCNTENSET1 is a 32-bit register.

Field descriptions

The AMCNTENSET1 bit assignments are:

31	0
P<n>, bit [n]	

P<n>, bit [n], for n = 0 to 31

Activity monitor event counter enable bit for AMEVCNTR1<n>.

Bits [31:N] are RAZ/WI. N is the value in AMCGCR.CG1NC.

Possible values of each bit are:

`0b0` When read, means that AMEVCNTR1<n> is disabled. When written, has no effect.

`0b1` When read, means that AMEVCNTR1<n> is enabled. When written, enables AMEVCNTR1<n>.

On a Cold reset, this field resets to `0`.

Accessing the AMCNTENSET1:

If the number of auxiliary activity monitor event counters implemented is zero, reads and writes of AMCNTENSET1 are CONSTRAINED UNPREDICTABLE, and accesses to the register behave as RAZ/WI.

Note

The number of auxiliary activity monitor counters implemented is zero exactly when AMCFGR.NCG == `0b0000`.

AMCNTENSET1 can be accessed through its memory-mapped interface:

Component	Offset	Instance
AMU	`0xC04`	AMCNTENSET1

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.5.11 AMCR, Activity Monitors Control Register

The AMCR characteristics are:

Purpose

Global control register for the activity monitors implementation. AMCR is applicable to both the architected and the auxiliary counter groups.

Configurations

External register AMCR[31:0] is architecturally mapped to AArch64 System register AMCR_EL0[31:0].

External register AMCR[31:0] is architecturally mapped to AArch32 System register AMCR[31:0].

The power domain of AMCR is IMPLEMENTATION DEFINED.

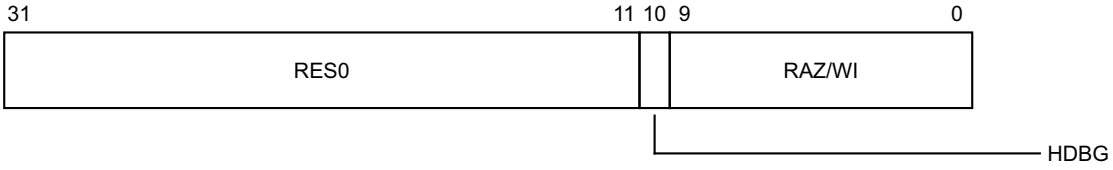
This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMCR are RES0.

Attributes

AMCR is a 32-bit register.

Field descriptions

The AMCR bit assignments are:



Bits [31:11]

Reserved, RES0.

HDBG, bit [10]

This bit controls whether activity monitor counting is halted when the PE is halted in Debug state.

`0b0` Activity monitors do not halt counting when the PE is halted in Debug state.

`0b1` Activity monitors halt counting when the PE is halted in Debug state.

Bits [9:0]

Reserved, RAZ/WI.

Accessing the AMCR:

AMCR can be accessed through its memory-mapped interface:

Component	Offset	Instance
AMU	`0xE04`	AMCR

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.5.12 AMDEVAFF0, Activity Monitors Device Affinity Register 0

The AMDEVAFF0 characteristics are:

Purpose

Copy of the low half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the AMU component relates to.

Configurations

The power domain of AMDEVAFF0 is IMPLEMENTATION DEFINED.

Implementation of this register is OPTIONAL.

This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMDEVAFF0 are RES0.

Attributes

AMDEVAFF0 is a 32-bit register.

Field descriptions

The AMDEVAFF0 bit assignments are:

31	0
MPIDR_EL1lo	

MPIDR_EL1lo, bits [31:0]

MPIDR_EL1 low half. Read-only copy of the low half of MPIDR_EL1, as seen from the highest implemented Exception level.

Accessing the AMDEVAFF0:

AMDEVAFF0 can be accessed through its memory-mapped interface:

Component	Offset	Instance
AMU	`0xFA8`	AMDEVAFF0

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.5.13 AMDEVAFF1, Activity Monitors Device Affinity Register 1

The AMDEVAFF1 characteristics are:

Purpose

Copy of the high half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the AMU component relates to.

Configurations

The power domain of AMDEVAFF1 is IMPLEMENTATION DEFINED.

Implementation of this register is OPTIONAL.

This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMDEVAFF1 are RES0.

Attributes

AMDEVAFF1 is a 32-bit register.

Field descriptions

The AMDEVAFF1 bit assignments are:

31	0
MPIDR_EL1hi	

MPIDR_EL1hi, bits [31:0]

MPIDR_EL1 high half. Read-only copy of the high half of MPIDR_EL1, as seen from the highest implemented Exception level.

Accessing the AMDEVAFF1:

AMDEVAFF1 can be accessed through its memory-mapped interface:

Component	Offset	Instance
AMU	`0xFAC`	AMDEVAFF1

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.5.14 AMDEVARCH, Activity Monitors Device Architecture Register

The AMDEVARCH characteristics are:

Purpose

Identifies the programmers' model architecture of the AMU component.

Configurations

The power domain of AMDEVARCH is IMPLEMENTATION DEFINED.

Implementation of this register is OPTIONAL.

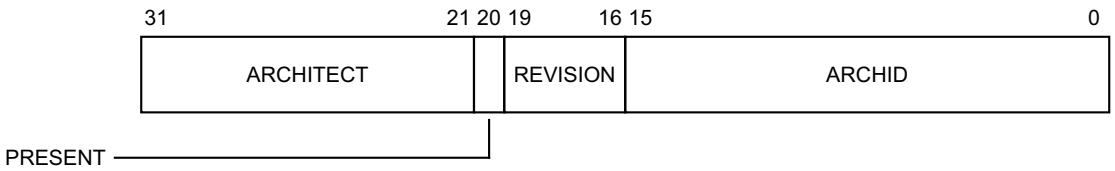
This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMDEVARCH are RES0.

Attributes

AMDEVARCH is a 32-bit register.

Field descriptions

The AMDEVARCH bit assignments are:



ARCHITECT, bits [31:21]

Defines the architecture of the component. For AMU, this is Arm Limited.

Bits [31:28] are the JEP106 continuation code, `0x4`.

Bits [27:21] are the JEP106 ID code, `0x3B`.

PRESENT, bit [20]

When set to 1, indicates that the DEVARCH is present.

This field is 1 in Armv8.

REVISION, bits [19:16]

Defines the architecture revision. For architectures defined by Arm this is the minor revision.

`0b0000` Architecture revision is AMUv1.

All other values are reserved.

ARCHID, bits [15:0]

Defines this part to be an AMU component. For architectures defined by Arm this is further subdivided.

For AMU:

- Bits [15:12] are the architecture version, `0x0`.
- Bits [11:0] are the architecture part number, `0xA66`.

This corresponds to AMU architecture version AMUv1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the AMDEVARCH:

AMDEVARCH can be accessed through its memory-mapped interface:

Component	Offset	Instance
AMU	`0xFBC`	AMDEVARCH

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.5.15 AMDEVTYPE, Activity Monitors Device Type Register

The AMDEVTYPE characteristics are:

Purpose

Indicates to a debugger that this component is part of a PE's performance monitor interface.

Configurations

The power domain of AMDEVTYPE is IMPLEMENTATION DEFINED.

Implementation of this register is OPTIONAL.

This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMDEVTYPE are RES0.

Attributes

AMDEVTYPE is a 32-bit register.

Field descriptions

The AMDEVTYPE bit assignments are:

31 … 8	7 … 4	3 … 0
RES0	SUB	MAJOR

Bits [31:8]

Reserved, RES0.

SUB, bits [7:4]

Subtype. Reads as `0x1`, to indicate this is a component within a PE.

MAJOR, bits [3:0]

Major type. Reads as `0x6`, to indicate this is a performance monitor component.

Accessing the AMDEVTYPE:

AMDEVTYPE can be accessed through its memory-mapped interface:

Component	Offset	Instance
AMU	`0xFCC`	AMDEVTYPE

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.5.16 AMEVCNTR0<n>, Activity Monitors Event Counter Registers 0, n = 0 - 15

The AMEVCNTR0<n> characteristics are:

Purpose

Provides access to the architected activity monitor event counters.

Configurations

External register AMEVCNTR0<n>[63:0] is architecturally mapped to AArch64 System register AMEVCNTR0<n>_EL0[63:0].

External register AMEVCNTR0<n>[63:0] is architecturally mapped to AArch32 System register AMEVCNTR0<n>[63:0].

The power domain of AMEVCNTR0<n> is IMPLEMENTATION DEFINED.

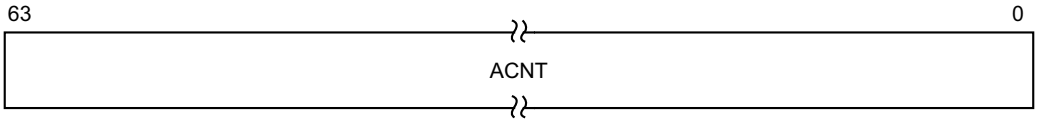
This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMEVCNTR0<n> are RES0.

Attributes

AMEVCNTR0<n> is a 64-bit register.

Field descriptions

The AMEVCNTR0<n> bit assignments are:



ACNT, bits [63:0]

Architected activity monitor event counter n.

Value of architected activity monitor event counter n, where n is the number of this register and is a number from 0 to 15.

If the counter is enabled, writes to this register have UNPREDICTABLE results.

On a Cold reset, this field resets to `0`.

Accessing the AMEVCNTR0<n>:

If <n> is greater than or equal to the number of architected activity monitor event counters, reads and writes of AMEVCNTR0<n> are CONSTRAINED UNPREDICTABLE, and accesses to the register behave as RAZ/WI.

Note

AMCGCR.CG0NC identifies the number of architected activity monitor event counters.

AMEVCNTR0<n>[31:0] can be accessed through its memory-mapped interface:

Component	Offset	Instance	Range
AMU	`0x000` + 8n	AMEVCNTR0<n>	31:0

This interface is accessible as follows:

- Accesses to AMEVCNTR0<n>[31:0] are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

AMEVCNTR0<n>[63:32] can be accessed through its memory-mapped interface:

Component	Offset	Instance	Range
AMU	`0x004` + 8n	AMEVCNTR0<n>	63:32

This interface is accessible as follows:

- Accesses to AMEVCNTR0<n>[63:32] are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.5.17 AMEVCNTR1<n>, Activity Monitors Event Counter Registers 1, n = 0 - 15

The AMEVCNTR1<n> characteristics are:

Purpose

Provides access to the auxiliary activity monitor event counters.

Configurations

External register AMEVCNTR1<n>[63:0] is architecturally mapped to AArch64 System register AMEVCNTR1<n>_EL0[63:0].

External register AMEVCNTR1<n>[63:0] is architecturally mapped to AArch32 System register AMEVCNTR1<n>[63:0].

The power domain of AMEVCNTR1<n> is IMPLEMENTATION DEFINED.

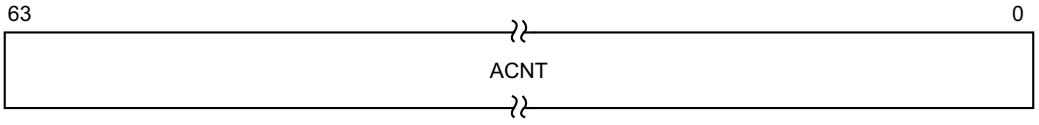
This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMEVCNTR1<n> are RES0.

Attributes

AMEVCNTR1<n> is a 64-bit register.

Field descriptions

The AMEVCNTR1<n> bit assignments are:



ACNT, bits [63:0]

Auxiliary activity monitor event counter n.

Value of auxiliary activity monitor event counter n, where n is the number of this register and is a number from 0 to 15.

If the counter is enabled, writes to this register have UNPREDICTABLE results.

On a Cold reset, this field resets to `0`.

Accessing the AMEVCNTR1<n>:

If <n> is greater than or equal to the number of auxiliary activity monitor event counters, reads and writes of AMEVCNTR1<n> are CONSTRAINED UNPREDICTABLE, and accesses to the register behave as RAZ/WI.

Note

AMCGCR.CG1NC identifies the number of auxiliary activity monitor event counters.

AMEVCNTR1<n>[31:0] can be accessed through its memory-mapped interface:

Component	Offset	Instance	Range
AMU	`0x100` + 8n	AMEVCNTR1<n>	31:0

This interface is accessible as follows:

- Accesses to AMEVCNTR1<n>[31:0] are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

AMEVCNTR1<n>[63:32] can be accessed through its memory-mapped interface:

Component	Offset	Instance	Range
AMU	`0x104` + 8n	AMEVCNTR1<n>	63:32

This interface is accessible as follows:

- Accesses to AMEVCNTR1<n>[63:32] are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.5.18 AMEVTYPER0<n>, Activity Monitors Event Type Registers 0, n = 0 - 15

The AMEVTYPER0<n> characteristics are:

Purpose

Provides information on the events that an architected activity monitor event counter AMEVCNTR0<n> counts.

Configurations

External register AMEVTYPER0<n>[31:0] is architecturally mapped to AArch64 System register AMEVTYPER0<n>_EL0[31:0].

External register AMEVTYPER0<n>[31:0] is architecturally mapped to AArch32 System register AMEVTYPER0<n>[31:0].

The power domain of AMEVTYPER0<n> is IMPLEMENTATION DEFINED.

This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMEVTYPER0<n> are RES0.

Attributes

AMEVTYPER0<n> is a 32-bit register.

Field descriptions

The AMEVTYPER0<n> bit assignments are:

31 – 25	24 – 16	15 – 0
RAZ	RES0	evtCount

Bits [31:25]

Reserved, RAZ.

Bits [24:16]

Reserved, RES0.

evtCount, bits [15:0]

Event to count. The event number of the event that is counted by the architected activity monitor event counter AMEVCNTR0<n>. The value of this field is architecturally mandated for each architected counter.

The following table shows the mapping between required event numbers and the corresponding counters:

`0x0011` **When n == 0** Processor frequency cycles

`0x4004` **When n == 1** Constant frequency cycles

`0x0008` **When n == 2** Instructions retired

`0x4005` **When n == 3** Memory stall cycles

Accessing the AMEVTYPER0<n>:

If <n> is greater than or equal to the number of architected activity monitor event counters, reads and writes of AMEVTYPER0<n> are CONSTRAINED UNPREDICTABLE, and accesses to the register behave as RAZ/WI.

Note

AMCGCR.CG0NC identifies the number of architected activity monitor event counters.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

AMEVTYPER0<n> can be accessed through its memory-mapped interface:

Component	Offset	Instance
AMU	`0x400` + 4n	AMEVTYPER0<n>

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.5.19 AMEVTYPER1<n>, Activity Monitors Event Type Registers 1, n = 0 - 15

The AMEVTYPER1<n> characteristics are:

Purpose

Provides information on the events that an auxiliary activity monitor event counter AMEVCNTR1<n> counts.

Configurations

External register AMEVTYPER1<n>[31:0] is architecturally mapped to AArch64 System register AMEVTYPER1<n>_EL0[31:0].

External register AMEVTYPER1<n>[31:0] is architecturally mapped to AArch32 System register AMEVTYPER1<n>[31:0].

The power domain of AMEVTYPER1<n> is IMPLEMENTATION DEFINED.

This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMEVTYPER1<n> are RES0.

Attributes

AMEVTYPER1<n> is a 32-bit register.

Field descriptions

The AMEVTYPER1<n> bit assignments are:

31 – 25	24 – 16	15 – 0
RAZ	RES0	evtCount

Bits [31:25]

Reserved, RAZ.

Bits [24:16]

Reserved, RES0.

evtCount, bits [15:0]

Event to count. The event number of the event that is counted by the auxiliary activity monitor event counter AMEVCNTR1<n>.

It is IMPLEMENTATION DEFINED what values are supported by each counter.

If software writes a value to this field which is not supported by the corresponding counter AMEVCNTR1<n>, then:

- It is UNPREDICTABLE which event will be counted.
- The value read back is UNKNOWN.

Note

The event counted by AMEVCNTR1<n> might be fixed at implementation. In this case, the field is read-only and writes are UNDEFINED.

If the corresponding counter AMEVCNTR1<n> is enabled, writes to this register have UNPREDICTABLE results.

Accessing the AMEVTYPER1<n>:

If <n> is greater than or equal to the number of auxiliary activity monitor event counters, reads and writes of AMEVTYPER1<n> are CONSTRAINED UNPREDICTABLE, and accesses to the register behave as RAZ/WI.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

AMCGCR.CG1NC identifies the number of auxiliary activity monitor event counters.

AMEVTYPER1<n> can be accessed through its memory-mapped interface:

Component	Offset	Instance
AMU	`0x480` + 4n	AMEVTYPER1<n>

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.5.20 AMIIDR, Activity Monitors Implementation Identification Register

The AMIIDR characteristics are:

Purpose

Defines the implementer and revisions of the AMU.

Configurations

The power domain of AMIIDR is IMPLEMENTATION DEFINED.

This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMIIDR are RES0.

Attributes

AMIIDR is a 32-bit register.

Field descriptions

The AMIIDR bit assignments are:

31 – 20	19 – 16	15 – 12	11 – 0
ProductID	Variant	Revision	Implementer

ProductID, bits [31:20]

This field is an AMU part identifier.

The value of this field is IMPLEMENTATION DEFINED.

If AMPIDR0 is implemented, AMPIDR0.PART_0 matches bits [27:20] of this field.

If AMPIDR1 is implemented, AMPIDR1.PART_1 matches bits [31:28] of this field.

Variant, bits [19:16]

This field distinguishes product variants or major revisions of the product.

The value of this field is IMPLEMENTATION DEFINED.

If AMPIDR2 is implemented, AMPIDR2.REVISION matches AMIIDR.Variant.

Revision, bits [15:12]

This field distinguishes minor revisions of the product.

The value of this field is IMPLEMENTATION DEFINED.

If AMPIDR3 is implemented, AMPIDR3.REVAND matches AMIIDR.Revision.

Implementer, bits [11:0]

Contains the JEP106 code of the company that implemented the AMU.

For an Arm implementation, this field reads as `0x43B`.

Bits [11:8] contain the JEP106 continuation code of the implementer.

Bit 7 is RES0

Bits [6:0] contain the JEP106 identity code of the implementer.

If AMPIDR4 is implemented, AMPIDR4.DES_2 matches bits [11:8] of this field.

If AMPIDR2 is implemented, AMPIDR2.DES_1 matches bits [6:4] of this field.

If AMPIDR1 is implemented, AMPIDR1.DES_0 matches bits [3:0] of this field.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the AMIIDR:

AMIIDR can be accessed through its memory-mapped interface:

Component	Offset	Instance
AMU	`0xE08`	AMIIDR

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.5.21 AMPIDR0, Activity Monitors Peripheral Identification Register 0

The AMPIDR0 characteristics are:

Purpose

Provides information to identify an activity monitors component.

For more information, see *About the Peripheral identification scheme* on page K2-7889 in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

Configurations

The power domain of AMPIDR0 is IMPLEMENTATION DEFINED.

Implementation of this register is OPTIONAL.

This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMPIDR0 are RES0.

Attributes

AMPIDR0 is a 32-bit register.

Field descriptions

The AMPIDR0 bit assignments are:

31 … 8	7 … 0
RES0	PART_0

Bits [31:8]

Reserved, RES0.

PART_0, bits [7:0]

Part number, least significant byte.

The value of this field is IMPLEMENTATION DEFINED.

Accessing the AMPIDR0:

AMPIDR0 can be accessed through its memory-mapped interface:

Component	Offset	Instance
AMU	`0xFE0`	AMPIDR0

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.5.22 AMPIDR1, Activity Monitors Peripheral Identification Register 1

The AMPIDR1 characteristics are:

Purpose

Provides information to identify an activity monitors component.

For more information, see *About the Peripheral identification scheme* on page K2-7889 in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

Configurations

The power domain of AMPIDR1 is IMPLEMENTATION DEFINED.

Implementation of this register is OPTIONAL.

This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMPIDR1 are RES0.

Attributes

AMPIDR1 is a 32-bit register.

Field descriptions

The AMPIDR1 bit assignments are:

31 … 8	7 … 4	3 … 0
RES0	DES_0	PART_1

Bits [31:8]

Reserved, RES0.

DES_0, bits [7:4]

Designer, least significant nibble of JEP106 ID code.

The value of this field is IMPLEMENTATION DEFINED. For Arm Limited, this field is `0b1011`.

PART_1, bits [3:0]

Part number, most significant nibble.

The value of this field is IMPLEMENTATION DEFINED.

Accessing the AMPIDR1:

AMPIDR1 can be accessed through its memory-mapped interface:

Component	Offset	Instance
AMU	`0xFE4`	AMPIDR1

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.5.23 AMPIDR2, Activity Monitors Peripheral Identification Register 2

The AMPIDR2 characteristics are:

Purpose

Provides information to identify an activity monitors component.

For more information, see *About the Peripheral identification scheme* on page K2-7889 in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

Configurations

The power domain of AMPIDR2 is IMPLEMENTATION DEFINED.

Implementation of this register is OPTIONAL.

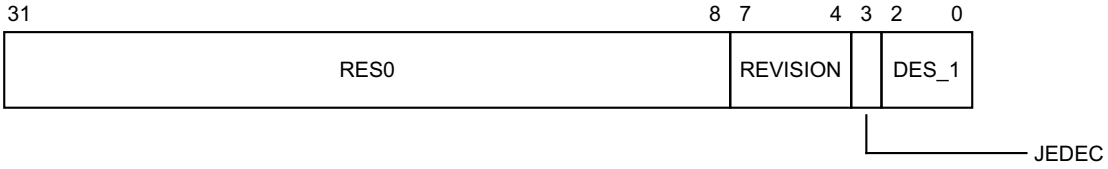
This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMPIDR2 are RES0.

Attributes

AMPIDR2 is a 32-bit register.

Field descriptions

The AMPIDR2 bit assignments are:



Bits [31:8]

Reserved, RES0.

REVISION, bits [7:4]

Part major revision. Parts can also use this field to extend Part number to 16-bits.

The value of this field is IMPLEMENTATION DEFINED.

JEDEC, bit [3]

RAO. Indicates a JEP106 identity code is used.

DES_1, bits [2:0]

Designer, most significant bits of JEP106 ID code.

The value of this field is IMPLEMENTATION DEFINED. For Arm Limited, this field is `0b011`.

Accessing the AMPIDR2:

AMPIDR2 can be accessed through its memory-mapped interface:

Component	Offset	Instance
AMU	`0xFE8`	AMPIDR2

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.5.24 AMPIDR3, Activity Monitors Peripheral Identification Register 3

The AMPIDR3 characteristics are:

Purpose

Provides information to identify an activity monitors component.

For more information, see *About the Peripheral identification scheme* on page K2-7889 in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

Configurations

The power domain of AMPIDR3 is IMPLEMENTATION DEFINED.

Implementation of this register is OPTIONAL.

This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMPIDR3 are RES0.

Attributes

AMPIDR3 is a 32-bit register.

Field descriptions

The AMPIDR3 bit assignments are:

31 … 8	7 … 4	3 … 0
RES0	REVAND	CMOD

Bits [31:8]

Reserved, RES0.

REVAND, bits [7:4]

Part minor revision. Parts using AMPIDR2.REVISION as an extension to the Part number must use this field as a major revision number.

The value of this field is IMPLEMENTATION DEFINED.

CMOD, bits [3:0]

Customer modified. Indicates someone other than the Designer has modified the component.

The value of this field is IMPLEMENTATION DEFINED.

Accessing the AMPIDR3:

AMPIDR3 can be accessed through its memory-mapped interface:

Component	Offset	Instance
AMU	`0xFEC`	AMPIDR3

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.5.25 AMPIDR4, Activity Monitors Peripheral Identification Register 4

The AMPIDR4 characteristics are:

Purpose

Provides information to identify an activity monitors component.

For more information, see *About the Peripheral identification scheme* on page K2-7889 in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

Configurations

The power domain of AMPIDR4 is IMPLEMENTATION DEFINED.

Implementation of this register is OPTIONAL.

This register is present only when AMUv1 is implemented. Otherwise, direct accesses to AMPIDR4 are RES0.

Attributes

AMPIDR4 is a 32-bit register.

Field descriptions

The AMPIDR4 bit assignments are:

31 … 8	7 … 4	3 … 0
RES0	SIZE	DES_2

Bits [31:8]

Reserved, RES0.

SIZE, bits [7:4]

Size of the component. Log_2 of the number of 4KB pages from the start of the component to the end of the component ID registers.

This field reads as `0b0000`.

DES_2, bits [3:0]

Designer. JEP106 continuation code, least significant nibble.

The value of this field is IMPLEMENTATION DEFINED. For Arm Limited, this field is `0b0100`.

Accessing the AMPIDR4:

AMPIDR4 can be accessed through its memory-mapped interface:

Component	Offset	Instance
AMU	`0xFD0`	AMPIDR4

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.6 Generic Timer memory-mapped registers overview

The Generic Timer memory-mapped registers are implemented as multiple register frames, with each register frame having its own base address, as follows:

- A single CNTCTLBase register frame, at base address CNTCTLBase.
- Between one and seven CNTBase*N* register frames, each with its own base address CNTBase*N*.
- For each CNTBase*N* register frame, if required, a CNTEL0Base*N* register frame, at base address CNTEL0Base*N*, that provides an EL0 view of the CNTBase*N* register frame.

For more information, see:

- *Memory-mapped timer components* on page I2-7236.
- *The CNTBaseN and CNTEL0BaseN frames* on page I2-7238. This section includes the memory map of the CNTBase*N* and CNTBase*N* register frames.
- *The CNTCTLBase frame* on page I2-7237. This section includes the memory map of the CNTCTLBase register frame.

Note

Providing a complete set of features in a system level implementation on page K5-7914 gives an implementation example for a system level implementation of the Generic Timer.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.7 Generic Timer memory-mapped register descriptions

This section describes the Generic Timer registers. *Generic Timer memory-mapped registers overview* on page I5-7365 gives an overview of these registers, and includes links to their memory maps.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.7.1 CNTACR<n>, Counter-timer Access Control Registers, n = 0 - 7

The CNTACR<n> characteristics are:

Purpose

Provides top-level access controls for the elements of a timer frame. CNTACR<n> provides the controls for frame CNTBaseN.

In addition to the CNTACR<n> control:

- CNTNSAR controls whether CNTACR<n> is accessible by Non-secure accesses.
- If frame CNTEL0BaseN is implemented, the CNTEL0ACR in frame CNTBaseN provides additional control of accesses to frame CNTEL0BaseN.

Configurations

The power domain of CNTACR<n> is IMPLEMENTATION DEFINED.

For more information see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-7231.

Implemented only if the value of CNTTIDR.Frame<n> is 1.

An implementation of the counters might not provide configurable access to some or all of the features. In this case, the associated field in the CNTACR<n> register is:

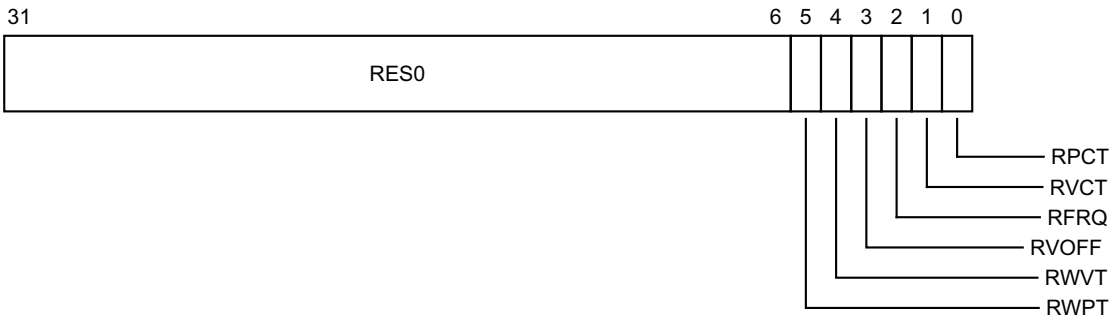
- RAZ/WI if access is always denied.
- RAO/WI if access is always permitted.

Attributes

CNTACR<n> is a 32-bit register.

Field descriptions

The CNTACR<n> bit assignments are:



Bits [31:6]

Reserved, RES0.

RWPT, bit [5]

Read/write access to the EL1 Physical Timer registers CNTP_CVAL, CNTP_TVAL, and CNTP_CTL, in frame <n>. The possible values of this bit are:

`0b0` No access to the EL1 Physical Timer registers in frame <n>. The registers are RES0.

`0b1` Read/write access to the EL1 Physical Timer registers in frame <n>.

This field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

RWVT, bit [4]

Read/write access to the Virtual Timer register CNTV_CVAL, CNTV_TVAL, and CNTV_CTL, in frame <n>. The possible values of this bit are:

`0b0` No access to the Virtual Timer registers in frame <n>. The registers are RES0.

`0b1` Read/write access to the Virtual Timer registers in frame <n>.

This field resets to an architecturally UNKNOWN value.

RVOFF, bit [3]

Read-only access to CNTVOFF, in frame <n>. The possible values of this bit are:

`0b0` No access to CNTVOFF in frame <n>. The register is RES0.

`0b1` Read-only access to CNTVOFF in frame <n>.

This field resets to an architecturally UNKNOWN value.

RFRQ, bit [2]

Read-only access to CNTFRQ, in frame <n>. The possible values of this bit are:

`0b0` No access to CNTFRQ in frame <n>. The register is RES0.

`0b1` Read-only access to CNTFRQ in frame <n>.

This field resets to an architecturally UNKNOWN value.

RVCT, bit [1]

Read-only access to CNTVCT, in frame <n>. The possible values of this bit are:

`0b0` No access to CNTVCT in frame <n>. The register is RES0.

`0b1` Read-only access to CNTVCT in frame <n>.

This field resets to an architecturally UNKNOWN value.

RPCT, bit [0]

Read-only access to CNTPCT, in frame <n>. The possible values of this bit are:

`0b0` No access to CNTPCT in frame <n>. The register is RES0.

`0b1` Read-only access to CNTPCT in frame <n>.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTACR<n>:

In a system that recognizes two Security states:

- CNTACR<n> is always accessible by Secure accesses.
- CNTNSAR.NS<n> determines whether CNTACR<n> is accessible by Non-secure accesses.

CNTACR<n> can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTCTLBase	`0x040` + 4n	CNTACR<n>

This interface is accessible as follows:

- Accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.7.2 CNTCR, Counter Control Register

The CNTCR characteristics are:

Purpose

Enables the counter, controls the counter frequency setting, and controls counter behavior during debug.

Configurations

The power domain of CNTCR is IMPLEMENTATION DEFINED.

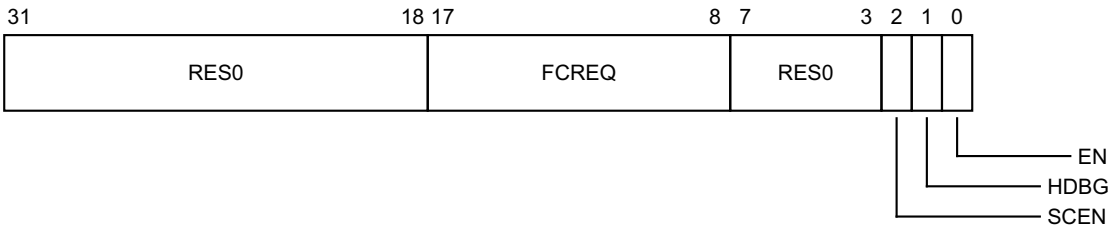
For more information see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-7231.

Attributes

CNTCR is a 32-bit register.

Field descriptions

The CNTCR bit assignments are:



Bits [31:18]

Reserved, RES0.

FCREQ, bits [17:8]

Frequency change request. Indicates the number of the entry in the Frequency modes table to select.

Selecting an unimplemented entry, or an entry that contains 0, has no effect on the counter.

The maximum number of entries in the Frequency modes table is IMPLEMENTATION DEFINED up to a maximum of 1004 entries, see *The Frequency modes table* on page I2-7233. An implementation is only required to implement an FCREQ field that can hold values from 0 to the highest supported Frequency modes table entry. Any unrequired most-significant bits of FCREQ can be implemented as RES0.

This field resets to `0`.

Bits [7:3]

Reserved, RES0.

SCEN, bit [2]

When ARMv8.4-CNTSC is implemented:

Scale Enable.

`0b0` Scaling is not enabled. The counter value is incremented by `0x1.0000000` for each counter tick.

`0b1` Scaling is enabled. The counter is incremented by CNTSCR.ScaleVal for each counter tick.

The SCEN bit can only be changed when the counter is disabled, when CNTCR.EN == 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the value of CNTCR.SCEN changes when CNTCR.EN == 1 then:

- The counter value becomes UNKNOWN.
- The counter value remains UNKNOWN on future ticks of the clock.

When the CNTCV register in the CNTControlBase frame of the memory mapped counter module is written to, the accumulated fraction information is reset to zero.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HDBG, bit [1]

Halt-on-debug. Controls whether a Halt-on-debug signal halts the system counter:

`0b0` System counter ignores Halt-on-debug.

`0b1` Asserted Halt-on-debug signal halts system counter update.

This field resets to an architecturally UNKNOWN value.

EN, bit [0]

Enables the counter:

`0b0` System counter disabled.

`0b1` System counter enabled.

This field resets to `0`.

Accessing the CNTCR:

In a system that supports Secure and Non-secure memory maps the CNTControlBase frame, that includes this register, is implemented only in the Secure memory map.

CNTCR can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTControlBase	`0x000`	CNTCR

This interface is accessible as follows:

- Accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.7.3 CNTCV, Counter Count Value register

The CNTCV characteristics are:

Purpose

Indicates the current count value.

Configurations

The power domain of CNTCV is IMPLEMENTATION DEFINED.

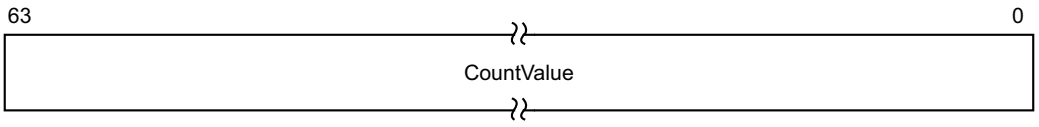
For more information see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-7231.

Attributes

CNTCV is a 64-bit register.

Field descriptions

The CNTCV bit assignments are:



CountValue, bits [63:0]

Indicates the counter value.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTCV:

Frame	Accessibility
CNTControlBase	RW
CNTReadBase	RO

A write to CNTCV must be visible in the CNTPCT register of each running processor in a finite time.

For the instance of the register in the CNTControlBase frame:

- In a system that supports Secure and Non-secure memory maps the CNTControlBase frame, and therefore this register instance, is implemented only in the Secure memory map.
- If the counter is enabled, the effect of writing to the register is UNKNOWN.

In an implementation that supports 64-bit atomic memory accesses, this register must be accessible using a 64-bit atomic access.

CNTCV[63:0] can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTControlBase	`0x008`	CNTCV	63:0

This interface is accessible as follows:

- Accesses to CNTCV[63:0] are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CNTCV[63:0] can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTReadBase	`0x000`	CNTCV	63:0

This interface is accessible as follows:

- Accesses to CNTCV[63:0] are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.7.4 CNTEL0ACR, Counter-timer EL0 Access Control Register

The CNTEL0ACR characteristics are:

Purpose

An implementation of CNTEL0ACR in the frame at CNTBaseN controls whether the CNTPCT, CNTVCT, CNTFRQ, EL1 Physical Timer, and Virtual Timer registers are visible in the frame at CNTEL0BaseN.

Configurations

The power domain of CNTEL0ACR is IMPLEMENTATION DEFINED.

Implementation of this register is OPTIONAL.

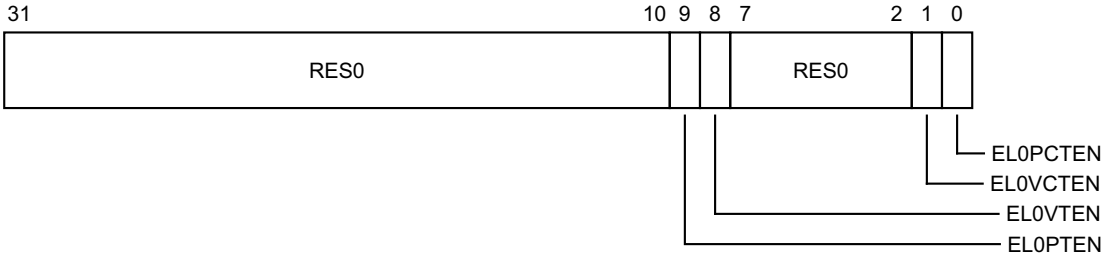
For more information see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-7231.

Attributes

CNTEL0ACR is a 32-bit register.

Field descriptions

The CNTEL0ACR bit assignments are:



Bits [31:10]

Reserved, RES0.

EL0PTEN, bit [9]

Second view read/write access control for the EL1 Physical Timer registers. This bit controls whether the CNTP_CVAL, CNTP_TVAL, and CNTP_CTL registers in the current CNTBaseN frame are also accessible in the corresponding CNTEL0BaseN frame. The possible values of this bit are:

`0b0` No access. Registers are RES0 in the second view.

`0b1` Access permitted. If the registers are accessible in the current frame then they are accessible in the second view.

This field resets to an architecturally UNKNOWN value.

EL0VTEN, bit [8]

Second view read/write access control for the Virtual Timer registers. This bit controls whether the CNTV_CVAL, CNTV_TVAL, and CNTV_CTL registers in the current CNTBaseN frame are also accessible in the corresponding CNTEL0BaseN frame. The possible values of this bit are:

`0b0` No access. Registers are RES0 in the second view.

`0b1` Access permitted. If the registers are accessible in the current frame then they are accessible in the second view.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The definition of this bit means that, if the Virtual Timer registers are not implemented in the current CNTBaseN frame, then the Virtual Timer register addresses are RES0 in the corresponding CNTEL0BaseN frame, regardless of the value of this bit.

This field resets to an architecturally UNKNOWN value.

Bits [7:2]

Reserved, RES0.

EL0VCTEN, bit [1]

Second view read access control for CNTVCT and CNTFRQ. The possible values of this bit are:

`0b0` CNTVCT is not visible in the second view.
If EL0PCTEN is set to 0, CNTFRQ is not visible in the second view.

`0b1` Access permitted. If CNTVCT and CNTFRQ are visible in the current frame then they are visible in the second view.

This field resets to an architecturally UNKNOWN value.

EL0PCTEN, bit [0]

Second view read access control for CNTPCT and CNTFRQ. The possible values of this bit are:

`0b0` CNTPCT is not visible in the second view.
If EL0VCTEN is set to 0, CNTFRQ is not visible in the second view.

`0b1` Access permitted. If CNTPCT and CNTFRQ are visible in the current frame then they are visible in the second view.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTEL0ACR:

CNTEL0ACR can be implemented in any implemented CNTBaseN frame.

CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames on page I2-7239 describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses.

If CNTEL0ACR is not implemented in an implemented CNTBaseN frame:

- The register location in that frame is RAZ/WI.
- If the corresponding CNTEL0BaseN frame is implemented, the registers CNTFRQ, CNTP_CTL, CNTP_CVAL, CNTP_TVAL, CNTPCT, CNTV_CTL, CNTV_CVAL, CNTV_TVAL, and CNTVCT are not visible in that frame.

CNTEL0ACR can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTBaseN	`0x014`	CNTEL0ACR

This interface is accessible as follows:

- Accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.7.5 CNTFID0, Counter Frequency ID

The CNTFID0 characteristics are:

Purpose

Indicates the base frequency of the system counter.

Configurations

The power domain of CNTFID0 is IMPLEMENTATION DEFINED.

For more information see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-7231.

The possible frequencies for the system counter are stored in the Frequency modes table as 32-bit words starting with the base frequency, CNTFID0. For more information see *The Frequency modes table* on page I2-7233.

The final entry in the Frequency modes table must be followed by a 32-bit word of zero value, to mark the end of the table.

Typically, the Frequency modes table will be in read-only memory. However, a system implementation might use read/write memory for the table, and initialize the table entries as part of its start-up sequence.

If the Frequency modes table is in read/write memory, Arm strongly recommends that the table is not updated once the system is running.

Attributes

CNTFID0 is a 32-bit register.

Field descriptions

The CNTFID0 bit assignments are:

31 … 0
Frequency

Frequency, bits [31:0]

The base frequency of the system counter, in Hz.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTFID0:

It is IMPLEMENTATION DEFINED whether this register is RO or RW

In a system that supports Secure and Non-secure memory maps the CNTControlBase frame, that includes this register, is implemented only in the Secure memory map.

CNTFID0 can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTControlBase	`0x020`	CNTFID0

This interface is accessible as follows:

- Accesses to this register are RO or RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.7.6 CNTFID<n>, Counter Frequency IDs, n > 0, n = 1 - 1003

The CNTFID<n> characteristics are:

Purpose

Indicates alternative system counter update frequencies.

Configurations

The power domain of CNTFID<n> is IMPLEMENTATION DEFINED.

For more information see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-7231.

The possible frequencies for the system counter are stored in the Frequency modes table as 32-bit words starting with the base frequency, CNTFID0, see *The Frequency modes table* on page I2-7233.

The number of CNTFID<n> registers is IMPLEMENTATION DEFINED, and the only required CNTFID<n> register is CNTFID0.

The final entry in the Frequency modes table must be followed by a 32-bit word of zero value, to mark the end of the table.

The architecture can support up to 1004 entries in the Frequency modes table, including the zero-word end marker, and the number of entries is IMPLEMENTATION DEFINED up to this limit. For an implementation that includes registers in the IMPLEMENTATION DEFINED register space `0x0C0-0x0FC`, the maximum number of entries in the Frequency modes table is 40, including the zero-word end marker.

Typically, the Frequency modes table will be in read-only memory. However, a system implementation might use read/write memory for the table, and initialize the table entries as part of its start-up sequence.

If the Frequency modes table is in read/write memory, Arm strongly recommends that the table is not updated once the system is running.

Attributes

CNTFID<n> is a 32-bit register.

Field descriptions

The CNTFID<n> bit assignments are:

31 … 0
Frequency

Frequency, bits [31:0]

A system counter update frequency, in Hz. Must be an exact divisor of the base frequency. Arm strongly recommends that all frequency values in the Frequency modes table are integer power-of-two divisors of the base frequency.

When the system timer is operating at a lower frequency than the base frequency, the increment applied at each counter update is given by:

increment = (base frequency) / (selected frequency)

This field resets to an architecturally UNKNOWN value.

Accessing the CNTFID<n>:

It is IMPLEMENTATION DEFINED whether this register is RO or RW

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In a system that supports Secure and Non-secure memory maps the CNTControlBase frame, that includes these registers, is implemented only in the Secure memory map.

CNTFID<n> can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTControlBase	`0x020` + 4n	CNTFID<n>

This interface is accessible as follows:

- Accesses to this register are RO or RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.7.7 CNTFRQ, Counter-timer Frequency

The CNTFRQ characteristics are:

Purpose

This register is provided so that software can discover the frequency of the system counter. The instance of the register in the CNTCTLBase frame must be programmed with this value as part of system initialization. The value of the register is not interpreted by hardware.

Configurations

The power domain of CNTFRQ is IMPLEMENTATION DEFINED.

For more information see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-7231.

Attributes

CNTFRQ is a 32-bit register.

Field descriptions

The CNTFRQ bit assignments are:

31	0
Clock frequency	

Bits [31:0]

Clock frequency. Indicates the system counter clock frequency, in Hz.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTFRQ:

CNTFRQ must be implemented as an RW register in the CNTCTLBase frame.

In a system that recognizes two Security states, the instance of the register in the CNTCTLBase frame is only accessible by Secure accesses.

CNTFRQ can be implemented as a RO register in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame.

CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames on page I2-7239 describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses.

For an implemented CNTBaseN frame:

- CNTFRQ is accessible in that frame, as a RO register, if the value of CNTACR<n>.RFRQ is 1.
- Otherwise, the CNTFRQ address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTFRQ is accessible as a RO register in that frame if both:
  - CNTFRQ is accessible in the corresponding CNTBaseN frame.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— Either the value of CNTEL0ACR.EL0VCTEN is 1 or the value of CNTEL0ACR.EL0PCTEN is 1.

- Otherwise, the CNTFRQ address in that frame is RAZ/WI.

CNTFRQ can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTBaseN	`0x010`	CNTFRQ

This interface is accessible as follows:

- Accesses to this register are RO.

CNTFRQ can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTEL0BaseN	`0x010`	CNTFRQ

This interface is accessible as follows:

- Accesses to this register are RO.

CNTFRQ can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTCTLBase	`0x000`	CNTFRQ

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.7.8 CNTID, Counter Identification Register

The CNTID characteristics are:

Purpose

Indicates whether counter scaling is implemented.

Configurations

The power domain of CNTID is IMPLEMENTATION DEFINED.

This register is present only when ARMv8.4-CNTSC is implemented. Otherwise, direct accesses to CNTID are RES0.

Attributes

CNTID is a 32-bit register.

Field descriptions

The CNTID bit assignments are:

31 … 4	3 … 0
RES0	CNTSC

Bits [31:4]

Reserved, RES0.

CNTSC, bits [3:0]

Indicates whether Counter Scaling is implemented

`0b0000` Counter scaling is not implemented.

`0b0001` Counter scaling is implemented.

All other values are reserved.

Accessing the CNTID:

In a system that supports Secure and Non-secure memory maps, the CNTControlBase frame, that includes this register, is implemented only in the Secure memory map.

CNTID can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTControlBase	`0x1C`	CNTID

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.7.9 CNTNSAR, Counter-timer Non-secure Access Register

The CNTNSAR characteristics are:

Purpose

Provides the highest-level control of whether frames CNTBaseN and CNTEL0BaseN are accessible by Non-secure accesses.

Configurations

The power domain of CNTNSAR is IMPLEMENTATION DEFINED.

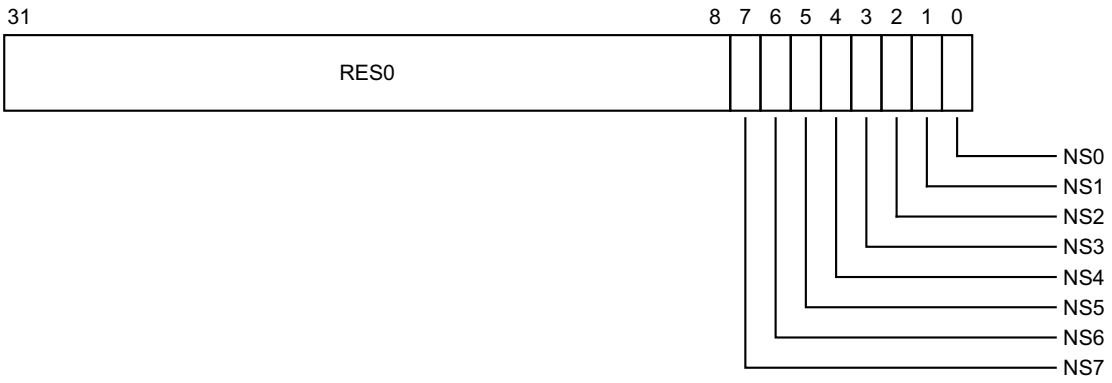
For more information see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-7231.

Attributes

CNTNSAR is a 32-bit register.

Field descriptions

The CNTNSAR bit assignments are:



Bits [31:8]

Reserved, RES0.

NS<n>, bit [n], for n = 0 to 7

Non-secure access to frame n. The possible values of this bit are:

`0b0` Secure access only. Behaves as RES0 to Non-secure accesses.

`0b1` Secure and Non-secure accesses permitted.

This bit also determines whether, in the CNTCTLBase frame, CNTACR<n> and CNTVOFF<n> are accessible to Non-secure accesses.

If frame CNTBase<n>:

- Is not implemented, then NS<n> is RES0.
- Is not Configurable access, and is accessible only by Secure accesses, then NS<n> is RES0.
- Is not Configurable access, and is accessible by both Secure and Non-secure accesses, then NS<n> is RES1.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTNSAR:

In a system that recognizes two Security states, this register is only accessible by Secure accesses.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CNTNSAR can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTCTLBase	`0x004`	CNTNSAR

This interface is accessible as follows:

- Accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.7.10 CNTP_CTL, Counter-timer Physical Timer Control

The CNTP_CTL characteristics are:

Purpose

Control register for the EL1 physical timer.

Configurations

The power domain of CNTP_CTL is IMPLEMENTATION DEFINED.

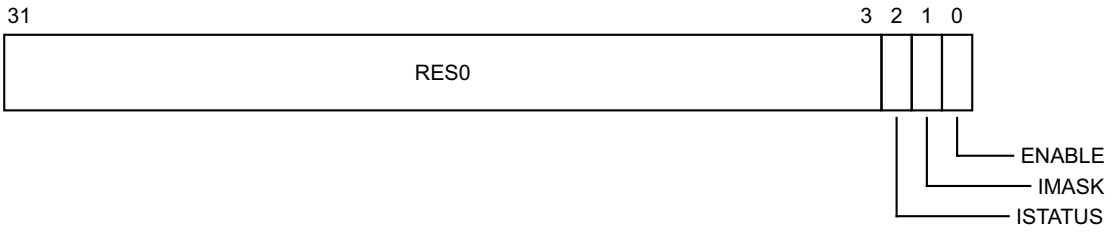
For more information see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-7231.

Attributes

CNTP_CTL is a 32-bit register.

Field descriptions

The CNTP_CTL bit assignments are:



Bits [31:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

`0b0` Timer condition is not met.

`0b1` Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

For more information see *Operation of the CompareValue views of the timers* on page D11-2847 and *Operation of the TimerValue views of the timers* on page D11-2848.

This bit is read-only.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

`0b0` Timer interrupt is not masked by the IMASK bit.

`0b1` Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

This field resets to an architecturally UNKNOWN value.

ENABLE, bit [0]

Enables the timer. Permitted values are:

`0b0` Timer disabled.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTP_TVAL continues to count down.

Note

Disabling the output signal might be a power-saving option.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTP_CTL:

CNTP_CTL can be implemented in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame.

CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames on page I2-7239 describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses.

For an implemented CNTBaseN frame:

- CNTP_CTL is accessible in that frame if the value of CNTACR<n>.RWPT is 1.
- Otherwise, the CNTP_CTL address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTP_CTL is accessible in that frame if both:
  - CNTP_CTL is accessible in the corresponding CNTBaseN frame:
  - The value of CNTEL0ACR.EL0PTEN is 1.
- Otherwise, the CNTP_CTL address in that frame is RAZ/WI.

CNTP_CTL can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTBaseN	`0x02C`	CNTP_CTL

This interface is accessible as follows:

- Accesses to this register are RW.

CNTP_CTL can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTEL0BaseN	`0x02C`	CNTP_CTL

This interface is accessible as follows:

- Accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.7.11 CNTP_CVAL, Counter-timer Physical Timer CompareValue

The CNTP_CVAL characteristics are:

Purpose

Holds the 64-bit compare value for the EL1 physical timer.

Configurations

The power domain of CNTP_CVAL is IMPLEMENTATION DEFINED.

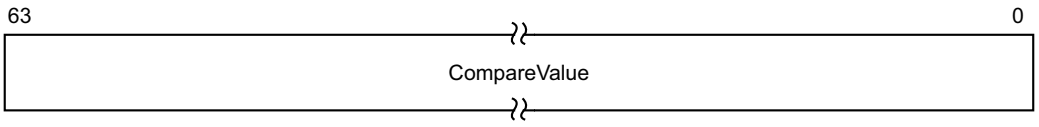
For more information see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-7231.

Attributes

CNTP_CVAL is a 64-bit register.

Field descriptions

The CNTP_CVAL bit assignments are:



CompareValue, bits [63:0]

Holds the EL1 physical timer CompareValue.

When CNTP_CTL.ENABLE is 1, the timer condition is met when (CNTPCT - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTP_CTL.ISTATUS is set to 1.
- An interrupt is generated if CNTP_CTL.IMASK is 0.

When CNTP_CTL.ENABLE is 0, the timer condition is not met, but CNTPCT continues to count.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTP_CVAL:

CNTP_CVAL can be implemented in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame.

CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames on page I2-7239 describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses.

For an implemented CNTBaseN frame:

- CNTP_CVAL is accessible in that frame if the value of CNTACR<n>.RWPT is 1.
- Otherwise, the CNTP_CVAL address in that frame is RAZ/WI.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For an implemented CNTEL0BaseN frame:

- CNTP_CVAL is accessible in that frame if both:
  - CNTP_CVAL is accessible in the corresponding CNTBaseN frame:
  - The value of CNTEL0ACR.EL0PTEN is 1.
- Otherwise, the CNTP_CVAL address in that frame is RAZ/WI.

If the implementation supports 64-bit atomic accesses, then the CNTP_CVAL register must be accessible as an atomic 64-bit value.

CNTP_CVAL[31:0] can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	`0x020`	CNTP_CVAL	31:0

This interface is accessible as follows:

- Accesses to CNTP_CVAL[31:0] are RW.

CNTP_CVAL[31:0] can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	`0x020`	CNTP_CVAL	31:0

This interface is accessible as follows:

- Accesses to CNTP_CVAL[31:0] are RW.

CNTP_CVAL[63:32] can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	`0x024`	CNTP_CVAL	63:32

This interface is accessible as follows:

- Accesses to CNTP_CVAL[63:32] are RW.

CNTP_CVAL[63:32] can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	`0x024`	CNTP_CVAL	63:32

This interface is accessible as follows:

- Accesses to CNTP_CVAL[63:32] are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.7.12 CNTP_TVAL, Counter-timer Physical Timer TimerValue

The CNTP_TVAL characteristics are:

Purpose

Holds the timer value for the EL1 physical timer.

Configurations

The power domain of CNTP_TVAL is IMPLEMENTATION DEFINED.

For more information see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-7231.

Attributes

CNTP_TVAL is a 32-bit register.

Field descriptions

The CNTP_TVAL bit assignments are:

31 … 0
TimerValue

TimerValue, bits [31:0]

The TimerValue view of the EL1 physical timer.

On a read of this register:

- If CNTP_CTL.ENABLE is 0, the value returned is UNKNOWN.
- If CNTP_CTL.ENABLE is 1, the value returned is (CompareValue - CNTPCT).

On a write of this register, CompareValue is set to (CNTPCT + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTP_CTL.ENABLE is 1, the timer condition is met when (CNTPCT - CompareValue) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTP_CTL.ISTATUS is set to 1.
- If CNTP_CTL.IMASK is 0, an interrupt is generated.

When CNTP_CTL.ENABLE is 0, the timer condition is not met, but CNTPCT continues to count, so the TimerValue view appears to continue to count down.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTP_TVAL:

CNTP_TVAL can be implemented in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame.

CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames on page I2-7239 describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For an implemented CNTBaseN frame:

- CNTP_TVAL is accessible in that frame if the value of CNTACR<n>.RWPT is 1.
- Otherwise, the CNTP_TVAL address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTP_TVAL is accessible in that frame if both:
  - CNTP_TVAL is accessible in the corresponding CNTBaseN frame:
  - The value of CNTEL0ACR.EL0PTEN is 1.
- Otherwise, the CNTP_TVAL address in that frame is RAZ/WI.

CNTP_TVAL can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTBaseN	`0x028`	CNTP_TVAL

This interface is accessible as follows:

- Accesses to this register are RW.

CNTP_TVAL can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTEL0BaseN	`0x028`	CNTP_TVAL

This interface is accessible as follows:

- Accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.7.13 CNTPCT, Counter-timer Physical Count

The CNTPCT characteristics are:

Purpose

Holds the 64-bit physical count value.

Configurations

The power domain of CNTPCT is IMPLEMENTATION DEFINED.

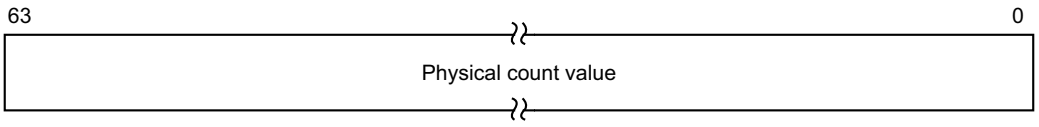
For more information see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-7231.

Attributes

CNTPCT is a 64-bit register.

Field descriptions

The CNTPCT bit assignments are:



Bits [63:0]

Physical count value.

Accessing the CNTPCT:

CNTPCT can be implemented in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame, as a RO register.

CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames on page I2-7239 describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses.

For an implemented CNTBaseN frame:

- CNTPCT is accessible in that frame, as a RO register, if the value of CNTACR<n>.RPCT is 1.
- Otherwise, the CNTPCT address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTPCT is accessible in that frame if both:
  - CNTPCT is accessible in the corresponding CNTBaseN frame.
  - The value of CNTEL0ACR.EL0PCTEN is 1.
- Otherwise, the CNTPCT address in that frame is RAZ/WI.

If the implementation supports 64-bit atomic accesses, then the CNTPCT register must be accessible as an atomic 64-bit value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CNTPCT[31:0] can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	`0x000`	CNTPCT	31:0

This interface is accessible as follows:

- Accesses to CNTPCT[31:0] are RO.

CNTPCT[31:0] can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	`0x000`	CNTPCT	31:0

This interface is accessible as follows:

- Accesses to CNTPCT[31:0] are RO.

CNTPCT[63:32] can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	`0x004`	CNTPCT	63:32

This interface is accessible as follows:

- Accesses to CNTPCT[63:32] are RO.

CNTPCT[63:32] can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	`0x004`	CNTPCT	63:32

This interface is accessible as follows:

- Accesses to CNTPCT[63:32] are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.7.14 CNTSCR, Counter Scale Register

The CNTSCR characteristics are:

Purpose

Enables the counter, controls the counter frequency setting, and controls counter behavior during debug.

Configurations

The power domain of CNTSCR is IMPLEMENTATION DEFINED.

This register is present only when ARMv8.4-CNTSC is implemented. Otherwise, direct accesses to CNTSCR are RES0.

For more information see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-7231.

Attributes

CNTSCR is a 32-bit register.

Field descriptions

The CNTSCR bit assignments are:

31 … 0
ScaleVal

ScaleVal, bits [31:0]

Scale Value

When counter scaling is enabled, ScaleVal is the amount added to the counter value for every counter tick.

Counter tick is defined as one period of the current operating frequency of the Generic counter.

ScaleVal is expressed as an unsigned fixed point number with an 8-bit integer value and a 24-bit fractional value.

CNTSCR.ScaleVal can only be changed when CNTCR.EN == 0. If the value of this field is changed when CNTCR.EN == 1:

- The counter value becomes UNKNOWN.
- The counter value remains UNKNOWN on future ticks of the clock.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTSCR:

In a system that supports Secure and Non-secure memory maps the CNTControlBase frame, that includes this register, is implemented only in the Secure memory map.

CNTSCR can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTControlBase	`0x10`	CNTSCR

This interface is accessible as follows:

- Accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.7.15 CNTSR, Counter Status Register

The CNTSR characteristics are:

Purpose

Provides counter frequency status information.

Configurations

The power domain of CNTSR is IMPLEMENTATION DEFINED.

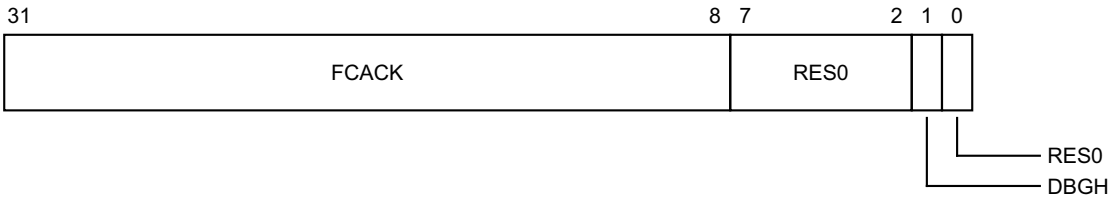
For more information see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-7231.

Attributes

CNTSR is a 32-bit register.

Field descriptions

The CNTSR bit assignments are:



FCACK, bits [31:8]

Frequency change acknowledge. Indicates the currently selected entry in the Frequency modes table, see *The Frequency modes table* on page I2-7233.

This field resets to `0`.

Bits [7:2]

Reserved, RES0.

DBGH, bit [1]

Indicates whether the counter is halted because the Halt-on-debug signal is asserted:

`0b0` Counter is not halted.

`0b1` Counter is halted.

This field resets to an architecturally UNKNOWN value.

Bit [0]

Reserved, RES0.

Accessing the CNTSR:

In a system that supports Secure and Non-secure memory maps the CNTControlBase frame, that includes this register, is implemented only in the Secure memory map.

CNTSR can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTControlBase	`0x004`	CNTSR

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.7.16 CNTTIDR, Counter-timer Timer ID Register

The CNTTIDR characteristics are:

Purpose

Indicates the implemented timers in the memory map, and their features. For each value of N from 0 to 7 it indicates whether:

- Frame CNTBaseN is a view of an implemented timer.
- Frame CNTBaseN has a second view, CNTEL0BaseN.
- Frame CNTBaseN has a virtual timer capability.

Configurations

The power domain of CNTTIDR is IMPLEMENTATION DEFINED.

For more information see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-7231.

Attributes

CNTTIDR is a 32-bit register.

Field descriptions

The CNTTIDR bit assignments are:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
Frame7	Frame6	Frame5	Frame4	Frame3	Frame2	Frame1	Frame0

Frame<n>, bits [4n+3:4n], for n = 0 to 7

A 4-bit field indicating the features of frame CNTBase<n>.

Bit[3] of the field is RES0.

Bit[2], the FEL0 subfield, indicates whether frame CNTBase<n> has a second view, CNTEL0Base<n>. The possible values of this bit are:

Bit[2]	Meaning
`0b0`	Frame<n> does not have a second view. The CNTEL0ACR register in the first view of the frame is RES0
`0b1`	Frame<n> has a second view, CNTEL0Base<n>.

If bit[0] is 0, bit[2] is RES0.

Bit[1], the FVI subfield, indicates whether both:

- Frame CNTBase<n> implements the virtual timer registers CNTV_CVAL, CNTV_TVAL, and CNTV_CTL.
- This CNTCTLBase frame implements the virtual timer offset register CNTVOFF<n>.

The possible values of bit[1] are:

Bit[1]	Meaning
`0b0`	Frame<n> does not have virtual capability. The virtual time and offset registers are RES0.
`0b1`	Frame<n> has virtual capability. The virtual time and offset registers are implemented

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If bit[0] is 0, bit[1] is RES0.

Bit[0], the FI subfield, indicates whether frame CNTBase<n> is implemented. The possible values of this bit are:

Bit[0]	Meaning
`0b0`	Frame<n> is not implemented. All registers associated with the frame are RES0.
`0b1`	Frame<n> is implemented

Accessing the CNTTIDR:

In a system that recognizes two Security states this register is accessible by both Secure and Non-secure accesses.

CNTTIDR can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTCTLBase	`0x008`	CNTTIDR

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.7.17 CNTV_CTL, Counter-timer Virtual Timer Control

The CNTV_CTL characteristics are:

Purpose

Control register for the virtual timer.

Configurations

The power domain of CNTV_CTL is IMPLEMENTATION DEFINED.

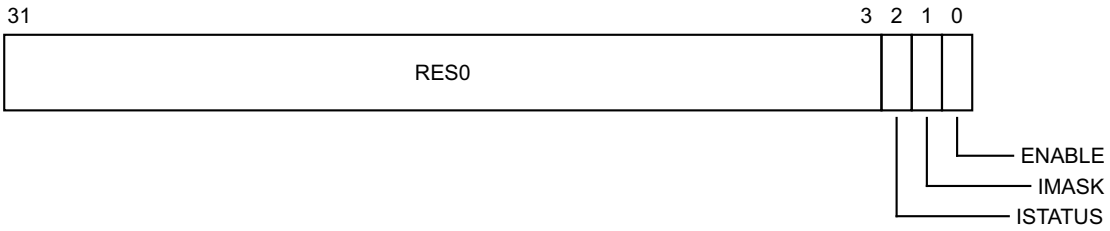
For more information see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-7231.

Attributes

CNTV_CTL is a 32-bit register.

Field descriptions

The CNTV_CTL bit assignments are:



Bits [31:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

`0b0` Timer condition is not met.

`0b1` Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

For more information see *Operation of the CompareValue views of the timers* on page D11-2847 and *Operation of the TimerValue views of the timers* on page D11-2848.

This bit is read-only.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

`0b0` Timer interrupt is not masked by the IMASK bit.

`0b1` Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

This field resets to an architecturally UNKNOWN value.

ENABLE, bit [0]

Enables the timer. Permitted values are:

`0b0` Timer disabled.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTV_TVAL continues to count down.

Note

Disabling the output signal might be a power-saving option.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTV_CTL:

CNTV_CTL can be implemented in any implemented CNTBaseN frame that has virtual timer capability, and in the corresponding CNTEL0BaseN frame.

CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames on page I2-7239 describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses.

For an implemented CNTBaseN frame that has virtual timer capability:

- CNTV_CTL is accessible in that frame if the value of CNTACR<n>.RWVT is 1.
- Otherwise, the CNTV_CTL address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTV_CTL is accessible in that frame if both:
  - CNTV_CTL is accessible in the corresponding CNTBaseN frame:
  - The value of CNTEL0ACR.EL0VTEN is 1.
- Otherwise, the CNTV_CTL address in that frame is RAZ/WI.

CNTV_CTL can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTBaseN	`0x03C`	CNTV_CTL

This interface is accessible as follows:

- Accesses to this register are RW.

CNTV_CTL can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTEL0BaseN	`0x03C`	CNTV_CTL

This interface is accessible as follows:

- Accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.7.18 CNTV_CVAL, Counter-timer Virtual Timer CompareValue

The CNTV_CVAL characteristics are:

Purpose

Holds the 64-bit compare value for the virtual timer.

Configurations

The power domain of CNTV_CVAL is IMPLEMENTATION DEFINED.

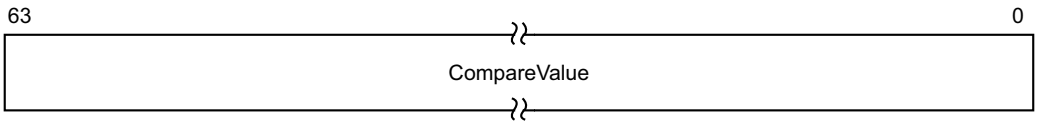
For more information see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-7231.

Attributes

CNTV_CVAL is a 64-bit register.

Field descriptions

The CNTV_CVAL bit assignments are:



CompareValue, bits [63:0]

Holds the virtual timer CompareValue.

When CNTV_CTL.ENABLE is 1, the timer condition is met when (CNTVCT - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTV_CTL.ISTATUS is set to 1.
- An interrupt is generated if CNTV_CTL.IMASK is 0.

When CNTV_CTL.ENABLE is 0, the timer condition is not met, but CNTVCT continues to count.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTV_CVAL:

CNTV_CVAL can be implemented in any implemented CNTBaseN frame that has virtual timer capability, and in the corresponding CNTEL0BaseN frame.

CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames on page I2-7239 describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses.

For an implemented CNTBaseN frame that has virtual timer capability:

- CNTV_CVAL is accessible in that frame if the value of CNTACR<n>.RWVT is 1.
- Otherwise, the CNTV_CVAL address in that frame is RAZ/WI.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For an implemented CNTEL0BaseN frame:

- CNTV_CVAL is accessible in that frame if both:
  - CNTV_CVAL is accessible in the corresponding CNTBaseN frame:
  - The value of CNTEL0ACR.EL0VTEN is 1.
- Otherwise, the CNTV_CVAL address in that frame is RAZ/WI.

If the implementation supports 64-bit atomic accesses, then the CNTV_CVAL register must be accessible as an atomic 64-bit value.

CNTV_CVAL[31:0] can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	`0x030`	CNTV_CVAL	31:0

This interface is accessible as follows:

- Accesses to CNTV_CVAL[31:0] are RW.

CNTV_CVAL[31:0] can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	`0x030`	CNTV_CVAL	31:0

This interface is accessible as follows:

- Accesses to CNTV_CVAL[31:0] are RW.

CNTV_CVAL[63:32] can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	`0x034`	CNTV_CVAL	63:32

This interface is accessible as follows:

- Accesses to CNTV_CVAL[63:32] are RW.

CNTV_CVAL[63:32] can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	`0x034`	CNTV_CVAL	63:32

This interface is accessible as follows:

- Accesses to CNTV_CVAL[63:32] are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.7.19 CNTV_TVAL, Counter-timer Virtual Timer TimerValue

The CNTV_TVAL characteristics are:

Purpose

Holds the timer value for the virtual timer.

Configurations

The power domain of CNTV_TVAL is IMPLEMENTATION DEFINED.

For more information see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-7231.

Attributes

CNTV_TVAL is a 32-bit register.

Field descriptions

The CNTV_TVAL bit assignments are:

31 … 0
TimerValue

TimerValue, bits [31:0]

The TimerValue view of the virtual timer.

On a read of this register:

- If CNTV_CTL.ENABLE is 0, the value returned is UNKNOWN.
- If CNTV_CTL.ENABLE is 1, the value returned is (CompareValue - CNTVCT).

On a write of this register, CompareValue is set to (CNTVCT + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTV_CTL.ENABLE is 1, the timer condition is met when (CNTVCT - CompareValue) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTV_CTL.ISTATUS is set to 1.
- If CNTV_CTL.IMASK is 0, an interrupt is generated.

When CNTV_CTL.ENABLE is 0, the timer condition is not met, but CNTVCT continues to count, so the TimerValue view appears to continue to count down.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTV_TVAL:

CNTV_TVAL can be implemented in any implemented CNTBaseN frame that has virtual timer capability, and in the corresponding CNTEL0BaseN frame.

CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames on page I2-7239 describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For an implemented CNTBaseN frame that has virtual timer capability:

- CNTV_TVAL is accessible in that frame if the value of CNTACR<n>.RWVT is 1.
- Otherwise, the CNTV_TVAL address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTV_TVAL is accessible in that frame if both:
  - CNTV_TVAL is accessible in the corresponding CNTBaseN frame:
  - The value of CNTEL0ACR.EL0VTEN is 1.
- Otherwise, the CNTV_TVAL address in that frame is RAZ/WI.

CNTV_TVAL can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTBaseN	`0x038`	CNTV_TVAL

This interface is accessible as follows:

- Accesses to this register are RW.

CNTV_TVAL can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTEL0BaseN	`0x038`	CNTV_TVAL

This interface is accessible as follows:

- Accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.7.20 CNTVCT, Counter-timer Virtual Count

The CNTVCT characteristics are:

Purpose

Holds the 64-bit virtual count value.

Configurations

The power domain of CNTVCT is IMPLEMENTATION DEFINED.

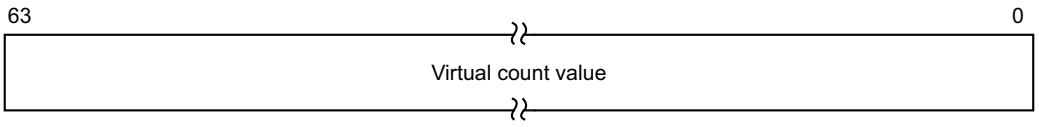
For more information see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-7231.

Attributes

CNTVCT is a 64-bit register.

Field descriptions

The CNTVCT bit assignments are:



Bits [63:0]

Virtual count value.

Accessing the CNTVCT:

CNTVCT can be implemented in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame, as a RO register.

CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames on page I2-7239 describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses.

For an implemented CNTBaseN frame:

- CNTVCT is accessible in that frame, as a RO register, if the value of CNTACR<n>.RVCT is 1.
- Otherwise, the CNTVCT address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTVCT is accessible in that frame if both:
  - CNTVCT is accessible in the corresponding CNTBaseN frame.
  - The value of CNTEL0ACR.EL0VCTEN is 1.
- Otherwise, the CNTVCT address in that frame is RAZ/WI.

If the implementation supports 64-bit atomic accesses, then the CNTVCT register must be accessible as an atomic 64-bit value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CNTVCT[31:0] can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	`0x008`	CNTVCT	31:0

This interface is accessible as follows:

- Accesses to CNTVCT[31:0] are RO.

CNTVCT[31:0] can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	`0x008`	CNTVCT	31:0

This interface is accessible as follows:

- Accesses to CNTVCT[31:0] are RO.

CNTVCT[63:32] can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	`0x00C`	CNTVCT	63:32

This interface is accessible as follows:

- Accesses to CNTVCT[63:32] are RO.

CNTVCT[63:32] can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	`0x00C`	CNTVCT	63:32

This interface is accessible as follows:

- Accesses to CNTVCT[63:32] are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.7.21 CNTVOFF, Counter-timer Virtual Offset

The CNTVOFF characteristics are:

Purpose

Holds the 64-bit virtual offset for a CNTBaseN frame that has virtual timer capability. This is the offset between real time and virtual time.

Configurations

The power domain of CNTVOFF is IMPLEMENTATION DEFINED.

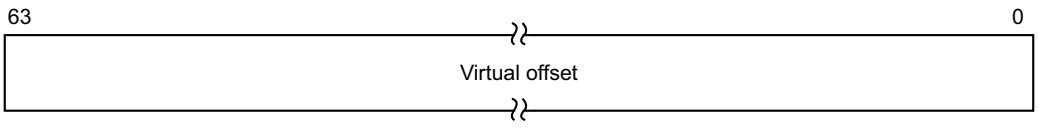
For more information see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-7231.

Attributes

CNTVOFF is a 64-bit register.

Field descriptions

The CNTVOFF bit assignments are:



Bits [63:0]

Virtual offset.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTVOFF:

CNTVOFF is implemented, as a RO register, in any implemented CNTBaseN frame that has virtual timer capability.

CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames on page I2-7239 describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses.

For an implemented CNTBaseN frame that has virtual timer capability:

- CNTVOFF is accessible in that frame, as a RO register, if the value of CNTACR<n>.RVOFF is 1.
- Otherwise, the CNTVOFF address in that frame is RAZ/WI.

Note

CNTVOFF is never visible in any CNTEL0BaseN frame. This means that the CNTVOFF address in any implemented CNTEL0BaseN frame is RAZ/WI.

In an implementation that supports 64-bit atomic accesses, a CNTVOFF{<n>} register must be accessible as an atomic 64-bit value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CNTVOFF[31:0] can be accessed through its memory-mapped interface:

Component	Frame	Offset	Range
Timer	CNTBaseN	`0x018`	31:0

This interface is accessible as follows:

- Accesses to CNTVOFF[31:0] are RO.

CNTVOFF[63:32] can be accessed through its memory-mapped interface:

Component	Frame	Offset	Range
Timer	CNTBaseN	`0x01C`	63:32

This interface is accessible as follows:

- Accesses to CNTVOFF[63:32] are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.7.22 CNTVOFF<n>, Counter-timer Virtual Offsets, n = 0 - 7

The CNTVOFF<n> characteristics are:

Purpose

Holds the 64-bit virtual offset for frame CNTBase<n>. This is the offset between real time and virtual time.

Configurations

The implementation of CNTVOFF<n> is OPTIONAL.

The power domain of CNTVOFF<n> is IMPLEMENTATION DEFINED.

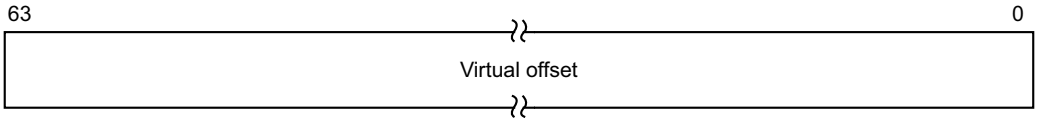
For more information see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-7231.

Attributes

CNTVOFF<n> is a 64-bit register.

Field descriptions

The CNTVOFF<n> bit assignments are:



Bits [63:0]

Virtual offset.

This field resets to an architecturally UNKNOWN value.

Accessing the CNTVOFF<n>:

In the CNTCTLBase frame a CNTVOFF<n> register must be implemented, as a RW register, for each CNTBaseN frame that has virtual timer capability. For more information see *CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames* on page I2-7239.

Note

The value of <n> in an instance of CNTVOFF<n> specifies the value of N for the associated CNTBaseN frame.

In a system that recognizes two Security states, for any CNTVOFF<n> register in the CNTCTLBase frame:

- CNTVOFF<n> is always accessible by Secure accesses.
- CNTNSAR.NS<n> determines whether CNTVOFF<n> is accessible by Non-secure accesses.

The register location of any unimplemented CNTVOFF<n> register in the CNTCTLBase frame is RAZ/WI.

The CNTVOFF<n> register is accessible in the CNTBaseN frame using CNTVOFF.

In an implementation that supports 64-bit atomic accesses, then the CNTVOFF<n> registers must be accessible as atomic 64-bit values.

CNTVOFF<n>[31:0] can be accessed through its memory-mapped interface:

Component	Frame	Offset	Range
Timer	CNTCTLBase	`0x080` + 8n	31:0

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This interface is accessible as follows:

- Accesses to CNTVOFF<n>[31:0] are RW.

CNTVOFF<n>[63:32] can be accessed through its memory-mapped interface:

Component	Frame	Offset	Range
Timer	CNTCTLBase	`0x084` + 8n	63:32

This interface is accessible as follows:

- Accesses to CNTVOFF<n>[63:32] are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.7.23 CounterID<n>, Counter ID registers, n = 0 - 11

The CounterID<n> characteristics are:

Purpose

IMPLEMENTATION DEFINED identification registers 0 to 11 for the memory-mapped Generic Timer.

Configurations

The power domain of CounterID<n> is IMPLEMENTATION DEFINED.

For more information see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-7231.

These registers are implemented independently in each of the implemented Generic Timer memory-mapped frames.

If the implementation of the Counter ID registers requires an architecture version, the value for this version of the Arm Generic Timer is version 0.

The Counter ID registers can be implemented as a set of CoreSight ID registers, comprising Peripheral ID Registers and Component ID Registers. An implementation of these registers for the Generic Timer must use a Component class value of `0xF`.

Attributes

CounterID<n> is a 32-bit register.

Field descriptions

The CounterID<n> bit assignments are:

31 … 0
IMPLEMENTATION DEFINED

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

Accessing the CounterID<n>:

These registers must be implemented, as RO registers, in every implemented Generic Timer memory-mapped frame.

For the CNTCTLBase frame, in a system that recognizes two Security states these registers are accessible by both Secure and Non-secure accesses.

For the CNTControlBase frame, in a system that supports Secure and Non-secure memory maps the frame is implemented only in the Secure memory map, meaning these registers are implemented only in the Secure memory map.

For the CNTBaseN frames, *CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames* on page I2-7239 describes the status fields that identify whether a frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CounterID<n> can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTControlBase	`0xFD0` + 4n	CounterID<n>

This interface is accessible as follows:

- Accesses to this register are RO.

CounterID<n> can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTReadBase	`0xFD0` + 4n	CounterID<n>

This interface is accessible as follows:

- Accesses to this register are RO.

CounterID<n> can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTBaseN	`0xFD0` + 4n	CounterID<n>

This interface is accessible as follows:

- Accesses to this register are RO.

CounterID<n> can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTEL0BaseN	`0xFD0` + 4n	CounterID<n>

This interface is accessible as follows:

- Accesses to this register are RO.

CounterID<n> can be accessed through its memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTCTLBase	`0xFD0` + 4n	CounterID<n>

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8 RAS Registers

This section describes the RAS registers.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.1 ERRCIDR0, Component Identification Register 0

The ERRCIDR0 characteristics are:

Purpose

Provides discovery information about the component.

For more information, see *About the Peripheral identification scheme* on page K2-7889.

Configurations

Implementation of this register is OPTIONAL.

Attributes

ERRCIDR0 is a 32-bit register.

Field descriptions

The ERRCIDR0 bit assignments are:

31 … 8	7 … 0
RES0	PRMBL_0

Bits [31:8]

Reserved, RES0.

PRMBL_0, bits [7:0]

Component identification preamble, segment 0.

This field reads as `0x0D`.

Accessing the ERRCIDR0:

ERRCIDR0 can be accessed through its memory-mapped interface:

Component	Offset
RAS	`0xFF0`

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.2 ERRCIDR1, Component Identification Register 1

The ERRCIDR1 characteristics are:

Purpose

Provides discovery information about the component.

For more information, see *About the Peripheral identification scheme* on page K2-7889.

Configurations

Implementation of this register is OPTIONAL.

Attributes

ERRCIDR1 is a 32-bit register.

Field descriptions

The ERRCIDR1 bit assignments are:

31 … 8	7 … 4	3 … 0
RES0	CLASS	PRMBL_1

Bits [31:8]

Reserved, RES0.

CLASS, bits [7:4]

Component class.

`0b1111` Generic peripheral with IMPLEMENTATION DEFINED register layout.

Other values are defined by the CoreSight Architecture.

This field reads as `0xF`.

PRMBL_1, bits [3:0]

Component identification preamble, segment 1.

This field reads as `0x0`.

Accessing the ERRCIDR1:

ERRCIDR1 can be accessed through its memory-mapped interface:

Component	Offset
RAS	`0xFF4`

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.3 ERRCIDR2, Component Identification Register 2

The ERRCIDR2 characteristics are:

Purpose

Provides discovery information about the component.

For more information, see *About the Peripheral identification scheme* on page K2-7889.

Configurations

Implementation of this register is OPTIONAL.

Attributes

ERRCIDR2 is a 32-bit register.

Field descriptions

The ERRCIDR2 bit assignments are:

31 … 8	7 … 0
RES0	PRMBL_2

Bits [31:8]

Reserved, RES0.

PRMBL_2, bits [7:0]

Component identification preamble, segment 2.

This field reads as `0x05`.

Accessing the ERRCIDR2:

ERRCIDR2 can be accessed through its memory-mapped interface:

Component	Offset
RAS	`0xFF8`

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.4 ERRCIDR3, Component Identification Register 3

The ERRCIDR3 characteristics are:

Purpose

Provides discovery information about the component.

For more information, see *About the Peripheral identification scheme* on page K2-7889.

Configurations

Implementation of this register is OPTIONAL.

Attributes

ERRCIDR3 is a 32-bit register.

Field descriptions

The ERRCIDR3 bit assignments are:

31 … 8	7 … 0
RES0	PRMBL_3

Bits [31:8]

Reserved, RES0.

PRMBL_3, bits [7:0]

Component identification preamble, segment 3.

This field reads as `0xB1`.

Accessing the ERRCIDR3:

ERRCIDR3 can be accessed through its memory-mapped interface:

Component	Offset
RAS	`0xFFC`

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.5 ERRCRICR0, Critical Error Interrupt Configuration Register 0

The ERRCRICR0 characteristics are:

Purpose

Interrupt configuration register.

Configurations

External register ERRCRICR0 is architecturally mapped to External register ERRIRQCR<n>.

This register is present only when ARMv8.4-RAS is implemented. Otherwise, direct accesses to ERRCRICR0 are RES0.

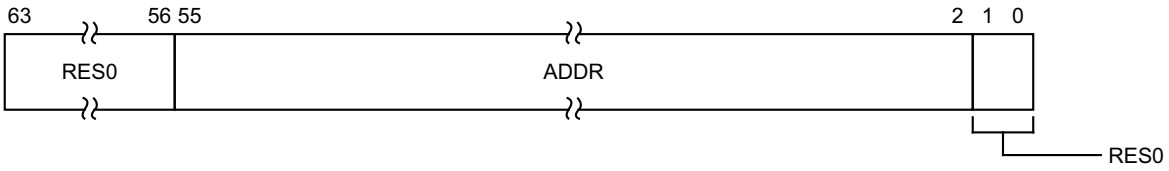
Present only if interrupt configuration registers use the recommended format. Otherwise, this register is RES0.

Attributes

ERRCRICR0 is a 64-bit register.

Field descriptions

The ERRCRICR0 bit assignments are:



Bits [63:56]

Reserved, RES0.

ADDR, bits [55:2]

Message Signaled Interrupt address.

Specifies the address that the component writes to when signaling an interrupt.

The size of a physical address is IMPLEMENTATION DEFINED. Unimplemented high-order physical address bits are RES0.

The following resets apply:

- On a Error recovery reset, this field resets to an architecturally UNKNOWN value.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [1:0]

Reserved, RES0.

Accessing the ERRCRICR0:

ERRCRICR0 can be accessed through its memory-mapped interface:

Component	Offset	Instance
RAS	`0xEA0`	ERRCRICR0

This interface is accessible as follows:

- Accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.6 ERRCRICR1, Critical Error Interrupt Configuration Register 1

The ERRCRICR1 characteristics are:

Purpose

Interrupt configuration register.

Configurations

External register ERRCRICR1[31:0] is architecturally mapped to External register ERRIRQCR<n>[31:0].

This register is present only when ARMv8.4-RAS is implemented. Otherwise, direct accesses to ERRCRICR1 are RES0.

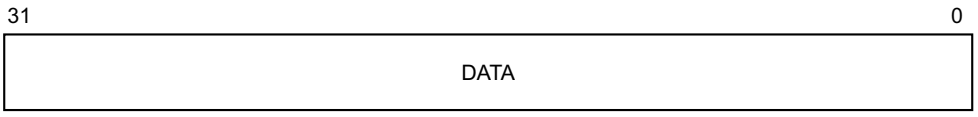
Present only if interrupt configuration registers use the recommended format. Otherwise, this register is RES0.

Attributes

ERRCRICR1 is a 32-bit register.

Field descriptions

The ERRCRICR1 bit assignments are:



DATA, bits [31:0]

Payload for a message signaled interrupt.

The following resets apply:

- On a Error recovery reset, this field resets to an architecturally UNKNOWN value.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing the ERRCRICR1:

ERRCRICR1 can be accessed through its memory-mapped interface:

Component	Offset	Instance
RAS	`0xEA8`	ERRCRICR1

This interface is accessible as follows:

- Accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.7 ERRCRICR2, Critical Error Interrupt Configuration Register 2

The ERRCRICR2 characteristics are:

Purpose

Interrupt configuration register.

Configurations

External register ERRCRICR2[31:0] is architecturally mapped to External register ERRIRQCR<n>[63:32].

This register is present only when ARMv8.4-RAS is implemented. Otherwise, direct accesses to ERRCRICR2 are RES0.

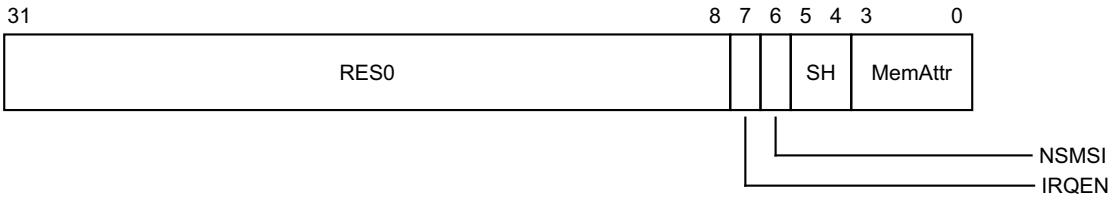
Present only if interrupt configuration registers use the recommended format. Otherwise, this register is RES0.

Attributes

ERRCRICR2 is a 32-bit register.

Field descriptions

The ERRCRICR2 bit assignments are:



Bits [31:8]

Reserved, RES0.

IRQEN, bit [7]

Message Signaled Interrupt enable.

Enables generation of message signaled interrupts.

`0b0` Message signaled interrupts are disabled.

`0b1` Message signaled interrupts are enabled.

If the component does not support disabling message signaled interrupts, this bit is RES0.

The following resets apply:

- On a Error recovery reset, this field resets to `0`.
- On a Cold reset, this field resets to `0`.

NSMSI, bit [6]

Security attribute.

Defines the physical address space for message signaled interrupts.

`0b0` Physical address space for message signaled interrupts is Secure.

`0b1` Physical address space for message signaled interrupts is Non-secure.

If the component prohibits Non-secure writes and does not support configuring the Security attribute, then the Security attribute for message signaled interrupts is IMPLEMENTATION DEFINED.

If the component allows Non-secure writes, then the Security attribute used for message signaled interrupts is Non-secure.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This bit is RES0 if any of the following are true:

- The component allows Non-secure writes.
- The component does not support configuring the Security attribute.

On a Cold reset, this field resets to an IMPLEMENTATION DEFINED value.

SH, bits [5:4]

Shareability.

Defines the Shareability domain for message signaled interrupts.

`0b00` Message signaled interrupts are in the Not shared Shareability domain.

`0b10` Message signaled interrupts are in the Outer Shareable Shareability domain.

`0b11` Message signaled interrupts are in the Inner Shareable Shareability domain.

If the component does not support configuring the Shareability domain, this field is RES0, meaning the Shareability domain for message signaled interrupts is IMPLEMENTATION DEFINED.

The following resets apply:

- On a Error recovery reset, this field resets to an architecturally UNKNOWN value.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

MemAttr, bits [3:0]

Memory type.

Defines the memory type for message signaled interrupts. The values which correspond to each memory type are:

`0b0000` Device-nGnRnE.

`0b0001` Device-nGnRE.

`0b0010` Device-nGRE.

`0b0011` Device-GRE.

`0b0101` Outer Non-cacheable, Inner Non-cacheable.

`0b0110` Outer Non-cacheable, Inner Write-Through Cacheable.

`0b0111` Outer Non-cacheable, Inner Write-Back Cacheable.

`0b1001` Outer Write-Through Cacheable, Inner Non-cacheable.

`0b1010` Outer Write-Through Cacheable, Inner Write-Through Cacheable.

`0b1011` Outer Write-Through Cacheable, Inner Write-Back Cacheable.

`0b1101` Outer Write-Back Cacheable, Inner Non-cacheable.

`0b1110` Outer Write-Back Cacheable, Inner Write-Through Cacheable.

`0b1111` Outer Write-Back Cacheable, Inner Write-Back Cacheable.

If the component does not support configuring the memory type, this field is RES0, meaning the memory type used for message signaled interrupts is IMPLEMENTATION DEFINED.

Note

This is the same format as the VMSAv8-64 stage 2 memory region attributes.

The following resets apply:

- On a Error recovery reset, this field resets to an architecturally UNKNOWN value.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the ERRCRICR2:

ERRCRICR2 can be accessed through its memory-mapped interface:

Component	Offset	Instance
RAS	`0xEAC`	ERRCRICR2

This interface is accessible as follows:

- Accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.8 ERRDEVAFF, Device Affinity Register

The ERRDEVAFF characteristics are:

Purpose

If RAS System Architecture v1.1 is not implemented, a group of error records that is associated with a PE or group or PEs must have affinity with either a single PE or all the PEs at an affinity level.

For a group of error records that has affinity with a single PE or a group of PEs, ERRDEVAFF is a copy of MPIDR_EL1 or part of MPIDR_EL1:

- If the group of error records has affinity with a single PE, the affinity level is 0, ERRDEVAFF reads the same value as MPIDR_EL1, and ERRDEVAFF.F0V reads-as-one to indicate affinity level 0.
- If the group of error records has affinity with a group of PEs, the affinity level is 1, 2, or 3, parts of ERRDEVAFF reads the same value as parts of MPIDR_EL1, and the rest of ERRDEVAFF indicates the level.

For example, if the group of PEs is a subset of the PEs at affinity level 1 then all of the following are true:

- All the PEs in the group have the same values in MPIDR_EL1.{Aff3,Aff2}, and these values are equal to ERRDEVAFF.{Aff3,Aff2}.
- ERRDEVAFF.Aff1 is nonzero and not `0x80`, and ERRDEVAFF.{Aff0,F0V} read-as-zero, to indicate at least affinity level 1. The subset of PEs at level 1 that the group of error records has affinity with is indicated by the least-significant set bit in ERRDEVAFF.Aff1. In this example, if ERRDEVAFF.Aff1[2:0] is `0b100`, then the group of error records has affinity with the up-to 8 PEs that have MPIDR_EL1.Aff1[7:3] == ERRDEVAFF.Aff1[7:3].

Configurations

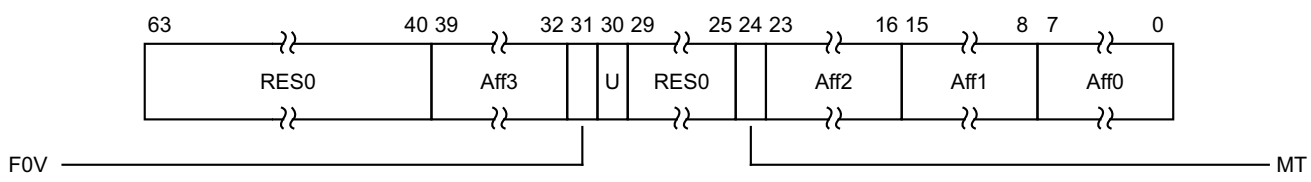
This register is present only when the group of error records has affinity with a PE or cluster of PEs. Otherwise, direct accesses to ERRDEVAFF are RES0.

Attributes

ERRDEVAFF is a 64-bit register.

Field descriptions

The ERRDEVAFF bit assignments are:



Bits [63:40]

Reserved, RES0.

Aff3, bits [39:32]

PE affinity level 3. The MPIDR_EL1.Aff3 field, viewed from the highest Exception level of the associated PE or PEs.

F0V, bit [31]

Indicates that the ERRDEVAFF.Aff0 field is valid.

`0b0` ERRDEVAFF.Aff0 is not valid, and the PE affinity is above level 0 or a subset of level 0.

`0b1` ERRDEVAFF.Aff0 is valid, and the PE affinity is at level 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

U, bit [30]

When ERRDEVAFF.F0V == 0b1:

Uniprocessor. The MPIDR_EL1.U bit, viewed from the highest Exception level of the associated PE.

Otherwise:

Reserved, UNKNOWN.

Bits [29:25]

Reserved, RES0.

MT, bit [24]

When ERRDEVAFF.F0V == 0b1:

Multithreaded. The MPIDR_EL1.MT bit, viewed from the highest Exception level of the associated PE.

Otherwise:

Reserved, UNKNOWN.

Aff2, bits [23:16]

When affine with a PE or PEs at affinity level 2 or below:

PE affinity level 2. The MPIDR_EL1.Aff2 field, viewed from the highest Exception level of the associated PE or PEs.

When affine with a sub-set of PEs at affinity level 2:

PE affinity level 2. Defines part of the MPIDR_EL1.Aff2 field, viewed from the highest Exception level of the associated PEs.

`0bxxxxxxx1` ERRDEVAFF.Aff2[7:1] is the value of MPIDR_EL1.Aff2[7:1], viewed from the highest Exception level of the associated PEs.

`0bxxxxxx10` ERRDEVAFF.Aff2[7:2] is the value of MPIDR_EL1.Aff2[7:2], viewed from the highest Exception level of the associated PEs.

`0bxxxxx100` ERRDEVAFF.Aff2[7:3] is the value of MPIDR_EL1.Aff2[7:3], viewed from the highest Exception level of the associated PEs.

`0bxxxx1000` ERRDEVAFF.Aff2[7:4] is the value of MPIDR_EL1.Aff2[7:4], viewed from the highest Exception level of the associated PEs.

`0bxxx10000` ERRDEVAFF.Aff2[7:5] is the value of MPIDR_EL1.Aff2[7:5], viewed from the highest Exception level of the associated PEs.

`0bxx100000` ERRDEVAFF.Aff2[7:6] is the value of MPIDR_EL1.Aff2[7:6], viewed from the highest Exception level of the associated PEs.

`0bx1000000` ERRDEVAFF.Aff2[7] is the value of MPIDR_EL1.Aff2[7], viewed from the highest Exception level of the associated PEs.

Otherwise:

PE affinity level 2. Indicates whether the PE affinity is at level 3.

`0x80` PE affinity is at level 3.

All other values are reserved.

Aff1, bits [15:8]

When affine with a PE or PEs at affinity level 1 or below:

PE affinity level 1. The MPIDR_EL1.Aff1 field, viewed from the highest Exception level of the associated PE or PEs.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When affine with a sub-set of PEs at affinity level 1:

PE affinity level 1. Defines part of the MPIDR_EL1.Aff1 field, viewed from the highest Exception level of the associated PEs.

`0bxxxxxxx1` ERRDEVAFF.Aff1[7:1] is the value of MPIDR_EL1.Aff1[7:1], viewed from the highest Exception level of the associated PEs.

`0bxxxxxx10` ERRDEVAFF.Aff1[7:2] is the value of MPIDR_EL1.Aff1[7:2], viewed from the highest Exception level of the associated PEs.

`0bxxxxx100` ERRDEVAFF.Aff1[7:3] is the value of MPIDR_EL1.Aff1[7:3], viewed from the highest Exception level of the associated PEs.

`0bxxxx1000` ERRDEVAFF.Aff1[7:4] is the value of MPIDR_EL1.Aff1[7:4], viewed from the highest Exception level of the associated PEs.

`0bxxx10000` ERRDEVAFF.Aff1[7:5] is the value of MPIDR_EL1.Aff1[7:5], viewed from the highest Exception level of the associated PEs.

`0bxx100000` ERRDEVAFF.Aff1[7:6] is the value of MPIDR_EL1.Aff1[7:6], viewed from the highest Exception level of the associated PEs.

`0bx1000000` ERRDEVAFF.Aff1[7] is the value of MPIDR_EL1.Aff1[7], viewed from the highest Exception level of the associated PEs.

Otherwise:

PE affinity level 1. Indicates whether the PE affinity is at level 2.

`0x00` PE affinity is above level 2 or a subset of level 2.

`0x80` PE affinity is at level 2.

Aff0, bits [7:0]

When affine with a PE at affinity level 0:

PE affinity level 0. The MPIDR_EL1.Aff0 field, viewed from the highest Exception level of the associated PE.

When affine with a sub-set of PEs at affinity level 0:

PE affinity level 0. Defines part of the MPIDR_EL1.Aff0 field, viewed from the highest Exception level of the associated PEs.

`0bxxxxxxx1` ERRDEVAFF.Aff0[7:1] is the value of MPIDR_EL1.Aff0[7:1], viewed from the highest Exception level of the associated PEs.

`0bxxxxxx10` ERRDEVAFF.Aff0[7:2] is the value of MPIDR_EL1.Aff0[7:2], viewed from the highest Exception level of the associated PEs.

`0bxxxxx100` ERRDEVAFF.Aff0[7:3] is the value of MPIDR_EL1.Aff0[7:3], viewed from the highest Exception level of the associated PEs.

`0bxxxx1000` ERRDEVAFF.Aff0[7:4] is the value of MPIDR_EL1.Aff0[7:4], viewed from the highest Exception level of the associated PEs.

`0bxxx10000` ERRDEVAFF.Aff0[7:5] is the value of MPIDR_EL1.Aff0[7:5], viewed from the highest Exception level of the associated PEs.

`0bxx100000` ERRDEVAFF.Aff0[7:6] is the value of MPIDR_EL1.Aff0[7:6], viewed from the highest Exception level of the associated PEs.

`0bx1000000` ERRDEVAFF.Aff0[7] is the value of MPIDR_EL1.Aff0[7], viewed from the highest Exception level of the associated PEs.

Otherwise:

PE affinity level 0. Indicates whether the PE affinity is at level 1.

`0x00` PE affinity is above level 1 or a subset of level 1.

`0x80` PE affinity is at level 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the ERRDEVAFF:

ERRDEVAFF can be accessed through its memory-mapped interface:

Component	Offset
RAS	`0xFA8`

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.9 ERRDEVARCH, Device Architecture Register

The ERRDEVARCH characteristics are:

Purpose

Provides discovery information for the component.

Configurations

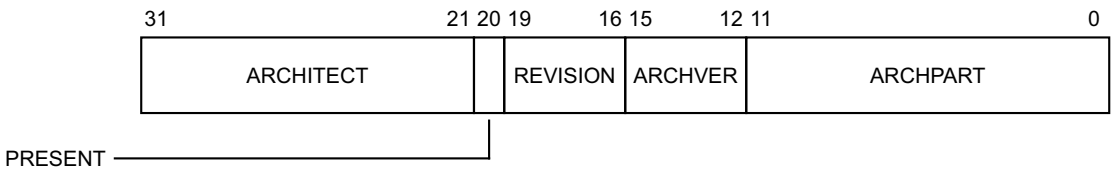
There are no configuration notes.

Attributes

ERRDEVARCH is a 32-bit register.

Field descriptions

The ERRDEVARCH bit assignments are:



ARCHITECT, bits [31:21]

Architect. Defines the architect of the component. Bits [31:28] are the JEP106 continuation code (JEP106 bank ID, minus 1) and bits [27:21] are the JEP106 ID code.

`0b01000111011` JEP106 continuation code `0x4`, ID code `0x3B`. Arm Limited.

Other values are defined by the JEDEC JEP106 standard.

This field reads as `0x23B`.

PRESENT, bit [20]

DEVARCH Present. Defines that the DEVARCH register is present.

`0b0` Device Architecture information not present.

`0b1` Device Architecture information present.

This bit reads as `0b1`.

REVISION, bits [19:16]

Revision. Defines the architecture revision of the component.

`0b0000` RAS System Architecture v1.0.

`0b0001` RAS System Architecture v1.1. As `0b0000` and also:

- Simplifies ERR<n>STATUS.
- Adds support for additional ERR<n>MISC<m> registers.
- Adds support for the optional RAS Timestamp Extension.
- Adds support for the optional RAS Common Fault Injection Model Extension.

All other values are reserved.

ARCHVER, bits [15:12]

Architecture Version. Defines the architecture version of the component.

`0b0000` RAS System Architecture v1.

All other values are reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHVER is ARCHID[15:12].

This field reads as `0b0000`.

ARCHPART, bits [11:0]

Architecture Part. Defines the architecture of the component.

`0xA00` RAS System Architecture.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHPART is ARCHID[11:0].

This field reads as `0xA00`.

Accessing the ERRDEVARCH:

ERRDEVARCH can be accessed through its memory-mapped interface:

Component	Offset
RAS	`0xFBC`

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.10 ERRDEVID, Device Configuration Register

The ERRDEVID characteristics are:

Purpose

Provides discovery information for the component.

Configurations

There are no configuration notes.

Attributes

ERRDEVID is a 32-bit register.

Field descriptions

The ERRDEVID bit assignments are:

31 … 16	15 … 0
RES0	NUM

Bits [31:16]

Reserved, RES0.

NUM, bits [15:0]

IMPLEMENTATION DEFINED.

Highest numbered index of the error records in this group, plus one. Each implemented record is owned by a node. A node might own multiple records.

This manual describes a group of error records accessed via a standard 4KB memory-mapped peripheral. For a 4KB peripheral, up to 24 error records can be accessed if the Common Fault Injection Model is implemented, and up to 56 otherwise.

Accessing the ERRDEVID:

ERRDEVID can be accessed through its memory-mapped interface:

Component	Offset
RAS	`0xFC8`

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.11 ERRERICR0, Error Recovery Interrupt Configuration Register 0

The ERRERICR0 characteristics are:

Purpose

Interrupt configuration register.

Configurations

External register ERRERICR0 is architecturally mapped to External register ERRIRQCR<n>.

This register is present only when RAS is implemented. Otherwise, direct accesses to ERRERICR0 are RES0.

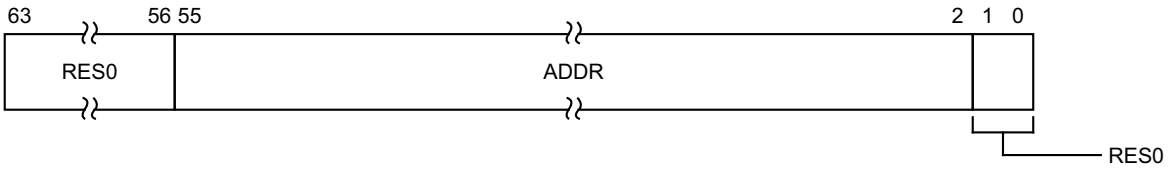
Present only if interrupt configuration registers use the recommended format. Otherwise, this register is RES0.

Attributes

ERRERICR0 is a 64-bit register.

Field descriptions

The ERRERICR0 bit assignments are:



Bits [63:56]

Reserved, RES0.

ADDR, bits [55:2]

Message Signaled Interrupt address.

Specifies the address that the component writes to when signaling an interrupt.

The size of a physical address is IMPLEMENTATION DEFINED. Unimplemented high-order physical address bits are RES0.

The following resets apply:

- On a Error recovery reset, this field resets to an architecturally UNKNOWN value.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [1:0]

Reserved, RES0.

Accessing the ERRERICR0:

ERRERICR0 can be accessed through its memory-mapped interface:

Component	Offset	Instance
RAS	`0xE90`	ERRERICR0

This interface is accessible as follows:

- Accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.12 ERRERICR1, Error Recovery Interrupt Configuration Register 1

The ERRERICR1 characteristics are:

Purpose

Interrupt configuration register.

Configurations

External register ERRERICR1[31:0] is architecturally mapped to External register ERRIRQCR<n>[31:0].

This register is present only when RAS is implemented. Otherwise, direct accesses to ERRERICR1 are RES0.

Present only if interrupt configuration registers use the recommended format. Otherwise, this register is RES0.

Attributes

ERRERICR1 is a 32-bit register.

Field descriptions

The ERRERICR1 bit assignments are:

31 … 0
DATA

DATA, bits [31:0]

Payload for a message signaled interrupt.

The following resets apply:

- On a Error recovery reset, this field resets to an architecturally UNKNOWN value.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing the ERRERICR1:

ERRERICR1 can be accessed through its memory-mapped interface:

Component	Offset	Instance
RAS	`0xE98`	ERRERICR1

This interface is accessible as follows:

- Accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.13 ERRERICR2, Error Recovery Interrupt Configuration Register 2

The ERRERICR2 characteristics are:

Purpose

Interrupt configuration register.

Configurations

External register ERRERICR2[31:0] is architecturally mapped to External register ERRIRQCR<n>[63:32].

This register is present only when RAS is implemented. Otherwise, direct accesses to ERRERICR2 are RES0.

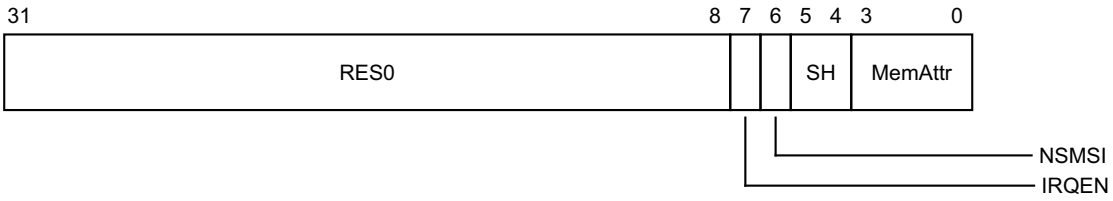
Present only if interrupt configuration registers use the recommended format. Otherwise, this register is RES0.

Attributes

ERRERICR2 is a 32-bit register.

Field descriptions

The ERRERICR2 bit assignments are:



Bits [31:8]

Reserved, RES0.

IRQEN, bit [7]

Message Signaled Interrupt enable.

Enables generation of message signaled interrupts.

`0b0` Message signaled interrupts are disabled.

`0b1` Message signaled interrupts are enabled.

If the component does not support disabling message signaled interrupts, this bit is RES0.

The following resets apply:

- On a Error recovery reset, this field resets to `0`.
- On a Cold reset, this field resets to `0`.

NSMSI, bit [6]

Security attribute.

Defines the physical address space for message signaled interrupts.

`0b0` Physical address space for message signaled interrupts is Secure.

`0b1` Physical address space for message signaled interrupts is Non-secure.

If the component prohibits Non-secure writes and does not support configuring the Security attribute, then the Security attribute for message signaled interrupts is IMPLEMENTATION DEFINED.

If the component allows Non-secure writes, then the Security attribute used for message signaled interrupts is Non-secure.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This bit is RES0 if any of the following are true:

- The component allows Non-secure writes.
- The component does not support configuring the Security attribute.

On a Cold reset, this field resets to an IMPLEMENTATION DEFINED value.

SH, bits [5:4]

Shareability.

Defines the Shareability domain for message signaled interrupts.

`0b00` Message signaled interrupts are in the Not shared Shareability domain.

`0b10` Message signaled interrupts are in the Outer Shareable Shareability domain.

`0b11` Message signaled interrupts are in the Inner Shareable Shareability domain.

If the component does not support configuring the Shareability domain, this field is RES0, meaning the Shareability domain for message signaled interrupts is IMPLEMENTATION DEFINED.

The following resets apply:

- On a Error recovery reset, this field resets to an architecturally UNKNOWN value.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

MemAttr, bits [3:0]

Memory type.

Defines the memory type for message signaled interrupts. The values which correspond to each memory type are:

`0b0000` Device-nGnRnE.

`0b0001` Device-nGnRE.

`0b0010` Device-nGRE.

`0b0011` Device-GRE.

`0b0101` Outer Non-cacheable, Inner Non-cacheable.

`0b0110` Outer Non-cacheable, Inner Write-Through Cacheable.

`0b0111` Outer Non-cacheable, Inner Write-Back Cacheable.

`0b1001` Outer Write-Through Cacheable, Inner Non-cacheable.

`0b1010` Outer Write-Through Cacheable, Inner Write-Through Cacheable.

`0b1011` Outer Write-Through Cacheable, Inner Write-Back Cacheable.

`0b1101` Outer Write-Back Cacheable, Inner Non-cacheable.

`0b1110` Outer Write-Back Cacheable, Inner Write-Through Cacheable.

`0b1111` Outer Write-Back Cacheable, Inner Write-Back Cacheable.

If the component does not support configuring the memory type, this field is RES0, meaning the memory type used for message signaled interrupts is IMPLEMENTATION DEFINED.

Note

This is the same format as the VMSAv8-64 stage 2 memory region attributes.

The following resets apply:

- On a Error recovery reset, this field resets to an architecturally UNKNOWN value.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the ERRERICR2:

ERRERICR2 can be accessed through its memory-mapped interface:

Component	Offset	Instance
RAS	`0xE9C`	ERRERICR2

This interface is accessible as follows:

- Accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.14 ERRFHICR0, Fault-Handling Interrupt Configuration Register 0

The ERRFHICR0 characteristics are:

Purpose

Interrupt configuration register.

Configurations

External register ERRFHICR0 is architecturally mapped to External register ERRIRQCR<n>.

This register is present only when RAS is implemented. Otherwise, direct accesses to ERRFHICR0 are RES0.

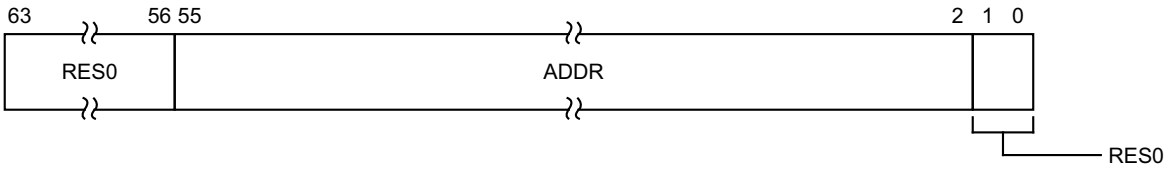
Present only if interrupt configuration registers use the recommended format. Otherwise, this register is RES0.

Attributes

ERRFHICR0 is a 64-bit register.

Field descriptions

The ERRFHICR0 bit assignments are:



Bits [63:56]

Reserved, RES0.

ADDR, bits [55:2]

Message Signaled Interrupt address.

Specifies the address that the component writes to when signaling an interrupt.

The size of a physical address is IMPLEMENTATION DEFINED. Unimplemented high-order physical address bits are RES0.

The following resets apply:

- On a Error recovery reset, this field resets to an architecturally UNKNOWN value.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [1:0]

Reserved, RES0.

Accessing the ERRFHICR0:

ERRFHICR0 can be accessed through its memory-mapped interface:

Component	Offset	Instance
RAS	`0xE80`	ERRFHICR0

This interface is accessible as follows:

- Accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.15 ERRFHICR1, Fault-Handling Interrupt Configuration Register 1

The ERRFHICR1 characteristics are:

Purpose

Interrupt configuration register.

Configurations

External register ERRFHICR1[31:0] is architecturally mapped to External register ERRIRQCR<n>[31:0].

This register is present only when RAS is implemented. Otherwise, direct accesses to ERRFHICR1 are RES0.

Present only if interrupt configuration registers use the recommended format. Otherwise, this register is RES0.

Attributes

ERRFHICR1 is a 32-bit register.

Field descriptions

The ERRFHICR1 bit assignments are:

31 … 0
DATA

DATA, bits [31:0]

Payload for a message signaled interrupt.

The following resets apply:

- On a Error recovery reset, this field resets to an architecturally UNKNOWN value.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing the ERRFHICR1:

ERRFHICR1 can be accessed through its memory-mapped interface:

Component	Offset	Instance
RAS	`0xE88`	ERRFHICR1

This interface is accessible as follows:

- Accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.16 ERRFHICR2, Fault-Handling Interrupt Configuration Register 2

The ERRFHICR2 characteristics are:

Purpose

Interrupt configuration register.

Configurations

External register ERRFHICR2[31:0] is architecturally mapped to External register ERRIRQCR<n>[63:32].

This register is present only when RAS is implemented. Otherwise, direct accesses to ERRFHICR2 are RES0.

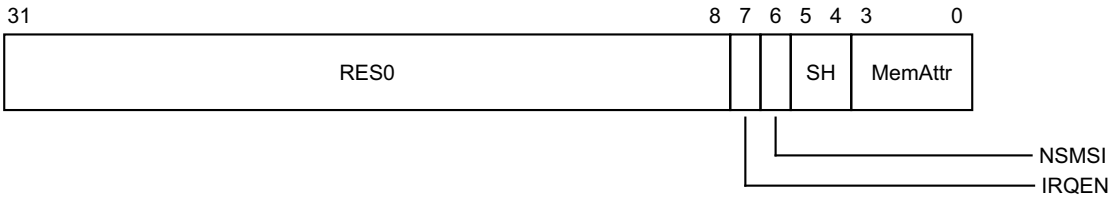
Present only if interrupt configuration registers use the recommended format. Otherwise, this register is RES0.

Attributes

ERRFHICR2 is a 32-bit register.

Field descriptions

The ERRFHICR2 bit assignments are:



Bits [31:8]

Reserved, RES0.

IRQEN, bit [7]

Message Signaled Interrupt enable.

Enables generation of message signaled interrupts.

`0b0` Message signaled interrupts are disabled.

`0b1` Message signaled interrupts are enabled.

If the component does not support disabling message signaled interrupts, this bit is RES0.

The following resets apply:

- On a Error recovery reset, this field resets to `0`.
- On a Cold reset, this field resets to `0`.

NSMSI, bit [6]

Security attribute.

Defines the physical address space for message signaled interrupts.

`0b0` Physical address space for message signaled interrupts is Secure.

`0b1` Physical address space for message signaled interrupts is Non-secure.

If the component prohibits Non-secure writes and does not support configuring the Security attribute, then the Security attribute for message signaled interrupts is IMPLEMENTATION DEFINED.

If the component allows Non-secure writes, then the Security attribute used for message signaled interrupts is Non-secure.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This bit is RES0 if any of the following are true:

- The component allows Non-secure writes.
- The component does not support configuring the Security attribute.

On a Cold reset, this field resets to an IMPLEMENTATION DEFINED value.

SH, bits [5:4]

Shareability.

Defines the Shareability domain for message signaled interrupts.

`0b00` Message signaled interrupts are in the Not shared Shareability domain.

`0b10` Message signaled interrupts are in the Outer Shareable Shareability domain.

`0b11` Message signaled interrupts are in the Inner Shareable Shareability domain.

If the component does not support configuring the Shareability domain, this field is RES0, meaning the Shareability domain for message signaled interrupts is IMPLEMENTATION DEFINED.

The following resets apply:

- On a Error recovery reset, this field resets to an architecturally UNKNOWN value.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

MemAttr, bits [3:0]

Memory type.

Defines the memory type for message signaled interrupts. The values which correspond to each memory type are:

`0b0000` Device-nGnRnE.

`0b0001` Device-nGnRE.

`0b0010` Device-nGRE.

`0b0011` Device-GRE.

`0b0101` Outer Non-cacheable, Inner Non-cacheable.

`0b0110` Outer Non-cacheable, Inner Write-Through Cacheable.

`0b0111` Outer Non-cacheable, Inner Write-Back Cacheable.

`0b1001` Outer Write-Through Cacheable, Inner Non-cacheable.

`0b1010` Outer Write-Through Cacheable, Inner Write-Through Cacheable.

`0b1011` Outer Write-Through Cacheable, Inner Write-Back Cacheable.

`0b1101` Outer Write-Back Cacheable, Inner Non-cacheable.

`0b1110` Outer Write-Back Cacheable, Inner Write-Through Cacheable.

`0b1111` Outer Write-Back Cacheable, Inner Write-Back Cacheable.

If the component does not support configuring the memory type, this field is RES0, meaning the memory type used for message signaled interrupts is IMPLEMENTATION DEFINED.

Note

This is the same format as the VMSAv8-64 stage 2 memory region attributes.

The following resets apply:

- On a Error recovery reset, this field resets to an architecturally UNKNOWN value.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the ERRFHICR2:

ERRFHICR2 can be accessed through its memory-mapped interface:

Component	Offset	Instance
RAS	`0xE8C`	ERRFHICR2

This interface is accessible as follows:

- Accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.17 ERRGSR, Error Group Status Register

The ERRGSR characteristics are:

Purpose

Shows the status for the records in the group.

Configurations

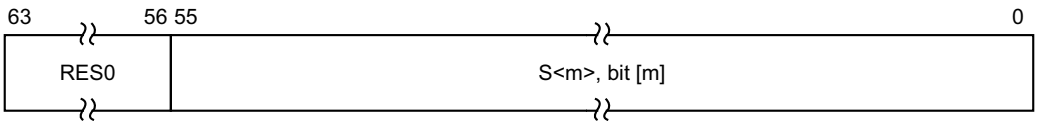
This manual describes a group of error records accessed via a standard 4KB memory-mapped peripheral. For a 4KB peripheral, up to 24 error records can be accessed if the Common Fault Injection Model is implemented, and up to 56 otherwise.

Attributes

ERRGSR is a 64-bit register.

Field descriptions

The ERRGSR bit assignments are:



Bits [63:56]

Reserved, RES0.

S<m>, bit [m], for m = 0 to 55

The status for Error Record <m>. A read-only copy of ERR<n>STATUS.V.

`0b0` No error.

`0b1` One or more errors.

If the Common Fault Injection Model is implemented, up-to 24 records can be implemented meaning bits [55:24] are RES0.

This bit is RES0 if any of the following are true:

- The record is not implemented.
- The record does not support this type of reporting.

Accessing the ERRGSR:

ERRGSR can be accessed through its memory-mapped interface:

Component	Offset
RAS	`0xE00`

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.18 ERRIIDR, Implementation Identification Register

The ERRIIDR characteristics are:

Purpose

Defines the implementer of the component.

Configurations

Implementation of this register is OPTIONAL.

This register is present only when ARMv8.4-RAS is implemented. Otherwise, direct accesses to ERRIIDR are RES0.

Attributes

ERRIIDR is a 32-bit register.

Field descriptions

The ERRIIDR bit assignments are:

31 – 20	19 – 16	15 – 12	11 – 0
ProductID	Variant	Revision	Implementer

ProductID, bits [31:20]

IMPLEMENTATION DEFINED.

Part number, bits [11:0]. The part number is selected by the designer of the component.

If ERRPIDR0 and ERRPIDR1 are implemented, ERRPIDR0.PART_0 matches bits [7:0] of ERRIIDR.ProductID and ERRPIDR1.PART_1 matches bits [11:8] of ERRIIDR.ProductID.

Variant, bits [19:16]

IMPLEMENTATION DEFINED.

Component major revision.

This field distinguishes product variants or major revisions of the product.

If ERRPIDR2 is implemented, ERRPIDR2.REVISION matches ERRIIDR.Variant.

Revision, bits [15:12]

IMPLEMENTATION DEFINED.

Component minor revision.

This field distinguishes minor revisions of the product.

If ERRPIDR3 is implemented, ERRPIDR3.REVAND matches ERRIIDR.Revision.

Implementer, bits [11:0]

IMPLEMENTATION DEFINED.

Contains the JEP106 code of the company that implemented the RAS component. For an Arm implementation, this field has the value `0x43B`.

Bits [11:8] contain the JEP106 continuation code of the implementer, and bits [6:0] contain the JEP106 identity code of the implementer. Bit 7 is RES0.

If ERRPIDR4 is implemented, ERRPIDR2 is implemented, and ERRPIDR1 is implemented, ERRPIDR4.DES_2 matches bits [11:8] of ERRIIDR.Implementer, ERRPIDR2.DES_1 matches bits [6:4] of ERRIIDR.Implementer, and ERRPIDR1.DES_0 matches bits [3:0] of ERRIIDR.Implementer.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the ERRIIDR:

ERRIIDR can be accessed through its memory-mapped interface:

Component	Offset
RAS	`0xE10`

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.19 ERRIRQCR<n>, Generic Error Interrupt Configuration Register, n = 0 - 15

The ERRIRQCR<n> characteristics are:

Purpose

The ERRIRQCR<n> registers are IMPLEMENTATION DEFINED interrupt configuration registers.

The architecture provides a recommended format for the ERRIRQCR<n> registers. The registers provided by the recommended layout are:

- ERRFHICR0, ERRFHICR1, and ERRFHICR2, the fault-handling interrupt configuration registers. ERRFHICR<m> maps to ERRIRQCR0 and ERRIRQCR1.
- ERRERICR0, ERRERICR1, and ERRERICR2, the error recovery interrupt configuration registers. ERRERICR<m> maps to ERRIRQCR2 and ERRIRQCR3.
- If ARMv8.4-RAS is implemented, ERRCRICR0, ERRCRICR1, and ERRCRICR2, the critical error interrupt configuration registers. ERRFHICR<m> maps to ERRIRQCR4 and ERRIRQCR5.
- ERRIRQSR, the error interrupt status register. ERRIRQSR maps to ERRIRQCR15.

This register describes the generic IMPLEMENTATION DEFINED format of the interrupt configuration registers, when the recommended layout is not used.

For details of this, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*.

Configurations

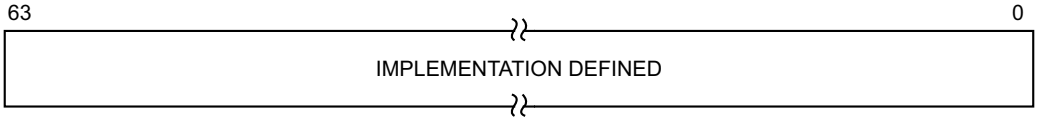
This register is present only when the interrupt configuration registers are implemented. Otherwise, direct accesses to ERRIRQCR<n> are RES0.

Attributes

ERRIRQCR<n> is a 64-bit register.

Field descriptions

The ERRIRQCR<n> bit assignments are:



IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED controls. The content of these registers is IMPLEMENTATION DEFINED.

Accessing the ERRIRQCR<n>:

ERRIRQCR<n> can be accessed through its memory-mapped interface:

Component	Offset	Instance
RAS	`0xE80` + 8n	ERRIRQCR<n>

This interface is accessible as follows:

- Accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.20 ERRIRQSR, Error Interrupt Status Register

The ERRIRQSR characteristics are:

Purpose

Interrupt status register.

Configurations

External register ERRIRQSR is architecturally mapped to External register ERRIRQCR<n>.

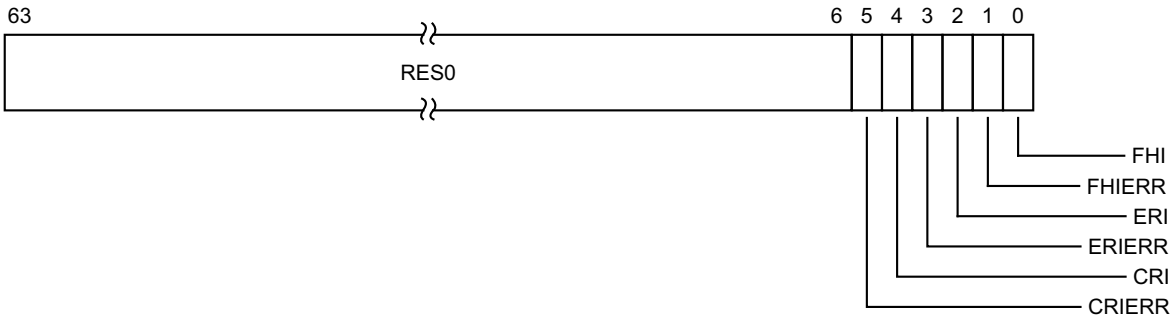
This register is present only when interrupt configuration registers use the recommended format. Otherwise, direct accesses to ERRIRQSR are RES0.

Attributes

ERRIRQSR is a 64-bit register.

Field descriptions

The ERRIRQSR bit assignments are:



Bits [63:6]

Reserved, RES0.

CRIERR, bit [5]

When Critical Error Interrupt is implemented:

Critical Error Interrupt error.

`0b0` Critical Error Interrupt write has not returned an error since this bit was last cleared to zero.

`0b1` Critical Error Interrupt write has returned an error since this bit was last cleared to zero.

This bit is read/write-one-to-clear.

The following resets apply:

- On a Error recovery reset, this field resets to an architecturally UNKNOWN value.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

CRI, bit [4]

When Critical Error Interrupt is implemented:

Critical Error Interrupt write in progress.

`0b0` Critical Error Interrupt write not in progress.

`0b1` Critical Error Interrupt write in progress.

Software must not disable an interrupt whilst the write is in progress.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

This bit does not indicate whether an interrupt is active, but rather whether a write triggered by the interrupt is in progress.

To determine whether an interrupt is active, software must examine the individual ERR<n>STATUS registers.

Access to this field is RO.

Otherwise:

Reserved, RES0.

ERIERR, bit [3]

When Error Recovery Interrupt is implemented:

Error Recovery Interrupt error.

`0b0` Error Recovery Interrupt write has not returned an error since this bit was last cleared to zero.

`0b1` Error Recovery Interrupt write has returned an error since this bit was last cleared to zero.

This bit is read/write-one-to-clear.

The following resets apply:

- On a Error recovery reset, this field resets to an architecturally UNKNOWN value.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ERI, bit [2]

When Error Recovery Interrupt is implemented:

Error Recovery Interrupt write in progress.

`0b0` Error Recovery Interrupt write not in progress.

`0b1` Error Recovery Interrupt write in progress.

Software must not disable an interrupt whilst the write is in progress.

Note

This bit does not indicate whether an interrupt is active, but rather whether a write triggered by the interrupt is in progress.

To determine whether an interrupt is active, software must examine the individual ERR<n>STATUS registers.

Access to this field is RO.

Otherwise:

Reserved, RES0.

FHIERR, bit [1]

When Fault Handling Interrupt is implemented:

Fault Handling Interrupt error.

`0b0` Fault Handling Interrupt write has not returned an error since this bit was last cleared to zero.

`0b1` Fault Handling Interrupt write has returned an error since this bit was last cleared to zero.

This bit is read/write-one-to-clear.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following resets apply:

- On a Error recovery reset, this field resets to an architecturally UNKNOWN value.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FHI, bit [0]

When Fault Handling Interrupt is implemented:

Fault Handling Interrupt write in progress.

`0b0` Fault Handling Interrupt write not in progress.

`0b1` Fault Handling Interrupt write in progress.

Software must not disable an interrupt whilst the write is in progress.

Note

This bit does not indicate whether an interrupt is active, but rather whether a write triggered by the interrupt is in progress.

To determine whether an interrupt is active, software must examine the individual ERR<n>STATUS registers.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Accessing the ERRIRQSR:

ERRIRQSR can be accessed through its memory-mapped interface:

Component	Offset
RAS	`0xEF8`

This interface is accessible as follows:

- Accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.21 ERR<n>ADDR, Error Record Address Register, n = 0 - 65534

The ERR<n>ADDR characteristics are:

Purpose

If an address is associated with a detected error, then this must be written to the address register when the error is recorded. It is IMPLEMENTATION DEFINED how the recorded addresses map to the software-visible physical addresses. Software might have to reconstruct the actual physical addresses using the identity of the node and knowledge of the system.

Configurations

This register is present only when error record <n> is implemented and the error record includes an address associated with an error. Otherwise, direct accesses to ERR<n>ADDR are RES0.

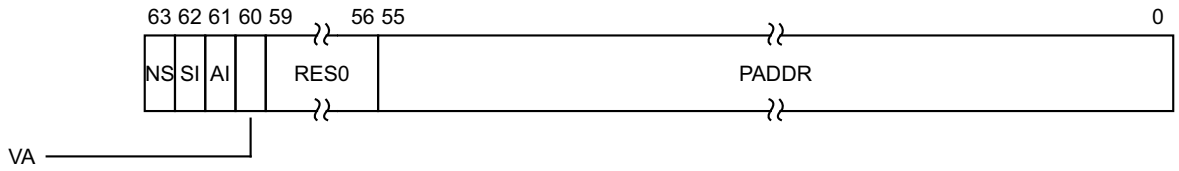
ERR<n>FR describes the features implemented by the node that owns error record <n>. <q> is the index of the first error record owned by the same node as error record <n>. If the node owns a single record, then q = n.

Attributes

ERR<n>ADDR is a 64-bit register.

Field descriptions

The ERR<n>ADDR bit assignments are:



NS, bit [63]

Non-secure attribute.

`0b0` The address is Secure.

`0b1` The address is Non-secure.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

SI, bit [62]

Secure Incorrect. Indicates whether the NS bit is valid.

`0b0` The NS bit is correct. That is, it matches the programmers' view of the Non-secure attribute for this recorded location.

`0b1` The NS bit might not be correct, and might not match the programmers' view of the Non-secure attribute for the recorded location.

It is IMPLEMENTATION DEFINED whether this bit is read-only or read/write.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

AI, bit [61]

Address Incorrect. Indicates whether the PADDR field is a valid physical address that is known to match the programmers' view of the physical address for the recorded location.

`0b0` The PADDR field is a valid physical address. That is, it matches the programmers' view of the physical address for the recorded location.

`0b1` The PADDR field might not be a valid physical address, and might not match the programmers' view of the physical address for the recorded location.

It is IMPLEMENTATION DEFINED whether this bit is read-only or read/write.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

VA, bit [60]

Virtual Address. Indicates whether the PADDR field is a virtual address.

`0b0` The PADDR field is not a virtual address.

`0b1` The PADDR field is a virtual address.

No context information is provided for the virtual address. When ERR<n>ADDR.VA == `0b1`, ERR<n>ADDR.{NS,SI,AI} must read as {0,1,1}.

Support for this bit is optional. If this bit is not implemented and the PADDR field is a virtual address, then ERR<n>ADDR.{NS,SI,AI} must read as {0,1,1}.

It is IMPLEMENTATION DEFINED whether this bit is read-only or read/write.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [59:56]

Reserved, RES0.

PADDR, bits [55:0]

Physical Address. Address of the recorded location. If the physical address size implemented by this component is smaller than the size of this field, then high-order bits are unimplemented and either RES0 or have a fixed read-only IMPLEMENTATION DEFINED value. Low-order address bits might also be unimplemented and RES0, for example, if the physical address is always aligned to the size of a protection granule.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing the ERR<n>ADDR:

ERR<n>ADDR ignores writes if ERR<n>STATUS.AV == `0b1`.

ERR<n>ADDR can be accessed through its memory-mapped interface:

Component	Offset	Instance
RAS	`0x018` + 64n	ERR<n>ADDR

This interface is accessible as follows:

- Accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.22 ERR<n>CTLR, Error Record Control Register, n = 0 - 65534

The ERR<n>CTLR characteristics are:

Purpose

The error control register contains enable bits for the node that writes to this record:

- Enabling error detection and correction.
- Enabling the critical error, error recovery, and fault handling interrupts.
- Enabling in-band error response for Uncorrected errors.

For each bit, if the selected node does not support the feature, then the bit is RES0. The definition of each record is IMPLEMENTATION DEFINED.

Configurations

This register is present only when error record <n> is implemented and error record <n> is the first error record owned by a node. Otherwise, direct accesses to ERR<n>CTLR are RES0.

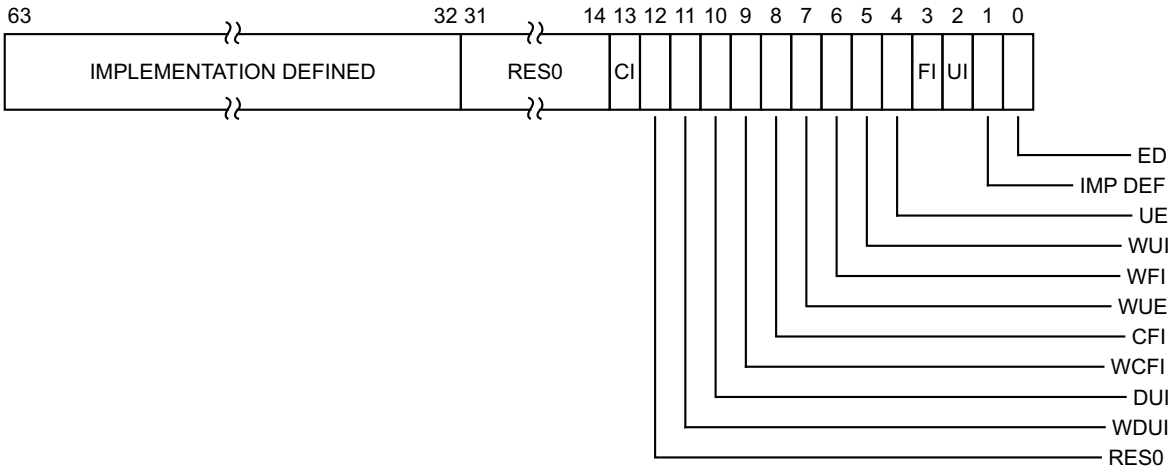
ERR<n>FR describes the features implemented by the node.

Attributes

ERR<n>CTLR is a 64-bit register.

Field descriptions

The ERR<n>CTLR bit assignments are:



IMPLEMENTATION DEFINED, bits [63:32]

IMPLEMENTATION DEFINED.

Reserved for IMPLEMENTATION DEFINED controls. Must permit SBZP write policy for software.

Bits [31:14]

Reserved, RES0.

CI, bit [13]

When ERR<n>FR.CI == 0b10:

Critical error interrupt enable. When enabled, the critical error interrupt is generated for a critical error condition.

`0b0` Critical error interrupt not generated for critical errors. Critical errors are treated as Uncontained errors.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` Critical error interrupt generated for critical errors.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [12]

Reserved, RES0.

WDUI, bit [11]

When ERR<n>FR.DUI == 0b11:

Error recovery interrupt for deferred errors on writes enable.

When enabled, the error recovery interrupt is generated for detected Deferred errors on writes.

`0b0` Error recovery interrupt not generated for deferred errors on writes.

`0b1` Error recovery interrupt generated for deferred errors on writes.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DUI, bit [10]

When ERR<n>FR.DUI == 0b10:

Error recovery interrupt for deferred errors enable.

When ERR<n>FR.DUI == `0b10`, this control applies to errors arising from both reads and writes.

When enabled, the error recovery interrupt is generated for all detected Deferred errors.

`0b0` Error recovery interrupt not generated for deferred errors.

`0b1` Error recovery interrupt generated for deferred errors.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

When ERR<n>FR.DUI == 0b11:

Error recovery interrupt for deferred errors on reads enable.

When ERR<n>FR.DUI == `0b11`, this bit is named RDUI.

When enabled, the error recovery interrupt is generated for detected Deferred errors on reads.

`0b0` Error recovery interrupt not generated for deferred errors on reads.

`0b1` Error recovery interrupt generated for deferred errors on reads.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise:

Reserved, RES0.

WCFI, bit [9]

When ERR<n>FR.CFI == 0b11:

Fault handling interrupt for Corrected errors on writes enable.

When enabled:

- If the node implements Corrected error counters for writes, then the fault handling interrupt is generated when a counter overflows and the overflow bit for the counter is set to `0b1`. For more information, see ERR<n>MISC0.
- Otherwise, the fault handling interrupt is also generated for detected Corrected errors on writes.

`0b0` Fault handling interrupt not generated for Corrected errors on writes.

`0b1` Fault handling interrupt generated for Corrected errors on writes.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

CFI, bit [8]

When ERR<n>FR.CFI == 0b10:

Fault handling interrupt for Corrected errors enable.

When ERR<n>FR.CFI == `0b10`, this control applies to errors arising from both reads and writes.

When enabled:

- If the node implements Corrected error counters, then the fault handling interrupt is generated when a counter overflows and the overflow bit for the counter is set to `0b1`. For more information, see ERR<n>MISC0.
- Otherwise, the fault handling interrupt is also generated for all detected Corrected errors.

`0b0` Fault handling interrupt not generated for Corrected errors.

`0b1` Fault handling interrupt generated for Corrected errors.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

When ERR<n>FR.CFI == 0b11:

Fault handling interrupt for Corrected errors on reads enable.

When ERR<n>FR.CFI == `0b11`, this bit is named RCFI.

When enabled:

- If the node implements Corrected error counters for reads, then the fault handling interrupt is generated when a counter overflows and the overflow bit for the counter is set to `0b1`. For more information, see ERR<n>MISC0.
- Otherwise, the fault handling interrupt is also generated for detected Corrected errors on reads.

`0b0` Fault handling interrupt not generated for Corrected errors on reads.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` Fault handling interrupt generated for Corrected errors on reads.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

WUE, bit [7]

When ERR<n>FR.UE == 0b11:

In-band Uncorrected error reporting on writes enable.

When enabled, responses to writes that detect an Uncorrected error that cannot be deferred are signaled in-band as a detected Uncorrected error (External Abort).

`0b0` External Abort response for Uncorrected errors on writes disabled.

`0b1` External Abort response for Uncorrected errors on writes enabled.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

WFI, bit [6]

When ERR<n>FR.FI == 0b11:

Fault handling interrupt on writes enable.

When enabled:

- The fault handling interrupt is generated for detected Deferred errors and Uncorrected errors.
- If the corresponding fault handling interrupt for Corrected errors control is not implemented:
  - If the node implements Corrected error counters for writes, then the fault handling interrupt is also generated when a counter overflows and the overflow bit for the counter is set to `0b1`.
  - Otherwise, the fault handling interrupt is also generated for detected Corrected errors on writes.

`0b0` Fault handling interrupt on writes disabled.

`0b1` Fault handling interrupt on writes enabled.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

WUI, bit [5]

When ERR<n>FR.UI == 0b11:

Uncorrected error recovery interrupt on writes enable.

When enabled, the error recovery interrupt is generated for detected Uncorrected errors on writes that are not deferred.

`0b0` Error recovery interrupt on writes disabled.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` Error recovery interrupt on writes enabled.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

UE, bit [4]

When ERR<n>FR.UE == 0b10:

In-band Uncorrected error reporting enable.

When ERR<n>FR.UE == `0b10`, this control applies to errors arising from both reads and writes.

When enabled, responses to transactions that detect an Uncorrected error that cannot be deferred are signaled in-band as a detected Uncorrected error (External Abort).

`0b0` External Abort response for Uncorrected errors disabled.

`0b1` External Abort response for Uncorrected errors enabled.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

When ERR<n>FR.UE == 0b11:

In-band Uncorrected error reporting on reads enable.

When ERR<n>FR.UE == `0b11`, this bit is named RUE.

When enabled, responses to reads that detect an Uncorrected error that cannot be deferred are signaled in-band as a detected Uncorrected error (External Abort).

`0b0` External Abort response for Uncorrected errors on reads disabled.

`0b1` External Abort response for Uncorrected errors on reads enabled.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FI, bit [3]

When ERR<n>FR.FI == 0b10:

Fault handling interrupt enable.

When ERR<n>FR.FI == `0b10`, this control applies to errors arising from both reads and writes.

When enabled:

- The fault handling interrupt is generated for all detected Deferred errors and Uncorrected errors.
- If the fault handling interrupt for Corrected errors control is not implemented:
  - — If the node implements Corrected error counters, then the fault handling interrupt is also generated when a counter overflows and the overflow bit for the counter is set to `0b1`.
  - — Otherwise, the fault handling interrupt is also generated for all detected Corrected errors.

`0b0` Fault handling interrupt disabled.

`0b1` Fault handling interrupt enabled.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

When ERR<n>FR.FI == 0b11:

Fault handling interrupt on reads enable.

When ERR<n>FR.FI == `0b11`, this bit is named RFI.

When enabled:

- The fault handling interrupt is generated for detected Deferred errors and Uncorrected errors.
- If the corresponding fault handling interrupt for Corrected errors control is not implemented:
  - — If the node implements Corrected error counters for reads, then the fault handling interrupt is also generated when a counter overflows and the overflow bit for the counter is set to `0b1`.
  - — Otherwise, the fault handling interrupt is also generated for detected Corrected errors on reads.

`0b0` Fault handling interrupt on reads disabled.

`0b1` Fault handling interrupt on reads enabled.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

UI, bit [2]

When ERR<n>FR.UI == 0b10:

Uncorrected error recovery interrupt enable.

When ERR<n>FR.UI == `0b10`, this control applies to errors arising from both reads and writes.

When enabled, the error recovery interrupt is generated for all detected Uncorrected errors that are not deferred.

`0b0` Error recovery interrupt disabled.

`0b1` Error recovery interrupt enabled.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

When ERR<n>FR.UI == 0b11:

Uncorrected error recovery interrupt on reads enable.

When ERR<n>FR.UI == `0b11`, this bit is named RUI.

When enabled, the error recovery interrupt is generated for detected Uncorrected errors on reads that are not deferred.

`0b0` Error recovery interrupt on reads disabled.

`0b1` Error recovery interrupt on reads enabled.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IMPLEMENTATION DEFINED, bit [1]

IMPLEMENTATION DEFINED.

Reserved for IMPLEMENTATION DEFINED controls. Must permit SBZP write policy for software.

ED, bit [0]

When ERR<n>FR.ED == 0b10:

Error reporting and logging enable. When disabled, the node behaves as if error detection and correction are disabled, and no errors are recorded or signaled by the node. Arm recommends that, when disabled, correct error detection and correction codes are written for writes, unless disabled by an IMPLEMENTATION DEFINED control for error injection.

`0b0` Error reporting disabled.

`0b1` Error reporting enabled.

It is IMPLEMENTATION DEFINED whether the node fully disables error detection and correction when reporting is disabled. That is, even with error reporting disabled, the node might continue to silently correct errors. Uncorrectable errors might result in corrupt data being silently propagated by the node.

Note

If this node requires initialization after Cold reset to prevent signaling false errors, then Arm recommends this bit is set to `0b0` on Cold reset, meaning errors are not reported from Cold reset. This allows boot software to initialize a node without signaling errors. Software can enable error reporting after the node is initialized. Otherwise, the Cold reset value is IMPLEMENTATION DEFINED. If the Cold reset value is `0b1`, the reset values of other controls in this register are also IMPLEMENTATION DEFINED and should not be UNKNOWN.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an IMPLEMENTATION DEFINED value.

Otherwise:

Reserved, RES0.

Accessing the ERR<n>CTLR:

ERR<n>CTLR can be accessed through its memory-mapped interface:

Component	Offset	Instance
RAS	`0x008` + 64n	ERR<n>CTLR

This interface is accessible as follows:

- Accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.23 ERR<n>FR, Error Record Feature Register, n = 0 - 65534

The ERR<n>FR characteristics are:

Purpose

Defines whether <n> is the first record owned by a node:

- If <n> is the first error record owned by a node, then ERR<n>FR.ED != `0b00`.
- If <n> is not the first error record owned by a node, then ERR<n>FR.ED == `0b00`.

If <n> is the first record owned by the node, defines which of the common architecturally-defined features are implemented by the node and, of the implemented features, which are software programmable.

Configurations

This register is present only when error record <n> is implemented. Otherwise, direct accesses to ERR<n>FR are RES0.

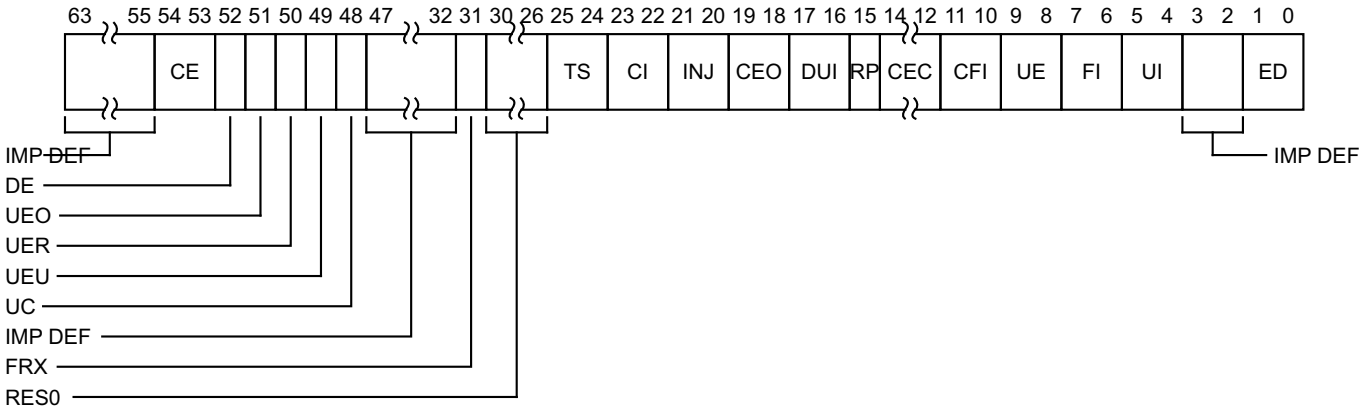
Attributes

ERR<n>FR is a 64-bit register.

Field descriptions

The ERR<n>FR bit assignments are:

When ERR<n>FR.ED != 0b00:



IMPLEMENTATION DEFINED, bits [63:55]

When ERR<n>FR.FRX == 0b0:

IMPLEMENTATION DEFINED.

Reserved for identifying IMPLEMENTATION DEFINED controls.

Otherwise:

Reserved, RES0.

CE, bits [54:53]

When ERR<n>FR.FRX == 0b1:

Corrected Error recording. Describes the types of Corrected Error the node can record.

`0b00` The node does not record any type of Corrected Error.

`0b01` The node can record transient or persistent Corrected Errors (Corrected Errors that are recorded as ERR<n>STATUS.CE == `0b01` and `0b11`).

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b10` The node can record of a non-specific Corrected Error (a Corrected Error that is recorded as ERR<n>STATUS.CE == `0b10`).

`0b11` The node can record any type of Corrected Error.

When ERR<n>FR.FRX == 0b0:

IMPLEMENTATION DEFINED.

Reserved for identifying IMPLEMENTATION DEFINED controls.

Otherwise:

Reserved, RES0.

DE, bit [52]

When ERR<n>FR.FRX == 0b1:

Deferred Error recording. Describes whether the node can record this type of error.

`0b0` The node does not record this type of error.

`0b1` The node can record this type of error.

When ERR<n>FR.FRX == 0b0:

IMPLEMENTATION DEFINED.

Reserved for identifying IMPLEMENTATION DEFINED controls.

Otherwise:

Reserved, RES0.

UEO, bit [51]

When ERR<n>FR.FRX == 0b1:

Latent or Restartable Error recording. Describes whether the node can record this type of error.

`0b0` The node does not record this type of error.

`0b1` The node can record this type of error.

When ERR<n>FR.FRX == 0b0:

IMPLEMENTATION DEFINED.

Reserved for identifying IMPLEMENTATION DEFINED controls.

Otherwise:

Reserved, RES0.

UER, bit [50]

When ERR<n>FR.FRX == 0b1:

Signaled or Recoverable Error recording. Describes whether the node can record this type of error.

`0b0` The node does not record this type of error.

`0b1` The node can record this type of error.

When ERR<n>FR.FRX == 0b0:

IMPLEMENTATION DEFINED.

Reserved for identifying IMPLEMENTATION DEFINED controls.

Otherwise:

Reserved, RES0.

UEU, bit [49]

When ERR<n>FR.FRX == 0b1:

Unrecoverable Error recording. Describes whether the node can record this type of error.

`0b0` The node does not record this type of error.

`0b1` The node can record this type of error.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When ERR<n>FR.FRX == 0b0:

IMPLEMENTATION DEFINED.

Reserved for identifying IMPLEMENTATION DEFINED controls.

Otherwise:

Reserved, RES0.

UC, bit [48]

When ERR<n>FR.FRX == 0b1:

Uncontainable Error recording. Describes whether the node can record this type of error.

`0b0` The node does not record this type of error.

`0b1` The node can record this type of error.

When ERR<n>FR.FRX == 0b0:

IMPLEMENTATION DEFINED.

Reserved for identifying IMPLEMENTATION DEFINED controls.

Otherwise:

Reserved, RES0.

IMPLEMENTATION DEFINED, bits [47:32]

IMPLEMENTATION DEFINED.

Reserved for identifying IMPLEMENTATION DEFINED controls.

FRX, bit [31]

Feature Register extension. Defines whether ERR<n>FR[63:48] are architecturally defined.

`0b0` ERR<n>FR[63:48] are IMPLEMENTATION DEFINED.

`0b1` ERR<n>FR[63:48] are defined by the architecture.

Bits [30:26]

Reserved, RES0.

TS, bits [25:24]

Timestamp Extension. Indicates whether, for each error record <m> owned by this node, ERR<n>MISC3 is used as the timestamp register, and, if it is, the timebase used by the timestamp.

`0b00` The node does not support a timestamp register.

`0b01` The node implements a timestamp register. The timestamp uses the same timebase as the system Generic Timer.

Note

For an error record which has an affinity to a PE, this is the same timer that is visible through CNTPCT_EL0 at the highest Exception level on that PE.

`0b10` The node implements a timestamp register. The timebase for the timestamp is IMPLEMENTATION DEFINED.

All other values are reserved.

CI, bits [23:22]

Critical error interrupt. Indicates whether the critical error interrupt and associated controls are implemented.

`0b00` Does not support the critical error interrupt. ERR<n>CTLR.CI is RES0.

`0b01` Critical error interrupt is supported and always enabled. ERR<n>CTLR.CI is RES0.

`0b10` Critical error interrupt is supported and controllable using ERR<n>CTLR.CI.

All other values are reserved.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

INJ, bits [21:20]

Fault Injection Extension. Indicates whether the RAS Common Fault Injection Model Extension is implemented.

`0b00` The node does not support the RAS Common Fault Injection Model Extension.

`0b01` The node implements the RAS Common Fault Injection Model Extension. See ERR<n>PFGF for more information.

All other values are reserved.

CEO, bits [19:18]

When ERR<n>FR.CEC != 0b00:

Corrected Error overwrite. Indicates the behavior when a second Corrected error is detected after a first Corrected error has been recorded by an error record <m> owned by the node.

`0b00` Counts Corrected errors if a counter is implemented. Keeps the previous error syndrome. If the counter overflows, or no counter is implemented, then ERR<n>STATUS.OF is set to `0b1`.

`0b01` Counts Corrected errors. If ERR<n>STATUS.OF == `0b1` before the Corrected error is counted, then keeps the previous syndrome. Otherwise the previous syndrome is overwritten. If the counter overflows, then ERR<n>STATUS.OF is set to `0b1`.

All other values are reserved.

Otherwise:

Reserved, RES0.

DUI, bits [17:16]

When ERR<n>FR.UI != 0b00:

Error recovery interrupt for deferred errors control. Indicates whether the control for enabling error recovery interrupts on deferred errors are implemented.

`0b00` Does not support the control for enabling error recovery interrupts on deferred errors. ERR<n>CTLR.DUI is RES0.

`0b10` Control for enabling error recovery interrupts on deferred errors is supported and controllable using ERR<n>CTLR.DUI.

`0b11` Control for enabling error recovery interrupts on deferred errors is supported and controllable using ERR<n>CTLR.WDUI for writes and ERR<n>CTLR.RDUI for reads.

All other values are reserved.

Otherwise:

Reserved, RES0.

RP, bit [15]

When ERR<n>FR.CEC != 0b00:

Repeat counter. Indicates whether the node implements the repeat Corrected error counter in ERR<n>MISC0 for each error record <m> owned by the node that implements the standard Corrected error counter.

`0b0` A single CE counter is implemented.

`0b1` A first (repeat) counter and a second (other) counter are implemented. The repeat counter is the same size as the primary error counter.

Otherwise:

Reserved, RES0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CEC, bits [14:12]

Corrected Error Counter. Indicates whether the node implements the standard Corrected error counter (CE counter) mechanisms in ERR<n>MISC0 for each error record <m> owned by the node that can record countable errors.

`0b000` Does not implement the standard Corrected error counter model.

`0b010` Implements an 8-bit Corrected error counter in ERR<n>MISC0[39:32].

`0b100` Implements a 16-bit Corrected error counter in ERR<n>MISC0[47:32].

All other values are reserved.

Note

Implementations might include other error counter models, or might include the standard model and not indicate this in ERR<n>FR.

CFI, bits [11:10]

When ERR<n>FR.FI != 0b00:

Fault handling interrupt for corrected errors. Indicates whether the control for enabling fault handling interrupts on corrected errors are implemented.

`0b00` Does not support the control for enabling fault handling interrupts on corrected errors. ERR<n>CTLR.CFI is RES0.

`0b10` Control for enabling fault handling interrupts on corrected errors is supported and controllable using ERR<n>CTLR.CFI.

`0b11` Control for enabling fault handling interrupts on corrected errors is supported and controllable using ERR<n>CTLR.WCFI for writes and ERR<n>CTLR.RCFI for reads.

All other values are reserved.

Otherwise:

Reserved, RES0.

UE, bits [9:8]

In-band uncorrected error reporting. Indicates whether the in-band uncorrected error reporting (External Aborts) and associated controls are implemented.

`0b00` Does not support the in-band uncorrected error reporting (External Aborts). ERR<n>CTLR.UE is RES0.

`0b01` In-band uncorrected error reporting (External Aborts) is supported and always enabled. ERR<n>CTLR.UE is RES0.

`0b10` In-band uncorrected error reporting (External Aborts) is supported and controllable using ERR<n>CTLR.UE.

`0b11` In-band uncorrected error reporting (External Aborts) is supported and controllable using ERR<n>CTLR.WUE for writes and ERR<n>CTLR.RUE for reads.

FI, bits [7:6]

Fault handling interrupt. Indicates whether the fault handling interrupt and associated controls are implemented.

`0b00` Does not support the fault handling interrupt. ERR<n>CTLR.FI is RES0.

`0b01` Fault handling interrupt is supported and always enabled. ERR<n>CTLR.FI is RES0.

`0b10` Fault handling interrupt is supported and controllable using ERR<n>CTLR.FI.

`0b11` Fault handling interrupt is supported and controllable using ERR<n>CTLR.WFI for writes and ERR<n>CTLR.RFI for reads.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

UI, bits [5:4]

Error recovery interrupt for uncorrected errors. Indicates whether the error handling interrupt and associated controls are implemented.

`0b00` Does not support the error handling interrupt. ERR<n>CTLR.UI is RES0.

`0b01` Error handling interrupt is supported and always enabled. ERR<n>CTLR.UI is RES0.

`0b10` Error handling interrupt is supported and controllable using ERR<n>CTLR.UI.

`0b11` Error handling interrupt is supported and controllable using ERR<n>CTLR.WUI for writes and ERR<n>CTLR.RUI for reads.

IMPLEMENTATION DEFINED, bits [3:2]

IMPLEMENTATION DEFINED.

ED, bits [1:0]

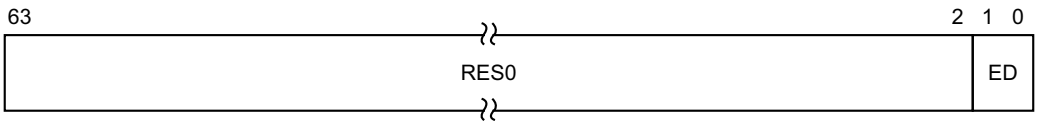
Error reporting and logging. Indicates whether error record <n> is the first record owned the node, and, if so, whether it implements the controls for enabling and disabling error reporting and logging.

`0b01` Error reporting and logging always enabled. ERR<n>CTLR.ED is RES0.

`0b10` Error reporting and logging is controllable using ERR<n>CTLR.ED.

All other values are reserved.

When ERR<n>FR.ED == 0b00:



Bits [63:2]

Reserved, RES0.

ED, bits [1:0]

Error reporting and logging. Indicates error record <n> is not the first record owned the node.

`0b00` Error record <n> is not the first record owned by the node.

This field reads as `0b00`.

Accessing the ERR<n>FR:

ERR<n>FR can be accessed through its memory-mapped interface:

Component	Offset	Instance
RAS	`0x000` + 64n	ERR<n>FR

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.24 ERR<n>MISC0, Error Record Miscellaneous Register 0, n = 0 - 65534

The ERR<n>MISC0 characteristics are:

Purpose

IMPLEMENTATION DEFINED error syndrome register. The miscellaneous syndrome registers might contain:

- Information to identify the FRU in which the error was detected, and might contain enough information to locate the error within that FRU.
- A Corrected error counter or counters.
- Other state information not present in the corresponding status and address registers.

If the node that owns error record <n> implements architecturally-defined error counters (ERR<n>FR.CEC != `0b000`), and error record <n> can record countable errors, then ERR<n>MISC0 implements the architecturally-defined error counter or counters.

For details of this, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*.

Configurations

This register is present only when error record <n> is implemented. Otherwise, direct accesses to ERR<n>MISC0 are RES0.

ERR<n>FR describes the features implemented by the node that owns error record <n>. <q> is the index of the first error record owned by the same node as error record <n>. If the node owns a single record, then q = n.

For IMPLEMENTATION DEFINED fields in ERR<n>MISC0, writing zero must always be supported to return the error record to an initial quiescent state.

In particular, if any IMPLEMENTATION DEFINED syndrome fields might generate a Fault Handling or Error Recovery Interrupt request, writing zero is sufficient to deactivate the Interrupt request.

Fields that are read-only, non-zero, and ignore writes are compliant with this requirement.

Note

Arm recommends that any IMPLEMENTATION DEFINED syndrome fields that can generate a Fault Handling, Error Recovery, Critical, or IMPLEMENTATION DEFINED, interrupt request are disabled at Cold reset and are enabled by software writing an IMPLEMENTATION DEFINED non-zero value to an IMPLEMENTATION DEFINED field in ERR<q>CTRL.

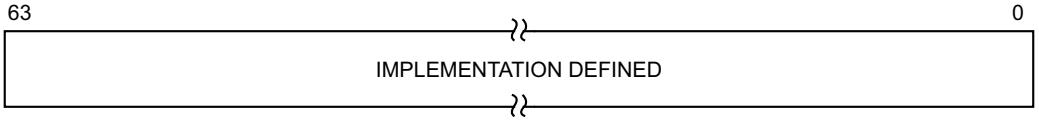
Attributes

ERR<n>MISC0 is a 64-bit register.

Field descriptions

The ERR<n>MISC0 bit assignments are:

When ERR<q>FR.CEC == 0b000:



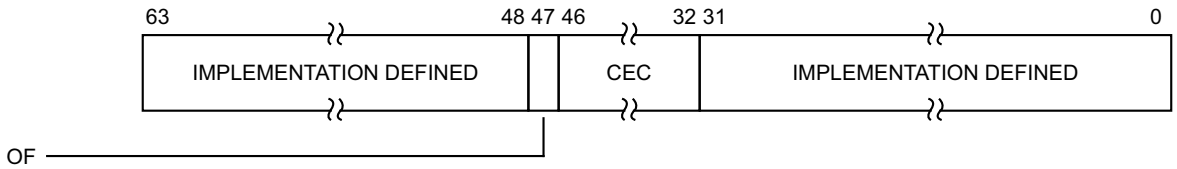
IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED syndrome.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When ERR<q>FR.CEC == 0b100 and ERR<q>FR.RP == 0:



IMPLEMENTATION DEFINED, bits [63:48]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED syndrome.

OF, bit [47]

Sticky overflow bit. Set to 1 when the Corrected error count field is incremented and wraps through zero.

`0b0` Counter has not overflowed.

`0b1` Counter has overflowed.

A direct write that modifies this bit might indirectly set ERR<n>STATUS.OF to an UNKNOWN value and a direct write to ERR<n>STATUS.OF that clears it to zero might indirectly set this bit to an UNKNOWN value.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

CEC, bits [46:32]

Corrected error count. Incremented for each Corrected error. It is IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether Deferred and Uncorrected errors are counted.

The following resets apply:

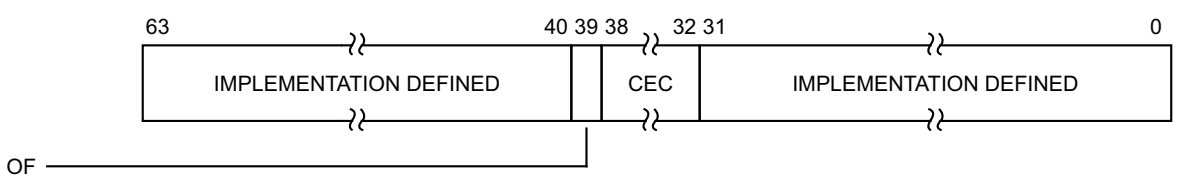
- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED syndrome.

When ERR<q>FR.CEC == 0b010 and ERR<q>FR.RP == 0:



IMPLEMENTATION DEFINED, bits [63:40]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED syndrome.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

OF, bit [39]

Sticky overflow bit. Set to 1 when the Corrected error count field is incremented and wraps through zero.

`0b0` Counter has not overflowed.

`0b1` Counter has overflowed.

A direct write that modifies this bit might indirectly set ERR<n>STATUS.OF to an UNKNOWN value and a direct write to ERR<n>STATUS.OF that clears it to zero might indirectly set this bit to an UNKNOWN value.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

CEC, bits [38:32]

Corrected error count. Incremented for each Corrected error. It is IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether Deferred and Uncorrected errors are counted.

The following resets apply:

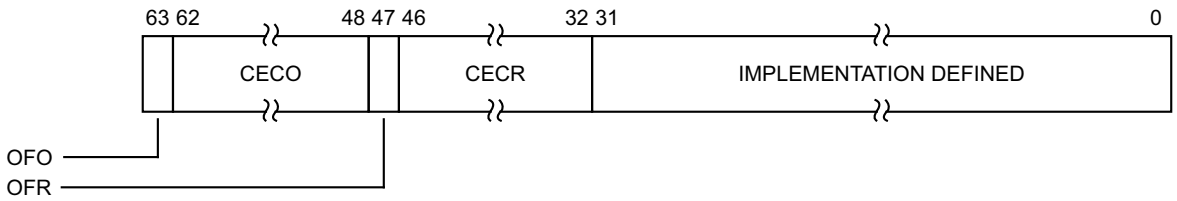
- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED syndrome.

When ERR<q>FR.CEC == 0b100 and ERR<q>FR.RP == 1:



OFO, bit [63]

Sticky overflow bit, other. Set to 1 when the Corrected error count, other, field is incremented and wraps through zero.

`0b0` Other counter has not overflowed.

`0b1` Other counter has overflowed.

A direct write that modifies this bit might indirectly set ERR<n>STATUS.OF to an UNKNOWN value and a direct write to ERR<n>STATUS.OF that clears it to zero might indirectly set this bit to an UNKNOWN value.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

CECO, bits [62:48]

Corrected error count, other. Incremented for each countable error that is not accounted for by incrementing CECR.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

OFR, bit [47]

Sticky overflow bit, repeat. Set to 1 when the Corrected error count, repeat, field is incremented and wraps through zero.

`0b0` Repeat counter has not overflowed.

`0b1` Repeat counter has overflowed.

A direct write that modifies this bit might indirectly set ERR<n>STATUS.OF to an UNKNOWN value and a direct write to ERR<n>STATUS.OF that clears it to zero might indirectly set this bit to an UNKNOWN value.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

CECR, bits [46:32]

Corrected error count, repeat. Incremented for the first countable error, which also records other syndrome for the error, and subsequently for each countable error that matches the recorded other syndrome. Corrected errors are countable errors. It is IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether Deferred and Uncorrected errors are countable errors.

Note

For example, the other syndrome might include the set and way information for an error detected in a cache. This might be recorded in the IMPLEMENTATION DEFINED ERR<n>MISC<m> fields on a first Corrected error. CECR is then incremented for each subsequent Corrected Error in the same set and way.

The following resets apply:

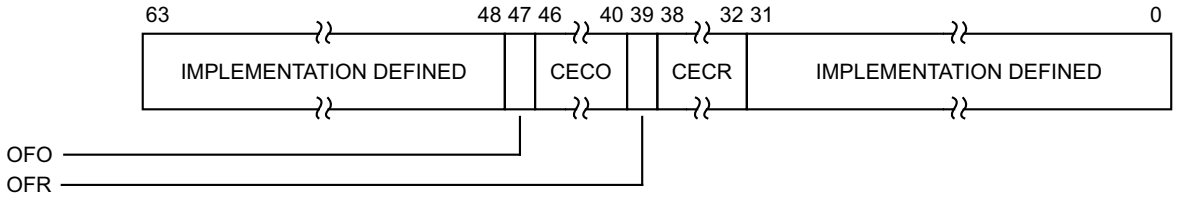
- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED syndrome.

When ERR<q>FR.CEC == 0b010 and ERR<q>FR.RP == 1:



IMPLEMENTATION DEFINED, bits [63:48]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED syndrome.

OFO, bit [47]

Sticky overflow bit, other. Set to 1 when the Corrected error count, other, field is incremented and wraps through zero.

`0b0` Other counter has not overflowed.

`0b1` Other counter has overflowed.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A direct write that modifies this bit might indirectly set ERR<n>STATUS.OF to an UNKNOWN value and a direct write to ERR<n>STATUS.OF that clears it to zero might indirectly set this bit to an UNKNOWN value.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

CECO, bits [46:40]

Corrected error count, other. Incremented for each countable error that is not accounted for by incrementing CECR.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

OFR, bit [39]

Sticky overflow bit, repeat. Set to 1 when the Corrected error count, repeat, field is incremented and wraps through zero.

`0b0` Repeat counter has not overflowed.

`0b1` Repeat counter has overflowed.

A direct write that modifies this bit might indirectly set ERR<n>STATUS.OF to an UNKNOWN value and a direct write to ERR<n>STATUS.OF that clears it to zero might indirectly set this bit to an UNKNOWN value.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

CECR, bits [38:32]

Corrected error count, repeat. Incremented for the first countable error, which also records other syndrome for the error, and subsequently for each countable error that matches the recorded other syndrome. Corrected errors are countable errors. It is IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether Deferred and Uncorrected errors are countable errors.

Note

For example, the other syndrome might include the set and way information for an error detected in a cache. This might be recorded in the IMPLEMENTATION DEFINED ERR<n>MISC<m> fields on a first Corrected error. CECR is then incremented for each subsequent Corrected Error in the same set and way.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED syndrome.

Accessing the ERR<n>MISC0:

Reads from ERR<n>MISC0 return an IMPLEMENTATION DEFINED value and writes have IMPLEMENTATION DEFINED behavior.

Arm recommends that miscellaneous syndrome for multiple errors, such as a corrected error counter, is read/write.

When ERR<n>STATUS.MV == `0b1`, the miscellaneous syndrome specific to the most recently recorded error should ignore writes.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

These recommendations allow a counter to be reset in the presence of a persistent error, while preventing specific information, such as that identifying a FRU, from being lost if an error is detected while the previous error is being logged.

ERR<n>MISC0 can be accessed through its memory-mapped interface:

Component	Offset	Instance
RAS	`0x020` + 64n	ERR<n>MISC0

This interface is accessible as follows:

- Accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.25 ERR<n>MISC1, Error Record Miscellaneous Register 1, n = 0 - 65534

The ERR<n>MISC1 characteristics are:

Purpose

IMPLEMENTATION DEFINED error syndrome register. The miscellaneous syndrome registers might contain:

- Information to identify the FRU in which the error was detected, and might contain enough information to locate the error within that FRU.
- A Corrected error counter or counters.
- Other state information not present in the corresponding status and address registers.

Configurations

This register is present only when error record <n> is implemented. Otherwise, direct accesses to ERR<n>MISC1 are RES0.

ERR<n>FR describes the features implemented by the node that owns error record <n>. <q> is the index of the first error record owned by the same node as error record <n>. If the node owns a single record, then q = n.

For IMPLEMENTATION DEFINED fields in ERR<n>MISC1, writing zero must always be supported to return the error record to an initial quiescent state.

In particular, if any IMPLEMENTATION DEFINED syndrome fields might generate a Fault Handling or Error Recovery Interrupt request, writing zero is sufficient to deactivate the Interrupt request.

Fields that are read-only, non-zero, and ignore writes are compliant with this requirement.

Note

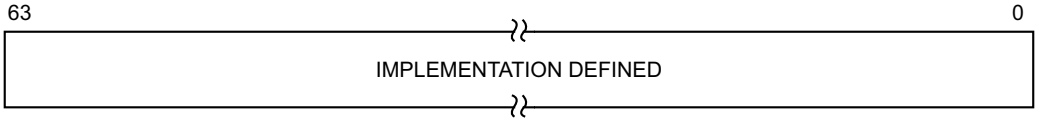
Arm recommends that any IMPLEMENTATION DEFINED syndrome fields that can generate a Fault Handling, Error Recovery, Critical, or IMPLEMENTATION DEFINED, interrupt request are disabled at Cold reset and are enabled by software writing an IMPLEMENTATION DEFINED non-zero value to an IMPLEMENTATION DEFINED field in ERR<q>CTRL.

Attributes

ERR<n>MISC1 is a 64-bit register.

Field descriptions

The ERR<n>MISC1 bit assignments are:



IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED syndrome.

Accessing the ERR<n>MISC1:

Reads from ERR<n>MISC1 return an IMPLEMENTATION DEFINED value and writes have IMPLEMENTATION DEFINED behavior.

Arm recommends that miscellaneous syndrome for multiple errors, such as a corrected error counter, is read/write.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When ERR<n>STATUS.MV == `0b1`, the miscellaneous syndrome specific to the most recently recorded error should ignore writes.

Note

These recommendations allow a counter to be reset in the presence of a persistent error, while preventing specific information, such as that identifying a FRU, from being lost if an error is detected while the previous error is being logged.

ERR<n>MISC1 can be accessed through its memory-mapped interface:

Component	Offset	Instance
RAS	`0x028` + 64n	ERR<n>MISC1

This interface is accessible as follows:

- Accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.26 ERR<n>MISC2, Error Record Miscellaneous Register 2, n = 0 - 65534

The ERR<n>MISC2 characteristics are:

Purpose

IMPLEMENTATION DEFINED error syndrome register. The miscellaneous syndrome registers might contain:

- Information to identify the FRU in which the error was detected, and might contain enough information to locate the error within that FRU.
- A Corrected error counter or counters.
- Other state information not present in the corresponding status and address registers.

Configurations

This register is present only when error record <n> is implemented. Otherwise, direct accesses to ERR<n>MISC2 are RES0.

ERR<n>FR describes the features implemented by the node that owns error record <n>. <q> is the index of the first error record owned by the same node as error record <n>. If the node owns a single record, then q = n.

For IMPLEMENTATION DEFINED fields in ERR<n>MISC2, writing zero must always be supported to return the error record to an initial quiescent state.

In particular, if any IMPLEMENTATION DEFINED syndrome fields might generate a Fault Handling or Error Recovery Interrupt request, writing zero is sufficient to deactivate the Interrupt request.

Fields that are read-only, non-zero, and ignore writes are compliant with this requirement.

If RAS System Architecture v1.1 is not implemented, Arm recommends that ERR<n>MISC2 does not require zeroing to return the record to a quiescent state.

Note

Arm recommends that any IMPLEMENTATION DEFINED syndrome fields that can generate a Fault Handling, Error Recovery, Critical, or IMPLEMENTATION DEFINED, interrupt request are disabled at Cold reset and are enabled by software writing an IMPLEMENTATION DEFINED non-zero value to an IMPLEMENTATION DEFINED field in ERR<q>CTRL.

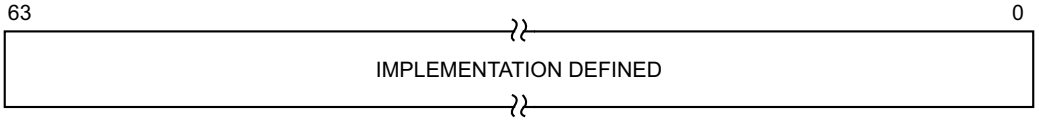
It is IMPLEMENTATION DEFINED whether ERR<n>MISC2 is present if RAS System Architecture v1.1 is not implemented. ERR<n>MISC2 is RES0 if not present.

Attributes

ERR<n>MISC2 is a 64-bit register.

Field descriptions

The ERR<n>MISC2 bit assignments are:



IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED syndrome.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the ERR<n>MISC2:

Reads from ERR<n>MISC2 return an IMPLEMENTATION DEFINED value and writes have IMPLEMENTATION DEFINED behavior.

Arm recommends that miscellaneous syndrome for multiple errors, such as a corrected error counter, is read/write.

When ERR<n>STATUS.MV == `0b1`, the miscellaneous syndrome specific to the most recently recorded error should ignore writes.

Note

These recommendations allow a counter to be reset in the presence of a persistent error, while preventing specific information, such as that identifying a FRU, from being lost if an error is detected while the previous error is being logged.

ERR<n>MISC2 can be accessed through its memory-mapped interface:

Component	Offset	Instance
RAS	`0x030` + 64n	ERR<n>MISC2

This interface is accessible as follows:

- Accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.27 ERR<n>MISC3, Error Record Miscellaneous Register 3, n = 0 - 65534

The ERR<n>MISC3 characteristics are:

Purpose

IMPLEMENTATION DEFINED error syndrome register. The miscellaneous syndrome registers might contain:

- Information to identify the FRU in which the error was detected, and might contain enough information to locate the error within that FRU.
- A Corrected error counter or counters.
- Other state information not present in the corresponding status and address registers.

If the node that owns error record n supports the RAS Timestamp Extension (ERR<n>FR.TS != `0b00`), then ERR<n>MISC3 contains the timestamp value for error record n when the error was detected. Otherwise the contents of ERR<n>MISC3 are IMPLEMENTATION DEFINED.

For details of this, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*.

Configurations

This register is present only when error record <n> is implemented. Otherwise, direct accesses to ERR<n>MISC3 are RES0.

ERR<n>FR describes the features implemented by the node that owns error record <n>. <q> is the index of the first error record owned by the same node as error record <n>. If the node owns a single record, then q = n.

For IMPLEMENTATION DEFINED fields in ERR<n>MISC3, writing zero must always be supported to return the error record to an initial quiescent state.

In particular, if any IMPLEMENTATION DEFINED syndrome fields might generate a Fault Handling or Error Recovery Interrupt request, writing zero is sufficient to deactivate the Interrupt request.

Fields that are read-only, non-zero, and ignore writes are compliant with this requirement.

If RAS System Architecture v1.1 is not implemented, Arm recommends that ERR<n>MISC3 does not require zeroing to return the record to a quiescent state.

Note

Arm recommends that any IMPLEMENTATION DEFINED syndrome fields that can generate a Fault Handling, Error Recovery, Critical, or IMPLEMENTATION DEFINED, interrupt request are disabled at Cold reset and are enabled by software writing an IMPLEMENTATION DEFINED non-zero value to an IMPLEMENTATION DEFINED field in ERR<q>CTRL.

It is IMPLEMENTATION DEFINED whether ERR<n>MISC3 is present if RAS System Architecture v1.1 is not implemented. ERR<n>MISC3 is RES0 if not present.

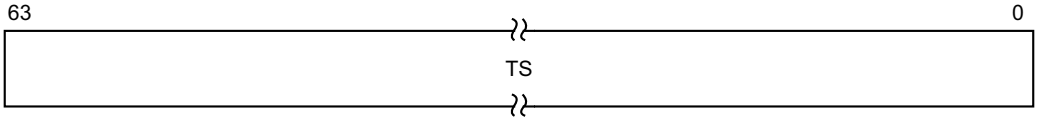
Attributes

ERR<n>MISC3 is a 64-bit register.

Field descriptions

The ERR<n>MISC3 bit assignments are:

When ERR<q>FR.TS != 0b00:



Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

TS, bits [63:0]

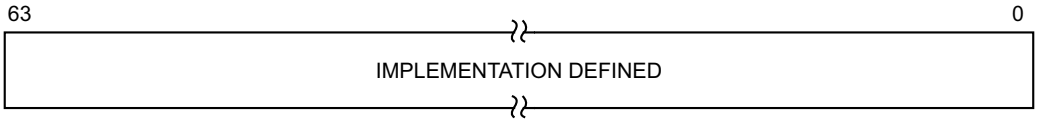
Timestamp. Timestamp value recorded when the error was detected. Valid only if ERR<n>STATUS.V == `0b1`.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO or RW.

When ERR<q>FR.TS == 0b00:



IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED syndrome.

Accessing the ERR<n>MISC3:

Reads from ERR<n>MISC3 return an IMPLEMENTATION DEFINED value and writes have IMPLEMENTATION DEFINED behavior.

Arm recommends that miscellaneous syndrome for multiple errors, such as a corrected error counter, is read/write.

When ERR<n>STATUS.MV == `0b1`, the miscellaneous syndrome specific to the most recently recorded error should ignore writes.

Note

These recommendations allow a counter to be reset in the presence of a persistent error, while preventing specific information, such as that identifying a FRU, from being lost if an error is detected while the previous error is being logged.

ERR<n>MISC3 can be accessed through its memory-mapped interface:

Component	Offset	Instance
RAS	`0x038` + 64n	ERR<n>MISC3

This interface is accessible as follows:

- Accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.28 ERR<n>PFGCDN, Pseudo-fault Generation Countdown Register, n = 0 - 65534

The ERR<n>PFGCDN characteristics are:

Purpose

Generates one of the errors enabled in the corresponding ERR<n>PFGCTL register.

For details of this, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*.

Configurations

This register is present only when error record <n> is implemented, the node implements the RAS Common Fault Injection Model Extension (ERR<n>FR.INJ != 0b00) and error record <n> is the first error record owned by a node. Otherwise, direct accesses to ERR<n>PFGCDN are RES0.

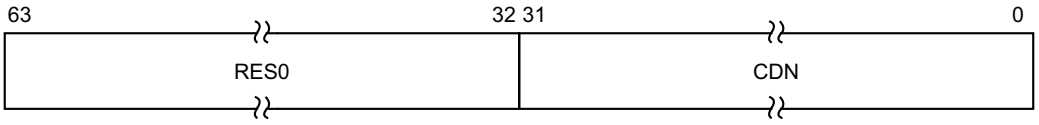
ERR<n>FR describes the features implemented by the node.

Attributes

ERR<n>PFGCDN is a 64-bit register.

Field descriptions

The ERR<n>PFGCDN bit assignments are:



Bits [63:32]

Reserved, RES0.

CDN, bits [31:0]

Countdown value.

This field is copied to Error Generation Counter when either:

- Software writes ERR<n>PFGCTL.CDNEN with 1.
- The Error Generation Counter decrements to zero and ERR<n>PFGCTL.R == `0b1`.

While ERR<n>PFGCTL.CDNEN == `0b1` and the Error Generation Counter is nonzero, the counter decrements by 1 for each cycle at an IMPLEMENTATION DEFINED clock rate. When the counter reaches 0, one of the errors enabled in the ERR<n>PFGCTL register is generated.

Note

The current Error Generation Counter value is not visible to software.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing the ERR<n>PFGCDN:

ERR<n>PFGCDN can be accessed through its memory-mapped interface:

Component	Offset	Instance
RAS	`0x810` + 64n	ERR<n>PFGCDN

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This interface is accessible as follows:

- Accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.29 ERR<n>PFGCTL, Pseudo-fault Generation Control Register, n = 0 - 65534

The ERR<n>PFGCTL characteristics are:

Purpose

Enables controlled fault generation.

Configurations

This register is present only when error record <n> is implemented, the node implements the RAS Common Fault Injection Model Extension (ERR<n>FR.INJ != 0b00) and error record <n> is the first error record owned by a node. Otherwise, direct accesses to ERR<n>PFGCTL are RES0.

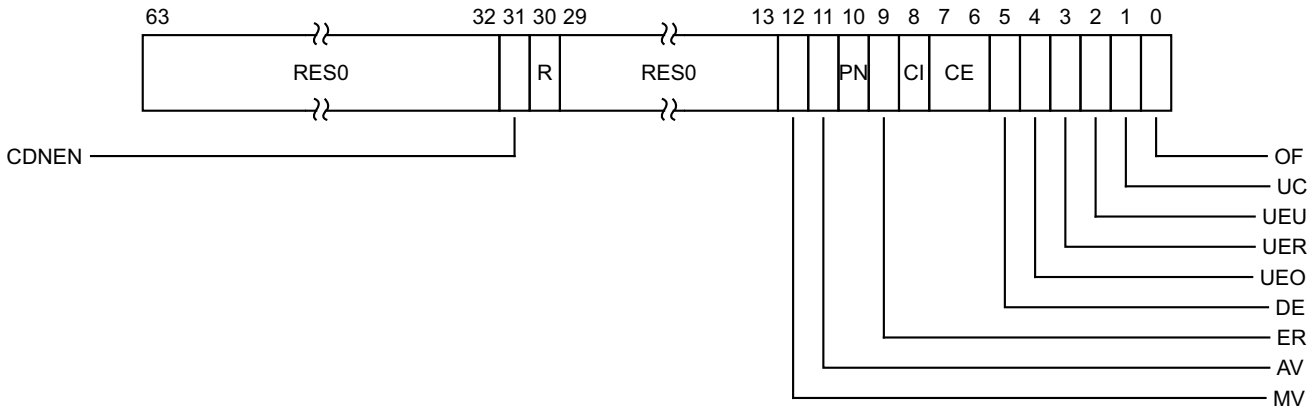
ERR<n>FR describes the features implemented by the node.

Attributes

ERR<n>PFGCTL is a 64-bit register.

Field descriptions

The ERR<n>PFGCTL bit assignments are:



Bits [63:32]

Reserved, RES0.

CDNEN, bit [31]

Countdown Enable. Controls transfers from the value that is held in the ERR<n>PFGCDN into the Error Generation Counter and enables this counter.

`0b0` The Error Generation Counter is disabled.

`0b1` The Error Generation Counter is enabled. On a write of `0b1` to this bit, the Error Generation Counter is set to ERR<n>PFGCDN.CDN.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to `0`.

R, bit [30]

Restart. Controls whether, upon reaching zero, the Error Generation Counter restarts from the ERR<n>PFGCDN value or stops.

`0b0` On reaching 0, the Error Generation Counter will stop.

`0b1` On reaching 0, the Error Generation Counter is set to ERR<n>PFGCDN.CDN.

This bit is RES0 if the node does not support this control.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [29:13]

Reserved, RES0.

MV, bit [12]

Miscellaneous syndrome. The value that is written to ERR<n>STATUS.MV when an injected error is recorded.

`0b0` ERR<n>STATUS.MV is set to `0b0` when an injected error is recorded.

`0b1` ERR<n>STATUS.MV is set to `0b1` when an injected error is recorded.

This bit reads-as-one if the node always records some syndrome in ERR<n>MISC<m>, setting ERR<n>STATUS.MV to 1, when an injected error is recorded. This bit is RES0 if the node does not support this control.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

AV, bit [11]

Address syndrome. The value that is written to ERR<n>STATUS.AV when an injected error is recorded.

`0b0` ERR<n>STATUS.AV is set to `0b0` when an injected error is recorded.

`0b1` ERR<n>STATUS.AV is set to `0b1` when an injected error is recorded.

This bit reads-as-one if the node always sets ERR<n>STATUS.AV to `0b1` when an injected error is recorded. This bit is RES0 if the node does not support this control.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

PN, bit [10]

Poison flag. The value that is written to ERR<n>STATUS.PN when an injected error is recorded.

`0b0` ERR<n>STATUS.PN is set to `0b0` when an injected error is recorded.

`0b1` ERR<n>STATUS.PN is set to `0b1` when an injected error is recorded.

This bit is RES0 if the node does not support this control.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

ER, bit [9]

Error Reported flag. The value that is written to ERR<n>STATUS.ER when an injected error is recorded.

`0b0` ERR<n>STATUS.ER is set to `0b0` when an injected error is recorded.

`0b1` ERR<n>STATUS.ER is set to `0b1` when an injected error is recorded.

This bit is RES0 if the node does not support this control.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CI, bit [8]

Critical Error flag. The value that is written to ERR<n>STATUS.CI when an injected error is recorded.

`0b0` ERR<n>STATUS.CI is set to `0b0` when an injected error is recorded.

`0b1` ERR<n>STATUS.CI is set to `0b1` when an injected error is recorded.

This bit is RES0 if the node does not support this control.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

CE, bits [7:6]

Corrected Error generation enable. Controls the type of Corrected Error condition that might be generated.

`0b00` No error of this type will be generated.

`0b01` A non-specific Corrected Error, that is, a Corrected Error that is recorded as ERR<n>STATUS.CE == `0b10`, might be generated when the Error Generation Counter decrements to zero.

`0b10` A transient Corrected Error, that is, a Corrected Error that is recorded as ERR<n>STATUS.CE == `0b01`, might be generated when the Error Generation Counter decrements to zero.

`0b11` A persistent Corrected Error, that is, a Corrected Error that is recorded as ERR<n>STATUS.CE == `0b11`, might be generated when the Error Generation Counter decrements to zero.

The set of permitted values for this field is defined by ERR<n>PFGF.CE.

This field is RES0 if the node does not support this control.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

DE, bit [5]

Deferred Error generation enable. Controls whether this type of error condition might be generated. It is IMPLEMENTATION DEFINED whether the error is generated if the data is not consumed.

`0b0` No error of this type will be generated.

`0b1` An error of this type might be generated when the Error Generation Counter decrements to zero.

This bit is RES0 if the node does not support this control.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

UEO, bit [4]

Latent or Restartable Error generation enable. Controls whether this type of error condition might be generated. It is IMPLEMENTATION DEFINED whether the error is generated if the data is not consumed.

`0b0` No error of this type will be generated.

`0b1` An error of this type might be generated when the Error Generation Counter decrements to zero.

This bit is RES0 if the node does not support this control.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

UER, bit [3]

Signaled or Recoverable Error generation enable. Controls whether this type of error condition might be generated. It is IMPLEMENTATION DEFINED whether the error is generated if the data is not consumed.

`0b0` No error of this type will be generated.

`0b1` An error of this type might be generated when the Error Generation Counter decrements to zero.

This bit is RES0 if the node does not support this control.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

UEU, bit [2]

Unrecoverable Error generation enable. Controls whether this type of error condition might be generated. It is IMPLEMENTATION DEFINED whether the error is generated if the data is not consumed.

`0b0` No error of this type will be generated.

`0b1` An error of this type might be generated when the Error Generation Counter decrements to zero.

This bit is RES0 if the node does not support this control.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

UC, bit [1]

Uncontainable Error generation enable. Controls whether this type of error condition might be generated. It is IMPLEMENTATION DEFINED whether the error is generated if the data is not consumed.

`0b0` No error of this type will be generated.

`0b1` An error of this type might be generated when the Error Generation Counter decrements to zero.

This bit is RES0 if the node does not support this control.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

OF, bit [0]

Overflow flag. The value that is written to ERR<n>STATUS.OF when an injected error is recorded.

`0b0` ERR<n>STATUS.OF is set to `0b0` when an injected error is recorded.

`0b1` ERR<n>STATUS.OF is set to `0b1` when an injected error is recorded.

This bit is RES0 if the node does not support this control.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the ERR<n>PFGCTL:

ERR<n>PFGCTL can be accessed through its memory-mapped interface:

Component	Offset	Instance
RAS	`0x808` + 64n	ERR<n>PFGCTL

This interface is accessible as follows:

- Accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.30 ERR<n>PFGF, Pseudo-fault Generation Feature Register, n = 0 - 65534

The ERR<n>PFGF characteristics are:

Purpose

Defines which common architecturally-defined fault generation features are implemented.

Configurations

This register is present only when error record <n> is implemented, the node implements the RAS Common Fault Injection Model Extension (ERR<n>FR.INJ != 0b00) and error record <n> is the first error record owned by a node. Otherwise, direct accesses to ERR<n>PFGF are RES0.

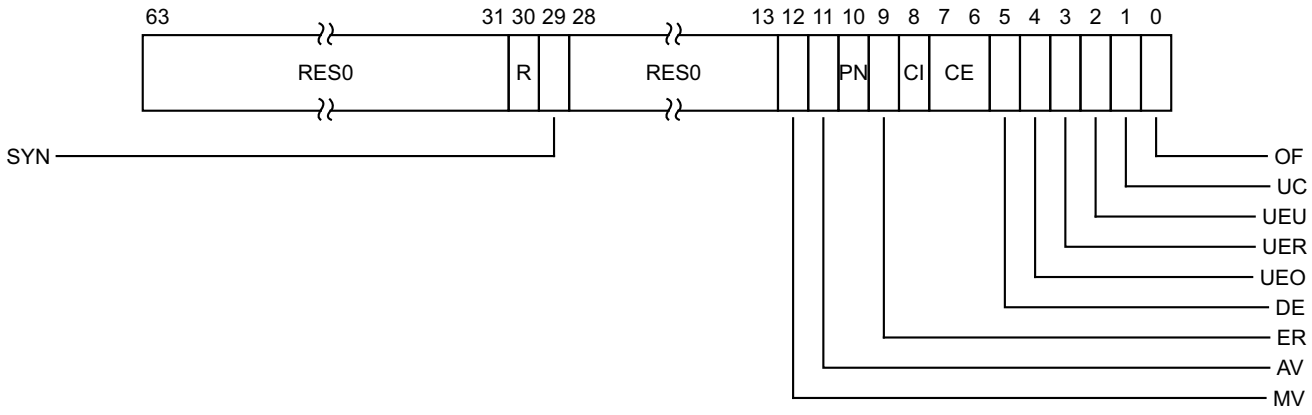
ERR<n>FR describes the features implemented by the node.

Attributes

ERR<n>PFGF is a 64-bit register.

Field descriptions

The ERR<n>PFGF bit assignments are:



Bits [63:31]

Reserved, RES0.

R, bit [30]

Restartable. Support for Error Generation Counter restart mode.

`0b0` The node does not support this feature.

`0b1` Feature controllable.

SYN, bit [29]

Syndrome. Fault syndrome injection.

`0b0` When an injected error is recorded, the node sets ERR<n>STATUS.{IERR, SERR} to IMPLEMENTATION DEFINED values. ERR<n>STATUS.{IERR, SERR} are UNKNOWN when ERR<n>STATUS.V == `0b0`.

`0b1` When an injected error is recorded, the node does not update the ERR<n>STATUS.{IERR, SERR} fields. ERR<n>STATUS.{IERR, SERR} are writable when ERR<n>STATUS.V == `0b0`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

If ERR<n>PFGF.SYN == `0b1`, software can write specific values into the ERR<n>STATUS.{IERR, SERR} fields when setting up a fault injection event. The sets of values that can be written to these fields is IMPLEMENTATION DEFINED.

Bits [28:13]

Reserved, RES0.

MV, bit [12]

Miscellaneous syndrome.

Additional syndrome injection. Defines whether software can control all or part of the syndrome recorded in the ERR<n>MISC<m> registers when an injected error is recorded.

It is IMPLEMENTATION DEFINED which syndrome fields in ERR<n>MISC<m> this refers to, as some fields might always be recorded by an error. For example, a Corrected Error counter.

`0b0` When an injected error is recorded, the node might record IMPLEMENTATION DEFINED additional syndrome in ERR<n>MISC<m>. If any syndrome is recorded in ERR<n>MISC<m>, then ERR<n>STATUS.MV is set to `0b1`.

`0b1` When an injected error is recorded, the node does not update all the syndrome fields in the ERR<n>MISC<m> and does one of:

- The node does not update any fields in ERR<n>MISC<m> and sets ERR<n>STATUS.MV to ERR<n>PFGCTL.MV.
- The node records some syndrome in ERR<n>MISC<m> and sets ERR<n>STATUS.MV to `0b1`. ERR<n>PGFCTL.MV is RAO.

The syndrome fields that the node does not update are unchanged and must be writable when ERR<n>STATUS.MV == `0b0`.

Note

If ERR<n>PFGF.MV == `0b1`, software can write specific values into the ERR<n>MISC<m> registers when setting up a fault injection event. The values that can be written to these registers are IMPLEMENTATION DEFINED.

AV, bit [11]

Address syndrome. Address syndrome injection.

`0b0` When an injected error is recorded, the node either sets ERR<n>ADDR and ERR<n>STATUS.AV for the access, or leaves these unchanged.

`0b1` When an injected error is recorded, the node does not update ERR<n>ADDR and does one of:

- Sets ERR<n>STATUS.AV to ERR<n>PFGCTL.AV.
- Sets ERR<n>STATUS.AV to `0b1`. ERR<n>PFGCTL.AV is RAO.

ERR<n>ADDR must be writable when ERR<n>STATUS.AV == `0b0`.

Note

If ERR<n>PFGF.AV == `0b1`, software can write a specific value into ERR<n>ADDR when setting up a fault injection event.

PN, bit [10]

Poison flag. Describes how the fault generation feature of the node sets the ERR<n>STATUS.PN status flag.

`0b0` When an injected error is recorded, it is IMPLEMENTATION DEFINED whether the node sets ERR<n>STATUS.PN to `0b1`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0b1` When an injected error is recorded, ERR<n>STATUS.PN is set to ERR<n>PFGCTL.PN.

This behavior replaces the architecture-defined rules for setting the PN bit.

This bit reads-as-zero if the node does not support this flag.

ER, bit [9]

Error Reported flag. Describes how the fault generation feature of the node sets the ERR<n>STATUS.ER status flag.

`0b0` When an injected error is recorded, the node sets ERR<n>STATUS.ER according to the architecture-defined rules for setting the ER bit.

`0b1` When an injected error is recorded, ERR<n>STATUS.ER is set to ERR<n>PFGCTL.ER. This behavior replaces the architecture-defined rules for setting the ER bit.

This bit reads-as-zero if the node does not support this flag.

CI, bit [8]

Critical Error flag. Describes how the fault generation feature of the node sets the ERR<n>STATUS.CI status flag.

`0b0` When an injected error is recorded, it is IMPLEMENTATION DEFINED whether the node sets ERR<n>STATUS.CI to `0b1`.

`0b1` When an injected error is recorded, ERR<n>STATUS.CI is set to ERR<n>PFGCTL.CI.

This behavior replaces the architecture-defined rules for setting the CI bit.

This bit reads-as-zero if the node does not support this flag.

CE, bits [7:6]

Corrected Error generation. Describes the types of Corrected Error that the fault generation feature of the node can generate.

`0b00` The fault generation feature of the node cannot generate this type of error.

`0b01` The fault generation feature of the node allows generation of a non-specific Corrected Error, that is, a Corrected Error that is recorded as ERR<n>STATUS.CE == `0b10`.

`0b11` The fault generation feature of the node allows generation of transient or persistent Corrected Errors, that is, Corrected Errors that are recorded as ERR<n>STATUS.CE == `0b01` and `0b11`.

All other values are reserved.

This field reads-as-zeros if the node does not support this type of error.

DE, bit [5]

Deferred Error generation. Describes whether the fault generation feature of the node can generate this type of error.

`0b0` The fault generation feature of the node cannot generate this type of error.

`0b1` The fault generation feature of the node allows generation of this type of error.

This bit reads-as-zero if the node does not support this type of error.

UEO, bit [4]

Latent or Restartable Error generation. Describes whether the fault generation feature of the node can generate this type of error.

`0b0` The fault generation feature of the node cannot generate this type of error.

`0b1` The fault generation feature of the node allows generation of this type of error.

This bit reads-as-zero if the node does not support this type of error.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

UER, bit [3]

Signaled or Recoverable Error generation. Describes whether the fault generation feature of the node can generate this type of error.

`0b0` The fault generation feature of the node cannot generate this type of error.

`0b1` The fault generation feature of the node allows generation of this type of error.

This bit reads-as-zero if the node does not support this type of error.

UEU, bit [2]

Unrecoverable Error generation. Describes whether the fault generation feature of the node can generate this type of error.

`0b0` The fault generation feature of the node cannot generate this type of error.

`0b1` The fault generation feature of the node allows generation of this type of error.

This bit reads-as-zero if the node does not support this type of error.

UC, bit [1]

Uncontainable Error generation. Describes whether the fault generation feature of the node can generate this type of error.

`0b0` The fault generation feature of the node cannot generate this type of error.

`0b1` The fault generation feature of the node allows generation of this type of error.

This bit reads-as-zero if the node does not support this type of error.

OF, bit [0]

Overflow flag. Describes how the fault generation feature of the node sets the ERR<n>STATUS.OF status flag.

`0b0` When an injected error is recorded, the node sets ERR<n>STATUS.OF according to the architecture-defined rules for setting the OF bit.

`0b1` When an injected error is recorded, ERR<n>STATUS.OF is set to ERR<n>PFGCTL.OF. This behavior replaces the architecture-defined rules for setting the OF bit.

This bit reads-as-zero if the node does not support this flag.

Accessing the ERR<n>PFGF:

ERR<n>PFGF can be accessed through its memory-mapped interface:

Component	Offset	Instance
RAS	`0x800` + 64n	ERR<n>PFGF

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.31 ERR<n>STATUS, Error Record Primary Status Register, n = 0 - 65534

The ERR<n>STATUS characteristics are:

Purpose

Contains status information for the error record, including:

- Whether any error has been detected (valid).
- Whether any detected error was not corrected, and returned to a master.
- Whether any detected error was not corrected and deferred.
- Whether an error record has been discarded because additional errors have been detected before the first error was handled by software (overflow).
- Whether any error has been reported.
- Whether the other error record registers contain valid information.
- Whether the error was reported because poison data was detected or because a corrupt value was detected by an error detection code.
- A primary error code.
- An IMPLEMENTATION DEFINED extended error code.

Within this register:

- The {AV, V, MV} bits are valid bits that define whether the error record registers are valid.
- The {UE, OF, CE, DE, UET} bits encode the types of error or errors recorded.
- The {CI, ER, PN, IERR, SERR} fields are syndrome fields.

Configurations

This register is present only when error record <n> is implemented. Otherwise, direct accesses to ERR<n>STATUS are RES0.

ERR<n>FR describes the features implemented by the node that owns error record <n>. <q> is the index of the first error record owned by the same node as error record <n>. If the node owns a single record, then q = n.

For IMPLEMENTATION DEFINED fields in ERR<n>STATUS, writing zero must always be supported to return the error record to an initial quiescent state.

In particular, if any IMPLEMENTATION DEFINED syndrome fields might generate a Fault Handling or Error Recovery Interrupt request, writing zero is sufficient to deactivate the Interrupt request.

Fields that are read-only, non-zero, and ignore writes are compliant with this requirement.

Note

Arm recommends that any IMPLEMENTATION DEFINED syndrome fields that can generate a Fault Handling, Error Recovery, Critical, or IMPLEMENTATION DEFINED, interrupt request are disabled at Cold reset and are enabled by software writing an IMPLEMENTATION DEFINED non-zero value to an IMPLEMENTATION DEFINED field in ERR<q>CTRL.

Attributes

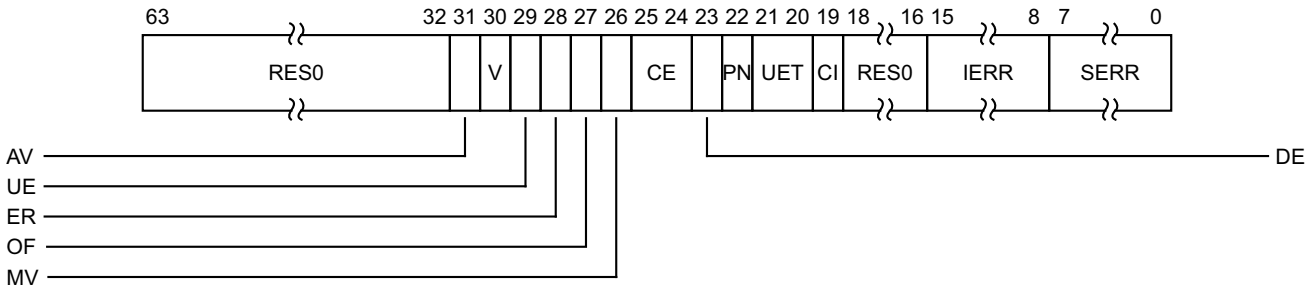
ERR<n>STATUS is a 64-bit register.

Field descriptions

The ERR<n>STATUS bit assignments are:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When RAS System Architecture v1.1 is implemented:



Bits [63:32]

Reserved, RES0.

AV, bit [31]

When the error record includes an address associated with an error:

Address Valid.

`0b0` ERR<n>ADDR not valid.

`0b1` ERR<n>ADDR contains an address associated with the highest priority error recorded by this record.

This bit is read/write-one-to-clear.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to `0`.

Otherwise:

Reserved, RES0.

V, bit [30]

Status Register Valid.

`0b0` ERR<n>STATUS not valid.

`0b1` ERR<n>STATUS valid. At least one error has been recorded.

This bit is read/write-one-to-clear.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to `0`.

UE, bit [29]

Uncorrected Error.

`0b0` No errors have been detected, or all detected errors have been either corrected or deferred.

`0b1` At least one detected error was not corrected and not deferred.

When clearing ERR<n>STATUS.V to `0b0`, if this bit is nonzero, then software must write `0b1` to this bit to clear this bit to zero.

This bit is not valid and reads UNKNOWN if ERR<n>STATUS.V == `0b0`.

This bit is read/write-one-to-clear.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ER, bit [28]

Error Reported.

`0b0` No in-band error (External Abort) reported.

`0b1` An External Abort was signaled by the node to the master making the access or other transaction. This can be because any of the following are true:

- The applicable one of the ERR<n>CTLR.{WUE,RUE,UE} bits is implemented and was set to `0b1` when an Uncorrected error was detected.
- The applicable one of the ERR<n>CTLR.{WUE,RUE,UE} bits is not implemented and the node always reports errors.

It is IMPLEMENTATION DEFINED whether this bit can be set to `0b1` by a Deferred error.

When clearing ERR<n>STATUS.V to `0b0`, if this bit is nonzero, then software must write `0b1` to this bit to clear this bit to zero.

This bit is not valid and reads UNKNOWN if any of the following are true:

- ERR<n>STATUS.V == `0b0`.
- ERR<n>STATUS.UE == `0b0` and this bit is never set to `0b1` by a Deferred error.
- ERR<n>STATUS.{UE,DE} == {0,0} and this bit can be set to `0b1` by a Deferred error.

This bit is read/write-one-to-clear.

Note

An External Abort signaled by the node might be masked and not generate any exception.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

OF, bit [27]

Overflow.

Indicates that multiple errors have been detected. This bit is set to `0b1` when one of the following occurs:

- A Corrected error counter is implemented, an error is counted, and the counter overflows.
- ERR<n>STATUS.V was previously set to `0b1`, a Corrected error counter is not implemented, and a Corrected error is recorded.
- ERR<n>STATUS.V was previously set to `0b1`, and a type of error other than a Corrected error is recorded.

Otherwise, this bit is unchanged when an error is recorded.

If a Corrected error counter is implemented:

- A direct write that modifies the counter overflow flag indirectly might set this bit to an UNKNOWN value.
- A direct write to this bit that clears this bit to zero might indirectly set the counter overflow flag to an UNKNOWN value.

`0b0` Since this bit was last cleared to zero, no error syndrome has been discarded and, if a Corrected error counter is implemented, it has not overflowed.

`0b1` Since this bit was last cleared to zero, at least one error syndrome has been discarded or, if a Corrected error counter is implemented, it might have overflowed.

When clearing ERR<n>STATUS.V to `0b0`, if this bit is nonzero, then software must write `0b1` to this bit to clear this bit to zero.

This bit is not valid and reads UNKNOWN if ERR<n>STATUS.V == `0b0`.

This bit is read/write-one-to-clear.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

MV, bit [26]

When the error record includes an additional information for an error:

Miscellaneous Registers Valid.

`0b0` ERR<n>MISC<m> not valid.

`0b1` The IMPLEMENTATION DEFINED contents of the ERR<n>MISC<m> registers contains additional information for an error recorded by this record.

This bit is read/write-one-to-clear.

Note

If the ERR<n>MISC<m> registers can contain additional information for a previously recorded error, then the contents must be self-describing to software or a user. For example, certain fields might relate only to Corrected errors, and other fields only to the most recent error that was not discarded.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to `0`.

Otherwise:

Reserved, RES0.

CE, bits [25:24]

Corrected Error.

`0b00` No errors were corrected.

`0b01` At least one transient error was corrected.

`0b10` At least one error was corrected.

`0b11` At least one persistent error was corrected.

The mechanism by which a node detects whether a correctable error is transient or persistent is IMPLEMENTATION DEFINED. If no such mechanism is implemented, then the node sets this field to `0b10` when an error is corrected.

When clearing ERR<n>STATUS.V to `0b0`, if this field is nonzero, then software must write ones to this field to clear this field to zero.

This field is not valid and reads UNKNOWN if ERR<n>STATUS.V == `0b0`.

This field is read/write-ones-to-clear. Writing a value other than all-zeros or all-ones sets this field to an UNKNOWN value.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

DE, bit [23]

Deferred Error.

`0b0` No errors were deferred.

`0b1` At least one error was not corrected and deferred.

Support for deferring errors is IMPLEMENTATION DEFINED.

When clearing ERR<n>STATUS.V to `0b0`, if this bit is nonzero, then software must write `0b1` to this bit to clear this bit to zero.

This bit is not valid and reads UNKNOWN if ERR<n>STATUS.V == `0b0`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This bit is read/write-one-to-clear.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

PN, bit [22]

Poison.

`0b0` Uncorrected error or Deferred error recorded because a corrupt value was detected, for example, by an error detection code (EDC).

Note

If a producer node detects a corrupt value and defers the error by producing a poison value, then this bit is set to `0b0` at the producer node.

`0b1` Uncorrected error or Deferred error recorded because a poison value was detected.

Note

This might only be an indication of poison, because, in some EDC schemes, a poison value is encoded as an unlikely form of corrupt data, meaning it is possible to mistake a corrupt value as a poison value.

It is IMPLEMENTATION DEFINED whether a node can distinguish a poison value from a corrupt value.

When clearing ERR<n>STATUS.V to `0b0`, if this bit is nonzero, then software must write `0b1` to this bit to clear this bit to zero.

This bit is not valid and reads UNKNOWN if any of the following are true:

- ERR<n>STATUS.V == `0b0`.
- ERR<n>STATUS.{DE,UE} == {0,0}.

This bit is read/write-one-to-clear.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

UET, bits [21:20]

Uncorrected Error Type. Describes the state of the component after detecting or consuming an Uncorrected error.

`0b00` Uncorrected error, Uncontainable error (UC).

`0b01` Uncorrected error, Unrecoverable error (UEU).

`0b10` Uncorrected error, Latent or Restartable error (UEO).

`0b11` Uncorrected error, Signaled or Recoverable error (UER).

When clearing ERR<n>STATUS.V to `0b0`, if this field is nonzero, then software must write ones to this field to clear this field to zero.

This field is not valid and reads UNKNOWN if any of the following are true:

- ERR<n>STATUS.V == `0b0`.
- ERR<n>STATUS.UE == `0b0`.

This field is read/write-ones-to-clear. Writing a value other than all-zeros or all-ones sets this field to an UNKNOWN value.

Note

Software might use the information in the error record registers to determine what recovery is necessary.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

CI, bit [19]

Critical Error. Indicates whether a critical error condition has been recorded.

`0b0` No critical error condition.

`0b1` Critical error condition.

When clearing ERR<n>STATUS.V to `0b0`, if this bit is nonzero, then software must write `0b1` to this bit to clear this bit to zero.

This bit is not valid and reads UNKNOWN if ERR<n>STATUS.V == `0b0`.

This bit is read/write-one-to-clear.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [18:16]

Reserved, RES0.

IERR, bits [15:8]

IMPLEMENTATION DEFINED error code. Used with any primary error code SERR value. Further IMPLEMENTATION DEFINED information can be placed in the MISC registers.

This field is not valid and reads UNKNOWN if ERR<n>STATUS.V == `0b0`.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

SERR, bits [7:0]

Architecturally-defined primary error code. The primary error code might be used by a fault handling agent to triage an error without requiring device-specific code. For example, to count and threshold corrected errors in software, or generate a short log entry.

`0x00` No error.

`0x01` IMPLEMENTATION DEFINED error.

`0x02` Data value from (non-associative) internal memory. For example, ECC from on-chip SRAM or buffer.

`0x03` IMPLEMENTATION DEFINED pin. For example, **nSEI** pin.

`0x04` Assertion failure. For example, consistency failure.

`0x05` Error detected on internal data path. For example, parity on ALU result.

`0x06` Data value from associative memory. For example, ECC error on cache data.

`0x07` Address/control value from associative memory. For example, ECC error on cache tag.

`0x08` Data value from a TLB. For example, ECC error on TLB data.

`0x09` Address/control value from a TLB. For example, ECC error on TLB tag.

`0x0A` Data value from producer. For example, parity error on write data bus.

`0x0B` Address/control value from producer. For example, parity error on address bus.

`0x0C` Data value from (non-associative) external memory. For example, ECC error in SDRAM.

`0x0D` Illegal address (software fault). For example, access to unpopulated memory.

`0x0E` Illegal access (software fault). For example, byte write to word register.

`0x0F` Illegal state (software fault). For example, device not ready.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0x10` Internal data register. For example, parity on a SIMD&FP register. For a PE, all general-purpose, stack pointer, SIMD&FP, and SVE registers are data registers.

`0x11` Internal control register. For example, Parity on a System register. For a PE, all registers other than general-purpose, stack pointer, SIMD&FP, and SVE registers are control registers.

`0x12` Error response from slave. For example, error response from cache write-back.

`0x13` External timeout. For example, timeout on interaction with another node.

`0x14` Internal timeout. For example, timeout on interface within the node.

`0x15` Deferred error from slave not supported at master. For example, poisoned data received from a slave by a master that cannot defer the error further.

`0x16` Deferred error from master not supported at slave. For example, poisoned data received from a master by a slave that cannot defer the error further.

`0x17` Deferred error from slave passed through. For example, poisoned data received from a slave and returned to a master.

`0x18` Deferred error from master passed through. For example, poisoned data received from a master and deferred to a slave.

`0x19` Error recorded by PCIe error logs. Indicates that the node has recorded an error in a PCIe error log. This might be the PCIe device status register, AER, DVSEC, or other mechanisms defined by PCIe.

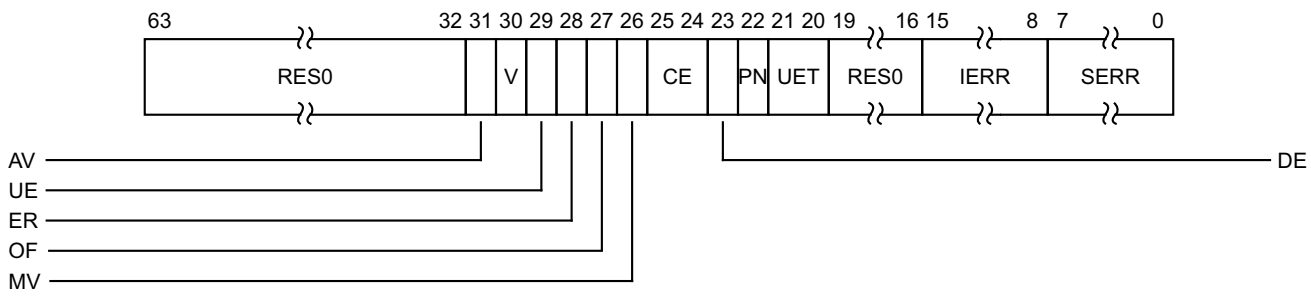
All other values are reserved.

This field is not valid and reads UNKNOWN if ERR<n>STATUS.V == `0b0`.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

When RAS System Architecture v1.0 is implemented:



Bits [63:32]

Reserved, RES0.

AV, bit [31]

When the error record includes an address associated with an error:

Address Valid.

`0b0` ERR<n>ADDR not valid.

`0b1` ERR<n>ADDR contains an address associated with the highest priority error recorded by this record.

This bit ignores writes if ERR<n>STATUS.{CE,DE,UE} != {`0b00`,0,0}, and the highest priority of these is not being cleared to zero in the same write.

This bit is read/write-one-to-clear.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to `0`.

Otherwise:

Reserved, RES0.

V, bit [30]

Status Register Valid.

`0b0` ERR<n>STATUS not valid.

`0b1` ERR<n>STATUS valid. At least one error has been recorded.

This bit ignores writes if ERR<n>STATUS.{CE,DE,UE} != {`0b00`,0,0}, and is not being cleared to `0b0` in the same write.

This bit is read/write-one-to-clear.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to `0`.

UE, bit [29]

Uncorrected Error.

`0b0` No errors have been detected, or all detected errors have been either corrected or deferred.

`0b1` At least one detected error was not corrected and not deferred.

When clearing ERR<n>STATUS.V to `0b0`, if this bit is nonzero, then software must write `0b1` to this bit to clear this bit to zero.

This bit is not valid and reads UNKNOWN if ERR<n>STATUS.V == `0b0`. This bit ignores writes if ERR<n>STATUS.OF == `0b1` and is not being cleared to `0b0` in the same write.

This bit is read/write-one-to-clear.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

ER, bit [28]

Error Reported.

`0b0` No in-band error (External Abort) reported.

`0b1` An External Abort was signaled by the node to the master making the access or other transaction. This can be because any of the following are true:

- The applicable one of the ERR<n>CTLR.{WUE,RUE,UE} bits is implemented and was set to `0b1` when an Uncorrected error was detected.
- The applicable one of the ERR<n>CTLR.{WUE,RUE,UE} bits is not implemented and the node always reports errors.

It is IMPLEMENTATION DEFINED whether this bit can be set to `0b1` by a Deferred error.

If this bit is nonzero, then software must write `0b1` to this bit to clear this bit to zero, when any of:

- Clearing ERR<n>STATUS.V to `0b0`.
- Clearing ERR<n>STATUS.UE to `0b0`, if this bit is never set to `0b1` by a Deferred error.
- Clearing ERR<n>STATUS.{UE,DE} to {0,0}, if this bit can be set to `0b1` by a Deferred error.

This bit is not valid and reads UNKNOWN if any of the following are true:

- ERR<n>STATUS.V == `0b0`.
- ERR<n>STATUS.UE == `0b0` and this bit is never set to `0b1` by a Deferred error.
- ERR<n>STATUS.{UE,DE} == {0,0} and this bit can be set to `0b1` by a Deferred error.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This bit ignores writes if ERR<n>STATUS.{CE,DE,UE} != {0b00,0,0}, and the highest priority of these is not being cleared to zero in the same write.

This bit is read/write-one-to-clear.

Note

An External Abort signaled by the node might be masked and not generate any exception.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

OF, bit [27]

Overflow.

Indicates that multiple errors have been detected. This bit is set to 0b1 when one of the following occurs:

- An Uncorrected error is detected and ERR<n>STATUS.UE == 0b1.
- A Deferred error is detected, ERR<n>STATUS.UE == 0b0 and ERR<n>STATUS.DE == 0b1.
- A Corrected error is detected, no Corrected error counter is implemented, ERR<n>STATUS.UE == 0b0, ERR<n>STATUS.DE == 0b0, and ERR<n>STATUS.CE != 0b00. ERR<n>STATUS.CE might be updated for the new Corrected error.
- A Corrected error counter is implemented, ERR<n>STATUS.UE == 0b0, ERR<n>STATUS.DE == 0b0, and the counter overflows.

It is IMPLEMENTATION DEFINED whether this bit is set to 0b1 when one of the following occurs:

- A Deferred error is detected and ERR<n>STATUS.UE == 0b1.
- A Corrected error is detected, no Corrected error counter is implemented, and either or both the ERR<n>STATUS.UE or ERR<n>STATUS.DE bits are set to 0b1.
- A Corrected error counter is implemented, either or both the ERR<n>STATUS.UE or ERR<n>STATUS.DE bits are set to 0b1, and the counter overflows.

It is IMPLEMENTATION DEFINED whether this bit is cleared to 0b0 when one of the following occurs:

- An Uncorrected error is detected and ERR<n>STATUS.UE == 0b0.
- A Deferred error is detected, ERR<n>STATUS.UE == 0b0 and ERR<n>STATUS.DE == 0b0.
- A Corrected error is detected, ERR<n>STATUS.UE == 0b0, ERR<n>STATUS.DE == 0b0 and ERR<n>STATUS.CE == 0b00.

The IMPLEMENTATION DEFINED clearing of this bit might also depend on the value of the other error status bits.

If a Corrected error counter is implemented:

- A direct write that modifies the counter overflow flag indirectly might set this bit to an UNKNOWN value.
- A direct write to this bit that clears this bit to 0b0 might indirectly set the counter overflow flag to an UNKNOWN value.

0b0 If ERR<n>STATUS.UE == 0b1, then no error syndrome for an Uncorrected error has been discarded.

If ERR<n>STATUS.UE == 0b0 and ERR<n>STATUS.DE == 0b1, then no error syndrome for a Deferred error has been discarded.

If ERR<n>STATUS.UE == 0b0, ERR<n>STATUS.DE == 0b0, and a Corrected error counter is implemented, then the counter has not overflowed.

If ERR<n>STATUS.UE == 0b0, ERR<n>STATUS.DE == 0b0, ERR<n>STATUS.CE != 0b00, and no Corrected error counter is implemented, then no error syndrome for a Corrected error has been discarded.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

This bit might have been set to `0b1` when an error syndrome was discarded and later cleared to `0b0` when a higher priority syndrome was recorded.

`0b1` At least one error syndrome has been discarded or, if a Corrected error counter is implemented, it might have overflowed.

When clearing ERR<n>STATUS.V to `0b0`, if this bit is nonzero, then software must write `0b1` to this bit to clear this bit to zero.

This bit is not valid and reads UNKNOWN if ERR<n>STATUS.V == `0b0`.

This bit is read/write-one-to-clear.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

MV, bit [26]

When the error record includes an additional information for an error:

Miscellaneous Registers Valid.

`0b0` ERR<n>MISC<m> not valid.

`0b1` The IMPLEMENTATION DEFINED contents of the ERR<n>MISC<m> registers contains additional information for an error recorded by this record.

This bit ignores writes if ERR<n>STATUS.{CE,DE,UE} != {`0b00`,0,0}, and the highest priority of these is not being cleared to zero in the same write.

This bit is read/write-one-to-clear.

Note

If the ERR<n>MISC<m> registers can contain additional information for a previously recorded error, then the contents must be self-describing to software or a user. For example, certain fields might relate only to Corrected errors, and other fields only to the most recent error that was not discarded.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to `0`.

Otherwise:

Reserved, RES0.

CE, bits [25:24]

Corrected Error.

`0b00` No errors were corrected.

`0b01` At least one transient error was corrected.

`0b10` At least one error was corrected.

`0b11` At least one persistent error was corrected.

The mechanism by which a node detects whether a correctable error is transient or persistent is IMPLEMENTATION DEFINED. If no such mechanism is implemented, then the node sets this field to `0b10` when an error is corrected.

When clearing ERR<n>STATUS.V to `0b0`, if this field is nonzero, then software must write ones to this field to clear this field to zero.

This field is not valid and reads UNKNOWN if ERR<n>STATUS.V == `0b0`. This field ignores writes if ERR<n>STATUS.OF == `0b1` and is not being cleared to `0b0` in the same write.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field is read/write-ones-to-clear. Writing a value other than all-zeros or all-ones sets this field to an UNKNOWN value.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

DE, bit [23]

Deferred Error.

`0b0` No errors were deferred.

`0b1` At least one error was not corrected and deferred.

Support for deferring errors is IMPLEMENTATION DEFINED.

When clearing ERR<n>STATUS.V to `0b0`, if this bit is nonzero, then software must write `0b1` to this bit to clear this bit to zero.

This bit is not valid and reads UNKNOWN if ERR<n>STATUS.V == `0b0`. This bit ignores writes if ERR<n>STATUS.OF == `0b1` and is not being cleared to `0b0` in the same write.

This bit is read/write-one-to-clear.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

PN, bit [22]

Poison.

`0b0` Uncorrected error or Deferred error recorded because a corrupt value was detected, for example, by an error detection code (EDC).

Note

If a producer node detects a corrupt value and defers the error by producing a poison value, then this bit is set to `0b0` at the producer node.

`0b1` Uncorrected error or Deferred error recorded because a poison value was detected.

Note

This might only be an indication of poison, because, in some EDC schemes, a poison value is encoded as an unlikely form of corrupt data, meaning it is possible to mistake a corrupt value as a poison value.

It is IMPLEMENTATION DEFINED whether a node can distinguish a poison value from a corrupt value.

If this bit is nonzero, then software must write `0b1` to this bit to clear this bit to zero, when any of:

- Clearing ERR<n>STATUS.V to `0b0`.
- Clearing both ERR<n>STATUS.{DE, UE} to `0b0`.

This bit is not valid and reads UNKNOWN if any of the following are true:

- ERR<n>STATUS.V == `0b0`.
- ERR<n>STATUS.{DE,UE} == {0,0}.

This bit ignores writes if ERR<n>STATUS.{CE,DE,UE} != {`0b00`,0,0}, and the highest priority of these is not being cleared to zero in the same write.

This bit is read/write-one-to-clear.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

UET, bits [21:20]

Uncorrected Error Type. Describes the state of the component after detecting or consuming an Uncorrected error.

`0b00` Uncorrected error, Uncontainable error (UC).

`0b01` Uncorrected error, Unrecoverable error (UEU).

`0b10` Uncorrected error, Latent or Restartable error (UEO).

`0b11` Uncorrected error, Signaled or Recoverable error (UER).

If this field is nonzero, then software must write ones to this field to clear this field to zero, when any of:

- Clearing ERR<n>STATUS.V to `0b0`.
- Clearing ERR<n>STATUS.UE to `0b0`.

This field is not valid and reads UNKNOWN if any of the following are true:

- ERR<n>STATUS.V == `0b0`.
- ERR<n>STATUS.UE == `0b0`.

This field ignores writes if ERR<n>STATUS.{CE,DE,UE} != {`0b00`,0,0}, and the highest priority of these is not being cleared to zero in the same write.

This field is read/write-ones-to-clear. Writing a value other than all-zeros or all-ones sets this field to an UNKNOWN value.

Note

Software might use the information in the error record registers to determine what recovery is necessary.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [19:16]

Reserved, RES0.

IERR, bits [15:8]

IMPLEMENTATION DEFINED error code. Used with any primary error code SERR value. Further IMPLEMENTATION DEFINED information can be placed in the MISC registers.

This field is not valid and reads UNKNOWN if ERR<n>STATUS.V == `0b0`. This field ignores writes if ERR<n>STATUS.{CE,DE,UE} != {`0b00`,0,0}, and the highest priority of these is not being cleared to zero in the same write.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

SERR, bits [7:0]

Architecturally-defined primary error code. The primary error code might be used by a fault handling agent to triage an error without requiring device-specific code. For example, to count and threshold corrected errors in software, or generate a short log entry.

`0x00` No error.

`0x01` IMPLEMENTATION DEFINED error.

`0x02` Data value from (non-associative) internal memory. For example, ECC from on-chip SRAM or buffer.

`0x03` IMPLEMENTATION DEFINED pin. For example, **nSEI** pin.

`0x04` Assertion failure. For example, consistency failure.

`0x05` Error detected on internal data path. For example, parity on ALU result.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0x06` Data value from associative memory. For example, ECC error on cache data.

`0x07` Address/control value from associative memory. For example, ECC error on cache tag.

`0x08` Data value from a TLB. For example, ECC error on TLB data.

`0x09` Address/control value from a TLB. For example, ECC error on TLB tag.

`0x0A` Data value from producer. For example, parity error on write data bus.

`0x0B` Address/control value from producer. For example, parity error on address bus.

`0x0C` Data value from (non-associative) external memory. For example, ECC error in SDRAM.

`0x0D` Illegal address (software fault). For example, access to unpopulated memory.

`0x0E` Illegal access (software fault). For example, byte write to word register.

`0x0F` Illegal state (software fault). For example, device not ready.

`0x10` Internal data register. For example, parity on a SIMD&FP register. For a PE, all general-purpose, stack pointer, SIMD&FP, and SVE registers are data registers.

`0x11` Internal control register. For example, Parity on a System register. For a PE, all registers other than general-purpose, stack pointer, SIMD&FP, and SVE registers are control registers.

`0x12` Error response from slave. For example, error response from cache write-back.

`0x13` External timeout. For example, timeout on interaction with another node.

`0x14` Internal timeout. For example, timeout on interface within the node.

`0x15` Deferred error from slave not supported at master. For example, poisoned data received from a slave by a master that cannot defer the error further.

`0x16` Deferred error from master not supported at slave. For example, poisoned data received from a master by a slave that cannot defer the error further.

`0x17` Deferred error from slave passed through. For example, poisoned data received from a slave and returned to a master.

`0x18` Deferred error from master passed through. For example, poisoned data received from a master and deferred to a slave.

`0x19` Error recorded by PCIe error logs. Indicates that the node has recorded an error in a PCIe error log. This might be the PCIe device status register, AER, DVSEC, or other mechanisms defined by PCIe.

All other values are reserved.

This field is not valid and reads UNKNOWN if ERR<n>STATUS.V == `0b0`. This field ignores writes if ERR<n>STATUS.{CE,DE,UE} != {`0b00`,0,0}, and the highest priority of these is not being cleared to zero in the same write.

The following resets apply:

- On a Error recovery reset, the value of this field is unchanged.
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing the ERR<n>STATUS:

The {AV, V, UE, ER, OF, MV, CE, DE, PN, UET, CI} fields are write-one-to-clear, meaning writes of zero are ignored, and a write of one or all-ones to the field clears the field to zero. The {IERR, SERR} fields are read/write fields, although the set of permitted values that can be written to the fields is IMPLEMENTATION DEFINED.

After reading ERR<n>STATUS, software must clear the valid bits in the register to allow new errors to be recorded. However, between reading the register and clearing the valid bits, a new error might have overwritten the register. To prevent this error being lost by software, the register prevents updates to fields that might have been updated by a new error.

When RAS System Architecture v1.0 is implemented:

- Writes to the {UE, DE, CE} fields are ignored if the OF bit is set and is not being cleared.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Writes to the V bit are ignored if any of the {UE, DE, CE} fields are nonzero and are not being cleared.
- Writes to the {AV, MV} bits and {ER, PN, UET, IERR, SERR} syndrome fields are ignored if the highest priority error status field is nonzero and not being cleared. The error status fields in priority order from highest to lowest, are UE, DE, and CE.

When RAS System Architecture v1.1 is implemented, a write to the register is ignored if all of:

- Any of {V, UE, OF, CE, DE} fields are nonzero before the write.
- The write does not clear the nonzero {V, UE, OF, CE, DE} fields to zero by writing ones to the applicable field or fields.

To ensure correct and portable operation, when software is clearing the valid bits in the register to allow new errors to be recorded, software must:

- Determine which fields must be cleared to zero by reading ERR<n>STATUS.
- Write ones to all the write-one-to-clear fields that are nonzero.
- Write zero to all the read/write fields.
- Write zero to all the write-one-to-clear fields that are zero.

ERR<n>STATUS can be accessed through its memory-mapped interface:

Component	Offset	Instance
RAS	`0x010` + 64n	ERR<n>STATUS

This interface is accessible as follows:

- Accesses to this register are RW.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.32 ERRPIDR0, Peripheral Identification Register 0

The ERRPIDR0 characteristics are:

Purpose

Provides discovery information about the component.

For more information, see *About the Peripheral identification scheme* on page K2-7889.

Configurations

Implementation of this register is OPTIONAL.

Attributes

ERRPIDR0 is a 32-bit register.

Field descriptions

The ERRPIDR0 bit assignments are:

31 … 8	7 … 0
RES0	PART_0

Bits [31:8]

Reserved, RES0.

PART_0, bits [7:0]

IMPLEMENTATION DEFINED.

Part number, bits [7:0].

The part number is selected by the designer of the component. The designer chooses whether to use a 12-bit or a 16-bit part number:

- If a 12-bit part number is used, it is stored in ERRPIDR1.PART_1 and ERRPIDR0.PART_0. There are 8 bits, ERRPIDR2.REVISION and ERRPIDR3.REVAND, available to define the revision of the component.
- If a 16-bit part number is used, it is stored in ERRPIDR2.PART_2, ERRPIDR1.PART_1 and ERRPIDR0.PART_0. There are 4 bits, ERRPIDR3.REVISION, available to define the revision of the component.

Accessing the ERRPIDR0:

ERRPIDR0 can be accessed through its memory-mapped interface:

Component	Offset
RAS	`0xFE0`

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.33 ERRPIDR1, Peripheral Identification Register 1

The ERRPIDR1 characteristics are:

Purpose

Provides discovery information about the component.

For more information, see *About the Peripheral identification scheme* on page K2-7889.

Configurations

Implementation of this register is OPTIONAL.

Attributes

ERRPIDR1 is a 32-bit register.

Field descriptions

The ERRPIDR1 bit assignments are:

31 … 8	7 … 4	3 … 0
RES0	DES_0	PART_1

Bits [31:8]

Reserved, RES0.

DES_0, bits [7:4]

IMPLEMENTATION DEFINED.

Designer, JEP106 identification code, bits [3:0]. ERRPIDR1.DES_0 and ERRPIDR2.DES_1 together form the JEDEC-assigned JEP106 identification code for the designer of the component. The parity bit in the JEP106 identification code is not included. The code identifies the designer of the component, which might not be not the same as the implementer of the device containing the component. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org.

Note

For a component designed by Arm Limited, the JEP106 identification code is `0x3B`.

PART_1, bits [3:0]

IMPLEMENTATION DEFINED.

Part number, bits [11:8].

The part number is selected by the designer of the component. The designer chooses whether to use a 12-bit or a 16-bit part number:

- If a 12-bit part number is used, it is stored in ERRPIDR1.PART_1 and ERRPIDR0.PART_0. There are 8 bits, ERRPIDR2.REVISION and ERRPIDR3.REVAND, available to define the revision of the component.
- If a 16-bit part number is used, it is stored in ERRPIDR2.PART_2, ERRPIDR1.PART_1 and ERRPIDR0.PART_0. There are 4 bits, ERRPIDR3.REVISION, available to define the revision of the component.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the ERRPIDR1:

ERRPIDR1 can be accessed through its memory-mapped interface:

Component	Offset
RAS	`0xFE4`

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.34 ERRPIDR2, Peripheral Identification Register 2

The ERRPIDR2 characteristics are:

Purpose

Provides discovery information about the component.

For more information, see *About the Peripheral identification scheme* on page K2-7889.

Configurations

Implementation of this register is OPTIONAL.

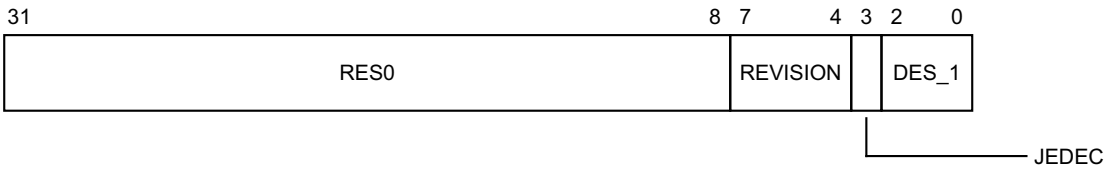
Attributes

ERRPIDR2 is a 32-bit register.

Field descriptions

The ERRPIDR2 bit assignments are:

When the component uses a 12-bit part number:



Bits [31:8]

Reserved, RES0.

REVISION, bits [7:4]

IMPLEMENTATION DEFINED.

Component major revision. ERRPIDR2.REVISION and ERRPIDR3.REVAND together form the revision number of the component, with ERRPIDR2.REVISION being the most significant part and ERRPIDR3.REVAND the least significant part. When a component is changed, ERRPIDR2.REVISION or ERRPIDR3.REVAND must be increased to ensure that software can differentiate the different revisions of the component. If ERRPIDR2.REVISION is increased then ERRPIDR3.REVAND should be set to `0b0000`.

JEDEC, bit [3]

JEDEC-assigned JEP106 implementer code is used.

This bit reads as one.

DES_1, bits [2:0]

IMPLEMENTATION DEFINED.

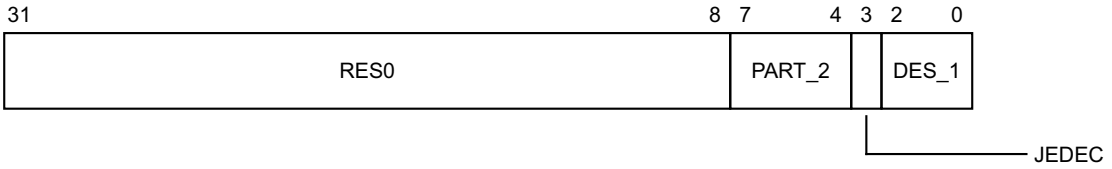
Designer, JEP106 identification code, bits [6:4]. ERRPIDR1.DES_0 and ERRPIDR2.DES_1 together form the JEDEC-assigned JEP106 identification code for the designer of the component. The parity bit in the JEP106 identification code is not included. The code identifies the designer of the component, which might not be not the same as the implementer of the device containing the component. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org.

Note

For a component designed by Arm Limited, the JEP106 identification code is `0x3B`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When the component uses a 16-bit part number:



Bits [31:8]

Reserved, RES0.

PART_2, bits [7:4]

IMPLEMENTATION DEFINED.

Part number, bits [15:12].

The part number is selected by the designer of the component. The designer chooses whether to use a 12-bit or a 16-bit part number:

- If a 12-bit part number is used, it is stored in ERRPIDR1.PART_1 and ERRPIDR0.PART_0. There are 8 bits, ERRPIDR2.REVISION and ERRPIDR3.REVAND, available to define the revision of the component.
- If a 16-bit part number is used, it is stored in ERRPIDR2.PART_2, ERRPIDR1.PART_1 and ERRPIDR0.PART_0. There are 4 bits, ERRPIDR3.REVISION, available to define the revision of the component.

JEDEC, bit [3]

JEDEC-assigned JEP106 implementer code is used.

This bit reads as one.

DES_1, bits [2:0]

IMPLEMENTATION DEFINED.

Designer, JEP106 identification code, bits [6:4]. ERRPIDR1.DES_0 and ERRPIDR2.DES_1 together form the JEDEC-assigned JEP106 identification code for the designer of the component. The parity bit in the JEP106 identification code is not included. The code identifies the designer of the component, which might not be not the same as the implementer of the device containing the component. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org.

Note

For a component designed by Arm Limited, the JEP106 identification code is `0x3B`.

Accessing the ERRPIDR2:

ERRPIDR2 can be accessed through its memory-mapped interface:

Component	Offset
RAS	`0xFE8`

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.35 ERRPIDR3, Peripheral Identification Register 3

The ERRPIDR3 characteristics are:

Purpose

Provides discovery information about the component.

For more information, see *About the Peripheral identification scheme* on page K2-7889.

Configurations

Implementation of this register is OPTIONAL.

Attributes

ERRPIDR3 is a 32-bit register.

Field descriptions

The ERRPIDR3 bit assignments are:

When the component uses a 12-bit part number:

31 … 8	7 … 4	3 … 0
RES0	REVAND	CMOD

Bits [31:8]

Reserved, RES0.

REVAND, bits [7:4]

IMPLEMENTATION DEFINED.

Component minor revision. ERRPIDR2.REVISION and ERRPIDR3.REVAND together form the revision number of the component, with ERRPIDR2.REVISION being the most significant part and ERRPIDR3.REVAND the least significant part. When a component is changed, ERRPIDR2.REVISION or ERRPIDR3.REVAND must be increased to ensure that software can differentiate the different revisions of the component. If ERRPIDR2.REVISION is increased then ERRPIDR3.REVAND should be set to `0b0000`.

CMOD, bits [3:0]

IMPLEMENTATION DEFINED.

Customer Modified.

Indicates the component has been modified.

A value of `0b0000` means the component is not modified from the original design.

Any other value means the component has been modified in an IMPLEMENTATION DEFINED way.

For any two components with the same Unique Component Identifier:

- If the value of the CMOD fields of both components equals zero, the components are identical.
- If the CMOD fields of both components have the same non-zero value, it does not necessarily mean that they have the same modifications.
- If the value of the CMOD field of either of the two components is non-zero, they might not be identical, even though they have the same Unique Component Identifier.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When the component uses a 16-bit part number:

31 … 8	7 … 4	3 … 0
RES0	REVISION	CMOD

Bits [31:8]

Reserved, RES0.

REVISION, bits [7:4]

IMPLEMENTATION DEFINED.

Component revision. When a component is changed, ERRPIDR3.REVISION must be increased to ensure that software can differentiate the different revisions of the component.

CMOD, bits [3:0]

IMPLEMENTATION DEFINED.

Customer Modified.

Indicates the component has been modified.

A value of `0b0000` means the component is not modified from the original design.

Any other value means the component has been modified in an IMPLEMENTATION DEFINED way.

For any two components with the same Unique Component Identifier:

- If the value of the CMOD fields of both components equals zero, the components are identical.
- If the CMOD fields of both components have the same non-zero value, it does not necessarily mean that they have the same modifications.
- If the value of the CMOD field of either of the two components is non-zero, they might not be identical, even though they have the same Unique Component Identifier.

Accessing the ERRPIDR3:

ERRPIDR3 can be accessed through its memory-mapped interface:

Component	Offset
RAS	`0xFEC`

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

I5.8.36 ERRPIDR4, Peripheral Identification Register 4

The ERRPIDR4 characteristics are:

Purpose

Provides discovery information about the component.

For more information, see *About the Peripheral identification scheme* on page K2-7889.

Configurations

Implementation of this register is OPTIONAL.

Attributes

ERRPIDR4 is a 32-bit register.

Field descriptions

The ERRPIDR4 bit assignments are:

31 … 8	7 … 4	3 … 0
RES0	SIZE	DES_2

Bits [31:8]

Reserved, RES0.

SIZE, bits [7:4]

IMPLEMENTATION DEFINED.

Size of the component.

The distance from the start of the address space used by this component to the end of the component identification registers.

A value of `0b0000` means one of the following is true:

- The component uses a single 4KB block.
- The component uses an IMPLEMENTATION DEFINED number of 4KB blocks.

Any other value means the component occupies $2^{\text{ERRPIDR4.SIZE}}$ 4KB blocks.

Using this field to indicate the size of the component is deprecated. This field might not correctly indicate the size of the component. Arm recommends that software determine the size of the component from the Unique Component Identifier fields, and other IMPLEMENTATION DEFINED registers in the component.

DES_2, bits [3:0]

IMPLEMENTATION DEFINED.

Designer, JEP106 continuation code. This is the JEDEC-assigned JEP106 bank identifier for the designer of the component, minus 1. The code identifies the designer of the component, which might not be not the same as the implementer of the device containing the component. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org.

Note

For a component designed by Arm Limited, the JEP106 bank is 5, meaning this field has the value `0x4`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accessing the ERRPIDR4:

ERRPIDR4 can be accessed through its memory-mapped interface:

Component	Offset
RAS	`0xFD0`

This interface is accessible as follows:

- Accesses to this register are RO.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Part J

Architectural Pseudocode

Chapter J1
Armv8 Pseudocode

This chapter contains pseudocode that describes many features of the Armv8 architecture. It contains the following sections:

- *Pseudocode for AArch64 operation* on page J1-7508.
- *Pseudocode for AArch32 operation* on page J1-7627.
- *Shared pseudocode* on page J1-7705.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

J1.1 Pseudocode for AArch64 operation

This section holds the pseudocode for execution in AArch64 state. Functions that are listed in this section are identified as `AArch64.FunctionName`. Some of these functions have an equivalent AArch32 function, `AArch32.FunctionName`. This section is organized by functional groups, with the functional groups being indicated by hierarchical path names, for example `aarch64/debug/breakpoint`.

The top-level sections of the AArch64 pseudocode hierarchy are:

- *aarch64/debug*.
- *aarch64/exceptions* on page J1-7518.
- *aarch64/functions* on page J1-7539.
- *aarch64/instrs* on page J1-7597.
- *aarch64/translation* on page J1-7604.

J1.1.1 aarch64/debug

This section includes the following pseudocode functions:

- *aarch64/debug/breakpoint/AArch64.BreakpointMatch*.
- *aarch64/debug/breakpoint/AArch64.BreakpointValueMatch* on page J1-7509.
- *aarch64/debug/breakpoint/AArch64.StateMatch* on page J1-7510.
- *aarch64/debug/enables/AArch64.GenerateDebugExceptions* on page J1-7511.
- *aarch64/debug/enables/AArch64.GenerateDebugExceptionsFrom* on page J1-7511.
- *aarch64/debug/pmu/AArch64.CheckForPMUOverflow* on page J1-7512.
- *aarch64/debug/pmu/AArch64.CountEvents* on page J1-7512.
- *aarch64/debug/statisticalprofiling/CheckProfilingBufferAccess* on page J1-7513.
- *aarch64/debug/statisticalprofiling/CheckStatisticalProfilingAccess* on page J1-7513.
- *aarch64/debug/statisticalprofiling/CollectContextIDR1* on page J1-7514.
- *aarch64/debug/statisticalprofiling/CollectContextIDR2* on page J1-7514.
- *aarch64/debug/statisticalprofiling/CollectPhysicalAddress* on page J1-7514.
- *aarch64/debug/statisticalprofiling/CollectRecord* on page J1-7514.
- *aarch64/debug/statisticalprofiling/CollectTimeStamp* on page J1-7515.
- *aarch64/debug/statisticalprofiling/OpType* on page J1-7515.
- *aarch64/debug/statisticalprofiling/ProfilingBufferEnabled* on page J1-7515.
- *aarch64/debug/statisticalprofiling/ProfilingBufferOwner* on page J1-7515.
- *aarch64/debug/statisticalprofiling/ProfilingSynchronizationBarrier* on page J1-7516.
- *aarch64/debug/statisticalprofiling/StatisticalProfilingEnabled* on page J1-7516.
- *aarch64/debug/statisticalprofiling/SysRegAccess* on page J1-7516.
- *aarch64/debug/statisticalprofiling/TimeStamp* on page J1-7516.
- *aarch64/debug/takeexceptiondbg/AArch64.TakeExceptionInDebugState* on page J1-7516.
- *aarch64/debug/watchpoint/AArch64.WatchpointByteMatch* on page J1-7517.
- *aarch64/debug/watchpoint/AArch64.WatchpointMatch* on page J1-7518.

aarch64/debug/breakpoint/AArch64.BreakpointMatch

```
// AArch64.BreakpointMatch()
// =========================
// Breakpoint matching in an AArch64 translation regime.

boolean AArch64.BreakpointMatch(integer n, bits(64) vaddress, AccType acctype, integer size)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert n <= UInt(ID_AA64DFR0_EL1.BRPs);

    enabled = DBGBCR_EL1[n].E == '1';
    ispriv = PSTATE.EL != EL0;
    linked = DBGBCR_EL1[n].BT == '0x01';
    isbreakpnt = TRUE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    linked_to = FALSE;

    state_match = AArch64.StateMatch(DBGBCR_EL1[n].SSC, DBGBCR_EL1[n].HMC, DBGBCR_EL1[n].PMC,
                                     linked, DBGBCR_EL1[n].LBN, isbreakpnt, acctype, ispriv);
    value_match = AArch64.BreakpointValueMatch(n, vaddress, linked_to);

    if HaveAnyAArch32() && size == 4 then                 // Check second halfword
        // If the breakpoint address and BAS of an Address breakpoint match the address of the
        // second halfword of an instruction, but not the address of the first halfword, it is
        // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
        // event.
        match_i = AArch64.BreakpointValueMatch(n, vaddress + 2, linked_to);
        if !value_match && match_i then
            value_match = ConstrainUnpredictableBool();
    if vaddress<1> == '1' && DBGBCR_EL1[n].BAS == '1111' then
        // The above notwithstanding, if DBGBCR_EL1[n].BAS == '1111', then it is CONSTRAINED
        // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
        // at the address DBGBVR_EL1[n]+2.
        if value_match then value_match = ConstrainUnpredictableBool();

    match = value_match && state_match && enabled;

    return match;
```

aarch64/debug/breakpoint/AArch64.BreakpointValueMatch

```
// AArch64.BreakpointValueMatch()
// ==============================

boolean AArch64.BreakpointValueMatch(integer n, bits(64) vaddress, boolean linked_to)

    // "n" is the identity of the breakpoint unit to match against.
    // "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
    //   matching breakpoints.
    // "linked_to" is TRUE if this is a call from StateMatch for linking.

    // If a non-existent breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
    // no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
    if n > UInt(ID_AA64DFR0_EL1.BRPs) then
        (c, n) = ConstrainUnpredictableInteger(0, UInt(ID_AA64DFR0_EL1.BRPs));
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return FALSE;

    // If this breakpoint is not enabled, it cannot generate a match. (This could also happen on a
    // call from StateMatch for linking).
    if DBGBCR_EL1[n].E == '0' then return FALSE;

    context_aware = (n >= UInt(ID_AA64DFR0_EL1.BRPs) - UInt(ID_AA64DFR0_EL1.CTX_CMPs));

    // If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
    dbgtype = DBGBCR_EL1[n].BT;

    if ((dbgtype IN {'011x','11xx'} && !HaveVirtHostExt()) ||              // Context matching
          dbgtype == '010x' ||                                             // Reserved
          (dbgtype != '0x0x' && !context_aware) ||                         // Context matching
          (dbgtype == '1xxx' && !HaveEL(EL2))) then                        // EL2 extension
        (c, dbgtype) = ConstrainUnpredictableBits();
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return FALSE;
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    // Determine what to compare against.
    match_addr = (dbgtype == '0x0x');
    match_vmid = (dbgtype == '10xx');
    match_cid  = (dbgtype == '001x');
    match_cid1 = (dbgtype IN { '101x', 'x11x'});
    match_cid2 = (dbgtype == '11xx');
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    linked     = (dbgtype == 'xxx1');

    // If this is a call from StateMatch, return FALSE if the breakpoint is not programmed for a
    // VMID and/or context ID match, of if not context-aware. The above assertions mean that the
    // code can just test for match_addr == TRUE to confirm all these things.
    if linked_to && (!linked || match_addr) then return FALSE;

    // If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
    if !linked_to && linked && !match_addr then return FALSE;

    // Do the comparison.
    if match_addr then
        byte = UInt(vaddress<1:0>);
        if HaveAnyAArch32() then
            // T32 instructions can be executed at EL0 in an AArch64 translation regime.
            assert byte IN {0,2};                // "vaddress" is halfword aligned
            byte_select_match = (DBGBCR_EL1[n].BAS<byte> == '1');
        else
            assert byte == 0;                    // "vaddress" is word aligned
            byte_select_match = TRUE;            // DBGBCR_EL1[n].BAS<byte> is RES1
        top = AddrTop(vaddress, TRUE, PSTATE.EL);
        BVR_match = vaddress<top:2> == DBGBVR_EL1[n]<top:2> && byte_select_match;
    elsif match_cid then
        if IsInHost() then
            BVR_match = (CONTEXTIDR_EL2 == DBGBVR_EL1[n]<31:0>);
        else
            BVR_match = (PSTATE.EL IN {EL0, EL1} && CONTEXTIDR_EL1 == DBGBVR_EL1[n]<31:0>);
    elsif match_cid1 then
        BVR_match = (PSTATE.EL IN {EL0, EL1} && !IsInHost() && CONTEXTIDR_EL1 == DBGBVR_EL1[n]<31:0>);
    if match_vmid then
        if !Have16bitVMID() || VTCR_EL2.VS == '0' then
            vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
            bvr_vmid = ZeroExtend(DBGBVR_EL1[n]<39:32>, 16);
        else
            vmid = VTTBR_EL2.VMID;
            bvr_vmid = DBGBVR_EL1[n]<47:32>;
        BXVR_match = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                      !IsInHost() &&
                      vmid == bvr_vmid);
    elsif match_cid2 then
        BXVR_match = (!IsSecure() && HaveVirtHostExt() &&
                      DBGBVR_EL1[n]<63:32> == CONTEXTIDR_EL2);

    bvr_match_valid = (match_addr || match_cid || match_cid1);
    bxvr_match_valid = (match_vmid || match_cid2);

    match = (!bxvr_match_valid || BXVR_match) && (!bvr_match_valid || BVR_match);

    return match;
```

aarch64/debug/breakpoint/AArch64.StateMatch

```
// AArch64.StateMatch()
// ====================
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch64.StateMatch(bits(2) SSC, bit HMC, bits(2) PxC, boolean linked, bits(4) LBN,
                           boolean isbreakpnt, AccType acctype, boolean ispriv)
    // "SSC", "HMC", "PxC" are the control fields from the DBGBCR[n] or DBGWCR[n] register.
    // "linked" is TRUE if this is a linked breakpoint/watchpoint type.
    // "LBN" is the linked breakpoint number from the DBGBCR[n] or DBGWCR[n] register.
    // "isbreakpnt" is TRUE for breakpoints, FALSE for watchpoints.
    // "ispriv" is valid for watchpoints, and selects between privileged and unprivileged accesses.

    // If parameters are set to a reserved type, behaves as either disabled or a defined type
    (c, SSC, HMC, PxC) = CheckValidStateMatch(SSC, HMC, PxC, isbreakpnt);
    if c == Constraint_DISABLED then return FALSE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    // Otherwise the HMC,SSC,PxC values are either valid or the values returned by
    // CheckValidStateMatch are valid.

    EL3_match = HaveEL(EL3) && HMC == '1' && SSC<0> == '0';
    EL2_match = HaveEL(EL2) && ((HMC == '1' && (SSC:PxC != '1000')) || SSC == '11');
    EL1_match = PxC<0> == '1';
    EL0_match = PxC<1> == '1';

    if HaveNV2Ext() && acctype == AccType_NV2REGISTER && !isbreakpnt then
        priv_match = EL2_match;
    elsif !ispriv && !isbreakpnt then
        priv_match = EL0_match;
    else
        case PSTATE.EL of
            when EL3  priv_match = EL3_match;
            when EL2  priv_match = EL2_match;
            when EL1  priv_match = EL1_match;
            when EL0  priv_match = EL0_match;

    case SSC of
        when '00'  security_state_match = TRUE;                        // Both
        when '01'  security_state_match = !IsSecure();                 // Non-secure only
        when '10'  security_state_match = IsSecure();                  // Secure only
        when '11'  security_state_match = (HMC == '1' || IsSecure());  // HMC=1 -> Both, 0 -> Secure
only

    if linked then
        // "LBN" must be an enabled context-aware breakpoint unit. If it is not context-aware then
        // it is CONSTRAINED UNPREDICTABLE whether this gives no match, or LBN is mapped to some
        // UNKNOWN breakpoint that is context-aware.
        lbn = UInt(LBN);
        first_ctx_cmp = (UInt(ID_AA64DFR0_EL1.BRPs) - UInt(ID_AA64DFR0_EL1.CTX_CMPs));
        last_ctx_cmp = UInt(ID_AA64DFR0_EL1.BRPs);
        if (lbn < first_ctx_cmp || lbn > last_ctx_cmp) then
            (c, lbn) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp);
            assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
            case c of
                when Constraint_DISABLED  return FALSE;      // Disabled
                when Constraint_NONE      linked = FALSE;    // No linking
                // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint

    if linked then
        vaddress = bits(64) UNKNOWN;
        linked_to = TRUE;
        linked_match = AArch64.BreakpointValueMatch(lbn, vaddress, linked_to);

    return priv_match && security_state_match && (!linked || linked_match);
```

aarch64/debug/enables/AArch64.GenerateDebugExceptions

```
// AArch64.GenerateDebugExceptions()
// ================================

boolean AArch64.GenerateDebugExceptions()
    return AArch64.GenerateDebugExceptionsFrom(PSTATE.EL, IsSecure(), PSTATE.D);
```

aarch64/debug/enables/AArch64.GenerateDebugExceptionsFrom

```
// AArch64.GenerateDebugExceptionsFrom()
// ====================================

boolean AArch64.GenerateDebugExceptionsFrom(bits(2) from, boolean secure, bit mask)

    if OSLSR_EL1.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    route_to_el2 = HaveEL(EL2) && (!secure || IsSecureEL2Enabled()) && (HCR_EL2.TGE == '1' ||
MDCR_EL2.TDE == '1');
    target = (if route_to_el2 then EL2 else EL1);
    enabled = !HaveEL(EL3) || !secure || MDCR_EL3.SDD == '0';

    if from == target then
        enabled = enabled && MDSCR_EL1.KDE == '1' && mask == '0';
    else
        enabled = enabled && UInt(target) > UInt(from);

    return enabled;
```

aarch64/debug/pmu/AArch64.CheckForPMUOverflow

```
// AArch64.CheckForPMUOverflow()
// =============================
// Signal Performance Monitors overflow IRQ and CTI overflow events

boolean AArch64.CheckForPMUOverflow()

    pmuirq = PMCR_EL0.E == '1' && PMINTENSET_EL1<31> == '1' && PMOVSSET_EL0<31> == '1';
    for n = 0 to UInt(PMCR_EL0.N) - 1
        if HaveEL(EL2) then
            E = (if n < UInt(MDCR_EL2.HPMN) then PMCR_EL0.E else MDCR_EL2.HPME);
        else
            E = PMCR_EL0.E;
        if E == '1' && PMINTENSET_EL1<n> == '1' && PMOVSSET_EL0<n> == '1' then pmuirq = TRUE;

    SetInterruptRequestLevel(InterruptID_PMUIRQ, if pmuirq then HIGH else LOW);

    CTI_SetEventLevel(CrossTriggerIn_PMUOverflow, if pmuirq then HIGH else LOW);

    // The request remains set until the condition is cleared. (For example, an interrupt handler
    // or cross-triggered event handler clears the overflow status flag by writing to PMOVSCLR_EL0.)

    return pmuirq;
```

aarch64/debug/pmu/AArch64.CountEvents

```
// AArch64.CountEvents()
// =====================
// Return TRUE if counter "n" should count its event. For the cycle counter, n == 31.

boolean AArch64.CountEvents(integer n)
    assert n == 31 || n < UInt(PMCR_EL0.N);

    // Event counting is disabled in Debug state
    debug = Halted();

    // In Non-secure state, some counters are reserved for EL2
    if HaveEL(EL2) then
        E = if n < UInt(MDCR_EL2.HPMN) || n == 31 then PMCR_EL0.E else MDCR_EL2.HPME;
    else
        E = PMCR_EL0.E;
    enabled = E == '1' && PMCNTENSET_EL0<n> == '1';

    // Event counting in Secure state is prohibited unless any one of:
    // * EL3 is not implemented
    // * EL3 is using AArch64 and MDCR_EL3.SPME == 1
    prohibited = HaveEL(EL3) && IsSecure() && MDCR_EL3.SPME == '0';

    // Event counting at EL2 is prohibited if all of:
    // * The HPMD Extension is implemented
    // * Executing at EL2
    // * PMNx is not reserved for EL2
    // * MDCR_EL2.HPMD == 1
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if !prohibited && HaveEL(EL2) && HaveHPMDExt() && PSTATE.EL == EL2 && (n < UInt(MDCR_EL2.HPMN) || n
== 31) then
        prohibited = (MDCR_EL2.HPMD == '1');

    // The IMPLEMENTATION DEFINED authentication interface might override software controls
    if prohibited && !HaveNoSecurePMUDisableOverride() then
        prohibited = !ExternalSecureNoninvasiveDebugEnabled();
    // For the cycle counter, PMCR_EL0.DP enables counting when otherwise prohibited
    if prohibited && n == 31 then prohibited = (PMCR_EL0.DP == '1');

    // Event counting can be filtered by the {P, U, NSK, NSU, NSH, M, SH} bits
    filter = if n == 31 then PMCCFILTR else PMEVTYPER[n];

    P = filter<31>;
    U = filter<30>;
    NSK = if HaveEL(EL3) then filter<29> else '0';
    NSU = if HaveEL(EL3) then filter<28> else '0';
    NSH = if HaveEL(EL2) then filter<27> else '0';
    M = if HaveEL(EL3) then filter<26> else '0';
    SH = if HaveSecureEL2Ext() then filter<24> else '0';

    case PSTATE.EL of
        when EL0 filtered = if IsSecure() then U == '1' else U != NSU;
        when EL1 filtered = if IsSecure() then P == '1' else P != NSK;
        when EL2 filtered = (if IsSecure() then NSH == SH else NSH == '0');
        when EL3 filtered = (M != P);

    return !debug && enabled && !prohibited && !filtered;
```

aarch64/debug/statisticalprofiling/CheckProfilingBufferAccess

```
// CheckProfilingBufferAccess()
// ============================

SysRegAccess CheckProfilingBufferAccess()
    if !HaveStatisticalProfiling() || PSTATE.EL == EL0 || UsingAArch32() then
        return SysRegAccess_UNDEFINED;

    if PSTATE.EL == EL1 && EL2Enabled() && MDCR_EL2.E2PB<0> != '1' then
        return SysRegAccess_TrapToEL2;

    if HaveEL(EL3) && PSTATE.EL != EL3 && MDCR_EL3.NSPB != SCR_EL3.NS:'1' then
        return SysRegAccess_TrapToEL3;

    return SysRegAccess_OK;
```

aarch64/debug/statisticalprofiling/CheckStatisticalProfilingAccess

```
// CheckStatisticalProfilingAccess()
// =================================

SysRegAccess CheckStatisticalProfilingAccess()
    if !HaveStatisticalProfiling() || PSTATE.EL == EL0 || UsingAArch32() then
        return SysRegAccess_UNDEFINED;

    if PSTATE.EL == EL1 && EL2Enabled() && MDCR_EL2.TPMS == '1' then
        return SysRegAccess_TrapToEL2;

    if HaveEL(EL3) && PSTATE.EL != EL3 && MDCR_EL3.NSPB != SCR_EL3.NS:'1' then
        return SysRegAccess_TrapToEL3;

    return SysRegAccess_OK;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch64/debug/statisticalprofiling/CollectContextIDR1

```
// CollectContextIDR1()
// ====================

boolean CollectContextIDR1()
    if !StatisticalProfilingEnabled() then return FALSE;
    if  PSTATE.EL == EL2 then return FALSE;
    if EL2Enabled() && HCR_EL2.TGE == '1' then return FALSE;
    return PMSCR_EL1.CX == '1';
```

aarch64/debug/statisticalprofiling/CollectContextIDR2

```
// CollectContextIDR2()
// ====================

boolean CollectContextIDR2()
    if !StatisticalProfilingEnabled() then return FALSE;
    if  EL2Enabled() then return FALSE;
    return PMSCR_EL2.CX == '1';
```

aarch64/debug/statisticalprofiling/CollectPhysicalAddress

```
// CollectPhysicalAddress()
// ========================

boolean CollectPhysicalAddress()
    if !StatisticalProfilingEnabled() then return FALSE;
    (secure, el) = ProfilingBufferOwner();
    if !secure && HaveEL(EL2) then
        return PMSCR_EL2.PA == '1' && (el == EL2 || PMSCR_EL1.PA == '1');
    else
        return PMSCR_EL1.PA == '1';
```

aarch64/debug/statisticalprofiling/CollectRecord

```
// CollectRecord()
// ===============

boolean CollectRecord(bits(64) events, integer total_latency, OpType optype)
    assert StatisticalProfilingEnabled();

    // Filtering by event
    if PMSFCR_EL1.FE == '1' && !IsZero(PMSEVFR_EL1) then
        bits(64) mask = 0xFFFF0000FF00F0AA<63:0>;   // Bits [63:48,31:24,15:12,7,5,3,1]
        if HaveStatisticalProfiling() then
            mask<11> = '1';                         // Alignment flag
            if HaveSVE() then mask<18:17> = Ones(); // Predicate flags
        e = events AND mask;
        m = PMSEVFR_EL1 AND mask;
        if !IsZero(NOT(e) AND m) then return FALSE;

    // Filtering by type
    if PMSFCR_EL1.FT == '1' && !IsZero(PMSFCR_EL1.<B,LD,ST>) then
        case optype of
            when OpType_Branch
                if PMSFCR_EL1.B == '0' then return FALSE;
            when OpType_Load
                if PMSFCR_EL1.LD == '0' then return FALSE;
            when OpType_Store
                if PMSFCR_EL1.ST == '0' then return FALSE;
            when OpType_LoadAtomic
                if PMSFCR_EL1.<LD,ST> == '00' then return FALSE;
            otherwise
                return FALSE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    // Filtering by latency
    if PMSFCR_EL1.FL == '1' && !IsZero(PMSLATFR_EL1.MINLAT) then
        if total_latency < UInt(PMSLATFR_EL1.MINLAT) then
            return FALSE;

    // Check for UNPREDICTABLE cases
    if ((PMSFCR_EL1.FE == '1' && !IsZero(PMSEVFR_EL1)) ||
        (PMSFCR_EL1.FT == '1' && !IsZero(PMSFCR_EL1.<B,LD,ST>)) ||
        (PMSFCR_EL1.FL == '1' && !IsZero(PMSLATFR_EL1.MINLAT))) then
        return ConstrainUnpredictableBool();

    return TRUE;
```

aarch64/debug/statisticalprofiling/CollectTimeStamp

```
// CollectTimeStamp()
// ==================

TimeStamp CollectTimeStamp()
    if !StatisticalProfilingEnabled() then return TimeStamp_None;
    (secure, el) = ProfilingBufferOwner();
    if el == EL2 then
        if PMSCR_EL2.TS == '0' then return TimeStamp_None;
    else
        if PMSCR_EL1.TS == '0' then return TimeStamp_None;
    if EL2Enabled() then
        pct = PMSCR_EL2.PCT == '01' && (el == EL2 || PMSCR_EL1.PCT == '01');
    else
        pct = PMSCR_EL1.PCT == '01';
    return (if pct then TimeStamp_Physical else TimeStamp_Virtual);
```

aarch64/debug/statisticalprofiling/OpType

```
enumeration OpType {
 OpType_Load,                 // Any memory-read operation other than atomics, compare-and-swap, and
swap
 OpType_Store,                // Any memory-write operation, including atomics without return
 OpType_LoadAtomic,           // Atomics with return, compare-and-swap and swap
 OpType_Branch,               // Software write to the PC
 OpType_Other                 // Any other class of operation
 };
```

aarch64/debug/statisticalprofiling/ProfilingBufferEnabled

```
// ProfilingBufferEnabled()
// ========================

boolean ProfilingBufferEnabled()
    if !HaveStatisticalProfiling() then return FALSE;
    (secure, el) = ProfilingBufferOwner();
    non_secure_bit = if secure then '0' else '1';
    return (!ELUsingAArch32(el) && non_secure_bit == SCR_EL3.NS &&
            PMBLIMITR_EL1.E == '1' && PMBSR_EL1.S == '0');
```

aarch64/debug/statisticalprofiling/ProfilingBufferOwner

```
// ProfilingBufferOwner()
// ======================

(boolean, bits(2)) ProfilingBufferOwner()
    secure = if HaveEL(EL3) then (MDCR_EL3.NSPB<1> == '0') else IsSecure();
    el = if !secure && HaveEL(EL2) && MDCR_EL2.E2PB == '00' then EL2 else EL1;
    return (secure, el);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch64/debug/statisticalprofiling/ProfilingSynchronizationBarrier

```
// Barrier to ensure that all existing profiling data has been formatted, and profiling buffer
// addresses have been translated such that writes to the profiling buffer have been initiated.
// A following DSB completes when writes to the profiling buffer have completed.
ProfilingSynchronizationBarrier();
```

aarch64/debug/statisticalprofiling/StatisticalProfilingEnabled

```
// StatisticalProfilingEnabled()
// =============================

boolean StatisticalProfilingEnabled()
    if !HaveStatisticalProfiling() || UsingAArch32() || !ProfilingBufferEnabled() then
        return FALSE;

    in_host = EL2Enabled() && HCR_EL2.TGE == '1';
    (secure, el) = ProfilingBufferOwner();
    if UInt(el) <  UInt(PSTATE.EL) || secure != IsSecure() || (in_host && el == EL1)  then
        return FALSE;

    case PSTATE.EL of
        when EL3  Unreachable();
        when EL2  spe_bit = PMSCR_EL2.E2SPE;
        when EL1  spe_bit = PMSCR_EL1.E1SPE;
        when EL0  spe_bit = (if in_host then PMSCR_EL2.E0HSPE else PMSCR_EL1.E0SPE);

    return spe_bit == '1';
```

aarch64/debug/statisticalprofiling/SysRegAccess

```
enumeration SysRegAccess { SysRegAccess_OK,
                           SysRegAccess_UNDEFINED,
                           SysRegAccess_TrapToEL1,
                           SysRegAccess_TrapToEL2,
                           SysRegAccess_TrapToEL3 };
```

aarch64/debug/statisticalprofiling/TimeStamp

```
enumeration TimeStamp {
    TimeStamp_None,        // No timestamp
    TimeStamp_CoreSight,   // CoreSight time (IMPLEMENTATION DEFINED)
    TimeStamp_Virtual,     // Physical counter value minus CNTVOFF_EL2
    TimeStamp_Physical };  // Physical counter value with no offset
```

aarch64/debug/takeexceptiondbg/AArch64.TakeExceptionInDebugState

```
// AArch64.TakeExceptionInDebugState()
// ===================================
// Take an exception in Debug state to an Exception Level using AArch64.

AArch64.TakeExceptionInDebugState(bits(2) target_el, ExceptionRecord exception)
    assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);

    sync_errors = HaveIESB() && SCTLR[target_el].IESB == '1';
    if HaveDoubleFaultExt() then
        sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3);
    // SCTLR[].IESB might be ignored in Debug state.
    if !ConstrainUnpredictableBool() then
        sync_errors = FALSE;

    SynchronizeContext();

    // If coming from AArch32 state, the top parts of the X[] registers might be set to zero
    from_32 = UsingAArch32();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if from_32 then AArch64.MaybeZeroRegisterUppers();
    MaybeZeroSVEUppers(target_el);

    AArch64.ReportException(exception, target_el);

    PSTATE.EL = target_el;
    PSTATE.nRW = '0';
    PSTATE.SP = '1';

    SPSR[] = bits(32) UNKNOWN;
    ELR[] = bits(64) UNKNOWN;

    // PSTATE.{SS,D,A,I,F} are not observable and ignored in Debug state, so behave as if UNKNOWN.
    PSTATE.<SS,D,A,I,F> = bits(5) UNKNOWN;
    PSTATE.IL = '0';
    if from_32 then                             // Coming from AArch32
        PSTATE.IT = '00000000';
        PSTATE.T = '0';                         // PSTATE.J is RES0
    if (HavePANExt() && (PSTATE.EL == EL1 || (PSTATE.EL == EL2 && ELIsInHost(EL0))) &&
        SCTLR[].SPAN == '0') then
        PSTATE.PAN = '1';
    if HaveUAOExt() then PSTATE.UAO = '0';
    if HaveBTIExt() then PSTATE.BTYPE = '00';
    if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
    if HaveMTEExt() then PSTATE.TCO = '1';

    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(32) UNKNOWN;

    EDSCR.ERR = '1';
    UpdateEDSCRFields();                        // Update EDSCR processor state flags.

    if sync_errors then
        SynchronizeErrors();

    EndOfInstruction();
```

aarch64/debug/watchpoint/AArch64.WatchpointByteMatch

```
// AArch64.WatchpointByteMatch()
// ============================

boolean AArch64.WatchpointByteMatch(integer n, AccType acctype, bits(64) vaddress)

    el = if HaveNV2Ext() && acctype == AccType_NV2REGISTER then EL2 else PSTATE.EL;
    top = AddrTop(vaddress, FALSE, el);
    bottom = if DBGWVR_EL1[n]<2> == '1' then 2 else 3;            // Word or doubleword
    byte_select_match = (DBGWCR_EL1[n].BAS<UInt(vaddress<bottom-1:0>)> != '0');
    mask = UInt(DBGWCR_EL1[n].MASK);

    // If DBGWCR_EL1[n].MASK is non-zero value and DBGWCR_EL1[n].BAS is not set to '11111111', or
    // DBGWCR_EL1[n].BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
    // UNPREDICTABLE.
    if mask > 0 && !IsOnes(DBGWCR_EL1[n].BAS) then
        byte_select_match = ConstrainUnpredictableBool();
    else
        LSB = (DBGWCR_EL1[n].BAS AND NOT(DBGWCR_EL1[n].BAS - 1));  MSB = (DBGWCR_EL1[n].BAS + LSB);
        if !IsZero(MSB AND (MSB - 1)) then                      // Not contiguous
            byte_select_match = ConstrainUnpredictableBool();
            bottom = 3;                                         // For the whole doubleword

    // If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask > 0 && mask <= 2 then
        (c, mask) = ConstrainUnpredictableInteger(3, 31);
        assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
        case c of
            when Constraint_DISABLED  return FALSE;             // Disabled
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            when Constraint_NONE      mask = 0;                // No masking
            // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value

    if mask > bottom then
        WVR_match = (vaddress<top:mask> == DBGWVR_EL1[n]<top:mask>);
        // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
        if WVR_match && !IsZero(DBGWVR_EL1[n]<mask-1:bottom>) then
            WVR_match = ConstrainUnpredictableBool();
    else
        WVR_match = vaddress<top:bottom> == DBGWVR_EL1[n]<top:bottom>;

    return WVR_match && byte_select_match;
```

aarch64/debug/watchpoint/AArch64.WatchpointMatch

```
// AArch64.WatchpointMatch()
// =========================
// Watchpoint matching in an AArch64 translation regime.

boolean AArch64.WatchpointMatch(integer n, bits(64) vaddress, integer size, boolean ispriv,
                                AccType acctype, boolean iswrite)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert n <= UInt(ID_AA64DFR0_EL1.WRPs);

    // "ispriv" is FALSE for LDTR/STTR instructions executed at EL1 and all
    // load/stores at EL0, TRUE for all other load/stores. "iswrite" is TRUE for stores, FALSE for
    // loads.
    enabled = DBGWCR_EL1[n].E == '1';
    linked = DBGWCR_EL1[n].WT == '1';
    isbreakpnt = FALSE;

    state_match = AArch64.StateMatch(DBGWCR_EL1[n].SSC, DBGWCR_EL1[n].HMC, DBGWCR_EL1[n].PAC,
                                     linked, DBGWCR_EL1[n].LBN, isbreakpnt, acctype, ispriv);

    ls_match = (DBGWCR_EL1[n].LSC<(if iswrite then 1 else 0)> == '1');

    value_match = FALSE;
    for byte = 0 to size - 1
        value_match = value_match || AArch64.WatchpointByteMatch(n, acctype, vaddress + byte);

    return value_match && state_match && ls_match && enabled;
```

J1.1.2 aarch64/exceptions

This section includes the following pseudocode functions:

- *aarch64/exceptions/aborts/AArch64.Abort* on page J1-7519.
- *aarch64/exceptions/aborts/AArch64.AbortSyndrome* on page J1-7520.
- *aarch64/exceptions/aborts/AArch64.CheckPCAlignment* on page J1-7520.
- *aarch64/exceptions/aborts/AArch64.DataAbort* on page J1-7520.
- *aarch64/exceptions/aborts/AArch64.EffectiveTCF* on page J1-7521.
- *aarch64/exceptions/aborts/AArch64.InstructionAbort* on page J1-7521.
- *aarch64/exceptions/aborts/AArch64.PCAlignmentFault* on page J1-7521.
- *aarch64/exceptions/aborts/AArch64.ReportTagCheckFail* on page J1-7522.
- *aarch64/exceptions/aborts/AArch64.SPAlignmentFault* on page J1-7522.
- *aarch64/exceptions/aborts/AArch64.TagCheckFail* on page J1-7522.
- *aarch64/exceptions/aborts/AArch64.TagCheckFault* on page J1-7523.
- *aarch64/exceptions/aborts/BranchTargetException* on page J1-7523.
- *aarch64/exceptions/asynch/AArch64.TakePhysicalFIQException* on page J1-7523.
- *aarch64/exceptions/asynch/AArch64.TakePhysicalIRQException* on page J1-7524.
- *aarch64/exceptions/asynch/AArch64.TakePhysicalSErrorException* on page J1-7524.
- *aarch64/exceptions/asynch/AArch64.TakeVirtualFIQException* on page J1-7524.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *aarch64/exceptions/asynch/AArch64.TakeVirtualIRQException* on page J1-7525.
- *aarch64/exceptions/asynch/AArch64.TakeVirtualSErrorException* on page J1-7525.
- *aarch64/exceptions/debug/AArch64.BreakpointException* on page J1-7525.
- *aarch64/exceptions/debug/AArch64.SoftwareBreakpoint* on page J1-7526.
- *aarch64/exceptions/debug/AArch64.SoftwareStepException* on page J1-7526.
- *aarch64/exceptions/debug/AArch64.VectorCatchException* on page J1-7526.
- *aarch64/exceptions/debug/AArch64.WatchpointException* on page J1-7527.
- *aarch64/exceptions/exceptions/AArch64.ExceptionClass* on page J1-7527.
- *aarch64/exceptions/exceptions/AArch64.ReportException* on page J1-7528.
- *aarch64/exceptions/exceptions/AArch64.ResetControlRegisters* on page J1-7528.
- *aarch64/exceptions/exceptions/AArch64.TakeReset* on page J1-7528.
- *aarch64/exceptions/ieeefp/AArch64.FPTrappedException* on page J1-7529.
- *aarch64/exceptions/syscalls/AArch64.CallHypervisor* on page J1-7530.
- *aarch64/exceptions/syscalls/AArch64.CallSecureMonitor* on page J1-7530.
- *aarch64/exceptions/syscalls/AArch64.CallSupervisor* on page J1-7530.
- *aarch64/exceptions/takeexception/AArch64.TakeException* on page J1-7531.
- *aarch64/exceptions/traps/AArch64.AArch32SystemAccessTrap* on page J1-7532.
- *aarch64/exceptions/traps/AArch64.AArch32SystemAccessTrapSyndrome* on page J1-7532.
- *aarch64/exceptions/traps/AArch64.AdvSIMDFPAccessTrap* on page J1-7534.
- *aarch64/exceptions/traps/AArch64.CheckCP15InstrCoarseTraps* on page J1-7534.
- *aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDEnabled* on page J1-7534.
- *aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDTrap* on page J1-7535.
- *aarch64/exceptions/traps/AArch64.CheckForERetTrap* on page J1-7535.
- *aarch64/exceptions/traps/AArch64.CheckForSMCUndefOrTrap* on page J1-7535.
- *aarch64/exceptions/traps/AArch64.CheckForSVCTrap* on page J1-7536.
- *aarch64/exceptions/traps/AArch64.CheckForWFxTrap* on page J1-7536.
- *aarch64/exceptions/traps/AArch64.CheckIllegalState* on page J1-7536.
- *aarch64/exceptions/traps/AArch64.MonitorModeTrap* on page J1-7537.
- *aarch64/exceptions/traps/AArch64.SystemAccessTrap* on page J1-7537.
- *aarch64/exceptions/traps/AArch64.SystemAccessTrapSyndrome* on page J1-7537.
- *aarch64/exceptions/traps/AArch64.UndefinedFault* on page J1-7538.
- *aarch64/exceptions/traps/AArch64.WFxTrap* on page J1-7538.
- *aarch64/exceptions/traps/CheckFPAdvSIMDEnabled64* on page J1-7538.

aarch64/exceptions/aborts/AArch64.Abort

```
// AArch64.Abort()
// ===============
// Abort and Debug exception handling in an AArch64 translation regime.

AArch64.Abort(bits(64) vaddress, FaultRecord fault)

    if IsDebugException(fault) then
        if fault.acctype == AccType_IFETCH then
            if UsingAArch32() && fault.debugmoe == DebugException_VectorCatch then
                AArch64.VectorCatchException(fault);
            else
                AArch64.BreakpointException(fault);
        else
            AArch64.WatchpointException(vaddress, fault);
    elsif fault.acctype == AccType_IFETCH then
        AArch64.InstructionAbort(vaddress, fault);
    else
        AArch64.DataAbort(vaddress, fault);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch64/exceptions/aborts/AArch64.AbortSyndrome

```
// AArch64.AbortSyndrome()
// =======================
// Creates an exception syndrome record for Abort and Watchpoint exceptions
// from an AArch64 translation regime.

ExceptionRecord AArch64.AbortSyndrome(Exception exceptype, FaultRecord fault, bits(64) vaddress)
    exception = ExceptionSyndrome(exceptype);

    d_side = exceptype IN {Exception_DataAbort, Exception_NV2DataAbort, Exception_Watchpoint,
Exception_NV2Watchpoint};

    exception.syndrome = AArch64.FaultSyndrome(d_side, fault);
    exception.vaddress = ZeroExtend(vaddress);
    if IPAValid(fault) then
        exception.ipavalid = TRUE;
        exception.NS = fault.ipaddress.NS;
        exception.ipaddress = fault.ipaddress.address;
    else
        exception.ipavalid = FALSE;

    return exception;
```

aarch64/exceptions/aborts/AArch64.CheckPCAlignment

```
// AArch64.CheckPCAlignment()
// ==========================

AArch64.CheckPCAlignment()

    bits(64) pc = ThisInstrAddr();
    if pc<1:0> != '00' then
        AArch64.PCAlignmentFault();
```

aarch64/exceptions/aborts/AArch64.DataAbort

```
// AArch64.DataAbort()
// ===================

AArch64.DataAbort(bits(64) vaddress, FaultRecord fault)
    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1' && IsExternalAbort(fault);
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() && (HCR_EL2.TGE == '1' ||
                    (HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault)) ||
                    (HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER) ||
                    IsSecondStage(fault)));

    bits(64) preferred_exception_return = ThisInstrAddr();
    if (HaveDoubleFaultExt() && (PSTATE.EL == EL3 || route_to_el3) &&
        IsExternalAbort(fault) && SCR_EL3.EASE == '1') then
        vect_offset = 0x180;
    else
        vect_offset = 0x0;
    if HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER then
        exception = AArch64.AbortSyndrome(Exception_NV2DataAbort, fault, vaddress);
    else
        exception = AArch64.AbortSyndrome(Exception_DataAbort, fault, vaddress);
    if PSTATE.EL == EL3 || route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch64/exceptions/aborts/AArch64.EffectiveTCF

```
// AArch64.EffectiveTCF()
// ======================
// Returns the TCF field applied to Tag Check Fails in the given Exception Level.

bits(2) AArch64.EffectiveTCF(bits(2) el)
    bits(2) tcf;

    if el == EL3 then
        tcf = SCTLR_EL3.TCF;
    elsif el == EL2 then
        tcf = SCTLR_EL2.TCF;
    elsif el == EL1 then
        tcf = SCTLR_EL1.TCF;
    elsif el == EL0 && HCR_EL2.<E2H,TGE> == '11' then
        tcf = SCTLR_EL2.TCF0;
    elsif el == EL0 && HCR_EL2.<E2H,TGE> != '11' then
        tcf = SCTLR_EL1.TCF0;

    if tcf == '11' then
        (-,tcf) = ConstrainUnpredictableBits();

    return tcf;
```

aarch64/exceptions/aborts/AArch64.InstructionAbort

```
// AArch64.InstructionAbort()
// ==========================

AArch64.InstructionAbort(bits(64) vaddress, FaultRecord fault)
    // External aborts on instruction fetch must be taken synchronously
    if HaveDoubleFaultExt() then assert fault.statuscode != Fault_AsyncExternal;
    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1' && IsExternalAbort(fault);
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || IsSecondStage(fault) ||
                    (HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault))));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = AArch64.AbortSyndrome(Exception_InstructionAbort, fault, vaddress);

    if PSTATE.EL == EL3 || route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/aborts/AArch64.PCAlignmentFault

```
// AArch64.PCAlignmentFault()
// ==========================
// Called on unaligned program counter in AArch64 state.

AArch64.PCAlignmentFault()

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_PCAlignment);
    exception.vaddress = ThisInstrAddr();

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/aborts/AArch64.ReportTagCheckFail

```
// AArch64.ReportTagCheckFail()
// ============================
// Records a tag fail exception into the appropriate TCFR_ELx.

AArch64.ReportTagCheckFail(bits(2) el, bit ttbr)
    if el == EL3 then
        assert ttbr == '0';
        TFSR_EL3.TF0 = '1';
    elsif el == EL2 then
        if ttbr == '0' then
            TFSR_EL2.TF0 = '1';
        else
            TFSR_EL2.TF1 = '1';
    elsif el == EL1 then
        if ttbr == '0' then
            TFSR_EL1.TF0 = '1';
        else
            TFSR_EL1.TF1 = '1';
    elsif el == EL0 then
        if ttbr == '0' then
            TFSRE0_EL1.TF0 = '1';
        else
            TFSRE0_EL1.TF1 = '1';
```

aarch64/exceptions/aborts/AArch64.SPAlignmentFault

```
// AArch64.SPAlignmentFault()
// ==========================
// Called on an unaligned stack pointer in AArch64 state.

AArch64.SPAlignmentFault()

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SPAlignment);

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/aborts/AArch64.TagCheckFail

```
// AArch64.TagCheckFail()
// ======================
// Handle a tag check fail condition.

AArch64.TagCheckFail(bits(64) vaddress, boolean iswrite)
    bits(2) tcf = AArch64.EffectiveTCF(PSTATE.EL);
    if tcf == '01' then
        AArch64.TagCheckFault(vaddress, iswrite);
    elsif tcf == '10' then
        AArch64.ReportTagCheckFail(PSTATE.EL, vaddress<55>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch64/exceptions/aborts/AArch64.TagCheckFault

```
// AArch64.TagCheckFault()
// =======================
// Raise a tag check fail exception.

AArch64.TagCheckFault(bits(64) va, boolean write)
    bits(2) target_el;
    bits(64) preferred_exception_return = ThisInstrAddr();
    integer vect_offset = 0x0;

    if PSTATE.EL == EL0 then
        target_el = if HCR_EL2.TGE == '0' then EL1 else EL2;
    else
        target_el = PSTATE.EL;

    exception = ExceptionSyndrome(Exception_DataAbort);
    exception.syndrome<5:0> = '010001';
    if write then
        exception.syndrome<6> = '1';
    exception.vaddress = bits(4) UNKNOWN : va<59:0>;

    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/aborts/BranchTargetException

```
// BranchTargetException
// =====================
// Raise branch target exception.

AArch64.BranchTargetException(bits(52) vaddress)

    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_BranchTarget);
    exception.syndrome<1:0>   = PSTATE.BTYPE;
    exception.syndrome<24:2>  = Zeros();         // RES0

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/asynch/AArch64.TakePhysicalFIQException

```
// AArch64.TakePhysicalFIQException()
// ==================================

AArch64.TakePhysicalFIQException()

    route_to_el3 = HaveEL(EL3) && SCR_EL3.FIQ == '1';
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || HCR_EL2.FMO == '1'));
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x100;
    exception = ExceptionSyndrome(Exception_FIQ);

    if route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        assert PSTATE.EL != EL3;
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        assert PSTATE.EL IN {EL0, EL1};
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/asynch/AArch64.TakePhysicalIRQException

```
// AArch64.TakePhysicalIRQException()
// ==================================
// Take an enabled physical IRQ exception.

AArch64.TakePhysicalIRQException()

    route_to_el3 = HaveEL(EL3) && SCR_EL3.IRQ == '1';
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || HCR_EL2.IMO == '1'));
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x80;

    exception = ExceptionSyndrome(Exception_IRQ);

    if route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        assert PSTATE.EL != EL3;
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        assert PSTATE.EL IN {EL0, EL1};
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/asynch/AArch64.TakePhysicalSErrorException

```
// AArch64.TakePhysicalSErrorException()
// =====================================

AArch64.TakePhysicalSErrorException(boolean impdef_syndrome, bits(24) syndrome)

    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1';
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || (!IsInHost() && HCR_EL2.AMO == '1')));
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x180;

    exception = ExceptionSyndrome(Exception_SError);
    exception.syndrome<24> = if impdef_syndrome then '1' else '0';
    exception.syndrome<23:0> = syndrome;

    ClearPendingPhysicalSError();

    if PSTATE.EL == EL3 || route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/asynch/AArch64.TakeVirtualFIQException

```
// AArch64.TakeVirtualFIQException()
// =================================

AArch64.TakeVirtualFIQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1';  // Virtual IRQ enabled if TGE==0 and FMO==1

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x100;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    exception = ExceptionSyndrome(Exception_FIQ);

    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/asynch/AArch64.TakeVirtualIRQException

```
// AArch64.TakeVirtualIRQException()
// ================================

AArch64.TakeVirtualIRQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1';  // Virtual IRQ enabled if TGE==0 and IMO==1

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x80;

    exception = ExceptionSyndrome(Exception_IRQ);

    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/asynch/AArch64.TakeVirtualSErrorException

```
// AArch64.TakeVirtualSErrorException()
// ====================================

AArch64.TakeVirtualSErrorException(boolean impdef_syndrome, bits(24) syndrome)

    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';  // Virtual SError enabled if TGE==0 and AMO==1

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x180;

    exception = ExceptionSyndrome(Exception_SError);
    if HaveRASExt() then
        exception.syndrome<24>   = VSESR_EL2.IDS;
        exception.syndrome<23:0> = VSESR_EL2.ISS;
    else
        exception.syndrome<24> = if impdef_syndrome then '1' else '0';
        if impdef_syndrome then exception.syndrome<23:0> = syndrome;

    ClearPendingVirtualSError();
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/debug/AArch64.BreakpointException

```
// AArch64.BreakpointException()
// =============================

AArch64.BreakpointException(FaultRecord fault)
    assert PSTATE.EL != EL3;

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    vaddress = bits(64) UNKNOWN;
    exception = AArch64.AbortSyndrome(Exception_Breakpoint, fault, vaddress);

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch64/exceptions/debug/AArch64.SoftwareBreakpoint

```
// AArch64.SoftwareBreakpoint()
// ============================

AArch64.SoftwareBreakpoint(bits(16) immediate)

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} &&
                    EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SoftwareBreakpoint);
    exception.syndrome<15:0> = immediate;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/debug/AArch64.SoftwareStepException

```
// AArch64.SoftwareStepException()
// ===============================

AArch64.SoftwareStepException()
    assert PSTATE.EL != EL3;

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SoftwareStep);
    if SoftwareStep_DidNotStep() then
        exception.syndrome<24> = '0';
    else
        exception.syndrome<24> = '1';
        exception.syndrome<6> = if SoftwareStep_SteppedEX() then '1' else '0';

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/debug/AArch64.VectorCatchException

```
// AArch64.VectorCatchException()
// ==============================
// Vector Catch taken from EL0 or EL1 to EL2. This can only be called when debug exceptions are
// being routed to EL2, as Vector Catch is a legacy debug event.

AArch64.VectorCatchException(FaultRecord fault)
    assert PSTATE.EL != EL2;
    assert EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1');

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    vaddress = bits(64) UNKNOWN;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    exception = AArch64.AbortSyndrome(Exception_VectorCatch, fault, vaddress);

    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/debug/AArch64.WatchpointException

```
// AArch64.WatchpointException()
// =============================

AArch64.WatchpointException(bits(64) vaddress, FaultRecord fault)
    assert PSTATE.EL != EL3;

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    if HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER then
        exception = AArch64.AbortSyndrome(Exception_NV2Watchpoint, fault, vaddress);
    else
        exception = AArch64.AbortSyndrome(Exception_Watchpoint, fault, vaddress);

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/exceptions/AArch64.ExceptionClass

```
// AArch64.ExceptionClass()
// ========================
// Returns the Exception Class and Instruction Length fields to be reported in ESR

(integer,bit) AArch64.ExceptionClass(Exception exceptype, bits(2) target_el)

    il = if ThisInstrLength() == 32 then '1' else '0';
    from_32 = UsingAArch32();
    assert from_32 || il == '1';            // AArch64 instructions always 32-bit

    case exceptype of
        when Exception_Uncategorized        ec = 0x00; il = '1';
        when Exception_WFxTrap              ec = 0x01;
        when Exception_CP15RTTrap           ec = 0x03;           assert from_32;
        when Exception_CP15RRTTrap          ec = 0x04;           assert from_32;
        when Exception_CP14RTTrap           ec = 0x05;           assert from_32;
        when Exception_CP14DTTrap           ec = 0x06;           assert from_32;
        when Exception_AdvSIMDFPAccessTrap  ec = 0x07;
        when Exception_FPIDTrap             ec = 0x08;
        when Exception_PACTrap              ec = 0x09;
        when Exception_CP14RRTTrap          ec = 0x0C;           assert from_32;
        when Exception_BranchTarget         ec = 0x0D;
        when Exception_IllegalState         ec = 0x0E; il = '1';
        when Exception_SupervisorCall       ec = 0x11;
        when Exception_HypervisorCall       ec = 0x12;
        when Exception_MonitorCall          ec = 0x13;
        when Exception_SystemRegisterTrap   ec = 0x18;           assert !from_32;
        when Exception_SVEAccessTrap        ec = 0x19;           assert !from_32;
        when Exception_ERetTrap             ec = 0x1A;
        when Exception_PACFail              ec = 0x1C;
        when Exception_InstructionAbort     ec = 0x20; il = '1';
        when Exception_PCAlignment          ec = 0x22; il = '1';
        when Exception_DataAbort            ec = 0x24;
        when Exception_NV2DataAbort         ec = 0x25;
        when Exception_SPAlignment          ec = 0x26; il = '1'; assert !from_32;
        when Exception_FPTrappedException   ec = 0x28;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        when Exception_SError               ec = 0x2F; il = '1';
        when Exception_Breakpoint           ec = 0x30; il = '1';
        when Exception_SoftwareStep         ec = 0x32; il = '1';
        when Exception_Watchpoint           ec = 0x34; il = '1';
        when Exception_NV2Watchpoint        ec = 0x35; il = '1';
        when Exception_SoftwareBreakpoint   ec = 0x38;
        when Exception_VectorCatch          ec = 0x3A; il = '1'; assert from_32;
        otherwise                           Unreachable();

    if ec IN {0x20,0x24,0x30,0x32,0x34} && target_el == PSTATE.EL then
        ec = ec + 1;

    if ec IN {0x11,0x12,0x13,0x28,0x38} && !from_32 then
        ec = ec + 4;

    return (ec,il);
```

aarch64/exceptions/exceptions/AArch64.ReportException

```
// AArch64.ReportException()
// =========================
// Report syndrome information for exception taken to AArch64 state.

AArch64.ReportException(ExceptionRecord exception, bits(2) target_el)

    Exception exceptype = exception.exceptype;

    (ec,il) = AArch64.ExceptionClass(exceptype, target_el);
    iss = exception.syndrome;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
        il = '1';

    ESR[target_el] = ec<5:0>:il:iss;

    if exceptype IN {Exception_InstructionAbort, Exception_PCAlignment, Exception_DataAbort,
                Exception_NV2DataAbort, Exception_NV2Watchpoint,
                Exception_Watchpoint} then
        FAR[target_el] = exception.vaddress;
    else
        FAR[target_el] = bits(64) UNKNOWN;

    if target_el == EL2 then
        if exception.ipavalid then
            HPFAR_EL2<43:4> = exception.ipaddress<51:12>;
            if IsSecureEL2Enabled() && IsSecure() then
                HPFAR_EL2.NS = exception.NS;
            else
                HPFAR_EL2.NS = '0';
        else
            HPFAR_EL2<43:4> = bits(40) UNKNOWN;

    return;
```

aarch64/exceptions/exceptions/AArch64.ResetControlRegisters

```
// Resets System registers and memory-mapped control registers that have architecturally-defined
// reset values to those values.
AArch64.ResetControlRegisters(boolean cold_reset);
```

aarch64/exceptions/exceptions/AArch64.TakeReset

```
// AArch64.TakeReset()
// ===================
// Reset into AArch64 state
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
AArch64.TakeReset(boolean cold_reset)
    assert !HighestELUsingAArch32();

    // Enter the highest implemented Exception level in AArch64 state
    PSTATE.nRW = '0';
    if HaveEL(EL3) then
        PSTATE.EL = EL3;
    elsif HaveEL(EL2) then
        PSTATE.EL = EL2;
    else
        PSTATE.EL = EL1;

    // Reset the system registers and other system components
    AArch64.ResetControlRegisters(cold_reset);

    // Reset all other PSTATE fields
    PSTATE.SP = '1';              // Select stack pointer
    PSTATE.<D,A,I,F>  = '1111';   // All asynchronous exceptions masked
    PSTATE.SS = '0';              // Clear software step bit
    PSTATE.DIT = '0';             // PSTATE.DIT is reset to 0 when resetting into AArch64
    PSTATE.IL = '0';              // Clear Illegal Execution state bit

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // ELR_ELx and SPSR_ELx have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch64.ResetGeneralRegisters();
    AArch64.ResetSIMDFPRegisters();
    AArch64.ResetSpecialRegisters();
    ResetExternalDebugRegisters(cold_reset);

    bits(64) rv;                      // IMPLEMENTATION DEFINED reset vector

    if HaveEL(EL3) then
        rv = RVBAR_EL3;
    elsif HaveEL(EL2) then
        rv = RVBAR_EL2;
    else
        rv = RVBAR_EL1;

    // The reset vector must be correctly aligned
    assert IsZero(rv<63:PAMax()>) && IsZero(rv<1:0>);

    BranchTo(rv, BranchType_RESET);
```

aarch64/exceptions/ieeefp/AArch64.FPTrappedException

```
// AArch64.FPTrappedException()
// ============================

AArch64.FPTrappedException(boolean is_ase, integer element, bits(8) accumulated_exceptions)
    exception = ExceptionSyndrome(Exception_FPTrappedException);
    if is_ase then
        if boolean IMPLEMENTATION_DEFINED "vector instructions set TFV to 1" then
            exception.syndrome<23> = '1';                          // TFV
        else
            exception.syndrome<23> = '0';                          // TFV
    else
        exception.syndrome<23> = '1';                              // TFV
    exception.syndrome<10:8> = bits(3) UNKNOWN;                    // VECITR
    if exception.syndrome<23> == '1' then
        exception.syndrome<7,4:0> = accumulated_exceptions<7,4:0>; // IDF,IXF,UFF,OFF,DZF,IOF
    else
        exception.syndrome<7,4:0> = bits(6) UNKNOWN;

    route_to_el2 = EL2Enabled() && HCR_EL2.TGE == '1';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/syscalls/AArch64.CallHypervisor

```
// AArch64.CallHypervisor()
// ========================
// Performs a HVC call

AArch64.CallHypervisor(bits(16) immediate)
    assert HaveEL(EL2);

    if UsingAArch32() then AArch32.ITAdvance();
    SSAdvance();
    bits(64) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_HypervisorCall);
    exception.syndrome<15:0> = immediate;

    if PSTATE.EL == EL3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/syscalls/AArch64.CallSecureMonitor

```
// AArch64.CallSecureMonitor()
// ===========================

AArch64.CallSecureMonitor(bits(16) immediate)
    assert HaveEL(EL3) && !ELUsingAArch32(EL3);
    if UsingAArch32() then AArch32.ITAdvance();
    SSAdvance();
    bits(64) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_MonitorCall);
    exception.syndrome<15:0> = immediate;

    AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/syscalls/AArch64.CallSupervisor

```
// AArch64.CallSupervisor()
// ========================
// Calls the Supervisor

AArch64.CallSupervisor(bits(16) immediate)

    if UsingAArch32() then AArch32.ITAdvance();
    SSAdvance();
    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';

    bits(64) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SupervisorCall);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    exception.syndrome<15:0> = immediate;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/takeexception/AArch64.TakeException

```
// AArch64.TakeException()
// =======================
// Take an exception to an Exception Level using AArch64.

AArch64.TakeException(bits(2) target_el, ExceptionRecord exception,
                      bits(64) preferred_exception_return, integer vect_offset)
    assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);

    sync_errors = HaveIESB() && SCTLR[target_el].IESB == '1';
    if HaveDoubleFaultExt() then
        sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3);
    if sync_errors && InsertIESBBeforeException(target_el) then
        SynchronizeErrors();
        iesb_req = FALSE;
        sync_errors = FALSE;
        TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);

    SynchronizeContext();

    // If coming from AArch32 state, the top parts of the X[] registers might be set to zero
    from_32 = UsingAArch32();
    if from_32 then AArch64.MaybeZeroRegisterUppers();
    MaybeZeroSVEUppers(target_el);

    if UInt(target_el) > UInt(PSTATE.EL) then
        boolean lower_32;
        if target_el == EL3 then
            if EL2Enabled() then
                lower_32 = ELUsingAArch32(EL2);
            else
                lower_32 = ELUsingAArch32(EL1);
        elsif IsInHost() && PSTATE.EL == EL0 && target_el == EL2 then
            lower_32 = ELUsingAArch32(EL0);
        else
            lower_32 = ELUsingAArch32(target_el - 1);
        vect_offset = vect_offset + (if lower_32 then 0x600 else 0x400);

    elsif PSTATE.SP == '1' then
        vect_offset = vect_offset + 0x200;

    spsr = GetPSRFromPSTATE();

    if PSTATE.EL == EL1 && target_el == EL1 && EL2Enabled() then
        if HaveNV2Ext() && (HCR_EL2.<NV,NV1,NV2> == '100' || HCR_EL2.<NV,NV1,NV2> == '111') then
            spsr<3:2> = '10';
        else
            if HaveNVExt() && HCR_EL2.<NV,NV1> == '10' then
                spsr<3:2> = '10';

    if HaveBTIExt() then
        // SPSR[].BTYPE is only guaranteed valid for these exception types
        if exception.exceptype IN {Exception_SError, Exception_IRQ, Exception_FIQ,
                                   Exception_SoftwareStep, Exception_PCAlignment,
                                   Exception_InstructionAbort, Exception_Breakpoint,
                                   Exception_VectorCatch, Exception_SoftwareBreakpoint,
                                   Exception_IllegalState, Exception_BranchTarget} then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            zero_btype = FALSE;
        else
            zero_btype = ConstrainUnpredictableBool();
        if zero_btype then spsr<11:10> = '00';

    if HaveNV2Ext() && exception.exceptype == Exception_NV2DataAbort && target_el == EL3 then
        // external aborts are configured to be taken to EL3
        exception.exceptype = Exception_DataAbort;
    if !(exception.exceptype IN {Exception_IRQ, Exception_FIQ}) then
        AArch64.ReportException(exception, target_el);

    PSTATE.EL = target_el;
    PSTATE.nRW = '0';
    PSTATE.SP = '1';

    SPSR[] = spsr;
    ELR[] = preferred_exception_return;

    PSTATE.SS = '0';
    PSTATE.<D,A,I,F> = '1111';
    PSTATE.IL = '0';
    if from_32 then                             // Coming from AArch32
        PSTATE.IT = '00000000';
        PSTATE.T = '0';                         // PSTATE.J is RES0
    if (HavePANExt() && (PSTATE.EL == EL1 || (PSTATE.EL == EL2 && ELIsInHost(EL0))) &&
        SCTLR[].SPAN == '0') then
        PSTATE.PAN = '1';
    if HaveUAOExt() then PSTATE.UAO = '0';
    if HaveBTIExt() then PSTATE.BTYPE = '00';
    if HaveSSBSExt() then PSTATE.SSBS = SCTLR[].DSSBS;
    if HaveMTEExt() then PSTATE.TCO = '1';

    BranchTo(VBAR[]<63:11>:vect_offset<10:0>, BranchType_EXCEPTION);

    if sync_errors then
        SynchronizeErrors();
        iesb_req = TRUE;
        TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);

    EndOfInstruction();
```

aarch64/exceptions/traps/AArch64.AArch32SystemAccessTrap

```
// AArch64.AArch32SystemAccessTrap()
// ================================
// Trapped AARCH32 system register access.

AArch64.AArch32SystemAccessTrap(bits(2) target_el, integer ec)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = AArch64.AArch32SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/traps/AArch64.AArch32SystemAccessTrapSyndrome

```
// AArch64.AArch32SystemAccessTrapSyndrome()
// =========================================
// Returns the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS, VMSR
instructions,
// other than traps that are due to HCPTR or CPACR.

ExceptionRecord AArch64.AArch32SystemAccessTrapSyndrome(bits(32) instr, integer ec)
    ExceptionRecord exception;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    case ec of
        when 0x0    exception = ExceptionSyndrome(Exception_Uncategorized);
        when 0x3    exception = ExceptionSyndrome(Exception_CP15RTTrap);
        when 0x4    exception = ExceptionSyndrome(Exception_CP15RRTTrap);
        when 0x5    exception = ExceptionSyndrome(Exception_CP14RTTrap);
        when 0x6    exception = ExceptionSyndrome(Exception_CP14DTTrap);
        when 0x7    exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        when 0x8    exception = ExceptionSyndrome(Exception_FPIDTrap);
        when 0xC    exception = ExceptionSyndrome(Exception_CP14RRTTrap);
        otherwise   Unreachable();

    bits(20) iss = Zeros();

    if exception.exceptype IN {Exception_FPIDTrap, Exception_CP14RTTrap, Exception_CP15RTTrap} then
        // Trapped MRC/MCR, VMRS on FPSID
        if exception.exceptype != Exception_FPIDTrap then    // When trap is not for VMRS
            iss<19:17> = instr<7:5>;           // opc2
            iss<16:14> = instr<23:21>;         // opc1
            iss<13:10> = instr<19:16>;         // CRn
            iss<4:1>   = instr<3:0>;           // CRm
        else
            iss<19:17> = '000';
            iss<16:14> = '111';
            iss<13:10> = instr<19:16>;         // reg
            iss<4:1>   = '0000';

        if instr<20> == '1' && instr<15:12> == '1111' then    // MRC, Rt==15
            iss<9:5> = '11111';
        elsif instr<20> == '0' && instr<15:12> == '1111' then // MCR, Rt==15
            iss<9:5> = bits(5) UNKNOWN;
        else
            iss<9:5> = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>;
    elsif exception.exceptype IN {Exception_CP14RRTTrap, Exception_AdvSIMDFPAccessTrap,
Exception_CP15RRTTrap} then
        // Trapped MRRC/MCRR, VMRS/VMSR
        iss<19:16> = instr<7:4>;          // opc1
        if instr<19:16> == '1111' then    // Rt2==15
            iss<14:10> = bits(5) UNKNOWN;
        else
            iss<14:10> = LookUpRIndex(UInt(instr<19:16>), PSTATE.M)<4:0>;

        if instr<15:12> == '1111' then    // Rt==15
            iss<9:5> = bits(5) UNKNOWN;
        else
            iss<9:5> = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>;
        iss<4:1>   = instr<3:0>;         // CRm
    elsif exception.exceptype == Exception_CP14DTTrap then
        // Trapped LDC/STC
        iss<19:12> = instr<7:0>;         // imm8
        iss<4>     = instr<23>;          // U
        iss<2:1>   = instr<24,21>;       // P,W
        if instr<19:16> == '1111' then   // Rn==15, LDC(Literal addressing)/STC
            iss<9:5> = bits(5) UNKNOWN;
            iss<3>   = '1';
    elsif exception.exceptype == Exception_Uncategorized then
        // Trapped for unknown reason
        iss<9:5> = LookUpRIndex(UInt(instr<19:16>), PSTATE.M)<4:0>; // Rn
        iss<3>   = '0';

    iss<0> = instr<20>;                  // Direction

    exception.syndrome<24:20> = ConditionSyndrome();
    exception.syndrome<19:0>  = iss;

    return exception;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch64/exceptions/traps/AArch64.AdvSIMDFPAccessTrap

```
// AArch64.AdvSIMDFPAccessTrap()
// ============================
// Trapped access to Advanced SIMD or FP registers due to CPACR[].

AArch64.AdvSIMDFPAccessTrap(bits(2) target_el)
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    route_to_el2 = (target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1');

    if route_to_el2 then
        exception = ExceptionSyndrome(Exception_Uncategorized);
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        exception.syndrome<24:20> = ConditionSyndrome();
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

    return;
```

aarch64/exceptions/traps/AArch64.CheckCP15InstrCoarseTraps

```
// AArch64.CheckCP15InstrCoarseTraps()
// ===================================
// Check for coarse-grained AArch32 CP15 traps in HSTR_EL2 and HCR_EL2.

boolean AArch64.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)

    // Check for coarse-grained Hyp traps
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        // Check for MCR, MRC, MCRR and MRRC disabled by HSTR_EL2<CRn/CRm>
        major = if nreg == 1 then CRn else CRm;
        if !IsInHost() && !(major IN {4,14}) && HSTR_EL2<major> == '1' then
            return TRUE;

        // Check for MRC and MCR disabled by HCR_EL2.TIDCP
        if (HCR_EL2.TIDCP == '1' && nreg == 1 &&
            ((CRn == 9  && CRm IN {0,1,2,    5,6,7,8   }) ||
             (CRn == 10 && CRm IN {0,1,    4,      8   }) ||
             (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}))) then
            return TRUE;

    return FALSE;
```

aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDEnabled

```
// AArch64.CheckFPAdvSIMDEnabled()
// ===============================
// Check against CPACR[]

AArch64.CheckFPAdvSIMDEnabled()
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check if access disabled in CPACR_EL1
        case CPACR_EL1.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);

    AArch64.CheckFPAdvSIMDTrap();               // Also check against CPTR_EL2 and CPTR_EL3
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDTrap

```
// AArch64.CheckFPAdvSIMDTrap()
// ============================
// Check against CPTR_EL2 and CPTR_EL3.

AArch64.CheckFPAdvSIMDTrap()
    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        // Check if access disabled in CPTR_EL2
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            case CPTR_EL2.FPEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
        else
            if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);

    if HaveEL(EL3) then
        // Check if access disabled in CPTR_EL3
        if CPTR_EL3.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL3);

    return;
```

aarch64/exceptions/traps/AArch64.CheckForERetTrap

```
// AArch64.CheckForERetTrap()
// ==========================
// Check for trap on ERET, ERETAA, ERETAB instruction

AArch64.CheckForERetTrap(boolean eret_with_pac, boolean pac_uses_key_a)

    route_to_el2 = FALSE;
    // Non-secure EL1 execution of ERET, ERETAA, ERETAB when either HCR_EL2.NV or HFGITR_EL2.ERET is
set,
    // is trapped to EL2
    route_to_el2 = PSTATE.EL == EL1 && EL2Enabled() &&                    ( (HaveNVExt() && HCR_EL2.NV
== '1') ||                        (HaveFGTExt() && HCR_EL2.<E2H, TGE> != '11' &&
(!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.ERET == '1') );
    if route_to_el2 then
        ExceptionRecord exception;
        bits(64) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x0;
        exception = ExceptionSyndrome(Exception_ERetTrap);
        if !eret_with_pac then                              // ERET
            exception.syndrome<1> = '0';
            exception.syndrome<0> = '0';                    // RES0
        else
            exception.syndrome<1> = '1';
            if pac_uses_key_a then                          // ERETAA
                exception.syndrome<0> = '0';
            else    // ERETAB
                exception.syndrome<0> = '1';
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/traps/AArch64.CheckForSMCUndefOrTrap

```
// AArch64.CheckForSMCUndefOrTrap()
// ================================
// Check for UNDEFINED or trap on SMC instruction

AArch64.CheckForSMCUndefOrTrap(bits(16) imm)
    route_to_el2 = PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.TSC == '1';
    if PSTATE.EL == EL0 then UNDEFINED;
    if !HaveEL(EL3) then
        if PSTATE.EL == EL1 && EL2Enabled() then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            if HaveNVExt() && HCR_EL2.NV == '1' && HCR_EL2.TSC == '1' then
                route_to_el2 = TRUE;
            else
                UNDEFINED;
        else
            UNDEFINED;
    else
        route_to_el2 = PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.TSC == '1';
    if route_to_el2 then
        bits(64) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x0;
        exception = ExceptionSyndrome(Exception_MonitorCall);
        exception.syndrome<15:0> = imm;
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/traps/AArch64.CheckForSVCTrap

```
// AArch64.CheckForSVCTrap()
// =========================
// Check for trap on SVC instruction

AArch64.CheckForSVCTrap(bits(16) immediate)
    if HaveFGTExt() then
        route_to_el2 = FALSE;
        if PSTATE.EL == EL0 then
            route_to_el2 = !ELUsingAArch32(EL0) && !ELUsingAArch32(EL1)  && EL2Enabled() &&
HFGITR_EL2.SVC_EL0 == '1' &&                          (HCR_EL2.<E2H, TGE> != '11' && (!HaveEL(EL3) ||
SCR_EL3.FGTEn == '1'));

        elsif PSTATE.EL == EL1 then
        route_to_el2 = !ELUsingAArch32(EL1) && EL2Enabled() && HFGITR_EL2.SVC_EL1 == '1' &&
(HCR_EL2.<E2H, TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1'));

        if route_to_el2 then
            exception = ExceptionSyndrome(Exception_SupervisorCall);
            exception.syndrome<15:0> = immediate;
            bits(64) preferred_exception_return = ThisInstrAddr();
            vect_offset = 0x0;

            AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/traps/AArch64.CheckForWFxTrap

```
// AArch64.CheckForWFxTrap()
// =========================
// Check for trap on WFE or WFI instruction

AArch64.CheckForWFxTrap(bits(2) target_el, boolean is_wfe)
    assert HaveEL(target_el);

    case target_el of
        when EL1 trap = (if is_wfe then SCTLR[].nTWE else SCTLR[].nTWI) == '0';
        when EL2 trap = (if is_wfe then HCR_EL2.TWE else HCR_EL2.TWI) == '1';
        when EL3 trap = (if is_wfe then SCR_EL3.TWE else SCR_EL3.TWI) == '1';
    if trap then
        AArch64.WFxTrap(target_el, is_wfe);
```

aarch64/exceptions/traps/AArch64.CheckIllegalState

```
// AArch64.CheckIllegalState()
// ===========================
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

AArch64.CheckIllegalState()
    if PSTATE.IL == '1' then
        route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        bits(64) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x0;

        exception = ExceptionSyndrome(Exception_IllegalState);

        if UInt(PSTATE.EL) > UInt(EL1) then
            AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
        elsif route_to_el2 then
            AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
        else
            AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/traps/AArch64.MonitorModeTrap

```
// AArch64.MonitorModeTrap()
// =========================
// Trapped use of Monitor mode features in a Secure EL1 AArch32 mode

AArch64.MonitorModeTrap()
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_Uncategorized);

    if IsSecureEL2Enabled() then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/traps/AArch64.SystemAccessTrap

```
// AArch64.SystemAccessTrap()
// ==========================
// Trapped access to AArch64 system register or system instruction.

AArch64.SystemAccessTrap(bits(2) target_el, integer ec)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = AArch64.SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/traps/AArch64.SystemAccessTrapSyndrome

```
// AArch64.SystemAccessTrapSyndrome()
// ==================================
// Returns the syndrome information for traps on AArch64 MSR/MRS instructions.

ExceptionRecord AArch64.SystemAccessTrapSyndrome(bits(32) instr, integer ec)
    ExceptionRecord exception;
    case ec of
        when 0x0                                                    // Trapped access due to unknown
reason.
            exception = ExceptionSyndrome(Exception_Uncategorized);
        when 0x7                                                    // Trapped access to SVE, Advance
SIMD&FP system register.
            exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
            exception.syndrome<24:20> = ConditionSyndrome();
        when 0x18                                                   // Trapped access to system
register or system instruction.
            exception = ExceptionSyndrome(Exception_SystemRegisterTrap);
            instr = ThisInstr();
            exception.syndrome<21:20> = instr<20:19>;          // Op0
            exception.syndrome<19:17> = instr<7:5>;            // Op2
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            exception.syndrome<16:14> = instr<18:16>;          // Op1
            exception.syndrome<13:10> = instr<15:12>;          // CRn
            exception.syndrome<9:5>   = instr<4:0>;            // Rt
            exception.syndrome<4:1>   = instr<11:8>;           // CRm
            exception.syndrome<0>     = instr<21>;             // Direction
        when 0x19                                                  // Trapped access to SVE System
register
            exception = ExceptionSyndrome(Exception_SVEAccessTrap);
        otherwise
            Unreachable();

    return exception;
```

aarch64/exceptions/traps/AArch64.UndefinedFault

```
// AArch64.UndefinedFault()
// ========================

AArch64.UndefinedFault()

    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_Uncategorized);

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/traps/AArch64.WFxTrap

```
// AArch64.WFxTrap()
// =================

AArch64.WFxTrap(bits(2) target_el, boolean is_wfe)
    assert UInt(target_el) > UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_WFxTrap);
    exception.syndrome<24:20> = ConditionSyndrome();
    exception.syndrome<0> = if is_wfe then '1' else '0';

    if target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/traps/CheckFPAdvSIMDEnabled64

```
// CheckFPAdvSIMDEnabled64()
// =========================
// AArch64 instruction wrapper

CheckFPAdvSIMDEnabled64()
    AArch64.CheckFPAdvSIMDEnabled();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

J1.1.3 aarch64/functions

This section includes the following pseudocode functions:

- *aarch64/functions/aborts/AArch64.CreateFaultRecord* on page J1-7542.
- *aarch64/functions/aborts/AArch64.FaultSyndrome* on page J1-7542.
- *aarch64/functions/exclusive/AArch64.ExclusiveMonitorsPass* on page J1-7543.
- *aarch64/functions/exclusive/AArch64.IsExclusiveVA* on page J1-7543.
- *aarch64/functions/exclusive/AArch64.MarkExclusiveVA* on page J1-7543.
- *aarch64/functions/exclusive/AArch64.SetExclusiveMonitors* on page J1-7543.
- *aarch64/functions/fusedrstep/FPRSqrtStepFused* on page J1-7544.
- *aarch64/functions/fusedrstep/FPRecipStepFused* on page J1-7544.
- *aarch64/functions/memory/AArch64.AccessIsTagChecked* on page J1-7545.
- *aarch64/functions/memory/AArch64.AddressWithAllocationTag* on page J1-7545.
- *aarch64/functions/memory/AArch64.AllocationTagFromAddress* on page J1-7546.
- *aarch64/functions/memory/AArch64.CheckAlignment* on page J1-7546.
- *aarch64/functions/memory/AArch64.CheckTag* on page J1-7546.
- *aarch64/functions/memory/AArch64.MemSingle* on page J1-7546.
- *aarch64/functions/memory/AArch64.MemTag* on page J1-7547.
- *aarch64/functions/memory/AArch64.PhysicalTag* on page J1-7548.
- *aarch64/functions/memory/AArch64.TranslateAddressForAtomicAccess* on page J1-7548.
- *aarch64/functions/memory/CheckSPAlignment* on page J1-7549.
- *aarch64/functions/memory/IsBlockDescriptorNTBitValid* on page J1-7549.
- *aarch64/functions/memory/IsTagCheckedInstruction* on page J1-7549.
- *aarch64/functions/memory/Mem* on page J1-7549.
- *aarch64/functions/memory/MemAtomic* on page J1-7550.
- *aarch64/functions/memory/MemAtomicCompareAndSwap* on page J1-7551.
- *aarch64/functions/memory/NVMem* on page J1-7551.
- *aarch64/functions/memory/SetTagCheckedInstruction* on page J1-7552.
- *aarch64/functions/memory/_MemTag* on page J1-7552.
- *aarch64/functions/pac/addpac/AddPAC* on page J1-7552.
- *aarch64/functions/pac/addpacda/AddPACDA* on page J1-7553.
- *aarch64/functions/pac/addpacdb/AddPACDB* on page J1-7554.
- *aarch64/functions/pac/addpacga/AddPACGA* on page J1-7555.
- *aarch64/functions/pac/addpacia/AddPACIA* on page J1-7555.
- *aarch64/functions/pac/addpacib/AddPACIB* on page J1-7556.
- *aarch64/functions/pac/auth/AArch64.PACFailException* on page J1-7556.
- *aarch64/functions/pac/auth/Auth* on page J1-7557.
- *aarch64/functions/pac/authda/AuthDA* on page J1-7558.
- *aarch64/functions/pac/authdb/AuthDB* on page J1-7558.
- *aarch64/functions/pac/authia/AuthIA* on page J1-7559.
- *aarch64/functions/pac/authib/AuthIB* on page J1-7560.
- *aarch64/functions/pac/calcbottompacbit/CalculateBottomPACBit* on page J1-7560.
- *aarch64/functions/pac/calculatetbi/CalculateTBI* on page J1-7561.
- *aarch64/functions/pac/computepac/ComputePAC* on page J1-7561.
- *aarch64/functions/pac/computepac/PACCellInvShuffle* on page J1-7562.
- *aarch64/functions/pac/computepac/PACCellShuffle* on page J1-7562.
- *aarch64/functions/pac/computepac/PACInvSub* on page J1-7563.
- *aarch64/functions/pac/computepac/PACMult* on page J1-7563.
- *aarch64/functions/pac/computepac/PACSub* on page J1-7564.
- *aarch64/functions/pac/computepac/RotCell* on page J1-7564.
- *aarch64/functions/pac/computepac/TweakCellInvRot* on page J1-7564.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *aarch64/functions/pac/computepac/TweakCellRot* on page J1-7565.
- *aarch64/functions/pac/computepac/TweakInvShuffle* on page J1-7565.
- *aarch64/functions/pac/computepac/TweakShuffle* on page J1-7565.
- *aarch64/functions/pac/pac/HaveEnhancedPAC* on page J1-7566.
- *aarch64/functions/pac/pac/HaveEnhancedPAC2* on page J1-7566.
- *aarch64/functions/pac/pac/HaveFPAC* on page J1-7566.
- *aarch64/functions/pac/pac/HaveFPACCombined* on page J1-7566.
- *aarch64/functions/pac/pac/HavePACExt* on page J1-7566.
- *aarch64/functions/pac/pac/PtrHasUpperAndLowerAddRanges* on page J1-7566.
- *aarch64/functions/pac/strip/Strip* on page J1-7566.
- *aarch64/functions/pac/trappacuse/TrapPACUse* on page J1-7567.
- *aarch64/functions/ras/AArch64.ESBOperation* on page J1-7567.
- *aarch64/functions/ras/AArch64.PhysicalSErrorSyndrome* on page J1-7568.
- *aarch64/functions/ras/AArch64.ReportDeferredSError* on page J1-7568.
- *aarch64/functions/ras/AArch64.vESBOperation* on page J1-7568.
- *aarch64/functions/registers/AArch64.MaybeZeroRegisterUppers* on page J1-7568.
- *aarch64/functions/registers/AArch64.ResetGeneralRegisters* on page J1-7569.
- *aarch64/functions/registers/AArch64.ResetSIMDFPRegisters* on page J1-7569.
- *aarch64/functions/registers/AArch64.ResetSpecialRegisters* on page J1-7569.
- *aarch64/functions/registers/AArch64.ResetSystemRegisters* on page J1-7570.
- *aarch64/functions/registers/PC* on page J1-7570.
- *aarch64/functions/registers/SP* on page J1-7570.
- *aarch64/functions/registers/V* on page J1-7570.
- *aarch64/functions/registers/Vpart* on page J1-7571.
- *aarch64/functions/registers/X* on page J1-7571.
- *aarch64/functions/sve/AArch32.IsFPEnabled* on page J1-7572.
- *aarch64/functions/sve/AArch64.IsFPEnabled* on page J1-7572.
- *aarch64/functions/sve/CeilPow2* on page J1-7573.
- *aarch64/functions/sve/CheckSVEEnabled* on page J1-7573.
- *aarch64/functions/sve/DecodePredCount* on page J1-7574.
- *aarch64/functions/sve/ElemFFR* on page J1-7574.
- *aarch64/functions/sve/ElemP* on page J1-7574.
- *aarch64/functions/sve/FFR* on page J1-7575.
- *aarch64/functions/sve/FPCompareNE* on page J1-7575.
- *aarch64/functions/sve/FPCompareUN* on page J1-7575.
- *aarch64/functions/sve/FPConvertSVE* on page J1-7575.
- *aarch64/functions/sve/FPExpA* on page J1-7576.
- *aarch64/functions/sve/FPExpCoefficient* on page J1-7576.
- *aarch64/functions/sve/FPMinNormal* on page J1-7579.
- *aarch64/functions/sve/FPOne* on page J1-7579.
- *aarch64/functions/sve/FPPointFive* on page J1-7579.
- *aarch64/functions/sve/FPProcess* on page J1-7579.
- *aarch64/functions/sve/FPScale* on page J1-7580.
- *aarch64/functions/sve/FPTrigMAdd* on page J1-7580.
- *aarch64/functions/sve/FPTrigMAddCoefficient* on page J1-7580.
- *aarch64/functions/sve/FPTrigSMul* on page J1-7581.
- *aarch64/functions/sve/FPTrigSSel* on page J1-7581.
- *aarch64/functions/sve/FirstActive* on page J1-7582.
- *aarch64/functions/sve/FloorPow2* on page J1-7582.
- *aarch64/functions/sve/HaveSVE* on page J1-7582.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *aarch64/functions/sve/HaveSVEFP32MatMulExt* on page J1-7582.
- *aarch64/functions/sve/HaveSVEFP64MatMulExt* on page J1-7582.
- *aarch64/functions/sve/ImplementedSVEVectorLength* on page J1-7582.
- *aarch64/functions/sve/IsEven* on page J1-7583.
- *aarch64/functions/sve/IsFPEnabled* on page J1-7583.
- *aarch64/functions/sve/IsSVEEnabled* on page J1-7583.
- *aarch64/functions/sve/LastActive* on page J1-7584.
- *aarch64/functions/sve/LastActiveElement* on page J1-7584.
- *aarch64/functions/sve/MAX_PL* on page J1-7584.
- *aarch64/functions/sve/MAX_VL* on page J1-7584.
- *aarch64/functions/sve/MaybeZeroSVEUppers* on page J1-7584.
- *aarch64/functions/sve/MemNF* on page J1-7585.
- *aarch64/functions/sve/MemSingleNF* on page J1-7585.
- *aarch64/functions/sve/NoneActive* on page J1-7586.
- *aarch64/functions/sve/P* on page J1-7586.
- *aarch64/functions/sve/PL* on page J1-7586.
- *aarch64/functions/sve/PredTest* on page J1-7586.
- *aarch64/functions/sve/ReducePredicated* on page J1-7587.
- *aarch64/functions/sve/Reverse* on page J1-7587.
- *aarch64/functions/sve/SVEAccessTrap* on page J1-7587.
- *aarch64/functions/sve/SVECmp* on page J1-7587.
- *aarch64/functions/sve/SVEMoveMaskPreferred* on page J1-7588.
- *aarch64/functions/sve/System* on page J1-7588.
- *aarch64/functions/sve/VL* on page J1-7588.
- *aarch64/functions/sve/Z* on page J1-7589.
- *aarch64/functions/sysregisters/CNTKCTL* on page J1-7589.
- *aarch64/functions/sysregisters/CNTKCTLType* on page J1-7589.
- *aarch64/functions/sysregisters/CPACR* on page J1-7589.
- *aarch64/functions/sysregisters/CPACRType* on page J1-7590.
- *aarch64/functions/sysregisters/ELR* on page J1-7590.
- *aarch64/functions/sysregisters/ESR* on page J1-7590.
- *aarch64/functions/sysregisters/ESRType* on page J1-7591.
- *aarch64/functions/sysregisters/FAR* on page J1-7591.
- *aarch64/functions/sysregisters/MAIR* on page J1-7591.
- *aarch64/functions/sysregisters/MAIRType* on page J1-7592.
- *aarch64/functions/sysregisters/SCTLR* on page J1-7592.
- *aarch64/functions/sysregisters/SCTLRType* on page J1-7592.
- *aarch64/functions/sysregisters/VBAR* on page J1-7592.
- *aarch64/functions/system/AArch64.AllocationTagAccessIsEnabled* on page J1-7593.
- *aarch64/functions/system/AArch64.CheckSystemAccess* on page J1-7593.
- *aarch64/functions/system/AArch64.ChooseNonExcludedTag* on page J1-7593.
- *aarch64/functions/system/AArch64.ExecutingATS1xPInstr* on page J1-7593.
- *aarch64/functions/system/AArch64.ExecutingBROrBLROrRetInstr* on page J1-7594.
- *aarch64/functions/system/AArch64.ExecutingBTIInstr* on page J1-7594.
- *aarch64/functions/system/AArch64.ExecutingERETInstr* on page J1-7594.
- *aarch64/functions/system/AArch64.GetMinELSecurityState* on page J1-7594.
- *aarch64/functions/system/AArch64.NextRandomTagBit* on page J1-7595.
- *aarch64/functions/system/AArch64.RandomTag* on page J1-7595.
- *aarch64/functions/system/AArch64.SysInstr* on page J1-7595.
- *aarch64/functions/system/AArch64.SysInstrWithResult* on page J1-7595.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *aarch64/functions/system/AArch64.SysRegRead* on page J1-7595.
- *aarch64/functions/system/AArch64.SysRegWrite* on page J1-7595.
- *aarch64/functions/system/BTypeCompatible* on page J1-7596.
- *aarch64/functions/system/BTypeCompatible_BTI* on page J1-7596.
- *aarch64/functions/system/BTypeCompatible_PACIXSP* on page J1-7596.
- *aarch64/functions/system/BTypeNext* on page J1-7596.
- *aarch64/functions/system/InGuardedPage* on page J1-7596.
- *aarch64/functions/system/SetBTypeCompatible* on page J1-7596.
- *aarch64/functions/system/SetBTypeNext* on page J1-7596.
- *aarch64/functions/system/_ChooseRandomNonExcludedTag* on page J1-7597.

aarch64/functions/aborts/AArch64.CreateFaultRecord

```
// AArch64.CreateFaultRecord()
// ===========================

FaultRecord AArch64.CreateFaultRecord(Fault statuscode, bits(52) ipaddress, boolean NS,
                                      integer level, AccType acctype, boolean write, bit extflag,
                                      bits(2) errortype, boolean secondstage, boolean s2fs1walk)

    FaultRecord fault;
    fault.statuscode = statuscode;
    fault.domain = bits(4) UNKNOWN;         // Not used from AArch64
    fault.debugmoe = bits(4) UNKNOWN;       // Not used from AArch64
    fault.errortype = errortype;
    fault.ipaddress.NS = if NS then '1' else '0';
    fault.ipaddress.address = ipaddress;
    fault.level = level;
    fault.acctype = acctype;
    fault.write = write;
    fault.extflag = extflag;
    fault.secondstage = secondstage;
    fault.s2fs1walk = s2fs1walk;

    return fault;
```

aarch64/functions/aborts/AArch64.FaultSyndrome

```
// AArch64.FaultSyndrome()
// =======================
// Creates an exception syndrome value for Abort and Watchpoint exceptions taken to
// an Exception Level using AArch64.

bits(25) AArch64.FaultSyndrome(boolean d_side, FaultRecord fault)
    assert fault.statuscode != Fault_None;

    bits(25) iss = Zeros();
    if HaveRASExt() && IsExternalSyncAbort(fault) then iss<12:11> = fault.errortype; // SET
    if d_side then
        if IsSecondStage(fault) && !fault.s2fs1walk then iss<24:14> = LSInstructionSyndrome();
        if HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER then
            iss<13> = '1';   // Value of '1' indicates fault is generated by use of VNCR_EL2
        if fault.acctype IN {AccType_DC, AccType_DC_UNPRIV, AccType_IC, AccType_AT} then
            iss<8> = '1';  iss<6> = '1';
        else
            iss<6> = if fault.write then '1' else '0';
    if IsExternalAbort(fault) then iss<9> = fault.extflag;
    iss<7> = if fault.s2fs1walk then '1' else '0';
    iss<5:0> = EncodeLDFSC(fault.statuscode, fault.level);

    return iss;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch64/functions/exclusive/AArch64.ExclusiveMonitorsPass

```
// AArch64.ExclusiveMonitorsPass()
// ==============================

// Return TRUE if the Exclusives monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

boolean AArch64.ExclusiveMonitorsPass(bits(64) address, integer size)

    // It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
    // before or after the check on the local Exclusives monitor. As a result a failure
    // of the local monitor can occur on some implementations even if the memory
    // access would give an memory abort.

    acctype = AccType_ATOMIC;
    iswrite = TRUE;

    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);

    passed = AArch64.IsExclusiveVA(address, ProcessorID(), size);
    if !passed then
        return FALSE;

    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
    ClearExclusiveLocal(ProcessorID());

    if passed then
        if memaddrdesc.memattrs.shareable then
            passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    return passed;
```

aarch64/functions/exclusive/AArch64.IsExclusiveVA

```
// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual
// address region of size bytes starting at address.
//
// It is permitted (but not required) for this function to return FALSE and
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
//
// It is always safe to return TRUE which will check the physical address only.
boolean AArch64.IsExclusiveVA(bits(64) address, integer processorid, integer size);
```

aarch64/functions/exclusive/AArch64.MarkExclusiveVA

```
// Optionally record an exclusive access to the virtual address region of size bytes
// starting at address for processorid.
AArch64.MarkExclusiveVA(bits(64) address, integer processorid, integer size);
```

aarch64/functions/exclusive/AArch64.SetExclusiveMonitors

```
// AArch64.SetExclusiveMonitors()
// ==============================

// Sets the Exclusives monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.

AArch64.SetExclusiveMonitors(bits(64) address, integer size)
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    acctype = AccType_ATOMIC;
    iswrite = FALSE;
    aligned = (address == Align(address, size));

    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;

    if memaddrdesc.memattrs.shareable then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

    AArch64.MarkExclusiveVA(address, ProcessorID(), size);
```

aarch64/functions/fusedrstep/FPRSqrtStepFused

```
// FPRSqrtStepFused()
// ==================

bits(N) FPRSqrtStepFused(bits(N) op1, bits(N) op2)
    assert N IN {16, 32, 64};
    bits(N) result;
    op1 = FPNeg(op1);
    (type1,sign1,value1) = FPUnpack(op1, FPCR);
    (type2,sign2,value2) = FPUnpack(op2, FPCR);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, FPCR);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPOnePointFive('0');
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        else
            // Fully fused multiply-add and halve
            result_value = (3.0 + (value1 * value2)) / 2.0;
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                sign = if FPRoundingMode(FPCR) == FPRounding_NEGINF then '1' else '0';
                result = FPZero(sign);
            else
                result = FPRound(result_value, FPCR);
    return result;
```

aarch64/functions/fusedrstep/FPRecipStepFused

```
// FPRecipStepFused()
// ==================

bits(N) FPRecipStepFused(bits(N) op1, bits(N) op2)
    assert N IN {16, 32, 64};
    bits(N) result;
    op1 = FPNeg(op1);
    (type1,sign1,value1) = FPUnpack(op1, FPCR);
    (type2,sign2,value2) = FPUnpack(op2, FPCR);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, FPCR);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo('0');
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        else
            // Fully fused multiply-add
            result_value = 2.0 + (value1 * value2);
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                sign = if FPRoundingMode(FPCR) == FPRounding_NEGINF then '1' else '0';
                result = FPZero(sign);
            else
                result = FPRound(result_value, FPCR);
    return result;
```

aarch64/functions/memory/AArch64.AccessIsTagChecked

```
// AArch64.AccessIsTagChecked()
// ============================
// TRUE if a given access is tag-checked, FALSE otherwise.

boolean AArch64.AccessIsTagChecked(bits(64) vaddr, AccType acctype)
    if PSTATE.M<4> == '1' then return FALSE;

    if EffectiveTBI(vaddr, FALSE, PSTATE.EL) == '0' then
        return FALSE;

    if EffectiveTCMA(vaddr, PSTATE.EL) == '1' && (vaddr<59:55> == '00000' || vaddr<59:55> == '11111')
then
        return FALSE;

    if !AArch64.AllocationTagAccessIsEnabled() then
        return FALSE;

    if acctype IN {AccType_IFETCH, AccType_PTW} then
        return FALSE;

    if acctype == AccType_NV2REGISTER then
        return FALSE;

    if PSTATE.TCO=='1' then
        return FALSE;

    if !IsTagCheckedInstruction() then
        return FALSE;

    return TRUE;
```

aarch64/functions/memory/AArch64.AddressWithAllocationTag

```
// AArch64.AddressWithAllocationTag()
// ==================================
// Generate a 64-bit value containing a Logical Address Tag from a 64-bit
// virtual address and an Allocation Tag.
// If the extension is disabled, treats the Allocation Tag as '0000'.

bits(64) AArch64.AddressWithAllocationTag(bits(64) address, bits(4) allocation_tag)
    bits(64) result = address;
    bits(4) tag;
    if AArch64.AllocationTagAccessIsEnabled() then
        tag = allocation_tag;
    else
        tag = '0000';
    result<59:56> = tag;
    return result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch64/functions/memory/AArch64.AllocationTagFromAddress

```
// AArch64.AllocationTagFromAddress()
// ==================================
// Generate an ALlocation Tag from a 64-bit value containing a Logical Address Tag.

bits(4) AArch64.AllocationTagFromAddress(bits(64) tagged_address)
    return tagged_address<59:56>;
```

aarch64/functions/memory/AArch64.CheckAlignment

```
// AArch64.CheckAlignment()
// ========================

boolean AArch64.CheckAlignment(bits(64) address, integer alignment, AccType acctype,
                               boolean iswrite)

    aligned = (address == Align(address, alignment));
    atomic  = acctype IN { AccType_ATOMIC, AccType_ATOMICRW, AccType_ORDEREDATOMIC,
AccType_ORDEREDATOMICRW };
    ordered = acctype IN { AccType_ORDERED, AccType_ORDEREDRW, AccType_LIMITEDORDERED,
AccType_ORDEREDATOMIC, AccType_ORDEREDATOMICRW };
    vector  = acctype == AccType_VEC;
    if SCTLR[].A == '1' then check = TRUE;
    elsif HaveUA16Ext() then
        check = (UInt(address<0+:4>) + alignment > 16) && ((ordered && SCTLR[].nAA == '0') || atomic);
    else check = atomic || ordered;

    if check && !aligned then
        secondstage = FALSE;
        AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));

    return aligned;
```

aarch64/functions/memory/AArch64.CheckTag

```
// AArch64.CheckTag()
// ==================
// Performs a Tag Check operation for a memory access and returns
// whether the check passed

boolean AArch64.CheckTag(AddressDescriptor memaddrdesc, bits(4) ptag, boolean write)
    if memaddrdesc.memattrs.tagged then
        return ptag == _MemTag[memaddrdesc];
    else
        return TRUE;
```

aarch64/functions/memory/AArch64.MemSingle

```
// AArch64.MemSingle[] - non-assignment (read) form
// ================================================
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean wasaligned]
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    bits(size*8) value;
    iswrite = FALSE;

    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, wasaligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);
    if HaveMTEExt() then
        if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
            bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
            if !AArch64.CheckTag(memaddrdesc, ptag, iswrite) then
                AArch64.TagCheckFail(ZeroExtend(address, 64), iswrite);
    value = _Mem[memaddrdesc, size, accdesc];
    return value;

// AArch64.MemSingle[] - assignment (write) form
// ==============================================
// Perform an atomic, little-endian write of 'size' bytes.

AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean wasaligned] = bits(size*8)
value
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    iswrite = TRUE;

    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, wasaligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareable then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);
    if HaveMTEExt() then
        if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
            bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
            if !AArch64.CheckTag(memaddrdesc, ptag, iswrite) then
                AArch64.TagCheckFail(ZeroExtend(address, 64), iswrite);
    _Mem[memaddrdesc, size, accdesc] = value;
    return;
```

aarch64/functions/memory/AArch64.MemTag

```
// AArch64.MemTag[] - non-assignment (read) form
// ==============================================
// Load an Allocation Tag from memory.

bits(4) AArch64.MemTag[bits(64) address, AccType acctype]
    assert acctype == AccType_NORMAL;
    AddressDescriptor memaddrdesc;
    bits(4) value;
    iswrite = FALSE;

    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, TRUE, TAG_GRANULE);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Return the granule tag if tagging is enabled...
    if AArch64.AllocationTagAccessIsEnabled() && memaddrdesc.memattrs.tagged then
        return _MemTag[memaddrdesc];
    else
        // ...otherwise read tag as zero.
        return '0000';

// AArch64.MemTag[] - assignment (write) form
// ==========================================
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
// Store an Allocation Tag to memory.

AArch64.MemTag[bits(64) address, AccType acctype] = bits(4) value
    assert acctype == AccType_NORMAL;
    AddressDescriptor memaddrdesc;
    iswrite = TRUE;

    // Stores of allocation tags must be aligned
    if address != Align(address, TAG_GRANULE) then
        boolean secondstage = FALSE;
        AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));

    wasaligned = TRUE;
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, wasaligned, TAG_GRANULE);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Memory array access
    if AArch64.AllocationTagAccessIsEnabled() && memaddrdesc.memattrs.tagged then
        _MemTag[memaddrdesc] = value;
```

aarch64/functions/memory/AArch64.PhysicalTag

```
// AArch64.PhysicalTag()
// =====================
// Generate a Physical Tag from a Logical Tag in an address

bits(4) AArch64.PhysicalTag(bits(64) vaddr)
    return vaddr<59:56>;
```

aarch64/functions/memory/AArch64.TranslateAddressForAtomicAccess

```
// AArch64.TranslateAddressForAtomicAccess()
// =========================================
// Performs an alignment check for atomic memory operations.
// Also translates 64-bit Virtual Address into Physical Address.

AddressDescriptor AArch64.TranslateAddressForAtomicAccess(bits(64) address, integer sizeinbits)
    boolean iswrite = FALSE;
    size = sizeinbits DIV 8;

    assert size IN {1, 2, 4, 8, 16};

    aligned = AArch64.CheckAlignment(address, size, AccType_ATOMICRW, iswrite);

    // MMU or MPU lookup
    memaddrdesc = AArch64.TranslateAddress(address, AccType_ATOMICRW, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareable then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    if HaveMTEExt() && AArch64.AccessIsTagChecked(address, AccType_ATOMICRW) then
        bits(4) ptag = AArch64.PhysicalTag(address);
        if !AArch64.CheckTag(memaddrdesc, ptag, iswrite) then
            AArch64.TagCheckFail(address, iswrite);

    return memaddrdesc;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch64/functions/memory/CheckSPAlignment

```
// CheckSPAlignment()
// ==================
// Check correct stack pointer alignment for AArch64 state.

CheckSPAlignment()
    bits(64) sp = SP[];
    if PSTATE.EL == EL0 then
        stack_align_check = (SCTLR[].SA0 != '0');
    else
        stack_align_check = (SCTLR[].SA != '0');

    if stack_align_check && sp != Align(sp, 16) then
        AArch64.SPAlignmentFault();

    return;
```

aarch64/functions/memory/IsBlockDescriptorNTBitValid

```
// If the implementation supports changing the block size without a break-before-make
// approach, then for implementations that have level 1 or 2 support, the nT bit in
// the block descriptor is valid.
boolean IsBlockDescriptorNTBitValid();
```

aarch64/functions/memory/IsTagCheckedInstruction

```
// Returns True if the current instruction uses tag-checked memory access,
// False otherwise.
boolean IsTagCheckedInstruction();
```

aarch64/functions/memory/Mem

```
// Mem[] - non-assignment (read) form
// ==================================
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch64.MemSingle directly.

bits(size*8) Mem[bits(64) address, integer size, AccType acctype]
    assert size IN {1, 2, 4, 8, 16};
    bits(size*8) value;
    boolean iswrite = FALSE;

    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    if size != 16 || !(acctype IN {AccType_VEC, AccType_VECSTREAM}) then
        atomic = aligned;
    else
        // 128-bit SIMD&FP loads are treated as a pair of 64-bit single-copy atomic accesses
        // 64-bit aligned.
        atomic = address == Align(address, 8);

    if !atomic then
        assert size > 1;
        value<7:0> = AArch64.MemSingle[address, 1, acctype, aligned];

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        if !aligned then
            c = ConstrainUnpredictable();
            assert c IN {Constraint_FAULT, Constraint_NONE};
            if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            value<8*i+7:8*i> = AArch64.MemSingle[address+i, 1, acctype, aligned];
    elsif size == 16 && acctype IN {AccType_VEC, AccType_VECSTREAM} then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        value<63:0>   = AArch64.MemSingle[address,   8, acctype, aligned];
        value<127:64> = AArch64.MemSingle[address+8, 8, acctype, aligned];
    else
        value = AArch64.MemSingle[address, size, acctype, aligned];

    if (HaveNV2Ext() && acctype == AccType_NV2REGISTER && SCTLR_EL2.EE == '1') || BigEndian() then
        value = BigEndianReverse(value);
    return value;

// Mem[] - assignment (write) form
// ==============================
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem[bits(64) address, integer size, AccType acctype] = bits(size*8) value
    boolean iswrite = TRUE;

    if (HaveNV2Ext() && acctype == AccType_NV2REGISTER && SCTLR_EL2.EE == '1') || BigEndian() then
        value = BigEndianReverse(value);

    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    if size != 16 || !(acctype IN {AccType_VEC, AccType_VECSTREAM}) then
        atomic = aligned;
    else
        // 128-bit SIMD&FP stores are treated as a pair of 64-bit single-copy atomic accesses
        // 64-bit aligned.
        atomic = address == Align(address, 8);

    if !atomic then
        assert size > 1;
        AArch64.MemSingle[address, 1, acctype, aligned] = value<7:0>;

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        if !aligned then
            c = ConstrainUnpredictable();
            assert c IN {Constraint_FAULT, Constraint_NONE};
            if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            AArch64.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>;
    elsif size == 16 && acctype IN {AccType_VEC, AccType_VECSTREAM} then
        AArch64.MemSingle[address,   8, acctype, aligned] = value<63:0>;
        AArch64.MemSingle[address+8, 8, acctype, aligned] = value<127:64>;
    else
        AArch64.MemSingle[address, size, acctype, aligned] = value;
    return;
```

aarch64/functions/memory/MemAtomic

```
// MemAtomic()
// ===========
// Performs load and store memory operations for a given virtual address.

bits(size) MemAtomic(bits(64) address, MemAtomicOp op, bits(size) value, AccType ldacctype, AccType
stacctype)
    bits(size) newvalue;
    memaddrdesc = AArch64.TranslateAddressForAtomicAccess(address, size);
    ldaccdesc = CreateAccessDescriptor(ldacctype);
    staccdesc = CreateAccessDescriptor(stacctype);

    // All observers in the shareability domain observe the
    // following load and store atomically.
    oldvalue = _Mem[memaddrdesc, size DIV 8, ldaccdesc];
    if BigEndian() then
        oldvalue = BigEndianReverse(oldvalue);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    case op of
        when MemAtomicOp_ADD   newvalue = oldvalue + value;
        when MemAtomicOp_BIC   newvalue = oldvalue AND NOT(value);
        when MemAtomicOp_EOR   newvalue = oldvalue EOR value;
        when MemAtomicOp_ORR   newvalue = oldvalue OR value;
        when MemAtomicOp_SMAX  newvalue = if SInt(oldvalue) > SInt(value) then oldvalue else value;
        when MemAtomicOp_SMIN  newvalue = if SInt(oldvalue) > SInt(value) then value else oldvalue;
        when MemAtomicOp_UMAX  newvalue = if UInt(oldvalue) > UInt(value) then oldvalue else value;
        when MemAtomicOp_UMIN  newvalue = if UInt(oldvalue) > UInt(value) then value else oldvalue;
        when MemAtomicOp_SWP   newvalue = value;

    if BigEndian() then
        newvalue = BigEndianReverse(newvalue);
    _Mem[memaddrdesc, size DIV 8, staccdesc] = newvalue;

    // Load operations return the old (pre-operation) value
    return oldvalue;
```

aarch64/functions/memory/MemAtomicCompareAndSwap

```
// MemAtomicCompareAndSwap()
// =========================
// Compares the value stored at the passed-in memory address against the passed-in expected
// value. If the comparison is successful, the value at the passed-in memory address is swapped
// with the passed-in new_value.

bits(size) MemAtomicCompareAndSwap(bits(64) address, bits(size) expectedvalue,
                                   bits(size) newvalue, AccType ldacctype, AccType stacctype)
    memaddrdesc = AArch64.TranslateAddressForAtomicAccess(address, size);
    ldaccdesc = CreateAccessDescriptor(ldacctype);
    staccdesc = CreateAccessDescriptor(stacctype);

    // All observers in the shareability domain observe the
    // following load and store atomically.
    oldvalue = _Mem[memaddrdesc, size DIV 8, ldaccdesc];
    if BigEndian() then
        oldvalue = BigEndianReverse(oldvalue);

    if oldvalue == expectedvalue then
        if BigEndian() then
            newvalue = BigEndianReverse(newvalue);
        _Mem[memaddrdesc, size DIV 8, staccdesc] = newvalue;
    return oldvalue;
```

aarch64/functions/memory/NVMem

```
// NVMem[] - non-assignment form
// =============================
// This function is the load memory access for the transformed System register read access
// when Enhanced Nested Virtualisation is enabled with HCR_EL2.NV2 = 1.
// The address for the load memory access is calculated using
// the formula SignExtend(VNCR_EL2.BADDR : Offset<11:0>, 64) where,
//  * VNCR_EL2.BADDR holds the base address of the memory location, and
//  * Offset is the unique offset value defined architecturally for each System register that
//    supports transformation of register access to memory access.

bits(64) NVMem[integer offset]
    assert offset > 0;
    bits(64) address = SignExtend(VNCR_EL2.BADDR:offset<11:0>, 64);
    return Mem[address, 8, AccType_NV2REGISTER];

// NVMem[] - assignment form
// =========================
// This function is the store memory access for the transformed System register write access
// when Enhanced Nested Virtualisation is enabled with HCR_EL2.NV2 = 1.
// The address for the store memory access is calculated using
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
// the formula SignExtend(VNCR_EL2.BADDR : Offset<11:0>, 64) where,
//  * VNCR_EL2.BADDR holds the base address of the memory location, and
//  * Offset is the unique offset value defined architecturally for each System register that
//    supports transformation of register access to memory access.

NVMem[integer offset] = bits(64) value
    assert offset > 0;
    bits(64) address = SignExtend(VNCR_EL2.BADDR:offset<11:0>, 64);
    Mem[address, 8, AccType_NV2REGISTER] = value;
    return;
```

aarch64/functions/memory/SetTagCheckedInstruction

```
// Flag the current instruction as using/not using memory tag checking.
SetTagCheckedInstruction(boolean checked);
```

aarch64/functions/memory/_MemTag

```
// This _MemTag[] accessor is the hardware operation which perform a single-copy atomic,
// Allocation Tag granule aligned, memory access from the tag in PA space.
//
// The function address the array using desc.paddress which supplies:
// * A 52-bit physical address
// * A single NS bit to select between Secure and Non-secure parts of the array.
//
// The accdesc descriptor describes the access type: normal, exclusive, ordered, streaming,
// etc and other parameters required to access the physical memory or for setting syndrome
// register in the event of an external abort.
bits(4) _MemTag[AddressDescriptor desc, AccessDescriptor accdesc];

// This _MemTag[] accessor is the hardware operation which perform a single-copy atomic,
// Allocation Tag granule aligned, memory access to the tag in PA space.
//
// The functions address the array using desc.paddress which supplies:
// * A 52-bit physical address
// * A single NS bit to select between Secure and Non-secure parts of the array.
//
// The accdesc descriptor describes the access type: normal, exclusive, ordered, streaming,
// etc and other parameters required to access the physical memory or for setting syndrome
// register in the event of an external abort.
_MemTag[AddressDescriptor desc, AccessDescriptor accdesc] = bits(4) value;
```

aarch64/functions/pac/addpac/AddPAC

```
// AddPAC()
// ========
// Calculates the pointer authentication code for a 64-bit quantity and then
// inserts that into pointer authentication code field of that 64-bit quantity.

bits(64) AddPAC(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data)
    bits(64) PAC;
    bits(64) result;
    bits(64) ext_ptr;
    bits(64) extfield;
    bit selbit;
    boolean tbi = CalculateTBI(ptr, data);
    integer top_bit = if tbi then 55 else 63;

    // If tagged pointers are in use for a regime with two TTBRs, use bit<55> of
    // the pointer to select between upper and lower ranges, and preserve this.
    // This handles the awkward case where there is apparently no correct choice between
    // the upper and lower address range - ie an addr of 1xxxxxxx0... with TBI0=0 and TBI1=1
    // and 0xxxxxxx1 with TBI1=0 and TBI0=1:
    if PtrHasUpperAndLowerAddRanges() then
        assert S1TranslationRegime() IN {EL1, EL2};
        if S1TranslationRegime() == EL1 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            // EL1 translation regime registers
            if data then
                if TCR_EL1.TBI1 == '1' || TCR_EL1.TBI0 == '1' then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            else
                if ((TCR_EL1.TBI1 == '1' && TCR_EL1.TBID1 == '0') ||
                    (TCR_EL1.TBI0 == '1' && TCR_EL1.TBID0 == '0')) then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
        else
            // EL2 translation regime registers
            if data then
                if TCR_EL2.TBI1 == '1' || TCR_EL2.TBI0 == '1' then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            else
                if ((TCR_EL2.TBI1 == '1' && TCR_EL2.TBID1 == '0') ||
                    (TCR_EL2.TBI0 == '1' && TCR_EL2.TBID0 == '0')) then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
    else selbit = if tbi then ptr<55> else ptr<63>;

    integer bottom_PAC_bit = CalculateBottomPACBit(selbit);

    // The pointer authentication code field takes all the available bits in between
    extfield = Replicate(selbit, 64);

    // Compute the pointer authentication code for a ptr with good extension bits
    if tbi then
        ext_ptr = ptr<63:56>:extfield<(56-bottom_PAC_bit)-1:0>:ptr<bottom_PAC_bit-1:0>;
    else
        ext_ptr = extfield<(64-bottom_PAC_bit)-1:0>:ptr<bottom_PAC_bit-1:0>;

    PAC = ComputePAC(ext_ptr, modifier, K<127:64>, K<63:0>);

    // Check if the ptr has good extension bits and corrupt the pointer authentication code if not
    if !IsZero(ptr<top_bit:bottom_PAC_bit>) && !IsOnes(ptr<top_bit:bottom_PAC_bit>) then
        if HaveEnhancedPAC() then
            PAC = 0x0000000000000000<63:0>;
        elsif !HaveEnhancedPAC2() then
            PAC<top_bit-1> = NOT(PAC<top_bit-1>);

    // preserve the determination between upper and lower address at bit<55> and insert PAC
    if !HaveEnhancedPAC2() then
        if tbi then
            result = ptr<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bottom_PAC_bit-1:0>;
        else
            result = PAC<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bottom_PAC_bit-1:0>;
    else
        if tbi then
            result = ptr<63:56>:selbit:(ptr<54:bottom_PAC_bit> EOR
PAC<54:bottom_PAC_bit>):ptr<bottom_PAC_bit-1:0>;
        else
            result = (ptr<63:56> EOR PAC<63:56>):selbit:(ptr<54:bottom_PAC_bit> EOR
                        PAC<54:bottom_PAC_bit>):ptr<bottom_PAC_bit-1:0>;
    return result;
```

aarch64/functions/pac/addpacda/AddPACDA

```
// AddPACDA()
// ==========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of X, Y and the
// APDAKey_EL1.

bits(64) AddPACDA(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APDAKey_EL1;

    APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnDA else SCTLR_EL2.EnDA;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                        (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnDA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else return AddPAC(X, Y, APDAKey_EL1, TRUE);
```

aarch64/functions/pac/addpacdb/AddPACDB

```
// AddPACDB()
// ==========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of X, Y and the
// APDBKey_EL1.

bits(64) AddPACDB(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APDBKey_EL1;

    APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnDB else SCTLR_EL2.EnDB;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                        (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnDB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        when EL3
            Enable = SCTLR_EL3.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else return AddPAC(X, Y, APDBKey_EL1, TRUE);
```

aarch64/functions/pac/addpacga/AddPACGA

```
// AddPACGA()
// ==========
// Returns a 64-bit value where the lower 32 bits are 0, and the upper 32 bits contain
// a 32-bit pointer authentication code which is derived using a cryptographic
// algorithm as a combination of X, Y and the APGAKey_EL1.

bits(64) AddPACGA(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(128) APGAKey_EL1;

    APGAKey_EL1 = APGAKeyHi_EL1<63:0> : APGAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                       (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else return ComputePAC(X, Y, APGAKey_EL1<127:64>, APGAKey_EL1<63:0>)<63:32>:Zeros(32);
```

aarch64/functions/pac/addpacia/AddPACIA

```
// AddPACIA()
// ==========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of X, Y, and the
// APIAKey_EL1.

bits(64) AddPACIA(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APIAKey_EL1;

    APIAKey_EL1 = APIAKeyHi_EL1<63:0>:APIAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnIA else SCTLR_EL2.EnIA;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                       (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        when EL1
            Enable = SCTLR_EL1.EnIA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else return AddPAC(X, Y, APIAKey_EL1, FALSE);
```

aarch64/functions/pac/addpacib/AddPACIB

```
// AddPACIB()
// ==========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of X, Y and the
// APIBKey_EL1.

bits(64) AddPACIB(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APIBKey_EL1;

    APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnIB else SCTLR_EL2.EnIB;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                       (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnIB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else return AddPAC(X, Y, APIBKey_EL1, FALSE);
```

aarch64/functions/pac/auth/AArch64.PACFailException

```
// AArch64.PACFailException()
// ==========================
// Generates a PAC Fail Exception

AArch64.PACFailException(bits(2) syndrome)
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_PACFail);
    exception.syndrome<1:0>   = syndrome;
    exception.syndrome<24:2>  = Zeros();                // RES0

    if UInt(PSTATE.EL) > UInt(EL0) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/functions/pac/auth/Auth

```
// Auth()
// ======
// Restores the upper bits of the address to be all zeros or all ones (based on the
// value of bit[55]) and computes and checks the pointer authentication code. If the
// check passes, then the restored address is returned. If the check fails, the
// second-top and third-top bits of the extension bits in the pointer authentication code
// field are corrupted to ensure that accessing the address will give a translation fault.

bits(64) Auth(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data, bit key_number, boolean
is_combined)
    bits(64) PAC;
    bits(64) result;
    bits(64) original_ptr;
    bits(2) error_code;
    bits(64) extfield;

    // Reconstruct the extension field used of adding the PAC to the pointer
    boolean tbi = CalculateTBI(ptr, data);
    integer bottom_PAC_bit = CalculateBottomPACBit(ptr<55>);
    extfield = Replicate(ptr<55>, 64);

    if tbi then
        original_ptr = ptr<63:56>:extfield<56-bottom_PAC_bit-1:0>:ptr<bottom_PAC_bit-1:0>;
    else
        original_ptr = extfield<64-bottom_PAC_bit-1:0>:ptr<bottom_PAC_bit-1:0>;

    PAC = ComputePAC(original_ptr, modifier, K<127:64>, K<63:0>);
    // Check pointer authentication code
    if tbi then
        if !HaveEnhancedPAC2() then
            if PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit> then
                result = original_ptr;
            else
                error_code = key_number:NOT(key_number);
                result = original_ptr<63:55>:error_code:original_ptr<52:0>;
        else
            result = ptr;
            result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
            if HaveFPACCombined() || (HaveFPAC() && !is_combined) then
                if result<54:bottom_PAC_bit> != Replicate(result<55>, (55-bottom_PAC_bit)) then
                    error_code = (if data then '1' else '0'):key_number;
                    AArch64.PACFailException(error_code);
    else
        if !HaveEnhancedPAC2() then
            if PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit> && PAC<63:56> == ptr<63:56> then
                result = original_ptr;
            else
                error_code = key_number:NOT(key_number);
                result = original_ptr<63>:error_code:original_ptr<60:0>;
        else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            result = ptr;
            result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
            result<63:56> = result<63:56> EOR PAC<63:56>;
            if HaveFPACCombined() || (HaveFPAC() && !is_combined) then
                if result<63:bottom_PAC_bit> != Replicate(result<55>, (64-bottom_PAC_bit)) then
                    error_code = (if data then '1' else '0'):key_number;
                    AArch64.PACFailException(error_code);
    return result;
```

aarch64/functions/pac/authda/AuthDA

```
// AuthDA()
// ========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of X, using the same
// algorithm and key as AddPACDA().

bits(64) AuthDA(bits(64) X, bits(64) Y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APDAKey_EL1;

    APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnDA else SCTLR_EL2.EnDA;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                       (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnDA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else  return Auth(X, Y, APDAKey_EL1, TRUE, '0', is_combined);
```

aarch64/functions/pac/authdb/AuthDB

```
// AuthDB()
// ========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a
// pointer authentication code in the pointer authentication code field bits of X, using
// the same algorithm and key as AddPACDB().

bits(64) AuthDB(bits(64) X, bits(64) Y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APDBKey_EL1;

    APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnDB else SCTLR_EL2.EnDB;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                        (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnDB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else  return Auth(X, Y, APDBKey_EL1, TRUE, '1', is_combined);
```

aarch64/functions/pac/authia/AuthIA

```
// AuthIA()
// ========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of X, using the same
// algorithm and key as AddPACIA().

bits(64) AuthIA(bits(64) X, bits(64) Y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APIAKey_EL1;

    APIAKey_EL1 = APIAKeyHi_EL1<63:0> : APIAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnIA else SCTLR_EL2.EnIA;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                       (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnIA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else  return Auth(X, Y, APIAKey_EL1, FALSE, '0', is_combined);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch64/functions/pac/authib/AuthIB

```
// AuthIB()
// ========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of X, using the same
// algorithm and key as AddPACIB().

bits(64) AuthIB(bits(64) X, bits(64) Y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APIBKey_EL1;

    APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnIB else SCTLR_EL2.EnIB;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                       (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnIB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else  return Auth(X, Y, APIBKey_EL1, FALSE, '1', is_combined);
```

aarch64/functions/pac/calcbottompacbit/CalculateBottomPACBit

```
// CalculateBottomPACBit()
// =======================

integer CalculateBottomPACBit(bit top_bit)
    integer tsz_field;

    if PtrHasUpperAndLowerAddRanges() then
        assert S1TranslationRegime() IN {EL1, EL2};
        if S1TranslationRegime() == EL1 then
            // EL1 translation regime registers
            tsz_field = if top_bit == '1' then UInt(TCR_EL1.T1SZ) else UInt(TCR_EL1.T0SZ);
            using64k = if top_bit == '1' then TCR_EL1.TG1 == '11' else TCR_EL1.TG0 == '01';
        else
            // EL2 translation regime registers
            assert HaveEL(EL2);
            tsz_field = if top_bit == '1' then UInt(TCR_EL2.T1SZ) else UInt(TCR_EL2.T0SZ);
            using64k = if top_bit == '1' then TCR_EL2.TG1 == '11' else TCR_EL2.TG0 == '01';
    else
        tsz_field = if PSTATE.EL == EL2 then UInt(TCR_EL2.T0SZ) else UInt(TCR_EL3.T0SZ);
        using64k = if PSTATE.EL == EL2 then TCR_EL2.TG0 == '01' else TCR_EL3.TG0 == '01';

    max_limit_tsz_field = (if !HaveSmallPageTblExt() then 39 else if using64k then 47 else 48);
    if tsz_field > max_limit_tsz_field then
        // TCR_ELx.TySZ is out of range
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        c = ConstrainUnpredictable();
        assert c IN {Constraint_FORCE, Constraint_NONE};
        if c == Constraint_FORCE then tsz_field = max_limit_tsz_field;
    tszmin = if using64k && VAMax() == 52 then 12 else 16;
    if tsz_field < tszmin then
        c = ConstrainUnpredictable();
        assert c IN {Constraint_FORCE, Constraint_NONE};
        if c == Constraint_FORCE then tsz_field = tszmin;
    return (64-tsz_field);
```

aarch64/functions/pac/calculatetbi/CalculateTBI

```
// CalculateTBI()
// ==============

boolean CalculateTBI(bits(64) ptr, boolean data)
    boolean tbi = FALSE;

    if PtrHasUpperAndLowerAddRanges() then
        assert S1TranslationRegime() IN {EL1, EL2};
        if S1TranslationRegime() == EL1 then
        // EL1 translation regime registers
            if data then
                tbi = if ptr<55> == '1' then TCR_EL1.TBI1 == '1' else TCR_EL1.TBI0 == '1';
            else
                if ptr<55> == '1' then
                    tbi = TCR_EL1.TBI1 == '1' && TCR_EL1.TBID1 == '0';
                else
                    tbi = TCR_EL1.TBI0 == '1' && TCR_EL1.TBID0 == '0';
        else
            // EL2 translation regime registers
            if data then
                tbi = if ptr<55> == '1' then TCR_EL2.TBI1 == '1' else TCR_EL2.TBI0 == '1';
            else
                if ptr<55> == '1' then
                    tbi = TCR_EL2.TBI1 == '1' && TCR_EL2.TBID1 == '0';
                else
                    tbi = TCR_EL2.TBI0 == '1' && TCR_EL2.TBID0 == '0';
    elsif PSTATE.EL == EL2 then
        tbi = if data then TCR_EL2.TBI=='1' else TCR_EL2.TBI=='1' && TCR_EL2.TBID=='0';
    elsif PSTATE.EL == EL3 then
        tbi = if data then TCR_EL3.TBI=='1' else TCR_EL3.TBI=='1' && TCR_EL3.TBID=='0';

    return tbi;
```

aarch64/functions/pac/computepac/ComputePAC

```
array bits(64) RC[0..4];

bits(64) ComputePAC(bits(64) data, bits(64) modifier, bits(64) key0, bits(64) key1)
    bits(64)  workingval;
    bits(64)  runningmod;
    bits(64)  roundkey;
    bits(64)  modk0;
    constant bits(64) Alpha = 0xC0AC29B7C97C50DD<63:0>;

    RC[0] = 0x0000000000000000<63:0>;
    RC[1] = 0x13198A2E03707344<63:0>;
    RC[2] = 0xA4093822299F31D0<63:0>;
    RC[3] = 0x082EFA98EC4E6C89<63:0>;
    RC[4] = 0x452821E638D01377<63:0>;

    modk0 = key0<0>:key0<63:2>:(key0<63> EOR key0<1>);
    runningmod = modifier;
    workingval = data EOR key0;
    for i = 0 to 4
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        roundkey = key1 EOR runningmod;
        workingval  = workingval EOR roundkey;
        workingval = workingval EOR RC[i];
        if i > 0 then
            workingval = PACCellShuffle(workingval);
            workingval = PACMult(workingval);
        workingval = PACSub(workingval);
        runningmod = TweakShuffle(runningmod<63:0>);
    roundkey = modk0 EOR runningmod;
    workingval = workingval EOR roundkey;
    workingval = PACCellShuffle(workingval);
    workingval = PACMult(workingval);
    workingval = PACSub(workingval);
    workingval = PACCellShuffle(workingval);
    workingval = PACMult(workingval);
    workingval = key1 EOR workingval;
    workingval = PACCellInvShuffle(workingval);
    workingval = PACInvSub(workingval);
    workingval = PACMult(workingval);
    workingval = PACCellInvShuffle(workingval);
    workingval = workingval EOR key0;
    workingval = workingval EOR runningmod;
    for i = 0 to 4
        workingval = PACInvSub(workingval);
        if i < 4 then
            workingval = PACMult(workingval);
            workingval = PACCellInvShuffle(workingval);
        runningmod = TweakInvShuffle(runningmod<63:0>);
        roundkey = key1 EOR runningmod;
        workingval = workingval EOR RC[4-i];
        workingval = workingval EOR roundkey;
        workingval = workingval EOR Alpha;
    workingval = workingval EOR modk0;

    return workingval;
```

aarch64/functions/pac/computepac/PACCellInvShuffle

```
// PACCellInvShuffle()
// ===================

bits(64) PACCellInvShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<15:12>;
    outdata<7:4> = indata<27:24>;
    outdata<11:8> = indata<51:48>;
    outdata<15:12> = indata<39:36>;
    outdata<19:16> = indata<59:56>;
    outdata<23:20> = indata<47:44>;
    outdata<27:24> = indata<7:4>;
    outdata<31:28> = indata<19:16>;
    outdata<35:32> = indata<35:32>;
    outdata<39:36> = indata<55:52>;
    outdata<43:40> = indata<31:28>;
    outdata<47:44> = indata<11:8>;
    outdata<51:48> = indata<23:20>;
    outdata<55:52> = indata<3:0>;
    outdata<59:56> = indata<43:40>;
    outdata<63:60> = indata<63:60>;
    return outdata;
```

aarch64/functions/pac/computepac/PACCellShuffle

```
// PACCellShuffle()
// ================
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(64) PACCellShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<55:52>;
    outdata<7:4> = indata<27:24>;
    outdata<11:8> = indata<47:44>;
    outdata<15:12> = indata<3:0>;
    outdata<19:16> = indata<31:28>;
    outdata<23:20> = indata<51:48>;
    outdata<27:24> = indata<7:4>;
    outdata<31:28> = indata<43:40>;
    outdata<35:32> = indata<35:32>;
    outdata<39:36> = indata<15:12>;
    outdata<43:40> = indata<59:56>;
    outdata<47:44> = indata<23:20>;
    outdata<51:48> = indata<11:8>;
    outdata<55:52> = indata<39:36>;
    outdata<59:56> = indata<19:16>;
    outdata<63:60> = indata<63:60>;
    return outdata;
```

aarch64/functions/pac/computepac/PACInvSub

```
// PACInvSub()
// ===========

bits(64) PACInvSub(bits(64) Tinput)
    // This is a 4-bit substitution from the PRINCE-family cipher

    bits(64) Toutput;
    for i = 0 to 15
        case Tinput<4*i+3:4*i> of
            when '0000'  Toutput<4*i+3:4*i> = '0101';
            when '0001'  Toutput<4*i+3:4*i> = '1110';
            when '0010'  Toutput<4*i+3:4*i> = '1101';
            when '0011'  Toutput<4*i+3:4*i> = '1000';
            when '0100'  Toutput<4*i+3:4*i> = '1010';
            when '0101'  Toutput<4*i+3:4*i> = '1011';
            when '0110'  Toutput<4*i+3:4*i> = '0001';
            when '0111'  Toutput<4*i+3:4*i> = '1001';
            when '1000'  Toutput<4*i+3:4*i> = '0010';
            when '1001'  Toutput<4*i+3:4*i> = '0110';
            when '1010'  Toutput<4*i+3:4*i> = '1111';
            when '1011'  Toutput<4*i+3:4*i> = '0000';
            when '1100'  Toutput<4*i+3:4*i> = '0100';
            when '1101'  Toutput<4*i+3:4*i> = '1100';
            when '1110'  Toutput<4*i+3:4*i> = '0111';
            when '1111'  Toutput<4*i+3:4*i> = '0011';
    return Toutput;
```

aarch64/functions/pac/computepac/PACMult

```
// PACMult()
// =========

bits(64) PACMult(bits(64) Sinput)
    bits(4)  t0;
    bits(4)  t1;
    bits(4)  t2;
    bits(4)  t3;
    bits(64) Soutput;

    for i = 0 to 3
        t0<3:0> = RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 1) EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 2);
        t0<3:0> = t0<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
        t1<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1) EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 1);
        t1<3:0> = t1<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 2);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        t2<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 2) EOR RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 1);
        t2<3:0> = t2<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
        t3<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1) EOR RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 2);
        t3<3:0> = t3<3:0> EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 1);
        Soutput<4*i+3:4*i> = t3<3:0>;
        Soutput<4*(i+4)+3:4*(i+4)> = t2<3:0>;
        Soutput<4*(i+8)+3:4*(i+8)> = t1<3:0>;
        Soutput<4*(i+12)+3:4*(i+12)> = t0<3:0>;
    return Soutput;
```

aarch64/functions/pac/computepac/PACSub

```
// PACSub()
// ========

bits(64) PACSub(bits(64) Tinput)
    // This is a 4-bit substitution from the PRINCE-family cipher
    bits(64) Toutput;
    for i = 0 to 15
        case Tinput<4*i+3:4*i> of
            when '0000'  Toutput<4*i+3:4*i> = '1011';
            when '0001'  Toutput<4*i+3:4*i> = '0110';
            when '0010'  Toutput<4*i+3:4*i> = '1000';
            when '0011'  Toutput<4*i+3:4*i> = '1111';
            when '0100'  Toutput<4*i+3:4*i> = '1100';
            when '0101'  Toutput<4*i+3:4*i> = '0000';
            when '0110'  Toutput<4*i+3:4*i> = '1001';
            when '0111'  Toutput<4*i+3:4*i> = '1110';
            when '1000'  Toutput<4*i+3:4*i> = '0011';
            when '1001'  Toutput<4*i+3:4*i> = '0111';
            when '1010'  Toutput<4*i+3:4*i> = '0100';
            when '1011'  Toutput<4*i+3:4*i> = '0101';
            when '1100'  Toutput<4*i+3:4*i> = '1101';
            when '1101'  Toutput<4*i+3:4*i> = '0010';
            when '1110'  Toutput<4*i+3:4*i> = '0001';
            when '1111'  Toutput<4*i+3:4*i> = '1010';
    return Toutput;
```

aarch64/functions/pac/computepac/RotCell

```
// RotCell()
// =========

bits(4) RotCell(bits(4) incell, integer amount)
    bits(8) tmp;
    bits(4) outcell;

    // assert amount>3 || amount<1;
    tmp<7:0> = incell<3:0>:incell<3:0>;
    outcell = tmp<7-amount:4-amount>;
    return outcell;
```

aarch64/functions/pac/computepac/TweakCellInvRot

```
// TweakCellInvRot()
// =================

bits(4) TweakCellInvRot(bits(4)incell)
    bits(4) outcell;
    outcell<3> = incell<2>;
    outcell<2> = incell<1>;
    outcell<1> = incell<0>;
    outcell<0> = incell<0> EOR incell<3>;
    return outcell;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch64/functions/pac/computepac/TweakCellRot

```
// TweakCellRot()
// ==============

bits(4) TweakCellRot(bits(4) incell)
    bits(4) outcell;
    outcell<3> = incell<0> EOR incell<1>;
    outcell<2> = incell<3>;
    outcell<1> = incell<2>;
    outcell<0> = incell<1>;
    return outcell;
```

aarch64/functions/pac/computepac/TweakInvShuffle

```
// TweakInvShuffle()
// =================

bits(64) TweakInvShuffle(bits(64)indata)
    bits(64) outdata;
    outdata<3:0> = TweakCellInvRot(indata<51:48>);
    outdata<7:4> = indata<55:52>;
    outdata<11:8> = indata<23:20>;
    outdata<15:12> = indata<27:24>;
    outdata<19:16> = indata<3:0>;
    outdata<23:20> = indata<7:4>;
    outdata<27:24> = TweakCellInvRot(indata<11:8>);
    outdata<31:28> = indata<15:12>;
    outdata<35:32> = TweakCellInvRot(indata<31:28>);
    outdata<39:36> = TweakCellInvRot(indata<63:60>);
    outdata<43:40> = TweakCellInvRot(indata<59:56>);
    outdata<47:44> = TweakCellInvRot(indata<19:16>);
    outdata<51:48> = indata<35:32>;
    outdata<55:52> = indata<39:36>;
    outdata<59:56> = indata<43:40>;
    outdata<63:60> = TweakCellInvRot(indata<47:44>);
    return outdata;
```

aarch64/functions/pac/computepac/TweakShuffle

```
// TweakShuffle()
// ==============

bits(64) TweakShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<19:16>;
    outdata<7:4> = indata<23:20>;
    outdata<11:8> = TweakCellRot(indata<27:24>);
    outdata<15:12> = indata<31:28>;
    outdata<19:16> = TweakCellRot(indata<47:44>);
    outdata<23:20> = indata<11:8>;
    outdata<27:24> = indata<15:12>;
    outdata<31:28> = TweakCellRot(indata<35:32>);
    outdata<35:32> = indata<51:48>;
    outdata<39:36> = indata<55:52>;
    outdata<43:40> = indata<59:56>;
    outdata<47:44> = TweakCellRot(indata<63:60>);
    outdata<51:48> = TweakCellRot(indata<3:0>);
    outdata<55:52> = indata<7:4>;
    outdata<59:56> = TweakCellRot(indata<43:40>);
    outdata<63:60> = TweakCellRot(indata<39:36>);
    return outdata;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch64/functions/pac/pac/HaveEnhancedPAC

```
// HaveEnhancedPAC()
// =================
// Returns TRUE if support for EnhancedPAC is implemented, FALSE otherwise.

boolean HaveEnhancedPAC()
    return ( HavePACExt()
        && boolean IMPLEMENTATION_DEFINED "Has enhanced PAC functionality" );
```

aarch64/functions/pac/pac/HaveEnhancedPAC2

```
// HaveEnhancedPAC2()
// ==================
// Returns TRUE if support for EnhancedPAC2 is implemented, FALSE otherwise.

boolean HaveEnhancedPAC2()
    return HasArchVersion(ARMv8p6) || (HasArchVersion(ARMv8p3) && boolean IMPLEMENTATION_DEFINED "Has
enhanced PAC 2 functionality");
```

aarch64/functions/pac/pac/HaveFPAC

```
// HaveFPAC()
// ==========
// Returns TRUE if support for FPAC is implemented, FALSE otherwise.

boolean HaveFPAC()
    return HaveEnhancedPAC2() && boolean IMPLEMENTATION_DEFINED "Has FPAC functionality";
```

aarch64/functions/pac/pac/HaveFPACCombined

```
// HaveFPACCombined()
// ==================
// Returns TRUE if support for FPACCombined is implemented, FALSE otherwise.

boolean HaveFPACCombined()
    return HaveFPAC() && boolean IMPLEMENTATION_DEFINED "Has FPAC Combined functionality";
```

aarch64/functions/pac/pac/HavePACExt

```
// HavePACExt()
// ============
// Returns TRUE if support for the PAC extension is implemented, FALSE otherwise.

boolean HavePACExt()
    return HasArchVersion(ARMv8p3);
```

aarch64/functions/pac/pac/PtrHasUpperAndLowerAddRanges

```
// PtrHasUpperAndLowerAddRanges()
// ==============================
// Returns TRUE if the pointer has upper and lower address ranges, FALSE otherwise.

boolean PtrHasUpperAndLowerAddRanges()
    return PSTATE.EL == EL1 || PSTATE.EL == EL0 || (PSTATE.EL == EL2 && HCR_EL2.E2H == '1');
```

aarch64/functions/pac/strip/Strip

```
// Strip()
// =======
// Strip() returns a 64-bit value containing A, but replacing the pointer authentication
// code field bits with the extension of the address bits. This can apply to either
// instructions or data, where, as the use of tagged pointers is distinct, it might be
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
// handled differently.

bits(64) Strip(bits(64) A, boolean data)
    bits(64) original_ptr;
    bits(64) extfield;
    boolean tbi = CalculateTBI(A, data);
    integer bottom_PAC_bit = CalculateBottomPACBit(A<55>);
    extfield = Replicate(A<55>, 64);

    if tbi then
        original_ptr = A<63:56>:extfield< 56-bottom_PAC_bit-1:0>:A<bottom_PAC_bit-1:0>;
    else
        original_ptr = extfield< 64-bottom_PAC_bit-1:0>:A<bottom_PAC_bit-1:0>;

    return original_ptr;
```

aarch64/functions/pac/trappacuse/TrapPACUse

```
// TrapPACUse()
// ============
// Used for the trapping of the pointer authentication functions by higher exception
// levels.

TrapPACUse(bits(2) target_el)
    assert HaveEL(target_el) && target_el !=  EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr();
    ExceptionRecord exception;
    vect_offset = 0;
    exception = ExceptionSyndrome(Exception_PACTrap);
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

aarch64/functions/ras/AArch64.ESBOperation

```
// AArch64.ESBOperation()
// ======================
// Perform the AArch64 ESB operation, either for ESB executed in AArch64 state, or for
// ESB in AArch32 state when SError interrupts are routed to an Exception level using
// AArch64

AArch64.ESBOperation()

    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1';
    route_to_el2 = (EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || HCR_EL2.AMO == '1'));

    target = if route_to_el3 then EL3 elsif route_to_el2 then EL2 else EL1;

    if target == EL1 then
        mask_active = PSTATE.EL IN {EL0, EL1};
    elsif HaveVirtHostExt() && target == EL2 && HCR_EL2.<E2H,TGE> == '11' then
        mask_active = PSTATE.EL IN {EL0, EL2};
    else
        mask_active = PSTATE.EL == target;

    mask_set = (PSTATE.A == '1' && (!HaveDoubleFaultExt() || SCR_EL3.EA == '0' ||
                                    PSTATE.EL != EL3 || SCR_EL3.NMEA == '0'));
    intdis = Halted() || ExternalDebugInterruptsDisabled(target);
    masked = (UInt(target) < UInt(PSTATE.EL)) || intdis || (mask_active && mask_set);

    // Check for a masked Physical SError pending
    if IsPhysicalSErrorPending() && masked then
        // This function might be called for an interworking case, and INTdis is masking
        // the SError interrupt.
        if ELUsingAArch32(S1TranslationRegime()) then
            syndrome32 = AArch32.PhysicalSErrorSyndrome();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            DISR = AArch32.ReportDeferredSError(syndrome32.AET, syndrome32.ExT);
        else
            implicit_esb = FALSE;
            syndrome64 = AArch64.PhysicalSErrorSyndrome(implicit_esb);
            DISR_EL1 = AArch64.ReportDeferredSError(syndrome64)<31:0>;
        ClearPendingPhysicalSError();               // Set ISR_EL1.A to 0

    return;
```

aarch64/functions/ras/AArch64.PhysicalSErrorSyndrome

```
// Return the SError syndrome
bits(25) AArch64.PhysicalSErrorSyndrome(boolean implicit_esb);
```

aarch64/functions/ras/AArch64.ReportDeferredSError

```
// AArch64.ReportDeferredSError()
// ==============================
// Generate deferred SError syndrome

bits(64) AArch64.ReportDeferredSError(bits(25) syndrome)
    bits(64) target;
    target<31>   = '1';              // A
    target<24>   = syndrome<24>;     // IDS
    target<23:0> = syndrome<23:0>;   // ISS
    return target;
```

aarch64/functions/ras/AArch64.vESBOperation

```
// AArch64.vESBOperation()
// =======================
// Perform the AArch64 ESB operation for virtual SError interrupts, either for ESB
// executed in AArch64 state, or for ESB in AArch32 state with EL2 using AArch64 state

AArch64.vESBOperation()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();

    // If physical SError interrupts are routed to EL2, and TGE is not set, then a virtual
    // SError interrupt might be pending
    vSEI_enabled = HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';
    vSEI_pending = vSEI_enabled && HCR_EL2.VSE == '1';
    vintdis      = Halted() || ExternalDebugInterruptsDisabled(EL1);
    vmasked      = vintdis || PSTATE.A == '1';

    // Check for a masked virtual SError pending
    if vSEI_pending && vmasked then
        // This function might be called for the interworking case, and INTdis is masking
        // the virtual SError interrupt.
        if ELUsingAArch32(EL1) then
            VDISR = AArch32.ReportDeferredSError(VDFSR<15:14>, VDFSR<12>);
        else
            VDISR_EL2 = AArch64.ReportDeferredSError(VSESR_EL2<24:0>)<31:0>;
        HCR_EL2.VSE = '0';                       // Clear pending virtual SError

    return;
```

aarch64/functions/registers/AArch64.MaybeZeroRegisterUppers

```
// AArch64.MaybeZeroRegisterUppers()
// =================================
// On taking an exception to  AArch64 from AArch32, it is CONSTRAINED UNPREDICTABLE whether the top
// 32 bits of registers visible at any lower Exception level using AArch32 are set to zero.

AArch64.MaybeZeroRegisterUppers()
    assert UsingAArch32();         // Always called from AArch32 state before entering AArch64 state
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then
        first = 0;  last = 14;  include_R15 = FALSE;
    elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) then
        first = 0;  last = 30;  include_R15 = FALSE;
    else
        first = 0;  last = 30;  include_R15 = TRUE;

    for n = first to last
        if (n != 15 || include_R15) && ConstrainUnpredictableBool() then
            _R[n]<63:32> = Zeros();

    return;
```

aarch64/functions/registers/AArch64.ResetGeneralRegisters

```
// AArch64.ResetGeneralRegisters()
// ==============================

AArch64.ResetGeneralRegisters()

    for i = 0 to 30
        X[i] = bits(64) UNKNOWN;

    return;
```

aarch64/functions/registers/AArch64.ResetSIMDFPRegisters

```
// AArch64.ResetSIMDFPRegisters()
// ==============================

AArch64.ResetSIMDFPRegisters()

    for i = 0 to 31
        V[i] = bits(128) UNKNOWN;

    return;
```

aarch64/functions/registers/AArch64.ResetSpecialRegisters

```
// AArch64.ResetSpecialRegisters()
// ==============================

AArch64.ResetSpecialRegisters()

    // AArch64 special registers
    SP_EL0 = bits(64) UNKNOWN;
    SP_EL1 = bits(64) UNKNOWN;
    SPSR_EL1 = bits(32) UNKNOWN;
    ELR_EL1  = bits(64) UNKNOWN;
    if HaveEL(EL2) then
        SP_EL2 = bits(64) UNKNOWN;
        SPSR_EL2 = bits(32) UNKNOWN;
        ELR_EL2  = bits(64) UNKNOWN;
    if HaveEL(EL3) then
        SP_EL3 = bits(64) UNKNOWN;
        SPSR_EL3 = bits(32) UNKNOWN;
        ELR_EL3  = bits(64) UNKNOWN;

    // AArch32 special registers that are not architecturally mapped to AArch64 registers
    if HaveAArch32EL(EL1) then
        SPSR_fiq = bits(32) UNKNOWN;
        SPSR_irq = bits(32) UNKNOWN;
        SPSR_abt = bits(32) UNKNOWN;
        SPSR_und = bits(32) UNKNOWN;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    // External debug special registers
    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(32) UNKNOWN;

    return;
```

aarch64/functions/registers/AArch64.ResetSystemRegisters

```
AArch64.ResetSystemRegisters(boolean cold_reset);
```

aarch64/functions/registers/PC

```
// PC - non-assignment form
// ========================
// Read program counter.

bits(64) PC[]
    return _PC;
```

aarch64/functions/registers/SP

```
// SP[] - assignment form
// ======================
// Write to stack pointer from either a 32-bit or a 64-bit value.

SP[] = bits(width) value
    assert width IN {32,64};
    if PSTATE.SP == '0' then
        SP_EL0 = ZeroExtend(value);
    else
        case PSTATE.EL of
            when EL0  SP_EL0 = ZeroExtend(value);
            when EL1  SP_EL1 = ZeroExtend(value);
            when EL2  SP_EL2 = ZeroExtend(value);
            when EL3  SP_EL3 = ZeroExtend(value);
    return;

// SP[] - non-assignment form
// ==========================
// Read stack pointer with implicit slice of 8, 16, 32 or 64 bits.

bits(width) SP[]
    assert width IN {8,16,32,64};
    if PSTATE.SP == '0' then
        return SP_EL0<width-1:0>;
    else
        case PSTATE.EL of
            when EL0  return SP_EL0<width-1:0>;
            when EL1  return SP_EL1<width-1:0>;
            when EL2  return SP_EL2<width-1:0>;
            when EL3  return SP_EL3<width-1:0>;
```

aarch64/functions/registers/V

```
// V[] - assignment form
// =====================
// Write to SIMD&FP register with implicit extension from
// 8, 16, 32, 64 or 128 bits.

V[integer n] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64,128};
    integer vlen = if IsSVEEnabled(PSTATE.EL) then VL else 128;
    if ConstrainUnpredictableBool() then
        _Z[n] = ZeroExtend(value);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        _Z[n]<vlen-1:0> = ZeroExtend(value);

// V[] - non-assignment form
// =========================
// Read from SIMD&FP register with implicit slice of 8, 16
// 32, 64 or 128 bits.

bits(width) V[integer n]
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64,128};
    return _Z[n]<width-1:0>;
```

aarch64/functions/registers/Vpart

```
// Vpart[] - non-assignment form
// =============================
// Reads a 128-bit SIMD&FP register in up to two parts:
//  part 0 returns the bottom 8, 16, 32 or 64 bits of a value held in the register;
//  part 1 returns the top half of the bottom 64 bits or the top half of the 128-bit
//  value held in the register.

bits(width) Vpart[integer n, integer part]
    assert n >= 0 && n <= 31;
    assert part IN {0, 1};
    if part == 0 then
        assert width < 128;
        return V[n];
    else
        assert width IN {32,64};
        bits(128) vreg = V[n];
        return vreg<(width * 2)-1:width>;

// Vpart[] - assignment form
// =========================
// Writes a 128-bit SIMD&FP register in up to two parts:
//  part 0 zero extends a 8, 16, 32, or 64-bit value to fill the whole register;
//  part 1 inserts a 64-bit value into the top half of the register.

Vpart[integer n, integer part] = bits(width) value
    assert n >= 0 && n <= 31;
    assert part IN {0, 1};
    if part == 0 then
        assert width < 128;
        V[n] = value;
    else
        assert width == 64;
        bits(64) vreg = V[n];
        V[n] = value<63:0> : vreg;
```

aarch64/functions/registers/X

```
// X[] - assignment form
// =====================
// Write to general-purpose register from either a 32-bit or a 64-bit value.

X[integer n] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width IN {32,64};
    if n != 31 then
        _R[n] = ZeroExtend(value);
    return;

// X[] - non-assignment form
// =========================
// Read from general-purpose register with implicit slice of 8, 16, 32 or 64 bits.
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(width) X[integer n]
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64};
    if n != 31 then
        return _R[n]<width-1:0>;
    else
        return Zeros(width);
```

aarch64/functions/sve/AArch32.IsFPEnabled

```
// AArch32.IsFPEnabled()
// =====================
// Returns TRUE if access to the SIMD&FP instructions or System registers are
// enabled at the target exception level in AArch32 state and FALSE otherwise.

boolean AArch32.IsFPEnabled(bits(2) el)
    if el == EL0 && !ELUsingAArch32(EL1) then
        return AArch64.IsFPEnabled(el);

    if HaveEL(EL3) && ELUsingAArch32(EL3) && !IsSecure() then
        // Check if access disabled in NSACR
        if NSACR.cp10 == '0' then return FALSE;

    if el IN {EL0, EL1} then
        // Check if access disabled in CPACR
        case CPACR.cp10 of
            when '00' disabled = TRUE;
            when '01' disabled = el == EL0;
            when '10' disabled = ConstrainUnpredictableBool();
            when '11' disabled = FALSE;
        if disabled then return FALSE;

    if el IN {EL0, EL1, EL2} then
        if EL2Enabled() then
            if !ELUsingAArch32(EL2) then
                if CPTR_EL2.TFP == '1' then return FALSE;
            else
                if HCPTR.TCP10 == '1' then return FALSE;

    if HaveEL(EL3) && !ELUsingAArch32(EL3) then
        // Check if access disabled in CPTR_EL3
        if CPTR_EL3.TFP == '1' then return FALSE;

    return TRUE;
```

aarch64/functions/sve/AArch64.IsFPEnabled

```
// AArch64.IsFPEnabled()
// =====================
// Returns TRUE if access to the SIMD&FP instructions or System registers are
// enabled at the target exception level in AArch64 state and FALSE otherwise.

boolean AArch64.IsFPEnabled(bits(2) el)
    // Check if access disabled in CPACR_EL1
    if el IN {EL0, EL1} && !IsInHost() then
        // Check FP&SIMD at EL0/EL1
        case CPACR_EL1.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = el == EL0;
            when '11' disabled = FALSE;
        if disabled then return FALSE;

    // Check if access disabled in CPTR_EL2
    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            case CPTR_EL2.FPEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = el == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then return FALSE;
        else
            if CPTR_EL2.TFP == '1' then return FALSE;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.TFP == '1' then return FALSE;

    return TRUE;
```

aarch64/functions/sve/CeilPow2

```
// CeilPow2()
// ==========

// For a positive integer X, return the smallest power of 2 >= X

integer CeilPow2(integer x)
    if x == 0 then return 0;
    if x == 1 then return 2;
    return FloorPow2(x - 1) * 2;
```

aarch64/functions/sve/CheckSVEEnabled

```
// CheckSVEEnabled()
// =================
// Checks for traps on SVE instructions and instructions that
// access SVE System registers.

CheckSVEEnabled()
    // Check if access disabled in CPACR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check SVE at EL0/EL1
        case CPACR_EL1.ZEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then SVEAccessTrap(EL1);

        // Check SIMD&FP at EL0/EL1
        case CPACR_EL1.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);

    // Check if access disabled in CPTR_EL2
    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            // Check SVE at EL2
            case CPTR_EL2.ZEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then SVEAccessTrap(EL2);

            // Check SIMD&FP at EL2
            case CPTR_EL2.FPEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        else
            if CPTR_EL2.TZ == '1' then SVEAccessTrap(EL2);
            if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.EZ == '0' then SVEAccessTrap(EL3);
        if CPTR_EL3.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL3);
```

aarch64/functions/sve/DecodePredCount

```
// DecodePredCount()
// =================

integer DecodePredCount(bits(5) pattern, integer esize)
    integer elements = VL DIV esize;
    integer numElem;
    case pattern of
        when '00000' numElem = FloorPow2(elements);
        when '00001' numElem = if elements >= 1 then 1 else 0;
        when '00010' numElem = if elements >= 2 then 2 else 0;
        when '00011' numElem = if elements >= 3 then 3 else 0;
        when '00100' numElem = if elements >= 4 then 4 else 0;
        when '00101' numElem = if elements >= 5 then 5 else 0;
        when '00110' numElem = if elements >= 6 then 6 else 0;
        when '00111' numElem = if elements >= 7 then 7 else 0;
        when '01000' numElem = if elements >= 8 then 8 else 0;
        when '01001' numElem = if elements >= 16 then 16 else 0;
        when '01010' numElem = if elements >= 32 then 32 else 0;
        when '01011' numElem = if elements >= 64 then 64 else 0;
        when '01100' numElem = if elements >= 128 then 128 else 0;
        when '01101' numElem = if elements >= 256 then 256 else 0;
        when '11101' numElem = elements - (elements MOD 4);
        when '11110' numElem = elements - (elements MOD 3);
        when '11111' numElem = elements;
        otherwise    numElem = 0;
    return numElem;
```

aarch64/functions/sve/ElemFFR

```
// ElemFFR[] - non-assignment form
// ===============================

bit ElemFFR[integer e, integer esize]
    return ElemP[_FFR, e, esize];

// ElemFFR[] - assignment form
// ===========================

ElemFFR[integer e, integer esize] = bit value
    integer psize = esize DIV 8;
    integer n = e * psize;
    assert n >= 0 && (n + psize) <= PL;
    _FFR<n+psize-1:n> = ZeroExtend(value, psize);
    return;
```

aarch64/functions/sve/ElemP

```
// ElemP[] - non-assignment form
// =============================

bit ElemP[bits(N) pred, integer e, integer esize]
    integer n = e * (esize DIV 8);
    assert n >= 0 && n < N;
    return pred<n>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
// ElemP[] - assignment form
// =========================

ElemP[bits(N) &pred, integer e, integer esize] = bit value
    integer psize = esize DIV 8;
    integer n = e * psize;
    assert n >= 0 && (n + psize) <= N;
    pred<n+psize-1:n> = ZeroExtend(value, psize);
    return;
```

aarch64/functions/sve/FFR

```
// FFR[] - non-assignment form
// ===========================

bits(width) FFR[]
    assert width == PL;
    return _FFR<width-1:0>;

// FFR[] - assignment form
// =======================

FFR[] = bits(width) value
    assert width == PL;
    if ConstrainUnpredictableBool() then
        _FFR = ZeroExtend(value);
    else
        _FFR<width-1:0> = value;
```

aarch64/functions/sve/FPCompareNE

```
// FPCompareNE()
// =============

boolean FPCompareNE(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
        result = TRUE;
        if type1==FPType_SNaN || type2==FPType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpcr);
    else // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 != value2);
    return result;
```

aarch64/functions/sve/FPCompareUN

```
// FPCompareUN()
// =============

boolean FPCompareUN(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPType_SNaN || type2==FPType_SNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);
    return (type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN);
```

aarch64/functions/sve/FPConvertSVE

```
// FPConvertSVE()
// ==============

bits(M) FPConvertSVE(bits(N) op, FPCRType fpcr, FPRounding rounding)
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    fpcr.AHP = '0';
    return FPConvert(op, fpcr, rounding);

// FPConvertSVE()
// ==============

bits(M) FPConvertSVE(bits(N) op, FPCRType fpcr)
    fpcr.AHP = '0';
    return FPConvert(op, fpcr, FPRoundingMode(fpcr));
```

aarch64/functions/sve/FPExpA

```
// FPExpA()
// ========

bits(N) FPExpA(bits(N) op)
    assert N IN {16,32,64};
    bits(N) result;
    bits(N) coeff;
    integer idx = if N == 16 then UInt(op<4:0>) else UInt(op<5:0>);
    coeff = FPExpCoefficient[idx];
    if N == 16 then
        result<15:0> = '0':op<9:5>:coeff<9:0>;
    elsif N == 32 then
        result<31:0> = '0':op<13:6>:coeff<22:0>;
    else // N == 64
        result<63:0> = '0':op<16:6>:coeff<51:0>;

    return result;
```

aarch64/functions/sve/FPExpCoefficient

```
// FPExpCoefficient()
// ==================

bits(N) FPExpCoefficient[integer index]
    assert N IN {16,32,64};
    integer result;

    if N == 16 then
        case index of
            when  0 result = 0x0000;
            when  1 result = 0x0016;
            when  2 result = 0x002d;
            when  3 result = 0x0045;
            when  4 result = 0x005d;
            when  5 result = 0x0075;
            when  6 result = 0x008e;
            when  7 result = 0x00a8;
            when  8 result = 0x00c2;
            when  9 result = 0x00dc;
            when 10 result = 0x00f8;
            when 11 result = 0x0114;
            when 12 result = 0x0130;
            when 13 result = 0x014d;
            when 14 result = 0x016b;
            when 15 result = 0x0189;
            when 16 result = 0x01a8;
            when 17 result = 0x01c8;
            when 18 result = 0x01e8;
            when 19 result = 0x0209;
            when 20 result = 0x022b;
            when 21 result = 0x024e;
            when 22 result = 0x0271;
            when 23 result = 0x0295;
            when 24 result = 0x02ba;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        when 25 result = 0x02e0;
        when 26 result = 0x0306;
        when 27 result = 0x032e;
        when 28 result = 0x0356;
        when 29 result = 0x037f;
        when 30 result = 0x03a9;
        when 31 result = 0x03d4;

elsif N == 32 then
    case index of
        when  0 result = 0x000000;
        when  1 result = 0x0164d2;
        when  2 result = 0x02cd87;
        when  3 result = 0x043a29;
        when  4 result = 0x05aac3;
        when  5 result = 0x071f62;
        when  6 result = 0x08980f;
        when  7 result = 0x0a14d5;
        when  8 result = 0x0b95c2;
        when  9 result = 0x0d1adf;
        when 10 result = 0x0ea43a;
        when 11 result = 0x1031dc;
        when 12 result = 0x11c3d3;
        when 13 result = 0x135a2b;
        when 14 result = 0x14f4f0;
        when 15 result = 0x16942d;
        when 16 result = 0x1837f0;
        when 17 result = 0x19e046;
        when 18 result = 0x1b8d3a;
        when 19 result = 0x1d3eda;
        when 20 result = 0x1ef532;
        when 21 result = 0x20b051;
        when 22 result = 0x227043;
        when 23 result = 0x243516;
        when 24 result = 0x25fed7;
        when 25 result = 0x27cd94;
        when 26 result = 0x29a15b;
        when 27 result = 0x2b7a3a;
        when 28 result = 0x2d583f;
        when 29 result = 0x2f3b79;
        when 30 result = 0x3123f6;
        when 31 result = 0x3311c4;
        when 32 result = 0x3504f3;
        when 33 result = 0x36fd92;
        when 34 result = 0x38fbaf;
        when 35 result = 0x3aff5b;
        when 36 result = 0x3d08a4;
        when 37 result = 0x3f179a;
        when 38 result = 0x412c4d;
        when 39 result = 0x4346cd;
        when 40 result = 0x45672a;
        when 41 result = 0x478d75;
        when 42 result = 0x49b9be;
        when 43 result = 0x4bec15;
        when 44 result = 0x4e248c;
        when 45 result = 0x506334;
        when 46 result = 0x52a81e;
        when 47 result = 0x54f35b;
        when 48 result = 0x5744fd;
        when 49 result = 0x599d16;
        when 50 result = 0x5bfbb8;
        when 51 result = 0x5e60f5;
        when 52 result = 0x60ccdf;
        when 53 result = 0x633f89;
        when 54 result = 0x65b907;
        when 55 result = 0x68396a;
        when 56 result = 0x6ac0c7;
        when 57 result = 0x6d4f30;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            when 58 result = 0x6fe4ba;
            when 59 result = 0x728177;
            when 60 result = 0x75257d;
            when 61 result = 0x77d0df;
            when 62 result = 0x7a83b3;
            when 63 result = 0x7d3e0c;

    else // N == 64
        case index of
            when  0 result = 0x0000000000000;
            when  1 result = 0x02C9A3E778061;
            when  2 result = 0x059B0D3158574;
            when  3 result = 0x0874518759BC8;
            when  4 result = 0x0B5586CF9890F;
            when  5 result = 0x0E3EC32D3D1A2;
            when  6 result = 0x11301D0125B51;
            when  7 result = 0x1429AAEA92DE0;
            when  8 result = 0x172B83C7D517B;
            when  9 result = 0x1A35BEB6FCB75;
            when 10 result = 0x1D4873168B9AA;
            when 11 result = 0x2063B88628CD6;
            when 12 result = 0x2387A6E756238;
            when 13 result = 0x26B4565E27CDD;
            when 14 result = 0x29E9DF51FDEE1;
            when 15 result = 0x2D285A6E4030B;
            when 16 result = 0x306FE0A31B715;
            when 17 result = 0x33C08B26416FF;
            when 18 result = 0x371A7373AA9CB;
            when 19 result = 0x3A7DB34E59FF7;
            when 20 result = 0x3DEA64C123422;
            when 21 result = 0x4160A21F72E2A;
            when 22 result = 0x44E086061892D;
            when 23 result = 0x486A2B5C13CD0;
            when 24 result = 0x4BFDAD5362A27;
            when 25 result = 0x4F9B2769D2CA7;
            when 26 result = 0x5342B569D4F82;
            when 27 result = 0x56F4736B527DA;
            when 28 result = 0x5AB07DD485429;
            when 29 result = 0x5E76F15AD2148;
            when 30 result = 0x6247EB03A5585;
            when 31 result = 0x6623882552225;
            when 32 result = 0x6A09E667F3BCD;
            when 33 result = 0x6DFB23C651A2F;
            when 34 result = 0x71F75E8EC5F74;
            when 35 result = 0x75FEB564267C9;
            when 36 result = 0x7A11473EB0187;
            when 37 result = 0x7E2F336CF4E62;
            when 38 result = 0x82589994CCE13;
            when 39 result = 0x868D99B4492ED;
            when 40 result = 0x8ACE5422AA0DB;
            when 41 result = 0x8F1AE99157736;
            when 42 result = 0x93737B0CDC5E5;
            when 43 result = 0x97D829FDE4E50;
            when 44 result = 0x9C49182A3F090;
            when 45 result = 0xA0C667B5DE565;
            when 46 result = 0xA5503B23E255D;
            when 47 result = 0xA9E6B5579FDBF;
            when 48 result = 0xAE89F995AD3AD;
            when 49 result = 0xB33A2B84F15FB;
            when 50 result = 0xB7F76F2FB5E47;
            when 51 result = 0xBCC1E904BC1D2;
            when 52 result = 0xC199BDD85529C;
            when 53 result = 0xC67F12E57D14B;
            when 54 result = 0xCB720DCEF9069;
            when 55 result = 0xD072D4A07897C;
            when 56 result = 0xD5818DCFBA487;
            when 57 result = 0xDA9E603DB3285;
            when 58 result = 0xDFC97337B9B5F;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            when 59 result = 0xE502EE78B3FF6;
            when 60 result = 0xEA4AFA2A490DA;
            when 61 result = 0xEFA1BEE615A27;
            when 62 result = 0xF50765B6E4540;
            when 63 result = 0xFA7C1819E90D8;

    return result<N-1:0>;
```

aarch64/functions/sve/FPMinNormal

```
// FPMinNormal()
// =============

bits(N) FPMinNormal(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = Zeros(E-1):'1';
    frac = Zeros(F);
    return sign : exp : frac;
```

aarch64/functions/sve/FPOne

```
// FPOne()
// =======

bits(N) FPOne(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = '0':Ones(E-1);
    frac = Zeros(F);
    return sign : exp : frac;
```

aarch64/functions/sve/FPPointFive

```
// FPPointFive()
// =============

bits(N) FPPointFive(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = '0':Ones(E-2):'0';
    frac = Zeros(F);
    return sign : exp : frac;
```

aarch64/functions/sve/FPProcess

```
// FPProcess()
// ===========

bits(N) FPProcess(bits(N) input)
    bits(N) result;
    assert N IN {16,32,64};
    (fptype,sign,value) = FPUnpack(input, FPCR);
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, input, FPCR);
    elsif fptype == FPType_Infinity then
        result = FPInfinity(sign);
    elsif fptype == FPType_Zero then
        result = FPZero(sign);
    else
        result = FPRound(value, FPCR);
    return result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch64/functions/sve/FPScale

```
// FPScale()
// =========

bits(N) FPScale(bits (N) op, integer scale, FPCRType fpcr)
    assert N IN {16,32,64};
    (fptype,sign,value) = FPUnpack(op, fpcr);
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, op, fpcr);
    elsif fptype == FPType_Zero then
        result = FPZero(sign);
    elsif fptype == FPType_Infinity then
        result = FPInfinity(sign);
    else
        result = FPRound(value * (2.0^scale), fpcr);
    return result;
```

aarch64/functions/sve/FPTrigMAdd

```
// FPTrigMAdd()
// ============

bits(N) FPTrigMAdd(integer x, bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    assert x >= 0;
    assert x < 8;
    bits(N) coeff;

    if op2<N-1> == '1' then
        x = x + 8;
    op2<N-1> = '0';

    coeff = FPTrigMAddCoefficient[x];
    result = FPMulAdd(coeff, op1, op2, fpcr);

    return result;
```

aarch64/functions/sve/FPTrigMAddCoefficient

```
// FPTrigMAddCoefficient()
// =======================

bits(N) FPTrigMAddCoefficient[integer index]
    assert N IN {16,32,64};
    integer result;

    if N == 16 then
        case index of
            when  0 result = 0x3c00;
            when  1 result = 0xb155;
            when  2 result = 0x2030;
            when  3 result = 0x0000;
            when  4 result = 0x0000;
            when  5 result = 0x0000;
            when  6 result = 0x0000;
            when  7 result = 0x0000;
            when  8 result = 0x3c00;
            when  9 result = 0xb800;
            when 10 result = 0x293a;
            when 11 result = 0x0000;
            when 12 result = 0x0000;
            when 13 result = 0x0000;
            when 14 result = 0x0000;
            when 15 result = 0x0000;
    elsif N == 32 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        case index of
            when  0 result = 0x3f800000;
            when  1 result = 0xbe2aaaab;
            when  2 result = 0x3c088886;
            when  3 result = 0xb95008b9;
            when  4 result = 0x36369d6d;
            when  5 result = 0x00000000;
            when  6 result = 0x00000000;
            when  7 result = 0x00000000;
            when  8 result = 0x3f800000;
            when  9 result = 0xbf000000;
            when 10 result = 0x3d2aaaa6;
            when 11 result = 0xbab60705;
            when 12 result = 0x37cd37cc;
            when 13 result = 0x00000000;
            when 14 result = 0x00000000;
            when 15 result = 0x00000000;
    else // N == 64
        case index of
            when  0 result = 0x3ff0000000000000;
            when  1 result = 0xbfc5555555555543;
            when  2 result = 0x3f8111111110f30c;
            when  3 result = 0xbf2a01a019b92fc6;
            when  4 result = 0x3ec71de351f3d22b;
            when  5 result = 0xbe5ae5e2b60f7b91;
            when  6 result = 0x3de5d8408868552f;
            when  7 result = 0x0000000000000000;
            when  8 result = 0x3ff0000000000000;
            when  9 result = 0xbfe0000000000000;
            when 10 result = 0x3fa5555555555536;
            when 11 result = 0xbf56c16c16c13a0b;
            when 12 result = 0x3efa01a019b1e8d8;
            when 13 result = 0xbe927e4f7282f468;
            when 14 result = 0x3e21ee96d2641b13;
            when 15 result = 0xbda8f76380fbb401;

    return result<N-1:0>;
```

aarch64/functions/sve/FPTrigSMul

```
// FPTrigSMul()
// ============

bits(N) FPTrigSMul(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    result = FPMul(op1, op1, fpcr);
    (fptype, sign, value) = FPUnpack(result, fpcr);
    if (fptype != FPType_QNaN) && (fptype != FPType_SNaN) then
        result<N-1> = op2<0>;

    return result;
```

aarch64/functions/sve/FPTrigSSel

```
// FPTrigSSel()
// ============

bits(N) FPTrigSSel(bits(N) op1, bits(N) op2)
    assert N IN {16,32,64};
    bits(N) result;

    if op2<0> == '1' then
        result = FPOne(op2<1>);
    else
        result = op1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        result<N-1> = result<N-1> EOR op2<1>;

    return result;
```

aarch64/functions/sve/FirstActive

```
// FirstActive()
// =============

bit FirstActive(bits(N) mask, bits(N) x, integer esize)
    integer elements = N DIV (esize DIV 8);
    for e = 0 to elements-1
        if ElemP[mask, e, esize] == '1' then return ElemP[x, e, esize];
    return '0';
```

aarch64/functions/sve/FloorPow2

```
// FloorPow2()
// ===========
// For a positive integer X, return the largest power of 2 <= X

integer FloorPow2(integer x)
    assert x >= 0;
    integer n = 1;
    if x == 0 then return 0;
    while x >= 2^n do
        n = n + 1;
    return 2^(n - 1);
```

aarch64/functions/sve/HaveSVE

```
// HaveSVE()
// =========

boolean HaveSVE()
    return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Have SVE ISA";
```

aarch64/functions/sve/HaveSVEFP32MatMulExt

```
// HaveSVEFP32MatMulExt()
// ======================
// Returns TRUE if single-precision floating-point matrix multiply instruction support implemented and
FALSE otherwise.

boolean HaveSVEFP32MatMulExt()
    return HaveSVE() && boolean IMPLEMENTATION_DEFINED "Have SVE FP32 Matrix Multiply extension";
```

aarch64/functions/sve/HaveSVEFP64MatMulExt

```
// HaveSVEFP64MatMulExt()
// ======================
// Returns TRUE if double-precision floating-point matrix multiply instruction support implemented and
FALSE otherwise.

boolean HaveSVEFP64MatMulExt()
    return HaveSVE() && boolean IMPLEMENTATION_DEFINED "Have SVE FP64 Matrix Multiply extension";
```

aarch64/functions/sve/ImplementedSVEVectorLength

```
// ImplementedSVEVectorLength()
// ============================
// Reduce SVE vector length to a supported value (e.g. power of two)
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer ImplementedSVEVectorLength(integer nbits)
    return integer IMPLEMENTATION_DEFINED;
```

aarch64/functions/sve/IsEven

```
// IsEven()
// ========

boolean IsEven(integer val)
    return val MOD 2 == 0;
```

aarch64/functions/sve/IsFPEnabled

```
// IsFPEnabled()
// =============
// Returns TRUE if accesses to the Advanced SIMD and floating-point
// registers are enabled at the target exception level in the current
// execution state and FALSE otherwise.

boolean IsFPEnabled(bits(2) el)
    if ELUsingAArch32(el) then
        return AArch32.IsFPEnabled(el);
    else
        return AArch64.IsFPEnabled(el);
```

aarch64/functions/sve/IsSVEEnabled

```
// IsSVEEnabled()
// ==============
// Returns TRUE if access to SVE instructions and System registers is
// enabled at the target exception level and FALSE otherwise.

boolean IsSVEEnabled(bits(2) el)
    if ELUsingAArch32(el) then
        return FALSE;

    // Check if access disabled in CPACR_EL1
    if el IN {EL0, EL1} && !IsInHost() then
        // Check SVE at EL0/EL1
        case CPACR_EL1.ZEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = el == EL0;
            when '11' disabled = FALSE;
        if disabled then return FALSE;

    // Check if access disabled in CPTR_EL2
    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            case CPTR_EL2.ZEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = el == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then return FALSE;
        else
            if CPTR_EL2.TZ == '1' then return FALSE;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.EZ == '0' then return FALSE;

    return TRUE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch64/functions/sve/LastActive

```
// LastActive()
// ============

bit LastActive(bits(N) mask, bits(N) x, integer esize)
    integer elements = N DIV (esize DIV 8);
    for e = elements-1 downto 0
        if ElemP[mask, e, esize] == '1' then return ElemP[x, e, esize];
    return '0';
```

aarch64/functions/sve/LastActiveElement

```
// LastActiveElement()
// ===================

integer LastActiveElement(bits(N) mask, integer esize)
    assert esize IN {8, 16, 32, 64};
    integer elements = VL DIV esize;
    for e = elements-1 downto 0
        if ElemP[mask, e, esize] == '1' then return e;
    return -1;
```

aarch64/functions/sve/MAX_PL

```
constant integer MAX_PL = 256;
```

aarch64/functions/sve/MAX_VL

```
constant integer MAX_VL = 2048;
```

aarch64/functions/sve/MaybeZeroSVEUppers

```
// MaybeZeroSVEUppers()
// ====================

MaybeZeroSVEUppers(bits(2) target_el)
    boolean lower_enabled;

    if UInt(target_el) <= UInt(PSTATE.EL) || !IsSVEEnabled(target_el) then
        return;

    if target_el == EL3 then
        if EL2Enabled() then
            lower_enabled = IsFPEnabled(EL2);
        else
            lower_enabled = IsFPEnabled(EL1);
    else
        lower_enabled = IsFPEnabled(target_el - 1);

    if lower_enabled then
        integer vl = if IsSVEEnabled(PSTATE.EL) then VL else 128;
        integer pl = vl DIV 8;
        for n = 0 to 31
            if ConstrainUnpredictableBool() then
                _Z[n] = ZeroExtend(_Z[n]<vl-1:0>);
        for n = 0 to 15
            if ConstrainUnpredictableBool() then
                _P[n] = ZeroExtend(_P[n]<pl-1:0>);
        if ConstrainUnpredictableBool() then
            _FFR = ZeroExtend(_FFR<pl-1:0>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch64/functions/sve/MemNF

```
// MemNF[] - non-assignment form
// ============================

(bits(8*size), boolean) MemNF[bits(64) address, integer size, AccType acctype]
    assert size IN {1, 2, 4, 8, 16};
    bits(8*size) value;

    aligned = (address == Align(address, size));
    A = SCTLR[].A;

    if !aligned && (A == '1') then
        return (bits(8*size) UNKNOWN, TRUE);

    atomic = aligned || size == 1;

    if !atomic then
        (value<7:0>, bad) = MemSingleNF[address, 1, acctype, aligned];

        if bad then
            return (bits(8*size) UNKNOWN, TRUE);

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        if !aligned then
            c = ConstrainUnpredictable();
            assert c IN {Constraint_FAULT, Constraint_NONE};
            if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            (value<8*i+7:8*i>, bad) = MemSingleNF[address+i, 1, acctype, aligned];

            if bad then
                return (bits(8*size) UNKNOWN, TRUE);
    else
        (value, bad) = MemSingleNF[address, size, acctype, aligned];
        if bad then
            return (bits(8*size) UNKNOWN, TRUE);

    if BigEndian() then
        value = BigEndianReverse(value);

    return (value, FALSE);
```

aarch64/functions/sve/MemSingleNF

```
// MemSingleNF[] - non-assignment form
// ===================================

(bits(8*size), boolean) MemSingleNF[bits(64) address, integer size, AccType acctype, boolean wasaligned]
    bits(8*size) value;
    boolean iswrite = FALSE;
    AddressDescriptor memaddrdesc;

    // Implementation may suppress NF load for any reason
    if ConstrainUnpredictableBool() then
        return (bits(8*size) UNKNOWN, TRUE);

    // MMU or MPU
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, wasaligned, size);

    // Non-fault load from Device memory must not be performed externally
    if memaddrdesc.memattrs.memtype == MemType_Device then
        return (bits(8*size) UNKNOWN, TRUE);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return (bits(8*size) UNKNOWN, TRUE);

    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);
    if HaveMTEExt() then
        if AArch64.AccessIsTagChecked(address, acctype) then
            bits(4) ptag = AArch64.PhysicalTag(address);
            if !AArch64.CheckTag(memaddrdesc, ptag, iswrite) then
                return (bits(8*size) UNKNOWN, TRUE);
    value = _Mem[memaddrdesc, size, accdesc];

    return (value, FALSE);
```

aarch64/functions/sve/NoneActive

```
// NoneActive()
// ============

bit NoneActive(bits(N) mask, bits(N) x, integer esize)
    integer elements = N DIV (esize DIV 8);
    for e = 0 to elements-1
        if ElemP[mask, e, esize] == '1' && ElemP[x, e, esize] == '1' then return '0';
    return '1';
```

aarch64/functions/sve/P

```
// P[] - non-assignment form
// =========================

bits(width) P[integer n]
    assert n >= 0 && n <= 31;
    assert width == PL;
    return _P[n]<width-1:0>;

// P[] - assignment form
// =====================

P[integer n] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width == PL;
    if ConstrainUnpredictableBool() then
        _P[n] = ZeroExtend(value);
    else
        _P[n]<width-1:0> = value;
```

aarch64/functions/sve/PL

```
// PL - non-assignment form
// ========================

integer PL
    return VL DIV 8;
```

aarch64/functions/sve/PredTest

```
// PredTest()
// ==========

bits(4) PredTest(bits(N) mask, bits(N) result, integer esize)
    bit n = FirstActive(mask, result, esize);
    bit z = NoneActive(mask, result, esize);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    bit c = NOT LastActive(mask, result, esize);
    bit v = '0';
    return n:z:c:v;
```

aarch64/functions/sve/ReducePredicated

```
// ReducePredicated()
// ==================

bits(esize) ReducePredicated(ReduceOp op, bits(N) input, bits(M) mask, bits(esize) identity)
    assert(N == M * 8);
    integer p2bits = CeilPow2(N);
    bits(p2bits) operand;
    integer elements = p2bits DIV esize;

    for e = 0 to elements-1
        if e * esize < N && ElemP[mask, e, esize] == '1' then
            Elem[operand, e, esize] = Elem[input, e, esize];
        else
            Elem[operand, e, esize] = identity;

    return Reduce(op, operand, esize);
```

aarch64/functions/sve/Reverse

```
// Reverse()
// =========
// Reverse subwords of M bits in an N-bit word

bits(N) Reverse(bits(N) word, integer M)
    bits(N) result;
    integer sw = N DIV M;
    assert N == sw * M;
    for s = 0 to sw-1
        Elem[result, sw - 1 - s, M] = Elem[word, s, M];
    return result;
```

aarch64/functions/sve/SVEAccessTrap

```
// SVEAccessTrap()
// ===============
// Trapped access to SVE registers due to CPACR_EL1, CPTR_EL2, or CPTR_EL3.

SVEAccessTrap(bits(2) target_el)
    assert UInt(target_el) >= UInt(PSTATE.EL) && target_el != EL0 && HaveEL(target_el);
    route_to_el2 = target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1';

    exception = ExceptionSyndrome(Exception_SVEAccessTrap);
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    if route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

aarch64/functions/sve/SVECmp

```
enumeration SVECmp { Cmp_EQ, Cmp_NE, Cmp_GE, Cmp_GT, Cmp_LT, Cmp_LE, Cmp_UN };
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch64/functions/sve/SVEMoveMaskPreferred

```
// SVEMoveMaskPreferred()
// ======================
// Return FALSE if a bitmask immediate encoding would generate an immediate
// value that could also be represented by a single DUP instruction.
// Used as a condition for the preferred MOV<-DUPM alias.

boolean SVEMoveMaskPreferred(bits(13) imm13)
    bits(64) imm;
    (imm, -) = DecodeBitMasks(imm13<12>, imm13<5:0>, imm13<11:6>, TRUE);

    // Check for 8 bit immediates
    if !IsZero(imm<7:0>) then
        // Check for 'ffffffffffffffxy' or '00000000000000xy'
        if IsZero(imm<63:7>) || IsOnes(imm<63:7>) then
            return FALSE;

        // Check for 'ffffffxyffffffxy' or '000000xy000000xy'
        if imm<63:32> == imm<31:0> && (IsZero(imm<31:7>) || IsOnes(imm<31:7>)) then
            return FALSE;

        // Check for 'ffxyffxyffxyffxy' or '00xy00xy00xy00xy'
        if imm<63:32> == imm<31:0> && imm<31:16> == imm<15:0> && (IsZero(imm<15:7>) ||
IsOnes(imm<15:7>)) then
            return FALSE;

        // Check for 'xyxyxyxyxyxyxyxy'
        if imm<63:32> == imm<31:0> && imm<31:16> == imm<15:0> && (imm<15:8> == imm<7:0>) then
            return FALSE;

    // Check for 16 bit immediates
    else
        // Check for 'ffffffffffffxy00' or '000000000000xy00'
        if IsZero(imm<63:15>) || IsOnes(imm<63:15>) then
            return FALSE;

        // Check for 'ffffxy00ffffxy00' or '0000xy000000xy00'
        if imm<63:32> == imm<31:0> && (IsZero(imm<31:7>) || IsOnes(imm<31:7>)) then
            return FALSE;

        // Check for 'xy00xy00xy00xy00'
        if imm<63:32> == imm<31:0> && imm<31:16> == imm<15:0> then
            return FALSE;

    return TRUE;
```

aarch64/functions/sve/System

```
array bits(MAX_VL) _Z[0..31];
array bits(MAX_PL) _P[0..15];
bits(MAX_PL) _FFR;
```

aarch64/functions/sve/VL

```
// VL - non-assignment form
// ========================

integer VL
    integer vl;

    if PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost()) then
        vl = UInt(ZCR_EL1.LEN);

    if PSTATE.EL == EL2 || (PSTATE.EL == EL0 && IsInHost()) then
        vl = UInt(ZCR_EL2.LEN);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        vl = Min(vl, UInt(ZCR_EL2.LEN));

    if PSTATE.EL == EL3 then
        vl = UInt(ZCR_EL3.LEN);
    elsif HaveEL(EL3) then
        vl = Min(vl, UInt(ZCR_EL3.LEN));

    vl = (vl + 1) * 128;
    vl = ImplementedSVEVectorLength(vl);

    return vl;
```

aarch64/functions/sve/Z

```
// Z[] - non-assignment form
// =========================

bits(width) Z[integer n]
    assert n >= 0 && n <= 31;
    assert width == VL;
    return _Z[n]<width-1:0>;

// Z[] - assignment form
// =====================

Z[integer n] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width == VL;
    if ConstrainUnpredictableBool() then
        _Z[n] = ZeroExtend(value);
    else
        _Z[n]<width-1:0> = value;
```

aarch64/functions/sysregisters/CNTKCTL

```
// CNTKCTL[] - non-assignment form
// ===============================

CNTKCTLType CNTKCTL[]
    bits(32) r;
    if IsInHost() then
        r = CNTHCTL_EL2;
        return r;
    r = CNTKCTL_EL1;
    return r;
```

aarch64/functions/sysregisters/CNTKCTLType

```
type CNTKCTLType;
```

aarch64/functions/sysregisters/CPACR

```
// CPACR[] - non-assignment form
// =============================

CPACRType CPACR[]
    bits(32) r;
    if IsInHost() then
        r = CPTR_EL2;
        return r;
    r = CPACR_EL1;
    return r;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch64/functions/sysregisters/CPACRType

```
type CPACRType;
```

aarch64/functions/sysregisters/ELR

```
// ELR[] - non-assignment form
// ==========================

bits(64) ELR[bits(2) el]
    bits(64) r;
    case el of
        when EL1  r = ELR_EL1;
        when EL2  r = ELR_EL2;
        when EL3  r = ELR_EL3;
        otherwise Unreachable();
    return r;

// ELR[] - non-assignment form
// ==========================

bits(64) ELR[]
    assert PSTATE.EL != EL0;
    return ELR[PSTATE.EL];

// ELR[] - assignment form
// =======================

ELR[bits(2) el] = bits(64) value
    bits(64) r = value;
    case el of
        when EL1  ELR_EL1 = r;
        when EL2  ELR_EL2 = r;
        when EL3  ELR_EL3 = r;
        otherwise Unreachable();
    return;

// ELR[] - assignment form
// =======================

ELR[] = bits(64) value
    assert PSTATE.EL != EL0;
    ELR[PSTATE.EL] = value;
    return;
```

aarch64/functions/sysregisters/ESR

```
// ESR[] - non-assignment form
// ==========================

ESRType ESR[bits(2) regime]
    bits(32) r;
    case regime of
        when EL1  r = ESR_EL1;
        when EL2  r = ESR_EL2;
        when EL3  r = ESR_EL3;
        otherwise Unreachable();
    return r;

// ESR[] - non-assignment form
// ==========================

ESRType ESR[]
    return ESR[S1TranslationRegime()];

// ESR[] - assignment form
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
// =======================

ESR[bits(2) regime] = ESRType value
    bits(32) r = value;
    case regime of
        when EL1  ESR_EL1 = r;
        when EL2  ESR_EL2 = r;
        when EL3  ESR_EL3 = r;
        otherwise Unreachable();
    return;

// ESR[] - assignment form
// =======================

ESR[] = ESRType value
    ESR[S1TranslationRegime()] = value;
```

aarch64/functions/sysregisters/ESRType

```
type ESRType;
```

aarch64/functions/sysregisters/FAR

```
// FAR[] - non-assignment form
// ===========================

bits(64) FAR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = FAR_EL1;
        when EL2  r = FAR_EL2;
        when EL3  r = FAR_EL3;
        otherwise Unreachable();
    return r;

// FAR[] - non-assignment form
// ===========================

bits(64) FAR[]
    return FAR[S1TranslationRegime()];

// FAR[] - assignment form
// =======================

FAR[bits(2) regime] = bits(64) value
    bits(64) r = value;
    case regime of
        when EL1  FAR_EL1 = r;
        when EL2  FAR_EL2 = r;
        when EL3  FAR_EL3 = r;
        otherwise Unreachable();
    return;

// FAR[] - assignment form
// =======================

FAR[] = bits(64) value
    FAR[S1TranslationRegime()] = value;
    return;
```

aarch64/functions/sysregisters/MAIR

```
// MAIR[] - non-assignment form
// ============================

MAIRType MAIR[bits(2) regime]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    bits(64) r;
    case regime of
        when EL1  r = MAIR_EL1;
        when EL2  r = MAIR_EL2;
        when EL3  r = MAIR_EL3;
        otherwise Unreachable();
    return r;

// MAIR[] - non-assignment form
// ============================

MAIRType MAIR[]
    return MAIR[S1TranslationRegime()];
```

aarch64/functions/sysregisters/MAIRType

```
type MAIRType;
```

aarch64/functions/sysregisters/SCTLR

```
// SCTLR[] - non-assignment form
// =============================

SCTLRType SCTLR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = SCTLR_EL1;
        when EL2  r = SCTLR_EL2;
        when EL3  r = SCTLR_EL3;
        otherwise Unreachable();
    return r;

// SCTLR[] - non-assignment form
// =============================

SCTLRType SCTLR[]
    return SCTLR[S1TranslationRegime()];
```

aarch64/functions/sysregisters/SCTLRType

```
type SCTLRType;
```

aarch64/functions/sysregisters/VBAR

```
// VBAR[] - non-assignment form
// ============================

bits(64) VBAR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = VBAR_EL1;
        when EL2  r = VBAR_EL2;
        when EL3  r = VBAR_EL3;
        otherwise Unreachable();
    return r;

// VBAR[] - non-assignment form
// ============================

bits(64) VBAR[]
    return VBAR[S1TranslationRegime()];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch64/functions/system/AArch64.AllocationTagAccessIsEnabled

```
// AArch64.AllocationTagAccessIsEnabled()
// ======================================
// Check whether access to Allocation Tags is enabled.

boolean AArch64.AllocationTagAccessIsEnabled()
    if SCR_EL3.ATA == '0' && PSTATE.EL IN {EL0, EL1, EL2} then
        return FALSE;
    elsif HCR_EL2.ATA == '0' && PSTATE.EL IN {EL0, EL1} && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11'
then
        return FALSE;
    elsif SCTLR_EL3.ATA == '0' && PSTATE.EL == EL3 then
        return FALSE;
    elsif SCTLR_EL2.ATA == '0' && PSTATE.EL == EL2 then
        return FALSE;
    elsif SCTLR_EL1.ATA == '0' && PSTATE.EL == EL1 then
        return FALSE;
    elsif SCTLR_EL2.ATA0 == '0' && PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' then
        return FALSE;
    elsif SCTLR_EL1.ATA0 == '0' && PSTATE.EL == EL0 && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') then
        return FALSE;
    else
        return TRUE;
```

aarch64/functions/system/AArch64.CheckSystemAccess

```
// AArch64.CheckSystemAccess()
// ===========================
// Checks for traps by NV access.

AArch64.CheckSystemAccess(bits(2) op0, bits(3) op1, bits(4) crn,
                          bits(4) crm, bits(3) op2, bits(5) rt, bit read)

    return;
```

aarch64/functions/system/AArch64.ChooseNonExcludedTag

```
// AArch64.ChooseNonExcludedTag()
// ==============================
// Return a tag derived from the start and the offset values, excluding
// any tags in the given mask.

bits(4) AArch64.ChooseNonExcludedTag(bits(4) tag, bits(4) offset, bits(16) exclude)
    if IsOnes(exclude) then
        return '0000';

    if offset == '0000' then
        while exclude<UInt(tag)> == '1' do
            tag = tag + '0001';

    while offset != '0000' do
        offset = offset - '0001';
        tag = tag + '0001';
        while exclude<UInt(tag)> == '1' do
            tag = tag + '0001';

    return tag;
```

aarch64/functions/system/AArch64.ExecutingATS1xPInstr

```
// AArch64.ExecutingATS1xPInstr()
// ==============================
// Return TRUE if current instruction is AT S1E1R/WP

boolean AArch64.ExecutingATS1xPInstr()
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if !HavePrivATExt() then return FALSE;

    instr = ThisInstr();
    if instr<22+:10> == '1101010100' then
        op1  = instr<16+:3>;
        CRn  = instr<12+:4>;
        CRm  = instr<8+:4>;
        op2  = instr<5+:3>;
        return op1 == '000' && CRn == '0111' && CRm == '1001' && op2 IN {'000','001'};
    else
        return FALSE;
```

aarch64/functions/system/AArch64.ExecutingBROrBLROrRetInstr

```
// AArch64.ExecutingBROrBLROrRetInstr()
// ====================================
// Returns TRUE if current instruction is a BR, BLR, RET, B[L]RA[B][Z], or RETA[B].

boolean AArch64.ExecutingBROrBLROrRetInstr()
    if !HaveBTIExt() then return FALSE;

    instr = ThisInstr();
    if instr<31:25> == '1101011' && instr<20:16> == '11111' then
        opc = instr<24:21>;
        return opc != '0101';
    else
        return FALSE;
```

aarch64/functions/system/AArch64.ExecutingBTIInstr

```
// AArch64.ExecutingBTIInstr()
// ===========================
// Returns TRUE if current instruction is a BTI.

boolean AArch64.ExecutingBTIInstr()
    if !HaveBTIExt() then return FALSE;

    instr = ThisInstr();
    if instr<31:22> == '1101010100' && instr<21:12> == '0000110010' && instr<4:0> == '11111' then
        CRm  = instr<11:8>;
        op2  = instr<7:5>;
        return (CRm == '0100' && op2<0> == '0');
    else
        return FALSE;
```

aarch64/functions/system/AArch64.ExecutingERETInstr

```
// AArch64.ExecutingERETInstr()
// ============================
// Returns TRUE if current instruction is ERET.

boolean AArch64.ExecutingERETInstr()
    instr = ThisInstr();
    return instr<31:12> == '11010110100111110000';
```

aarch64/functions/system/AArch64.GetMinELSecurityState

```
// AArch64.GetMinELSecurityState()
// ===============================
// Return the minimum exception level and security state required.

(bits(2), boolean) AArch64.GetMinELSecurityState(bits(3) op1)
    boolean need_secure = FALSE;
    bits(2) min_EL;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    case op1 of
        when '00x', '010'
            min_EL = EL1;
        when '011'
            min_EL = EL0;
        when '100'
            min_EL = EL2;
        when '101'
            if !HaveVirtHostExt() then
                UNDEFINED;
            min_EL = EL2;
        when '110'
            min_EL = EL3;
        when '111'
            min_EL = EL1;
            need_secure = TRUE;
    return (min_EL, need_secure);
```

aarch64/functions/system/AArch64.NextRandomTagBit

```
// AArch64.NextRandomTagBit()
// ==========================
// Generate a random bit suitable for generating a random Allocation Tag.

bit AArch64.NextRandomTagBit()
    bits(16) lfsr = RGSR_EL1.SEED;
    bit top = lfsr<5> EOR lfsr<3> EOR lfsr<2> EOR lfsr<0>;
    RGSR_EL1.SEED = top:lfsr<15:1>;
    return top;
```

aarch64/functions/system/AArch64.RandomTag

```
// AArch64.RandomTag()
// ===================
// Generate a random Allocation Tag.

bits(4) AArch64.RandomTag()
    bits(4) tag;
    for i = 0 to 3
        tag<i> = AArch64.NextRandomTagBit();
    return tag;
```

aarch64/functions/system/AArch64.SysInstr

```
// Execute a system instruction with write (source operand).
AArch64.SysInstr(integer op0, integer op1, integer crn, integer crm, integer op2, bits(64) val);
```

aarch64/functions/system/AArch64.SysInstrWithResult

```
// Execute a system instruction with read (result operand).
// Returns the result of the instruction.
bits(64) AArch64.SysInstrWithResult(integer op0, integer op1, integer crn, integer crm, integer op2);
```

aarch64/functions/system/AArch64.SysRegRead

```
// Read from a system register and return the contents of the register.
bits(64) AArch64.SysRegRead(integer op0, integer op1, integer crn, integer crm, integer op2);
```

aarch64/functions/system/AArch64.SysRegWrite

```
// Write to a system register.
AArch64.SysRegWrite(integer op0, integer op1, integer crn, integer crm, integer op2, bits(64) val);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch64/functions/system/BTypeCompatible

```
boolean BTypeCompatible;
```

aarch64/functions/system/BTypeCompatible_BTI

```
// BTypeCompatible_BTI
// ===================
// This function determines whether a given hint encoding is compatible with the current value of
// PSTATE.BTYPE. A value of TRUE here indicates a valid Branch Target Identification instruction.

boolean BTypeCompatible_BTI(bits(2) hintcode)
    case hintcode of
        when '00'
            return FALSE;
        when '01'
            return PSTATE.BTYPE != '11';
        when '10'
            return PSTATE.BTYPE != '10';
        when '11'
            return TRUE;
```

aarch64/functions/system/BTypeCompatible_PACIXSP

```
// BTypeCompatible_PACIXSP()
// =========================
// Returns TRUE if PACIASP, PACIBSP instruction is implicit compatible with PSTATE.BTYPE,
// FALSE otherwise.

boolean BTypeCompatible_PACIXSP()
    if PSTATE.BTYPE IN {'01', '10'} then
        return TRUE;
    elsif PSTATE.BTYPE == '11' then
        index = if PSTATE.EL == EL0 then 35 else 36;
        return SCTLR[]<index> == '0';
    else
        return FALSE;
```

aarch64/functions/system/BTypeNext

```
bits(2) BTypeNext;
```

aarch64/functions/system/InGuardedPage

```
boolean InGuardedPage;
```

aarch64/functions/system/SetBTypeCompatible

```
// SetBTypeCompatible()
// ====================
// Sets the value of BTypeCompatible global variable used by BTI

SetBTypeCompatible(boolean x)
    BTypeCompatible = x;
```

aarch64/functions/system/SetBTypeNext

```
// SetBTypeNext()
// ==============
// Set the value of BTypeNext global variable used by BTI

SetBTypeNext(bits(2) x)
    BTypeNext = x;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch64/functions/system/_ChooseRandomNonExcludedTag

```
// The _ChooseRandomNonExcludedTag function is used when GCR_EL1.RRND == '1' to generate random
// Allocation Tags.
//
// The resulting Allocation Tag is selected from the set [0,15], excluding any Allocation Tag where
// exclude[tag_value] == 1. If 'exclude' is all ones, the returned Allocation Tag is '0000'.
//
// This function is expected to generate a non-deterministic selection from the set of non-excluded
// Allocation Tags. A reasonable implementation is described by the Pseudocode used when
// GCR_EL1.RRND is 0, but with a non-deterministic implementation of NextRandomTagBit().
bits(4) _ChooseRandomNonExcludedTag(bits(16) exclude);
```

J1.1.4 aarch64/instrs

This section includes the following pseudocode functions:

- *aarch64/instrs/branch/eret/AArch64.ExceptionReturn*.
- *aarch64/instrs/countop/CountOp* on page J1-7598.
- *aarch64/instrs/extendreg/DecodeRegExtend* on page J1-7598.
- *aarch64/instrs/extendreg/ExtendReg* on page J1-7598.
- *aarch64/instrs/extendreg/ExtendType* on page J1-7599.
- *aarch64/instrs/float/arithmetic/max-min/fpmaxminop/FPMaxMinOp* on page J1-7599.
- *aarch64/instrs/float/arithmetic/unary/fpunaryop/FPUnaryOp* on page J1-7599.
- *aarch64/instrs/float/convert/fpconvop/FPConvOp* on page J1-7599.
- *aarch64/instrs/integer/bitfield/bfxpreferred/BFXPreferred* on page J1-7599.
- *aarch64/instrs/integer/bitmasks/DecodeBitMasks* on page J1-7600.
- *aarch64/instrs/integer/ins-ext/insert/movewide/movewideop/MoveWideOp* on page J1-7600.
- *aarch64/instrs/integer/logical/movwpreferred/MoveWidePreferred* on page J1-7600.
- *aarch64/instrs/integer/shiftreg/DecodeShift* on page J1-7601.
- *aarch64/instrs/integer/shiftreg/ShiftReg* on page J1-7601.
- *aarch64/instrs/integer/shiftreg/ShiftType* on page J1-7601.
- *aarch64/instrs/logicalop/LogicalOp* on page J1-7601.
- *aarch64/instrs/memory/memop/MemAtomicOp* on page J1-7601.
- *aarch64/instrs/memory/memop/MemOp* on page J1-7602.
- *aarch64/instrs/memory/prefetch/Prefetch* on page J1-7602.
- *aarch64/instrs/system/barriers/barrierop/MemBarrierOp* on page J1-7602.
- *aarch64/instrs/system/hints/syshintop/SystemHintOp* on page J1-7602.
- *aarch64/instrs/system/register/cpsr/pstatefield/PSTATEField* on page J1-7602.
- *aarch64/instrs/system/sysops/sysop/SysOp* on page J1-7603.
- *aarch64/instrs/system/sysops/sysop/SystemOp* on page J1-7604.
- *aarch64/instrs/vector/arithmetic/binary/uniform/logical/bsl-eor/vbitop/VBitOp* on page J1-7604.
- *aarch64/instrs/vector/arithmetic/unary/cmp/compareop/CompareOp* on page J1-7604.
- *aarch64/instrs/vector/logical/immediateop/ImmediateOp* on page J1-7604.
- *aarch64/instrs/vector/reduce/reduceop/Reduce* on page J1-7604.
- *aarch64/instrs/vector/reduce/reduceop/ReduceOp* on page J1-7604.

aarch64/instrs/branch/eret/AArch64.ExceptionReturn

```
// AArch64.ExceptionReturn()
// =========================

AArch64.ExceptionReturn(bits(64) new_pc, bits(32) spsr)

    SynchronizeContext();

    sync_errors = HaveIESB() && SCTLR[].IESB == '1';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if HaveDoubleFaultExt() then
        sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3);
    if sync_errors then
        SynchronizeErrors();
        iesb_req = TRUE;
        TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
    // Attempts to change to an illegal state will invoke the Illegal Execution state mechanism
    SetPSTATEFromPSR(spsr);
    ClearExclusiveLocal(ProcessorID());
    SendEventLocal();

    if PSTATE.IL == '1' && spsr<4> == '1' && spsr<20> == '0' then
        // If the exception return is illegal, PC[63:32,1:0] are UNKNOWN
        new_pc<63:32> = bits(32) UNKNOWN;
        new_pc<1:0> = bits(2) UNKNOWN;
    elsif UsingAArch32() then               // Return to AArch32
        // ELR_ELx[1:0] or ELR_ELx[0] are treated as being 0, depending on the target instruction set
state
        if PSTATE.T == '1' then
            new_pc<0> = '0';                // T32
        else
            new_pc<1:0> = '00';             // A32
    else                                    // Return to AArch64
        // ELR_ELx[63:56] might include a tag
        new_pc = AArch64.BranchAddr(new_pc);

    if UsingAArch32() then
        // 32 most significant bits are ignored.
        BranchTo(new_pc<31:0>, BranchType_ERET);
    else
        BranchToAddr(new_pc, BranchType_ERET);
```

aarch64/instrs/countop/CountOp

```
enumeration CountOp     {CountOp_CLZ, CountOp_CLS, CountOp_CNT};
```

aarch64/instrs/extendreg/DecodeRegExtend

```
// DecodeRegExtend()
// =================
// Decode a register extension option

ExtendType DecodeRegExtend(bits(3) op)
    case op of
        when '000' return ExtendType_UXTB;
        when '001' return ExtendType_UXTH;
        when '010' return ExtendType_UXTW;
        when '011' return ExtendType_UXTX;
        when '100' return ExtendType_SXTB;
        when '101' return ExtendType_SXTH;
        when '110' return ExtendType_SXTW;
        when '111' return ExtendType_SXTX;
```

aarch64/instrs/extendreg/ExtendReg

```
// ExtendReg()
// ===========
// Perform a register extension and shift

bits(N) ExtendReg(integer reg, ExtendType exttype, integer shift)
    assert shift >= 0 && shift <= 4;
    bits(N) val = X[reg];
    boolean unsigned;
    integer len;

    case exttype of
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        when ExtendType_SXTB unsigned = FALSE; len = 8;
        when ExtendType_SXTH unsigned = FALSE; len = 16;
        when ExtendType_SXTW unsigned = FALSE; len = 32;
        when ExtendType_SXTX unsigned = FALSE; len = 64;
        when ExtendType_UXTB unsigned = TRUE;  len = 8;
        when ExtendType_UXTH unsigned = TRUE;  len = 16;
        when ExtendType_UXTW unsigned = TRUE;  len = 32;
        when ExtendType_UXTX unsigned = TRUE;  len = 64;

    // Note the extended width of the intermediate value and
    // that sign extension occurs from bit <len+shift-1>, not
    // from bit <len-1>. This is equivalent to the instruction
    //   [SU]BFIZ Rtmp, Rreg, #shift, #len
    // It may also be seen as a sign/zero extend followed by a shift:
    //   LSL(Extend(val<len-1:0>, N, unsigned), shift);

    len = Min(len, N - shift);
    return Extend(val<len-1:0> : Zeros(shift), N, unsigned);
```

aarch64/instrs/extendreg/ExtendType

```
enumeration ExtendType  {ExtendType_SXTB, ExtendType_SXTH, ExtendType_SXTW, ExtendType_SXTX,
                         ExtendType_UXTB, ExtendType_UXTH, ExtendType_UXTW, ExtendType_UXTX};
```

aarch64/instrs/float/arithmetic/max-min/fpmaxminop/FPMaxMinOp

```
enumeration FPMaxMinOp  {FPMaxMinOp_MAX, FPMaxMinOp_MIN,
                         FPMaxMinOp_MAXNUM, FPMaxMinOp_MINNUM};
```

aarch64/instrs/float/arithmetic/unary/fpunaryop/FPUnaryOp

```
enumeration FPUnaryOp   {FPUnaryOp_ABS, FPUnaryOp_MOV,
                         FPUnaryOp_NEG, FPUnaryOp_SQRT};
```

aarch64/instrs/float/convert/fpconvop/FPConvOp

```
enumeration FPConvOp    {FPConvOp_CVT_FtoI, FPConvOp_CVT_ItoF,
                         FPConvOp_MOV_FtoI, FPConvOp_MOV_ItoF
                         , FPConvOp_CVT_FtoI_JS
};
```

aarch64/instrs/integer/bitfield/bfxpreferred/BFXPreferred

```
// BFXPreferred()
// ==============
//
// Return TRUE if UBFX or SBFX is the preferred disassembly of a
// UBFM or SBFM bitfield instruction. Must exclude more specific
// aliases UBFIZ, SBFIZ, UXT[BH], SXT[BHW], LSL, LSR and ASR.

boolean BFXPreferred(bit sf, bit uns, bits(6) imms, bits(6) immr)
    integer S = UInt(imms);
    integer R = UInt(immr);

    // must not match UBFIZ/SBFIX alias
    if UInt(imms) < UInt(immr) then
        return FALSE;

    // must not match LSR/ASR/LSL alias (imms == 31 or 63)
    if imms == sf:'11111' then
        return FALSE;

    // must not match UXTx/SXTx alias
    if immr == '000000' then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        // must not match 32-bit UXT[BH] or SXT[BH]
        if sf == '0' && imms IN {'000111', '001111'} then
            return FALSE;
        // must not match 64-bit SXT[BHW]
        if sf:uns == '10' && imms IN {'000111', '001111', '011111'} then
            return FALSE;

    // must be UBFX/SBFX alias
    return TRUE;
```

aarch64/instrs/integer/bitmasks/DecodeBitMasks

```
// DecodeBitMasks()
// ================

// Decode AArch64 bitfield and logical immediate masks which use a similar encoding structure

(bits(M), bits(M)) DecodeBitMasks(bit immN, bits(6) imms, bits(6) immr, boolean immediate)
    bits(M) tmask, wmask;
    bits(6) levels;

    // Compute log2 of element size
    // 2^len must be in range [2, M]
    len = HighestSetBit(immN:NOT(imms));
    if len < 1 then UNDEFINED;
    assert M >= (1 << len);

    // Determine S, R and S - R parameters
    levels = ZeroExtend(Ones(len), 6);

    // For logical immediates an all-ones value of S is reserved
    // since it would generate a useless all-ones result (many times)
    if immediate && (imms AND levels) == levels then
        UNDEFINED;

    S = UInt(imms AND levels);
    R = UInt(immr AND levels);
    diff = S - R;    // 6-bit subtract with borrow

    esize = 1 << len;
    d = UInt(diff<len-1:0>);
    welem = ZeroExtend(Ones(S + 1), esize);
    telem = ZeroExtend(Ones(d + 1), esize);
    wmask = Replicate(ROR(welem, R));
    tmask = Replicate(telem);
    return (wmask, tmask);
```

aarch64/instrs/integer/ins-ext/insert/movewide/movewideop/MoveWideOp

```
enumeration MoveWideOp  {MoveWideOp_N, MoveWideOp_Z, MoveWideOp_K};
```

aarch64/instrs/integer/logical/movwpreferred/MoveWidePreferred

```
// MoveWidePreferred()
// ===================
//
// Return TRUE if a bitmask immediate encoding would generate an immediate
// value that could also be represented by a single MOVZ or MOVN instruction.
// Used as a condition for the preferred MOV<-ORR alias.

boolean MoveWidePreferred(bit sf, bit immN, bits(6) imms, bits(6) immr)
    integer S = UInt(imms);
    integer R = UInt(immr);
    integer width = if sf == '1' then 64 else 32;

    // element size must equal total immediate size
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if sf == '1' && immN:imms != '1xxxxxx' then
        return FALSE;
    if sf == '0' && immN:imms != '00xxxxx' then
        return FALSE;

    // for MOVZ must contain no more than 16 ones
    if S < 16 then
        // ones must not span halfword boundary when rotated
        return (-R MOD 16) <= (15 - S);

    // for MOVN must contain no more than 16 zeros
    if S >= width - 15 then
        // zeros must not span halfword boundary when rotated
        return (R MOD 16) <= (S - (width - 15));

    return FALSE;
```

aarch64/instrs/integer/shiftreg/DecodeShift

```
// DecodeShift()
// =============
// Decode shift encodings

ShiftType DecodeShift(bits(2) op)
    case op of
        when '00'  return ShiftType_LSL;
        when '01'  return ShiftType_LSR;
        when '10'  return ShiftType_ASR;
        when '11'  return ShiftType_ROR;
```

aarch64/instrs/integer/shiftreg/ShiftReg

```
// ShiftReg()
// ==========
// Perform shift of a register operand

bits(N) ShiftReg(integer reg, ShiftType shiftype, integer amount)
    bits(N) result = X[reg];
    case shiftype of
        when ShiftType_LSL result = LSL(result, amount);
        when ShiftType_LSR result = LSR(result, amount);
        when ShiftType_ASR result = ASR(result, amount);
        when ShiftType_ROR result = ROR(result, amount);
    return result;
```

aarch64/instrs/integer/shiftreg/ShiftType

```
enumeration ShiftType   {ShiftType_LSL, ShiftType_LSR, ShiftType_ASR, ShiftType_ROR};
```

aarch64/instrs/logicalop/LogicalOp

```
enumeration LogicalOp   {LogicalOp_AND, LogicalOp_EOR, LogicalOp_ORR};
```

aarch64/instrs/memory/memop/MemAtomicOp

```
enumeration MemAtomicOp {MemAtomicOp_ADD,
                         MemAtomicOp_BIC,
                         MemAtomicOp_EOR,
                         MemAtomicOp_ORR,
                         MemAtomicOp_SMAX,
                         MemAtomicOp_SMIN,
                         MemAtomicOp_UMAX,
                         MemAtomicOp_UMIN,
                         MemAtomicOp_SWP};
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch64/instrs/memory/memop/MemOp

```
enumeration MemOp {MemOp_LOAD, MemOp_STORE, MemOp_PREFETCH};
```

aarch64/instrs/memory/prefetch/Prefetch

```
// Prefetch()
// ==========

// Decode and execute the prefetch hint on ADDRESS specified by PRFOP

Prefetch(bits(64) address, bits(5) prfop)
    PrefetchHint hint;
    integer target;
    boolean stream;

    case prfop<4:3> of
        when '00' hint = Prefetch_READ;         // PLD: prefetch for load
        when '01' hint = Prefetch_EXEC;         // PLI: preload instructions
        when '10' hint = Prefetch_WRITE;        // PST: prepare for store
        when '11' return;                       // unallocated hint
    target = UInt(prfop<2:1>);                  // target cache level
    stream = (prfop<0> != '0');                 // streaming (non-temporal)
    Hint_Prefetch(address, hint, target, stream);
    return;
```

aarch64/instrs/system/barriers/barrierop/MemBarrierOp

```
enumeration MemBarrierOp   {  MemBarrierOp_DSB         // Data Synchronization Barrier
                            , MemBarrierOp_DMB         // Data Memory Barrier
                            , MemBarrierOp_ISB         // Instruction Synchronization Barrier
                            , MemBarrierOp_SSBB        // Speculative Synchronization Barrier to VA
                            , MemBarrierOp_PSSBB       // Speculative Synchronization Barrier to PA
                            , MemBarrierOp_SB          // Speculation Barrier
                           };
```

aarch64/instrs/system/hints/syshintop/SystemHintOp

```
enumeration SystemHintOp {
    SystemHintOp_NOP,
    SystemHintOp_YIELD,
    SystemHintOp_WFE,
    SystemHintOp_WFI,
    SystemHintOp_SEV,
    SystemHintOp_SEVL,
    SystemHintOp_DGH,
    SystemHintOp_ESB,
    SystemHintOp_PSB,
    SystemHintOp_TSB,
    SystemHintOp_BTI,
    SystemHintOp_CSDB
};
```

aarch64/instrs/system/register/cpsr/pstatefield/PSTATEField

```
enumeration PSTATEField {PSTATEField_DAIFSet, PSTATEField_DAIFClr,
                         PSTATEField_PAN, // Armv8.1
                         PSTATEField_UAO, // Armv8.2
                         PSTATEField_DIT, // Armv8.4
                         PSTATEField_SSBS,
                         PSTATEField_TCO, // Armv8.5
                         PSTATEField_SP
                         };
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch64/instrs/system/sysops/sysop/SysOp

```
// SysOp()
// =======

SystemOp SysOp(bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2)
    case op1:CRn:CRm:op2 of
        when '000 0111 1000 000' return Sys_AT;   // S1E1R
        when '100 0111 1000 000' return Sys_AT;   // S1E2R
        when '110 0111 1000 000' return Sys_AT;   // S1E3R
        when '000 0111 1000 001' return Sys_AT;   // S1E1W
        when '100 0111 1000 001' return Sys_AT;   // S1E2W
        when '110 0111 1000 001' return Sys_AT;   // S1E3W
        when '000 0111 1000 010' return Sys_AT;   // S1E0R
        when '000 0111 1000 011' return Sys_AT;   // S1E0W
        when '100 0111 1000 100' return Sys_AT;   // S12E1R
        when '100 0111 1000 101' return Sys_AT;   // S12E1W
        when '100 0111 1000 110' return Sys_AT;   // S12E0R
        when '100 0111 1000 111' return Sys_AT;   // S12E0W
        when '011 0111 0100 001' return Sys_DC;   // ZVA
        when '000 0111 0110 001' return Sys_DC;   // IVAC
        when '000 0111 0110 010' return Sys_DC;   // ISW
        when '011 0111 1010 001' return Sys_DC;   // CVAC
        when '000 0111 1010 010' return Sys_DC;   // CSW
        when '011 0111 1011 001' return Sys_DC;   // CVAU
        when '011 0111 1110 001' return Sys_DC;   // CIVAC
        when '000 0111 1110 010' return Sys_DC;   // CISW
        when '011 0111 1101 001' return Sys_DC;   // CVADP
        when '000 0111 0001 000' return Sys_IC;   // IALLUIS
        when '000 0111 0101 000' return Sys_IC;   // IALLU
        when '011 0111 0101 001' return Sys_IC;   // IVAU
        when '100 1000 0000 001' return Sys_TLBI; // IPAS2E1IS
        when '100 1000 0000 101' return Sys_TLBI; // IPAS2LE1IS
        when '000 1000 0011 000' return Sys_TLBI; // VMALLE1IS
        when '100 1000 0011 000' return Sys_TLBI; // ALLE2IS
        when '110 1000 0011 000' return Sys_TLBI; // ALLE3IS
        when '000 1000 0011 001' return Sys_TLBI; // VAE1IS
        when '100 1000 0011 001' return Sys_TLBI; // VAE2IS
        when '110 1000 0011 001' return Sys_TLBI; // VAE3IS
        when '000 1000 0011 010' return Sys_TLBI; // ASIDE1IS
        when '000 1000 0011 011' return Sys_TLBI; // VAAE1IS
        when '100 1000 0011 100' return Sys_TLBI; // ALLE1IS
        when '000 1000 0011 101' return Sys_TLBI; // VALE1IS
        when '100 1000 0011 101' return Sys_TLBI; // VALE2IS
        when '110 1000 0011 101' return Sys_TLBI; // VALE3IS
        when '100 1000 0011 110' return Sys_TLBI; // VMALLS12E1IS
        when '000 1000 0011 111' return Sys_TLBI; // VAALE1IS
        when '100 1000 0100 001' return Sys_TLBI; // IPAS2E1
        when '100 1000 0100 101' return Sys_TLBI; // IPAS2LE1
        when '000 1000 0111 000' return Sys_TLBI; // VMALLE1
        when '100 1000 0111 000' return Sys_TLBI; // ALLE2
        when '110 1000 0111 000' return Sys_TLBI; // ALLE3
        when '000 1000 0111 001' return Sys_TLBI; // VAE1
        when '100 1000 0111 001' return Sys_TLBI; // VAE2
        when '110 1000 0111 001' return Sys_TLBI; // VAE3
        when '000 1000 0111 010' return Sys_TLBI; // ASIDE1
        when '000 1000 0111 011' return Sys_TLBI; // VAAE1
        when '100 1000 0111 100' return Sys_TLBI; // ALLE1
        when '000 1000 0111 101' return Sys_TLBI; // VALE1
        when '100 1000 0111 101' return Sys_TLBI; // VALE2
        when '110 1000 0111 101' return Sys_TLBI; // VALE3
        when '100 1000 0111 110' return Sys_TLBI; // VMALLS12E1
        when '000 1000 0111 111' return Sys_TLBI; // VAALE1
    return Sys_SYS;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch64/instrs/system/sysops/sysop/SystemOp

```
enumeration SystemOp {Sys_AT, Sys_DC, Sys_IC, Sys_TLBI, Sys_SYS};
```

aarch64/instrs/vector/arithmetic/binary/uniform/logical/bsl-eor/vbitop/VBitOp

```
enumeration VBitOp      {VBitOp_VBIF, VBitOp_VBIT, VBitOp_VBSL, VBitOp_VEOR};
```

aarch64/instrs/vector/arithmetic/unary/cmp/compareop/CompareOp

```
enumeration CompareOp   {CompareOp_GT, CompareOp_GE, CompareOp_EQ,
                         CompareOp_LE, CompareOp_LT};
```

aarch64/instrs/vector/logical/immediateop/ImmediateOp

```
enumeration ImmediateOp {ImmediateOp_MOVI, ImmediateOp_MVNI,
                         ImmediateOp_ORR, ImmediateOp_BIC};
```

aarch64/instrs/vector/reduce/reduceop/Reduce

```
// Reduce()
// ========

bits(esize) Reduce(ReduceOp op, bits(N) input, integer esize)
    integer half;
    bits(esize) hi;
    bits(esize) lo;
    bits(esize) result;

    if N == esize then
        return input<esize-1:0>;

    half = N DIV 2;
    hi = Reduce(op, input<N-1:half>, esize);
    lo = Reduce(op, input<half-1:0>, esize);

    case op of
        when ReduceOp_FMINNUM
            result = FPMinNum(lo, hi, FPCR);
        when ReduceOp_FMAXNUM
            result = FPMaxNum(lo, hi, FPCR);
        when ReduceOp_FMIN
            result = FPMin(lo, hi, FPCR);
        when ReduceOp_FMAX
            result = FPMax(lo, hi, FPCR);
        when ReduceOp_FADD
            result = FPAdd(lo, hi, FPCR);
        when ReduceOp_ADD
            result = lo + hi;

    return result;
```

aarch64/instrs/vector/reduce/reduceop/ReduceOp

```
enumeration ReduceOp {ReduceOp_FMINNUM, ReduceOp_FMAXNUM,
                      ReduceOp_FMIN, ReduceOp_FMAX,
                      ReduceOp_FADD, ReduceOp_ADD};
```

J1.1.5 aarch64/translation

This section includes the following pseudocode functions:

- *aarch64/translation/attrs/AArch64.CombineS1S2Desc* on page J1-7605.
- *aarch64/translation/attrs/AArch64.InstructionDevice* on page J1-7606.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *aarch64/translation/attrs/AArch64.S1AttrDecode* on page J1-7606.
- *aarch64/translation/attrs/AArch64.TranslateAddressS1Off* on page J1-7607.
- *aarch64/translation/checks/AArch64.AccessIsPrivileged* on page J1-7608.
- *aarch64/translation/checks/AArch64.AccessUsesEL* on page J1-7608.
- *aarch64/translation/checks/AArch64.CheckPermission* on page J1-7609.
- *aarch64/translation/checks/AArch64.CheckS2Permission* on page J1-7610.
- *aarch64/translation/debug/AArch64.CheckBreakpoint* on page J1-7611.
- *aarch64/translation/debug/AArch64.CheckDebug* on page J1-7611.
- *aarch64/translation/debug/AArch64.CheckWatchpoint* on page J1-7611.
- *aarch64/translation/faults/AArch64.AccessFlagFault* on page J1-7612.
- *aarch64/translation/faults/AArch64.AddressSizeFault* on page J1-7612.
- *aarch64/translation/faults/AArch64.AlignmentFault* on page J1-7612.
- *aarch64/translation/faults/AArch64.AsynchExternalAbort* on page J1-7613.
- *aarch64/translation/faults/AArch64.DebugFault* on page J1-7613.
- *aarch64/translation/faults/AArch64.NoFault* on page J1-7613.
- *aarch64/translation/faults/AArch64.PermissionFault* on page J1-7613.
- *aarch64/translation/faults/AArch64.TranslationFault* on page J1-7614.
- *aarch64/translation/translation/AArch64.CheckAndUpdateDescriptor* on page J1-7614.
- *aarch64/translation/translation/AArch64.FirstStageTranslate* on page J1-7615.
- *aarch64/translation/translation/AArch64.FullTranslate* on page J1-7616.
- *aarch64/translation/translation/AArch64.SecondStageTranslate* on page J1-7616.
- *aarch64/translation/translation/AArch64.SecondStageWalk* on page J1-7617.
- *aarch64/translation/translation/AArch64.TranslateAddress* on page J1-7617.
- *aarch64/translation/walk/AArch64.TranslationTableWalk* on page J1-7618.

aarch64/translation/attrs/AArch64.CombineS1S2Desc

```
// AArch64.CombineS1S2Desc()
// =========================
// Combines the address descriptors from stage 1 and stage 2

AddressDescriptor AArch64.CombineS1S2Desc(AddressDescriptor s1desc, AddressDescriptor s2desc)

    AddressDescriptor result;

    result.paddress = s2desc.paddress;

    apply_force_writeback = HaveStage2MemAttrControl() && HCR_EL2.FWB == '1';
    if IsFault(s1desc) || IsFault(s2desc) then
        result = if IsFault(s1desc) then s1desc else s2desc;
    else
        result.fault = AArch64.NoFault();
        if s2desc.memattrs.memtype == MemType_Device || (
            (apply_force_writeback && s1desc.memattrs.memtype == MemType_Device &&
s2desc.memattrs.inner.attrs != '10') ||
            (!apply_force_writeback && s1desc.memattrs.memtype == MemType_Device) ) then
            result.memattrs.memtype = MemType_Device;
            if s1desc.memattrs.memtype == MemType_Normal then
                result.memattrs.device = s2desc.memattrs.device;
            elsif s2desc.memattrs.memtype == MemType_Normal then
                result.memattrs.device = s1desc.memattrs.device;
            else                    // Both Device
                result.memattrs.device = CombineS1S2Device(s1desc.memattrs.device,
                                                           s2desc.memattrs.device);
            result.memattrs.tagged = FALSE;
        // S1 can be either Normal or Device, S2 is Normal.
        else
            result.memattrs.memtype = MemType_Normal;
            result.memattrs.device = DeviceType UNKNOWN;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            result.memattrs.inner = CombineS1S2AttrHints(s1desc.memattrs.inner, s2desc.memattrs.inner);
            result.memattrs.outer = CombineS1S2AttrHints(s1desc.memattrs.outer, s2desc.memattrs.outer);
            result.memattrs.shareable = (s1desc.memattrs.shareable || s2desc.memattrs.shareable);
            result.memattrs.outershareable = (s1desc.memattrs.outershareable ||
                                              s2desc.memattrs.outershareable);
            result.memattrs.tagged = (s1desc.memattrs.tagged &&
                                      result.memattrs.inner.attrs == MemAttr_WB &&
                                      result.memattrs.inner.hints == MemHint_RWA &&
                                      result.memattrs.outer.attrs == MemAttr_WB &&
                                      result.memattrs.outer.hints == MemHint_RWA);

    result.memattrs = MemAttrDefaults(result.memattrs);

    return result;
```

aarch64/translation/attrs/AArch64.InstructionDevice

```
// AArch64.InstructionDevice()
// ===========================
// Instruction fetches from memory marked as Device but not execute-never might generate a
// Permission Fault but are otherwise treated as if from Normal Non-cacheable memory.

AddressDescriptor AArch64.InstructionDevice(AddressDescriptor addrdesc, bits(64) vaddress,
                                            bits(52) ipaddress, integer level,
                                            AccType acctype, boolean iswrite, boolean secondstage,
                                            boolean s2fs1walk)

    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_FAULT};

    if c == Constraint_FAULT then
        addrdesc.fault = AArch64.PermissionFault(ipaddress, boolean UNKNOWN,  level, acctype, iswrite,
                                                 secondstage, s2fs1walk);
    else
        addrdesc.memattrs.memtype = MemType_Normal;
        addrdesc.memattrs.inner.attrs = MemAttr_NC;
        addrdesc.memattrs.inner.hints = MemHint_No;
        addrdesc.memattrs.outer = addrdesc.memattrs.inner;
        addrdesc.memattrs.tagged = FALSE;
        addrdesc.memattrs = MemAttrDefaults(addrdesc.memattrs);

    return addrdesc;
```

aarch64/translation/attrs/AArch64.S1AttrDecode

```
// AArch64.S1AttrDecode()
// ======================
// Converts the Stage 1 attribute fields, using the MAIR, to orthogonal
// attributes and hints.

MemoryAttributes AArch64.S1AttrDecode(bits(2) SH, bits(3) attr, AccType acctype)

    MemoryAttributes memattrs;

    mair = MAIR[];
    index = 8 * UInt(attr);
    attrfield = mair<index+7:index>;

    memattrs.tagged = FALSE;
    if ((attrfield<7:4> != '0000' && attrfield<7:4> != '1111' && attrfield<3:0> == '0000') ||
        (attrfield<7:4> == '0000' && attrfield<3:0> != 'xx00')) then
        // Reserved, maps to an allocated value
        (-, attrfield) = ConstrainUnpredictableBits();
    if !HaveMTEExt() && attrfield<7:4> == '1111' && attrfield<3:0> == '0000' then
        // Reserved, maps to an allocated value
        (-, attrfield) = ConstrainUnpredictableBits();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if attrfield<7:4> == '0000' then            // Device
        memattrs.memtype = MemType_Device;
        case attrfield<3:0> of
            when '0000'  memattrs.device = DeviceType_nGnRnE;
            when '0100'  memattrs.device = DeviceType_nGnRE;
            when '1000'  memattrs.device = DeviceType_nGRE;
            when '1100'  memattrs.device = DeviceType_GRE;
            otherwise    Unreachable();         // Reserved, handled above

    elsif attrfield<3:0> != '0000'  then        // Normal
        memattrs.memtype = MemType_Normal;
        memattrs.outer = LongConvertAttrsHints(attrfield<7:4>, acctype);
        memattrs.inner = LongConvertAttrsHints(attrfield<3:0>, acctype);
        memattrs.shareable = SH<1> == '1';
        memattrs.outershareable = SH == '10';
    elsif HaveMTEExt() && attrfield == '11110000' then // Normal, Tagged if WB-RWA
        memattrs.memtype = MemType_Normal;
        memattrs.outer = LongConvertAttrsHints('1111', acctype); // WB_RWA
        memattrs.inner = LongConvertAttrsHints('1111', acctype); // WB_RWA
        memattrs.shareable = SH<1> == '1';
        memattrs.outershareable = SH == '10';
        memattrs.tagged = (memattrs.inner.attrs == MemAttr_WB &&
                           memattrs.inner.hints == MemHint_RWA &&
                           memattrs.outer.attrs == MemAttr_WB &&
                           memattrs.outer.hints == MemHint_RWA);
    else
        Unreachable();                          // Reserved, handled above

    return MemAttrDefaults(memattrs);
```

aarch64/translation/attrs/AArch64.TranslateAddressS1Off

```
// AArch64.TranslateAddressS1Off()
// ==============================
// Called for stage 1 translations when translation is disabled to supply a default translation.
// Note that there are additional constraints on instruction prefetching that are not described in
// this pseudocode.

TLBRecord AArch64.TranslateAddressS1Off(bits(64) vaddress, AccType acctype, boolean iswrite)
    assert !ELUsingAArch32(S1TranslationRegime());

    TLBRecord result;

    Top = AddrTop(vaddress, (acctype == AccType_IFETCH), PSTATE.EL);
    if !IsZero(vaddress<Top:PAMax()>) then
        level = 0;
        ipaddress = bits(52) UNKNOWN;
        secondstage = FALSE;
        s2fs1walk = FALSE;
        result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress,boolean UNKNOWN,  level, acctype,
                                                         iswrite, secondstage, s2fs1walk);
        return result;

    default_cacheable = (HasS2Translation() && HCR_EL2.DC == '1');

    if default_cacheable then
        // Use default cacheable settings
        result.addrdesc.memattrs.memtype = MemType_Normal;
        result.addrdesc.memattrs.inner.attrs = MemAttr_WB;      // Write-back
        result.addrdesc.memattrs.inner.hints = MemHint_RWA;
        result.addrdesc.memattrs.shareable = FALSE;
        result.addrdesc.memattrs.outershareable = FALSE;
        result.addrdesc.memattrs.tagged = HCR_EL2.DCT == '1';
    elsif acctype != AccType_IFETCH then
        // Treat data as Device
        result.addrdesc.memattrs.memtype = MemType_Device;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        result.addrdesc.memattrs.device = DeviceType_nGnRnE;
        result.addrdesc.memattrs.inner = MemAttrHints UNKNOWN;
        result.addrdesc.memattrs.tagged = FALSE;
    else
        // Instruction cacheability controlled by SCTLR_ELx.I
        cacheable = SCTLR[].I == '1';
        result.addrdesc.memattrs.memtype = MemType_Normal;
        if cacheable then
            result.addrdesc.memattrs.inner.attrs = MemAttr_WT;
            result.addrdesc.memattrs.inner.hints = MemHint_RA;
        else
            result.addrdesc.memattrs.inner.attrs = MemAttr_NC;
            result.addrdesc.memattrs.inner.hints = MemHint_No;
        result.addrdesc.memattrs.shareable = TRUE;
        result.addrdesc.memattrs.outershareable = TRUE;
        result.addrdesc.memattrs.tagged = FALSE;

    result.addrdesc.memattrs.outer = result.addrdesc.memattrs.inner;

    result.addrdesc.memattrs = MemAttrDefaults(result.addrdesc.memattrs);

    result.perms.ap = bits(3) UNKNOWN;
    result.perms.xn = '0';
    result.perms.pxn = '0';

    result.nG = bit UNKNOWN;
    result.contiguous = boolean UNKNOWN;
    result.domain = bits(4) UNKNOWN;
    result.level = integer UNKNOWN;
    result.blocksize = integer UNKNOWN;
    result.addrdesc.paddress.address = vaddress<51:0>;
    result.addrdesc.paddress.NS = if IsSecure() then '0' else '1';
    result.addrdesc.fault = AArch64.NoFault();
    return result;
```

aarch64/translation/checks/AArch64.AccessIsPrivileged

```
// AArch64.AccessIsPrivileged()
// ============================

boolean AArch64.AccessIsPrivileged(AccType acctype)

    el = AArch64.AccessUsesEL(acctype);

    if el == EL0 then
        ispriv = FALSE;
    elsif el == EL3 then
        ispriv = TRUE;
    elsif el == EL2 && (!IsInHost() || HCR_EL2.TGE == '0') then
        ispriv = TRUE;
    elsif HaveUAOExt() && PSTATE.UAO == '1' then
        ispriv = TRUE;
    else
        ispriv = (acctype != AccType_UNPRIV);

    return ispriv;
```

aarch64/translation/checks/AArch64.AccessUsesEL

```
// AArch64.AccessUsesEL()
// ======================
// Returns the Exception Level of the regime that will manage the translation for a given access type.

bits(2) AArch64.AccessUsesEL(AccType acctype)
    if acctype == AccType_UNPRIV then
        return EL0;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif acctype == AccType_NV2REGISTER then
        return EL2;
    else
        return PSTATE.EL;
```

aarch64/translation/checks/AArch64.CheckPermission

```
// AArch64.CheckPermission()
// =========================
// Function used for permission checking from AArch64 stage 1 translations

FaultRecord AArch64.CheckPermission(Permissions perms, bits(64) vaddress, integer level,
                                    bit NS, AccType acctype, boolean iswrite)
    assert !ELUsingAArch32(S1TranslationRegime());

    wxn = SCTLR[].WXN == '1';

    if (PSTATE.EL == EL0 ||
        IsInHost() ||
        (PSTATE.EL == EL1 && !HaveNV2Ext()) ||
        (PSTATE.EL == EL1 && HaveNV2Ext() && (acctype != AccType_NV2REGISTER || !ELIsInHost(EL2)))) then
        priv_r = TRUE;
        priv_w = perms.ap<2> == '0';
        user_r = perms.ap<1> == '1';
        user_w = perms.ap<2:1> == '01';

        ispriv = AArch64.AccessIsPrivileged(acctype);

        pan = if HavePANExt() then PSTATE.PAN else '0';
        if (EL2Enabled() && ((PSTATE.EL == EL1 && HaveNVExt() && HCR_EL2.<NV, NV1> == '11') ||
            (HaveNV2Ext() && acctype == AccType_NV2REGISTER && HCR_EL2.NV2 == '1'))) then
            pan = '0';
        is_ldst   = !(acctype IN {AccType_DC, AccType_DC_UNPRIV, AccType_AT, AccType_IFETCH});
        is_ats1xp = (acctype == AccType_AT && AArch64.ExecutingATS1xPInstr());
        if pan == '1' && user_r && ispriv && (is_ldst || is_ats1xp) then
            priv_r = FALSE;
            priv_w = FALSE;

        user_xn = perms.xn == '1' || (user_w && wxn);
        priv_xn = perms.pxn == '1' || (priv_w && wxn) || user_w;

        if ispriv then
            (r, w, xn) = (priv_r, priv_w, priv_xn);
        else
            (r, w, xn) = (user_r, user_w, user_xn);
    else
        // Access from EL2 or EL3
        r = TRUE;
        w = perms.ap<2> == '0';
        xn = perms.xn == '1' || (w && wxn);

    // Restriction on Secure instruction fetch
    if HaveEL(EL3) && IsSecure() && NS == '1' && SCR_EL3.SIF == '1' then
        xn = TRUE;

    if acctype == AccType_IFETCH then
        fail = xn;
        failedread = TRUE;
    elsif acctype IN { AccType_ATOMICRW, AccType_ORDEREDRW, AccType_ORDEREDATOMICRW } then
        fail = !r || !w;
        failedread = !r;
    elsif iswrite then
        fail = !w;
        failedread = FALSE;
    elsif acctype == AccType_DC && PSTATE.EL != EL0 then
        // DC maintenance instructions operating by VA, cannot fault from stage 1 translation,
        // other than DC IVAC, which requires write permission, and operations executed at EL0,
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        // which require read permission.
        fail = FALSE;
    else
        fail = !r;
        failedread = TRUE;

    if fail then
        secondstage = FALSE;
        s2fs1walk = FALSE;
        ipaddress = bits(52) UNKNOWN;
        return AArch64.PermissionFault(ipaddress,boolean UNKNOWN,  level, acctype,
                                       !failedread, secondstage, s2fs1walk);
    else
        return AArch64.NoFault();
```

aarch64/translation/checks/AArch64.CheckS2Permission

```
// AArch64.CheckS2Permission()
// ==========================
// Function used for permission checking from AArch64 stage 2 translations

FaultRecord AArch64.CheckS2Permission(Permissions perms, bits(64) vaddress, bits(52) ipaddress,
                                      integer level, AccType acctype, boolean iswrite, boolean NS,
                                      boolean s2fs1walk, boolean hwupdatewalk)

    assert (IsSecureEL2Enabled() || (HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2))) &&
HasS2Translation();

    r = perms.ap<1> == '1';
    w = perms.ap<2> == '1';
    if HaveExtendedExecuteNeverExt() then
        case perms.xn:perms.xxn of
            when '00'  xn = FALSE;
            when '01'  xn = PSTATE.EL == EL1;
            when '10'  xn = TRUE;
            when '11'  xn = PSTATE.EL == EL0;
    else
        xn = perms.xn == '1';
    // Stage 1 walk is checked as a read, regardless of the original type
    if acctype == AccType_IFETCH && !s2fs1walk then
        fail = xn;
        failedread = TRUE;
    elsif (acctype IN { AccType_ATOMICRW, AccType_ORDEREDRW, AccType_ORDEREDATOMICRW }) && !s2fs1walk
then
        fail = !r || !w;
        failedread = !r;
    elsif iswrite && !s2fs1walk then
        fail = !w;
        failedread = FALSE;
    elsif acctype == AccType_DC && PSTATE.EL != EL0 && !s2fs1walk then
        // DC maintenance instructions operating by VA, with the exception of DC IVAC, do
        // not generate Permission faults from stage 2 translation, other than when
        // performing a stage 1 translation table walk.
        fail = FALSE;
    elsif hwupdatewalk then
        fail = !w;
        failedread = !iswrite;
    else
        fail = !r;
        failedread = !iswrite;

    if fail then
        domain = bits(4) UNKNOWN;
        secondstage = TRUE;
        return AArch64.PermissionFault(ipaddress,NS,  level, acctype,
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
                                        !failedread, secondstage, s2fs1walk);
    else
        return AArch64.NoFault();
```

aarch64/translation/debug/AArch64.CheckBreakpoint

```
// AArch64.CheckBreakpoint()
// =========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch64
// translation regime, when either debug exceptions are enabled, or halting debug is enabled
// and halting is allowed.

FaultRecord AArch64.CheckBreakpoint(bits(64) vaddress, AccType acctype, integer size)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert (UsingAArch32() && size IN {2,4}) || size == 4;

    match = FALSE;

    for i = 0 to UInt(ID_AA64DFR0_EL1.BRPs)
        match_i = AArch64.BreakpointMatch(i, vaddress, acctype, size);
        match = match || match_i;

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Breakpoint;
        Halt(reason);
    elsif match then
        acctype = AccType_IFETCH;
        iswrite = FALSE;
        return AArch64.DebugFault(acctype, iswrite);
    else
        return AArch64.NoFault();
```

aarch64/translation/debug/AArch64.CheckDebug

```
// AArch64.CheckDebug()
// ====================
// Called on each access to check for a debug exception or entry to Debug state.

FaultRecord AArch64.CheckDebug(bits(64) vaddress, AccType acctype, boolean iswrite, integer size)

    FaultRecord fault = AArch64.NoFault();

    d_side = (acctype != AccType_IFETCH);
    if HaveNV2Ext() && acctype == AccType_NV2REGISTER then
        mask = '0';
        generate_exception = AArch64.GenerateDebugExceptionsFrom(EL2, IsSecure(), mask) && MDSCR_EL1.MDE
== '1';
    else
        generate_exception = AArch64.GenerateDebugExceptions() && MDSCR_EL1.MDE == '1';
    halt = HaltOnBreakpointOrWatchpoint();

    if generate_exception || halt then
        if d_side then
            fault = AArch64.CheckWatchpoint(vaddress, acctype, iswrite, size);
        else
            fault = AArch64.CheckBreakpoint(vaddress, acctype, size);

    return fault;
```

aarch64/translation/debug/AArch64.CheckWatchpoint

```
// AArch64.CheckWatchpoint()
// =========================
// Called before accessing the memory location of "size" bytes at "address",
// when either debug exceptions are enabled for the access, or halting debug
// is enabled and halting is allowed.
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
FaultRecord AArch64.CheckWatchpoint(bits(64) vaddress, AccType acctype,
                                    boolean iswrite, integer size)
    assert !ELUsingAArch32(S1TranslationRegime());

    match = FALSE;
    ispriv = AArch64.AccessIsPrivileged(acctype);

    for i = 0 to UInt(ID_AA64DFR0_EL1.WRPs)
        match = match || AArch64.WatchpointMatch(i, vaddress, size, ispriv, acctype, iswrite);

    if match && HaltOnBreakpointOrWatchpoint() then
        if acctype != AccType_NONFAULT && acctype != AccType_CNOTFIRST then
            reason = DebugHalt_Watchpoint;
            Halt(reason);
        else
            // Fault will be reported and cancelled
            return AArch64.DebugFault(acctype, iswrite);
    elsif match then
        return AArch64.DebugFault(acctype, iswrite);
    else
        return AArch64.NoFault();
```

aarch64/translation/faults/AArch64.AccessFlagFault

```
// AArch64.AccessFlagFault()
// ========================

FaultRecord AArch64.AccessFlagFault(bits(52) ipaddress,boolean NS, integer level,
                                    AccType acctype, boolean iswrite, boolean secondstage,
                                    boolean s2fs1walk)

    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_AccessFlag, ipaddress, NS, level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);
```

aarch64/translation/faults/AArch64.AddressSizeFault

```
// AArch64.AddressSizeFault()
// ==========================

FaultRecord AArch64.AddressSizeFault(bits(52) ipaddress,boolean NS, integer level,
                                     AccType acctype, boolean iswrite, boolean secondstage,
                                     boolean s2fs1walk)

    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_AddressSize, ipaddress, NS, level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);
```

aarch64/translation/faults/AArch64.AlignmentFault

```
// AArch64.AlignmentFault()
// ========================

FaultRecord AArch64.AlignmentFault(AccType acctype, boolean iswrite, boolean secondstage)

    ipaddress = bits(52) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    s2fs1walk = boolean UNKNOWN;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    return AArch64.CreateFaultRecord(Fault_Alignment, ipaddress, boolean UNKNOWN, level, acctype,
iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);
```

aarch64/translation/faults/AArch64.AsynchExternalAbort

```
// AArch64.AsynchExternalAbort()
// ============================
// Wrapper function for asynchronous external aborts

FaultRecord AArch64.AsynchExternalAbort(boolean parity, bits(2) errortype, bit extflag)

    faulttype = if parity then Fault_AsyncParity else Fault_AsyncExternal;
    ipaddress = bits(52) UNKNOWN;
    level = integer UNKNOWN;
    acctype = AccType_NORMAL;
    iswrite = boolean UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch64.CreateFaultRecord(faulttype, ipaddress, boolean UNKNOWN, level, acctype, iswrite,
extflag,
                                      errortype, secondstage, s2fs1walk);
```

aarch64/translation/faults/AArch64.DebugFault

```
// AArch64.DebugFault()
// ====================

FaultRecord AArch64.DebugFault(AccType acctype, boolean iswrite)

    ipaddress = bits(52) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch64.CreateFaultRecord(Fault_Debug, ipaddress, boolean UNKNOWN, level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);
```

aarch64/translation/faults/AArch64.NoFault

```
// AArch64.NoFault()
// =================

FaultRecord AArch64.NoFault()

    ipaddress = bits(52) UNKNOWN;
    level = integer UNKNOWN;
    acctype = AccType_NORMAL;
    iswrite = boolean UNKNOWN;
    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch64.CreateFaultRecord(Fault_None, ipaddress, boolean UNKNOWN, level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);
```

aarch64/translation/faults/AArch64.PermissionFault

```
// AArch64.PermissionFault()
// =========================
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
FaultRecord AArch64.PermissionFault(bits(52) ipaddress,boolean NS, integer level,
                                    AccType acctype, boolean iswrite, boolean secondstage,
                                    boolean s2fs1walk)

    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_Permission, ipaddress, NS, level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);
```

aarch64/translation/faults/AArch64.TranslationFault

```
// AArch64.TranslationFault()
// ==========================

FaultRecord AArch64.TranslationFault(bits(52) ipaddress, boolean NS, integer level,
                                     AccType acctype, boolean iswrite, boolean secondstage,
                                     boolean s2fs1walk)

    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_Translation, ipaddress, NS, level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);
```

aarch64/translation/translation/AArch64.CheckAndUpdateDescriptor

```
// AArch64.CheckAndUpdateDescriptor()
// ==================================
// Check and update translation table descriptor if hardware update is configured

FaultRecord AArch64.CheckAndUpdateDescriptor(DescriptorUpdate result, FaultRecord fault,
                                             boolean secondstage, bits(64) vaddress, AccType acctype,
                                             boolean iswrite, boolean s2fs1walk, boolean hwupdatewalk)

    boolean hw_update_AF = FALSE;
    boolean hw_update_AP = FALSE;

    // Check if access flag can be updated
    // Address translation instructions are permitted to update AF but not required
    if result.AF then
        if fault.statuscode == Fault_None || ConstrainUnpredictable() == Constraint_TRUE then
            hw_update_AF = TRUE;

    if result.AP && fault.statuscode == Fault_None then
        write_perm_req = (iswrite || acctype IN {AccType_ATOMICRW,AccType_ORDEREDRW,
AccType_ORDEREDATOMICRW }) && !s2fs1walk;
        hw_update_AP = (write_perm_req && !(acctype IN {AccType_AT, AccType_DC, AccType_DC_UNPRIV})) ||
hwupdatewalk;

    if hw_update_AF || hw_update_AP then
        if secondstage || !HasS2Translation() then
            descaddr2 = result.descaddr;
        else
            hwupdatewalk = TRUE;
            descaddr2 = AArch64.SecondStageWalk(result.descaddr, vaddress, acctype, iswrite, 8,
hwupdatewalk);
            if IsFault(descaddr2) then
                return descaddr2.fault;

        accdesc = CreateAccessDescriptor(AccType_ATOMICRW);
        desc = _Mem[descaddr2, 8, accdesc];
        el = AArch64.AccessUsesEL(acctype);
        case el of
            when EL3
                reversedescriptors = SCTLR_EL3.EE == '1';
            when EL2
                reversedescriptors = SCTLR_EL2.EE == '1';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            otherwise
                reversedescriptors = SCTLR_EL1.EE == '1';
        if reversedescriptors then
            desc = BigEndianReverse(desc);

        if hw_update_AF then
            desc<10> = '1';
        if hw_update_AP then
            desc<7> = (if secondstage then '1' else '0');

        _Mem[descaddr2,8,accdesc] = if reversedescriptors then BigEndianReverse(desc) else desc;

    return fault;
```

aarch64/translation/translation/AArch64.FirstStageTranslate

```
// AArch64.FirstStageTranslate()
// ============================
// Perform a stage 1 translation walk. The function used by Address Translation operations is
// similar except it uses the translation regime specified for the instruction.

AddressDescriptor AArch64.FirstStageTranslate(bits(64) vaddress, AccType acctype, boolean iswrite,
                                              boolean wasaligned, integer size)

    if HaveNV2Ext() && acctype == AccType_NV2REGISTER then
        s1_enabled = SCTLR_EL2.M == '1';
    elsif HasS2Translation() then
        s1_enabled = HCR_EL2.TGE == '0' && HCR_EL2.DC == '0' && SCTLR_EL1.M == '1';
    else
        s1_enabled = SCTLR[].M == '1';

    ipaddress = bits(52) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    if s1_enabled then                          // First stage enabled
        S1 = AArch64.TranslationTableWalk(ipaddress, TRUE, vaddress, acctype, iswrite, secondstage,
                                          s2fs1walk, size);
        permissioncheck = TRUE;
        if acctype == AccType_IFETCH then
            InGuardedPage = S1.GP == '1';       // Global state updated on instruction fetch that denotes
                                                // if the fetched instruction is from a guarded page.
    else
        S1 = AArch64.TranslateAddressS1Off(vaddress, acctype, iswrite);
        permissioncheck = FALSE;

    if !IsFault(S1.addrdesc) && UsingAArch32() && HaveTrapLoadStoreMultipleDeviceExt() &&
AArch32.ExecutingLSMInstr() then
        if S1.addrdesc.memattrs.memtype == MemType_Device && S1.addrdesc.memattrs.device !=
DeviceType_GRE then
            nTLSMD = if S1TranslationRegime() == EL2 then SCTLR_EL2.nTLSMD else SCTLR_EL1.nTLSMD;
            if nTLSMD == '0' then
                S1.addrdesc.fault = AArch64.AlignmentFault(acctype, iswrite, secondstage);

    // Check for unaligned data accesses to Device memory
    if ((!wasaligned && acctype != AccType_IFETCH) || (acctype == AccType_DCZVA))
        && !IsFault(S1.addrdesc) && S1.addrdesc.memattrs.memtype == MemType_Device then
        S1.addrdesc.fault = AArch64.AlignmentFault(acctype, iswrite, secondstage);
    if !IsFault(S1.addrdesc) && permissioncheck then
        S1.addrdesc.fault = AArch64.CheckPermission(S1.perms, vaddress, S1.level,
                                                    S1.addrdesc.paddress.NS,
                                                    acctype, iswrite);

    // Check for instruction fetches from Device memory not marked as execute-never. If there has
    // not been a Permission Fault then the memory is not marked execute-never.
    if (!IsFault(S1.addrdesc) && S1.addrdesc.memattrs.memtype == MemType_Device &&
        acctype == AccType_IFETCH) then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        S1.addrdesc = AArch64.InstructionDevice(S1.addrdesc, vaddress, ipaddress, S1.level,
                                                acctype, iswrite,
                                                secondstage, s2fs1walk);
    // Check and update translation table descriptor if required
    hwupdatewalk = FALSE;
    s2fs1walk = FALSE;
    S1.addrdesc.fault = AArch64.CheckAndUpdateDescriptor(S1.descupdate, S1.addrdesc.fault,
                                                         secondstage, vaddress, acctype,
                                                         iswrite, s2fs1walk, hwupdatewalk);

    return S1.addrdesc;
```

aarch64/translation/translation/AArch64.FullTranslate

```
// AArch64.FullTranslate()
// =======================
// Perform both stage 1 and stage 2 translation walks for the current translation regime. The
// function used by Address Translation operations is similar except it uses the translation
// regime specified for the instruction.

AddressDescriptor AArch64.FullTranslate(bits(64) vaddress, AccType acctype, boolean iswrite,
                                        boolean wasaligned, integer size)

    // First Stage Translation
    S1 = AArch64.FirstStageTranslate(vaddress, acctype, iswrite, wasaligned, size);
    if !IsFault(S1) && !(HaveNV2Ext() && acctype == AccType_NV2REGISTER) && HasS2Translation() then
        s2fs1walk = FALSE;
        hwupdatewalk = FALSE;
        result = AArch64.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk,
                                              size, hwupdatewalk);
    else
        result = S1;

    return result;
```

aarch64/translation/translation/AArch64.SecondStageTranslate

```
// AArch64.SecondStageTranslate()
// ==============================
// Perform a stage 2 translation walk. The function used by Address Translation operations is
// similar except it uses the translation regime specified for the instruction.

AddressDescriptor AArch64.SecondStageTranslate(AddressDescriptor S1, bits(64) vaddress,
                                               AccType acctype, boolean iswrite, boolean wasaligned,
                                               boolean s2fs1walk, integer size, boolean hwupdatewalk)
    assert HasS2Translation();

    s2_enabled = HCR_EL2.VM == '1' || HCR_EL2.DC == '1';
    secondstage = TRUE;

    if s2_enabled then                        // Second stage enabled
        ipaddress = S1.paddress.address<51:0>;
        NS = S1.paddress.NS == '1';
        S2 = AArch64.TranslationTableWalk(ipaddress, NS, vaddress, acctype, iswrite, secondstage,
                                          s2fs1walk, size);

        // Check for unaligned data accesses to Device memory
        if ((!wasaligned && acctype != AccType_IFETCH) || (acctype == AccType_DCZVA))
            && S2.addrdesc.memattrs.memtype == MemType_Device && !IsFault(S2.addrdesc) then
            S2.addrdesc.fault = AArch64.AlignmentFault(acctype, iswrite, secondstage);

        // Check for permissions on Stage2 translations
        if !IsFault(S2.addrdesc) then
            S2.addrdesc.fault = AArch64.CheckS2Permission(S2.perms, vaddress, ipaddress, S2.level,
                                                          acctype, iswrite, NS,s2fs1walk, hwupdatewalk);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        // Check for instruction fetches from Device memory not marked as execute-never. As there
        // has not been a Permission Fault then the memory is not marked execute-never.
        if (!s2fs1walk && !IsFault(S2.addrdesc) && S2.addrdesc.memattrs.memtype == MemType_Device &&
            acctype == AccType_IFETCH) then
            S2.addrdesc = AArch64.InstructionDevice(S2.addrdesc, vaddress, ipaddress, S2.level,
                                                    acctype, iswrite,
                                                    secondstage, s2fs1walk);

        if (s2fs1walk && !IsFault(S2.addrdesc) &&
            S2.addrdesc.memattrs.memtype == MemType_Device) then
            // Check for protected table walk.
            if HCR_EL2.PTW == '1' then
                S2.addrdesc.fault = AArch64.PermissionFault(ipaddress,
                                    NS,  S2.level,
                                    acctype, iswrite, secondstage, s2fs1walk);
            else
                // Translation table walk occurs as Normal Non-cacheable memory.
                S2.addrdesc.memattrs.memtype        = MemType_Normal;
                S2.addrdesc.memattrs.inner.attrs    = MemAttr_NC;
                S2.addrdesc.memattrs.outer.attrs    = MemAttr_NC;
                S2.addrdesc.memattrs.shareable      = TRUE;
                S2.addrdesc.memattrs.outershareable = TRUE;

        // Check and update translation table descriptor if required
        S2.addrdesc.fault = AArch64.CheckAndUpdateDescriptor(S2.descupdate, S2.addrdesc.fault,
                                                             secondstage, vaddress, acctype,
                                                             iswrite, s2fs1walk, hwupdatewalk);
        result = AArch64.CombineS1S2Desc(S1, S2.addrdesc);
    else
        result = S1;

    return result;
```

aarch64/translation/translation/AArch64.SecondStageWalk

```
// AArch64.SecondStageWalk()
// =========================
// Perform a stage 2 translation on a stage 1 translation page table walk access.

AddressDescriptor AArch64.SecondStageWalk(AddressDescriptor S1, bits(64) vaddress, AccType acctype,
                                          boolean iswrite, integer size, boolean hwupdatewalk)

    assert HasS2Translation();

    s2fs1walk = TRUE;
    wasaligned = TRUE;
    return AArch64.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk,
                                        size, hwupdatewalk);
```

aarch64/translation/translation/AArch64.TranslateAddress

```
// AArch64.TranslateAddress()
// ==========================
// Main entry point for translating an address

AddressDescriptor AArch64.TranslateAddress(bits(64) vaddress, AccType acctype, boolean iswrite,
                                           boolean wasaligned, integer size)

    result = AArch64.FullTranslate(vaddress, acctype, iswrite, wasaligned, size);

    if !(acctype IN {AccType_PTW, AccType_IC, AccType_AT}) && !IsFault(result) then
        result.fault = AArch64.CheckDebug(vaddress, acctype, iswrite, size);

    // Update virtual address for abort functions
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    result.vaddress = ZeroExtend(vaddress);

    return result;
```

aarch64/translation/walk/AArch64.TranslationTableWalk

```
// AArch64.TranslationTableWalk()
// ==============================
// Returns a result of a translation table walk
//
// Implementations might cache information from memory in any number of non-coherent TLB
// caching structures, and so avoid memory accesses that have been expressed in this
// pseudocode. The use of such TLBs is not expressed in this pseudocode.

TLBRecord AArch64.TranslationTableWalk(bits(52) ipaddress, boolean s1_nonsecure, bits(64) vaddress,
                                       AccType acctype, boolean iswrite, boolean secondstage,
                                       boolean s2fs1walk, integer size)
    if !secondstage then
        assert !ELUsingAArch32(S1TranslationRegime());
    else
        assert (IsSecureEL2Enabled() || (HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2))) &&
HasS2Translation();

    TLBRecord result;
    AddressDescriptor descaddr;
    bits(64) baseregister;
    bits(64) inputaddr;        // Input Address is 'vaddress' for stage 1, 'ipaddress' for stage 2
    bit nswalk;                    // Stage 2 translation table walks are to Secure or to Non-secure PA
space

    descaddr.memattrs.memtype = MemType_Normal;

    // Derived parameters for the page table walk:
    //  grainsize = Log2(Size of Table)         - Size of Table is 4KB, 16KB or 64KB in AArch64
    //  stride = Log2(Address per Level)        - Bits of address consumed at each level
    //  firstblocklevel = First level where a block entry is allowed
    //  ps = Physical Address size as encoded in TCR_EL1.IPS or TCR_ELx/VTCR_EL2.PS
    //  inputsize = Log2(Size of Input Address) - Input Address size in bits
    //  level = Level to start walk from
    // This means that the number of levels after start level = 3-level

    if !secondstage then
        // First stage translation
        inputaddr = ZeroExtend(vaddress);
        el = AArch64.AccessUsesEL(acctype);
        top = AddrTop(inputaddr, (acctype == AccType_IFETCH), el);
        if el == EL3 then
            largegrain = TCR_EL3.TG0 == '01';
            midgrain = TCR_EL3.TG0 == '10';
            inputsize = 64 - UInt(TCR_EL3.T0SZ);
            inputsize_max = if Have52BitVAExt() && largegrain then 52 else 48;
            inputsize_min = 64 - (if !HaveSmallPageTblExt() then 39 else if largegrain then 47 else 48);
            if inputsize < inputsize_min then
                c = ConstrainUnpredictable();
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then inputsize = inputsize_min;
            ps = TCR_EL3.PS;
            basefound = inputsize >= inputsize_min && inputsize <= inputsize_max &&
IsZero(inputaddr<top:inputsize>);
            disabled = FALSE;
            baseregister = TTBR0_EL3;
            descaddr.memattrs = WalkAttrDecode(TCR_EL3.SH0, TCR_EL3.ORGN0, TCR_EL3.IRGN0, secondstage);
            reversedescriptors = SCTLR_EL3.EE == '1';
            lookupsecure = TRUE;
            singlepriv = TRUE;
            update_AF = HaveAccessFlagUpdateExt() && TCR_EL3.HA == '1';
            update_AP = HaveDirtyBitModifierExt() && update_AF && TCR_EL3.HD == '1';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            hierattrsdisabled = AArch64.HaveHPDExt() && TCR_EL3.HPD == '1';
        elsif ELIsInHost(el) then
            if inputaddr<top> == '0' then
                largegrain = TCR_EL2.TG0 == '01';
                midgrain = TCR_EL2.TG0 == '10';
                inputsize = 64 - UInt(TCR_EL2.T0SZ);
                inputsize_max = if Have52BitVAExt() && largegrain then 52 else 48;
                inputsize_min = 64 - (if !HaveSmallPageTblExt() then 39 else if largegrain then 47 else
48);
                if inputsize < inputsize_min then
                    c = ConstrainUnpredictable();
                    assert c IN {Constraint_FORCE, Constraint_FAULT};
                    if c == Constraint_FORCE then inputsize = inputsize_min;
                basefound = inputsize >= inputsize_min && inputsize <= inputsize_max &&
IsZero(inputaddr<top:inputsize>);
                disabled = TCR_EL2.EPD0 == '1' || (PSTATE.EL == EL0 && HaveE0PDExt() && TCR_EL2.E0PD0 ==
'1');
                baseregister = TTBR0_EL2;
                descaddr.memattrs = WalkAttrDecode(TCR_EL2.SH0, TCR_EL2.ORGN0, TCR_EL2.IRGN0,
secondstage);
                hierattrsdisabled = AArch64.HaveHPDExt() && TCR_EL2.HPD0 == '1';
            else
                inputsize = 64 - UInt(TCR_EL2.T1SZ);
                largegrain = TCR_EL2.TG1 == '11';       // TG1 and TG0 encodings differ
                midgrain = TCR_EL2.TG1 == '01';
                inputsize_max = if Have52BitVAExt() && largegrain then 52 else 48;
                inputsize_min = 64 - (if !HaveSmallPageTblExt() then 39 else if largegrain then 47 else
48);
                if inputsize < inputsize_min then
                    c = ConstrainUnpredictable();
                    assert c IN {Constraint_FORCE, Constraint_FAULT};
                    if c == Constraint_FORCE then inputsize = inputsize_min;
                basefound = inputsize >= inputsize_min && inputsize <= inputsize_max &&
IsOnes(inputaddr<top:inputsize>);
                disabled = TCR_EL2.EPD1 == '1' || (PSTATE.EL == EL0 && HaveE0PDExt() && TCR_EL2.E0PD1 ==
'1');
                baseregister = TTBR1_EL2;
                descaddr.memattrs = WalkAttrDecode(TCR_EL2.SH1, TCR_EL2.ORGN1, TCR_EL2.IRGN1,
secondstage);
                hierattrsdisabled = AArch64.HaveHPDExt() && TCR_EL2.HPD1 == '1';
            ps = TCR_EL2.IPS;
            reversedescriptors = SCTLR_EL2.EE == '1';
            lookupsecure = if IsSecureEL2Enabled() then IsSecure() else FALSE;
            singlepriv = FALSE;
            update_AF = HaveAccessFlagUpdateExt() && TCR_EL2.HA == '1';
            update_AP = HaveDirtyBitModifierExt() && update_AF && TCR_EL2.HD == '1';
        elsif el == EL2 then
            inputsize = 64 - UInt(TCR_EL2.T0SZ);
            largegrain = TCR_EL2.TG0 == '01';
            midgrain = TCR_EL2.TG0 == '10';
            inputsize_max = if Have52BitVAExt() && largegrain then 52 else 48;
            inputsize_min = 64 - (if !HaveSmallPageTblExt() then 39 else if largegrain then 47 else 48);
            if inputsize < inputsize_min then
                c = ConstrainUnpredictable();
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then inputsize = inputsize_min;
            ps = TCR_EL2.PS;
            basefound = inputsize >= inputsize_min && inputsize <= inputsize_max &&
IsZero(inputaddr<top:inputsize>);
            disabled = FALSE;
            baseregister = TTBR0_EL2;
            descaddr.memattrs = WalkAttrDecode(TCR_EL2.SH0, TCR_EL2.ORGN0, TCR_EL2.IRGN0, secondstage);
            reversedescriptors = SCTLR_EL2.EE == '1';
            lookupsecure = if IsSecureEL2Enabled() then IsSecure() else FALSE;
            singlepriv = TRUE;
            update_AF = HaveAccessFlagUpdateExt() && TCR_EL2.HA == '1';
            update_AP = HaveDirtyBitModifierExt() && update_AF && TCR_EL2.HD == '1';
            hierattrsdisabled = AArch64.HaveHPDExt() && TCR_EL2.HPD == '1';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        else
            if inputaddr<top> == '0' then
                inputsize = 64 - UInt(TCR_EL1.T0SZ);
                largegrain = TCR_EL1.TG0 == '01';
                midgrain = TCR_EL1.TG0 == '10';
                inputsize_max = if Have52BitVAExt() && largegrain then 52 else 48;
                inputsize_min = 64 - (if !HaveSmallPageTblExt() then 39 else if largegrain then 47 else
48);
                if inputsize < inputsize_min then
                    c = ConstrainUnpredictable();
                    assert c IN {Constraint_FORCE, Constraint_FAULT};
                    if c == Constraint_FORCE then inputsize = inputsize_min;
                basefound = inputsize >= inputsize_min && inputsize <= inputsize_max &&
IsZero(inputaddr<top:inputsize>);
                disabled = TCR_EL1.EPD0 == '1' || (PSTATE.EL == EL0 && HaveE0PDExt() && TCR_EL1.E0PD0 ==
'1');
                disabled = disabled || (el == EL0 && acctype == AccType_NONFAULT && TCR_EL1.NFD0 ==
'1');
                baseregister = TTBR0_EL1;
                descaddr.memattrs = WalkAttrDecode(TCR_EL1.SH0, TCR_EL1.ORGN0, TCR_EL1.IRGN0,
secondstage);
                hierattrsdisabled = AArch64.HaveHPDExt() &&  TCR_EL1.HPD0 == '1';
            else
                inputsize = 64 - UInt(TCR_EL1.T1SZ);
                largegrain = TCR_EL1.TG1 == '11';       // TG1 and TG0 encodings differ
                midgrain = TCR_EL1.TG1 == '01';
                inputsize_max = if Have52BitVAExt() && largegrain then 52 else 48;
                inputsize_min = 64 - (if !HaveSmallPageTblExt() then 39 else if largegrain then 47 else
48);
                if inputsize < inputsize_min then
                    c = ConstrainUnpredictable();
                    assert c IN {Constraint_FORCE, Constraint_FAULT};
                    if c == Constraint_FORCE then inputsize = inputsize_min;
                basefound = inputsize >= inputsize_min && inputsize <= inputsize_max &&
IsOnes(inputaddr<top:inputsize>);
                disabled = TCR_EL1.EPD1 == '1' || (PSTATE.EL == EL0 && HaveE0PDExt() && TCR_EL1.E0PD1 ==
'1');
                disabled = disabled || (el == EL0 && acctype == AccType_NONFAULT && TCR_EL1.NFD1 ==
'1');
                baseregister = TTBR1_EL1;
                descaddr.memattrs = WalkAttrDecode(TCR_EL1.SH1, TCR_EL1.ORGN1, TCR_EL1.IRGN1,
secondstage);
                hierattrsdisabled = AArch64.HaveHPDExt() &&  TCR_EL1.HPD1 == '1';
            ps = TCR_EL1.IPS;
            reversedescriptors = SCTLR_EL1.EE == '1';
            lookupsecure = IsSecure();
            singlepriv = FALSE;
            update_AF = HaveAccessFlagUpdateExt() && TCR_EL1.HA == '1';
            update_AP = HaveDirtyBitModifierExt() && update_AF && TCR_EL1.HD == '1';
        if largegrain then
            grainsize = 16;                                         // Log2(64KB page size)
            firstblocklevel = (if Have52BitPAExt() then 1 else 2);      // Largest block is 4TB (2^42
bytes) for 52 bit PA
                                                                    // and 512MB (2^29 bytes)
otherwise
        elsif midgrain then
            grainsize = 14;                                         // Log2(16KB page size)
            firstblocklevel = 2;                                    // Largest block is 32MB (2^25
bytes)
        else // Small grain
            grainsize = 12;                                         // Log2(4KB page size)
            firstblocklevel = 1;                                    // Largest block is 1GB (2^30
bytes)
        stride = grainsize - 3;                                     // Log2(page size / 8 bytes)
        // The starting level is the number of strides needed to consume the input address
        level = 4 - (1 + ((inputsize - grainsize - 1) DIV stride));

    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        // Second stage translation
        inputaddr = ZeroExtend(ipaddress);
        if IsSecureBelowEL3() then
            // Second stage for Secure translation regime
            if s1_nonsecure then            // Non-secure IPA space
                t0size = VTCR_EL2.T0SZ;
                tg0    = VTCR_EL2.TG0;
                nswalk = VTCR_EL2.NSW;
            else                            // Secure IPA space
                t0size = VSTCR_EL2.T0SZ;
                tg0    = VSTCR_EL2.TG0;
                nswalk = VSTCR_EL2.SW;

            // Stage 2 translation accesses the Non-secure PA space or the Secure PA space
            if nswalk == '1' then
                // When walk is Non-secure, access must be to the Non-secure PA space
                nsaccess = '1';
            elsif !s1_nonsecure then
                // When walk is Secure and in the Secure IPA space,
                //  access is specified by VSTCR_EL2.SA
                nsaccess = VSTCR_EL2.SA;
            elsif VSTCR_EL2.SW == '1' || VSTCR_EL2.SA == '1' then
                // When walk is Secure and in the Non-secure IPA space,
                //  access is Non-secure when VSTCR_EL2.SA specifies the Non-secure PA space
                nsaccess = '1';
            else
                // When walk is Secure and in the Non-secure IPA space,
                //  if VSTCR_EL2.SA specifies the Secure PA space, access is specified by VTCR_EL2.NSA
                nsaccess = VTCR_EL2.NSA;
        else
            // Second stage for Non-secure translation regime
            t0size       = VTCR_EL2.T0SZ;
            tg0          = VTCR_EL2.TG0;
            nswalk       = '1';
            nsaccess     = '1';

        inputsize  = 64 - UInt(t0size);
        largegrain = tg0 == '01';
        midgrain   = tg0 == '10';

        inputsize_max = if Have52BitPAExt() && PAMax() == 52 && largegrain then 52 else 48;
        inputsize_min = 64 - (if !HaveSmallPageTblExt() then 39 else if largegrain then 47 else 48);
        if inputsize < inputsize_min then
            c = ConstrainUnpredictable();
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then inputsize = inputsize_min;
        ps = VTCR_EL2.PS;
        basefound = inputsize >= inputsize_min && inputsize <= inputsize_max &&
IsZero(inputaddr<63:inputsize>);
        disabled = FALSE;
        descaddr.memattrs = WalkAttrDecode(VTCR_EL2.SH0, VTCR_EL2.ORGN0, VTCR_EL2.IRGN0, secondstage);
        reversedescriptors = SCTLR_EL2.EE == '1';
        singlepriv = TRUE;
        update_AF = HaveAccessFlagUpdateExt() && VTCR_EL2.HA == '1';
        update_AP = HaveDirtyBitModifierExt() && update_AF && VTCR_EL2.HD == '1';

        if IsSecureEL2Enabled() then
            lookupsecure = !s1_nonsecure;
        else
            lookupsecure = FALSE;

        if lookupsecure then
            baseregister = VSTTBR_EL2;
            startlevel   = UInt(VSTCR_EL2.SL0);
        else
            baseregister = VTTBR_EL2;
            startlevel   = UInt(VTCR_EL2.SL0);
        if largegrain then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            grainsize = 16;                                         // Log2(64KB page size)
            level = 3 - startlevel;
            firstblocklevel = (if Have52BitPAExt() then 1 else 2);  // Largest block is 4TB (2^42 bytes)
for 52 bit PA
                                                                    // and 512MB (2^29 bytes) otherwise
        elsif midgrain then
            grainsize = 14;                                         // Log2(16KB page size)
            level = 3 - startlevel;
            firstblocklevel = 2;                                    // Largest block is 32MB (2^25
bytes)
        else // Small grain
            grainsize = 12;                                         // Log2(4KB page size)
            if HaveSmallPageTblExt() && startlevel == 3 then
                level = startlevel;                             // Startlevel 3 (VTCR_EL2.SL0 or
VSCTR_EL2.SL0 == 0b11) for 4KB granule
            else
                level = 2 - startlevel;
            firstblocklevel = 1;                                    // Largest block is 1GB (2^30 bytes)
        stride = grainsize - 3;                                     // Log2(page size / 8 bytes)

        // Limits on IPA controls based on implemented PA size. Level 0 is only
        // supported by small grain translations
        if largegrain then                              // 64KB pages
            // Level 1 only supported if implemented PA size is greater than 2^42 bytes
            if level == 0 || (level == 1 && PAMax() <= 42) then basefound = FALSE;
        elsif midgrain then                             // 16KB pages
            // Level 1 only supported if implemented PA size is greater than 2^40 bytes
            if level == 0 || (level == 1 && PAMax() <= 40) then basefound = FALSE;
        else                                            // Small grain, 4KB pages
            // Level 0 only supported if implemented PA size is greater than 2^42 bytes
            if level < 0 || (level == 0 && PAMax() <= 42) then basefound = FALSE;

        // If the inputsize exceeds the PAMax value, the behavior is CONSTRAINED UNPREDICTABLE
        inputsizecheck = inputsize;
        if inputsize > PAMax() && (!ELUsingAArch32(EL1) || inputsize > 40) then
            case ConstrainUnpredictable() of
                when Constraint_FORCE
                    // Restrict the inputsize to the PAMax value
                    inputsize = PAMax();
                    inputsizecheck = PAMax();
                when Constraint_FORCENOSLCHECK
                    // As FORCE, except use the configured inputsize in the size checks below
                    inputsize = PAMax();
                when Constraint_FAULT
                    // Generate a translation fault
                    basefound = FALSE;
                otherwise
                    Unreachable();

        // Number of entries in the starting level table =
        //     (Size of Input Address)/((Address per level)^(Num levels remaining)*(Size of Table))
        startsizecheck = inputsizecheck - ((3 - level)*stride + grainsize); // Log2(Num of entries)

        // Check for starting level table with fewer than 2 entries or longer than 16 pages.
        // Lower bound check is:  startsizecheck < Log2(2 entries)
        // Upper bound check is:  startsizecheck > Log2(pagesize/8*16)
        if startsizecheck < 1 || startsizecheck > stride + 4 then basefound = FALSE;
    if !basefound || disabled then
        level = 0;           // AArch32 reports this as a level 1 fault
        result.addrdesc.fault = AArch64.TranslationFault(ipaddress, s1_nonsecure,  level, acctype,
iswrite,
                                                         secondstage, s2fs1walk);
        return result;

    case ps of
        when '000'  outputsize = 32;
        when '001'  outputsize = 36;
        when '010'  outputsize = 40;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        when '011'  outputsize = 42;
        when '100'  outputsize = 44;
        when '101'  outputsize = 48;
        when '110'  outputsize = (if Have52BitPAExt() && largegrain then 52 else 48);
        otherwise   outputsize = integer IMPLEMENTATION_DEFINED "Reserved Intermediate Physical Address
size value";

    if outputsize > PAMax() then outputsize = PAMax();

    if outputsize < 48 && !IsZero(baseregister<47:outputsize>) then
        level = 0;
        result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress,s1_nonsecure,  level, acctype,
iswrite,
                                                         secondstage, s2fs1walk);
        return result;

    // Bottom bound of the Base address is:
    //     Log2(8 bytes per entry)+Log2(Number of entries in starting level table)
    // Number of entries in starting level table =
    //     (Size of Input Address)/((Address per level)^(Num levels remaining)*(Size of Table))
    baselowerbound = 3 + inputsize - ((3-level)*stride + grainsize);  // Log2(Num of entries*8)
    if outputsize == 52 then
        z = (if baselowerbound < 6 then 6 else baselowerbound);
        baseaddress = baseregister<5:2>:baseregister<47:z>:Zeros(z);
    else
        baseaddress = ZeroExtend(baseregister<47:baselowerbound>:Zeros(baselowerbound));

    ns_table = if lookupsecure then '0' else '1';
    ap_table = '00';
    xn_table = '0';
    pxn_table = '0';

    addrselecttop = inputsize - 1;

    apply_nvnv1_effect = HaveNVExt() && EL2Enabled() && HCR_EL2.<NV,NV1> == '11' &&
S1TranslationRegime() == EL1 && !secondstage;
    repeat
        addrselectbottom = (3-level)*stride + grainsize;

        bits(52) index = ZeroExtend(inputaddr<addrselecttop:addrselectbottom>:'000');
        descaddr.paddress.address = baseaddress OR index;
        descaddr.paddress.NS = if secondstage then nswalk else ns_table;

        // If there are two stages of translation, then the first stage table walk addresses
        // are themselves subject to translation
        if secondstage || !HasS2Translation() || (HaveNV2Ext() && acctype == AccType_NV2REGISTER) then
            descaddr2 = descaddr;
        else
            hwupdatewalk = FALSE;
            descaddr2 = AArch64.SecondStageWalk(descaddr, vaddress, acctype, iswrite, 8, hwupdatewalk);
            // Check for a fault on the stage 2 walk
            if IsFault(descaddr2) then
                result.addrdesc.fault = descaddr2.fault;
                return result;

        // Update virtual address for abort functions
        descaddr2.vaddress = ZeroExtend(vaddress);

        accdesc = CreateAccessDescriptorPTW(acctype, secondstage, s2fs1walk, level);
        desc = _Mem[descaddr2, 8, accdesc];

        if reversedescriptors then desc = BigEndianReverse(desc);

        if desc<0> == '0' || (desc<1:0> == '01' && (level == 3 ||
                                                    (HaveBlockBBM() && IsBlockDescriptorNTBitValid() &&
desc<16> == '1'))) then
             // Fault (00), Reserved (10), Block (01) at level 3, or Block(01) with nT bit set.
            result.addrdesc.fault = AArch64.TranslationFault(ipaddress, s1_nonsecure, level, acctype,
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
                                                                iswrite, secondstage, s2fs1walk);
                return result;

            // Valid Block, Page, or Table entry
            if desc<1:0> == '01' || level == 3 then                 // Block (01) or Page (11)
                blocktranslate = TRUE;
            else                                                    // Table (11)
                if (outputsize < 52 && largegrain && !IsZero(desc<15:12>)) || (outputsize < 48 &&
!IsZero(desc<47:outputsize>)) then
                    result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress,s1_nonsecure,  level,
acctype,
                                                                     iswrite, secondstage, s2fs1walk);
                    return result;

                if outputsize == 52 then
                    baseaddress = desc<15:12>:desc<47:grainsize>:Zeros(grainsize);
                else
                    baseaddress = ZeroExtend(desc<47:grainsize>:Zeros(grainsize));
                if !secondstage then
                    // Unpack the upper and lower table attributes
                    ns_table    = ns_table    OR desc<63>;
                if !secondstage && !hierattrsdisabled then
                    ap_table<1> = ap_table<1> OR desc<62>;       // read-only

                    if apply_nvnv1_effect then
                        pxn_table   = pxn_table   OR desc<60>;
                    else
                        xn_table    = xn_table    OR desc<60>;
                    // pxn_table and ap_table[0] apply in EL1&0 or EL2&0 translation regimes
                    if !singlepriv then
                        if !apply_nvnv1_effect then
                            pxn_table   = pxn_table   OR desc<59>;
                            ap_table<0> = ap_table<0> OR desc<61>;   // privileged

                level = level + 1;
                addrselecttop = addrselectbottom - 1;
                blocktranslate = FALSE;
        until blocktranslate;

        // Check block size is supported at this level
        if level < firstblocklevel then
            result.addrdesc.fault = AArch64.TranslationFault(ipaddress, s1_nonsecure, level, acctype,
                                                             iswrite, secondstage, s2fs1walk);
            return result;

        // Check for misprogramming of the contiguous bit
        if largegrain then
            num_ch_entries = 5;
        elsif midgrain then
            if level == 3 then
                num_ch_entries = 7;
            else num_ch_entries = 5;
        else num_ch_entries = 4;

        contiguousbitcheck = inputsize < (addrselectbottom + num_ch_entries);

        if contiguousbitcheck && desc<52> == '1' then
            if boolean IMPLEMENTATION_DEFINED "Translation fault on misprogrammed contiguous bit" then
                result.addrdesc.fault = AArch64.TranslationFault(ipaddress, s1_nonsecure, level, acctype,
                                                                 iswrite, secondstage, s2fs1walk);
                return result;

        // Unpack the descriptor into address and upper and lower block attributes
        if largegrain then
            outputaddress = desc<15:12>:desc<47:addrselectbottom>:inputaddr<addrselectbottom-1:0>;
        else
            outputaddress = ZeroExtend(desc<47:addrselectbottom>:inputaddr<addrselectbottom-1:0>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    // When 52-bit PA is supported, for 64 Kbyte translation granule,
    // block size might be larger than the supported output address size
    if outputsize < 52 && !IsZero(outputaddress<51:outputsize>) then
        result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress,s1_nonsecure,  level, acctype,
                                                         iswrite, secondstage, s2fs1walk);
        return result;

    // Check Access Flag
    if desc<10> == '0' then
        if !update_AF then
            result.addrdesc.fault = AArch64.AccessFlagFault(ipaddress,s1_nonsecure,  level, acctype,
                                                            iswrite, secondstage, s2fs1walk);
            return result;
        else
            result.descupdate.AF = TRUE;

    if update_AP && desc<51> == '1' then
        // If hw update of access permission field is configured consider AP[2] as '0' / S2AP[2] as '1'
        if !secondstage && desc<7> == '1' then
            desc<7> = '0';
            result.descupdate.AP = TRUE;
        elsif secondstage && desc<7> == '0' then
            desc<7> = '1';
            result.descupdate.AP = TRUE;

    // Required descriptor if AF or AP[2]/S2AP[2] needs update
    result.descupdate.descaddr = descaddr;

    if apply_nvnv1_effect then
        pxn = desc<54>;                                     // Bit[54] of the block/page descriptor
holds PXN instead of UXN
        xn = '0';                                           // XN is '0'
        ap = desc<7>:'01';                                  // Bit[6] of the block/page descriptor is
treated as '0' regardless of value programmed
    else
        xn = desc<54>;                                      // Bit[54] of the block/page descriptor
holds UXN
        pxn = desc<53>;                                     // Bit[53] of the block/page descriptor
holds PXN
        ap = desc<7:6>:'1';                                 // Bits[7:6] of the block/page descriptor
hold AP[2:1]
    contiguousbit = desc<52>;
    nG = desc<11>;
    sh = desc<9:8>;
    memattr = desc<5:2>;                                    // AttrIndx and NS bit in stage 1

    result.domain = bits(4) UNKNOWN;                        // Domains not used
    result.level = level;
    result.blocksize = 2^((3-level)*stride + grainsize);

    // Stage 1 translation regimes also inherit attributes from the tables
    if !secondstage then
        result.perms.xn      = xn OR xn_table;
        result.perms.ap<2>   = ap<2> OR ap_table<1>;        // Force read-only
        // PXN, nG and AP[1] apply in EL1&0 or EL2&0 stage 1 translation regimes
        if !singlepriv then
            result.perms.ap<1> = ap<1> AND NOT(ap_table<0>);  // Force privileged only
            result.perms.pxn   = pxn OR pxn_table;
            // Pages from Non-secure tables are marked non-global in Secure EL1&0
            if IsSecure() then
                result.nG = nG OR ns_table;
            else
                result.nG = nG;
        else
            result.perms.ap<1> = '1';
            result.perms.pxn   = '0';
            result.nG          = '0';
        result.GP = desc<50>;                               // Stage 1 block or pages might be guarded
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        result.perms.ap<0>   = '1';
        result.addrdesc.memattrs = AArch64.S1AttrDecode(sh, memattr<2:0>, acctype);
        result.addrdesc.paddress.NS = memattr<3> OR ns_table;
    else
        result.perms.ap<2:1> = ap<2:1>;
        result.perms.ap<0>   = '1';
        result.perms.xn      = xn;
        if HaveExtendedExecuteNeverExt() then result.perms.xxn = desc<53>;
        result.perms.pxn     = '0';
        result.nG            = '0';
        if s2fs1walk then
            result.addrdesc.memattrs = S2AttrDecode(sh, memattr, AccType_PTW);
        else
            result.addrdesc.memattrs = S2AttrDecode(sh, memattr, acctype);
        result.addrdesc.paddress.NS = nsaccess;

    result.addrdesc.paddress.address = outputaddress;
    result.addrdesc.fault = AArch64.NoFault();
    result.contiguous = contiguousbit == '1';
    if HaveCommonNotPrivateTransExt() then result.CnP = baseregister<0>;

    return result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

J1.2 Pseudocode for AArch32 operation

This section holds the pseudocode for execution in AArch32 state. Functions that are listed in this section are identified as `AArch32.FunctionName`. Some of these functions have an equivalent AArch64 function, `AArch64.FunctionName`. This section is organized by functional groups, with the functional groups being indicated by hierarchical path names, for example `aarch32/debug/breakpoint`.

Note

Many AArch32 pseudocode functions have not been updated to show the constraints on the Armv7 UNPREDICTABLE behaviors that are described in Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*. Where AArch32 pseudocode shows something to be UNPREDICTABLE, check Appendix K1 for possible constraints on the permitted behavior.

The top-level sections of the AArch32 pseudocode hierarchy are:

- *aarch32/debug*.
- *aarch32/exceptions* on page J1-7635.
- *aarch32/functions* on page J1-7655.
- *aarch32/translation* on page J1-7683.

J1.2.1 aarch32/debug

This section includes the following pseudocode functions:

- *aarch32/debug/VCRMatch/AArch32.VCRMatch*.
- *aarch32/debug/authentication/AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled* on page J1-7628.
- *aarch32/debug/breakpoint/AArch32.BreakpointMatch* on page J1-7628.
- *aarch32/debug/breakpoint/AArch32.BreakpointValueMatch* on page J1-7629.
- *aarch32/debug/breakpoint/AArch32.StateMatch* on page J1-7630.
- *aarch32/debug/enables/AArch32.GenerateDebugExceptions* on page J1-7631.
- *aarch32/debug/enables/AArch32.GenerateDebugExceptionsFrom* on page J1-7631.
- *aarch32/debug/pmu/AArch32.CheckForPMUOverflow* on page J1-7632.
- *aarch32/debug/pmu/AArch32.CountEvents* on page J1-7632.
- *aarch32/debug/takeexceptiondbg/AArch32.EnterHypModeInDebugState* on page J1-7633.
- *aarch32/debug/takeexceptiondbg/AArch32.EnterModeInDebugState* on page J1-7633.
- *aarch32/debug/takeexceptiondbg/AArch32.EnterMonitorModeInDebugState* on page J1-7634.
- *aarch32/debug/watchpoint/AArch32.WatchpointByteMatch* on page J1-7634.
- *aarch32/debug/watchpoint/AArch32.WatchpointMatch* on page J1-7635.

aarch32/debug/VCRMatch/AArch32.VCRMatch

```
// AArch32.VCRMatch()
// ==================

boolean AArch32.VCRMatch(bits(32) vaddress)

    if UsingAArch32() && ELUsingAArch32(EL1) && IsZero(vaddress<1:0>) && PSTATE.EL != EL2 then
        // Each bit position in this string corresponds to a bit in DBGVCR and an exception vector.
        match_word = Zeros(32);

        if vaddress<31:5> == ExcVectorBase()<31:5> then
            if HaveEL(EL3) && !IsSecure() then
                match_word<UInt(vaddress<4:2>) + 24> = '1';     // Non-secure vectors
            else
                match_word<UInt(vaddress<4:2>) + 0> = '1';      // Secure vectors (or no EL3)

        if HaveEL(EL3) && ELUsingAArch32(EL3) && IsSecure() && vaddress<31:5> == MVBAR<31:5> then
            match_word<UInt(vaddress<4:2>) + 8> = '1';          // Monitor vectors
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        // Mask out bits not corresponding to vectors.
        if !HaveEL(EL3) then
            mask = '00000000':'00000000':'00000000':'11011110'; // DBGVCR[31:8] are RES0
        elsif !ELUsingAArch32(EL3) then
            mask = '11011110':'00000000':'00000000':'11011110'; // DBGVCR[15:8] are RES0
        else
            mask = '11011110':'00000000':'11011100':'11011110';

        match_word = match_word AND DBGVCR AND mask;
        match = !IsZero(match_word);

        // Check for UNPREDICTABLE case - match on Prefetch Abort and Data Abort vectors
        if !IsZero(match_word<28:27,12:11,4:3>) && DebugTarget() == PSTATE.EL then
            match = ConstrainUnpredictableBool();
    else
        match = FALSE;

    return match;
```

aarch32/debug/authentication/AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled

```
// AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
// ========================================================

boolean AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
    // The definition of this function is IMPLEMENTATION DEFINED.
    // In the recommended interface, AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled returns
    // the state of the (DBGEN AND SPIDEN) signal.
    if !HaveEL(EL3) && !IsSecure() then return FALSE;
    return DBGEN == HIGH && SPIDEN == HIGH;
```

aarch32/debug/breakpoint/AArch32.BreakpointMatch

```
// AArch32.BreakpointMatch()
// =========================
// Breakpoint matching in an AArch32 translation regime.

(boolean,boolean) AArch32.BreakpointMatch(integer n, bits(32) vaddress,  integer size)
    assert ELUsingAArch32(S1TranslationRegime());
    assert n <= UInt(DBGDIDR.BRPs);

    enabled = DBGBCR[n].E == '1';
    ispriv = PSTATE.EL != EL0;
    linked = DBGBCR[n].BT == '0x01';
    isbreakpnt = TRUE;
    linked_to = FALSE;

    state_match = AArch32.StateMatch(DBGBCR[n].SSC, DBGBCR[n].HMC, DBGBCR[n].PMC,
                                     linked, DBGBCR[n].LBN, isbreakpnt, ispriv);
    (value_match, value_mismatch) = AArch32.BreakpointValueMatch(n, vaddress, linked_to);

    if size == 4 then                 // Check second halfword
        // If the breakpoint address and BAS of an Address breakpoint match the address of the
        // second halfword of an instruction, but not the address of the first halfword, it is
        // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
        // event.
        (match_i, mismatch_i) = AArch32.BreakpointValueMatch(n, vaddress + 2, linked_to);
        if !value_match && match_i then
            value_match = ConstrainUnpredictableBool();
        if value_mismatch && !mismatch_i then
            value_mismatch = ConstrainUnpredictableBool();
    if vaddress<1> == '1' && DBGBCR[n].BAS == '1111' then
        // The above notwithstanding, if DBGBCR[n].BAS == '1111', then it is CONSTRAINED
        // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        // at the address DBGBVR[n]+2.
        if value_match then value_match = ConstrainUnpredictableBool();
        if !value_mismatch then value_mismatch = ConstrainUnpredictableBool();

    match = value_match && state_match && enabled;
    mismatch = value_mismatch && state_match && enabled;

    return (match, mismatch);
```

aarch32/debug/breakpoint/AArch32.BreakpointValueMatch

```
// AArch32.BreakpointValueMatch()
// ==============================
// The first result is whether an Address Match or Context breakpoint is programmed on the
// instruction at "address". The second result is whether an Address Mismatch breakpoint is
// programmed on the instruction, that is, whether the instruction should be stepped.

(boolean,boolean) AArch32.BreakpointValueMatch(integer n, bits(32) vaddress, boolean linked_to)

    // "n" is the identity of the breakpoint unit to match against.
    // "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
    //   matching breakpoints.
    // "linked_to" is TRUE if this is a call from StateMatch for linking.

    // If a non-existent breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
    // no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
    if n > UInt(DBGDIDR.BRPs) then
        (c, n) = ConstrainUnpredictableInteger(0, UInt(DBGDIDR.BRPs));
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return (FALSE,FALSE);

    // If this breakpoint is not enabled, it cannot generate a match. (This could also happen on a
    // call from StateMatch for linking).
    if DBGBCR[n].E == '0' then return (FALSE,FALSE);

    context_aware = (n >= UInt(DBGDIDR.BRPs) - UInt(DBGDIDR.CTX_CMPs));

    // If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
    dbgtype = DBGBCR[n].BT;

    if ((dbgtype IN {'011x','11xx'} && !HaveVirtHostExt()) ||             // Context matching
          (dbgtype == '010x' && HaltOnBreakpointOrWatchpoint()) ||        // Address mismatch
          (dbgtype != '0x0x' && !context_aware) ||                        // Context matching
          (dbgtype == '1xxx' && !HaveEL(EL2))) then                       // EL2 extension
        (c, dbgtype) = ConstrainUnpredictableBits();
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return (FALSE,FALSE);
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    // Determine what to compare against.
    match_addr = (dbgtype == '0x0x');
    mismatch   = (dbgtype == '010x');
    match_vmid = (dbgtype == '10xx');
    match_cid1 = (dbgtype == 'xx1x');
    match_cid2 = (dbgtype == '11xx');
    linked     = (dbgtype == 'xxx1');

    // If this is a call from StateMatch, return FALSE if the breakpoint is not programmed for a
    // VMID and/or context ID match, of if not context-aware. The above assertions mean that the
    // code can just test for match_addr == TRUE to confirm all these things.
    if linked_to && (!linked || match_addr) then return (FALSE,FALSE);

    // If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
    if !linked_to && linked && !match_addr then return (FALSE,FALSE);

    // Do the comparison.
    if match_addr then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        byte = UInt(vaddress<1:0>);
        assert byte IN {0,2};                     // "vaddress" is halfword aligned
        byte_select_match = (DBGBCR[n].BAS<byte> == '1');
        BVR_match = vaddress<31:2> == DBGBVR[n]<31:2> && byte_select_match;
    elsif match_cid1 then
        BVR_match = (PSTATE.EL != EL2 && CONTEXTIDR == DBGBVR[n]<31:0>);
    if match_vmid then
        if ELUsingAArch32(EL2) then
            vmid = ZeroExtend(VTTBR.VMID, 16);
            bvr_vmid = ZeroExtend(DBGBXVR[n]<7:0>, 16);
        elsif !Have16bitVMID() || VTCR_EL2.VS == '0' then
            vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
            bvr_vmid = ZeroExtend(DBGBXVR[n]<7:0>, 16);
        else
            vmid = VTTBR_EL2.VMID;
            bvr_vmid = DBGBXVR[n]<15:0>;
        BXVR_match = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                      vmid == bvr_vmid);
    elsif match_cid2 then
        BXVR_match = (!IsSecure() && HaveVirtHostExt() &&
                      !ELUsingAArch32(EL2) &&
                      DBGBXVR[n]<31:0> == CONTEXTIDR_EL2);

    bvr_match_valid = (match_addr || match_cid1);
    bxvr_match_valid = (match_vmid || match_cid2);

    match = (!bxvr_match_valid || BXVR_match) && (!bvr_match_valid || BVR_match);

    return (match && !mismatch, !match && mismatch);
```

aarch32/debug/breakpoint/AArch32.StateMatch

```
// AArch32.StateMatch()
// ====================
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch32.StateMatch(bits(2) SSC, bit HMC, bits(2) PxC, boolean linked, bits(4) LBN,
                           boolean isbreakpnt,  boolean ispriv)
    // "SSC", "HMC", "PxC" are the control fields from the DBGBCR[n] or DBGWCR[n] register.
    // "linked" is TRUE if this is a linked breakpoint/watchpoint type.
    // "LBN" is the linked breakpoint number from the DBGBCR[n] or DBGWCR[n] register.
    // "isbreakpnt" is TRUE for breakpoints, FALSE for watchpoints.
    // "ispriv" is valid for watchpoints, and selects between privileged and unprivileged accesses.

    // If parameters are set to a reserved type, behaves as either disabled or a defined type
    (c, SSC, HMC, PxC) = CheckValidStateMatch(SSC, HMC, PxC, isbreakpnt);
    if c == Constraint_DISABLED then return FALSE;
    // Otherwise the HMC,SSC,PxC values are either valid or the values returned by
    // CheckValidStateMatch are valid.

    PL2_match = HaveEL(EL2) && ((HMC == '1' && (SSC:PxC != '1000')) || SSC == '11');
    PL1_match = PxC<0> == '1';
    PL0_match = PxC<1> == '1';
    SSU_match = isbreakpnt && HMC == '0' && PxC == '00' && SSC != '11';

    if !ispriv && !isbreakpnt then
        priv_match = PL0_match;
    elsif SSU_match then
        priv_match = PSTATE.M IN {M32_User,M32_Svc,M32_System};
    else
        case PSTATE.EL of
            when EL3  priv_match = PL1_match;           // EL3 and EL1 are both PL1
            when EL2  priv_match = PL2_match;
            when EL1  priv_match = PL1_match;
            when EL0  priv_match = PL0_match;

    case SSC of
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        when '00'  security_state_match = TRUE;                        // Both
        when '01'  security_state_match = !IsSecure();                 // Non-secure only
        when '10'  security_state_match = IsSecure();                  // Secure only
        when '11'  security_state_match = (HMC == '1' || IsSecure());  // HMC=1 -> Both, 0 -> Secure
only

    if linked then
        // "LBN" must be an enabled context-aware breakpoint unit. If it is not context-aware then
        // it is CONSTRAINED UNPREDICTABLE whether this gives no match, or LBN is mapped to some
        // UNKNOWN breakpoint that is context-aware.
        lbn = UInt(LBN);
        first_ctx_cmp = (UInt(DBGDIDR.BRPs) - UInt(DBGDIDR.CTX_CMPs));
        last_ctx_cmp = UInt(DBGDIDR.BRPs);
        if (lbn < first_ctx_cmp || lbn > last_ctx_cmp) then
            (c, lbn) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp);
            assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
            case c of
                when Constraint_DISABLED  return FALSE;      // Disabled
                when Constraint_NONE      linked = FALSE;    // No linking
                // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint

    if linked then
        vaddress = bits(32) UNKNOWN;
        linked_to = TRUE;
        (linked_match,-) = AArch32.BreakpointValueMatch(lbn, vaddress, linked_to);

    return priv_match && security_state_match && (!linked || linked_match);
```

aarch32/debug/enables/AArch32.GenerateDebugExceptions

```
// AArch32.GenerateDebugExceptions()
// ================================

boolean AArch32.GenerateDebugExceptions()
    return AArch32.GenerateDebugExceptionsFrom(PSTATE.EL, IsSecure());
```

aarch32/debug/enables/AArch32.GenerateDebugExceptionsFrom

```
// AArch32.GenerateDebugExceptionsFrom()
// ====================================

boolean AArch32.GenerateDebugExceptionsFrom(bits(2) from, boolean secure)

    if from == EL0 && !ELStateUsingAArch32(EL1, secure) then
        mask = bit UNKNOWN;                          // PSTATE.D mask, unused for EL0 case
        return AArch64.GenerateDebugExceptionsFrom(from, secure, mask);

    if DBGOSLSR.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;

    if HaveEL(EL3) && secure then
        spd = if ELUsingAArch32(EL3) then SDCR.SPD else MDCR_EL3.SPD32;
        if spd<1> == '1' then
            enabled = spd<0> == '1';
        else
            // SPD == 0b01 is reserved, but behaves the same as 0b00.
            enabled = AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled();
        if from == EL0 then enabled = enabled || SDER.SUIDEN == '1';
    else
        enabled = from != EL2;

    return enabled;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch32/debug/pmu/AArch32.CheckForPMUOverflow

```
// AArch32.CheckForPMUOverflow()
// =============================
// Signal Performance Monitors overflow IRQ and CTI overflow events

boolean AArch32.CheckForPMUOverflow()

    if !ELUsingAArch32(EL1) then return AArch64.CheckForPMUOverflow();
    pmuirq = PMCR.E == '1' && PMINTENSET<31> == '1' && PMOVSSET<31> == '1';
    for n = 0 to UInt(PMCR.N) - 1
        if HaveEL(EL2) then
            hpmn = if !ELUsingAArch32(EL2) then MDCR_EL2.HPMN else HDCR.HPMN;
            hpme = if !ELUsingAArch32(EL2) then MDCR_EL2.HPME else HDCR.HPME;
            E = (if n < UInt(hpmn) then PMCR.E else hpme);
        else
            E = PMCR.E;
        if E == '1' && PMINTENSET<n> == '1' && PMOVSSET<n> == '1' then pmuirq = TRUE;

    SetInterruptRequestLevel(InterruptID_PMUIRQ, if pmuirq then HIGH else LOW);

    CTI_SetEventLevel(CrossTriggerIn_PMUOverflow, if pmuirq then HIGH else LOW);

    // The request remains set until the condition is cleared. (For example, an interrupt handler
    // or cross-triggered event handler clears the overflow status flag by writing to PMOVSCLR_EL0.)

    return pmuirq;
```

aarch32/debug/pmu/AArch32.CountEvents

```
// AArch32.CountEvents()
// =====================
// Return TRUE if counter "n" should count its event. For the cycle counter, n == 31.

boolean AArch32.CountEvents(integer n)
    assert n == 31 || n < UInt(PMCR.N);

    if !ELUsingAArch32(EL1) then return AArch64.CountEvents(n);
    // Event counting is disabled in Debug state
    debug = Halted();

    // In Non-secure state, some counters are reserved for EL2
    if HaveEL(EL2) then
        hpmn = if !ELUsingAArch32(EL2) then MDCR_EL2.HPMN else HDCR.HPMN;
        hpme = if !ELUsingAArch32(EL2) then MDCR_EL2.HPME else HDCR.HPME;
        if HaveHPMDExt() then
            hpmd = if !ELUsingAArch32(EL2) then MDCR_EL2.HPMD else HDCR.HPMD;
        E = if n < UInt(hpmn) || n == 31 then PMCR.E else hpme;
    else
        E = PMCR.E;
    enabled = E == '1' && PMCNTENSET<n> == '1';

    // Event counting in Secure state is prohibited unless any one of:
    // * EL3 is not implemented
    // * EL3 is using AArch64 and MDCR_EL3.SPME == 1
    // * EL3 is using AArch32 and SDCR.SPME == 1
    // * Executing at EL0, and SDER.SUNIDEN == 1.
    spme = (if ELUsingAArch32(EL3) then SDCR.SPME else MDCR_EL3.SPME);
    prohibited = HaveEL(EL3) && IsSecure() && spme == '0' && (PSTATE.EL != EL0 || SDER.SUNIDEN == '0');

    // Event counting at EL2 is prohibited if all of:
    // * The HPMD Extension is implemented
    // * Executing at EL2
    // * PMNx is not reserved for EL2
    // * HDCR.HPMD == 1
    if !prohibited && HaveEL(EL2) && HaveHPMDExt() && PSTATE.EL == EL2 && (n < UInt(hpmn) || n == 31)
then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        prohibited = (hpmd == '1');

    // The IMPLEMENTATION DEFINED authentication interface might override software controls
    if prohibited && !HaveNoSecurePMUDisableOverride() then
        prohibited = !ExternalSecureNoninvasiveDebugEnabled();
    // For the cycle counter, PMCR.DP enables counting when otherwise prohibited
    if prohibited && n == 31 then prohibited = (PMCR.DP == '1');

    // Event counting can be filtered by the {P, U, NSK, NSU, NSH} bits
    filter = if n == 31 then PMCCFILTR else PMEVTYPER[n];

    P = filter<31>;
    U = filter<30>;
    NSK = if HaveEL(EL3) then filter<29> else '0';
    NSU = if HaveEL(EL3) then filter<28> else '0';
    NSH = if HaveEL(EL2) then filter<27> else '0';

    case PSTATE.EL of
        when EL0 filtered = if IsSecure() then U == '1' else U != NSU;
        when EL1 filtered = if IsSecure() then P == '1' else P != NSK;
        when EL2 filtered = (NSH == '0');
        when EL3 filtered = (P == '1');

    return !debug && enabled && !prohibited && !filtered;
```

aarch32/debug/takeexceptiondbg/AArch32.EnterHypModeInDebugState

```
// AArch32.EnterHypModeInDebugState()
// ==================================
// Take an exception in Debug state to Hyp mode.

AArch32.EnterHypModeInDebugState(ExceptionRecord exception)
    SynchronizeContext();
    assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2);

    AArch32.ReportHypEntry(exception);
    AArch32.WriteMode(M32_Hyp);
    SPSR[] = bits(32) UNKNOWN;
    ELR_hyp = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1';                             // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    PSTATE.E = HSCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
    EDSCR.ERR = '1';
    UpdateEDSCRFields();

    EndOfInstruction();
```

aarch32/debug/takeexceptiondbg/AArch32.EnterModeInDebugState

```
// AArch32.EnterModeInDebugState()
// ===============================
// Take an exception in Debug state to a mode other than Monitor and Hyp mode.

AArch32.EnterModeInDebugState(bits(5) target_mode)
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(target_mode);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    SPSR[] = bits(32) UNKNOWN;
    R[14] = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1';                             // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() && SCTLR.SPAN == '0' then PSTATE.PAN = '1';
    if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
    EDSCR.ERR = '1';
    UpdateEDSCRFields();                        // Update EDSCR processor state flags.

    EndOfInstruction();
```

aarch32/debug/takeexceptiondbg/AArch32.EnterMonitorModeInDebugState

```
// AArch32.EnterMonitorModeInDebugState()
// ======================================
// Take an exception in Debug state to Monitor mode.

AArch32.EnterMonitorModeInDebugState()
    SynchronizeContext();
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    from_secure = IsSecure();
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    SPSR[] = bits(32) UNKNOWN;
    R[14] = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1';                             // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTLR.SPAN == '0' then
            PSTATE.PAN = '1';
    if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    EDSCR.ERR = '1';
    UpdateEDSCRFields();                        // Update EDSCR processor state flags.

    EndOfInstruction();
```

aarch32/debug/watchpoint/AArch32.WatchpointByteMatch

```
// AArch32.WatchpointByteMatch()
// =============================

boolean AArch32.WatchpointByteMatch(integer n,  bits(32) vaddress)

    bottom = if DBGWVR[n]<2> == '1' then 2 else 3;            // Word or doubleword
    byte_select_match = (DBGWCR[n].BAS<UInt(vaddress<bottom-1:0>)> != '0');
    mask = UInt(DBGWCR[n].MASK);

    // If DBGWCR[n].MASK is non-zero value and DBGWCR[n].BAS is not set to '11111111', or
    // DBGWCR[n].BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
    // UNPREDICTABLE.
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if mask > 0 && !IsOnes(DBGWCR[n].BAS) then
        byte_select_match = ConstrainUnpredictableBool();
    else
        LSB = (DBGWCR[n].BAS AND NOT(DBGWCR[n].BAS - 1));  MSB = (DBGWCR[n].BAS + LSB);
        if !IsZero(MSB AND (MSB - 1)) then                     // Not contiguous
            byte_select_match = ConstrainUnpredictableBool();
            bottom = 3;                                        // For the whole doubleword

    // If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask > 0 && mask <= 2 then
        (c, mask) = ConstrainUnpredictableInteger(3, 31);
        assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
        case c of
            when Constraint_DISABLED  return FALSE;            // Disabled
            when Constraint_NONE      mask = 0;                // No masking
            // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value

    if mask > bottom then
        WVR_match = (vaddress<31:mask> == DBGWVR[n]<31:mask>);
        // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
        if WVR_match && !IsZero(DBGWVR[n]<mask-1:bottom>) then
            WVR_match = ConstrainUnpredictableBool();
    else
        WVR_match = vaddress<31:bottom> == DBGWVR[n]<31:bottom>;

    return WVR_match && byte_select_match;
```

aarch32/debug/watchpoint/AArch32.WatchpointMatch

```
// AArch32.WatchpointMatch()
// =========================
// Watchpoint matching in an AArch32 translation regime.

boolean AArch32.WatchpointMatch(integer n, bits(32) vaddress, integer size, boolean ispriv,
                                boolean iswrite)
    assert ELUsingAArch32(S1TranslationRegime());
    assert n <= UInt(DBGDIDR.WRPs);

    // "ispriv" is FALSE for LDRT/STRT instructions executed at EL1 and all
    // load/stores at EL0, TRUE for all other load/stores. "iswrite" is TRUE for stores, FALSE for
    // loads.
    enabled = DBGWCR[n].E == '1';
    linked = DBGWCR[n].WT == '1';
    isbreakpnt = FALSE;

    state_match = AArch32.StateMatch(DBGWCR[n].SSC, DBGWCR[n].HMC, DBGWCR[n].PAC,
                                     linked, DBGWCR[n].LBN, isbreakpnt, ispriv);

    ls_match = (DBGWCR[n].LSC<(if iswrite then 1 else 0)> == '1');

    value_match = FALSE;
    for byte = 0 to size - 1
        value_match = value_match || AArch32.WatchpointByteMatch(n, vaddress + byte);

    return value_match && state_match && ls_match && enabled;
```

J1.2.2 aarch32/exceptions

This section includes the following pseudocode functions:

- *aarch32/exceptions/aborts/AArch32.Abort* on page J1-7636.
- *aarch32/exceptions/aborts/AArch32.AbortSyndrome* on page J1-7637.
- *aarch32/exceptions/aborts/AArch32.CheckPCAlignment* on page J1-7637.
- *aarch32/exceptions/aborts/AArch32.ReportDataAbort* on page J1-7637.
- *aarch32/exceptions/aborts/AArch32.ReportPrefetchAbort* on page J1-7638.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *aarch32/exceptions/aborts/AArch32.TakeDataAbortException* on page J1-7639.
- *aarch32/exceptions/aborts/AArch32.TakePrefetchAbortException* on page J1-7639.
- *aarch32/exceptions/aborts/BranchTargetException* on page J1-7640.
- *aarch32/exceptions/asynch/AArch32.TakePhysicalFIQException* on page J1-7640.
- *aarch32/exceptions/asynch/AArch32.TakePhysicalIRQException* on page J1-7640.
- *aarch32/exceptions/asynch/AArch32.TakePhysicalSErrorException* on page J1-7641.
- *aarch32/exceptions/asynch/AArch32.TakeVirtualFIQException* on page J1-7642.
- *aarch32/exceptions/asynch/AArch32.TakeVirtualIRQException* on page J1-7642.
- *aarch32/exceptions/asynch/AArch32.TakeVirtualSErrorException* on page J1-7642.
- *aarch32/exceptions/debug/AArch32.SoftwareBreakpoint* on page J1-7643.
- *aarch32/exceptions/debug/DebugException* on page J1-7643.
- *aarch32/exceptions/exceptions/AArch32.CheckAdvSIMDOrFPRegisterTraps* on page J1-7643.
- *aarch32/exceptions/exceptions/AArch32.ExceptionClass* on page J1-7643.
- *aarch32/exceptions/exceptions/AArch32.GeneralExceptionsToAArch64* on page J1-7644.
- *aarch32/exceptions/exceptions/AArch32.ReportHypEntry* on page J1-7644.
- *aarch32/exceptions/exceptions/AArch32.ResetControlRegisters* on page J1-7645.
- *aarch32/exceptions/exceptions/AArch32.TakeReset* on page J1-7645.
- *aarch32/exceptions/exceptions/ExcVectorBase* on page J1-7646.
- *aarch32/exceptions/ieeefp/AArch32.FPTrappedException* on page J1-7646.
- *aarch32/exceptions/syscalls/AArch32.CallHypervisor* on page J1-7646.
- *aarch32/exceptions/syscalls/AArch32.CallSupervisor* on page J1-7646.
- *aarch32/exceptions/syscalls/AArch32.TakeHVCException* on page J1-7647.
- *aarch32/exceptions/syscalls/AArch32.TakeSMCException* on page J1-7647.
- *aarch32/exceptions/syscalls/AArch32.TakeSVCException* on page J1-7647.
- *aarch32/exceptions/takeexception/AArch32.EnterHypMode* on page J1-7647.
- *aarch32/exceptions/takeexception/AArch32.EnterMode* on page J1-7648.
- *aarch32/exceptions/takeexception/AArch32.EnterMonitorMode* on page J1-7648.
- *aarch32/exceptions/traps/AArch32.CheckAdvSIMDOrFPEnabled* on page J1-7649.
- *aarch32/exceptions/traps/AArch32.CheckFPAdvSIMDTrap* on page J1-7650.
- *aarch32/exceptions/traps/AArch32.CheckForSMCUndefOrTrap* on page J1-7650.
- *aarch32/exceptions/traps/AArch32.CheckForSVCTrap* on page J1-7650.
- *aarch32/exceptions/traps/AArch32.CheckForWFxTrap* on page J1-7651.
- *aarch32/exceptions/traps/AArch32.CheckITEnabled* on page J1-7651.
- *aarch32/exceptions/traps/AArch32.CheckIllegalState* on page J1-7652.
- *aarch32/exceptions/traps/AArch32.CheckSETENDEnabled* on page J1-7652.
- *aarch32/exceptions/traps/AArch32.SystemAccessTrap* on page J1-7652.
- *aarch32/exceptions/traps/AArch32.SystemAccessTrapSyndrome* on page J1-7652.
- *aarch32/exceptions/traps/AArch32.TakeHypTrapException* on page J1-7653.
- *aarch32/exceptions/traps/AArch32.TakeMonitorTrapException* on page J1-7654.
- *aarch32/exceptions/traps/AArch32.TakeUndefInstrException* on page J1-7654.
- *aarch32/exceptions/traps/AArch32.UndefinedFault* on page J1-7654.

aarch32/exceptions/aborts/AArch32.Abort

```
// AArch32.Abort()
// ===============
// Abort and Debug exception handling in an AArch32 translation regime.

AArch32.Abort(bits(32) vaddress, FaultRecord fault)

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = (HCR_EL2.TGE == '1' || IsSecondStage(fault) ||
                            (HaveRASExt() && HCR2.TEA == '1' && IsExternalAbort(fault)) ||
                            (IsDebugException(fault) && MDCR_EL2.TDE == '1'));

    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.EA == '1' && IsExternalAbort(fault);

    if route_to_aarch64 then
        AArch64.Abort(ZeroExtend(vaddress), fault);
    elsif fault.acctype == AccType_IFETCH then
        AArch32.TakePrefetchAbortException(vaddress, fault);
    else
        AArch32.TakeDataAbortException(vaddress, fault);
```

aarch32/exceptions/aborts/AArch32.AbortSyndrome

```
// AArch32.AbortSyndrome()
// =======================
// Creates an exception syndrome record for Abort  exceptions taken to Hyp mode
// from an AArch32 translation regime.

ExceptionRecord AArch32.AbortSyndrome(Exception exceptype, FaultRecord fault, bits(32) vaddress)
    exception = ExceptionSyndrome(exceptype);

    d_side = exceptype == Exception_DataAbort;

    exception.syndrome = AArch32.FaultSyndrome(d_side, fault);
    exception.vaddress = ZeroExtend(vaddress);
    if IPAValid(fault) then
        exception.ipavalid = TRUE;
        exception.NS = fault.ipaddress.NS;
        exception.ipaddress = ZeroExtend(fault.ipaddress.address);
    else
        exception.ipavalid = FALSE;

    return exception;
```

aarch32/exceptions/aborts/AArch32.CheckPCAlignment

```
// AArch32.CheckPCAlignment()
// ==========================

AArch32.CheckPCAlignment()

    bits(32) pc = ThisInstrAddr();
    if (CurrentInstrSet() == InstrSet_A32 && pc<1> == '1') || pc<0> == '1' then
        if AArch32.GeneralExceptionsToAArch64() then AArch64.PCAlignmentFault();

        // Generate an Alignment fault Prefetch Abort exception
        vaddress = pc;
        acctype = AccType_IFETCH;
        iswrite = FALSE;
        secondstage = FALSE;
        AArch32.Abort(vaddress, AArch32.AlignmentFault(acctype, iswrite, secondstage));
```

aarch32/exceptions/aborts/AArch32.ReportDataAbort

```
// AArch32.ReportDataAbort()
// =========================
// Report syndrome information for aborts taken to modes other than Hyp mode.

AArch32.ReportDataAbort(boolean route_to_monitor, FaultRecord fault, bits(32) vaddress)

    // The encoding used in the IFSR or DFSR can be Long-descriptor format or Short-descriptor
    // format.  Normally, the current translation table format determines the format. For an abort
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    // from Non-secure state to Monitor mode, the IFSR or DFSR uses the Long-descriptor format if
    // any of the following applies:
    // * The Secure TTBCR.EAE is set to 1.
    // * The abort is synchronous and either:
    //   - It is taken from Hyp mode.
    //   - It is taken from EL1 or EL0, and the Non-secure TTBCR.EAE is set to 1.
    long_format = FALSE;
    if route_to_monitor && !IsSecure() then
        long_format = TTBCR_S.EAE == '1';
        if !IsSErrorInterrupt(fault) && !long_format then
            long_format = PSTATE.EL == EL2 || TTBCR.EAE == '1';
    else
        long_format = TTBCR.EAE == '1';
    d_side = TRUE;
    if long_format then
        syndrome = AArch32.FaultStatusLD(d_side, fault);
    else
        syndrome = AArch32.FaultStatusSD(d_side, fault);

    if fault.acctype == AccType_IC then
        if (!long_format &&
            boolean IMPLEMENTATION_DEFINED "Report I-cache maintenance fault in IFSR") then
            i_syndrome = syndrome;
            syndrome<10,3:0> = EncodeSDFSC(Fault_ICacheMaint, 1);
        else
            i_syndrome = bits(32) UNKNOWN;
        if route_to_monitor then
            IFSR_S = i_syndrome;
        else
            IFSR = i_syndrome;

    if route_to_monitor then
        DFSR_S = syndrome;
        DFAR_S = vaddress;
    else
        DFSR = syndrome;
        DFAR = vaddress;

    return;
```

aarch32/exceptions/aborts/AArch32.ReportPrefetchAbort

```
// AArch32.ReportPrefetchAbort()
// ============================
// Report syndrome information for aborts taken to modes other than Hyp mode.

AArch32.ReportPrefetchAbort(boolean route_to_monitor, FaultRecord fault, bits(32) vaddress)
    // The encoding used in the IFSR can be Long-descriptor format or Short-descriptor format.
    // Normally, the current translation table format determines the format. For an abort from
    // Non-secure state to Monitor mode, the IFSR uses the Long-descriptor format if any of the
    // following applies:
    // * The Secure TTBCR.EAE is set to 1.
    // * It is taken from Hyp mode.
    // * It is taken from EL1 or EL0, and the Non-secure TTBCR.EAE is set to 1.
    long_format = FALSE;
    if route_to_monitor && !IsSecure() then
        long_format = TTBCR_S.EAE == '1' || PSTATE.EL == EL2 || TTBCR.EAE == '1';
    else
        long_format = TTBCR.EAE == '1';

    d_side = FALSE;
    if long_format then
        fsr = AArch32.FaultStatusLD(d_side, fault);
    else
        fsr = AArch32.FaultStatusSD(d_side, fault);

    if route_to_monitor then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        IFSR_S = fsr;
        IFAR_S = vaddress;
    else
        IFSR = fsr;
        IFAR = vaddress;

    return;
```

aarch32/exceptions/aborts/AArch32.TakeDataAbortException

```
// AArch32.TakeDataAbortException()
// ================================

AArch32.TakeDataAbortException(bits(32) vaddress, FaultRecord fault)
    route_to_monitor = HaveEL(EL3) && SCR.EA == '1' && IsExternalAbort(fault);
    route_to_hyp = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} &&
                    (HCR.TGE == '1' || IsSecondStage(fault) ||
                    (HaveRASExt() && HCR2.TEA == '1' && IsExternalAbort(fault)) ||
                    (IsDebugException(fault) && HDCR.TDE == '1')));
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x10;
    lr_offset = 8;

    if IsDebugException(fault) then DBGDSCRext.MOE = fault.debugmoe;
    if route_to_monitor then
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        exception = AArch32.AbortSyndrome(Exception_DataAbort, fault, vaddress);
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
```

aarch32/exceptions/aborts/AArch32.TakePrefetchAbortException

```
// AArch32.TakePrefetchAbortException()
// ====================================

AArch32.TakePrefetchAbortException(bits(32) vaddress, FaultRecord fault)
    route_to_monitor = HaveEL(EL3) && SCR.EA == '1' && IsExternalAbort(fault);
    route_to_hyp = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} &&
                    (HCR.TGE == '1' || IsSecondStage(fault) ||
                    (HaveRASExt() && HCR2.TEA == '1' && IsExternalAbort(fault)) ||
                    (IsDebugException(fault) && HDCR.TDE == '1')));

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0C;
    lr_offset = 4;

    if IsDebugException(fault) then DBGDSCRext.MOE = fault.debugmoe;
    if route_to_monitor then
        AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        if fault.statuscode == Fault_Alignment then             // PC Alignment fault
            exception = ExceptionSyndrome(Exception_PCAlignment);
            exception.vaddress = ThisInstrAddr();
        else
            exception = AArch32.AbortSyndrome(Exception_InstructionAbort, fault, vaddress);
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
```

aarch32/exceptions/aborts/BranchTargetException

```
// BranchTargetException
// =====================
// Raise branch target exception.

AArch64.BranchTargetException(bits(52) vaddress)

    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_BranchTarget);
    exception.syndrome<1:0>   = PSTATE.BTYPE;
    exception.syndrome<24:2>  = Zeros();         // RES0

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch32/exceptions/asynch/AArch32.TakePhysicalFIQException

```
// AArch32.TakePhysicalFIQException()
// ==================================

AArch32.TakePhysicalFIQException()

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = HCR_EL2.TGE == '1' || (HCR_EL2.FMO == '1' && !IsInHost());

    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.FIQ == '1';

    if route_to_aarch64 then AArch64.TakePhysicalFIQException();
    route_to_monitor = HaveEL(EL3) && SCR.FIQ == '1';
    route_to_hyp = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR.TGE == '1' || HCR.FMO == '1'));
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x1C;
    lr_offset = 4;
    if route_to_monitor then
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        exception = ExceptionSyndrome(Exception_FIQ);
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
        AArch32.EnterMode(M32_FIQ, preferred_exception_return, lr_offset, vect_offset);
```

aarch32/exceptions/asynch/AArch32.TakePhysicalIRQException

```
// AArch32.TakePhysicalIRQException()
// ==================================
// Take an enabled physical IRQ exception.

AArch32.TakePhysicalIRQException()
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = HCR_EL2.TGE == '1' || (HCR_EL2.IMO == '1' && !IsInHost());
    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.IRQ == '1';

    if route_to_aarch64 then AArch64.TakePhysicalIRQException();

    route_to_monitor = HaveEL(EL3) && SCR.IRQ == '1';
    route_to_hyp = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR.TGE == '1' || HCR.IMO == '1'));
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x18;
    lr_offset = 4;
    if route_to_monitor then
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        exception = ExceptionSyndrome(Exception_IRQ);
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
        AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);
```

aarch32/exceptions/asynch/AArch32.TakePhysicalSErrorException

```
// AArch32.TakePhysicalSErrorException()
// =====================================

AArch32.TakePhysicalSErrorException(boolean parity, bit extflag, bits(2) errortype,
                                    boolean impdef_syndrome, bits(24) full_syndrome)

    ClearPendingPhysicalSError();
    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);

    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = (HCR_EL2.TGE == '1' || (!IsInHost() && HCR_EL2.AMO == '1'));
    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.EA == '1';

    if route_to_aarch64 then
        AArch64.TakePhysicalSErrorException(impdef_syndrome, full_syndrome);

    route_to_monitor = HaveEL(EL3) && SCR.EA == '1';
    route_to_hyp = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR.TGE == '1' || HCR.AMO == '1'));
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x10;
    lr_offset = 8;

    fault = AArch32.AsynchExternalAbort(parity, errortype, extflag);
    vaddress = bits(32) UNKNOWN;
    if route_to_monitor then
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        exception = AArch32.AbortSyndrome(Exception_DataAbort, fault, vaddress);
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch32/exceptions/asynch/AArch32.TakeVirtualFIQException

```
// AArch32.TakeVirtualFIQException()
// ================================

AArch32.TakeVirtualFIQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    if ELUsingAArch32(EL2) then  // Virtual IRQ enabled if TGE==0 and FMO==1
        assert HCR.TGE == '0' && HCR.FMO == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1';
    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualFIQException();

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x1C;
    lr_offset = 4;

    AArch32.EnterMode(M32_FIQ, preferred_exception_return, lr_offset, vect_offset);
```

aarch32/exceptions/asynch/AArch32.TakeVirtualIRQException

```
// AArch32.TakeVirtualIRQException()
// ================================

AArch32.TakeVirtualIRQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();

    if ELUsingAArch32(EL2) then   // Virtual IRQs enabled if TGE==0 and IMO==1
        assert HCR.TGE == '0' && HCR.IMO == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1';

    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualIRQException();

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x18;
    lr_offset = 4;

    AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);
```

aarch32/exceptions/asynch/AArch32.TakeVirtualSErrorException

```
// AArch32.TakeVirtualSErrorException()
// ===================================

AArch32.TakeVirtualSErrorException(bit extflag, bits(2) errortype, boolean impdef_syndrome, bits(24)
full_syndrome)

    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    if ELUsingAArch32(EL2) then   // Virtual SError enabled if TGE==0 and AMO==1
        assert HCR.TGE == '0' && HCR.AMO == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';
    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualSErrorException(impdef_syndrome,
full_syndrome);

    route_to_monitor = FALSE;

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x10;
    lr_offset = 8;

    vaddress = bits(32) UNKNOWN;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    parity = FALSE;
    if HaveRASExt() then
        if ELUsingAArch32(EL2) then
            fault = AArch32.AsynchExternalAbort(FALSE, VDFSR.AET, VDFSR.ExT);
        else
            fault = AArch32.AsynchExternalAbort(FALSE, VSESR_EL2.AET, VSESR_EL2.ExT);
    else
        fault = AArch32.AsynchExternalAbort(parity, errortype, extflag);

    ClearPendingVirtualSError();
    AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
    AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
```

aarch32/exceptions/debug/AArch32.SoftwareBreakpoint

```
// AArch32.SoftwareBreakpoint()
// ============================

AArch32.SoftwareBreakpoint(bits(16) immediate)

    if (EL2Enabled() && !ELUsingAArch32(EL2) &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1')) || !ELUsingAArch32(EL1) then
        AArch64.SoftwareBreakpoint(immediate);
    vaddress = bits(32) UNKNOWN;
    acctype = AccType_IFETCH;           // Take as a Prefetch Abort
    iswrite = FALSE;
    entry = DebugException_BKPT;

    fault = AArch32.DebugFault(acctype, iswrite, entry);
    AArch32.Abort(vaddress, fault);
```

aarch32/exceptions/debug/DebugException

```
constant bits(4) DebugException_Breakpoint  = '0001';
constant bits(4) DebugException_BKPT        = '0011';
constant bits(4) DebugException_VectorCatch = '0101';
constant bits(4) DebugException_Watchpoint  = '1010';
```

aarch32/exceptions/exceptions/AArch32.CheckAdvSIMDOrFPRegisterTraps

```
// AArch32.CheckAdvSIMDOrFPRegisterTraps()
// =======================================
// Check if an instruction that accesses an Advanced SIMD and
// floating-point System register is trapped by an appropriate HCR.TIDx
// ID group trap control.

AArch32.CheckAdvSIMDOrFPRegisterTraps(bits(4) reg)

    if PSTATE.EL == EL1 && EL2Enabled() then
        tid0 = if ELUsingAArch32(EL2) then HCR.TID0 else HCR_EL2.TID0;
        tid3 = if ELUsingAArch32(EL2) then HCR.TID3 else HCR_EL2.TID3;

        if (tid0 == '1' && reg == '0000')                           // FPSID
          || (tid3 == '1' && reg IN {'0101', '0110', '0111'}) then  // MVFRx
            if ELUsingAArch32(EL2) then
                AArch32.SystemAccessTrap(M32_Hyp, 0x8);             // Exception_AdvSIMDFPAccessTrap
            else
                AArch64.AArch32SystemAccessTrap(EL2, 0x8);          // Exception_AdvSIMDFPAccessTrap
```

aarch32/exceptions/exceptions/AArch32.ExceptionClass

```
// AArch32.ExceptionClass()
// ========================
// Returns the Exception Class and Instruction Length fields to be reported in HSR
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
(integer,bit) AArch32.ExceptionClass(Exception exceptype)

    il = if ThisInstrLength() == 32 then '1' else '0';

    case exceptype of
        when Exception_Uncategorized        ec = 0x00; il = '1';
        when Exception_WFxTrap              ec = 0x01;
        when Exception_CP15RTTrap           ec = 0x03;
        when Exception_CP15RRTTrap          ec = 0x04;
        when Exception_CP14RTTrap           ec = 0x05;
        when Exception_CP14DTTrap           ec = 0x06;
        when Exception_AdvSIMDFPAccessTrap  ec = 0x07;
        when Exception_FPIDTrap             ec = 0x08;
        when Exception_PACTrap              ec = 0x09;
        when Exception_CP14RRTTrap          ec = 0x0C;
        when Exception_BranchTarget         ec = 0x0D;
        when Exception_IllegalState         ec = 0x0E; il = '1';
        when Exception_SupervisorCall       ec = 0x11;
        when Exception_HypervisorCall       ec = 0x12;
        when Exception_MonitorCall          ec = 0x13;
        when Exception_ERetTrap             ec = 0x1A;
        when Exception_PACFail              ec = 0x1C;
        when Exception_InstructionAbort     ec = 0x20; il = '1';
        when Exception_PCAlignment          ec = 0x22; il = '1';
        when Exception_DataAbort            ec = 0x24;
        when Exception_NV2DataAbort         ec = 0x25;
        when Exception_FPTrappedException   ec = 0x28;
        otherwise                           Unreachable();

    if ec IN {0x20,0x24} && PSTATE.EL == EL2 then
        ec = ec + 1;

    return (ec,il);
```

aarch32/exceptions/exceptions/AArch32.GeneralExceptionsToAArch64

```
// AArch32.GeneralExceptionsToAArch64()
// ====================================
// Returns TRUE if exceptions normally routed to EL1 are being handled at an Exception
// level using AArch64, because either EL1 is using AArch64 or TGE is in force and EL2
// is using AArch64.

boolean AArch32.GeneralExceptionsToAArch64()
    return ((PSTATE.EL == EL0 && !ELUsingAArch32(EL1)) ||
            (EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1'));
```

aarch32/exceptions/exceptions/AArch32.ReportHypEntry

```
// AArch32.ReportHypEntry()
// ========================
// Report syndrome information to Hyp mode registers.

AArch32.ReportHypEntry(ExceptionRecord exception)

    Exception exceptype = exception.exceptype;

    (ec,il) = AArch32.ExceptionClass(exceptype);
    iss = exception.syndrome;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
        il = '1';

    HSR = ec<5:0>:il:iss;

    if exceptype IN {Exception_InstructionAbort, Exception_PCAlignment} then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        HIFAR = exception.vaddress<31:0>;
        HDFAR = bits(32) UNKNOWN;
    elsif exceptype == Exception_DataAbort then
        HIFAR = bits(32) UNKNOWN;
        HDFAR = exception.vaddress<31:0>;

    if exception.ipavalid then
        HPFAR<31:4> = exception.ipaddress<39:12>;
    else
        HPFAR<31:4> = bits(28) UNKNOWN;

    return;
```

aarch32/exceptions/exceptions/AArch32.ResetControlRegisters

```
// Resets System registers and memory-mapped control registers that have architecturally-defined
// reset values to those values.
AArch32.ResetControlRegisters(boolean cold_reset);
```

aarch32/exceptions/exceptions/AArch32.TakeReset

```
// AArch32.TakeReset()
// ===================
// Reset into AArch32 state

AArch32.TakeReset(boolean cold_reset)
    assert HighestELUsingAArch32();

    // Enter the highest implemented Exception level in AArch32 state
    if HaveEL(EL3) then
        AArch32.WriteMode(M32_Svc);
        SCR.NS = '0';                     // Secure state
    elsif HaveEL(EL2) then
        AArch32.WriteMode(M32_Hyp);
    else
        AArch32.WriteMode(M32_Svc);

    // Reset the CP14 and CP15 registers and other system components
    AArch32.ResetControlRegisters(cold_reset);
    FPEXC.EN = '0';

    // Reset all other PSTATE fields, including instruction set and endianness according to the
    // SCTLR values produced by the above call to ResetControlRegisters()
    PSTATE.<A,I,F> = '111';       // All asynchronous exceptions masked
    PSTATE.IT = '00000000';       // IT block state reset
    PSTATE.T = SCTLR.TE;          // Instruction set: TE=0: A32, TE=1: T32. PSTATE.J is RES0.
    PSTATE.E = SCTLR.EE;          // Endianness: EE=0: little-endian, EE=1: big-endian
    PSTATE.IL = '0';              // Clear Illegal Execution state bit

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // R14 or ELR_hyp and SPSR have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch32.ResetGeneralRegisters();
    AArch32.ResetSIMDFPRegisters();
    AArch32.ResetSpecialRegisters();
    ResetExternalDebugRegisters(cold_reset);

    bits(32) rv;                      // IMPLEMENTATION DEFINED reset vector

    if HaveEL(EL3) then
        if MVBAR<0> == '1' then           // Reset vector in MVBAR
            rv = MVBAR<31:1>:'0';
        else
            rv = bits(32) IMPLEMENTATION_DEFINED "reset vector address";
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        rv = RVBAR<31:1>:'0';

    // The reset vector must be correctly aligned
    assert rv<0> == '0' && (PSTATE.T == '1' || rv<1> == '0');

    BranchTo(rv, BranchType_RESET);
```

aarch32/exceptions/exceptions/ExcVectorBase

```
// ExcVectorBase()
// ===============

bits(32) ExcVectorBase()
    if SCTLR.V == '1' then  // Hivecs selected, base = 0xFFFF0000
        return Ones(16):Zeros(16);
    else
        return VBAR<31:5>:Zeros(5);
```

aarch32/exceptions/ieeefp/AArch32.FPTrappedException

```
// AArch32.FPTrappedException()
// ============================

AArch32.FPTrappedException(bits(8) accumulated_exceptions)
    if AArch32.GeneralExceptionsToAArch64() then
        is_ase = FALSE;
        element = 0;
        AArch64.FPTrappedException(is_ase, element, accumulated_exceptions);
    FPEXC.DEX    = '1';
    FPEXC.TFV    = '1';
    FPEXC<7,4:0> = accumulated_exceptions<7,4:0>;                  // IDF,IXF,UFF,OFF,DZF,IOF
    FPEXC<10:8>  = '111';                                          // VECITR is RES1

    AArch32.TakeUndefInstrException();
```

aarch32/exceptions/syscalls/AArch32.CallHypervisor

```
// AArch32.CallHypervisor()
// ========================
// Performs a HVC call

AArch32.CallHypervisor(bits(16) immediate)
    assert HaveEL(EL2);

    if !ELUsingAArch32(EL2) then
        AArch64.CallHypervisor(immediate);
    else
        AArch32.TakeHVCException(immediate);
```

aarch32/exceptions/syscalls/AArch32.CallSupervisor

```
// AArch32.CallSupervisor()
// ========================
// Calls the Supervisor

AArch32.CallSupervisor(bits(16) immediate)

    if AArch32.CurrentCond() != '1110' then
        immediate = bits(16) UNKNOWN;
    if AArch32.GeneralExceptionsToAArch64() then
        AArch64.CallSupervisor(immediate);
    else
        AArch32.TakeSVCException(immediate);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch32/exceptions/syscalls/AArch32.TakeHVCException

```
// AArch32.TakeHVCException()
// ==========================

AArch32.TakeHVCException(bits(16) immediate)
    assert HaveEL(EL2) && ELUsingAArch32(EL2);

    AArch32.ITAdvance();
    SSAdvance();
    bits(32) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x08;

    exception = ExceptionSyndrome(Exception_HypervisorCall);
    exception.syndrome<15:0> = immediate;

    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
        AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
```

aarch32/exceptions/syscalls/AArch32.TakeSMCException

```
// AArch32.TakeSMCException()
// ==========================

AArch32.TakeSMCException()
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    AArch32.ITAdvance();
    SSAdvance();
    bits(32) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x08;
    lr_offset = 0;

    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
```

aarch32/exceptions/syscalls/AArch32.TakeSVCException

```
// AArch32.TakeSVCException()
// ==========================

AArch32.TakeSVCException(bits(16) immediate)

    AArch32.ITAdvance();
    SSAdvance();
    route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';

    bits(32) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x08;
    lr_offset = 0;

    if PSTATE.EL == EL2 || route_to_hyp then
        exception = ExceptionSyndrome(Exception_SupervisorCall);
        exception.syndrome<15:0> = immediate;
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.EnterMode(M32_Svc, preferred_exception_return, lr_offset, vect_offset);
```

aarch32/exceptions/takeexception/AArch32.EnterHypMode

```
// AArch32.EnterHypMode()
// ======================
// Take an exception to Hyp mode.
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
AArch32.EnterHypMode(ExceptionRecord exception, bits(32) preferred_exception_return,
                     integer vect_offset)
    SynchronizeContext();
    assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2);

    spsr = GetPSRFromPSTATE();
    if !(exception.exceptype IN {Exception_IRQ, Exception_FIQ}) then
        AArch32.ReportHypEntry(exception);
    AArch32.WriteMode(M32_Hyp);
    SPSR[] = spsr;
    ELR_hyp = preferred_exception_return;
    PSTATE.T = HSCTLR.TE;                       // PSTATE.J is RES0
    PSTATE.SS = '0';
    if !HaveEL(EL3) || SCR_GEN[].EA == '0' then PSTATE.A = '1';
    if !HaveEL(EL3) || SCR_GEN[].IRQ == '0' then PSTATE.I = '1';
    if !HaveEL(EL3) || SCR_GEN[].FIQ == '0' then PSTATE.F = '1';
    PSTATE.E = HSCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HaveSSBSExt() then PSTATE.SSBS = HSCTLR.DSSBS;
    BranchTo(HVBAR<31:5>:vect_offset<4:0>, BranchType_EXCEPTION);

    EndOfInstruction();
```

aarch32/exceptions/takeexception/AArch32.EnterMode

```
// AArch32.EnterMode()
// ===================
// Take an exception to a mode other than Monitor and Hyp mode.

AArch32.EnterMode(bits(5) target_mode, bits(32) preferred_exception_return, integer lr_offset,
                  integer vect_offset)
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

    spsr = GetPSRFromPSTATE();
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(target_mode);
    SPSR[] = spsr;
    R[14] = preferred_exception_return + lr_offset;
    PSTATE.T = SCTLR.TE;                        // PSTATE.J is RES0
    PSTATE.SS = '0';
    if target_mode == M32_FIQ then
        PSTATE.<A,I,F> = '111';
    elsif target_mode IN {M32_Abort, M32_IRQ} then
        PSTATE.<A,I> = '11';
    else
        PSTATE.I = '1';
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() && SCTLR.SPAN == '0' then PSTATE.PAN = '1';
    if HaveSSBSExt() then PSTATE.SSBS = SCTLR.DSSBS;
    BranchTo(ExcVectorBase()<31:5>:vect_offset<4:0>, BranchType_EXCEPTION);

    EndOfInstruction();
```

aarch32/exceptions/takeexception/AArch32.EnterMonitorMode

```
// AArch32.EnterMonitorMode()
// ==========================
// Take an exception to Monitor mode.

AArch32.EnterMonitorMode(bits(32) preferred_exception_return, integer lr_offset,
                         integer vect_offset)
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    SynchronizeContext();
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    from_secure = IsSecure();
    spsr = GetPSRFromPSTATE();
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    SPSR[] = spsr;
    R[14] = preferred_exception_return + lr_offset;
    PSTATE.T = SCTLR.TE;                        // PSTATE.J is RES0
    PSTATE.SS = '0';
    PSTATE.<A,I,F> = '111';
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTLR.SPAN == '0' then
            PSTATE.PAN = '1';
    if HaveSSBSExt() then PSTATE.SSBS = SCTLR.DSSBS;
    BranchTo(MVBAR<31:5>:vect_offset<4:0>, BranchType_EXCEPTION);

    EndOfInstruction();
```

aarch32/exceptions/traps/AArch32.CheckAdvSIMDOrFPEnabled

```
// AArch32.CheckAdvSIMDOrFPEnabled()
// =================================
// Check against CPACR, FPEXC, HCPTR, NSACR, and CPTR_EL3.

AArch32.CheckAdvSIMDOrFPEnabled(boolean fpexc_check, boolean advsimd)
    if PSTATE.EL == EL0 && (!HaveEL(EL2) || (!ELUsingAArch32(EL2) && HCR_EL2.TGE == '0')) &&
!ELUsingAArch32(EL1) then
        // The PE behaves as if FPEXC.EN is 1
        AArch64.CheckFPAdvSIMDEnabled();
    elsif PSTATE.EL == EL0 && HaveEL(EL2) && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' &&
!ELUsingAArch32(EL1) then
        if fpexc_check && HCR_EL2.RW == '0' then
            fpexc_en = bits(1) IMPLEMENTATION_DEFINED "FPEXC.EN value when TGE==1 and RW==0";
            if fpexc_en == '0' then UNDEFINED;
        AArch64.CheckFPAdvSIMDEnabled();
    else
        cpacr_asedis = CPACR.ASEDIS;
        cpacr_cp10 = CPACR.cp10;

        if HaveEL(EL3) && ELUsingAArch32(EL3) && !IsSecure() then
            // Check if access disabled in NSACR
            if NSACR.NSASEDIS == '1' then cpacr_asedis = '1';
            if NSACR.cp10 == '0' then cpacr_cp10 = '00';

        if PSTATE.EL != EL2 then
            // Check if Advanced SIMD disabled in CPACR
            if advsimd && cpacr_asedis == '1' then UNDEFINED;

            // Check if access disabled in CPACR
            case cpacr_cp10 of
                when '00' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0;
                when '10' disabled = ConstrainUnpredictableBool();
                when '11' disabled = FALSE;
            if disabled then UNDEFINED;

        // If required, check FPEXC enabled bit.
        if fpexc_check && FPEXC.EN == '0' then UNDEFINED;

        AArch32.CheckFPAdvSIMDTrap(advsimd);    // Also check against HCPTR and CPTR_EL3
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch32/exceptions/traps/AArch32.CheckFPAdvSIMDTrap

```
// AArch32.CheckFPAdvSIMDTrap()
// ============================
// Check against CPTR_EL2 and CPTR_EL3.

AArch32.CheckFPAdvSIMDTrap(boolean advsimd)
    if EL2Enabled() && !ELUsingAArch32(EL2) then
        AArch64.CheckFPAdvSIMDTrap();
    else
        if HaveEL(EL2) && !IsSecure() then
            hcptr_tase = HCPTR.TASE;
            hcptr_cp10 = HCPTR.TCP10;

            if HaveEL(EL3) && ELUsingAArch32(EL3) && !IsSecure() then
                // Check if access disabled in NSACR
                if NSACR.NSASEDIS == '1' then hcptr_tase = '1';
                if NSACR.cp10 == '0' then hcptr_cp10 = '1';

            // Check if access disabled in HCPTR
            if (advsimd && hcptr_tase == '1') || hcptr_cp10 == '1' then
                exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
                exception.syndrome<24:20> = ConditionSyndrome();

                if advsimd then
                    exception.syndrome<5> = '1';
                else
                    exception.syndrome<5> = '0';
                    exception.syndrome<3:0> = '1010';         // coproc field, always 0xA

                if PSTATE.EL == EL2 then
                    AArch32.TakeUndefInstrException(exception);
                else
                    AArch32.TakeHypTrapException(exception);

        if HaveEL(EL3) && !ELUsingAArch32(EL3) then
            // Check if access disabled in CPTR_EL3
            if CPTR_EL3.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL3);
    return;
```

aarch32/exceptions/traps/AArch32.CheckForSMCUndefOrTrap

```
// AArch32.CheckForSMCUndefOrTrap()
// ================================
// Check for UNDEFINED or trap on SMC instruction

AArch32.CheckForSMCUndefOrTrap()
    if !HaveEL(EL3) || PSTATE.EL == EL0 then
        UNDEFINED;

    if EL2Enabled() && !ELUsingAArch32(EL2) then
        AArch64.CheckForSMCUndefOrTrap(Zeros(16));
    else
        route_to_hyp = HaveEL(EL2) && !IsSecure() && PSTATE.EL == EL1 && HCR.TSC == '1';
        if route_to_hyp then
            exception = ExceptionSyndrome(Exception_MonitorCall);
            AArch32.TakeHypTrapException(exception);
```

aarch32/exceptions/traps/AArch32.CheckForSVCTrap

```
// AArch32.CheckForSVCTrap()
// =========================
// Check for trap on SVC instruction

AArch32.CheckForSVCTrap(bits(16) immediate)
    if HaveFGTExt() then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        route_to_el2 = FALSE;
        if PSTATE.EL == EL0 then
       route_to_el2 = !ELUsingAArch32(EL1) && EL2Enabled() && HFGITR_EL2.SVC_EL0 == '1' &&
(HCR_EL2.<E2H, TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')) ;

        if route_to_el2 then
            exception = ExceptionSyndrome(Exception_SupervisorCall);
            exception.syndrome<15:0> = immediate;
            bits(64) preferred_exception_return = ThisInstrAddr();
            vect_offset = 0x0;

            AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

aarch32/exceptions/traps/AArch32.CheckForWFxTrap

```
// AArch32.CheckForWFxTrap()
// =========================
// Check for trap on WFE or WFI instruction

AArch32.CheckForWFxTrap(bits(2) target_el, boolean is_wfe)
    assert HaveEL(target_el);

    // Check for routing to AArch64
    if !ELUsingAArch32(target_el) then
        AArch64.CheckForWFxTrap(target_el, is_wfe);
        return;
    case target_el of
        when EL1 trap = (if is_wfe then SCTLR.nTWE else SCTLR.nTWI) == '0';
        when EL2 trap = (if is_wfe then HCR.TWE else HCR.TWI) == '1';
        when EL3 trap = (if is_wfe then SCR.TWE else SCR.TWI) == '1';
    if trap then
        if target_el == EL1 && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.WFxTrap(target_el, is_wfe);
        if target_el == EL3 then
            AArch32.TakeMonitorTrapException();
        elsif target_el == EL2 then
            exception = ExceptionSyndrome(Exception_WFxTrap);
            exception.syndrome<24:20> = ConditionSyndrome();
            exception.syndrome<0> = if is_wfe then '1' else '0';
            AArch32.TakeHypTrapException(exception);
        else
            AArch32.TakeUndefInstrException();
```

aarch32/exceptions/traps/AArch32.CheckITEnabled

```
// AArch32.CheckITEnabled()
// ========================
// Check whether the T32 IT instruction is disabled.

AArch32.CheckITEnabled(bits(4) mask)
    if PSTATE.EL == EL2 then
        it_disabled = HSCTLR.ITD;
    else
        it_disabled = (if ELUsingAArch32(EL1) then SCTLR.ITD else SCTLR[].ITD);
    if it_disabled == '1' then
        if mask != '1000' then UNDEFINED;

        // Otherwise whether the IT block is allowed depends on hw1 of the next instruction.
        next_instr = AArch32.MemSingle[NextInstrAddr(), 2, AccType_IFETCH, TRUE];

        if next_instr IN {'11xxxxxxxxxxxxxx', '1011xxxxxxxxxxxx', '10100xxxxxxxxxxx',
                          '01001xxxxxxxxxxx', '010001xxx1111xxx', '010001xx1xxxx111'} then
            // It is IMPLEMENTATION DEFINED whether the Undefined Instruction exception is
            // taken on the IT instruction or the next instruction. This is not reflected in
            // the pseudocode, which always takes the exception on the IT instruction. This
            // also does not take into account cases where the next instruction is UNPREDICTABLE.
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            UNDEFINED;

    return;
```

aarch32/exceptions/traps/AArch32.CheckIllegalState

```
// AArch32.CheckIllegalState()
// ===========================
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

AArch32.CheckIllegalState()
    if AArch32.GeneralExceptionsToAArch64() then
        AArch64.CheckIllegalState();
    elsif PSTATE.IL == '1' then
        route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';

        bits(32) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x04;

        if PSTATE.EL == EL2 || route_to_hyp then
            exception = ExceptionSyndrome(Exception_IllegalState);
            if PSTATE.EL == EL2 then
                AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
            else
                AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
        else
            AArch32.TakeUndefInstrException();
```

aarch32/exceptions/traps/AArch32.CheckSETENDEnabled

```
// AArch32.CheckSETENDEnabled()
// ============================
// Check whether the AArch32 SETEND instruction is disabled.

AArch32.CheckSETENDEnabled()
    if PSTATE.EL == EL2 then
        setend_disabled = HSCTLR.SED;
    else
        setend_disabled = (if ELUsingAArch32(EL1) then SCTLR.SED else SCTLR[].SED);
    if setend_disabled == '1' then
        UNDEFINED;

    return;
```

aarch32/exceptions/traps/AArch32.SystemAccessTrap

```
// AArch32.SystemAccessTrap()
// ==========================
// Trapped  system register access.

AArch32.SystemAccessTrap(bits(5) mode, integer ec)
    (valid, target_el) = ELFromM32(mode);
    assert valid && HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    if target_el == EL2 then
        exception = AArch32.SystemAccessTrapSyndrome(ThisInstr(), ec);
        AArch32.TakeHypTrapException(exception);
    else
        AArch32.TakeUndefInstrException();
```

aarch32/exceptions/traps/AArch32.SystemAccessTrapSyndrome

```
// AArch32.SystemAccessTrapSyndrome()
// ==================================
// Returns the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS, VMSR
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
instructions,
 // other than traps that are due to HCPTR or CPACR.

 ExceptionRecord AArch32.SystemAccessTrapSyndrome(bits(32) instr, integer ec)
     ExceptionRecord exception;

     case ec of
         when 0x0    exception = ExceptionSyndrome(Exception_Uncategorized);
         when 0x3    exception = ExceptionSyndrome(Exception_CP15RTTrap);
         when 0x4    exception = ExceptionSyndrome(Exception_CP15RRTTrap);
         when 0x5    exception = ExceptionSyndrome(Exception_CP14RTTrap);
         when 0x6    exception = ExceptionSyndrome(Exception_CP14DTTrap);
         when 0x7    exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
         when 0x8    exception = ExceptionSyndrome(Exception_FPIDTrap);
         when 0xC    exception = ExceptionSyndrome(Exception_CP14RRTTrap);
         otherwise   Unreachable();

     bits(20) iss = Zeros();

     if exception.exceptype IN {Exception_FPIDTrap, Exception_CP14RTTrap, Exception_CP15RTTrap} then
         // Trapped MRC/MCR, VMRS on FPSID
         iss<13:10> = instr<19:16>;        // CRn, Reg in case of VMRS
         iss<8:5>   = instr<15:12>;        // Rt
         iss<9>     = '0';                 // RES0

         if exception.exceptype != Exception_FPIDTrap then    // When trap is not for VMRS
             iss<19:17> = instr<7:5>;          // opc2
             iss<16:14> = instr<23:21>;        // opc1
             iss<4:1>   = instr<3:0>;          //CRm
         else //VMRS Access
             iss<19:17> = '000';               //opc2 - Hardcoded for VMRS
             iss<16:14> = '111';               //opc1 - Hardcoded for VMRS
             iss<4:1>   = '0000';              //CRm  - Hardcoded for VMRS
     elsif exception.exceptype IN {Exception_CP14RRTTrap, Exception_AdvSIMDFPAccessTrap,
Exception_CP15RRTTrap} then
         // Trapped MRRC/MCRR, VMRS/VMSR
         iss<19:16> = instr<7:4>;          // opc1
         iss<13:10> = instr<19:16>;        // Rt2
         iss<8:5> = instr<15:12>;          // Rt
         iss<4:1>   = instr<3:0>;         // CRm
     elsif exception.exceptype == Exception_CP14DTTrap then
         // Trapped LDC/STC
         iss<19:12> = instr<7:0>;         // imm8
         iss<4>     = instr<23>;          // U
         iss<2:1>   = instr<24,21>;       // P,W
         if instr<19:16> == '1111' then   // Rn==15, LDC(Literal addressing)/STC
             iss<8:5> = bits(4) UNKNOWN;
             iss<3>   = '1';
     elsif exception.exceptype == Exception_Uncategorized then
         // Trapped for unknown reason
         iss<8:5> = instr<19:16>;         // Rn
         iss<3>   = '0';

     iss<0> = instr<20>;                  // Direction

     exception.syndrome<24:20> = ConditionSyndrome();
     exception.syndrome<19:0>  = iss;

     return exception;
```

aarch32/exceptions/traps/AArch32.TakeHypTrapException

```
 // AArch32.TakeHypTrapException()
 // ==============================
 // Exceptions routed to Hyp mode as a Hyp Trap exception.

 AArch32.TakeHypTrapException(integer ec)
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    exception = AArch32.SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch32.TakeHypTrapException(exception);

// AArch32.TakeHypTrapException()
// ==============================
// Exceptions routed to Hyp mode as a Hyp Trap exception.

AArch32.TakeHypTrapException(ExceptionRecord exception)
    assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2);

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x14;

    AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
```

aarch32/exceptions/traps/AArch32.TakeMonitorTrapException

```
// AArch32.TakeMonitorTrapException()
// ==================================
// Exceptions routed to Monitor mode as a Monitor Trap exception.

AArch32.TakeMonitorTrapException()
    assert HaveEL(EL3) && ELUsingAArch32(EL3);

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x04;
    lr_offset = if CurrentInstrSet() == InstrSet_A32 then 4 else 2;

    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
```

aarch32/exceptions/traps/AArch32.TakeUndefInstrException

```
// AArch32.TakeUndefInstrException()
// =================================

AArch32.TakeUndefInstrException()
    exception = ExceptionSyndrome(Exception_Uncategorized);
    AArch32.TakeUndefInstrException(exception);

// AArch32.TakeUndefInstrException()
// =================================

AArch32.TakeUndefInstrException(ExceptionRecord exception)

    route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x04;
    lr_offset = if CurrentInstrSet() == InstrSet_A32 then 4 else 2;

    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    elsif route_to_hyp then
        AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.EnterMode(M32_Undef, preferred_exception_return, lr_offset, vect_offset);
```

aarch32/exceptions/traps/AArch32.UndefinedFault

```
// AArch32.UndefinedFault()
// ========================

AArch32.UndefinedFault()

    if AArch32.GeneralExceptionsToAArch64() then AArch64.UndefinedFault();
    AArch32.TakeUndefInstrException();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

J1.2.3 aarch32/functions

This section includes the following pseudocode functions:

- *aarch32/functions/aborts/AArch32.CreateFaultRecord* on page J1-7657.
- *aarch32/functions/aborts/AArch32.DomainValid* on page J1-7657.
- *aarch32/functions/aborts/AArch32.FaultStatusLD* on page J1-7657.
- *aarch32/functions/aborts/AArch32.FaultStatusSD* on page J1-7658.
- *aarch32/functions/aborts/AArch32.FaultSyndrome* on page J1-7658.
- *aarch32/functions/aborts/EncodeSDFSC* on page J1-7658.
- *aarch32/functions/common/A32ExpandImm* on page J1-7659.
- *aarch32/functions/common/A32ExpandImm_C* on page J1-7659.
- *aarch32/functions/common/DecodeImmShift* on page J1-7660.
- *aarch32/functions/common/DecodeRegShift* on page J1-7660.
- *aarch32/functions/common/RRX* on page J1-7660.
- *aarch32/functions/common/RRX_C* on page J1-7660.
- *aarch32/functions/common/SRType* on page J1-7660.
- *aarch32/functions/common/Shift* on page J1-7660.
- *aarch32/functions/common/Shift_C* on page J1-7661.
- *aarch32/functions/common/T32ExpandImm* on page J1-7661.
- *aarch32/functions/common/T32ExpandImm_C* on page J1-7661.
- *aarch32/functions/coproc/AArch32.CheckCP15InstrCoarseTraps* on page J1-7662.
- *aarch32/functions/exclusive/AArch32.ExclusiveMonitorsPass* on page J1-7662.
- *aarch32/functions/exclusive/AArch32.IsExclusiveVA* on page J1-7663.
- *aarch32/functions/exclusive/AArch32.MarkExclusiveVA* on page J1-7663.
- *aarch32/functions/exclusive/AArch32.SetExclusiveMonitors* on page J1-7663.
- *aarch32/functions/float/CheckAdvSIMDEnabled* on page J1-7663.
- *aarch32/functions/float/CheckAdvSIMDOrVFPEnabled* on page J1-7664.
- *aarch32/functions/float/CheckCryptoEnabled32* on page J1-7664.
- *aarch32/functions/float/CheckVFPEnabled* on page J1-7664.
- *aarch32/functions/float/FPHalvedSub* on page J1-7664.
- *aarch32/functions/float/FPRSqrtStep* on page J1-7665.
- *aarch32/functions/float/FPRecipStep* on page J1-7665.
- *aarch32/functions/float/StandardFPSCRValue* on page J1-7665.
- *aarch32/functions/memory/AArch32.CheckAlignment* on page J1-7665.
- *aarch32/functions/memory/AArch32.MemSingle* on page J1-7666.
- *aarch32/functions/memory/Hint_PreloadData* on page J1-7667.
- *aarch32/functions/memory/Hint_PreloadDataForWrite* on page J1-7667.
- *aarch32/functions/memory/Hint_PreloadInstr* on page J1-7667.
- *aarch32/functions/memory/MemA* on page J1-7667.
- *aarch32/functions/memory/MemO* on page J1-7667.
- *aarch32/functions/memory/MemU* on page J1-7667.
- *aarch32/functions/memory/MemU_unpriv* on page J1-7668.
- *aarch32/functions/memory/Mem_with_type* on page J1-7668.
- *aarch32/functions/ras/AArch32.ESBOperation* on page J1-7669.
- *aarch32/functions/ras/AArch32.PhysicalSErrorSyndrome* on page J1-7670.
- *aarch32/functions/ras/AArch32.ReportDeferredSError* on page J1-7670.
- *aarch32/functions/ras/AArch32.SErrorSyndrome* on page J1-7670.
- *aarch32/functions/ras/AArch32.vESBOperation* on page J1-7670.
- *aarch32/functions/registers/AArch32.ResetGeneralRegisters* on page J1-7671.
- *aarch32/functions/registers/AArch32.ResetSIMDFPRegisters* on page J1-7671.
- *aarch32/functions/registers/AArch32.ResetSpecialRegisters* on page J1-7671.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *aarch32/functions/registers/AArch32.ResetSystemRegisters* on page J1-7672.
- *aarch32/functions/registers/ALUExceptionReturn* on page J1-7672.
- *aarch32/functions/registers/ALUWritePC* on page J1-7672.
- *aarch32/functions/registers/BXWritePC* on page J1-7672.
- *aarch32/functions/registers/BranchWritePC* on page J1-7672.
- *aarch32/functions/registers/D* on page J1-7673.
- *aarch32/functions/registers/Din* on page J1-7673.
- *aarch32/functions/registers/LR* on page J1-7673.
- *aarch32/functions/registers/LoadWritePC* on page J1-7673.
- *aarch32/functions/registers/LookUpRIndex* on page J1-7673.
- *aarch32/functions/registers/Monitor_mode_registers* on page J1-7674.
- *aarch32/functions/registers/PC* on page J1-7674.
- *aarch32/functions/registers/PCStoreValue* on page J1-7674.
- *aarch32/functions/registers/Q* on page J1-7674.
- *aarch32/functions/registers/Qin* on page J1-7674.
- *aarch32/functions/registers/R* on page J1-7675.
- *aarch32/functions/registers/RBankSelect* on page J1-7675.
- *aarch32/functions/registers/Rmode* on page J1-7675.
- *aarch32/functions/registers/S* on page J1-7676.
- *aarch32/functions/registers/SP* on page J1-7676.
- *aarch32/functions/registers/_Dclone* on page J1-7676.
- *aarch32/functions/system/AArch32.ExceptionReturn* on page J1-7676.
- *aarch32/functions/system/AArch32.ExecutingATS1xPInstr* on page J1-7677.
- *aarch32/functions/system/AArch32.ExecutingCP10or11Instr* on page J1-7677.
- *aarch32/functions/system/AArch32.ExecutingLSMInstr* on page J1-7677.
- *aarch32/functions/system/AArch32.ITAdvance* on page J1-7678.
- *aarch32/functions/system/AArch32.SysRegRead* on page J1-7678.
- *aarch32/functions/system/AArch32.SysRegRead64* on page J1-7678.
- *aarch32/functions/system/AArch32.SysRegReadCanWriteAPSR* on page J1-7678.
- *aarch32/functions/system/AArch32.SysRegWrite* on page J1-7678.
- *aarch32/functions/system/AArch32.SysRegWrite64* on page J1-7678.
- *aarch32/functions/system/AArch32.WriteMode* on page J1-7679.
- *aarch32/functions/system/AArch32.WriteModeByInstr* on page J1-7679.
- *aarch32/functions/system/BadMode* on page J1-7679.
- *aarch32/functions/system/BankedRegisterAccessValid* on page J1-7680.
- *aarch32/functions/system/CPSRWriteByInstr* on page J1-7680.
- *aarch32/functions/system/ConditionPassed* on page J1-7681.
- *aarch32/functions/system/CurrentCond* on page J1-7681.
- *aarch32/functions/system/InITBlock* on page J1-7681.
- *aarch32/functions/system/LastInITBlock* on page J1-7681.
- *aarch32/functions/system/SPSRWriteByInstr* on page J1-7681.
- *aarch32/functions/system/SPSRaccessValid* on page J1-7682.
- *aarch32/functions/system/SelectInstrSet* on page J1-7682.
- *aarch32/functions/v6simd/Sat* on page J1-7682.
- *aarch32/functions/v6simd/SignedSat* on page J1-7682.
- *aarch32/functions/v6simd/UnsignedSat* on page J1-7683.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch32/functions/aborts/AArch32.CreateFaultRecord

```
// AArch32.CreateFaultRecord()
// ===========================

FaultRecord AArch32.CreateFaultRecord(Fault statuscode, bits(40) ipaddress,   bits(4) domain,
                                      integer level, AccType acctype, boolean write, bit extflag,
                                      bits(4) debugmoe, bits(2) errortype, boolean secondstage, boolean
s2fs1walk)

    FaultRecord fault;
    fault.statuscode = statuscode;
    if (statuscode != Fault_None && PSTATE.EL != EL2 && TTBCR.EAE == '0' && !secondstage && !s2fs1walk
&&
        AArch32.DomainValid(statuscode, level)) then
        fault.domain = domain;
    else
        fault.domain = bits(4) UNKNOWN;
    fault.debugmoe = debugmoe;
    fault.errortype = errortype;
    fault.ipaddress.NS = bit UNKNOWN;
    fault.ipaddress.address = ZeroExtend(ipaddress);
    fault.level = level;
    fault.acctype = acctype;
    fault.write = write;
    fault.extflag = extflag;
    fault.secondstage = secondstage;
    fault.s2fs1walk = s2fs1walk;

    return fault;
```

aarch32/functions/aborts/AArch32.DomainValid

```
// AArch32.DomainValid()
// =====================
// Returns TRUE if the Domain is valid for a Short-descriptor translation scheme.

boolean AArch32.DomainValid(Fault statuscode, integer level)
    assert statuscode != Fault_None;

    case statuscode of
        when Fault_Domain
            return TRUE;
        when Fault_Translation, Fault_AccessFlag, Fault_SyncExternalOnWalk, Fault_SyncParityOnWalk
            return level == 2;
        otherwise
            return FALSE;
```

aarch32/functions/aborts/AArch32.FaultStatusLD

```
// AArch32.FaultStatusLD()
// =======================
// Creates an exception fault status value for Abort and Watchpoint exceptions taken
// to Abort mode using AArch32 and Long-descriptor format.

bits(32) AArch32.FaultStatusLD(boolean d_side, FaultRecord fault)
    assert fault.statuscode != Fault_None;

    bits(32) fsr = Zeros();
    if HaveRASExt() && IsAsyncAbort(fault) then fsr<15:14> = fault.errortype;
    if d_side then
        if fault.acctype IN {AccType_DC, AccType_IC, AccType_AT} then
            fsr<13> = '1'; fsr<11> = '1';
        else
            fsr<11> = if fault.write then '1' else '0';
    if IsExternalAbort(fault) then fsr<12> = fault.extflag;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    fsr<9> = '1';
    fsr<5:0> = EncodeLDFSC(fault.statuscode, fault.level);

    return fsr;
```

aarch32/functions/aborts/AArch32.FaultStatusSD

```
// AArch32.FaultStatusSD()
// =======================
// Creates an exception fault status value for Abort and Watchpoint exceptions taken
// to Abort mode using AArch32 and Short-descriptor format.

bits(32) AArch32.FaultStatusSD(boolean d_side, FaultRecord fault)
    assert fault.statuscode != Fault_None;

    bits(32) fsr = Zeros();
    if HaveRASExt() && IsAsyncAbort(fault) then fsr<15:14> = fault.errortype;
    if d_side then
        if fault.acctype IN {AccType_DC, AccType_IC, AccType_AT} then
            fsr<13> = '1'; fsr<11> = '1';
        else
            fsr<11> = if fault.write then '1' else '0';
    if IsExternalAbort(fault) then fsr<12> = fault.extflag;
    fsr<9> = '0';
    fsr<10,3:0> = EncodeSDFSC(fault.statuscode, fault.level);
    if d_side then
        fsr<7:4> = fault.domain;               // Domain field (data fault only)

    return fsr;
```

aarch32/functions/aborts/AArch32.FaultSyndrome

```
// AArch32.FaultSyndrome()
// =======================
// Creates an exception syndrome value for Abort and Watchpoint exceptions taken to
// AArch32 Hyp mode.

bits(25) AArch32.FaultSyndrome(boolean d_side, FaultRecord fault)
    assert fault.statuscode != Fault_None;

    bits(25) iss = Zeros();
    if HaveRASExt() && IsAsyncAbort(fault) then iss<11:10> = fault.errortype; // AET
    if d_side then
        if IsSecondStage(fault) && !fault.s2fs1walk then iss<24:14> = LSInstructionSyndrome();
        if fault.acctype IN {AccType_DC, AccType_DC_UNPRIV, AccType_IC, AccType_AT} then
            iss<8> = '1';  iss<6> = '1';
        else
            iss<6> = if fault.write then '1' else '0';
    if IsExternalAbort(fault) then iss<9> = fault.extflag;
    iss<7> = if fault.s2fs1walk then '1' else '0';
    iss<5:0> = EncodeLDFSC(fault.statuscode, fault.level);

    return iss;
```

aarch32/functions/aborts/EncodeSDFSC

```
// EncodeSDFSC()
// =============
// Function that gives the Short-descriptor FSR code for different types of Fault

bits(5) EncodeSDFSC(Fault statuscode, integer level)

    bits(5) result;
    case statuscode of
        when Fault_AccessFlag
            assert level IN {1,2};
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            result = if level == 1 then '00011' else '00110';
        when Fault_Alignment
            result = '00001';
        when Fault_Permission
            assert level IN {1,2};
            result = if level == 1 then '01101' else '01111';
        when Fault_Domain
            assert level IN {1,2};
            result = if level == 1 then '01001' else '01011';
        when Fault_Translation
            assert level IN {1,2};
            result = if level == 1 then '00101' else '00111';
        when Fault_SyncExternal
            result = '01000';
        when Fault_SyncExternalOnWalk
            assert level IN {1,2};
            result = if level == 1 then '01100' else '01110';
        when Fault_SyncParity
            result = '11001';
        when Fault_SyncParityOnWalk
            assert level IN {1,2};
            result = if level == 1 then '11100' else '11110';
        when Fault_AsyncParity
            result = '11000';
        when Fault_AsyncExternal
            result = '10110';
        when Fault_Debug
            result = '00010';
        when Fault_TLBConflict
            result = '10000';
        when Fault_Lockdown
            result = '10100';   // IMPLEMENTATION DEFINED
        when Fault_Exclusive
            result = '10101';   // IMPLEMENTATION DEFINED
        when Fault_ICacheMaint
            result = '00100';
        otherwise
            Unreachable();

    return result;
```

aarch32/functions/common/A32ExpandImm

```
// A32ExpandImm()
// ==============

bits(32) A32ExpandImm(bits(12) imm12)

    // PSTATE.C argument to following function call does not affect the imm32 result.
    (imm32, -) = A32ExpandImm_C(imm12, PSTATE.C);

    return imm32;
```

aarch32/functions/common/A32ExpandImm_C

```
// A32ExpandImm_C()
// ================

(bits(32), bit) A32ExpandImm_C(bits(12) imm12, bit carry_in)

    unrotated_value = ZeroExtend(imm12<7:0>, 32);
    (imm32, carry_out) = Shift_C(unrotated_value, SRType_ROR, 2*UInt(imm12<11:8>), carry_in);

    return (imm32, carry_out);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch32/functions/common/DecodeImmShift

```
// DecodeImmShift()
// ================

(SRType, integer) DecodeImmShift(bits(2) srtype, bits(5) imm5)

    case srtype of
        when '00'
            shift_t = SRType_LSL;  shift_n = UInt(imm5);
        when '01'
            shift_t = SRType_LSR;  shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '10'
            shift_t = SRType_ASR;  shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '11'
            if imm5 == '00000' then
                shift_t = SRType_RRX;  shift_n = 1;
            else
                shift_t = SRType_ROR;  shift_n = UInt(imm5);

    return (shift_t, shift_n);
```

aarch32/functions/common/DecodeRegShift

```
// DecodeRegShift()
// ================

SRType DecodeRegShift(bits(2) srtype)
    case srtype of
        when '00'  shift_t = SRType_LSL;
        when '01'  shift_t = SRType_LSR;
        when '10'  shift_t = SRType_ASR;
        when '11'  shift_t = SRType_ROR;
    return shift_t;
```

aarch32/functions/common/RRX

```
// RRX()
// =====

bits(N) RRX(bits(N) x, bit carry_in)
    (result, -) = RRX_C(x, carry_in);
    return result;
```

aarch32/functions/common/RRX_C

```
// RRX_C()
// =======

(bits(N), bit) RRX_C(bits(N) x, bit carry_in)
    result = carry_in : x<N-1:1>;
    carry_out = x<0>;
    return (result, carry_out);
```

aarch32/functions/common/SRType

```
enumeration SRType {SRType_LSL, SRType_LSR, SRType_ASR, SRType_ROR, SRType_RRX};
```

aarch32/functions/common/Shift

```
// Shift()
// =======
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(N) Shift(bits(N) value, SRType srtype, integer amount, bit carry_in)
    (result, -) = Shift_C(value, srtype, amount, carry_in);
    return result;
```

aarch32/functions/common/Shift_C

```
// Shift_C()
// =========

(bits(N), bit) Shift_C(bits(N) value, SRType srtype, integer amount, bit carry_in)
    assert !(srtype == SRType_RRX && amount != 1);

    if amount == 0 then
        (result, carry_out) = (value, carry_in);
    else
        case srtype of
            when SRType_LSL
                (result, carry_out) = LSL_C(value, amount);
            when SRType_LSR
                (result, carry_out) = LSR_C(value, amount);
            when SRType_ASR
                (result, carry_out) = ASR_C(value, amount);
            when SRType_ROR
                (result, carry_out) = ROR_C(value, amount);
            when SRType_RRX
                (result, carry_out) = RRX_C(value, carry_in);

    return (result, carry_out);
```

aarch32/functions/common/T32ExpandImm

```
// T32ExpandImm()
// ==============

bits(32) T32ExpandImm(bits(12) imm12)

    // PSTATE.C argument to following function call does not affect the imm32 result.
    (imm32, -) = T32ExpandImm_C(imm12, PSTATE.C);

    return imm32;
```

aarch32/functions/common/T32ExpandImm_C

```
// T32ExpandImm_C()
// ================

(bits(32), bit) T32ExpandImm_C(bits(12) imm12, bit carry_in)

    if imm12<11:10> == '00' then
        case imm12<9:8> of
            when '00'
                imm32 = ZeroExtend(imm12<7:0>, 32);
            when '01'
                imm32 = '00000000' : imm12<7:0> : '00000000' : imm12<7:0>;
            when '10'
                imm32 = imm12<7:0> : '00000000' : imm12<7:0> : '00000000';
            when '11'
                imm32 = imm12<7:0> : imm12<7:0> : imm12<7:0> : imm12<7:0>;
        carry_out = carry_in;
    else
        unrotated_value = ZeroExtend('1':imm12<6:0>, 32);
        (imm32, carry_out) = ROR_C(unrotated_value, UInt(imm12<11:7>));

    return (imm32, carry_out);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch32/functions/coproc/AArch32.CheckCP15InstrCoarseTraps

```
// AArch32.CheckCP15InstrCoarseTraps()
// ===================================
// Check for coarse-grained CP15 traps in HSTR and HCR.

boolean AArch32.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)

    // Check for coarse-grained Hyp traps
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        if PSTATE.EL == EL0 && !ELUsingAArch32(EL2) then
            return AArch64.CheckCP15InstrCoarseTraps(CRn, nreg, CRm);
        // Check for MCR, MRC, MCRR and MRRC disabled by HSTR<CRn/CRm>
        major = if nreg == 1 then CRn else CRm;
        if !(major IN {4,14}) && HSTR<major> == '1' then
            return TRUE;

        // Check for MRC and MCR disabled by HCR.TIDCP
        if (HCR.TIDCP == '1' && nreg == 1 &&
            ((CRn == 9  && CRm IN {0,1,2,    5,6,7,8   }) ||
             (CRn == 10 && CRm IN {0,1,    4,      8   }) ||
             (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}))) then
            return TRUE;

    return FALSE;
```

aarch32/functions/exclusive/AArch32.ExclusiveMonitorsPass

```
// AArch32.ExclusiveMonitorsPass()
// ===============================

// Return TRUE if the Exclusives monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

boolean AArch32.ExclusiveMonitorsPass(bits(32) address, integer size)

    // It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
    // before or after the check on the local Exclusives monitor. As a result a failure
    // of the local monitor can occur on some implementations even if the memory
    // access would give an memory abort.

    acctype = AccType_ATOMIC;
    iswrite = TRUE;

    aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);

    passed = AArch32.IsExclusiveVA(address, ProcessorID(), size);
    if !passed then
        return FALSE;

    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
    ClearExclusiveLocal(ProcessorID());

    if passed then
        if memaddrdesc.memattrs.shareable then
            passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    return passed;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch32/functions/exclusive/AArch32.IsExclusiveVA

```
// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual
// address region of size bytes starting at address.
//
// It is permitted (but not required) for this function to return FALSE and
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
//
// It is always safe to return TRUE which will check the physical address only.
boolean AArch32.IsExclusiveVA(bits(32) address, integer processorid, integer size);
```

aarch32/functions/exclusive/AArch32.MarkExclusiveVA

```
// Optionally record an exclusive access to the virtual address region of size bytes
// starting at address for processorid.
AArch32.MarkExclusiveVA(bits(32) address, integer processorid, integer size);
```

aarch32/functions/exclusive/AArch32.SetExclusiveMonitors

```
// AArch32.SetExclusiveMonitors()
// ==============================

// Sets the Exclusives monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.

AArch32.SetExclusiveMonitors(bits(32) address, integer size)

    acctype = AccType_ATOMIC;
    iswrite = FALSE;
    aligned = (address == Align(address, size));

    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;

    if memaddrdesc.memattrs.shareable then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

    AArch32.MarkExclusiveVA(address, ProcessorID(), size);
```

aarch32/functions/float/CheckAdvSIMDEnabled

```
// CheckAdvSIMDEnabled()
// =====================

CheckAdvSIMDEnabled()

    fpexc_check = TRUE;
    advsimd = TRUE;

    AArch32.CheckAdvSIMDOrFPEnabled(fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEnabled() occurs only if Advanced SIMD access is permitted

    // Make temporary copy of D registers
    // _Dclone[] is used as input data for instruction pseudocode
    for i = 0 to 31
        _Dclone[i] = D[i];

    return;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch32/functions/float/CheckAdvSIMDOrVFPEnabled

```
// CheckAdvSIMDOrVFPEnabled()
// ==========================

CheckAdvSIMDOrVFPEnabled(boolean include_fpexc_check, boolean advsimd)
    AArch32.CheckAdvSIMDOrFPEnabled(include_fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEnabled() occurs only if VFP access is permitted
    return;
```

aarch32/functions/float/CheckCryptoEnabled32

```
// CheckCryptoEnabled32()
// ======================

CheckCryptoEnabled32()
    CheckAdvSIMDEnabled();
    // Return from CheckAdvSIMDEnabled() occurs only if access is permitted
    return;
```

aarch32/functions/float/CheckVFPEnabled

```
// CheckVFPEnabled()
// =================

CheckVFPEnabled(boolean include_fpexc_check)
    advsimd = FALSE;
    AArch32.CheckAdvSIMDOrFPEnabled(include_fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEnabled() occurs only if VFP access is permitted
    return;
```

aarch32/functions/float/FPHalvedSub

```
// FPHalvedSub()
// =============

bits(N) FPHalvedSub(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1');
        elsif zero1 && zero2 && sign1 != sign2 then
            result = FPZero(sign1);
        else
            result_value = (value1 - value2) / 2.0;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr);
    return result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch32/functions/float/FPRSqrtStep

```
// FPRSqrtStep()
// =============

bits(N) FPRSqrtStep(bits(N) op1, bits(N) op2)
    assert N IN {16,32};
    FPCRType fpcr = StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        bits(N) product;
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero('0');
        else
            product = FPMul(op1, op2, fpcr);
        bits(N) three = FPThree('0');
        result = FPHalvedSub(three, product, fpcr);
    return result;
```

aarch32/functions/float/FPRecipStep

```
// FPRecipStep()
// =============

bits(N) FPRecipStep(bits(N) op1, bits(N) op2)
    assert N IN {16,32};
    FPCRType fpcr = StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        bits(N) product;
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero('0');
        else
            product = FPMul(op1, op2, fpcr);
        bits(N) two = FPTwo('0');
        result = FPSub(two, product, fpcr);
    return result;
```

aarch32/functions/float/StandardFPSCRValue

```
// StandardFPSCRValue()
// ====================

FPCRType StandardFPSCRValue()
    return '00000' : FPSCR.AHP : '110000' : FPSCR.FZ16 : '0000000000000000000';
```

aarch32/functions/memory/AArch32.CheckAlignment

```
// AArch32.CheckAlignment()
// ========================

boolean AArch32.CheckAlignment(bits(32) address, integer alignment, AccType acctype,
                               boolean iswrite)

    if PSTATE.EL == EL0 && !ELUsingAArch32(S1TranslationRegime()) then
        A = SCTLR[].A; //use AArch64 register, when higher Exception level is using AArch64
    elsif PSTATE.EL == EL2 then
        A = HSCTLR.A;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        A = SCTLR.A;
    aligned = (address == Align(address, alignment));
    atomic  = acctype IN { AccType_ATOMIC, AccType_ATOMICRW, AccType_ORDEREDATOMIC,
AccType_ORDEREDATOMICRW };
    ordered = acctype IN { AccType_ORDERED, AccType_ORDEREDRW, AccType_LIMITEDORDERED,
AccType_ORDEREDATOMIC, AccType_ORDEREDATOMICRW };
    vector  = acctype == AccType_VEC;

    // AccType_VEC is used for SIMD element alignment checks only
    check = (atomic || ordered || vector || A == '1');

    if check && !aligned then
        secondstage = FALSE;
        AArch32.Abort(address, AArch32.AlignmentFault(acctype, iswrite, secondstage));

    return aligned;
```

aarch32/functions/memory/AArch32.MemSingle

```
// AArch32.MemSingle[] - non-assignment (read) form
// ================================================
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean wasaligned]
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    bits(size*8) value;
    iswrite = FALSE;

    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, wasaligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);
    if HaveMTEExt() then
        if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
            bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
            if !AArch64.CheckTag(memaddrdesc, ptag, iswrite) then
                AArch64.TagCheckFail(ZeroExtend(address, 64), iswrite);
    value = _Mem[memaddrdesc, size, accdesc];
    return value;

// AArch32.MemSingle[] - assignment (write) form
// =============================================
// Perform an atomic, little-endian write of 'size' bytes.

AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean wasaligned] = bits(size*8)
value
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    iswrite = TRUE;

    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, wasaligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareable then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);
    if HaveMTEExt() then
        if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
            bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
            if !AArch64.CheckTag(memaddrdesc, ptag, iswrite) then
                AArch64.TagCheckFail(ZeroExtend(address, 64), iswrite);
    _Mem[memaddrdesc, size, accdesc] = value;
    return;
```

aarch32/functions/memory/Hint_PreloadData

```
Hint_PreloadData(bits(32) address);
```

aarch32/functions/memory/Hint_PreloadDataForWrite

```
Hint_PreloadDataForWrite(bits(32) address);
```

aarch32/functions/memory/Hint_PreloadInstr

```
Hint_PreloadInstr(bits(32) address);
```

aarch32/functions/memory/MemA

```
// MemA[] - non-assignment form
// ============================

bits(8*size) MemA[bits(32) address, integer size]
    acctype = AccType_ATOMIC;
    return Mem_with_type[address, size, acctype];

// MemA[] - assignment form
// ========================

MemA[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType_ATOMIC;
    Mem_with_type[address, size, acctype] = value;
    return;
```

aarch32/functions/memory/MemO

```
// MemO[] - non-assignment form
// ============================

bits(8*size) MemO[bits(32) address, integer size]
    acctype = AccType_ORDERED;
    return Mem_with_type[address, size, acctype];

// MemO[] - assignment form
// ========================

MemO[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType_ORDERED;
    Mem_with_type[address, size, acctype] = value;
    return;
```

aarch32/functions/memory/MemU

```
// MemU[] - non-assignment form
// ============================

bits(8*size) MemU[bits(32) address, integer size]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    acctype = AccType_NORMAL;
    return Mem_with_type[address, size, acctype];

// MemU[] - assignment form
// ========================

MemU[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType_NORMAL;
    Mem_with_type[address, size, acctype] = value;
    return;
```

aarch32/functions/memory/MemU_unpriv

```
// MemU_unpriv[] - non-assignment form
// ===================================

bits(8*size) MemU_unpriv[bits(32) address, integer size]
    acctype = AccType_UNPRIV;
    return Mem_with_type[address, size, acctype];

// MemU_unpriv[] - assignment form
// ===============================

MemU_unpriv[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType_UNPRIV;
    Mem_with_type[address, size, acctype] = value;
    return;
```

aarch32/functions/memory/Mem_with_type

```
// Mem_with_type[] - non-assignment (read) form
// ============================================
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch32.MemSingle directly.

bits(size*8) Mem_with_type[bits(32) address, integer size, AccType acctype]
    assert size IN {1, 2, 4, 8, 16};
    bits(size*8) value;
    boolean iswrite = FALSE;

    aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);
    if !aligned then
        assert size > 1;
        value<7:0> = AArch32.MemSingle[address, 1, acctype, aligned];

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        c = ConstrainUnpredictable();
        assert c IN {Constraint_FAULT, Constraint_NONE};
        if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            value<8*i+7:8*i> = AArch32.MemSingle[address+i, 1, acctype, aligned];
    else
        value = AArch32.MemSingle[address, size, acctype, aligned];

    if BigEndian() then
        value = BigEndianReverse(value);
    return value;

// Mem_with_type[] - assignment (write) form
// =========================================
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem_with_type[bits(32) address, integer size, AccType acctype] = bits(size*8) value
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    boolean iswrite = TRUE;

    if BigEndian() then
        value = BigEndianReverse(value);

    aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);

    if !aligned then
        assert size > 1;
        AArch32.MemSingle[address, 1, acctype, aligned] = value<7:0>;

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        c = ConstrainUnpredictable();
        assert c IN {Constraint_FAULT, Constraint_NONE};
        if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            AArch32.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>;
    else
        AArch32.MemSingle[address, size, acctype, aligned] = value;
    return;
```

aarch32/functions/ras/AArch32.ESBOperation

```
// AArch32.ESBOperation()
// ======================
// Perform the AArch32 ESB operation for ESB executed in AArch32 state

AArch32.ESBOperation()

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = HCR_EL2.TGE == '1' || HCR_EL2.AMO == '1';
    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.EA == '1';

    if route_to_aarch64 then
        AArch64.ESBOperation();
        return;

    route_to_monitor = HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.EA == '1';
    route_to_hyp = PSTATE.EL IN {EL0, EL1} && EL2Enabled() && (HCR.TGE == '1' || HCR.AMO == '1');

    if route_to_monitor then
        target = M32_Monitor;
    elsif route_to_hyp || PSTATE.M == M32_Hyp then
        target = M32_Hyp;
    else
        target = M32_Abort;

    if IsSecure() then
        mask_active = TRUE;
    elsif target == M32_Monitor then
        mask_active = SCR.AW == '1' && (!HaveEL(EL2) || (HCR.TGE == '0' && HCR.AMO == '0'));
    else
        mask_active = target == M32_Abort || PSTATE.M == M32_Hyp;

    mask_set = PSTATE.A == '1';
    (-, el)  = ELFromM32(target);
    intdis   = Halted() || ExternalDebugInterruptsDisabled(el);
    masked   = intdis || (mask_active && mask_set);

    // Check for a masked Physical SError pending
    if IsPhysicalSErrorPending() && masked then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        syndrome32 = AArch32.PhysicalSErrorSyndrome();
        DISR = AArch32.ReportDeferredSError(syndrome32.AET, syndrome32.ExT);
        ClearPendingPhysicalSError();

    return;
```

aarch32/functions/ras/AArch32.PhysicalSErrorSyndrome

```
// Return the SError syndrome
AArch32.SErrorSyndrome AArch32.PhysicalSErrorSyndrome();
```

aarch32/functions/ras/AArch32.ReportDeferredSError

```
// AArch32.ReportDeferredSError()
// ==============================
// Return deferred SError syndrome

bits(32) AArch32.ReportDeferredSError(bits(2) AET, bit ExT)
    bits(32) target;
    target<31> = '1';                       // A
    syndrome = Zeros(16);
    if PSTATE.EL == EL2 then
        syndrome<11:10> = AET;              // AET
        syndrome<9>     = ExT;              // EA
        syndrome<5:0>   = '010001';         // DFSC
    else
        syndrome<15:14> = AET;              // AET
        syndrome<12>    = ExT;              // ExT
        syndrome<9>     = TTBCR.EAE;        // LPAE
        if TTBCR.EAE == '1' then            // Long-descriptor format
            syndrome<5:0>    = '010001';    // STATUS
        else                                // Short-descriptor format
            syndrome<10,3:0> = '10110';     // FS
    if HaveAnyAArch64() then
        target<24:0> = ZeroExtend(syndrome);// Any RES0 fields must be set to zero
    else
        target<15:0> = syndrome;
    return target;
```

aarch32/functions/ras/AArch32.SErrorSyndrome

```
type AArch32.SErrorSyndrome is (
    bits(2) AET,
    bit ExT
)
```

aarch32/functions/ras/AArch32.vESBOperation

```
// AArch32.vESBOperation()
// =======================
// Perform the ESB operation for virtual SError interrupts executed in AArch32 state

AArch32.vESBOperation()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();

    // Check for EL2 using AArch64 state
    if !ELUsingAArch32(EL2) then
        AArch64.vESBOperation();
        return;

    // If physical SError interrupts are routed to Hyp mode, and TGE is not set, then a
    // virtual SError interrupt might be pending
    vSEI_enabled = HCR.TGE == '0' && HCR.AMO == '1';
    vSEI_pending = vSEI_enabled && HCR.VA == '1';
    vintdis      = Halted() || ExternalDebugInterruptsDisabled(EL1);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    vmasked      = vintdis || PSTATE.A == '1';

    // Check for a masked virtual SError pending
    if vSEI_pending && vmasked then
        VDISR = AArch32.ReportDeferredSError(VDFSR<15:14>, VDFSR<12>);
        HCR.VA = '0';                       // Clear pending virtual SError

    return;
```

aarch32/functions/registers/AArch32.ResetGeneralRegisters

```
// AArch32.ResetGeneralRegisters()
// ==============================

AArch32.ResetGeneralRegisters()

    for i = 0 to 7
        R[i] = bits(32) UNKNOWN;
    for i = 8 to 12
        Rmode[i, M32_User] = bits(32) UNKNOWN;
        Rmode[i, M32_FIQ] = bits(32) UNKNOWN;
    if HaveEL(EL2) then Rmode[13, M32_Hyp] = bits(32) UNKNOWN;   // No R14_hyp
    for i = 13 to 14
        Rmode[i, M32_User] = bits(32) UNKNOWN;
        Rmode[i, M32_FIQ] = bits(32) UNKNOWN;
        Rmode[i, M32_IRQ] = bits(32) UNKNOWN;
        Rmode[i, M32_Svc] = bits(32) UNKNOWN;
        Rmode[i, M32_Abort] = bits(32) UNKNOWN;
        Rmode[i, M32_Undef] = bits(32) UNKNOWN;
        if HaveEL(EL3) then Rmode[i, M32_Monitor] = bits(32) UNKNOWN;

    return;
```

aarch32/functions/registers/AArch32.ResetSIMDFPRegisters

```
// AArch32.ResetSIMDFPRegisters()
// ==============================

AArch32.ResetSIMDFPRegisters()

    for i = 0 to 15
        Q[i] = bits(128) UNKNOWN;

    return;
```

aarch32/functions/registers/AArch32.ResetSpecialRegisters

```
// AArch32.ResetSpecialRegisters()
// ==============================

AArch32.ResetSpecialRegisters()

    // AArch32 special registers
    SPSR_fiq = bits(32) UNKNOWN;
    SPSR_irq = bits(32) UNKNOWN;
    SPSR_svc = bits(32) UNKNOWN;
    SPSR_abt = bits(32) UNKNOWN;
    SPSR_und = bits(32) UNKNOWN;
    if HaveEL(EL2) then
        SPSR_hyp = bits(32) UNKNOWN;
        ELR_hyp = bits(32) UNKNOWN;
    if HaveEL(EL3) then
        SPSR_mon = bits(32) UNKNOWN;

    // External debug special registers
    DLR = bits(32) UNKNOWN;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    DSPSR = bits(32) UNKNOWN;

    return;
```

aarch32/functions/registers/AArch32.ResetSystemRegisters

```
AArch32.ResetSystemRegisters(boolean cold_reset);
```

aarch32/functions/registers/ALUExceptionReturn

```
// ALUExceptionReturn()
// ====================

ALUExceptionReturn(bits(32) address)
    if PSTATE.EL == EL2 then
        UNDEFINED;
    elsif PSTATE.M IN {M32_User,M32_System} then
        UNPREDICTABLE;          // UNDEFINED or NOP
    else
        AArch32.ExceptionReturn(address, SPSR[]);
```

aarch32/functions/registers/ALUWritePC

```
// ALUWritePC()
// ============

ALUWritePC(bits(32) address)
    if CurrentInstrSet() == InstrSet_A32 then
        BXWritePC(address, BranchType_INDIR);
    else
        BranchWritePC(address, BranchType_INDIR);
```

aarch32/functions/registers/BXWritePC

```
// BXWritePC()
// ===========

BXWritePC(bits(32) address, BranchType branch_type)
    if address<0> == '1' then
        SelectInstrSet(InstrSet_T32);
        address<0> = '0';
    else
        SelectInstrSet(InstrSet_A32);
        // For branches to an unaligned PC counter in A32 state, the processor takes the branch
        // and does one of:
        // * Forces the address to be aligned
        // * Leaves the PC unaligned, meaning the target generates a PC Alignment fault.
        if address<1> == '1' && ConstrainUnpredictableBool() then
            address<1> = '0';
    BranchTo(address, branch_type);
```

aarch32/functions/registers/BranchWritePC

```
// BranchWritePC()
// ===============

BranchWritePC(bits(32) address, BranchType branch_type)
    if CurrentInstrSet() == InstrSet_A32 then
        address<1:0> = '00';
    else
        address<0> = '0';
    BranchTo(address, branch_type);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch32/functions/registers/D

```
// D[] - non-assignment form
// =========================

bits(64) D[integer n]
    assert n >= 0 && n <= 31;
    base = (n MOD 2) * 64;
    bits(128) vreg = V[n DIV 2];
    return vreg<base+63:base>;

// D[] - assignment form
// =====================

D[integer n] = bits(64) value
    assert n >= 0 && n <= 31;
    base = (n MOD 2) * 64;
    bits(128) vreg = V[n DIV 2];
    vreg<base+63:base> = value;
    V[n DIV 2] = vreg;
    return;
```

aarch32/functions/registers/Din

```
// Din[] - non-assignment form
// ===========================

bits(64) Din[integer n]
    assert n >= 0 && n <= 31;
    return _Dclone[n];
```

aarch32/functions/registers/LR

```
// LR - assignment form
// ====================

LR = bits(32) value
    R[14] = value;
    return;

// LR - non-assignment form
// ========================

bits(32) LR
    return R[14];
```

aarch32/functions/registers/LoadWritePC

```
// LoadWritePC()
// =============

LoadWritePC(bits(32) address)
    BXWritePC(address, BranchType_INDIR);
```

aarch32/functions/registers/LookUpRIndex

```
// LookUpRIndex()
// ==============

integer LookUpRIndex(integer n, bits(5) mode)
    assert n >= 0 && n <= 14;

    case n of  // Select  index by mode:     usr fiq irq svc abt und hyp
        when 8     result = RBankSelect(mode,  8, 24,  8,  8,  8,  8,  8);
        when 9     result = RBankSelect(mode,  9, 25,  9,  9,  9,  9,  9);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        when 10    result = RBankSelect(mode, 10, 26, 10, 10, 10, 10, 10);
        when 11    result = RBankSelect(mode, 11, 27, 11, 11, 11, 11, 11);
        when 12    result = RBankSelect(mode, 12, 28, 12, 12, 12, 12, 12);
        when 13    result = RBankSelect(mode, 13, 29, 17, 19, 21, 23, 15);
        when 14    result = RBankSelect(mode, 14, 30, 16, 18, 20, 22, 14);
        otherwise  result = n;

    return result;
```

aarch32/functions/registers/Monitor_mode_registers

```
bits(32) SP_mon;
bits(32) LR_mon;
```

aarch32/functions/registers/PC

```
// PC - non-assignment form
// ========================

bits(32) PC
    return R[15];               // This includes the offset from AArch32 state
```

aarch32/functions/registers/PCStoreValue

```
// PCStoreValue()
// ==============

bits(32) PCStoreValue()
    // This function returns the PC value. On architecture versions before Armv7, it
    // is permitted to instead return PC+4, provided it does so consistently. It is
    // used only to describe A32 instructions, so it returns the address of the current
    // instruction plus 8 (normally) or 12 (when the alternative is permitted).
    return PC;
```

aarch32/functions/registers/Q

```
// Q[] - non-assignment form
// =========================

bits(128) Q[integer n]
    assert n >= 0 && n <= 15;
    return V[n];

// Q[] - assignment form
// =====================

Q[integer n] = bits(128) value
    assert n >= 0 && n <= 15;
    V[n] = value;
    return;
```

aarch32/functions/registers/Qin

```
// Qin[] - non-assignment form
// ===========================

bits(128) Qin[integer n]
    assert n >= 0 && n <= 15;
    return Din[2*n+1]:Din[2*n];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch32/functions/registers/R

```
// R[] - assignment form
// ====================

R[integer n] = bits(32) value
    Rmode[n, PSTATE.M] = value;
    return;

// R[] - non-assignment form
// =========================

bits(32) R[integer n]
    if n == 15 then
        offset = (if CurrentInstrSet() == InstrSet_A32 then 8 else 4);
        return _PC<31:0> + offset;
    else
        return Rmode[n, PSTATE.M];
```

aarch32/functions/registers/RBankSelect

```
// RBankSelect()
// =============

integer RBankSelect(bits(5) mode, integer usr, integer fiq, integer irq,
                    integer svc, integer abt, integer und, integer hyp)

    case mode of
        when M32_User    result = usr;  // User mode
        when M32_FIQ     result = fiq;  // FIQ mode
        when M32_IRQ     result = irq;  // IRQ mode
        when M32_Svc     result = svc;  // Supervisor mode
        when M32_Abort   result = abt;  // Abort mode
        when M32_Hyp     result = hyp;  // Hyp mode
        when M32_Undef   result = und;  // Undefined mode
        when M32_System  result = usr;  // System mode uses User mode registers
        otherwise        Unreachable(); // Monitor mode

    return result;
```

aarch32/functions/registers/Rmode

```
// Rmode[] - non-assignment form
// =============================

bits(32) Rmode[integer n, bits(5) mode]
    assert n >= 0 && n <= 14;

    // Check for attempted use of Monitor mode in Non-secure state.
    if !IsSecure() then assert mode != M32_Monitor;
    assert !BadMode(mode);

    if mode == M32_Monitor then
        if n == 13 then return SP_mon;
        elsif n == 14 then return LR_mon;
        else return _R[n]<31:0>;
    else
        return _R[LookUpRIndex(n, mode)]<31:0>;

// Rmode[] - assignment form
// =========================

Rmode[integer n, bits(5) mode] = bits(32) value
    assert n >= 0 && n <= 14;

    // Check for attempted use of Monitor mode in Non-secure state.
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if !IsSecure() then assert mode != M32_Monitor;
    assert !BadMode(mode);

    if mode == M32_Monitor then
        if n == 13 then SP_mon = value;
        elsif n == 14 then LR_mon = value;
        else _R[n]<31:0> = value;
    else
        // It is CONSTRAINED UNPREDICTABLE whether the upper 32 bits of the X
        // register are unchanged or set to zero. This is also tested for on
        // exception entry, as this applies to all AArch32 registers.
        if !HighestELUsingAArch32() && ConstrainUnpredictableBool() then
            _R[LookUpRIndex(n, mode)] = ZeroExtend(value);
        else
            _R[LookUpRIndex(n, mode)]<31:0> = value;

    return;
```

aarch32/functions/registers/S

```
// S[] - non-assignment form
// =========================

bits(32) S[integer n]
    assert n >= 0 && n <= 31;
    base = (n MOD 4) * 32;
    bits(128) vreg = V[n DIV 4];
    return vreg<base+31:base>;

// S[] - assignment form
// =====================

S[integer n] = bits(32) value
    assert n >= 0 && n <= 31;
    base = (n MOD 4) * 32;
    bits(128) vreg = V[n DIV 4];
    vreg<base+31:base> = value;
    V[n DIV 4] = vreg;
    return;
```

aarch32/functions/registers/SP

```
// SP - assignment form
// ====================

SP = bits(32) value
    R[13] = value;
    return;

// SP - non-assignment form
// ========================

bits(32) SP
    return R[13];
```

aarch32/functions/registers/_Dclone

```
array bits(64) _Dclone[0..31];
```

aarch32/functions/system/AArch32.ExceptionReturn

```
// AArch32.ExceptionReturn()
// =========================

AArch32.ExceptionReturn(bits(32) new_pc, bits(32) spsr)
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    SynchronizeContext();

    // Attempts to change to an illegal mode or state will invoke the Illegal Execution state
    // mechanism
    SetPSTATEFromPSR(spsr);
    ClearExclusiveLocal(ProcessorID());
    SendEventLocal();

    if PSTATE.IL == '1' then
        // If the exception return is illegal, PC[1:0] are UNKNOWN
        new_pc<1:0> = bits(2) UNKNOWN;
    else
        // LR[1:0] or LR[0] are treated as being 0, depending on the target instruction set state
        if PSTATE.T == '1' then
            new_pc<0> = '0';                // T32
        else
            new_pc<1:0> = '00';             // A32

    BranchTo(new_pc, BranchType_ERET);
```

aarch32/functions/system/AArch32.ExecutingATS1xPInstr

```
// AArch32.ExecutingATS1xPInstr()
// ==============================
// Return TRUE if current instruction is AT S1CPR/WP

boolean AArch32.ExecutingATS1xPInstr()
    if !HavePrivATExt() then return FALSE;

    instr = ThisInstr();
    if instr<24+:4> == '1110' && instr<8+:4> == '1110' then
        op1 = instr<21+:3>;
        CRn = instr<16+:4>;
        CRm = instr<0+:4>;
        op2 = instr<5+:3>;
        return (op1 == '000' && CRn == '0111' && CRm == '1001' && op2 IN {'000','001'});
    else
        return FALSE;
```

aarch32/functions/system/AArch32.ExecutingCP10or11Instr

```
// AArch32.ExecutingCP10or11Instr()
// ================================

boolean AArch32.ExecutingCP10or11Instr()
    instr =  ThisInstr();
    instr_set = CurrentInstrSet();
    assert instr_set IN {InstrSet_A32, InstrSet_T32};

    if instr_set == InstrSet_A32 then
        return ((instr<27:24> == '1110' || instr<27:25> == '110') && instr<11:8> == '101x');
    else // InstrSet_T32
        return (instr<31:28> == '111x' && (instr<27:24> == '1110' || instr<27:25> == '110') &&
instr<11:8> == '101x');
```

aarch32/functions/system/AArch32.ExecutingLSMInstr

```
// AArch32.ExecutingLSMInstr()
// ===========================
// Returns TRUE if processor is executing a Load/Store Multiple instruction

boolean AArch32.ExecutingLSMInstr()
    instr =  ThisInstr();
    instr_set = CurrentInstrSet();
    assert instr_set IN {InstrSet_A32, InstrSet_T32};
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if instr_set == InstrSet_A32 then
        return (instr<28+:4> != '1111' && instr<25+:3> == '100');
    else // InstrSet_T32
        if ThisInstrLength() == 16 then
            return (instr<12+:4> == '1100');
        else
            return (instr<25+:7> == '1110100' && instr<22> == '0');
```

aarch32/functions/system/AArch32.ITAdvance

```
// AArch32.ITAdvance()
// ===================

AArch32.ITAdvance()
    if PSTATE.IT<2:0> == '000' then
        PSTATE.IT = '00000000';
    else
        PSTATE.IT<4:0> = LSL(PSTATE.IT<4:0>, 1);
    return;
```

aarch32/functions/system/AArch32.SysRegRead

```
// Read from a 32-bit AArch32 System register and return the register's contents.
bits(32) AArch32.SysRegRead(integer cp_num, bits(32) instr);
```

aarch32/functions/system/AArch32.SysRegRead64

```
// Read from a 64-bit AArch32 System register and return the register's contents.
bits(64) AArch32.SysRegRead64(integer cp_num, bits(32) instr);
```

aarch32/functions/system/AArch32.SysRegReadCanWriteAPSR

```
// AArch32.SysRegReadCanWriteAPSR()
// ================================
// Determines whether the AArch32 System register read instruction can write to APSR flags.

boolean AArch32.SysRegReadCanWriteAPSR(integer cp_num, bits(32) instr)
    assert UsingAArch32();
    assert (cp_num IN {14,15});
    assert cp_num == UInt(instr<11:8>);

    opc1 = UInt(instr<23:21>);
    opc2 = UInt(instr<7:5>);
    CRn  = UInt(instr<19:16>);
    CRm  = UInt(instr<3:0>);

    if cp_num == 14 && opc1 == 0 && CRn == 0 && CRm == 1 && opc2 == 0 then // DBGDSCRint
        return TRUE;

    return FALSE;
```

aarch32/functions/system/AArch32.SysRegWrite

```
// Write to a 32-bit AArch32 System register.
AArch32.SysRegWrite(integer cp_num, bits(32) instr, bits(32) val);
```

aarch32/functions/system/AArch32.SysRegWrite64

```
// Write to a 64-bit AArch32 System register.
AArch32.SysRegWrite64(integer cp_num, bits(32) instr, bits(64) val);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch32/functions/system/AArch32.WriteMode

```
// AArch32.WriteMode()
// ===================
// Function for dealing with writes to PSTATE.M from AArch32 state only.
// This ensures that PSTATE.EL and PSTATE.SP are always valid.

AArch32.WriteMode(bits(5) mode)
    (valid,el) = ELFromM32(mode);
    assert valid;
    PSTATE.M   = mode;
    PSTATE.EL  = el;
    PSTATE.nRW = '1';
    PSTATE.SP  = (if mode IN {M32_User,M32_System} then '0' else '1');
    return;
```

aarch32/functions/system/AArch32.WriteModeByInstr

```
// AArch32.WriteModeByInstr()
// ==========================
// Function for dealing with writes to PSTATE.M from an AArch32 instruction, and ensuring that
// illegal state changes are correctly flagged in PSTATE.IL.

AArch32.WriteModeByInstr(bits(5) mode)
    (valid,el) = ELFromM32(mode);

    // 'valid' is set to FALSE if' mode' is invalid for this implementation or the current value
    // of SCR.NS/SCR_EL3.NS. Additionally, it is illegal for an instruction to write 'mode' to
    // PSTATE.EL if it would result in any of:
    // * A change to a mode that would cause entry to a higher Exception level.
    if UInt(el) > UInt(PSTATE.EL) then
        valid = FALSE;

    // * A change to or from Hyp mode.
    if (PSTATE.M == M32_Hyp || mode == M32_Hyp) && PSTATE.M != mode then
        valid = FALSE;

    // * When EL2 is implemented, the value of HCR.TGE is '1', a change to a Non-secure EL1 mode.
    if PSTATE.M == M32_Monitor && HaveEL(EL2) && el == EL1 && SCR.NS == '1' && HCR.TGE == '1' then
        valid = FALSE;

    if !valid then
        PSTATE.IL = '1';
    else
        AArch32.WriteMode(mode);
```

aarch32/functions/system/BadMode

```
// BadMode()
// =========

boolean BadMode(bits(5) mode)
    // Return TRUE if 'mode' encodes a mode that is not valid for this implementation
    case mode of
        when M32_Monitor
            valid = HaveAArch32EL(EL3);
        when M32_Hyp
            valid = HaveAArch32EL(EL2);
        when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32_Undef, M32_System
            // If EL3 is implemented and using AArch32, then these modes are EL3 modes in Secure
            // state, and EL1 modes in Non-secure state. If EL3 is not implemented or is using
            // AArch64, then these modes are EL1 modes.
            // Therefore it is sufficient to test this implementation supports EL1 using AArch32.
            valid = HaveAArch32EL(EL1);
        when M32_User
            valid = HaveAArch32EL(EL0);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        otherwise
            valid = FALSE;           // Passed an illegal mode value
    return !valid;
```

aarch32/functions/system/BankedRegisterAccessValid

```
// BankedRegisterAccessValid()
// ===========================
// Checks for MRS (Banked register) or MSR (Banked register) accesses to registers
// other than the SPSRs that are invalid. This includes ELR_hyp accesses.

BankedRegisterAccessValid(bits(5) SYSm, bits(5) mode)

    case SYSm of
        when '000xx', '00100'                        // R8_usr to R12_usr
            if mode != M32_FIQ then UNPREDICTABLE;
        when '00101'                                 // SP_usr
            if mode == M32_System then UNPREDICTABLE;
        when '00110'                                 // LR_usr
            if mode IN {M32_Hyp,M32_System} then UNPREDICTABLE;
        when '010xx', '0110x', '01110'               // R8_fiq to R12_fiq, SP_fiq, LR_fiq
            if mode == M32_FIQ then UNPREDICTABLE;
        when '1000x'                                 // LR_irq, SP_irq
            if mode == M32_IRQ then UNPREDICTABLE;
        when '1001x'                                 // LR_svc, SP_svc
            if mode == M32_Svc then UNPREDICTABLE;
        when '1010x'                                 // LR_abt, SP_abt
            if mode == M32_Abort then UNPREDICTABLE;
        when '1011x'                                 // LR_und, SP_und
            if mode == M32_Undef then UNPREDICTABLE;
        when '1110x'                                 // LR_mon, SP_mon
            if !HaveEL(EL3) || !IsSecure() || mode == M32_Monitor then UNPREDICTABLE;
        when '11110'                                 // ELR_hyp, only from Monitor or Hyp mode
            if !HaveEL(EL2) || !(mode IN {M32_Monitor,M32_Hyp}) then UNPREDICTABLE;
        when '11111'                                 // SP_hyp, only from Monitor mode
            if !HaveEL(EL2) || mode != M32_Monitor then UNPREDICTABLE;
        otherwise
            UNPREDICTABLE;

    return;
```

aarch32/functions/system/CPSRWriteByInstr

```
// CPSRWriteByInstr()
// ==================
// Update PSTATE.<N,Z,C,V,Q,GE,E,A,I,F,M> from a CPSR value written by an MSR instruction.

CPSRWriteByInstr(bits(32) value, bits(4) bytemask)
    privileged = PSTATE.EL != EL0;              // PSTATE.<A,I,F,M> are not writable at EL0

    // Write PSTATE from 'value', ignoring bytes masked by 'bytemask'
    if bytemask<3> == '1' then
        PSTATE.<N,Z,C,V,Q> = value<31:27>;
        // Bits <26:24> are ignored

    if bytemask<2> == '1' then
        // Bit <23> is RES0
        if privileged then
            PSTATE.PAN = value<22>;
        if HaveDITExt() then
            PSTATE.DIT = value<21>;
        // Bit <20> is RES0
        PSTATE.GE = value<19:16>;

    if bytemask<1> == '1' then
        // Bits <15:10> are RES0
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        PSTATE.E = value<9>;                    // PSTATE.E is writable at EL0
        if privileged then
            PSTATE.A = value<8>;

    if bytemask<0> == '1' then
        if privileged then
            PSTATE.<I,F> = value<7:6>;
            // Bit <5> is RES0
            // AArch32.WriteModeByInstr() sets PSTATE.IL to 1 if this is an illegal mode change.
            AArch32.WriteModeByInstr(value<4:0>);
    return;
```

aarch32/functions/system/ConditionPassed

```
// ConditionPassed()
// =================

boolean ConditionPassed()
    return ConditionHolds(AArch32.CurrentCond());
```

aarch32/functions/system/CurrentCond

```
bits(4) AArch32.CurrentCond();
```

aarch32/functions/system/InITBlock

```
// InITBlock()
// ===========

boolean InITBlock()
    if CurrentInstrSet() == InstrSet_T32 then
        return PSTATE.IT<3:0> != '0000';
    else
        return FALSE;
```

aarch32/functions/system/LastInITBlock

```
// LastInITBlock()
// ===============

boolean LastInITBlock()
    return (PSTATE.IT<3:0> == '1000');
```

aarch32/functions/system/SPSRWriteByInstr

```
// SPSRWriteByInstr()
// ==================

SPSRWriteByInstr(bits(32) value, bits(4) bytemask)

    new_spsr = SPSR[];

    if bytemask<3> == '1' then
        new_spsr<31:24> = value<31:24>;  // N,Z,C,V,Q flags, IT[1:0],J bits

    if bytemask<2> == '1' then
        new_spsr<23:16> = value<23:16>;  // IL bit, GE[3:0] flags

    if bytemask<1> == '1' then
        new_spsr<15:8> = value<15:8>;    // IT[7:2] bits, E bit, A interrupt mask

    if bytemask<0> == '1' then
        new_spsr<7:0> = value<7:0>;      // I,F interrupt masks, T bit, Mode bits
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    SPSR[] = new_spsr;                  // UNPREDICTABLE if User or System mode

    return;
```

aarch32/functions/system/SPSRaccessValid

```
// SPSRaccessValid()
// =================
// Checks for MRS (Banked register) or MSR (Banked register) accesses to the SPSRs
// that are UNPREDICTABLE

SPSRaccessValid(bits(5) SYSm, bits(5) mode)
    case SYSm of
        when '01110'                                                    // SPSR_fiq
            if mode == M32_FIQ   then UNPREDICTABLE;
        when '10000'                                                    // SPSR_irq
            if mode == M32_IRQ   then UNPREDICTABLE;
        when '10010'                                                    // SPSR_svc
            if mode == M32_Svc   then UNPREDICTABLE;
        when '10100'                                                    // SPSR_abt
            if mode == M32_Abort then UNPREDICTABLE;
        when '10110'                                                    // SPSR_und
            if mode == M32_Undef then UNPREDICTABLE;
        when '11100'                                                    // SPSR_mon
            if !HaveEL(EL3) || mode == M32_Monitor || !IsSecure() then UNPREDICTABLE;
        when '11110'                                                    // SPSR_hyp
            if !HaveEL(EL2) || mode != M32_Monitor then UNPREDICTABLE;
        otherwise
            UNPREDICTABLE;

    return;
```

aarch32/functions/system/SelectInstrSet

```
// SelectInstrSet()
// ================

SelectInstrSet(InstrSet iset)
    assert CurrentInstrSet() IN {InstrSet_A32, InstrSet_T32};
    assert iset IN {InstrSet_A32, InstrSet_T32};

    PSTATE.T = if iset == InstrSet_A32 then '0' else '1';

    return;
```

aarch32/functions/v6simd/Sat

```
// Sat()
// =====

bits(N) Sat(integer i, integer N, boolean unsigned)
    result = if unsigned then UnsignedSat(i, N) else SignedSat(i, N);
    return result;
```

aarch32/functions/v6simd/SignedSat

```
// SignedSat()
// ===========

bits(N) SignedSat(integer i, integer N)
    (result, -) = SignedSatQ(i, N);
    return result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch32/functions/v6simd/UnsignedSat

```
// UnsignedSat()
// =============

bits(N) UnsignedSat(integer i, integer N)
    (result, -) = UnsignedSatQ(i, N);
    return result;
```

J1.2.4 aarch32/translation

This section includes the following pseudocode functions:

- *aarch32/translation/attrs/AArch32.CombineS1S2Desc*.
- *aarch32/translation/attrs/AArch32.DefaultTEXDecode* on page J1-7684.
- *aarch32/translation/attrs/AArch32.InstructionDevice* on page J1-7685.
- *aarch32/translation/attrs/AArch32.RemappedTEXDecode* on page J1-7685.
- *aarch32/translation/attrs/AArch32.S1AttrDecode* on page J1-7686.
- *aarch32/translation/attrs/AArch32.TranslateAddressS1Off* on page J1-7687.
- *aarch32/translation/checks/AArch32.AccessIsPrivileged* on page J1-7688.
- *aarch32/translation/checks/AArch32.AccessUsesEL* on page J1-7688.
- *aarch32/translation/checks/AArch32.CheckDomain* on page J1-7688.
- *aarch32/translation/checks/AArch32.CheckPermission* on page J1-7689.
- *aarch32/translation/checks/AArch32.CheckS2Permission* on page J1-7690.
- *aarch32/translation/debug/AArch32.CheckBreakpoint* on page J1-7691.
- *aarch32/translation/debug/AArch32.CheckDebug* on page J1-7691.
- *aarch32/translation/debug/AArch32.CheckVectorCatch* on page J1-7692.
- *aarch32/translation/debug/AArch32.CheckWatchpoint* on page J1-7692.
- *aarch32/translation/faults/AArch32.AccessFlagFault* on page J1-7692.
- *aarch32/translation/faults/AArch32.AddressSizeFault* on page J1-7693.
- *aarch32/translation/faults/AArch32.AlignmentFault* on page J1-7693.
- *aarch32/translation/faults/AArch32.AsynchExternalAbort* on page J1-7693.
- *aarch32/translation/faults/AArch32.DebugFault* on page J1-7693.
- *aarch32/translation/faults/AArch32.DomainFault* on page J1-7694.
- *aarch32/translation/faults/AArch32.NoFault* on page J1-7694.
- *aarch32/translation/faults/AArch32.PermissionFault* on page J1-7694.
- *aarch32/translation/faults/AArch32.TranslationFault* on page J1-7694.
- *aarch32/translation/translation/AArch32.FirstStageTranslate* on page J1-7695.
- *aarch32/translation/translation/AArch32.FullTranslate* on page J1-7696.
- *aarch32/translation/translation/AArch32.SecondStageTranslate* on page J1-7696.
- *aarch32/translation/translation/AArch32.SecondStageWalk* on page J1-7697.
- *aarch32/translation/translation/AArch32.TranslateAddress* on page J1-7697.
- *aarch32/translation/walk/AArch32.TranslationTableWalkLD* on page J1-7698.
- *aarch32/translation/walk/AArch32.TranslationTableWalkSD* on page J1-7701.
- *aarch32/translation/walk/RemapRegsHaveResetValues* on page J1-7704.

aarch32/translation/attrs/AArch32.CombineS1S2Desc

```
// AArch32.CombineS1S2Desc()
// =========================
// Combines the address descriptors from stage 1 and stage 2

AddressDescriptor AArch32.CombineS1S2Desc(AddressDescriptor s1desc, AddressDescriptor s2desc)

    AddressDescriptor result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    result.paddress = s2desc.paddress;

    apply_force_writeback = HaveStage2MemAttrControl() && HCR_EL2.FWB == '1';
    if IsFault(s1desc) || IsFault(s2desc) then
        result = if IsFault(s1desc) then s1desc else s2desc;
    else
        result.fault = AArch32.NoFault();
        if s2desc.memattrs.memtype == MemType_Device || (
            (apply_force_writeback && s1desc.memattrs.memtype == MemType_Device &&
s2desc.memattrs.inner.attrs != '10') ||
            (!apply_force_writeback && s1desc.memattrs.memtype == MemType_Device) ) then
            result.memattrs.memtype = MemType_Device;
            if s1desc.memattrs.memtype == MemType_Normal then
                result.memattrs.device = s2desc.memattrs.device;
            elsif s2desc.memattrs.memtype == MemType_Normal then
                result.memattrs.device = s1desc.memattrs.device;
            else                    // Both Device
                result.memattrs.device = CombineS1S2Device(s1desc.memattrs.device,
                                                           s2desc.memattrs.device);
            result.memattrs.tagged = FALSE;
        // S1 can be either Normal or Device, S2 is Normal.
        else
            result.memattrs.memtype = MemType_Normal;
            result.memattrs.device = DeviceType UNKNOWN;
            result.memattrs.inner = CombineS1S2AttrHints(s1desc.memattrs.inner, s2desc.memattrs.inner);
            result.memattrs.outer = CombineS1S2AttrHints(s1desc.memattrs.outer, s2desc.memattrs.outer);
            result.memattrs.shareable = (s1desc.memattrs.shareable || s2desc.memattrs.shareable);
            result.memattrs.outershareable = (s1desc.memattrs.outershareable ||
                                              s2desc.memattrs.outershareable);
            result.memattrs.tagged = (s1desc.memattrs.tagged &&
                                      result.memattrs.inner.attrs == MemAttr_WB &&
                                      result.memattrs.inner.hints == MemHint_RWA &&
                                      result.memattrs.outer.attrs == MemAttr_WB &&
                                      result.memattrs.outer.hints == MemHint_RWA);

    result.memattrs = MemAttrDefaults(result.memattrs);

    return result;
```

aarch32/translation/attrs/AArch32.DefaultTEXDecode

```
// AArch32.DefaultTEXDecode()
// ==========================

MemoryAttributes AArch32.DefaultTEXDecode(bits(3) TEX, bit C, bit B, bit S, AccType acctype)

    MemoryAttributes memattrs;

    // Reserved values map to allocated values
    if (TEX == '001' && C:B == '01') || (TEX == '010' && C:B != '00') || TEX == '011' then
        bits(5) texcb;
        (-, texcb) = ConstrainUnpredictableBits();
        TEX = texcb<4:2>;  C = texcb<1>;  B = texcb<0>;

    case TEX:C:B of
        when '00000'
            // Device-nGnRnE
            memattrs.memtype = MemType_Device;
            memattrs.device = DeviceType_nGnRnE;
        when '00001', '01000'
            // Device-nGnRE
            memattrs.memtype = MemType_Device;
            memattrs.device = DeviceType_nGnRE;
        when '00010', '00011', '00100'
            // Write-back or Write-through Read allocate, or Non-cacheable
            memattrs.memtype = MemType_Normal;
            memattrs.inner = ShortConvertAttrsHints(C:B, acctype, FALSE);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            memattrs.outer = ShortConvertAttrsHints(C:B, acctype, FALSE);
            memattrs.shareable = (S == '1');
        when '00110'
            memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;
        when '00111'
            // Write-back Read and Write allocate
            memattrs.memtype = MemType_Normal;
            memattrs.inner = ShortConvertAttrsHints('01', acctype, FALSE);
            memattrs.outer = ShortConvertAttrsHints('01', acctype, FALSE);
            memattrs.shareable = (S == '1');
        when '1xxxx'
            // Cacheable, TEX<1:0> = Outer attrs, {C,B} = Inner attrs
            memattrs.memtype = MemType_Normal;
            memattrs.inner = ShortConvertAttrsHints(C:B, acctype, FALSE);
            memattrs.outer = ShortConvertAttrsHints(TEX<1:0>, acctype, FALSE);
            memattrs.shareable = (S == '1');
        otherwise
            // Reserved, handled above
            Unreachable();

    // transient bits are not supported in this format
    memattrs.inner.transient = FALSE;
    memattrs.outer.transient = FALSE;

    // distinction between inner and outer shareable is not supported in this format
    memattrs.outershareable = memattrs.shareable;
    memattrs.tagged = FALSE;

    return MemAttrDefaults(memattrs);
```

aarch32/translation/attrs/AArch32.InstructionDevice

```
// AArch32.InstructionDevice()
// ==========================
// Instruction fetches from memory marked as Device but not execute-never might generate a
// Permission Fault but are otherwise treated as if from Normal Non-cacheable memory.

AddressDescriptor AArch32.InstructionDevice(AddressDescriptor addrdesc, bits(32) vaddress,
                                            bits(40) ipaddress, integer level,  bits(4) domain,
                                            AccType acctype, boolean iswrite, boolean secondstage,
                                            boolean s2fs1walk)

    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_FAULT};

    if c == Constraint_FAULT then
        addrdesc.fault = AArch32.PermissionFault(ipaddress,  domain, level, acctype, iswrite,
                                                 secondstage, s2fs1walk);
    else
        addrdesc.memattrs.memtype = MemType_Normal;
        addrdesc.memattrs.inner.attrs = MemAttr_NC;
        addrdesc.memattrs.inner.hints = MemHint_No;
        addrdesc.memattrs.outer = addrdesc.memattrs.inner;
        addrdesc.memattrs.tagged = FALSE;
        addrdesc.memattrs = MemAttrDefaults(addrdesc.memattrs);

    return addrdesc;
```

aarch32/translation/attrs/AArch32.RemappedTEXDecode

```
// AArch32.RemappedTEXDecode()
// ==========================

MemoryAttributes AArch32.RemappedTEXDecode(bits(3) TEX, bit C, bit B, bit S, AccType acctype)

    MemoryAttributes memattrs;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    region = UInt(TEX<0>:C:B);         // TEX<2:1> are ignored in this mapping scheme
    if region == 6 then
        memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;
    else
        base = 2 * region;
        attrfield = PRRR<base+1:base>;

        if attrfield == '11' then      // Reserved, maps to allocated value
            (-, attrfield) = ConstrainUnpredictableBits();

        case attrfield of
            when '00'                  // Device-nGnRnE
                memattrs.memtype = MemType_Device;
                memattrs.device = DeviceType_nGnRnE;
            when '01'                  // Device-nGnRE
                memattrs.memtype = MemType_Device;
                memattrs.device = DeviceType_nGnRE;
            when '10'
                memattrs.memtype = MemType_Normal;
                memattrs.inner = ShortConvertAttrsHints(NMRR<base+1:base>, acctype, FALSE);
                memattrs.outer = ShortConvertAttrsHints(NMRR<base+17:base+16>, acctype, FALSE);
                s_bit = if S == '0' then PRRR.NS0 else PRRR.NS1;
                memattrs.shareable = (s_bit == '1');
                memattrs.outershareable = (s_bit == '1' && PRRR<region+24> == '0');
            when '11'
                Unreachable();

    // transient bits are not supported in this format
    memattrs.inner.transient = FALSE;
    memattrs.outer.transient = FALSE;
    memattrs.tagged = FALSE;

    return MemAttrDefaults(memattrs);
```

aarch32/translation/attrs/AArch32.S1AttrDecode

```
// AArch32.S1AttrDecode()
// ======================
// Converts the Stage 1 attribute fields, using the MAIR, to orthogonal
// attributes and hints.

MemoryAttributes AArch32.S1AttrDecode(bits(2) SH, bits(3) attr, AccType acctype)

    MemoryAttributes memattrs;

    if PSTATE.EL == EL2 then
        mair = HMAIR1:HMAIR0;
    else
        mair = MAIR1:MAIR0;
    index = 8 * UInt(attr);
    attrfield = mair<index+7:index>;

    memattrs.tagged = FALSE;
    if ((attrfield<7:4> != '0000' && attrfield<7:4> != '1111' && attrfield<3:0> == '0000') ||
        (attrfield<7:4> == '0000' && attrfield<3:0> != 'xx00')) then
        // Reserved, maps to an allocated value
        (-, attrfield) = ConstrainUnpredictableBits();
    if !HaveMTEExt() && attrfield<7:4> == '1111' && attrfield<3:0> == '0000' then
        // Reserved, maps to an allocated value
        (-, attrfield) = ConstrainUnpredictableBits();

    if attrfield<7:4> == '0000' then           // Device
        memattrs.memtype = MemType_Device;
        case attrfield<3:0> of
            when '0000'  memattrs.device = DeviceType_nGnRnE;
            when '0100'  memattrs.device = DeviceType_nGnRE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            when '1000'  memattrs.device = DeviceType_nGRE;
            when '1100'  memattrs.device = DeviceType_GRE;
            otherwise    Unreachable();         // Reserved, handled above

    elsif attrfield<3:0> != '0000'  then        // Normal
        memattrs.memtype = MemType_Normal;
        memattrs.outer = LongConvertAttrsHints(attrfield<7:4>, acctype);
        memattrs.inner = LongConvertAttrsHints(attrfield<3:0>, acctype);
        memattrs.shareable = SH<1> == '1';
        memattrs.outershareable = SH == '10';
    elsif HaveMTEExt() && attrfield == '11110000' then // Normal, Tagged if WB-RWA
        memattrs.memtype = MemType_Normal;
        memattrs.outer = LongConvertAttrsHints('1111', acctype); // WB_RWA
        memattrs.inner = LongConvertAttrsHints('1111', acctype); // WB_RWA
        memattrs.shareable = SH<1> == '1';
        memattrs.outershareable = SH == '10';
        memattrs.tagged = (memattrs.inner.attrs == MemAttr_WB &&
                           memattrs.inner.hints == MemHint_RWA &&
                           memattrs.outer.attrs == MemAttr_WB &&
                           memattrs.outer.hints == MemHint_RWA);
    else
        Unreachable();                          // Reserved, handled above

    return MemAttrDefaults(memattrs);
```

aarch32/translation/attrs/AArch32.TranslateAddressS1Off

```
// AArch32.TranslateAddressS1Off()
// ==============================
// Called for stage 1 translations when translation is disabled to supply a default translation.
// Note that there are additional constraints on instruction prefetching that are not described in
// this pseudocode.

TLBRecord AArch32.TranslateAddressS1Off(bits(32) vaddress, AccType acctype, boolean iswrite)
    assert ELUsingAArch32(S1TranslationRegime());

    TLBRecord result;

    default_cacheable = (HasS2Translation() && ((if ELUsingAArch32(EL2) then HCR.DC else HCR_EL2.DC) ==
'1'));

    if default_cacheable then
        // Use default cacheable settings
        result.addrdesc.memattrs.memtype = MemType_Normal;
        result.addrdesc.memattrs.inner.attrs = MemAttr_WB;      // Write-back
        result.addrdesc.memattrs.inner.hints = MemHint_RWA;
        result.addrdesc.memattrs.shareable = FALSE;
        result.addrdesc.memattrs.outershareable = FALSE;
        result.addrdesc.memattrs.tagged = HCR_EL2.DCT == '1';
    elsif acctype != AccType_IFETCH then
        // Treat data as Device
        result.addrdesc.memattrs.memtype = MemType_Device;
        result.addrdesc.memattrs.device = DeviceType_nGnRnE;
        result.addrdesc.memattrs.inner = MemAttrHints UNKNOWN;
        result.addrdesc.memattrs.tagged = FALSE;
    else
        // Instruction cacheability controlled by SCTLR/HSCTLR.I
        if PSTATE.EL == EL2 then
            cacheable = HSCTLR.I == '1';
        else
            cacheable = SCTLR.I == '1';
        result.addrdesc.memattrs.memtype = MemType_Normal;
        if cacheable then
            result.addrdesc.memattrs.inner.attrs = MemAttr_WT;
            result.addrdesc.memattrs.inner.hints = MemHint_RA;
        else
            result.addrdesc.memattrs.inner.attrs = MemAttr_NC;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            result.addrdesc.memattrs.inner.hints = MemHint_No;
        result.addrdesc.memattrs.shareable = TRUE;
        result.addrdesc.memattrs.outershareable = TRUE;
        result.addrdesc.memattrs.tagged = FALSE;

    result.addrdesc.memattrs.outer = result.addrdesc.memattrs.inner;

    result.addrdesc.memattrs = MemAttrDefaults(result.addrdesc.memattrs);

    result.perms.ap = bits(3) UNKNOWN;
    result.perms.xn = '0';
    result.perms.pxn = '0';

    result.nG = bit UNKNOWN;
    result.contiguous = boolean UNKNOWN;
    result.domain = bits(4) UNKNOWN;
    result.level = integer UNKNOWN;
    result.blocksize = integer UNKNOWN;
    result.addrdesc.paddress.address = ZeroExtend(vaddress);
    result.addrdesc.paddress.NS = if IsSecure() then '0' else '1';
    result.addrdesc.fault = AArch32.NoFault();
    return result;
```

aarch32/translation/checks/AArch32.AccessIsPrivileged

```
// AArch32.AccessIsPrivileged()
// ============================

boolean AArch32.AccessIsPrivileged(AccType acctype)

    el = AArch32.AccessUsesEL(acctype);

    if el == EL0 then
        ispriv = FALSE;
    elsif el != EL1 then
        ispriv = TRUE;
    else
        ispriv = (acctype != AccType_UNPRIV);

    return ispriv;
```

aarch32/translation/checks/AArch32.AccessUsesEL

```
// AArch32.AccessUsesEL()
// ======================
// Returns the Exception Level of the regime that will manage the translation for a given access type.

bits(2) AArch32.AccessUsesEL(AccType acctype)
    if acctype == AccType_UNPRIV then
        return EL0;
    else
        return PSTATE.EL;
```

aarch32/translation/checks/AArch32.CheckDomain

```
// AArch32.CheckDomain()
// =====================

(boolean, FaultRecord) AArch32.CheckDomain(bits(4) domain, bits(32) vaddress, integer level,
                                           AccType acctype, boolean iswrite)

    index = 2 * UInt(domain);
    attrfield = DACR<index+1:index>;

    if attrfield == '10' then          // Reserved, maps to an allocated value
        // Reserved value maps to an allocated value
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        (-, attrfield) = ConstrainUnpredictableBits();

    if attrfield == '00' then
        fault = AArch32.DomainFault(domain, level, acctype, iswrite);
    else
        fault = AArch32.NoFault();

    permissioncheck = (attrfield == '01');

    return (permissioncheck, fault);
```

aarch32/translation/checks/AArch32.CheckPermission

```
// AArch32.CheckPermission()
// =========================
// Function used for permission checking from AArch32 stage 1 translations

FaultRecord AArch32.CheckPermission(Permissions perms, bits(32) vaddress, integer level,
                                    bits(4) domain, bit NS, AccType acctype, boolean iswrite)
    assert ELUsingAArch32(S1TranslationRegime());

    if PSTATE.EL != EL2 then
        wxn = SCTLR.WXN == '1';
        if TTBCR.EAE == '1' || SCTLR.AFE == '1' || perms.ap<0> == '1' then
            priv_r = TRUE;
            priv_w = perms.ap<2> == '0';
            user_r = perms.ap<1> == '1';
            user_w = perms.ap<2:1> == '01';
        else
            priv_r = perms.ap<2:1> != '00';
            priv_w = perms.ap<2:1> == '01';
            user_r = perms.ap<1> == '1';
            user_w = FALSE;
        uwxn = SCTLR.UWXN == '1';

        ispriv = AArch32.AccessIsPrivileged(acctype);

        pan = if HavePANExt() then PSTATE.PAN else '0';
        is_ldst   = !(acctype IN {AccType_DC, AccType_DC_UNPRIV, AccType_AT, AccType_IFETCH});
        is_ats1xp = (acctype == AccType_AT && AArch32.ExecutingATS1xPInstr());
        if pan == '1' && user_r && ispriv && (is_ldst || is_ats1xp) then
            priv_r = FALSE;
            priv_w = FALSE;

        user_xn = !user_r || perms.xn == '1' || (user_w && wxn);
        priv_xn = (!priv_r || perms.xn == '1' || perms.pxn == '1' ||
                   (priv_w && wxn) || (user_w && uwxn));

        if ispriv then
            (r, w, xn) = (priv_r, priv_w, priv_xn);
        else
            (r, w, xn) = (user_r, user_w, user_xn);
    else
        // Access from EL2
        wxn = HSCTLR.WXN == '1';
        r = TRUE;
        w = perms.ap<2> == '0';
        xn = perms.xn == '1' || (w && wxn);

    // Restriction on Secure instruction fetch
    if HaveEL(EL3) && IsSecure() && NS == '1' then
        secure_instr_fetch = if ELUsingAArch32(EL3) then SCR.SIF else SCR_EL3.SIF;
        if secure_instr_fetch == '1' then xn = TRUE;

    if acctype == AccType_IFETCH then
        fail = xn;
        failedread = TRUE;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif acctype IN { AccType_ATOMICRW, AccType_ORDEREDRW, AccType_ORDEREDATOMICRW } then
        fail = !r || !w;
        failedread = !r;
    elsif acctype == AccType_DC then
        // DC maintenance instructions operating by VA, cannot fault from stage 1 translation.
        fail = FALSE;
    elsif iswrite then
        fail = !w;
        failedread = FALSE;
    else
        fail = !r;
        failedread = TRUE;

    if fail then
        secondstage = FALSE;
        s2fs1walk = FALSE;
        ipaddress = bits(40) UNKNOWN;
        return AArch32.PermissionFault(ipaddress,  domain, level, acctype,
                                       !failedread, secondstage, s2fs1walk);
    else
        return AArch32.NoFault();
```

aarch32/translation/checks/AArch32.CheckS2Permission

```
// AArch32.CheckS2Permission()
// ==========================
// Function used for permission checking from AArch32 stage 2 translations

FaultRecord AArch32.CheckS2Permission(Permissions perms, bits(32) vaddress, bits(40) ipaddress,
                                      integer level, AccType acctype, boolean iswrite,
                                      boolean s2fs1walk)

    assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2) && HasS2Translation();

    r = perms.ap<1> == '1';
    w = perms.ap<2> == '1';
    if HaveExtendedExecuteNeverExt() then
        case perms.xn:perms.xxn of
            when '00'  xn = !r;
            when '01'  xn = !r || PSTATE.EL == EL1;
            when '10'  xn = TRUE;
            when '11'  xn = !r || PSTATE.EL == EL0;
    else
        xn = !r || perms.xn == '1';
    // Stage 1 walk is checked as a read, regardless of the original type
    if acctype == AccType_IFETCH && !s2fs1walk then
        fail = xn;
        failedread = TRUE;
    elsif (acctype IN { AccType_ATOMICRW, AccType_ORDEREDRW, AccType_ORDEREDATOMICRW }) && !s2fs1walk
then
        fail = !r || !w;
        failedread = !r;
    elsif acctype == AccType_DC && !s2fs1walk then
        // DC maintenance instructions operating by VA, do not generate Permission faults
        // from stage 2 translation, other than from stage 1 translation table walk.
        fail = FALSE;
    elsif iswrite && !s2fs1walk then
        fail = !w;
        failedread = FALSE;
    else
        fail = !r;
        failedread = !iswrite;

    if fail then
        domain = bits(4) UNKNOWN;
        secondstage = TRUE;
        return AArch32.PermissionFault(ipaddress,  domain, level, acctype,
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
                                        !failedread, secondstage, s2fs1walk);
    else
        return AArch32.NoFault();
```

aarch32/translation/debug/AArch32.CheckBreakpoint

```
// AArch32.CheckBreakpoint()
// =========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime, when either debug exceptions are enabled, or halting debug is enabled
// and halting is allowed.

FaultRecord AArch32.CheckBreakpoint(bits(32) vaddress,  integer size)
    assert ELUsingAArch32(S1TranslationRegime());
    assert size IN {2,4};

    match = FALSE;
    mismatch = FALSE;

    for i = 0 to UInt(DBGDIDR.BRPs)
        (match_i, mismatch_i) = AArch32.BreakpointMatch(i, vaddress, size);
        match = match || match_i;
        mismatch = mismatch || mismatch_i;

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Breakpoint;
        Halt(reason);
    elsif (match || mismatch) then
        acctype = AccType_IFETCH;
        iswrite = FALSE;
        debugmoe = DebugException_Breakpoint;
        return AArch32.DebugFault(acctype, iswrite, debugmoe);
    else
        return AArch32.NoFault();
```

aarch32/translation/debug/AArch32.CheckDebug

```
// AArch32.CheckDebug()
// ====================
// Called on each access to check for a debug exception or entry to Debug state.

FaultRecord AArch32.CheckDebug(bits(32) vaddress, AccType acctype, boolean iswrite, integer size)

    FaultRecord fault = AArch32.NoFault();

    d_side = (acctype != AccType_IFETCH);
    generate_exception = AArch32.GenerateDebugExceptions() && DBGDSCRext.MDBGen == '1';
    halt = HaltOnBreakpointOrWatchpoint();
    // Relative priority of Vector Catch and Breakpoint exceptions not defined in the architecture
    vector_catch_first = ConstrainUnpredictableBool();

    if !d_side && vector_catch_first && generate_exception then
        fault = AArch32.CheckVectorCatch(vaddress, size);

    if fault.statuscode == Fault_None && (generate_exception || halt) then
        if d_side then
            fault = AArch32.CheckWatchpoint(vaddress, acctype, iswrite, size);
        else
            fault = AArch32.CheckBreakpoint(vaddress, size);

    if fault.statuscode == Fault_None && !d_side && !vector_catch_first && generate_exception then
        return AArch32.CheckVectorCatch(vaddress, size);

    return fault;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch32/translation/debug/AArch32.CheckVectorCatch

```
// AArch32.CheckVectorCatch()
// ==========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime, when debug exceptions are enabled.

FaultRecord AArch32.CheckVectorCatch(bits(32) vaddress, integer size)
    assert ELUsingAArch32(S1TranslationRegime());

    match = AArch32.VCRMatch(vaddress);
    if size == 4 && !match && AArch32.VCRMatch(vaddress + 2) then
        match = ConstrainUnpredictableBool();

    if match then
        acctype = AccType_IFETCH;
        iswrite = FALSE;
        debugmoe = DebugException_VectorCatch;
        return AArch32.DebugFault(acctype, iswrite, debugmoe);
    else
        return AArch32.NoFault();
```

aarch32/translation/debug/AArch32.CheckWatchpoint

```
// AArch32.CheckWatchpoint()
// =========================
// Called before accessing the memory location of "size" bytes at "address",
// when either debug exceptions are enabled for the access, or halting debug
// is enabled and halting is allowed.

FaultRecord AArch32.CheckWatchpoint(bits(32) vaddress, AccType acctype,
                                    boolean iswrite, integer size)
    assert ELUsingAArch32(S1TranslationRegime());

    match = FALSE;
    ispriv = AArch32.AccessIsPrivileged(acctype);

    for i = 0 to UInt(DBGDIDR.WRPs)
        match = match || AArch32.WatchpointMatch(i, vaddress, size, ispriv, iswrite);

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Watchpoint;
        Halt(reason);
    elsif match then
        debugmoe = DebugException_Watchpoint;
        return AArch32.DebugFault(acctype, iswrite, debugmoe);
    else
        return AArch32.NoFault();
```

aarch32/translation/faults/AArch32.AccessFlagFault

```
// AArch32.AccessFlagFault()
// =========================

FaultRecord AArch32.AccessFlagFault(bits(40) ipaddress, bits(4) domain, integer level,
                                    AccType acctype, boolean iswrite, boolean secondstage,
                                    boolean s2fs1walk)

    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch32.CreateFaultRecord(Fault_AccessFlag, ipaddress,  domain, level, acctype, iswrite,
                                     extflag, debugmoe, errortype, secondstage, s2fs1walk);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

aarch32/translation/faults/AArch32.AddressSizeFault

```
// AArch32.AddressSizeFault()
// ==========================

FaultRecord AArch32.AddressSizeFault(bits(40) ipaddress, bits(4) domain, integer level,
                                     AccType acctype, boolean iswrite, boolean secondstage,
                                     boolean s2fs1walk)

    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch32.CreateFaultRecord(Fault_AddressSize, ipaddress,  domain, level, acctype, iswrite,
                                     extflag, debugmoe, errortype, secondstage, s2fs1walk);
```

aarch32/translation/faults/AArch32.AlignmentFault

```
// AArch32.AlignmentFault()
// ========================

FaultRecord AArch32.AlignmentFault(AccType acctype, boolean iswrite, boolean secondstage)

    ipaddress = bits(40) UNKNOWN;
    domain = bits(4) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    s2fs1walk = boolean UNKNOWN;

    return AArch32.CreateFaultRecord(Fault_Alignment, ipaddress,  domain, level, acctype, iswrite,
                                     extflag, debugmoe, errortype, secondstage, s2fs1walk);
```

aarch32/translation/faults/AArch32.AsynchExternalAbort

```
// AArch32.AsynchExternalAbort()
// =============================
// Wrapper function for asynchronous external aborts

FaultRecord AArch32.AsynchExternalAbort(boolean parity, bits(2) errortype, bit extflag)

    faulttype = if parity then Fault_AsyncParity else Fault_AsyncExternal;
    ipaddress = bits(40) UNKNOWN;
    domain = bits(4) UNKNOWN;
    level = integer UNKNOWN;
    acctype = AccType_NORMAL;
    iswrite = boolean UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch32.CreateFaultRecord(faulttype, ipaddress,  domain, level, acctype, iswrite, extflag,
                                     debugmoe,  errortype, secondstage, s2fs1walk);
```

aarch32/translation/faults/AArch32.DebugFault

```
// AArch32.DebugFault()
// ====================

FaultRecord AArch32.DebugFault(AccType acctype, boolean iswrite, bits(4) debugmoe)

    ipaddress = bits(40) UNKNOWN;
    domain = bits(4) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch32.CreateFaultRecord(Fault_Debug, ipaddress,  domain, level, acctype, iswrite,
                                     extflag, debugmoe, errortype, secondstage, s2fs1walk);
```

aarch32/translation/faults/AArch32.DomainFault

```
// AArch32.DomainFault()
// =====================

FaultRecord AArch32.DomainFault(bits(4) domain, integer level, AccType acctype, boolean iswrite)

    ipaddress = bits(40) UNKNOWN;
    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch32.CreateFaultRecord(Fault_Domain, ipaddress,  domain, level, acctype, iswrite,
                                     extflag, debugmoe, errortype, secondstage, s2fs1walk);
```

aarch32/translation/faults/AArch32.NoFault

```
// AArch32.NoFault()
// =================

FaultRecord AArch32.NoFault()

    ipaddress = bits(40) UNKNOWN;
    domain = bits(4) UNKNOWN;
    level = integer UNKNOWN;
    acctype = AccType_NORMAL;
    iswrite = boolean UNKNOWN;
    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch32.CreateFaultRecord(Fault_None, ipaddress,  domain, level, acctype, iswrite,
                                     extflag, debugmoe, errortype, secondstage, s2fs1walk);
```

aarch32/translation/faults/AArch32.PermissionFault

```
// AArch32.PermissionFault()
// =========================

FaultRecord AArch32.PermissionFault(bits(40) ipaddress, bits(4) domain, integer level,
                                    AccType acctype, boolean iswrite, boolean secondstage,
                                    boolean s2fs1walk)

    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch32.CreateFaultRecord(Fault_Permission, ipaddress,  domain, level, acctype, iswrite,
                                     extflag, debugmoe, errortype, secondstage, s2fs1walk);
```

aarch32/translation/faults/AArch32.TranslationFault

```
// AArch32.TranslationFault()
// ==========================

FaultRecord AArch32.TranslationFault(bits(40) ipaddress,  bits(4) domain, integer level,
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
                                    AccType acctype, boolean iswrite, boolean secondstage,
                                    boolean s2fs1walk)

    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch32.CreateFaultRecord(Fault_Translation, ipaddress,  domain, level, acctype, iswrite,
                                    extflag, debugmoe, errortype, secondstage, s2fs1walk);
```

aarch32/translation/translation/AArch32.FirstStageTranslate

```
// AArch32.FirstStageTranslate()
// ============================
// Perform a stage 1 translation walk. The function used by Address Translation operations is
// similar except it uses the translation regime specified for the instruction.

AddressDescriptor AArch32.FirstStageTranslate(bits(32) vaddress, AccType acctype, boolean iswrite,
                                              boolean wasaligned, integer size)

    if PSTATE.EL == EL2 then
        s1_enabled = HSCTLR.M == '1';
    elsif EL2Enabled() then
        tge = (if ELUsingAArch32(EL2) then HCR.TGE else HCR_EL2.TGE);
        dc = (if ELUsingAArch32(EL2) then HCR.DC else HCR_EL2.DC);
        s1_enabled = tge == '0' && dc == '0' && SCTLR.M == '1';
    else
        s1_enabled = SCTLR.M == '1';

    ipaddress = bits(40) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    if s1_enabled then                          // First stage enabled
        use_long_descriptor_format = PSTATE.EL == EL2 || TTBCR.EAE == '1';
        if use_long_descriptor_format then
            S1 = AArch32.TranslationTableWalkLD(ipaddress, vaddress, acctype, iswrite, secondstage,
                                                s2fs1walk, size);
            permissioncheck = TRUE;  domaincheck = FALSE;
        else
            S1 = AArch32.TranslationTableWalkSD(vaddress, acctype, iswrite, size);
            permissioncheck = TRUE;  domaincheck = TRUE;
    else
        S1 = AArch32.TranslateAddressS1Off(vaddress, acctype, iswrite);
        permissioncheck = FALSE;  domaincheck = FALSE;

    if !IsFault(S1.addrdesc) && UsingAArch32() && HaveTrapLoadStoreMultipleDeviceExt() &&
AArch32.ExecutingLSMInstr() then
        if S1.addrdesc.memattrs.memtype == MemType_Device && S1.addrdesc.memattrs.device !=
DeviceType_GRE then
            nTLSMD = if S1TranslationRegime() == EL2 then HSCTLR.nTLSMD else SCTLR.nTLSMD;
            if nTLSMD == '0' then
                S1.addrdesc.fault = AArch32.AlignmentFault(acctype, iswrite, secondstage);

    // Check for unaligned data accesses to Device memory
    if ((!wasaligned && acctype != AccType_IFETCH) || (acctype == AccType_DCZVA))
        && !IsFault(S1.addrdesc) && S1.addrdesc.memattrs.memtype == MemType_Device then
        S1.addrdesc.fault = AArch32.AlignmentFault(acctype, iswrite, secondstage);
    if !IsFault(S1.addrdesc) && domaincheck then
        (permissioncheck, abort) = AArch32.CheckDomain(S1.domain, vaddress, S1.level, acctype,
                                                       iswrite);
        S1.addrdesc.fault = abort;

    if !IsFault(S1.addrdesc) && permissioncheck then
        S1.addrdesc.fault = AArch32.CheckPermission(S1.perms, vaddress, S1.level,
                                                    S1.domain, S1.addrdesc.paddress.NS,
                                                    acctype, iswrite);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    // Check for instruction fetches from Device memory not marked as execute-never. If there has
    // not been a Permission Fault then the memory is not marked execute-never.
    if (!IsFault(S1.addrdesc) && S1.addrdesc.memattrs.memtype == MemType_Device &&
        acctype == AccType_IFETCH) then
        S1.addrdesc = AArch32.InstructionDevice(S1.addrdesc, vaddress, ipaddress, S1.level,
                                                S1.domain, acctype, iswrite,
                                                secondstage, s2fs1walk);

    return S1.addrdesc;
```

aarch32/translation/translation/AArch32.FullTranslate

```
// AArch32.FullTranslate()
// =======================
// Perform both stage 1 and stage 2 translation walks for the current translation regime. The
// function used by Address Translation operations is similar except it uses the translation
// regime specified for the instruction.

AddressDescriptor AArch32.FullTranslate(bits(32) vaddress, AccType acctype, boolean iswrite,
                                        boolean wasaligned, integer size)

    // First Stage Translation
    S1 = AArch32.FirstStageTranslate(vaddress, acctype, iswrite, wasaligned, size);
    if !IsFault(S1) && !(HaveNV2Ext() && acctype == AccType_NV2REGISTER) && HasS2Translation() then
        s2fs1walk = FALSE;
        result = AArch32.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk,
                                              size);
    else
        result = S1;

    return result;
```

aarch32/translation/translation/AArch32.SecondStageTranslate

```
// AArch32.SecondStageTranslate()
// ==============================
// Perform a stage 2 translation walk. The function used by Address Translation operations is
// similar except it uses the translation regime specified for the instruction.

AddressDescriptor AArch32.SecondStageTranslate(AddressDescriptor S1, bits(32) vaddress,
                                               AccType acctype, boolean iswrite, boolean wasaligned,
                                               boolean s2fs1walk, integer size)
    assert HasS2Translation();
    assert IsZero(S1.paddress.address<47:40>);
    hwupdatewalk = FALSE;
    if !ELUsingAArch32(EL2) then
        return AArch64.SecondStageTranslate(S1, ZeroExtend(vaddress, 64), acctype, iswrite,
                                            wasaligned, s2fs1walk, size, hwupdatewalk);

    s2_enabled = HCR.VM == '1' || HCR.DC == '1';
    secondstage = TRUE;

    if s2_enabled then                        // Second stage enabled
        ipaddress = S1.paddress.address<39:0>;
        S2 = AArch32.TranslationTableWalkLD(ipaddress, vaddress, acctype, iswrite, secondstage,
                                            s2fs1walk, size);

        // Check for unaligned data accesses to Device memory
        if ((!wasaligned && acctype != AccType_IFETCH) || (acctype == AccType_DCZVA))
            && S2.addrdesc.memattrs.memtype == MemType_Device && !IsFault(S2.addrdesc) then
            S2.addrdesc.fault = AArch32.AlignmentFault(acctype, iswrite, secondstage);

        // Check for permissions on Stage2 translations
        if !IsFault(S2.addrdesc) then
            S2.addrdesc.fault = AArch32.CheckS2Permission(S2.perms, vaddress, ipaddress, S2.level,
                                                          acctype, iswrite, s2fs1walk);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        // Check for instruction fetches from Device memory not marked as execute-never. As there
        // has not been a Permission Fault then the memory is not marked execute-never.
        if (!s2fs1walk && !IsFault(S2.addrdesc) && S2.addrdesc.memattrs.memtype == MemType_Device &&
            acctype == AccType_IFETCH) then
            domain = bits(4) UNKNOWN;
            S2.addrdesc = AArch32.InstructionDevice(S2.addrdesc, vaddress, ipaddress, S2.level,
                                                    domain, acctype, iswrite,
                                                    secondstage, s2fs1walk);

        if (s2fs1walk && !IsFault(S2.addrdesc) &&
            S2.addrdesc.memattrs.memtype == MemType_Device) then
            // Check for protected table walk.
            if HCR.PTW == '1' then
                domain = bits(4) UNKNOWN;
                S2.addrdesc.fault = AArch32.PermissionFault(ipaddress,
                                          domain, S2.level,
                                        acctype, iswrite, secondstage, s2fs1walk);
            else
                // Translation table walk occurs as Normal Non-cacheable memory.
                S2.addrdesc.memattrs.memtype        = MemType_Normal;
                S2.addrdesc.memattrs.inner.attrs    = MemAttr_NC;
                S2.addrdesc.memattrs.outer.attrs    = MemAttr_NC;
                S2.addrdesc.memattrs.shareable      = TRUE;
                S2.addrdesc.memattrs.outershareable = TRUE;

        result = AArch32.CombineS1S2Desc(S1, S2.addrdesc);
    else
        result = S1;

    return result;
```

aarch32/translation/translation/AArch32.SecondStageWalk

```
// AArch32.SecondStageWalk()
// =========================
// Perform a stage 2 translation on a stage 1 translation page table walk access.

AddressDescriptor AArch32.SecondStageWalk(AddressDescriptor S1, bits(32) vaddress, AccType acctype,
                                          boolean iswrite, integer size)

    assert HasS2Translation();

    s2fs1walk = TRUE;
    wasaligned = TRUE;
    return AArch32.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk,
                                        size);
```

aarch32/translation/translation/AArch32.TranslateAddress

```
// AArch32.TranslateAddress()
// ==========================
// Main entry point for translating an address

AddressDescriptor AArch32.TranslateAddress(bits(32) vaddress, AccType acctype, boolean iswrite,
                                           boolean wasaligned, integer size)

    if !ELUsingAArch32(S1TranslationRegime()) then
        return AArch64.TranslateAddress(ZeroExtend(vaddress, 64), acctype, iswrite, wasaligned,
                                        size);
    result = AArch32.FullTranslate(vaddress, acctype, iswrite, wasaligned, size);

    if !(acctype IN {AccType_PTW, AccType_IC, AccType_AT}) && !IsFault(result) then
        result.fault = AArch32.CheckDebug(vaddress, acctype, iswrite, size);

    // Update virtual address for abort functions
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    result.vaddress = ZeroExtend(vaddress);

    return result;
```

aarch32/translation/walk/AArch32.TranslationTableWalkLD

```
// AArch32.TranslationTableWalkLD()
// ================================
// Returns a result of a translation table walk using the Long-descriptor format
//
// Implementations might cache information from memory in any number of non-coherent TLB
// caching structures, and so avoid memory accesses that have been expressed in this
// pseudocode. The use of such TLBs is not expressed in this pseudocode.

TLBRecord AArch32.TranslationTableWalkLD(bits(40) ipaddress, bits(32) vaddress,
                                         AccType acctype, boolean iswrite, boolean secondstage,
                                         boolean s2fs1walk, integer size)
    if !secondstage then
        assert ELUsingAArch32(S1TranslationRegime());
    else
        assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2) && HasS2Translation();

    TLBRecord result;
    AddressDescriptor descaddr;
    bits(64) baseregister;
    bits(40) inputaddr;        // Input Address is 'vaddress' for stage 1, 'ipaddress' for stage 2
    bit nswalk;                    // Stage 2 translation table walks are to Secure or to Non-secure PA
space

    domain = bits(4) UNKNOWN;

    descaddr.memattrs.memtype = MemType_Normal;

    // Fixed parameters for the page table walk:
    //  grainsize = Log2(Size of Table)         - Size of Table is 4KB in AArch32
    //  stride = Log2(Address per Level)        - Bits of address consumed at each level
    constant integer grainsize = 12;                   // Log2(4KB page size)
    constant integer stride = grainsize - 3;           // Log2(page size / 8 bytes)

    // Derived parameters for the page table walk:
    //  inputsize = Log2(Size of Input Address) - Input Address size in bits
    //  level = Level to start walk from
    // This means that the number of levels after start level = 3-level

    if !secondstage then
        // First stage translation
        inputaddr = ZeroExtend(vaddress);
        el = AArch32.AccessUsesEL(acctype);
        if el == EL2 then
            inputsize = 32 - UInt(HTCR.T0SZ);
            basefound = inputsize == 32 || IsZero(inputaddr<31:inputsize>);
            disabled = FALSE;
            baseregister = HTTBR;
            descaddr.memattrs = WalkAttrDecode(HTCR.SH0, HTCR.ORGN0, HTCR.IRGN0, secondstage);
            reversedescriptors = HSCTLR.EE == '1';
            lookupsecure = FALSE;
            singlepriv = TRUE;
            hierattrsdisabled = AArch32.HaveHPDExt() && HTCR.HPD == '1';
        else
            basefound = FALSE;
            disabled = FALSE;
            t0size = UInt(TTBCR.T0SZ);
            if t0size == 0 || IsZero(inputaddr<31:(32-t0size)>) then
                inputsize = 32 - t0size;
                basefound = TRUE;
                baseregister = TTBR0;
                descaddr.memattrs = WalkAttrDecode(TTBCR.SH0, TTBCR.ORGN0, TTBCR.IRGN0, secondstage);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            hierattrsdisabled = AArch32.HaveHPDExt() && TTBCR.T2E == '1' &&  TTBCR2.HPD0 == '1';
        t1size = UInt(TTBCR.T1SZ);
        if (t1size == 0 && !basefound) || (t1size > 0 && IsOnes(inputaddr<31:(32-t1size)>)) then
            inputsize = 32 - t1size;
            basefound = TRUE;
            baseregister = TTBR1;
            descaddr.memattrs = WalkAttrDecode(TTBCR.SH1, TTBCR.ORGN1, TTBCR.IRGN1, secondstage);
            hierattrsdisabled = AArch32.HaveHPDExt() && TTBCR.T2E == '1' &&  TTBCR2.HPD1 == '1';
        reversedescriptors = SCTLR.EE == '1';
        lookupsecure = IsSecure();
        singlepriv = FALSE;
    // The starting level is the number of strides needed to consume the input address
    level = 4 - (1 + ((inputsize - grainsize - 1) DIV stride));

else
    // Second stage translation
    inputaddr = ipaddress;
    inputsize = 32 - SInt(VTCR.T0SZ);
    // VTCR.S must match VTCR.T0SZ[3]
    if VTCR.S != VTCR.T0SZ<3> then
        (-, inputsize) = ConstrainUnpredictableInteger(32-7, 32+8);
    basefound = inputsize == 40 || IsZero(inputaddr<39:inputsize>);
    disabled = FALSE;
    descaddr.memattrs = WalkAttrDecode(VTCR.SH0, VTCR.ORGN0, VTCR.IRGN0, secondstage);
    reversedescriptors = HSCTLR.EE == '1';
    singlepriv = TRUE;

    lookupsecure = FALSE;
    baseregister = VTTBR;
    startlevel = UInt(VTCR.SL0);
    level = 2 - startlevel;
    if level <= 0 then basefound = FALSE;

    // Number of entries in the starting level table =
    //     (Size of Input Address)/((Address per level)^(Num levels remaining)*(Size of Table))
    startsizecheck = inputsize - ((3 - level)*stride + grainsize); // Log2(Num of entries)

    // Check for starting level table with fewer than 2 entries or longer than 16 pages.
    // Lower bound check is:  startsizecheck < Log2(2 entries)
    //   That is, VTCR.SL0 == '00' and SInt(VTCR.T0SZ) > 1, Size of Input Address < 2^31 bytes
    // Upper bound check is:  startsizecheck > Log2(pagesize/8*16)
    //   That is, VTCR.SL0 == '01' and SInt(VTCR.T0SZ) < -2, Size of Input Address > 2^34 bytes
    if startsizecheck < 1 || startsizecheck > stride + 4 then basefound = FALSE;
if !basefound || disabled then
    level = 1;           // AArch64 reports this as a level 0 fault
    result.addrdesc.fault = AArch32.TranslationFault(ipaddress,   domain, level, acctype, iswrite,
                                                     secondstage, s2fs1walk);
    return result;

if !IsZero(baseregister<47:40>) then
    level = 0;
    result.addrdesc.fault = AArch32.AddressSizeFault(ipaddress,  domain, level, acctype, iswrite,
                                                     secondstage, s2fs1walk);
    return result;

// Bottom bound of the Base address is:
//     Log2(8 bytes per entry)+Log2(Number of entries in starting level table)
// Number of entries in starting level table =
//     (Size of Input Address)/((Address per level)^(Num levels remaining)*(Size of Table))
baselowerbound = 3 + inputsize - ((3-level)*stride + grainsize);  // Log2(Num of entries*8)
baseaddress = baseregister<39:baselowerbound>:Zeros(baselowerbound);

ns_table = if lookupsecure then '0' else '1';
ap_table = '00';
xn_table = '0';
pxn_table = '0';

addrselecttop = inputsize - 1;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
repeat
    addrselectbottom = (3-level)*stride + grainsize;

    bits(40) index = ZeroExtend(inputaddr<addrselecttop:addrselectbottom>:'000');
    descaddr.paddress.address = ZeroExtend(baseaddress OR index);
    descaddr.paddress.NS = ns_table;

    // If there are two stages of translation, then the first stage table walk addresses
    // are themselves subject to translation
    if secondstage || !HasS2Translation() || (HaveNV2Ext() && acctype == AccType_NV2REGISTER) then
        descaddr2 = descaddr;
    else
        descaddr2 = AArch32.SecondStageWalk(descaddr, vaddress, acctype, iswrite, 8);
        // Check for a fault on the stage 2 walk
        if IsFault(descaddr2) then
            result.addrdesc.fault = descaddr2.fault;
            return result;

    // Update virtual address for abort functions
    descaddr2.vaddress = ZeroExtend(vaddress);

    accdesc = CreateAccessDescriptorPTW(acctype, secondstage, s2fs1walk, level);
    desc = _Mem[descaddr2, 8, accdesc];

    if reversedescriptors then desc = BigEndianReverse(desc);

    if desc<0> == '0' || (desc<1:0> == '01' && level == 3) then
         // Fault (00), Reserved (10), or Block (01) at level 3.
        result.addrdesc.fault = AArch32.TranslationFault(ipaddress,  domain, level, acctype,
                                                         iswrite, secondstage, s2fs1walk);
        return result;

    // Valid Block, Page, or Table entry
    if desc<1:0> == '01' || level == 3 then                 // Block (01) or Page (11)
        blocktranslate = TRUE;
    else                                                    // Table (11)
        if !IsZero(desc<47:40>) then
            result.addrdesc.fault = AArch32.AddressSizeFault(ipaddress,  domain, level, acctype,
                                                             iswrite, secondstage, s2fs1walk);
            return result;

        baseaddress = desc<39:grainsize>:Zeros(grainsize);
        if !secondstage then
            // Unpack the upper and lower table attributes
            ns_table    = ns_table    OR desc<63>;
        if !secondstage && !hierattrsdisabled then
            ap_table<1> = ap_table<1> OR desc<62>;       // read-only

            xn_table    = xn_table    OR desc<60>;
            // pxn_table and ap_table[0] apply only in EL1&0 translation regimes
            if !singlepriv then
                pxn_table   = pxn_table   OR desc<59>;
                ap_table<0> = ap_table<0> OR desc<61>;   // privileged

        level = level + 1;
        addrselecttop = addrselectbottom - 1;
        blocktranslate = FALSE;
until blocktranslate;

// Unpack the descriptor into address and upper and lower block attributes
outputaddress = desc<39:addrselectbottom>:inputaddr<addrselectbottom-1:0>;

// Check the output address is inside the supported range
if !IsZero(desc<47:40>) then
    result.addrdesc.fault = AArch32.AddressSizeFault(ipaddress,  domain, level, acctype,
                                                     iswrite, secondstage, s2fs1walk);
    return result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    // Check the access flag
    if desc<10> == '0' then
        result.addrdesc.fault = AArch32.AccessFlagFault(ipaddress,  domain, level, acctype,
                                                        iswrite, secondstage, s2fs1walk);
        return result;
    xn = desc<54>;                                          // Bit[54] of the block/page descriptor
holds UXN
    pxn = desc<53>;                                         // Bit[53] of the block/page descriptor
holds PXN
    ap = desc<7:6>:'1';                                     // Bits[7:6] of the block/page descriptor
hold AP[2:1]
    contiguousbit = desc<52>;
    nG = desc<11>;
    sh = desc<9:8>;
    memattr = desc<5:2>;                                    // AttrIndx and NS bit in stage 1

    result.domain = bits(4) UNKNOWN;                        // Domains not used
    result.level = level;
    result.blocksize = 2^((3-level)*stride + grainsize);

    // Stage 1 translation regimes also inherit attributes from the tables
    if !secondstage then
        result.perms.xn      = xn OR xn_table;
        result.perms.ap<2>   = ap<2> OR ap_table<1>;        // Force read-only
        // PXN, nG and AP[1] apply only in EL1&0 stage 1 translation regimes
        if !singlepriv then
            result.perms.ap<1> = ap<1> AND NOT(ap_table<0>);  // Force privileged only
            result.perms.pxn   = pxn OR pxn_table;
            // Pages from Non-secure tables are marked non-global in Secure EL1&0
            if IsSecure() then
                result.nG = nG OR ns_table;
            else
                result.nG = nG;
        else
            result.perms.ap<1> = '1';
            result.perms.pxn   = '0';
            result.nG          = '0';
        result.GP = desc<50>;                               // Stage 1 block or pages might be guarded
        result.perms.ap<0>   = '1';
        result.addrdesc.memattrs = AArch32.S1AttrDecode(sh, memattr<2:0>, acctype);
        result.addrdesc.paddress.NS = memattr<3> OR ns_table;
    else
        result.perms.ap<2:1> = ap<2:1>;
        result.perms.ap<0>   = '1';
        result.perms.xn      = xn;
        if HaveExtendedExecuteNeverExt() then result.perms.xxn = desc<53>;
        result.perms.pxn     = '0';
        result.nG            = '0';
        if s2fs1walk then
            result.addrdesc.memattrs = S2AttrDecode(sh, memattr, AccType_PTW);
        else
            result.addrdesc.memattrs = S2AttrDecode(sh, memattr, acctype);
        result.addrdesc.paddress.NS = '1';

    result.addrdesc.paddress.address = ZeroExtend(outputaddress);
    result.addrdesc.fault = AArch32.NoFault();
    result.contiguous = contiguousbit == '1';
    if HaveCommonNotPrivateTransExt() then result.CnP = baseregister<0>;

    return result;
```

aarch32/translation/walk/AArch32.TranslationTableWalkSD

```
// AArch32.TranslationTableWalkSD()
// ================================
// Returns a result of a translation table walk using the Short-descriptor format
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
//
// Implementations might cache information from memory in any number of non-coherent TLB
// caching structures, and so avoid memory accesses that have been expressed in this
// pseudocode. The use of such TLBs is not expressed in this pseudocode.

TLBRecord AArch32.TranslationTableWalkSD(bits(32) vaddress, AccType acctype, boolean iswrite,
                                         integer size)
    assert ELUsingAArch32(S1TranslationRegime());

    // This is only called when address translation is enabled
    TLBRecord         result;
    AddressDescriptor l1descaddr;
    AddressDescriptor l2descaddr;
    bits(40)      outputaddress;

    // Variables for Abort functions
    ipaddress = bits(40) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;
    NS = bit UNKNOWN;

    // Default setting of the domain
    domain = bits(4) UNKNOWN;

    // Determine correct Translation Table Base Register to use.
    bits(64) ttbr;
    n = UInt(TTBCR.N);
    if n == 0 || IsZero(vaddress<31:(32-n)>) then
        ttbr = TTBR0;
        disabled = (TTBCR.PD0 == '1');
    else
        ttbr = TTBR1;
        disabled = (TTBCR.PD1 == '1');
        n = 0;  // TTBR1 translation always works like N=0 TTBR0 translation

    // Check this Translation Table Base Register is not disabled.
    if disabled then
        level = 1;
        result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype, iswrite,
                                                         secondstage, s2fs1walk);
        return result;

    // Obtain descriptor from initial lookup.
    l1descaddr.paddress.address = ZeroExtend(ttbr<31:14-n>:vaddress<31-n:20>:'00');
    l1descaddr.paddress.NS = if IsSecure() then '0' else '1';
    IRGN = ttbr<0>:ttbr<6>;             // TTBR.IRGN
    RGN = ttbr<4:3>;                    // TTBR.RGN
    SH = ttbr<1>:ttbr<5>;               // TTBR.S:TTBR.NOS
    l1descaddr.memattrs = WalkAttrDecode(SH, RGN, IRGN, secondstage);

    if !HaveEL(EL2) || (IsSecure() && !IsSecureEL2Enabled()) then
        // if only 1 stage of translation
        l1descaddr2 = l1descaddr;
    else
        l1descaddr2 = AArch32.SecondStageWalk(l1descaddr, vaddress, acctype, iswrite, 4);
        // Check for a fault on the stage 2 walk
        if IsFault(l1descaddr2) then
            result.addrdesc.fault = l1descaddr2.fault;
            return result;

    // Update virtual address for abort functions
    l1descaddr2.vaddress = ZeroExtend(vaddress);

    accdesc = CreateAccessDescriptorPTW(acctype, secondstage, s2fs1walk, level);
    l1desc = _Mem[l1descaddr2, 4,accdesc];

    if SCTLR.EE == '1' then l1desc = BigEndianReverse(l1desc);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
// Process descriptor from initial lookup.
case l1desc<1:0> of
    when '00'                                           // Fault, Reserved
        level = 1;
        result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype,
                                                         iswrite, secondstage, s2fs1walk);
        return result;

    when '01'                                           // Large page or Small page
        domain = l1desc<8:5>;
        level = 2;
        pxn = l1desc<2>;
        NS = l1desc<3>;

        // Obtain descriptor from level 2 lookup.
        l2descaddr.paddress.address = ZeroExtend(l1desc<31:10>:vaddress<19:12>:'00');
        l2descaddr.paddress.NS = if IsSecure() then '0' else '1';
        l2descaddr.memattrs = l1descaddr.memattrs;

        if !HaveEL(EL2) || (IsSecure() && !IsSecureEL2Enabled()) then
            // if only 1 stage of translation
            l2descaddr2 = l2descaddr;
        else
            l2descaddr2 = AArch32.SecondStageWalk(l2descaddr, vaddress, acctype, iswrite, 4);
            // Check for a fault on the stage 2 walk
            if IsFault(l2descaddr2) then
                result.addrdesc.fault = l2descaddr2.fault;
                return result;

        // Update virtual address for abort functions
        l2descaddr2.vaddress = ZeroExtend(vaddress);

        accdesc = CreateAccessDescriptorPTW(acctype, secondstage, s2fs1walk, level);
        l2desc = _Mem[l2descaddr2, 4, accdesc];

        if SCTLR.EE == '1' then l2desc = BigEndianReverse(l2desc);

        // Process descriptor from level 2 lookup.
        if l2desc<1:0> == '00' then
            result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype,
                                                             iswrite, secondstage, s2fs1walk);
            return result;

        nG = l2desc<11>;
        S = l2desc<10>;
        ap = l2desc<9,5:4>;

        if SCTLR.AFE == '1' && l2desc<4> == '0' then
            // Armv8 VMSAv8-32 does not support hardware management of the Access flag.
            result.addrdesc.fault = AArch32.AccessFlagFault(ipaddress, domain, level, acctype,
                                                            iswrite, secondstage, s2fs1walk);
            return result;

        if l2desc<1> == '0' then                        // Large page
            xn = l2desc<15>;
            tex = l2desc<14:12>;
            c = l2desc<3>;
            b = l2desc<2>;
            blocksize = 64;
            outputaddress = ZeroExtend(l2desc<31:16>:vaddress<15:0>);
        else                                            // Small page
            tex = l2desc<8:6>;
            c = l2desc<3>;
            b = l2desc<2>;
            xn = l2desc<0>;
            blocksize = 4;
            outputaddress = ZeroExtend(l2desc<31:12>:vaddress<11:0>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        when '1x'                                           // Section or Supersection
            NS = l1desc<19>;
            nG = l1desc<17>;
            S = l1desc<16>;
            ap = l1desc<15,11:10>;
            tex = l1desc<14:12>;
            xn = l1desc<4>;
            c = l1desc<3>;
            b = l1desc<2>;
            pxn = l1desc<0>;
            level = 1;

            if SCTLR.AFE == '1' && l1desc<10> == '0' then
                // Armv8 VMSAv8-32 does not support hardware management of the Access flag.
                result.addrdesc.fault = AArch32.AccessFlagFault(ipaddress, domain, level, acctype,
                                                                iswrite, secondstage, s2fs1walk);
                return result;

            if l1desc<18> == '0' then                       // Section
                domain = l1desc<8:5>;
                blocksize = 1024;
                outputaddress = ZeroExtend(l1desc<31:20>:vaddress<19:0>);
            else                                            // Supersection
                domain = '0000';
                blocksize = 16384;
                outputaddress = l1desc<8:5>:l1desc<23:20>:l1desc<31:24>:vaddress<23:0>;

    // Decode the TEX, C, B and S bits to produce the TLBRecord's memory attributes
    if SCTLR.TRE == '0' then
        if RemapRegsHaveResetValues() then
            result.addrdesc.memattrs = AArch32.DefaultTEXDecode(tex, c, b, S, acctype);
        else
            result.addrdesc.memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;
    else
        result.addrdesc.memattrs = AArch32.RemappedTEXDecode(tex, c, b, S, acctype);

    // Set the rest of the TLBRecord, try to add it to the TLB, and return it.
    result.perms.ap = ap;
    result.perms.xn = xn;
    result.perms.pxn = pxn;
    result.nG = nG;
    result.domain = domain;
    result.level = level;
    result.blocksize = blocksize;
    result.addrdesc.paddress.address = ZeroExtend(outputaddress);
    result.addrdesc.paddress.NS = if IsSecure() then NS else '1';
    result.addrdesc.fault = AArch32.NoFault();

    return result;
```

aarch32/translation/walk/RemapRegsHaveResetValues

```
boolean RemapRegsHaveResetValues();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

J1.3 Shared pseudocode

This section holds the pseudocode that is common to execution in AArch64 state and in AArch32 state. Functions listed in this section are identified only by a `FunctionName`, without an `AArch64.` or `AArch32.` prefix. This section is organized by functional groups, with the functional groups being indicated by hierarchical path names, for example `shared/debug/DebugTarget`.

The top-level sections of the shared pseudocode hierarchy are:

- *shared/debug*.
- *shared/exceptions* on page J1-7725.
- *shared/functions* on page J1-7727.
- *shared/trace* on page J1-7818.
- *shared/translation* on page J1-7820.

J1.3.1 shared/debug

This section includes the following pseudocode functions:

- *shared/debug/ClearStickyErrors/ClearStickyErrors* on page J1-7706.
- *shared/debug/DebugTarget/DebugTarget* on page J1-7706.
- *shared/debug/DebugTarget/DebugTargetFrom* on page J1-7707.
- *shared/debug/DoubleLockStatus/DoubleLockStatus* on page J1-7707.
- *shared/debug/authentication/AllowExternalDebugAccess* on page J1-7707.
- *shared/debug/authentication/AllowExternalPMUAccess* on page J1-7708.
- *shared/debug/authentication/Debug_authentication* on page J1-7708.
- *shared/debug/authentication/ExternalInvasiveDebugEnabled* on page J1-7708.
- *shared/debug/authentication/ExternalNoninvasiveDebugAllowed* on page J1-7708.
- *shared/debug/authentication/ExternalNoninvasiveDebugEnabled* on page J1-7709.
- *shared/debug/authentication/ExternalSecureInvasiveDebugEnabled* on page J1-7709.
- *shared/debug/authentication/ExternalSecureNoninvasiveDebugEnabled* on page J1-7709.
- *shared/debug/authentication/IsAccessSecure* on page J1-7709.
- *shared/debug/authentication/IsCorePowered* on page J1-7709.
- *shared/debug/breakpoint/CheckValidStateMatch* on page J1-7709.
- *shared/debug/cti/CTI_SetEventLevel* on page J1-7710.
- *shared/debug/cti/CTI_SignalEvent* on page J1-7710.
- *shared/debug/cti/CrossTrigger* on page J1-7710.
- *shared/debug/dccanditr/CheckForDCCInterrupts* on page J1-7711.
- *shared/debug/dccanditr/DBGDTRRX_EL0* on page J1-7711.
- *shared/debug/dccanditr/DBGDTRTX_EL0* on page J1-7712.
- *shared/debug/dccanditr/DBGDTR_EL0* on page J1-7713.
- *shared/debug/dccanditr/DTR* on page J1-7713.
- *shared/debug/dccanditr/EDITR* on page J1-7713.
- *shared/debug/halting/DCPSInstruction* on page J1-7714.
- *shared/debug/halting/DRPSInstruction* on page J1-7715.
- *shared/debug/halting/DebugHalt* on page J1-7716.
- *shared/debug/halting/DisableITRAndResumeInstructionPrefetch* on page J1-7716.
- *shared/debug/halting/ExecuteA64* on page J1-7716.
- *shared/debug/halting/ExecuteT32* on page J1-7716.
- *shared/debug/halting/ExitDebugState* on page J1-7716.
- *shared/debug/halting/Halt* on page J1-7717.
- *shared/debug/halting/HaltOnBreakpointOrWatchpoint* on page J1-7718.
- *shared/debug/halting/Halted* on page J1-7718.
- *shared/debug/halting/HaltingAllowed* on page J1-7718.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *shared/debug/halting/Restarting* on page J1-7718.
- *shared/debug/halting/StopInstructionPrefetchAndEnableITR* on page J1-7718.
- *shared/debug/halting/UpdateEDSCRFields* on page J1-7718.
- *shared/debug/haltingevents/CheckExceptionCatch* on page J1-7719.
- *shared/debug/haltingevents/CheckHaltingStep* on page J1-7719.
- *shared/debug/haltingevents/CheckOSUnlockCatch* on page J1-7719.
- *shared/debug/haltingevents/CheckPendingOSUnlockCatch* on page J1-7720.
- *shared/debug/haltingevents/CheckPendingResetCatch* on page J1-7720.
- *shared/debug/haltingevents/CheckResetCatch* on page J1-7720.
- *shared/debug/haltingevents/CheckSoftwareAccessToDebugRegisters* on page J1-7720.
- *shared/debug/haltingevents/ExternalDebugRequest* on page J1-7720.
- *shared/debug/haltingevents/HaltingStep_DidNotStep* on page J1-7720.
- *shared/debug/haltingevents/HaltingStep_SteppedEX* on page J1-7721.
- *shared/debug/haltingevents/RunHaltingStep* on page J1-7721.
- *shared/debug/interrupts/ExternalDebugInterruptsDisabled* on page J1-7721.
- *shared/debug/interrupts/InterruptID* on page J1-7721.
- *shared/debug/interrupts/SetInterruptRequestLevel* on page J1-7721.
- *shared/debug/samplebasedprofiling/CreatePCSample* on page J1-7722.
- *shared/debug/samplebasedprofiling/EDPCSRlo* on page J1-7722.
- *shared/debug/samplebasedprofiling/PCSample* on page J1-7723.
- *shared/debug/samplebasedprofiling/PMPCSR* on page J1-7723.
- *shared/debug/softwarestep/CheckSoftwareStep* on page J1-7724.
- *shared/debug/softwarestep/DebugExceptionReturnSS* on page J1-7724.
- *shared/debug/softwarestep/SSAdvance* on page J1-7724.
- *shared/debug/softwarestep/SoftwareStep_DidNotStep* on page J1-7725.
- *shared/debug/softwarestep/SoftwareStep_SteppedEX* on page J1-7725.

shared/debug/ClearStickyErrors/ClearStickyErrors

```
// ClearStickyErrors()
// ===================

ClearStickyErrors()
    EDSCR.TXU = '0';            // Clear TX underrun flag
    EDSCR.RXO = '0';            // Clear RX overrun flag

    if Halted() then            // in Debug state
        EDSCR.ITO = '0';        // Clear ITR overrun flag

    // If halted and the ITR is not empty then it is UNPREDICTABLE whether the EDSCR.ERR is cleared.
    // The UNPREDICTABLE behavior also affects the instructions in flight, but this is not described
    // in the pseudocode.
    if Halted() && EDSCR.ITE == '0' && ConstrainUnpredictableBool() then
        return;
    EDSCR.ERR = '0';            // Clear cumulative error flag

    return;
```

shared/debug/DebugTarget/DebugTarget

```
// DebugTarget()
// =============
// Returns the debug exception target Exception level

bits(2) DebugTarget()
    secure = IsSecure();
    return DebugTargetFrom(secure);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/debug/DebugTarget/DebugTargetFrom

```
// DebugTargetFrom()
// =================

bits(2) DebugTargetFrom(boolean secure)
    if HaveEL(EL2) && (!secure || (HaveSecureEL2Ext() &&
            (!HaveEL(EL3) ||SCR_EL3.EEL2 == '1'))) then
        if ELUsingAArch32(EL2) then
            route_to_el2 = (HDCR.TDE == '1' || HCR.TGE == '1');
        else
            route_to_el2 = (MDCR_EL2.TDE == '1' || HCR_EL2.TGE == '1');
    else
        route_to_el2 = FALSE;

    if route_to_el2 then
        target = EL2;
    elsif HaveEL(EL3) && HighestELUsingAArch32() && secure then
        target = EL3;
    else
        target = EL1;

    return target;
```

shared/debug/DoubleLockStatus/DoubleLockStatus

```
// DoubleLockStatus()
// ==================
// Returns the state of the OS Double Lock.
//    FALSE if OSDLR_EL1.DLK == 0 or DBGPRCR_EL1.CORENPDRQ == 1 or the PE is in Debug state.
//    TRUE if OSDLR_EL1.DLK == 1 and DBGPRCR_EL1.CORENPDRQ == 0 and the PE is in Non-debug state.

boolean DoubleLockStatus()
    if !HaveDoubleLock() then
        return FALSE;
    elsif ELUsingAArch32(EL1) then
        return DBGOSDLR.DLK == '1' && DBGPRCR.CORENPDRQ == '0' && !Halted();
    else
        return OSDLR_EL1.DLK == '1' && DBGPRCR_EL1.CORENPDRQ == '0' && !Halted();
```

shared/debug/authentication/AllowExternalDebugAccess

```
// AllowExternalDebugAccess()
// ==========================
// Returns TRUE if an external debug interface access to the External debug registers
// is allowed, FALSE otherwise.

boolean AllowExternalDebugAccess()
    // The access may also be subject to OS Lock, power-down, etc.
    if HaveSecureExtDebugView() then
        return AllowExternalDebugAccess(IsAccessSecure());
    else
        return AllowExternalDebugAccess(ExternalSecureInvasiveDebugEnabled());

// AllowExternalDebugAccess()
// ==========================
// Returns TRUE if an external debug interface access to the External debug registers
// is allowed for the given Security state, FALSE otherwise.

boolean AllowExternalDebugAccess(boolean allow_secure)
    // The access may also be subject to OS Lock, power-down, etc.
    if HaveSecureExtDebugView() || ExternalInvasiveDebugEnabled() then
        if allow_secure then
            return TRUE;
        elsif HaveEL(EL3) then
            if ELUsingAArch32(EL3) then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
                return SDCR.EDAD == '0';
            else
                return MDCR_EL3.EDAD == '0';
        else
            return !IsSecure();
    else
        return FALSE;
```

shared/debug/authentication/AllowExternalPMUAccess

```
// AllowExternalPMUAccess()
// ========================
// Returns TRUE if an external debug interface access to the PMU registers is allowed, FALSE otherwise.

boolean AllowExternalPMUAccess()
    // The access may also be subject to OS Lock, power-down, etc.
    if HaveSecureExtDebugView() then
        return AllowExternalPMUAccess(IsAccessSecure());
    else
        return AllowExternalPMUAccess(ExternalSecureNoninvasiveDebugEnabled());

// AllowExternalPMUAccess()
// ========================
// Returns TRUE if an external debug interface access to the PMU registers is allowed for the given
// Security state, FALSE otherwise.

boolean AllowExternalPMUAccess(boolean allow_secure)
    // The access may also be subject to OS Lock, power-down, etc.
    if HaveSecureExtDebugView() || ExternalNoninvasiveDebugEnabled() then
        if allow_secure then
            return TRUE;
        elsif HaveEL(EL3) then
            if ELUsingAArch32(EL3) then
                return SDCR.EPMAD == '0';
            else
                return MDCR_EL3.EPMAD == '0';
        else
            return !IsSecure();
    else
        return FALSE;
```

shared/debug/authentication/Debug_authentication

```
signal DBGEN;
signal NIDEN;
signal SPIDEN;
signal SPNIDEN;
```

shared/debug/authentication/ExternalInvasiveDebugEnabled

```
// ExternalInvasiveDebugEnabled()
// ==============================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the DBGEN signal.

boolean ExternalInvasiveDebugEnabled()
    return DBGEN == HIGH;
```

shared/debug/authentication/ExternalNoninvasiveDebugAllowed

```
// ExternalNoninvasiveDebugAllowed()
// =================================
// Returns TRUE if Trace and PC Sample-based Profiling are allowed

boolean ExternalNoninvasiveDebugAllowed()
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    return (ExternalNoninvasiveDebugEnabled() &&
            (!IsSecure() || ExternalSecureNoninvasiveDebugEnabled() ||
             (ELUsingAArch32(EL1) && PSTATE.EL == EL0 && SDER.SUNIDEN == '1')));
```

shared/debug/authentication/ExternalNoninvasiveDebugEnabled

```
// ExternalNoninvasiveDebugEnabled()
// =================================
// This function returns TRUE if the ARMv8.4-Debug is implemented, otherwise this
// function is IMPLEMENTATION DEFINED.
// In the recommended interface, ExternalNoninvasiveDebugEnabled returns the state of the (DBGEN
// OR NIDEN) signal.

boolean ExternalNoninvasiveDebugEnabled()
    return !HaveNoninvasiveDebugAuth() || ExternalInvasiveDebugEnabled() || NIDEN == HIGH;
```

shared/debug/authentication/ExternalSecureInvasiveDebugEnabled

```
// ExternalSecureInvasiveDebugEnabled()
// ====================================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the (DBGEN AND SPIDEN) signal.
// CoreSight allows asserting SPIDEN without also asserting DBGEN, but this is not recommended.

boolean ExternalSecureInvasiveDebugEnabled()
    if !HaveEL(EL3) && !IsSecure() then return FALSE;
    return ExternalInvasiveDebugEnabled() && SPIDEN == HIGH;
```

shared/debug/authentication/ExternalSecureNoninvasiveDebugEnabled

```
// ExternalSecureNoninvasiveDebugEnabled()
// =======================================
// This function returns the value of ExternalSecureInvasiveDebugEnabled() when ARMv8.4-Debug
// is implemented. Otherwise, the definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the (DBGEN OR NIDEN) AND
// (SPIDEN OR SPNIDEN) signal.

boolean ExternalSecureNoninvasiveDebugEnabled()
    if !HaveEL(EL3) && !IsSecure() then return FALSE;
    if HaveNoninvasiveDebugAuth() then
        return ExternalNoninvasiveDebugEnabled() && (SPIDEN == HIGH || SPNIDEN == HIGH);
    else
        return ExternalSecureInvasiveDebugEnabled();
```

shared/debug/authentication/IsAccessSecure

```
// Returns TRUE when an access is Secure
boolean IsAccessSecure();
```

shared/debug/authentication/IsCorePowered

```
// Returns TRUE if the Core power domain is powered on, FALSE otherwise.
boolean IsCorePowered();
```

shared/debug/breakpoint/CheckValidStateMatch

```
// CheckValidStateMatch()
// ======================
// Checks for an invalid state match that will generate Constrained Unpredictable behaviour, otherwise
// returns Constraint_NONE.

(Constraint, bits(2), bit, bits(2)) CheckValidStateMatch(bits(2) SSC, bit HMC, bits(2) PxC, boolean
isbreakpnt)
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
boolean reserved = FALSE;

// Match 'Usr/Sys/Svc' only valid for AArch32 breakpoints
if (!isbreakpnt || !HaveAArch32EL(EL1)) && HMC:PxC == '000' && SSC != '11' then
    reserved = TRUE;

// Both EL3 and EL2 are not implemented
if !HaveEL(EL3) && !HaveEL(EL2) && (HMC != '0' || SSC != '00') then
    reserved = TRUE;

// EL3 is not implemented
if !HaveEL(EL3) && SSC IN {'01','10'} && HMC:SSC:PxC != '10100' then
    reserved = TRUE;

// EL3 using AArch64 only
if (!HaveEL(EL3) || HighestELUsingAArch32()) && HMC:SSC:PxC == '11000' then
    reserved = TRUE;

// EL2 is not implemented
if !HaveEL(EL2) && HMC:SSC:PxC == '11100' then
    reserved = TRUE;

// Secure EL2 is not implemented
if !HaveSecureEL2Ext() && (HMC:SSC:PxC)  IN {'01100','10100','x11x1'} then
    reserved = TRUE;

// Values that are not allocated in any architecture version
if (HMC:SSC:PxC) IN {'01110','100x0','10110','11x10'} then
    reserved = TRUE;

if reserved then
    // If parameters are set to a reserved type, behaves as either disabled or a defined type
    (c, <HMC,SSC,PxC>) = ConstrainUnpredictableBits();
    assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
    if c == Constraint_DISABLED then
        return (c, bits(2) UNKNOWN, bit UNKNOWN, bits(2) UNKNOWN);
    // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

return (Constraint_NONE, SSC, HMC, PxC);
```

shared/debug/cti/CTI_SetEventLevel

```
// Set a Cross Trigger multi-cycle input event trigger to the specified level.
CTI_SetEventLevel(CrossTriggerIn id, signal level);
```

shared/debug/cti/CTI_SignalEvent

```
// Signal a discrete event on a Cross Trigger input event trigger.
CTI_SignalEvent(CrossTriggerIn id);
```

shared/debug/cti/CrossTrigger

```
enumeration CrossTriggerOut {CrossTriggerOut_DebugRequest, CrossTriggerOut_RestartRequest,
                             CrossTriggerOut_IRQ,          CrossTriggerOut_RSVD3,
                             CrossTriggerOut_TraceExtIn0,  CrossTriggerOut_TraceExtIn1,
                             CrossTriggerOut_TraceExtIn2,  CrossTriggerOut_TraceExtIn3};

enumeration CrossTriggerIn  {CrossTriggerIn_CrossHalt,     CrossTriggerIn_PMUOverflow,
                             CrossTriggerIn_RSVD2,         CrossTriggerIn_RSVD3,
                             CrossTriggerIn_TraceExtOut0,  CrossTriggerIn_TraceExtOut1,
                             CrossTriggerIn_TraceExtOut2,  CrossTriggerIn_TraceExtOut3};
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/debug/dccanditr/CheckForDCCInterrupts

```
// CheckForDCCInterrupts()
// =======================

CheckForDCCInterrupts()
    commrx = (EDSCR.RXfull == '1');
    commtx = (EDSCR.TXfull == '0');

    // COMMRX and COMMTX support is optional and not recommended for new designs.
    // SetInterruptRequestLevel(InterruptID_COMMRX, if commrx then HIGH else LOW);
    // SetInterruptRequestLevel(InterruptID_COMMTX, if commtx then HIGH else LOW);

    // The value to be driven onto the common COMMIRQ signal.
    if ELUsingAArch32(EL1) then
        commirq = ((commrx && DBGDCCINT.RX == '1') ||
                   (commtx && DBGDCCINT.TX == '1'));
    else
        commirq = ((commrx && MDCCINT_EL1.RX == '1') ||
                   (commtx && MDCCINT_EL1.TX == '1'));
    SetInterruptRequestLevel(InterruptID_COMMIRQ, if commirq then HIGH else LOW);

    return;
```

shared/debug/dccanditr/DBGDTRRX_EL0

```
// DBGDTRRX_EL0[] (external write)
// ===============================
// Called on writes to debug register 0x08C.

DBGDTRRX_EL0[boolean memory_mapped] = bits(32) value

    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "signal slave-generated error";
        return;

    if EDSCR.ERR == '1' then return;                    // Error flag set: ignore write

    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then return;   // Software lock locked: ignore write

    if EDSCR.RXfull == '1' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0') then
        EDSCR.RXO = '1';  EDSCR.ERR = '1';              // Overrun condition: ignore write
        return;

    EDSCR.RXfull = '1';
    DTRRX = value;

    if Halted() && EDSCR.MA == '1' then
        EDSCR.ITE = '0';                            // See comments in EDITR[] (external write)
        if !UsingAArch32() then
            ExecuteA64(0xD5330501<31:0>);               // A64 "MRS X1,DBGDTRRX_EL0"
            ExecuteA64(0xB8004401<31:0>);               // A64 "STR W1,[X0],#4"
            X[1] = bits(64) UNKNOWN;
        else
            ExecuteT32(0xEE10<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/);  // T32 "MRS R1,DBGDTRRXint"
            ExecuteT32(0xF840<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/);  // T32 "STR R1,[R0],#4"
            R[1] = bits(32) UNKNOWN;
        // If the store aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
        if EDSCR.ERR == '1' then
            EDSCR.RXfull = bit UNKNOWN;
            DBGDTRRX_EL0 = bits(32) UNKNOWN;
        else
            // "MRS X1,DBGDTRRX_EL0" calls DBGDTR_EL0[] (read) which clears RXfull.
            assert EDSCR.RXfull == '0';

        EDSCR.ITE = '1';                                // See comments in EDITR[] (external write)
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    return;

// DBGDTRRX_EL0[] (external read)
// ==============================

bits(32) DBGDTRRX_EL0[boolean memory_mapped]
    return DTRRX;
```

shared/debug/dccanditr/DBGDTRTX_EL0

```
// DBGDTRTX_EL0[] (external read)
// ==============================
// Called on reads of debug register 0x080.

bits(32) DBGDTRTX_EL0[boolean memory_mapped]

    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "signal slave-generated error";
        return bits(32) UNKNOWN;

    underrun = EDSCR.TXfull == '0' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0');
    value = if underrun then bits(32) UNKNOWN else DTRTX;

    if EDSCR.ERR == '1' then return value;              // Error flag set: no side-effects

    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then           // Software lock locked: no side-effects
        return value;

    if underrun then
        EDSCR.TXU = '1';  EDSCR.ERR = '1';              // Underrun condition: block side-effects
        return value;                                   // Return UNKNOWN

    EDSCR.TXfull = '0';
    if Halted() && EDSCR.MA == '1' then
        EDSCR.ITE = '0';                                // See comments in EDITR[] (external write)

        if !UsingAArch32() then
            ExecuteA64(0xB8404401<31:0>);               // A64 "LDR W1,[X0],#4"
        else
            ExecuteT32(0xF850<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/);      // T32 "LDR R1,[R0],#4"
        // If the load aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
        if EDSCR.ERR == '1' then
            EDSCR.TXfull = bit UNKNOWN;
            DBGDTRTX_EL0 = bits(32) UNKNOWN;
        else
            if !UsingAArch32() then
                ExecuteA64(0xD5130501<31:0>);           // A64 "MSR DBGDTRTX_EL0,X1"
            else
                ExecuteT32(0xEE00<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/);  // T32 "MSR DBGDTRTXint,R1"
            // "MSR DBGDTRTX_EL0,X1" calls DBGDTR_EL0[] (write) which sets TXfull.
            assert EDSCR.TXfull == '1';
        if !UsingAArch32() then
            X[1] = bits(64) UNKNOWN;
        else
            R[1] = bits(32) UNKNOWN;
        EDSCR.ITE = '1';                                // See comments in EDITR[] (external write)

    return value;

// DBGDTRTX_EL0[] (external write)
// ==============================

DBGDTRTX_EL0[boolean memory_mapped] = bits(32) value
    // The Software lock is OPTIONAL.
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if memory_mapped && EDLSR.SLK == '1' then return;   // Software lock locked: ignore write
    DTRTX = value;
    return;
```

shared/debug/dccanditr/DBGDTR_EL0

```
// DBGDTR_EL0[] (write)
// ====================
// System register writes to DBGDTR_EL0, DBGDTRTX_EL0 (AArch64) and DBGDTRTXint (AArch32)

DBGDTR_EL0[] = bits(N) value
    // For MSR DBGDTRTX_EL0,<Rt>  N=32, value=X[t]<31:0>, X[t]<63:32> is ignored
    // For MSR DBGDTR_EL0,<Xt>    N=64, value=X[t]<63:0>
    assert N IN {32,64};
    if EDSCR.TXfull == '1' then
        value = bits(N) UNKNOWN;
    // On a 64-bit write, implement a half-duplex channel
    if N == 64 then DTRRX = value<63:32>;
    DTRTX = value<31:0>;        // 32-bit or 64-bit write
    EDSCR.TXfull = '1';
    return;

// DBGDTR_EL0[] (read)
// ===================
// System register reads of DBGDTR_EL0, DBGDTRRX_EL0 (AArch64) and DBGDTRRXint (AArch32)

bits(N) DBGDTR_EL0[]
    // For MRS <Rt>,DBGDTRTX_EL0  N=32, X[t]=Zeros(32):result
    // For MRS <Xt>,DBGDTR_EL0    N=64, X[t]=result
    assert N IN {32,64};
    bits(N) result;
    if EDSCR.RXfull == '0' then
        result = bits(N) UNKNOWN;
    else
        // On a 64-bit read, implement a half-duplex channel
        // NOTE: the word order is reversed on reads with regards to writes
        if N == 64 then result<63:32> = DTRTX;
        result<31:0> = DTRRX;
    EDSCR.RXfull = '0';
    return result;
```

shared/debug/dccanditr/DTR

```
bits(32) DTRRX;
bits(32) DTRTX;
```

shared/debug/dccanditr/EDITR

```
// EDITR[] (external write)
// ========================
// Called on writes to debug register 0x084.

EDITR[boolean memory_mapped] = bits(32) value
    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "signal slave-generated error";
        return;

    if EDSCR.ERR == '1' then return;                    // Error flag set: ignore write

    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then return;   // Software lock locked: ignore write

    if !Halted() then return;                           // Non-debug state: ignore write

    if EDSCR.ITE == '0' || EDSCR.MA == '1' then
        EDSCR.ITO = '1';  EDSCR.ERR = '1';              // Overrun condition: block write
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        return;

    // ITE indicates whether the processor is ready to accept another instruction; the processor
    // may support multiple outstanding instructions. Unlike the "InstrCompl" flag in [v7A] there
    // is no indication that the pipeline is empty (all instructions have completed). In this
    // pseudocode, the assumption is that only one instruction can be executed at a time,
    // meaning ITE acts like "InstrCompl".
    EDSCR.ITE = '0';

    if !UsingAArch32() then
        ExecuteA64(value);
    else
        ExecuteT32(value<15:0>/*hw1*/, value<31:16> /*hw2*/);

    EDSCR.ITE = '1';

    return;
```

shared/debug/halting/DCPSInstruction

```
// DCPSInstruction()
// =================
// Operation of the DCPS instruction in Debug state

DCPSInstruction(bits(2) target_el)

    SynchronizeContext();

    case target_el of
        when EL1
            if PSTATE.EL == EL2 || (PSTATE.EL == EL3 && !UsingAArch32()) then handle_el = PSTATE.EL;
            elsif EL2Enabled() && HCR_EL2.TGE == '1' then UNDEFINED;
            else handle_el = EL1;

        when EL2
            if !HaveEL(EL2) then UNDEFINED;
            elsif PSTATE.EL == EL3 && !UsingAArch32() then handle_el = EL3;
            elsif !IsSecureEL2Enabled() && IsSecure() then UNDEFINED;
            else handle_el = EL2;
        when EL3
            if EDSCR.SDD == '1' || !HaveEL(EL3) then UNDEFINED;
            handle_el = EL3;
        otherwise
            Unreachable();

    from_secure = IsSecure();
    if ELUsingAArch32(handle_el) then
        if PSTATE.M == M32_Monitor then SCR.NS = '0';
        assert UsingAArch32();                  // Cannot move from AArch64 to AArch32
        case handle_el of
            when EL1
                AArch32.WriteMode(M32_Svc);
                if HavePANExt() && SCTLR.SPAN == '0' then
                    PSTATE.PAN = '1';
            when EL2  AArch32.WriteMode(M32_Hyp);
            when EL3
                AArch32.WriteMode(M32_Monitor);
                if HavePANExt() then
                    if !from_secure then
                        PSTATE.PAN = '0';
                    elsif SCTLR.SPAN == '0' then
                        PSTATE.PAN = '1';
        if handle_el == EL2 then
            ELR_hyp = bits(32) UNKNOWN;  HSR = bits(32) UNKNOWN;
        else
            LR = bits(32) UNKNOWN;
        SPSR[] = bits(32) UNKNOWN;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        PSTATE.E = SCTLR[].EE;
        DLR = bits(32) UNKNOWN;  DSPSR = bits(32) UNKNOWN;

    else                                        // Targeting AArch64
        if UsingAArch32() then
            AArch64.MaybeZeroRegisterUppers();
        MaybeZeroSVEUppers(target_el);
        PSTATE.nRW = '0';  PSTATE.SP = '1';  PSTATE.EL = handle_el;
        if HavePANExt() && ((handle_el == EL1 && SCTLR_EL1.SPAN == '0') ||
                            (handle_el == EL2 && HCR_EL2.E2H == '1' &&
                            HCR_EL2.TGE == '1' && SCTLR_EL2.SPAN == '0')) then
            PSTATE.PAN = '1';
        ELR[] = bits(64) UNKNOWN;  SPSR[] = bits(32) UNKNOWN;  ESR[] = bits(32) UNKNOWN;
        DLR_EL0 = bits(64) UNKNOWN;  DSPSR_EL0 = bits(32) UNKNOWN;
        if HaveUAOExt() then PSTATE.UAO = '0';
        if HaveMTEExt() then PSTATE.TCO = '1';

    UpdateEDSCRFields();                        // Update EDSCR PE state flags
    sync_errors = HaveIESB() && SCTLR[].IESB == '1';
    if HaveDoubleFaultExt() && !UsingAArch32() then
        sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3);
    // SCTLR[].IESB might be ignored in Debug state.
    if !ConstrainUnpredictableBool() then
        sync_errors = FALSE;
    if sync_errors then
        SynchronizeErrors();
    return;
```

shared/debug/halting/DRPSInstruction

```
// DRPSInstruction()
// =================
// Operation of the A64 DRPS and T32 ERET instructions in Debug state

DRPSInstruction()

    SynchronizeContext();

    sync_errors = HaveIESB() && SCTLR[].IESB == '1';
    if HaveDoubleFaultExt() && !UsingAArch32() then
        sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3);
    // SCTLR[].IESB might be ignored in Debug state.
    if !ConstrainUnpredictableBool() then
        sync_errors = FALSE;
    if sync_errors then
        SynchronizeErrors();

    SetPSTATEFromPSR(SPSR[]);

    // PSTATE.{N,Z,C,V,Q,GE,SS,D,A,I,F} are not observable and ignored in Debug state, so
    // behave as if UNKNOWN.
    if UsingAArch32() then
        PSTATE.<N,Z,C,V,Q,GE,SS,A,I,F> = bits(13) UNKNOWN;
        //  In AArch32, all instructions are T32 and unconditional.
        PSTATE.IT = '00000000';  PSTATE.T = '1';        // PSTATE.J is RES0
        DLR = bits(32) UNKNOWN;  DSPSR = bits(32) UNKNOWN;
    else
        PSTATE.<N,Z,C,V,SS,D,A,I,F> = bits(9) UNKNOWN;
        DLR_EL0 = bits(64) UNKNOWN;  DSPSR_EL0 = bits(32) UNKNOWN;

    UpdateEDSCRFields();                            // Update EDSCR PE state flags

    return;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/debug/halting/DebugHalt

```
constant bits(6) DebugHalt_Breakpoint      = '000111';
constant bits(6) DebugHalt_EDBGRQ          = '010011';
constant bits(6) DebugHalt_Step_Normal     = '011011';
constant bits(6) DebugHalt_Step_Exclusive  = '011111';
constant bits(6) DebugHalt_OSUnlockCatch   = '100011';
constant bits(6) DebugHalt_ResetCatch      = '100111';
constant bits(6) DebugHalt_Watchpoint      = '101011';
constant bits(6) DebugHalt_HaltInstruction = '101111';
constant bits(6) DebugHalt_SoftwareAccess  = '110011';
constant bits(6) DebugHalt_ExceptionCatch  = '110111';
constant bits(6) DebugHalt_Step_NoSyndrome = '111011';
```

shared/debug/halting/DisableITRAndResumeInstructionPrefetch

```
DisableITRAndResumeInstructionPrefetch();
```

shared/debug/halting/ExecuteA64

```
// Execute an A64 instruction in Debug state.
ExecuteA64(bits(32) instr);
```

shared/debug/halting/ExecuteT32

```
// Execute a T32 instruction in Debug state.
ExecuteT32(bits(16) hw1, bits(16) hw2);
```

shared/debug/halting/ExitDebugState

```
// ExitDebugState()
// ================

ExitDebugState()
    assert Halted();
    SynchronizeContext();

    // Although EDSCR.STATUS signals that the PE is restarting, debuggers must use EDPRSR.SDR to
    // detect that the PE has restarted.
    EDSCR.STATUS = '000001';                          // Signal restarting
    EDESR<2:0> = '000';                               // Clear any pending Halting debug events

    bits(64) new_pc;
    bits(32) spsr;

    if UsingAArch32() then
        new_pc = ZeroExtend(DLR);
        spsr = DSPSR;
    else
        new_pc = DLR_EL0;
        spsr = DSPSR_EL0;
    // If this is an illegal return, SetPSTATEFromPSR() will set PSTATE.IL.
    SetPSTATEFromPSR(spsr);                           // Can update privileged bits, even at EL0

    if UsingAArch32() then
        if ConstrainUnpredictableBool() then new_pc<0> = '0';
        BranchTo(new_pc<31:0>, BranchType_DBGEXIT);    // AArch32 branch
    else
        // If targeting AArch32 then possibly zero the 32 most significant bits of the target PC
        if spsr<4> == '1' && ConstrainUnpredictableBool() then
            new_pc<63:32> = Zeros();
        BranchTo(new_pc, BranchType_DBGEXIT);          // A type of branch that is never predicted

    (EDSCR.STATUS,EDPRSR.SDR) = ('000010','1');        // Atomically signal restarted
    UpdateEDSCRFields();                              // Stop signalling PE state
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    DisableITRAndResumeInstructionPrefetch();

    return;
```

shared/debug/halting/Halt

```
// Halt()
// ======

Halt(bits(6) reason)

    CTI_SignalEvent(CrossTriggerIn_CrossHalt);  // Trigger other cores to halt

    bits(64) preferred_restart_address = ThisInstrAddr();
    spsr = GetPSRFromPSTATE();

    if UsingAArch32() then
        // If entering from AArch32 state, spsr<21> is the DIT bit which has to be moved for DSPSR
        spsr<24> = spsr<21>;
        spsr<21> = PSTATE.SS;                    // Always save the SS bit

    if (HaveBTIExt() &&
        !(reason IN {DebugHalt_Step_Normal, DebugHalt_Step_Exclusive, DebugHalt_Step_NoSyndrome,
                     DebugHalt_Breakpoint, DebugHalt_HaltInstruction}) &&
        ConstrainUnpredictableBool()) then
        DSPSR<11:10> = '00';

    if UsingAArch32() then
        DLR = preferred_restart_address<31:0>;
        DSPSR = spsr;
    else
        DLR_EL0 = preferred_restart_address;
        DSPSR_EL0 = spsr;

    EDSCR.ITE = '1';
    EDSCR.ITO = '0';
    if IsSecure() then
        EDSCR.SDD = '0';                         // If entered in Secure state, allow debug
    elsif HaveEL(EL3) then
        EDSCR.SDD = if ExternalSecureInvasiveDebugEnabled() then '0' else '1';
    else
        assert EDSCR.SDD == '1';                 // Otherwise EDSCR.SDD is RES1
    EDSCR.MA = '0';

    // PSTATE.{SS,D,A,I,F} are not observable and ignored in Debug state, so behave as if
    // UNKNOWN. PSTATE.{N,Z,C,V,Q,GE} are also not observable, but since these are not changed on
    // exception entry, this function also leaves them unchanged. PSTATE.{E,M,nRW,EL,SP} are
    // unchanged. PSTATE.IL is set to 0.
    if UsingAArch32() then
        PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
        //  In AArch32, all instructions are T32 and unconditional.
        PSTATE.IT = '00000000';
        PSTATE.T = '1';                          // PSTATE.J is RES0
    else
        PSTATE.<SS,D,A,I,F> = bits(5) UNKNOWN;
    PSTATE.IL = '0';

    StopInstructionPrefetchAndEnableITR();
    EDSCR.STATUS = reason;                       // Signal entered Debug state
    UpdateEDSCRFields();                         // Update EDSCR PE state flags.

    return;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/debug/halting/HaltOnBreakpointOrWatchpoint

```
// HaltOnBreakpointOrWatchpoint()
// ==============================
// Returns TRUE if the Breakpoint and Watchpoint debug events should be considered for Debug
// state entry, FALSE if they should be considered for a debug exception.

boolean HaltOnBreakpointOrWatchpoint()
    return HaltingAllowed() && EDSCR.HDE == '1' && OSLSR_EL1.OSLK == '0';
```

shared/debug/halting/Halted

```
// Halted()
// ========

boolean Halted()
    return !(EDSCR.STATUS IN {'000001', '000010'});                    // Halted
```

shared/debug/halting/HaltingAllowed

```
// HaltingAllowed()
// ================
// Returns TRUE if halting is currently allowed, FALSE if halting is prohibited.

boolean HaltingAllowed()
    if Halted() || DoubleLockStatus() then
        return FALSE;
    elsif IsSecure() then
        return ExternalSecureInvasiveDebugEnabled();
    else
        return ExternalInvasiveDebugEnabled();
```

shared/debug/halting/Restarting

```
// Restarting()
// ============

boolean Restarting()
    return EDSCR.STATUS == '000001';                                   // Restarting
```

shared/debug/halting/StopInstructionPrefetchAndEnableITR

```
StopInstructionPrefetchAndEnableITR();
```

shared/debug/halting/UpdateEDSCRFields

```
// UpdateEDSCRFields()
// ===================
// Update EDSCR PE state fields

UpdateEDSCRFields()

    if !Halted() then
        EDSCR.EL = '00';
        EDSCR.NS = bit UNKNOWN;
        EDSCR.RW = '1111';
    else
        EDSCR.EL = PSTATE.EL;
        EDSCR.NS = if IsSecure() then '0' else '1';

        bits(4) RW;
        RW<1> = if ELUsingAArch32(EL1) then '0' else '1';
        if PSTATE.EL != EL0 then
            RW<0> = RW<1>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        else
            RW<0> = if UsingAArch32() then '0' else '1';
        if !HaveEL(EL2) || (HaveEL(EL3) && SCR_GEN[].NS == '0' && !IsSecureEL2Enabled()) then
            RW<2> = RW<1>;
        else
            RW<2> = if ELUsingAArch32(EL2) then '0' else '1';
        if !HaveEL(EL3) then
            RW<3> = RW<2>;
        else
            RW<3> = if ELUsingAArch32(EL3) then '0' else '1';

        // The least-significant bits of EDSCR.RW are UNKNOWN if any higher EL is using AArch32.
        if RW<3> == '0' then RW<2:0> = bits(3) UNKNOWN;
        elsif RW<2> == '0' then RW<1:0> = bits(2) UNKNOWN;
        elsif RW<1> == '0' then RW<0> = bit UNKNOWN;
        EDSCR.RW = RW;
    return;
```

shared/debug/haltingevents/CheckExceptionCatch

```
// CheckExceptionCatch()
// =====================
// Check whether an Exception Catch debug event is set on the current Exception level

CheckExceptionCatch(boolean exception_entry)
    // Called after an exception entry or exit, that is, such that IsSecure() and PSTATE.EL are correct
    // for the exception target.
    base = if IsSecure() then 0 else 4;
    if HaltingAllowed() then
        if HaveExtendedECDebugEvents() then
            exception_exit = !exception_entry;
            ctrl = EDECCR<UInt(PSTATE.EL) + base + 8>:EDECCR<UInt(PSTATE.EL) + base>;
            case ctrl of
                when '00'  halt = FALSE;
                when '01'  halt = TRUE;
                when '10'  halt = (exception_exit == TRUE);
                when '11'  halt = (exception_entry == TRUE);
        else
            halt = (EDECCR<UInt(PSTATE.EL) + base> == '1');
        if halt then Halt(DebugHalt_ExceptionCatch);
```

shared/debug/haltingevents/CheckHaltingStep

```
// CheckHaltingStep()
// ==================
// Check whether EDESR.SS has been set by Halting Step

CheckHaltingStep()
    if HaltingAllowed() && EDESR.SS == '1' then
        // The STATUS code depends on how we arrived at the state where EDESR.SS == 1.
        if HaltingStep_DidNotStep() then
            Halt(DebugHalt_Step_NoSyndrome);
        elsif HaltingStep_SteppedEX() then
            Halt(DebugHalt_Step_Exclusive);
        else
            Halt(DebugHalt_Step_Normal);
```

shared/debug/haltingevents/CheckOSUnlockCatch

```
// CheckOSUnlockCatch()
// ====================
// Called on unlocking the OS Lock to pend an OS Unlock Catch debug event

CheckOSUnlockCatch()

    if (HaveDoPD() && CTIDEVCTL.OSUCE == '1')
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    || (!HaveDoPD() && EDECR.OSUCE == '1')
    then
        if !Halted() then EDESR.OSUC = '1';
```

shared/debug/haltingevents/CheckPendingOSUnlockCatch

```
// CheckPendingOSUnlockCatch()
// ===========================
// Check whether EDESR.OSUC has been set by an OS Unlock Catch debug event

CheckPendingOSUnlockCatch()
    if HaltingAllowed() && EDESR.OSUC == '1' then
        Halt(DebugHalt_OSUnlockCatch);
```

shared/debug/haltingevents/CheckPendingResetCatch

```
// CheckPendingResetCatch()
// ========================
// Check whether EDESR.RC has been set by a Reset Catch debug event

CheckPendingResetCatch()
    if HaltingAllowed() && EDESR.RC == '1' then
        Halt(DebugHalt_ResetCatch);
```

shared/debug/haltingevents/CheckResetCatch

```
// CheckResetCatch()
// =================
// Called after reset

CheckResetCatch()
    if (HaveDoPD() && CTIDEVCTL.RCE == '1') || (!HaveDoPD() && EDECR.RCE == '1') then
        EDESR.RC = '1';
        // If halting is allowed then halt immediately
        if HaltingAllowed() then Halt(DebugHalt_ResetCatch);
```

shared/debug/haltingevents/CheckSoftwareAccessToDebugRegisters

```
// CheckSoftwareAccessToDebugRegisters()
// =====================================
// Check for access to Breakpoint and Watchpoint registers.

CheckSoftwareAccessToDebugRegisters()
    os_lock = (if ELUsingAArch32(EL1) then DBGOSLSR.OSLK else OSLSR_EL1.OSLK);
    if HaltingAllowed() && EDSCR.TDA == '1' && os_lock == '0' then
        Halt(DebugHalt_SoftwareAccess);
```

shared/debug/haltingevents/ExternalDebugRequest

```
// ExternalDebugRequest()
// ======================

ExternalDebugRequest()
    if HaltingAllowed() then
        Halt(DebugHalt_EDBGRQ);
    // Otherwise the CTI continues to assert the debug request until it is taken.
```

shared/debug/haltingevents/HaltingStep_DidNotStep

```
// Returns TRUE if the previously executed instruction was executed in the inactive state, that is,
// if it was not itself stepped.
boolean HaltingStep_DidNotStep();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/debug/haltingevents/HaltingStep_SteppedEX

```
// Returns TRUE if the previously executed instruction was a Load-Exclusive class instruction
// executed in the active-not-pending state.
boolean HaltingStep_SteppedEX();
```

shared/debug/haltingevents/RunHaltingStep

```
// RunHaltingStep()
// ================

RunHaltingStep(boolean exception_generated, bits(2) exception_target, boolean syscall,
               boolean reset)
    // "exception_generated" is TRUE if the previous instruction generated a synchronous exception
    // or was cancelled by an asynchronous exception.
    //
    // if "exception_generated" is TRUE then "exception_target" is the target of the exception, and
    // "syscall" is TRUE if the exception is a synchronous exception where the preferred return
    // address is the instruction following that which generated the exception.
    //
    // "reset" is TRUE if exiting reset state into the highest EL.

    if reset then assert !Halted();             // Cannot come out of reset halted
    active = EDECR.SS == '1' && !Halted();

    if active && reset then                     // Coming out of reset with EDECR.SS set
        EDESR.SS = '1';
    elsif active && HaltingAllowed() then
        if exception_generated && exception_target == EL3 then
            advance = syscall || ExternalSecureInvasiveDebugEnabled();
        else
            advance = TRUE;
        if advance then EDESR.SS = '1';

    return;
```

shared/debug/interrupts/ExternalDebugInterruptsDisabled

```
// ExternalDebugInterruptsDisabled()
// =================================
// Determine whether EDSCR disables interrupts routed to 'target'

boolean ExternalDebugInterruptsDisabled(bits(2) target)
    case target of
        when EL3
            int_dis = EDSCR.INTdis == '11' && ExternalSecureInvasiveDebugEnabled();
        when EL2
            int_dis = EDSCR.INTdis == '1x' && ExternalInvasiveDebugEnabled();
        when EL1
            if IsSecure() then
                int_dis = EDSCR.INTdis == '1x' && ExternalSecureInvasiveDebugEnabled();
            else
                int_dis = EDSCR.INTdis != '00' && ExternalInvasiveDebugEnabled();
    return int_dis;
```

shared/debug/interrupts/InterruptID

```
enumeration InterruptID {InterruptID_PMUIRQ, InterruptID_COMMIRQ, InterruptID_CTIIRQ,
                         InterruptID_COMMRX, InterruptID_COMMTX};
```

shared/debug/interrupts/SetInterruptRequestLevel

```
// Set a level-sensitive interrupt to the specified level.
SetInterruptRequestLevel(InterruptID id, signal level);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/debug/samplebasedprofiling/CreatePCSample

```
// CreatePCSample()
// ================

CreatePCSample()
    // In a simple sequential execution of the program, CreatePCSample is executed each time the PE
    // executes an instruction that can be sampled. An implementation is not constrained such that
    // reads of EDPCSRlo return the current values of PC, etc.

    pc_sample.valid = ExternalNoninvasiveDebugAllowed() && !Halted();
    pc_sample.pc = ThisInstrAddr();
    pc_sample.el = PSTATE.EL;
    pc_sample.rw = if UsingAArch32() then '0' else '1';
    pc_sample.ns = if IsSecure() then '0' else '1';
    pc_sample.contextidr = if ELUsingAArch32(EL1) then CONTEXTIDR else CONTEXTIDR_EL1;
    pc_sample.has_el2 = EL2Enabled();

    if EL2Enabled() then
        if ELUsingAArch32(EL2) then
            pc_sample.vmid = ZeroExtend(VTTBR.VMID, 16);
        elsif !Have16bitVMID() || VTCR_EL2.VS == '0' then
            pc_sample.vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
        else
            pc_sample.vmid = VTTBR_EL2.VMID;
        if HaveVirtHostExt() && !ELUsingAArch32(EL2) then
            pc_sample.contextidr_el2 = CONTEXTIDR_EL2;
        else
            pc_sample.contextidr_el2 = bits(32) UNKNOWN;
        pc_sample.el0h = PSTATE.EL == EL0 && IsInHost();
    return;
```

shared/debug/samplebasedprofiling/EDPCSRlo

```
// EDPCSRlo[] (read)
// =================

bits(32) EDPCSRlo[boolean memory_mapped]

    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "signal slave-generated error";
        return bits(32) UNKNOWN;

    // The Software lock is OPTIONAL.
    update = !memory_mapped || EDLSR.SLK == '0';        // Software locked: no side-effects

    if pc_sample.valid then
        sample = pc_sample.pc<31:0>;
        if update then
            if HaveVirtHostExt() && EDSCR.SC2 == '1' then
                EDPCSRhi.PC = (if pc_sample.rw == '0' then Zeros(24) else pc_sample.pc<55:32>);
                EDPCSRhi.EL = pc_sample.el;
                EDPCSRhi.NS = pc_sample.ns;
            else
                EDPCSRhi = (if pc_sample.rw == '0' then Zeros(32) else pc_sample.pc<63:32>);
            EDCIDSR = pc_sample.contextidr;
            if HaveVirtHostExt() && EDSCR.SC2 == '1' then
                EDVIDSR = (if HaveEL(EL2) && pc_sample.ns == '1' then pc_sample.contextidr_el2
                           else bits(32) UNKNOWN);
            else
                if HaveEL(EL2) && pc_sample.ns == '1' && pc_sample.el IN {EL1,EL0} then
                    EDVIDSR.VMID = pc_sample.vmid;
                else
                    EDVIDSR.VMID = Zeros();
                EDVIDSR.NS = pc_sample.ns;
                EDVIDSR.E2 = (if pc_sample.el == EL2 then '1' else '0');
                EDVIDSR.E3 = (if pc_sample.el == EL3 then '1' else '0') AND pc_sample.rw;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            // The conditions for setting HV are not specified if PCSRhi is zero.
            // An example implementation may be "pc_sample.rw".
            EDVIDSR.HV = (if !IsZero(EDPCSRhi) then '1' else bit IMPLEMENTATION_DEFINED "0 or 1");
    else
        sample = Ones(32);
        if update then
            EDPCSRhi = bits(32) UNKNOWN;
            EDCIDSR = bits(32) UNKNOWN;
            EDVIDSR = bits(32) UNKNOWN;

    return sample;
```

shared/debug/samplebasedprofiling/PCSample

```
type PCSample is (
    boolean valid,
    bits(64) pc,
    bits(2) el,
    bit rw,
    bit ns,
    boolean has_el2,
    bits(32) contextidr,
    bits(32) contextidr_el2,
    boolean el0h,
    bits(16) vmid
)

PCSample pc_sample;
```

shared/debug/samplebasedprofiling/PMPCSR

```
// PMPCSR[] (read)
// ===============

bits(32) PMPCSR[boolean memory_mapped]

    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "signal slave-generated error";
        return bits(32) UNKNOWN;

    // The Software lock is OPTIONAL.
    update = !memory_mapped || PMLSR.SLK == '0';        // Software locked: no side-effects

    if pc_sample.valid then
        sample = pc_sample.pc<31:0>;
        if update then
            PMPCSR<55:32> = (if pc_sample.rw == '0' then Zeros(24) else pc_sample.pc<55:32>);
            PMPCSR.EL = pc_sample.el;
            PMPCSR.NS = pc_sample.ns;

            PMCID1SR = pc_sample.contextidr;
            PMCID2SR = if pc_sample.has_el2 then pc_sample.contextidr_el2 else bits(32) UNKNOWN;

            PMVIDSR.VMID = (if pc_sample.has_el2 && pc_sample.el IN {EL1,EL0} && !pc_sample.el0h
                            then pc_sample.vmid else bits(16) UNKNOWN);
    else
        sample = Ones(32);
        if update then
            PMPCSR<55:32>  = bits(24) UNKNOWN;
            PMPCSR.EL = bits(2) UNKNOWN;
            PMPCSR.NS = bit UNKNOWN;

            PMCID1SR = bits(32) UNKNOWN;
            PMCID2SR = bits(32) UNKNOWN;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            PMVIDSR.VMID = bits(16) UNKNOWN;

    return sample;
```

shared/debug/softwarestep/CheckSoftwareStep

```
// CheckSoftwareStep()
// ===================
// Take a Software Step exception if in the active-pending state

CheckSoftwareStep()

    // Other self-hosted debug functions will call AArch32.GenerateDebugExceptions() if called from
    // AArch32 state. However, because Software Step is only active when the debug target Exception
    // level is using AArch64, CheckSoftwareStep only calls AArch64.GenerateDebugExceptions().
    if !ELUsingAArch32(DebugTarget()) && AArch64.GenerateDebugExceptions() then
        if MDSCR_EL1.SS == '1' && PSTATE.SS == '0' then
            AArch64.SoftwareStepException();
```

shared/debug/softwarestep/DebugExceptionReturnSS

```
// DebugExceptionReturnSS()
// ========================
// Returns value to write to PSTATE.SS on an exception return or Debug state exit.

bit DebugExceptionReturnSS(bits(32) spsr)
    assert Halted() || Restarting() ||  PSTATE.EL != EL0;

    SS_bit = '0';

    if MDSCR_EL1.SS == '1' then
        if Restarting() then
            enabled_at_source = FALSE;
        elsif UsingAArch32() then
            enabled_at_source = AArch32.GenerateDebugExceptions();
        else
            enabled_at_source = AArch64.GenerateDebugExceptions();

        if IllegalExceptionReturn(spsr) then
            dest = PSTATE.EL;
        else
            (valid, dest) = ELFromSPSR(spsr);  assert valid;

        secure = IsSecureBelowEL3() || dest == EL3;
        if ELUsingAArch32(dest) then
            enabled_at_dest = AArch32.GenerateDebugExceptionsFrom(dest, secure);
        else
            mask = spsr<9>;
            enabled_at_dest = AArch64.GenerateDebugExceptionsFrom(dest, secure, mask);
        ELd = DebugTargetFrom(secure);
        if !ELUsingAArch32(ELd) && !enabled_at_source && enabled_at_dest then
            SS_bit = spsr<21>;
    return SS_bit;
```

shared/debug/softwarestep/SSAdvance

```
// SSAdvance()
// ===========
// Advance the Software Step state machine.

SSAdvance()

    // A simpler implementation of this function just clears PSTATE.SS to zero regardless of the
    // current Software Step state machine. However, this check is made to illustrate that the
    // processor only needs to consider advancing the state machine from the active-not-pending
    // state.
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    target = DebugTarget();
    step_enabled = !ELUsingAArch32(target) && MDSCR_EL1.SS == '1';
    active_not_pending = step_enabled && PSTATE.SS == '1';

    if active_not_pending then PSTATE.SS = '0';

    return;
```

shared/debug/softwarestep/SoftwareStep_DidNotStep

```
 // Returns TRUE if the previously executed instruction was executed in the inactive state, that is,
 // if it was not itself stepped.
 // Might return TRUE or FALSE if the previously executed instruction was an ISB or ERET executed
 // in the active-not-pending state, or if another exception was taken before the Software Step
exception.
 // Returns FALSE otherwise, indicating that the previously executed instruction was executed in the
 // active-not-pending state, that is, the instruction was stepped.
 boolean SoftwareStep_DidNotStep();
```

shared/debug/softwarestep/SoftwareStep_SteppedEX

```
 // Returns a value that describes the previously executed instruction. The result is valid only if
 // SoftwareStep_DidNotStep() returns FALSE.
 // Might return TRUE or FALSE if the instruction was an AArch32 LDREX that failed its condition code
test.
 // Otherwise returns TRUE if the instruction was a Load-Exclusive class instruction, and FALSE if the
 // instruction was not a Load-Exclusive class instruction.
 boolean SoftwareStep_SteppedEX();
```

J1.3.2 shared/exceptions

This section includes the following pseudocode functions:

- *shared/exceptions/exceptions/ConditionSyndrome*.
- *shared/exceptions/exceptions/Exception* on page J1-7726.
- *shared/exceptions/exceptions/ExceptionRecord* on page J1-7726.
- *shared/exceptions/exceptions/ExceptionSyndrome* on page J1-7727.
- *shared/exceptions/traps/ReservedValue* on page J1-7727.
- *shared/exceptions/traps/UnallocatedEncoding* on page J1-7727.

shared/exceptions/exceptions/ConditionSyndrome

```
 // ConditionSyndrome()
 // ===================
 // Return CV and COND fields of instruction syndrome

 bits(5) ConditionSyndrome()

     bits(5) syndrome;

     if UsingAArch32() then
         cond = AArch32.CurrentCond();
         if PSTATE.T == '0' then             // A32
             syndrome<4> = '1';
             // A conditional A32 instruction that is known to pass its condition code check
             // can be presented either with COND set to 0xE, the value for unconditional, or
             // the COND value held in the instruction.
             if ConditionHolds(cond) && ConstrainUnpredictableBool() then
                 syndrome<3:0> = '1110';
             else
                 syndrome<3:0> = cond;
         else                                // T32
             // When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
             //  * CV set to 0 and COND is set to an UNKNOWN value
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            //  * CV set to 1 and COND is set to the condition code for the condition that
            //    applied to the instruction.
            if boolean IMPLEMENTATION_DEFINED "Condition valid for trapped T32" then
                syndrome<4> = '1';
                syndrome<3:0> = cond;
            else
                syndrome<4> = '0';
                syndrome<3:0> = bits(4) UNKNOWN;
    else
        syndrome<4> = '1';
        syndrome<3:0> = '1110';

    return syndrome;
```

shared/exceptions/exceptions/Exception

```
enumeration Exception {Exception_Uncategorized,       // Uncategorized or unknown reason
                       Exception_WFxTrap,             // Trapped WFI or WFE instruction
                       Exception_CP15RTTrap,          // Trapped AArch32 MCR or MRC access to CP15
                       Exception_CP15RRTTrap,         // Trapped AArch32 MCRR or MRRC access to CP15
                       Exception_CP14RTTrap,          // Trapped AArch32 MCR or MRC access to CP14
                       Exception_CP14DTTrap,          // Trapped AArch32 LDC or STC access to CP14
                       Exception_AdvSIMDFPAccessTrap, // HCPTR-trapped access to SIMD or FP
                       Exception_FPIDTrap,            // Trapped access to SIMD or FP ID register
                       // Trapped BXJ instruction not supported in Armv8
                       Exception_PACTrap,             // Trapped invalid PAC use
                       Exception_CP14RRTTrap,         // Trapped MRRC access to CP14 from AArch32
                       Exception_IllegalState,        // Illegal Execution state
                       Exception_SupervisorCall,      // Supervisor Call
                       Exception_HypervisorCall,      // Hypervisor Call
                       Exception_MonitorCall,         // Monitor Call or Trapped SMC instruction
                       Exception_SystemRegisterTrap,  // Trapped MRS or MSR system register access
                       Exception_ERetTrap,            // Trapped invalid ERET use
                       Exception_InstructionAbort,    // Instruction Abort or Prefetch Abort
                       Exception_PCAlignment,         // PC alignment fault
                       Exception_DataAbort,           // Data Abort
                       Exception_NV2DataAbort,        // Data abort at EL1 reported as being from EL2
                       Exception_PACFail,             // PAC Authentication failure
                       Exception_SPAlignment,         // SP alignment fault
                       Exception_FPTrappedException,  // IEEE trapped FP exception
                       Exception_SError,              // SError interrupt
                       Exception_Breakpoint,          // (Hardware) Breakpoint
                       Exception_SoftwareStep,        // Software Step
                       Exception_Watchpoint,          // Watchpoint
                       Exception_NV2Watchpoint,       // Watchpoint at EL1 reported as being from EL2
                       Exception_SoftwareBreakpoint,  // Software Breakpoint Instruction
                       Exception_VectorCatch,         // AArch32 Vector Catch
                       Exception_IRQ,                 // IRQ interrupt
                       Exception_SVEAccessTrap,       // HCPTR trapped access to SVE
                       Exception_BranchTarget,        // Branch Target Identification
                       Exception_FIQ};                // FIQ interrupt
```

shared/exceptions/exceptions/ExceptionRecord

```
type ExceptionRecord is (Exception exceptype,         // Exception class
                         bits(25)  syndrome,          // Syndrome record
                         bits(64)  vaddress,          // Virtual fault address
                         boolean   ipavalid,          // Physical fault address for second stage faults
is valid
                         bits(1)   NS,                // Physical fault address for second stage faults
is Non-secure or secure
                         bits(52)  ipaddress)         // Physical fault address for second stage faults
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/exceptions/exceptions/ExceptionSyndrome

```
// ExceptionSyndrome()
// ===================
// Return a blank exception syndrome record for an exception of the given type.

ExceptionRecord ExceptionSyndrome(Exception exceptype)

    ExceptionRecord r;

    r.exceptype = exceptype;

    // Initialize all other fields
    r.syndrome = Zeros();
    r.vaddress = Zeros();
    r.ipavalid = FALSE;
    r.NS = '0';
    r.ipaddress = Zeros();

    return r;
```

shared/exceptions/traps/ReservedValue

```
// ReservedValue()
// ===============

ReservedValue()
    if UsingAArch32() && !AArch32.GeneralExceptionsToAArch64() then
        AArch32.TakeUndefInstrException();
    else
        AArch64.UndefinedFault();
```

shared/exceptions/traps/UnallocatedEncoding

```
// UnallocatedEncoding()
// =====================

UnallocatedEncoding()
    if UsingAArch32() && AArch32.ExecutingCP10or11Instr() then
        FPEXC.DEX = '0';
    if UsingAArch32() && !AArch32.GeneralExceptionsToAArch64() then
        AArch32.TakeUndefInstrException();
    else
        AArch64.UndefinedFault();
```

J1.3.3 shared/functions

This section includes the following pseudocode functions:

- *shared/functions/aborts/EncodeLDFSC* on page J1-7734.
- *shared/functions/aborts/IPAValid* on page J1-7735.
- *shared/functions/aborts/IsAsyncAbort* on page J1-7735.
- *shared/functions/aborts/IsDebugException* on page J1-7735.
- *shared/functions/aborts/IsExternalAbort* on page J1-7735.
- *shared/functions/aborts/IsExternalSyncAbort* on page J1-7736.
- *shared/functions/aborts/IsFault* on page J1-7736.
- *shared/functions/aborts/IsSErrorInterrupt* on page J1-7736.
- *shared/functions/aborts/IsSecondStage* on page J1-7736.
- *shared/functions/aborts/LSInstructionSyndrome* on page J1-7736.
- *shared/functions/common/ASR* on page J1-7737.
- *shared/functions/common/ASR_C* on page J1-7737.
- *shared/functions/common/Abs* on page J1-7737.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *shared/functions/common/Align* on page J1-7737.
- *shared/functions/common/BitCount* on page J1-7737.
- *shared/functions/common/CountLeadingSignBits* on page J1-7738.
- *shared/functions/common/CountLeadingZeroBits* on page J1-7738.
- *shared/functions/common/Elem* on page J1-7738.
- *shared/functions/common/Extend* on page J1-7738.
- *shared/functions/common/HighestSetBit* on page J1-7738.
- *shared/functions/common/Int* on page J1-7739.
- *shared/functions/common/IsOnes* on page J1-7739.
- *shared/functions/common/IsZero* on page J1-7739.
- *shared/functions/common/IsZeroBit* on page J1-7739.
- *shared/functions/common/LSL* on page J1-7739.
- *shared/functions/common/LSL_C* on page J1-7739.
- *shared/functions/common/LSR* on page J1-7740.
- *shared/functions/common/LSR_C* on page J1-7740.
- *shared/functions/common/LowestSetBit* on page J1-7740.
- *shared/functions/common/Max* on page J1-7740.
- *shared/functions/common/Min* on page J1-7740.
- *shared/functions/common/Ones* on page J1-7741.
- *shared/functions/common/ROR* on page J1-7741.
- *shared/functions/common/ROR_C* on page J1-7741.
- *shared/functions/common/Replicate* on page J1-7741.
- *shared/functions/common/RoundDown* on page J1-7741.
- *shared/functions/common/RoundTowardsZero* on page J1-7741.
- *shared/functions/common/RoundUp* on page J1-7742.
- *shared/functions/common/SInt* on page J1-7742.
- *shared/functions/common/SignExtend* on page J1-7742.
- *shared/functions/common/UInt* on page J1-7742.
- *shared/functions/common/ZeroExtend* on page J1-7742.
- *shared/functions/common/Zeros* on page J1-7742.
- *shared/functions/crc/BitReverse* on page J1-7743.
- *shared/functions/crc/HaveCRCExt* on page J1-7743.
- *shared/functions/crc/Poly32Mod2* on page J1-7743.
- *shared/functions/crypto/AESInvMixColumns* on page J1-7743.
- *shared/functions/crypto/AESInvShiftRows* on page J1-7744.
- *shared/functions/crypto/AESInvSubBytes* on page J1-7744.
- *shared/functions/crypto/AESMixColumns* on page J1-7744.
- *shared/functions/crypto/AESShiftRows* on page J1-7745.
- *shared/functions/crypto/AESSubBytes* on page J1-7745.
- *shared/functions/crypto/FFmul02* on page J1-7746.
- *shared/functions/crypto/FFmul03* on page J1-7746.
- *shared/functions/crypto/FFmul09* on page J1-7747.
- *shared/functions/crypto/FFmul0B* on page J1-7747.
- *shared/functions/crypto/FFmul0D* on page J1-7747.
- *shared/functions/crypto/FFmul0E* on page J1-7748.
- *shared/functions/crypto/HaveAESExt* on page J1-7748.
- *shared/functions/crypto/HaveBit128PMULLExt* on page J1-7748.
- *shared/functions/crypto/HaveSHA1Ext* on page J1-7749.
- *shared/functions/crypto/HaveSHA256Ext* on page J1-7749.
- *shared/functions/crypto/HaveSHA3Ext* on page J1-7749.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *shared/functions/crypto/HaveSHA512Ext* on page J1-7749.
- *shared/functions/crypto/HaveSM3Ext* on page J1-7749.
- *shared/functions/crypto/HaveSM4Ext* on page J1-7749.
- *shared/functions/crypto/ROL* on page J1-7750.
- *shared/functions/crypto/SHA256hash* on page J1-7750.
- *shared/functions/crypto/SHAchoose* on page J1-7750.
- *shared/functions/crypto/SHAhashSIGMA0* on page J1-7750.
- *shared/functions/crypto/SHAhashSIGMA1* on page J1-7750.
- *shared/functions/crypto/SHAmajority* on page J1-7751.
- *shared/functions/crypto/SHAparity* on page J1-7751.
- *shared/functions/crypto/Sbox* on page J1-7751.
- *shared/functions/exclusive/ClearExclusiveByAddress* on page J1-7751.
- *shared/functions/exclusive/ClearExclusiveLocal* on page J1-7751.
- *shared/functions/exclusive/ClearExclusiveMonitors* on page J1-7751.
- *shared/functions/exclusive/ExclusiveMonitorsStatus* on page J1-7752.
- *shared/functions/exclusive/IsExclusiveGlobal* on page J1-7752.
- *shared/functions/exclusive/IsExclusiveLocal* on page J1-7752.
- *shared/functions/exclusive/MarkExclusiveGlobal* on page J1-7752.
- *shared/functions/exclusive/MarkExclusiveLocal* on page J1-7752.
- *shared/functions/exclusive/ProcessorID* on page J1-7752.
- *shared/functions/extension/AArch32.HaveHPDExt* on page J1-7752.
- *shared/functions/extension/AArch64.HaveHPDExt* on page J1-7752.
- *shared/functions/extension/Have52BitPAExt* on page J1-7752.
- *shared/functions/extension/Have52BitVAExt* on page J1-7753.
- *shared/functions/extension/HaveAArch32BF16Ext* on page J1-7753.
- *shared/functions/extension/HaveAArch32Int8MatMulExt* on page J1-7753.
- *shared/functions/extension/HaveAtomicExt* on page J1-7753.
- *shared/functions/extension/HaveBF16Ext* on page J1-7753.
- *shared/functions/extension/HaveBTIExt* on page J1-7753.
- *shared/functions/extension/HaveBlockBBM* on page J1-7753.
- *shared/functions/extension/HaveCommonNotPrivateTransExt* on page J1-7754.
- *shared/functions/extension/HaveDGHExt* on page J1-7754.
- *shared/functions/extension/HaveDITExt* on page J1-7754.
- *shared/functions/extension/HaveDOTPExt* on page J1-7754.
- *shared/functions/extension/HaveDoPD* on page J1-7754.
- *shared/functions/extension/HaveDoubleFaultExt* on page J1-7754.
- *shared/functions/extension/HaveDoubleLock* on page J1-7754.
- *shared/functions/extension/HaveE0PDExt* on page J1-7755.
- *shared/functions/extension/HaveEMPAMExt* on page J1-7755.
- *shared/functions/extension/HaveExtendedCacheSets* on page J1-7755.
- *shared/functions/extension/HaveExtendedECDebugEvents* on page J1-7755.
- *shared/functions/extension/HaveExtendedExecuteNeverExt* on page J1-7755.
- *shared/functions/extension/HaveFCADDExt* on page J1-7755.
- *shared/functions/extension/HaveFGTExt* on page J1-7755.
- *shared/functions/extension/HaveFJCVTZSExt* on page J1-7756.
- *shared/functions/extension/HaveFP16MulNoRoundingToFP32Ext* on page J1-7756.
- *shared/functions/extension/HaveFlagFormatExt* on page J1-7756.
- *shared/functions/extension/HaveFlagManipulateExt* on page J1-7756.
- *shared/functions/extension/HaveFrintExt* on page J1-7756.
- *shared/functions/extension/HaveHPMDExt* on page J1-7756.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *shared/functions/extension/HaveIDSExt* on page J1-7757.
- *shared/functions/extension/HaveIESB* on page J1-7757.
- *shared/functions/extension/HaveInt8MatMulExt* on page J1-7757.
- *shared/functions/extension/HaveMPAMExt* on page J1-7757.
- *shared/functions/extension/HaveMTEExt* on page J1-7757.
- *shared/functions/extension/HaveNV2Ext* on page J1-7757.
- *shared/functions/extension/HaveNVExt* on page J1-7758.
- *shared/functions/extension/HaveNoSecurePMUDisableOverride* on page J1-7758.
- *shared/functions/extension/HaveNoninvasiveDebugAuth* on page J1-7758.
- *shared/functions/extension/HavePANExt* on page J1-7758.
- *shared/functions/extension/HavePageBasedHardwareAttributes* on page J1-7758.
- *shared/functions/extension/HavePrivATExt* on page J1-7758.
- *shared/functions/extension/HaveQRDMLAHExt* on page J1-7758.
- *shared/functions/extension/HaveRASExt* on page J1-7759.
- *shared/functions/extension/HaveRNG* on page J1-7759.
- *shared/functions/extension/HaveSBExt* on page J1-7759.
- *shared/functions/extension/HaveSSBSExt* on page J1-7759.
- *shared/functions/extension/HaveSecureEL2Ext* on page J1-7759.
- *shared/functions/extension/HaveSecureExtDebugView* on page J1-7759.
- *shared/functions/extension/HaveSelfHostedTrace* on page J1-7759.
- *shared/functions/extension/HaveSmallPageTblExt* on page J1-7760.
- *shared/functions/extension/HaveStage2MemAttrControl* on page J1-7760.
- *shared/functions/extension/HaveStatisticalProfiling* on page J1-7760.
- *shared/functions/extension/HaveTraceExt* on page J1-7760.
- *shared/functions/extension/HaveTrapLoadStoreMultipleDeviceExt* on page J1-7760.
- *shared/functions/extension/HaveUA16Ext* on page J1-7760.
- *shared/functions/extension/HaveUAOExt* on page J1-7760.
- *shared/functions/extension/HaveVirtHostExt* on page J1-7761.
- *shared/functions/extension/InsertIESBBeforeException* on page J1-7761.
- *shared/functions/float/bfloat/BFAdd* on page J1-7761.
- *shared/functions/float/bfloat/BFMatMulAdd* on page J1-7761.
- *shared/functions/float/bfloat/BFMul* on page J1-7762.
- *shared/functions/float/bfloat/BFRound* on page J1-7762.
- *shared/functions/float/bfloat/BFUnpack* on page J1-7763.
- *shared/functions/float/bfloat/FPConvertBF* on page J1-7763.
- *shared/functions/float/bfloat/FPRoundCVBF* on page J1-7764.
- *shared/functions/float/fixedtofp/FixedToFP* on page J1-7764.
- *shared/functions/float/fpabs/FPAbs* on page J1-7765.
- *shared/functions/float/fpadd/FPAdd* on page J1-7765.
- *shared/functions/float/fpcompare/FPCompare* on page J1-7765.
- *shared/functions/float/fpcompareeq/FPCompareEQ* on page J1-7766.
- *shared/functions/float/fpcomparege/FPCompareGE* on page J1-7766.
- *shared/functions/float/fpcomparegt/FPCompareGT* on page J1-7766.
- *shared/functions/float/fpconvert/FPConvert* on page J1-7766.
- *shared/functions/float/fpconvertnan/FPConvertNaN* on page J1-7767.
- *shared/functions/float/fpcrtype/FPCRType* on page J1-7768.
- *shared/functions/float/fpdecoderm/FPDecodeRM* on page J1-7768.
- *shared/functions/float/fpdecoderounding/FPDecodeRounding* on page J1-7768.
- *shared/functions/float/fpdefaultnan/FPDefaultNaN* on page J1-7768.
- *shared/functions/float/fpdiv/FPDiv* on page J1-7768.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *shared/functions/float/fpexc/FPExc* on page J1-7769.
- *shared/functions/float/fpinfinity/FPInfinity* on page J1-7769.
- *shared/functions/float/fpmatmul/FPMatMulAdd* on page J1-7769.
- *shared/functions/float/fpmax/FPMax* on page J1-7769.
- *shared/functions/float/fpmaxnormal/FPMaxNormal* on page J1-7770.
- *shared/functions/float/fpmaxnum/FPMaxNum* on page J1-7770.
- *shared/functions/float/fpmin/FPMin* on page J1-7770.
- *shared/functions/float/fpminnum/FPMinNum* on page J1-7771.
- *shared/functions/float/fpmul/FPMul* on page J1-7771.
- *shared/functions/float/fpmuladd/FPMulAdd* on page J1-7771.
- *shared/functions/float/fpmuladdh/FPMulAddH* on page J1-7772.
- *shared/functions/float/fpmuladdh/FPProcessNaNs3H* on page J1-7773.
- *shared/functions/float/fpmulx/FPMulX* on page J1-7773.
- *shared/functions/float/fpneg/FPNeg* on page J1-7774.
- *shared/functions/float/fponepointfive/FPOnePointFive* on page J1-7774.
- *shared/functions/float/fpprocessexception/FPProcessException* on page J1-7774.
- *shared/functions/float/fpprocessnan/FPProcessNaN* on page J1-7774.
- *shared/functions/float/fpprocessnans/FPProcessNaNs* on page J1-7775.
- *shared/functions/float/fpprocessnans3/FPProcessNaNs3* on page J1-7775.
- *shared/functions/float/fprecipestimate/FPRecipEstimate* on page J1-7776.
- *shared/functions/float/fprecipestimate/RecipEstimate* on page J1-7777.
- *shared/functions/float/fprecpx/FPRecpX* on page J1-7777.
- *shared/functions/float/fpround/FPRound* on page J1-7778.
- *shared/functions/float/fpround/FPRoundCV* on page J1-7780.
- *shared/functions/float/fprounding/FPRounding* on page J1-7780.
- *shared/functions/float/fproundingmode/FPRoundingMode* on page J1-7780.
- *shared/functions/float/fproundint/FPRoundInt* on page J1-7780.
- *shared/functions/float/fproundintn/FPRoundIntN* on page J1-7781.
- *shared/functions/float/fprsqrtestimate/FPRSqrtEstimate* on page J1-7782.
- *shared/functions/float/fprsqrtestimate/RecipSqrtEstimate* on page J1-7783.
- *shared/functions/float/fpsqrt/FPSqrt* on page J1-7783.
- *shared/functions/float/fpsub/FPSub* on page J1-7784.
- *shared/functions/float/fpthree/FPThree* on page J1-7784.
- *shared/functions/float/fptofixed/FPToFixed* on page J1-7784.
- *shared/functions/float/fptofixedjs/FPToFixedJS* on page J1-7785.
- *shared/functions/float/fptwo/FPTwo* on page J1-7786.
- *shared/functions/float/fptype/FPType* on page J1-7786.
- *shared/functions/float/fpunpack/FPUnpack* on page J1-7786.
- *shared/functions/float/fpunpack/FPUnpackBase* on page J1-7786.
- *shared/functions/float/fpunpack/FPUnpackCV* on page J1-7788.
- *shared/functions/float/fpzero/FPZero* on page J1-7788.
- *shared/functions/float/vfpexpandimm/VFPExpandImm* on page J1-7788.
- *shared/functions/integer/AddWithCarry* on page J1-7788.
- *shared/functions/memory/AArch64.BranchAddr* on page J1-7789.
- *shared/functions/memory/AccType* on page J1-7789.
- *shared/functions/memory/AccessDescriptor* on page J1-7789.
- *shared/functions/memory/AddrTop* on page J1-7789.
- *shared/functions/memory/AddressDescriptor* on page J1-7790.
- *shared/functions/memory/Allocation* on page J1-7790.
- *shared/functions/memory/BigEndian* on page J1-7790.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *shared/functions/memory/BigEndianReverse* on page J1-7790.
- *shared/functions/memory/Cacheability* on page J1-7791.
- *shared/functions/memory/CreateAccessDescriptor* on page J1-7791.
- *shared/functions/memory/CreateAccessDescriptorPTW* on page J1-7791.
- *shared/functions/memory/DataMemoryBarrier* on page J1-7791.
- *shared/functions/memory/DataSynchronizationBarrier* on page J1-7791.
- *shared/functions/memory/DescriptorUpdate* on page J1-7791.
- *shared/functions/memory/DeviceType* on page J1-7791.
- *shared/functions/memory/EffectiveTBI* on page J1-7791.
- *shared/functions/memory/EffectiveTCMA* on page J1-7792.
- *shared/functions/memory/Fault* on page J1-7792.
- *shared/functions/memory/FaultRecord* on page J1-7793.
- *shared/functions/memory/FullAddress* on page J1-7793.
- *shared/functions/memory/Hint_Prefetch* on page J1-7793.
- *shared/functions/memory/MBReqDomain* on page J1-7793.
- *shared/functions/memory/MBReqTypes* on page J1-7793.
- *shared/functions/memory/MemAttrHints* on page J1-7794.
- *shared/functions/memory/MemType* on page J1-7794.
- *shared/functions/memory/MemoryAttributes* on page J1-7794.
- *shared/functions/memory/Permissions* on page J1-7794.
- *shared/functions/memory/PrefetchHint* on page J1-7794.
- *shared/functions/memory/SpeculativeStoreBypassBarrierToPA* on page J1-7794.
- *shared/functions/memory/SpeculativeStoreBypassBarrierToVA* on page J1-7794.
- *shared/functions/memory/TLBRecord* on page J1-7794.
- *shared/functions/memory/Tag* on page J1-7795.
- *shared/functions/memory/_Mem* on page J1-7795.
- *shared/functions/mpam/DefaultMPAMinfo* on page J1-7795.
- *shared/functions/mpam/DefaultPARTID* on page J1-7795.
- *shared/functions/mpam/DefaultPMG* on page J1-7795.
- *shared/functions/mpam/GenMPAMcurEL* on page J1-7795.
- *shared/functions/mpam/MAP_vPARTID* on page J1-7796.
- *shared/functions/mpam/MPAMisEnabled* on page J1-7797.
- *shared/functions/mpam/MPAMisVirtual* on page J1-7797.
- *shared/functions/mpam/genMPAM* on page J1-7797.
- *shared/functions/mpam/genMPAMel* on page J1-7797.
- *shared/functions/mpam/genPARTID* on page J1-7798.
- *shared/functions/mpam/genPMG* on page J1-7798.
- *shared/functions/mpam/getMPAM_PARTID* on page J1-7798.
- *shared/functions/mpam/getMPAM_PMG* on page J1-7799.
- *shared/functions/mpam/mapvpmw* on page J1-7799.
- *shared/functions/registers/BranchTo* on page J1-7800.
- *shared/functions/registers/BranchToAddr* on page J1-7800.
- *shared/functions/registers/BranchType* on page J1-7800.
- *shared/functions/registers/Hint_Branch* on page J1-7800.
- *shared/functions/registers/NextInstrAddr* on page J1-7800.
- *shared/functions/registers/ResetExternalDebugRegisters* on page J1-7801.
- *shared/functions/registers/ThisInstrAddr* on page J1-7801.
- *shared/functions/registers/_PC* on page J1-7801.
- *shared/functions/registers/_R* on page J1-7801.
- *shared/functions/sysregisters/SPSR* on page J1-7801.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *shared/functions/system/ArchVersion* on page J1-7802.
- *shared/functions/system/BranchTargetCheck* on page J1-7802.
- *shared/functions/system/ClearEventRegister* on page J1-7802.
- *shared/functions/system/ClearPendingPhysicalSError* on page J1-7802.
- *shared/functions/system/ClearPendingVirtualSError* on page J1-7802.
- *shared/functions/system/ConditionHolds* on page J1-7803.
- *shared/functions/system/ConsumptionOfSpeculativeDataBarrier* on page J1-7803.
- *shared/functions/system/CurrentInstrSet* on page J1-7803.
- *shared/functions/system/CurrentPL* on page J1-7803.
- *shared/functions/system/EL0* on page J1-7803.
- *shared/functions/system/EL2Enabled* on page J1-7803.
- *shared/functions/system/ELFromM32* on page J1-7804.
- *shared/functions/system/ELFromSPSR* on page J1-7804.
- *shared/functions/system/ELIsInHost* on page J1-7805.
- *shared/functions/system/ELStateUsingAArch32* on page J1-7805.
- *shared/functions/system/ELStateUsingAArch32K* on page J1-7805.
- *shared/functions/system/ELUsingAArch32* on page J1-7806.
- *shared/functions/system/ELUsingAArch32K* on page J1-7806.
- *shared/functions/system/EndOfInstruction* on page J1-7806.
- *shared/functions/system/EnterLowPowerState* on page J1-7806.
- *shared/functions/system/EventRegister* on page J1-7806.
- *shared/functions/system/GetPSRFromPSTATE* on page J1-7806.
- *shared/functions/system/HasArchVersion* on page J1-7807.
- *shared/functions/system/HaveAArch32EL* on page J1-7807.
- *shared/functions/system/HaveAnyAArch32* on page J1-7807.
- *shared/functions/system/HaveAnyAArch64* on page J1-7807.
- *shared/functions/system/HaveEL* on page J1-7807.
- *shared/functions/system/HaveELUsingSecurityState* on page J1-7808.
- *shared/functions/system/HaveFP16Ext* on page J1-7808.
- *shared/functions/system/HighestEL* on page J1-7808.
- *shared/functions/system/HighestELUsingAArch32* on page J1-7808.
- *shared/functions/system/Hint_DGH* on page J1-7808.
- *shared/functions/system/Hint_Yield* on page J1-7809.
- *shared/functions/system/IllegalExceptionReturn* on page J1-7809.
- *shared/functions/system/InstrSet* on page J1-7809.
- *shared/functions/system/InstructionSynchronizationBarrier* on page J1-7809.
- *shared/functions/system/InterruptPending* on page J1-7809.
- *shared/functions/system/IsEventRegisterSet* on page J1-7810.
- *shared/functions/system/IsHighestEL* on page J1-7810.
- *shared/functions/system/IsInHost* on page J1-7810.
- *shared/functions/system/IsPhysicalSErrorPending* on page J1-7810.
- *shared/functions/system/IsSecure* on page J1-7810.
- *shared/functions/system/IsSecureBelowEL3* on page J1-7810.
- *shared/functions/system/IsSecureEL2Enabled* on page J1-7811.
- *shared/functions/system/IsVirtualSErrorPending* on page J1-7811.
- *shared/functions/system/Mode_Bits* on page J1-7811.
- *shared/functions/system/PLOfEL* on page J1-7811.
- *shared/functions/system/PSTATE* on page J1-7811.
- *shared/functions/system/PrivilegeLevel* on page J1-7811.
- *shared/functions/system/ProcState* on page J1-7812.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *shared/functions/system/RestoredITBits* on page J1-7812.
- *shared/functions/system/SCRType* on page J1-7812.
- *shared/functions/system/SCR_GEN* on page J1-7813.
- *shared/functions/system/SendEvent* on page J1-7813.
- *shared/functions/system/SendEventLocal* on page J1-7813.
- *shared/functions/system/SetPSTATEFromPSR* on page J1-7813.
- *shared/functions/system/ShouldAdvanceIT* on page J1-7814.
- *shared/functions/system/SpeculationBarrier* on page J1-7814.
- *shared/functions/system/SynchronizeContext* on page J1-7814.
- *shared/functions/system/SynchronizeErrors* on page J1-7814.
- *shared/functions/system/TakeUnmaskedPhysicalSErrorInterrupts* on page J1-7814.
- *shared/functions/system/TakeUnmaskedSErrorInterrupts* on page J1-7814.
- *shared/functions/system/ThisInstr* on page J1-7814.
- *shared/functions/system/ThisInstrLength* on page J1-7814.
- *shared/functions/system/Unreachable* on page J1-7814.
- *shared/functions/system/UsingAArch32* on page J1-7814.
- *shared/functions/system/WaitForEvent* on page J1-7815.
- *shared/functions/system/WaitForInterrupt* on page J1-7815.
- *shared/functions/unpredictable/ConstrainUnpredictable* on page J1-7815.
- *shared/functions/unpredictable/ConstrainUnpredictableBits* on page J1-7815.
- *shared/functions/unpredictable/ConstrainUnpredictableBool* on page J1-7815.
- *shared/functions/unpredictable/ConstrainUnpredictableInteger* on page J1-7815.
- *shared/functions/unpredictable/Constraint* on page J1-7816.
- *shared/functions/vector/AdvSIMDExpandImm* on page J1-7816.
- *shared/functions/vector/MatMulAdd* on page J1-7817.
- *shared/functions/vector/PolynomialMult* on page J1-7817.
- *shared/functions/vector/SatQ* on page J1-7817.
- *shared/functions/vector/SignedSatQ* on page J1-7817.
- *shared/functions/vector/UnsignedRSqrtEstimate* on page J1-7818.
- *shared/functions/vector/UnsignedRecipEstimate* on page J1-7818.
- *shared/functions/vector/UnsignedSatQ* on page J1-7818.

shared/functions/aborts/EncodeLDFSC

```
// EncodeLDFSC()
// =============
// Function that gives the Long-descriptor FSC code for types of Fault

bits(6) EncodeLDFSC(Fault statuscode, integer level)

    bits(6) result;
    case statuscode of
        when Fault_AddressSize         result = '0000':level<1:0>; assert level IN {0,1,2,3};
        when Fault_AccessFlag          result = '0010':level<1:0>; assert level IN {1,2,3};
        when Fault_Permission          result = '0011':level<1:0>; assert level IN {1,2,3};
        when Fault_Translation         result = '0001':level<1:0>; assert level IN {0,1,2,3};
        when Fault_SyncExternal        result = '010000';
        when Fault_SyncExternalOnWalk  result = '0101':level<1:0>; assert level IN {0,1,2,3};
        when Fault_SyncParity          result = '011000';
        when Fault_SyncParityOnWalk    result = '0111':level<1:0>; assert level IN {0,1,2,3};
        when Fault_AsyncParity         result = '011001';
        when Fault_AsyncExternal       result = '010001';
        when Fault_Alignment           result = '100001';
        when Fault_Debug               result = '100010';
        when Fault_TLBConflict         result = '110000';
        when Fault_HWUpdateAccessFlag  result = '110001';
        when Fault_Lockdown            result = '110100';  // IMPLEMENTATION DEFINED
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        when Fault_Exclusive           result = '110101';  // IMPLEMENTATION DEFINED
        otherwise                      Unreachable();

    return result;
```

shared/functions/aborts/IPAValid

```
// IPAValid()
// ==========
// Return TRUE if the IPA is reported for the abort

boolean IPAValid(FaultRecord fault)
    assert fault.statuscode != Fault_None;

    if fault.s2fs1walk then
        return fault.statuscode IN {Fault_AccessFlag, Fault_Permission, Fault_Translation,
                                    Fault_AddressSize};
    elsif fault.secondstage then
        return fault.statuscode IN {Fault_AccessFlag, Fault_Translation, Fault_AddressSize};
    else
        return FALSE;
```

shared/functions/aborts/IsAsyncAbort

```
// IsAsyncAbort()
// ==============
// Returns TRUE if the abort currently being processed is an asynchronous abort, and FALSE
// otherwise.

boolean IsAsyncAbort(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsAsyncAbort()
// ==============

boolean IsAsyncAbort(FaultRecord fault)
    return IsAsyncAbort(fault.statuscode);
```

shared/functions/aborts/IsDebugException

```
// IsDebugException()
// ==================

boolean IsDebugException(FaultRecord fault)
    assert fault.statuscode != Fault_None;
    return fault.statuscode == Fault_Debug;
```

shared/functions/aborts/IsExternalAbort

```
// IsExternalAbort()
// =================
// Returns TRUE if the abort currently being processed is an external abort and FALSE otherwise.

boolean IsExternalAbort(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {Fault_SyncExternal, Fault_SyncParity, Fault_SyncExternalOnWalk,
Fault_SyncParityOnWalk,
                       Fault_AsyncExternal, Fault_AsyncParity });

// IsExternalAbort()
// =================
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
boolean IsExternalAbort(FaultRecord fault)
    return IsExternalAbort(fault.statuscode);
```

shared/functions/aborts/IsExternalSyncAbort

```
// IsExternalSyncAbort()
// =====================
// Returns TRUE if the abort currently being processed is an external synchronous abort and FALSE
otherwise.

boolean IsExternalSyncAbort(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {Fault_SyncExternal, Fault_SyncParity, Fault_SyncExternalOnWalk,
Fault_SyncParityOnWalk});

// IsExternalSyncAbort()
// =====================

boolean IsExternalSyncAbort(FaultRecord fault)
    return IsExternalSyncAbort(fault.statuscode);
```

shared/functions/aborts/IsFault

```
// IsFault()
// =========
// Return TRUE if a fault is associated with an address descriptor

boolean IsFault(AddressDescriptor addrdesc)
    return addrdesc.fault.statuscode != Fault_None;
```

shared/functions/aborts/IsSErrorInterrupt

```
// IsSErrorInterrupt()
// ===================
// Returns TRUE if the abort currently being processed is an SError interrupt, and FALSE
// otherwise.

boolean IsSErrorInterrupt(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsSErrorInterrupt()
// ===================

boolean IsSErrorInterrupt(FaultRecord fault)
    return IsSErrorInterrupt(fault.statuscode);
```

shared/functions/aborts/IsSecondStage

```
// IsSecondStage()
// ===============

boolean IsSecondStage(FaultRecord fault)
    assert fault.statuscode != Fault_None;

    return fault.secondstage;
```

shared/functions/aborts/LSInstructionSyndrome

```
bits(11) LSInstructionSyndrome();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/functions/common/ASR

```
// ASR()
// =====

bits(N) ASR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = ASR_C(x, shift);
    return result;
```

shared/functions/common/ASR_C

```
// ASR_C()
// =======

(bits(N), bit) ASR_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = SignExtend(x, shift+N);
    result = extended_x<shift+N-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);
```

shared/functions/common/Abs

```
// Abs()
// =====

integer Abs(integer x)
    return if x >= 0 then x else -x;

// Abs()
// =====

real Abs(real x)
    return if x >= 0.0 then x else -x;
```

shared/functions/common/Align

```
// Align()
// =======

integer Align(integer x, integer y)
    return y * (x DIV y);

// Align()
// =======

bits(N) Align(bits(N) x, integer y)
    return Align(UInt(x), y)<N-1:0>;
```

shared/functions/common/BitCount

```
// BitCount()
// ==========

integer BitCount(bits(N) x)
    integer result = 0;
    for i = 0 to N-1
        if x<i> == '1' then
            result = result + 1;
    return result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/functions/common/CountLeadingSignBits

```
// CountLeadingSignBits()
// ======================

integer CountLeadingSignBits(bits(N) x)
    return CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>);
```

shared/functions/common/CountLeadingZeroBits

```
// CountLeadingZeroBits()
// ======================

integer CountLeadingZeroBits(bits(N) x)
    return N - (HighestSetBit(x) + 1);
```

shared/functions/common/Elem

```
// Elem[] - non-assignment form
// ============================

bits(size) Elem[bits(N) vector, integer e, integer size]
    assert e >= 0 && (e+1)*size <= N;
    return vector<e*size+size-1 : e*size>;

// Elem[] - non-assignment form
// ============================

bits(size) Elem[bits(N) vector, integer e]
    return Elem[vector, e, size];

// Elem[] - assignment form
// ========================

Elem[bits(N) &vector, integer e, integer size] = bits(size) value
    assert e >= 0 && (e+1)*size <= N;
    vector<(e+1)*size-1:e*size> = value;
    return;

// Elem[] - assignment form
// ========================

Elem[bits(N) &vector, integer e] = bits(size) value
    Elem[vector, e, size] = value;
    return;
```

shared/functions/common/Extend

```
// Extend()
// ========

bits(N) Extend(bits(M) x, integer N, boolean unsigned)
    return if unsigned then ZeroExtend(x, N) else SignExtend(x, N);

// Extend()
// ========

bits(N) Extend(bits(M) x, boolean unsigned)
    return Extend(x, N, unsigned);
```

shared/functions/common/HighestSetBit

```
// HighestSetBit()
// ===============
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer HighestSetBit(bits(N) x)
    for i = N-1 downto 0
        if x<i> == '1' then return i;
    return -1;
```

shared/functions/common/Int

```
// Int()
// =====

integer Int(bits(N) x, boolean unsigned)
    result = if unsigned then UInt(x) else SInt(x);
    return result;
```

shared/functions/common/IsOnes

```
// IsOnes()
// ========

boolean IsOnes(bits(N) x)
    return x == Ones(N);
```

shared/functions/common/IsZero

```
// IsZero()
// ========

boolean IsZero(bits(N) x)
    return x == Zeros(N);
```

shared/functions/common/IsZeroBit

```
// IsZeroBit()
// ===========

bit IsZeroBit(bits(N) x)
    return if IsZero(x) then '1' else '0';
```

shared/functions/common/LSL

```
// LSL()
// =====

bits(N) LSL(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSL_C(x, shift);
    return result;
```

shared/functions/common/LSL_C

```
// LSL_C()
// =======

(bits(N), bit) LSL_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = x : Zeros(shift);
    result = extended_x<N-1:0>;
    carry_out = extended_x<N>;
    return (result, carry_out);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/functions/common/LSR

```
// LSR()
// =====

bits(N) LSR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSR_C(x, shift);
    return result;
```

shared/functions/common/LSR_C

```
// LSR_C()
// =======

(bits(N), bit) LSR_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = ZeroExtend(x, shift+N);
    result = extended_x<shift+N-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);
```

shared/functions/common/LowestSetBit

```
// LowestSetBit()
// ==============

integer LowestSetBit(bits(N) x)
    for i = 0 to N-1
        if x<i> == '1' then return i;
    return N;
```

shared/functions/common/Max

```
// Max()
// =====

integer Max(integer a, integer b)
    return if a >= b then a else b;

// Max()
// =====

real Max(real a, real b)
    return if a >= b then a else b;
```

shared/functions/common/Min

```
// Min()
// =====

integer Min(integer a, integer b)
    return if a <= b then a else b;

// Min()
// =====

real Min(real a, real b)
    return if a <= b then a else b;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/functions/common/Ones

```
// Ones()
// ======

bits(N) Ones(integer N)
    return Replicate('1',N);

// Ones()
// ======

bits(N) Ones()
    return Ones(N);
```

shared/functions/common/ROR

```
// ROR()
// =====

bits(N) ROR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = ROR_C(x, shift);
    return result;
```

shared/functions/common/ROR_C

```
// ROR_C()
// =======

(bits(N), bit) ROR_C(bits(N) x, integer shift)
    assert shift != 0;
    m = shift MOD N;
    result = LSR(x,m) OR LSL(x,N-m);
    carry_out = result<N-1>;
    return (result, carry_out);
```

shared/functions/common/Replicate

```
// Replicate()
// ===========

bits(N) Replicate(bits(M) x)
    assert N MOD M == 0;
    return Replicate(x, N DIV M);

bits(M*N) Replicate(bits(M) x, integer N);
```

shared/functions/common/RoundDown

```
integer RoundDown(real x);
```

shared/functions/common/RoundTowardsZero

```
// RoundTowardsZero()
// ==================

integer RoundTowardsZero(real x)
    return if x == 0.0 then 0 else if x >= 0.0 then RoundDown(x) else RoundUp(x);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/functions/common/RoundUp

```
integer RoundUp(real x);
```

shared/functions/common/SInt

```
// SInt()
// ======

integer SInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    if x<N-1> == '1' then result = result - 2^N;
    return result;
```

shared/functions/common/SignExtend

```
// SignExtend()
// ============

bits(N) SignExtend(bits(M) x, integer N)
    assert N >= M;
    return Replicate(x<M-1>, N-M) : x;

// SignExtend()
// ============

bits(N) SignExtend(bits(M) x)
    return SignExtend(x, N);
```

shared/functions/common/UInt

```
// UInt()
// ======

integer UInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    return result;
```

shared/functions/common/ZeroExtend

```
// ZeroExtend()
// ============

bits(N) ZeroExtend(bits(M) x, integer N)
    assert N >= M;
    return Zeros(N-M) : x;

// ZeroExtend()
// ============

bits(N) ZeroExtend(bits(M) x)
    return ZeroExtend(x, N);
```

shared/functions/common/Zeros

```
// Zeros()
// =======

bits(N) Zeros(integer N)
    return Replicate('0',N);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
// Zeros()
// =======

bits(N) Zeros()
    return Zeros(N);
```

shared/functions/crc/BitReverse

```
// BitReverse()
// ============

bits(N) BitReverse(bits(N) data)
    bits(N) result;
    for i = 0 to N-1
        result<N-i-1> = data<i>;
    return result;
```

shared/functions/crc/HaveCRCExt

```
// HaveCRCExt()
// ============

boolean HaveCRCExt()
    return HasArchVersion(ARMv8p1) || boolean IMPLEMENTATION_DEFINED "Have CRC extension";
```

shared/functions/crc/Poly32Mod2

```
// Poly32Mod2()
// ============

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation

bits(32) Poly32Mod2(bits(N) data, bits(32) poly)
    assert N > 32;
    for i = N-1 downto 32
        if data<i> == '1' then
            data<i-1:0> = data<i-1:0> EOR (poly:Zeros(i-32));
    return data<31:0>;
```

shared/functions/crypto/AESInvMixColumns

```
// AESInvMixColumns()
// ==================
// Transformation in the Inverse Cipher that is the inverse of AESMixColumns.

bits(128) AESInvMixColumns(bits (128) op)
    bits(4*8) in0 = op< 96+:8> : op< 64+:8> : op< 32+:8> : op<  0+:8>;
    bits(4*8) in1 = op<104+:8> : op< 72+:8> : op< 40+:8> : op<  8+:8>;
    bits(4*8) in2 = op<112+:8> : op< 80+:8> : op< 48+:8> : op< 16+:8>;
    bits(4*8) in3 = op<120+:8> : op< 88+:8> : op< 56+:8> : op< 24+:8>;

    bits(4*8) out0;
    bits(4*8) out1;
    bits(4*8) out2;
    bits(4*8) out3;

    for c = 0 to 3
        out0<c*8+:8> =  FFmul0E(in0<c*8+:8>) EOR FFmul0B(in1<c*8+:8>) EOR FFmul0D(in2<c*8+:8>) EOR
FFmul09(in3<c*8+:8>);
        out1<c*8+:8> =  FFmul09(in0<c*8+:8>) EOR FFmul0E(in1<c*8+:8>) EOR FFmul0B(in2<c*8+:8>) EOR
FFmul0D(in3<c*8+:8>);
        out2<c*8+:8> =  FFmul0D(in0<c*8+:8>) EOR FFmul09(in1<c*8+:8>) EOR FFmul0E(in2<c*8+:8>) EOR
FFmul0B(in3<c*8+:8>);
        out3<c*8+:8> =  FFmul0B(in0<c*8+:8>) EOR FFmul0D(in1<c*8+:8>) EOR FFmul09(in2<c*8+:8>) EOR
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
FFmul0E(in3<c*8+:8>);

    return (
        out3<3*8+:8> : out2<3*8+:8> : out1<3*8+:8> : out0<3*8+:8> :
        out3<2*8+:8> : out2<2*8+:8> : out1<2*8+:8> : out0<2*8+:8> :
        out3<1*8+:8> : out2<1*8+:8> : out1<1*8+:8> : out0<1*8+:8> :
        out3<0*8+:8> : out2<0*8+:8> : out1<0*8+:8> : out0<0*8+:8>
    );
```

shared/functions/crypto/AESInvShiftRows

```
// AESInvShiftRows()
// =================
// Transformation in the Inverse Cipher that is inverse of AESShiftRows.

bits(128) AESInvShiftRows(bits(128) op)
    return (
        op< 24+:8> : op< 48+:8> : op< 72+:8> : op< 96+:8> :
        op<120+:8> : op< 16+:8> : op< 40+:8> : op< 64+:8> :
        op< 88+:8> : op<112+:8> : op<  8+:8> : op< 32+:8> :
        op< 56+:8> : op< 80+:8> : op<104+:8> : op<  0+:8>
    );
```

shared/functions/crypto/AESInvSubBytes

```
// AESInvSubBytes()
// ================
// Transformation in the Inverse Cipher that is the inverse of AESSubBytes.

bits(128) AESInvSubBytes(bits(128) op)
    // Inverse S-box values
    bits(16*16*8) GF2_inv = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x7d0c2155631469e126d677ba7e042b17<127:0> :
        /*E*/ 0x619953833cbbebc8b0f52aae4d3be0a0<127:0> :
        /*D*/ 0xef9cc9939f7ae52d0d4ab519a97f5160<127:0> :
        /*C*/ 0x5fec8027591012b131c7078833a8dd1f<127:0> :
        /*B*/ 0xf45acd78fec0db9a2079d2c64b3e56fc<127:0> :
        /*A*/ 0x1bbe18aa0e62b76f89c5291d711af147<127:0> :
        /*9*/ 0x6edf751ce837f9e28535ade72274ac96<127:0> :
        /*8*/ 0x73e6b4f0cecff297eadc674f4111913a<127:0> :
        /*7*/ 0x6b8a130103bdafc1020f3fca8f1e2cd0<127:0> :
        /*6*/ 0x0645b3b80558e4f70ad3bc8c00abd890<127:0> :
        /*5*/ 0x849d8da75746155edab9edfd5048706c<127:0> :
        /*4*/ 0x92b6655dcc5ca4d41698688664f6f872<127:0> :
        /*3*/ 0x25d18b6d49a25b76b224d92866a12e08<127:0> :
        /*2*/ 0x4ec3fa420b954cee3d23c2a632947b54<127:0> :
        /*1*/ 0xcbe9dec444438e3487ff2f9b8239e37c<127:0> :
        /*0*/ 0xfbd7f3819ea340bf38a53630d56a0952<127:0>
    );
    bits(128) out;
    for i = 0 to 15
        out<i*8+:8> = GF2_inv<UInt(op<i*8+:8>)*8+:8>;
    return out;
```

shared/functions/crypto/AESMixColumns

```
// AESMixColumns()
// ===============
// Transformation in the Cipher that takes all of the columns of the
// State and mixes their data (independently of one another) to
// produce new columns.

bits(128) AESMixColumns(bits (128) op)
    bits(4*8) in0 = op< 96+:8> : op< 64+:8> : op< 32+:8> : op<  0+:8>;
    bits(4*8) in1 = op<104+:8> : op< 72+:8> : op< 40+:8> : op<  8+:8>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    bits(4*8) in2 = op<112+:8> : op< 80+:8> : op< 48+:8> : op< 16+:8>;
    bits(4*8) in3 = op<120+:8> : op< 88+:8> : op< 56+:8> : op< 24+:8>;

    bits(4*8) out0;
    bits(4*8) out1;
    bits(4*8) out2;
    bits(4*8) out3;

    for c = 0 to 3
        out0<c*8+:8> =  FFmul02(in0<c*8+:8>) EOR FFmul03(in1<c*8+:8>) EOR        in2<c*8+:8>  EOR
in3<c*8+:8>;
        out1<c*8+:8> =         in0<c*8+:8>  EOR FFmul02(in1<c*8+:8>) EOR FFmul03(in2<c*8+:8>) EOR
in3<c*8+:8>;
        out2<c*8+:8> =          in0<c*8+:8>  EOR         in1<c*8+:8>  EOR FFmul02(in2<c*8+:8>) EOR
FFmul03(in3<c*8+:8>);
        out3<c*8+:8> =  FFmul03(in0<c*8+:8>) EOR         in1<c*8+:8>  EOR         in2<c*8+:8>  EOR
FFmul02(in3<c*8+:8>);

    return (
        out3<3*8+:8> : out2<3*8+:8> : out1<3*8+:8> : out0<3*8+:8> :
        out3<2*8+:8> : out2<2*8+:8> : out1<2*8+:8> : out0<2*8+:8> :
        out3<1*8+:8> : out2<1*8+:8> : out1<1*8+:8> : out0<1*8+:8> :
        out3<0*8+:8> : out2<0*8+:8> : out1<0*8+:8> : out0<0*8+:8>
    );
```

shared/functions/crypto/AESShiftRows

```
// AESShiftRows()
// ==============
// Transformation in the Cipher that processes the State by cyclically
// shifting the last three rows of the State by different offsets.

bits(128) AESShiftRows(bits(128) op)
    return (
        op< 88+:8> : op< 48+:8> : op<  8+:8> : op< 96+:8> :
        op< 56+:8> : op< 16+:8> : op<104+:8> : op< 64+:8> :
        op< 24+:8> : op<112+:8> : op< 72+:8> : op< 32+:8> :
        op<120+:8> : op< 80+:8> : op< 40+:8> : op<  0+:8>
    );
```

shared/functions/crypto/AESSubBytes

```
// AESSubBytes()
// =============
// Transformation in the Cipher that processes the State using a nonlinear
// byte substitution table (S-box) that operates on each of the State bytes
// independently.

bits(128) AESSubBytes(bits(128) op)
    // S-box values
    bits(16*16*8) GF2 = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x16bb54b00f2d99416842e6bf0d89a18c<127:0> :
        /*E*/ 0xdf2855cee9871e9b948ed9691198f8e1<127:0> :
        /*D*/ 0x9e1dc186b95735610ef6034866b53e70<127:0> :
        /*C*/ 0x8a8bbd4b1f74dde8c6b4a61c2e2578ba<127:0> :
        /*B*/ 0x08ae7a65eaf4566ca94ed58d6d37c8e7<127:0> :
        /*A*/ 0x79e4959162acd3c25c2406490a3a32e0<127:0> :
        /*9*/ 0xdb0b5ede14b8ee4688902a22dc4f8160<127:0> :
        /*8*/ 0x73195d643d7ea7c41744975fec130ccd<127:0> :
        /*7*/ 0xd2f3ff1021dab6bcf5389d928f40a351<127:0> :
        /*6*/ 0xa89f3c507f02f94585334d43fbaaefd0<127:0> :
        /*5*/ 0xcf584c4a39becb6a5bb1fc20ed00d153<127:0> :
        /*4*/ 0x842fe329b3d63b52a05a6e1b1a2c8309<127:0> :
        /*3*/ 0x75b227ebe28012079a059618c323c704<127:0> :
        /*2*/ 0x1531d871f1e5a534ccf73f362693fdb7<127:0> :
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        /*1*/ 0xc072a49cafa2d4adf04759fa7dc982ca<127:0> :
        /*0*/ 0x76abd7fe2b670130c56f6bf27b777c63<127:0>
    );
    bits(128) out;
    for i = 0 to 15
        out<i*8+:8> = GF2<UInt(op<i*8+:8>)*8+:8>;
    return out;
```

shared/functions/crypto/FFmul02

```
// FFmul02()
// =========

bits(8) FFmul02(bits(8) b)
    bits(256*8) FFmul_02 = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0xE5E7E1E3EDEFE9EBF5F7F1F3FDFFF9FB<127:0> :
        /*E*/ 0xC5C7C1C3CDCFC9CBD5D7D1D3DDDFD9DB<127:0> :
        /*D*/ 0xA5A7A1A3ADAFA9ABB5B7B1B3BDBFB9BB<127:0> :
        /*C*/ 0x858781838D8F898B959791939D9F999B<127:0> :
        /*B*/ 0x656761636D6F696B757771737D7F797B<127:0> :
        /*A*/ 0x454741434D4F494B555751535D5F595B<127:0> :
        /*9*/ 0x252721232D2F292B353731333D3F393B<127:0> :
        /*8*/ 0x050701030D0F090B151711131D1F191B<127:0> :
        /*7*/ 0xFEFCFAF8F6F4F2F0EEECEAE8E6E4E2E0<127:0> :
        /*6*/ 0xDEDCDAD8D6D4D2D0CECCCAC8C6C4C2C0<127:0> :
        /*5*/ 0xBEBCBAB8B6B4B2B0AEACAAA8A6A4A2A0<127:0> :
        /*4*/ 0x9E9C9A98969492908E8C8A8886848280<127:0> :
        /*3*/ 0x7E7C7A78767472706E6C6A6866646260<127:0> :
        /*2*/ 0x5E5C5A58565452504E4C4A4846444240<127:0> :
        /*1*/ 0x3E3C3A38363432302E2C2A2826242220<127:0> :
        /*0*/ 0x1E1C1A18161412100E0C0A0806040200<127:0>
    );
    return FFmul_02<UInt(b)*8+:8>;
```

shared/functions/crypto/FFmul03

```
// FFmul03()
// =========

bits(8) FFmul03(bits(8) b)
    bits(256*8) FFmul_03 = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x1A191C1F16151013020104070E0D080B<127:0> :
        /*E*/ 0x2A292C2F26252023323134373E3D383B<127:0> :
        /*D*/ 0x7A797C7F76757073626164676E6D686B<127:0> :
        /*C*/ 0x4A494C4F46454043525154575E5D585B<127:0> :
        /*B*/ 0xDAD9DCDFD6D5D0D3C2C1C4C7CECDC8CB<127:0> :
        /*A*/ 0xEAE9ECEFE6E5E0E3F2F1F4F7FEFDF8FB<127:0> :
        /*9*/ 0xBAB9BCBFB6B5B0B3A2A1A4A7AEADA8AB<127:0> :
        /*8*/ 0x8A898C8F86858083929194979E9D989B<127:0> :
        /*7*/ 0x818287848D8E8B88999A9F9C95969390<127:0> :
        /*6*/ 0xB1B2B7B4BDBEBBB8A9AAAFACA5A6A3A0<127:0> :
        /*5*/ 0xE1E2E7E4EDEEEBE8F9FAFFFCF5F6F3F0<127:0> :
        /*4*/ 0xD1D2D7D4DDDEDBD8C9CACFCCC5C6C3C0<127:0> :
        /*3*/ 0x414247444D4E4B48595A5F5C55565350<127:0> :
        /*2*/ 0x717277747D7E7B78696A6F6C65666360<127:0> :
        /*1*/ 0x212227242D2E2B28393A3F3C35363330<127:0> :
        /*0*/ 0x111217141D1E1B18090A0F0C05060300<127:0>
    );
    return FFmul_03<UInt(b)*8+:8>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/functions/crypto/FFmul09

```
// FFmul09()
// =========

bits(8) FFmul09(bits(8) b)
    bits(256*8) FFmul_09 = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x464F545D626B70790E071C152A233831<127:0> :
        /*E*/ 0xD6DFC4CDF2FBE0E99E978C85BAB3A8A1<127:0> :
        /*D*/ 0x7D746F6659504B42353C272E1118030A<127:0> :
        /*C*/ 0xEDE4FFF6C9C0DBD2A5ACB7BE8188939A<127:0> :
        /*B*/ 0x3039222B141D060F78716A635C554E47<127:0> :
        /*A*/ 0xA0A9B2BB848D969FE8E1FAF3CCC5DED7<127:0> :
        /*9*/ 0x0B0219102F263D34434A5158676E757C<127:0> :
        /*8*/ 0x9B928980BFB6ADA4D3DAC1C8F7FEE5EC<127:0> :
        /*7*/ 0xAAA3B8B18E879C95E2EBF0F9C6CFD4DD<127:0> :
        /*6*/ 0x3A3328211E170C05727B6069565F444D<127:0> :
        /*5*/ 0x9198838AB5BCA7AED9D0CBC2FDF4EFE6<127:0> :
        /*4*/ 0x0108131A252C373E49405B526D647F76<127:0> :
        /*3*/ 0xDCD5CEC7F8F1EAE3949D868FB0B9A2AB<127:0> :
        /*2*/ 0x4C455E5768617A73040D161F2029323B<127:0> :
        /*1*/ 0xE7EEF5FCC3CAD1D8AFA6BDB48B829990<127:0> :
        /*0*/ 0x777E656C535A41483F362D241B120900<127:0>
    );
    return FFmul_09<UInt(b)*8+:8>;
```

shared/functions/crypto/FFmul0B

```
// FFmul0B()
// =========

bits(8) FFmul0B(bits(8) b)
    bits(256*8) FFmul_0B = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0xA3A8B5BE8F849992FBF0EDE6D7DCC1CA<127:0> :
        /*E*/ 0x1318050E3F3429224B405D56676C717A<127:0> :
        /*D*/ 0xD8D3CEC5F4FFE2E9808B969DACA7BAB1<127:0> :
        /*C*/ 0x68637E75444F5259303B262D1C170A01<127:0> :
        /*B*/ 0x555E434879726F640D061B10212A373C<127:0> :
        /*A*/ 0xE5EEF3F8C9C2DFD4BDB6ABA0919A878C<127:0> :
        /*9*/ 0x2E2538330209141F767D606B5A514C47<127:0> :
        /*8*/ 0x9E958883B2B9A4AFC6CDD0DBEAE1FCF7<127:0> :
        /*7*/ 0x545F424978736E650C071A11202B363D<127:0> :
        /*6*/ 0xE4EFF2F9C8C3DED5BCB7AAA1909B868D<127:0> :
        /*5*/ 0x2F2439320308151E777C616A5B504D46<127:0> :
        /*4*/ 0x9F948982B3B8A5AEC7CCD1DAEBE0FDF6<127:0> :
        /*3*/ 0xA2A9B4BF8E859893FAF1ECE7D6DDC0CB<127:0> :
        /*2*/ 0x1219040F3E3528234A415C57666D707B<127:0> :
        /*1*/ 0xD9D2CFC4F5FEE3E8818A979CADA6BBB0<127:0> :
        /*0*/ 0x69627F74454E5358313A272C1D160B00<127:0>
    );
    return FFmul_0B<UInt(b)*8+:8>;
```

shared/functions/crypto/FFmul0D

```
// FFmul0D()
// =========

bits(8) FFmul0D(bits(8) b)
    bits(256*8) FFmul_0D = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x979A8D80A3AEB9B4FFF2E5E8CBC6D1DC<127:0> :
        /*E*/ 0x474A5D50737E69642F2235381B16010C<127:0> :
        /*D*/ 0x2C21363B1815020F44495E53707D6A67<127:0> :
        /*C*/ 0xFCF1E6EBC8C5D2DF94998E83A0ADBAB7<127:0> :
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        /*B*/ 0xFAF7E0EDCEC3D4D9929F8885A6ABBCB1<127:0> :
        /*A*/ 0x2A27303D1E130409424F5855767B6C61<127:0> :
        /*9*/ 0x414C5B5675786F622924333E1D10070A<127:0> :
        /*8*/ 0x919C8B86A5A8BFB2F9F4E3EECDC0D7DA<127:0> :
        /*7*/ 0x4D40575A7974636E25283F32111C0B06<127:0> :
        /*6*/ 0x9D90878AA9A4B3BEF5F8EFE2C1CCDBD6<127:0> :
        /*5*/ 0xF6FBECE1C2CFD8D59E938489AAA7B0BD<127:0> :
        /*4*/ 0x262B3C31121F08054E4354597A77606D<127:0> :
        /*3*/ 0x202D3A3714190E034845525F7C71666B<127:0> :
        /*2*/ 0xF0FDEAE7C4C9DED39895828FACA1B6BB<127:0> :
        /*1*/ 0x9B96818CAFA2B5B8F3FEE9E4C7CADDD0<127:0> :
        /*0*/ 0x4B46515C7F726568232E3934171A0D00<127:0>
    );
    return FFmul_0D<UInt(b)*8+:8>;
```

shared/functions/crypto/FFmul0E

```
// FFmul0E()
// =========

bits(8) FFmul0E(bits(8) b)
    bits(256*8) FFmul_0E = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x8D83919FB5BBA9A7FDF3E1EFC5CBD9D7<127:0> :
        /*E*/ 0x6D63717F555B49471D13010F252B3937<127:0> :
        /*D*/ 0x56584A446E60727C26283A341E10020C<127:0> :
        /*C*/ 0xB6B8AAA48E80929CC6C8DAD4FEF0E2EC<127:0> :
        /*B*/ 0x202E3C321816040A505E4C426866747A<127:0> :
        /*A*/ 0xC0CEDCD2F8F6E4EAB0BEACA28886949A<127:0> :
        /*9*/ 0xFBF5E7E9C3CDDFD18B859799B3BDAFA1<127:0> :
        /*8*/ 0x1B150709232D3F316B657779535D4F41<127:0> :
        /*7*/ 0xCCC2D0DEF4FAE8E6BCB2A0AE848A9896<127:0> :
        /*6*/ 0x2C22303E141A08065C52404E646A7876<127:0> :
        /*5*/ 0x17190B052F21333D67697B755F51434D<127:0> :
        /*4*/ 0xF7F9EBE5CFC1D3DD87899B95BFB1A3AD<127:0> :
        /*3*/ 0x616F7D735957454B111F0D032927353B<127:0> :
        /*2*/ 0x818F9D93B9B7A5ABF1FFEDE3C9C7D5DB<127:0> :
        /*1*/ 0xBAB4A6A8828C9E90CAC4D6D8F2FCEEE0<127:0> :
        /*0*/ 0x5A544648626C7E702A243638121C0E00<127:0>
    );
    return FFmul_0E<UInt(b)*8+:8>;
```

shared/functions/crypto/HaveAESExt

```
// HaveAESExt()
// ============
// TRUE if AES cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveAESExt()
    return boolean IMPLEMENTATION_DEFINED "Has AES Crypto instructions";
```

shared/functions/crypto/HaveBit128PMULLExt

```
// HaveBit128PMULLExt()
// ====================
// TRUE if 128 bit form of PMULL instructions support is implemented,
// FALSE otherwise.

boolean HaveBit128PMULLExt()
    return boolean IMPLEMENTATION_DEFINED "Has 128-bit form of PMULL instructions";
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/functions/crypto/HaveSHA1Ext

```
// HaveSHA1Ext()
// =============
// TRUE if SHA1 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA1Ext()
    return boolean IMPLEMENTATION_DEFINED "Has SHA1 Crypto instructions";
```

shared/functions/crypto/HaveSHA256Ext

```
// HaveSHA256Ext()
// ===============
// TRUE if SHA256 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA256Ext()
    return boolean IMPLEMENTATION_DEFINED "Has SHA256 Crypto instructions";
```

shared/functions/crypto/HaveSHA3Ext

```
// HaveSHA3Ext()
// =============
// TRUE if SHA3 cryptographic instructions support is implemented,
// and when SHA1 and SHA2 basic cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA3Ext()
    if !HasArchVersion(ARMv8p2) || !(HaveSHA1Ext() && HaveSHA256Ext()) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SHA3 Crypto instructions";
```

shared/functions/crypto/HaveSHA512Ext

```
// HaveSHA512Ext()
// ===============
// TRUE if SHA512 cryptographic instructions support is implemented,
// and when SHA1 and SHA2 basic cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA512Ext()
    if !HasArchVersion(ARMv8p2) || !(HaveSHA1Ext() && HaveSHA256Ext()) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SHA512 Crypto instructions";
```

shared/functions/crypto/HaveSM3Ext

```
// HaveSM3Ext()
// ============
// TRUE if SM3 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSM3Ext()
    if !HasArchVersion(ARMv8p2) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SM3 Crypto instructions";
```

shared/functions/crypto/HaveSM4Ext

```
// HaveSM4Ext()
// ============
// TRUE if SM4 cryptographic instructions support is implemented,
// FALSE otherwise.
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
boolean HaveSM4Ext()
    if !HasArchVersion(ARMv8p2) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SM4 Crypto instructions";
```

shared/functions/crypto/ROL

```
// ROL()
// =====

bits(N) ROL(bits(N) x, integer shift)
    assert shift >= 0 && shift <= N;
    if (shift == 0) then
        return x;
    return ROR(x, N-shift);
```

shared/functions/crypto/SHA256hash

```
// SHA256hash()
// ============

bits(128) SHA256hash(bits (128) X, bits(128) Y, bits(128) W, boolean part1)
    bits(32) chs, maj, t;

    for e = 0 to 3
        chs = SHAchoose(Y<31:0>, Y<63:32>, Y<95:64>);
        maj = SHAmajority(X<31:0>, X<63:32>, X<95:64>);
        t = Y<127:96> + SHAhashSIGMA1(Y<31:0>) + chs + Elem[W, e, 32];
        X<127:96> = t + X<127:96>;
        Y<127:96> = t + SHAhashSIGMA0(X<31:0>) + maj;
        <Y, X> = ROL(Y : X, 32);
    return (if part1 then X else Y);
```

shared/functions/crypto/SHAchoose

```
// SHAchoose()
// ===========

bits(32) SHAchoose(bits(32) x, bits(32) y, bits(32) z)
    return (((y EOR z) AND x) EOR z);
```

shared/functions/crypto/SHAhashSIGMA0

```
// SHAhashSIGMA0()
// ===============

bits(32) SHAhashSIGMA0(bits(32) x)
    return ROR(x, 2) EOR ROR(x, 13) EOR ROR(x, 22);
```

shared/functions/crypto/SHAhashSIGMA1

```
// SHAhashSIGMA1()
// ===============

bits(32) SHAhashSIGMA1(bits(32) x)
    return ROR(x, 6) EOR ROR(x, 11) EOR ROR(x, 25);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/functions/crypto/SHAmajority

```
// SHAmajority()
// =============

bits(32) SHAmajority(bits(32) x, bits(32) y, bits(32) z)
    return ((x AND y) OR ((x OR y) AND z));
```

shared/functions/crypto/SHAparity

```
// SHAparity()
// ===========

bits(32) SHAparity(bits(32) x, bits(32) y, bits(32) z)
    return (x EOR y EOR z);
```

shared/functions/crypto/Sbox

```
// Sbox()
// ======
// Used in SM4E crypto instruction

bits(8) Sbox(bits(8) sboxin)
    bits(8) sboxout;
    bits(2048) sboxstring =
0xd690e9fecce13db716b614c228fb2c052b679a762abe04c3aa441326498606999c4250f491ef987a33540b43edcfac62e4b31ca
9c908e89580df94fa758f3fa64707a7fcf37317ba83593c19e6854fa8686b81b27164da8bf8eb0f4b70569d351e240e5e6358d1a2
25227c3b01217887d40046579fd327524c3602e7a0c4c89eeabf8ad240c738b5a3f7f2cef96115a1e0ae5da49b341a55ad933230f
58cb1e31df6e22e8266ca60c02923ab0d534e6fd5db3745defd8e2f03ff6a726d6c5b518d1baf92bbddbc7f11d95c411f105ad80a
c13188a5cd7bbd2d74d012b8e5b4b08969974a0c96777e65b9f109c56ec68418f07dec3adc4d2079ee5f3ed7cb3948<2047:0>;

    sboxout = sboxstring<(255-UInt(sboxin))*8+7:(255-UInt(sboxin))*8>;
    return sboxout;
```

shared/functions/exclusive/ClearExclusiveByAddress

```
// Clear the global Exclusives monitors for all PEs EXCEPT processorid if they
// record any part of the physical address region of size bytes starting at paddress.
// It is IMPLEMENTATION DEFINED whether the global Exclusives monitor for processorid
// is also cleared if it records any part of the address region.
ClearExclusiveByAddress(FullAddress paddress, integer processorid, integer size);
```

shared/functions/exclusive/ClearExclusiveLocal

```
// Clear the local Exclusives monitor for the specified processorid.
ClearExclusiveLocal(integer processorid);
```

shared/functions/exclusive/ClearExclusiveMonitors

```
// ClearExclusiveMonitors()
// ========================

// Clear the local Exclusives monitor for the executing PE.

ClearExclusiveMonitors()
    ClearExclusiveLocal(ProcessorID());
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/functions/exclusive/ExclusiveMonitorsStatus

```
// Returns '0' to indicate success if the last memory write by this PE was to
// the same physical address region endorsed by ExclusiveMonitorsPass().
// Returns '1' to indicate failure if address translation resulted in a different
// physical address.
bit ExclusiveMonitorsStatus();
```

shared/functions/exclusive/IsExclusiveGlobal

```
// Return TRUE if the global Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.
boolean IsExclusiveGlobal(FullAddress paddress, integer processorid, integer size);
```

shared/functions/exclusive/IsExclusiveLocal

```
// Return TRUE if the local Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.
boolean IsExclusiveLocal(FullAddress paddress, integer processorid, integer size);
```

shared/functions/exclusive/MarkExclusiveGlobal

```
// Record the physical address region of size bytes starting at paddress in
// the global Exclusives monitor for processorid.
MarkExclusiveGlobal(FullAddress paddress, integer processorid, integer size);
```

shared/functions/exclusive/MarkExclusiveLocal

```
// Record the physical address region of size bytes starting at paddress in
// the local Exclusives monitor for processorid.
MarkExclusiveLocal(FullAddress paddress, integer processorid, integer size);
```

shared/functions/exclusive/ProcessorID

```
// Return the ID of the currently executing PE.
integer ProcessorID();
```

shared/functions/extension/AArch32.HaveHPDExt

```
// AArch32.HaveHPDExt()
// ====================

boolean AArch32.HaveHPDExt()
    return HasArchVersion(ARMv8p2);
```

shared/functions/extension/AArch64.HaveHPDExt

```
// AArch64.HaveHPDExt()
// ====================

boolean AArch64.HaveHPDExt()
    return HasArchVersion(ARMv8p1);
```

shared/functions/extension/Have52BitPAExt

```
// Have52BitPAExt()
// ================
// Returns TRUE if Large Physical Address extension
// support is implemented and FALSE otherwise.

boolean Have52BitPAExt()
    return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has large 52-bit PA/IPA support";
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/functions/extension/Have52BitVAExt

```
// Have52BitVAExt()
// ================
// Returns TRUE if Large Virtual Address extension
// support is implemented and FALSE otherwise.

boolean Have52BitVAExt()
    return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has large 52-bit VA support";
```

shared/functions/extension/HaveAArch32BF16Ext

```
// HaveAArch32BF16Ext()
// ====================
// Returns TRUE if AArch32 BFloat16 instruction support is implemented, and FALSE otherwise.

boolean HaveAArch32BF16Ext()
    return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has AArch32 BFloat16 extension";
```

shared/functions/extension/HaveAArch32Int8MatMulExt

```
// HaveAArch32Int8MatMulExt()
// ==========================
// Returns TRUE if AArch32 8-bit integer matrix multiply instruction support
// implemented, and FALSE otherwise.

boolean HaveAArch32Int8MatMulExt()
    return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has AArch32 Int8 Mat Mul
extension";
```

shared/functions/extension/HaveAtomicExt

```
// HaveAtomicExt()
// ===============

boolean HaveAtomicExt()
    return HasArchVersion(ARMv8p1);
```

shared/functions/extension/HaveBF16Ext

```
// HaveBF16Ext()
// =============
// Returns TRUE if AArch64 BFloat16 instruction support is implemented, and FALSE otherwise.

boolean HaveBF16Ext()
    return HasArchVersion(ARMv8p6) || (HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has
AArch64 BFloat16 extension");
```

shared/functions/extension/HaveBTIExt

```
// HaveBTIExt()
// ============
// Returns TRUE if support for Branch Target Indentification is implemented.

boolean HaveBTIExt()
    return HasArchVersion(ARMv8p5);
```

shared/functions/extension/HaveBlockBBM

```
// HaveBlockBBM()
// ==============
// Returns TRUE if support for changing block size without requring break-before-make is implemented.
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
boolean HaveBlockBBM()
    return HasArchVersion(ARMv8p4);
```

shared/functions/extension/HaveCommonNotPrivateTransExt

```
// HaveCommonNotPrivateTransExt()
// ==============================

boolean HaveCommonNotPrivateTransExt()
    return HasArchVersion(ARMv8p2);
```

shared/functions/extension/HaveDGHExt

```
// HaveDGHExt()
// ============
// Returns TRUE if Data Gathering Hint instruction support is implemented, and FALSE otherwise.

boolean HaveDGHExt()
    return boolean IMPLEMENTATION_DEFINED "Has AArch64 DGH extension";
```

shared/functions/extension/HaveDITExt

```
// HaveDITExt()
// ============

boolean HaveDITExt()
    return HasArchVersion(ARMv8p4);
```

shared/functions/extension/HaveDOTPExt

```
// HaveDOTPExt()
// =============
// Returns TRUE if Dot Product feature support is implemented, and FALSE otherwise.

boolean HaveDOTPExt()
    return HasArchVersion(ARMv8p4) || (HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has
Dot Product extension");
```

shared/functions/extension/HaveDoPD

```
// HaveDoPD()
// ==========
// Returns TRUE if Debug Over Power Down extension
// support is implemented and FALSE otherwise.

boolean HaveDoPD()
    return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has DoPD extension";
```

shared/functions/extension/HaveDoubleFaultExt

```
// HaveDoubleFaultExt()
// ====================

boolean HaveDoubleFaultExt()
    return (HasArchVersion(ARMv8p4) && HaveEL(EL3) && !ELUsingAArch32(EL3) && HaveIESB());
```

shared/functions/extension/HaveDoubleLock

```
// HaveDoubleLock()
// ================
// Returns TRUE if support for the OS Double Lock is implemented.
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
boolean HaveDoubleLock()
    return !HasArchVersion(ARMv8p4) || boolean IMPLEMENTATION_DEFINED "OS Double Lock is implemented";
```

shared/functions/extension/HaveE0PDExt

```
// HaveE0PDExt()
// =============
// Returns TRUE if support for constant fault times for unprivileged accesses
// to the memory map is implemented.

boolean HaveE0PDExt()
    return HasArchVersion(ARMv8p5);
```

shared/functions/extension/HaveEMPAMExt

```
// HaveEMPAMExt()
// ==============
// Returns TRUE if Enhanced MPAM is implemented, and FALSE otherwise.

boolean HaveEMPAMExt()
    return (HasArchVersion(ARMv8p6) &&
            HaveMPAMExt() &&
            boolean IMPLEMENTATION_DEFINED "Has enhanced MPAM extension");
```

shared/functions/extension/HaveExtendedCacheSets

```
// HaveExtendedCacheSets()
// =======================

boolean HaveExtendedCacheSets()
    return HasArchVersion(ARMv8p3);
```

shared/functions/extension/HaveExtendedECDebugEvents

```
// HaveExtendedECDebugEvents()
// ===========================

boolean HaveExtendedECDebugEvents()
    return HasArchVersion(ARMv8p2);
```

shared/functions/extension/HaveExtendedExecuteNeverExt

```
// HaveExtendedExecuteNeverExt()
// =============================

boolean HaveExtendedExecuteNeverExt()
    return HasArchVersion(ARMv8p2);
```

shared/functions/extension/HaveFCADDExt

```
// HaveFCADDExt()
// ==============

boolean HaveFCADDExt()
    return HasArchVersion(ARMv8p3);
```

shared/functions/extension/HaveFGTExt

```
// HaveFGTExt()
// ============
// Returns TRUE if Fine Grained Trap is implemented, and FALSE otherwise.
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
boolean HaveFGTExt()
    return HasArchVersion(ARMv8p6) && !ELUsingAArch32(EL2);
```

shared/functions/extension/HaveFJCVTZSExt

```
// HaveFJCVTZSExt()
// ================

boolean HaveFJCVTZSExt()
    return HasArchVersion(ARMv8p3);
```

shared/functions/extension/HaveFP16MulNoRoundingToFP32Ext

```
// HaveFP16MulNoRoundingToFP32Ext()
// ================================
// Returns TRUE if has FP16 multiply with no intermediate rounding accumulate to FP32 instructions,
// and FALSE otherwise

boolean HaveFP16MulNoRoundingToFP32Ext()
    if !HaveFP16Ext() then return FALSE;
    if HasArchVersion(ARMv8p4) then return TRUE;
    return (HasArchVersion(ARMv8p2) &&
            boolean IMPLEMENTATION_DEFINED "Has accumulate FP16 product into FP32 extension");
```

shared/functions/extension/HaveFlagFormatExt

```
// HaveFlagFormatExt()
// ===================
// Returns TRUE if flag format conversion instructions implemented.

boolean HaveFlagFormatExt()
    return HasArchVersion(ARMv8p5);
```

shared/functions/extension/HaveFlagManipulateExt

```
// HaveFlagManipulateExt()
// =======================
// Returns TRUE if flag manipulate instructions are implemented.

boolean HaveFlagManipulateExt()
    return HasArchVersion(ARMv8p4);
```

shared/functions/extension/HaveFrintExt

```
// HaveFrintExt()
// ==============
// Returns TRUE if FRINT instructions are implemented.

boolean HaveFrintExt()
    return HasArchVersion(ARMv8p5);
```

shared/functions/extension/HaveHPMDExt

```
// HaveHPMDExt()
// =============

boolean HaveHPMDExt()
    return HasArchVersion(ARMv8p1);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/functions/extension/HaveIDSExt

```
// HaveIDSExt()
// ============
// Returns TRUE if ID register handling feature is implemented.

boolean HaveIDSExt()
    return HasArchVersion(ARMv8p4);
```

shared/functions/extension/HaveIESB

```
// HaveIESB()
// ==========

boolean HaveIESB()
    return (HaveRASExt() &&
            boolean IMPLEMENTATION_DEFINED "Has Implicit Error Synchronization Barrier");
```

shared/functions/extension/HaveInt8MatMulExt

```
// HaveInt8MatMulExt()
// ===================
// Returns TRUE if AArch64 8-bit integer matrix multiply instruction support
// implemented, and FALSE otherwise.

boolean HaveInt8MatMulExt()
    return HasArchVersion(ARMv8p6) || (HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has
AArch64 Int8 Mat Mul extension");
```

shared/functions/extension/HaveMPAMExt

```
// HaveMPAMExt()
// =============
// Returns TRUE if MPAM is implemented, and FALSE otherwise.

boolean HaveMPAMExt()
    return (HasArchVersion(ARMv8p2) &&
            boolean IMPLEMENTATION_DEFINED "Has MPAM extension");
```

shared/functions/extension/HaveMTEExt

```
// HaveMTEExt()
// ============
// Returns TRUE if MTE implemented, and FALSE otherwise.

boolean HaveMTEExt()
    if !HasArchVersion(ARMv8p5) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has MTE extension";
```

shared/functions/extension/HaveNV2Ext

```
// HaveNV2Ext()
// ============
// Returns TRUE if Enhanced Nested Virtualization is implemented.

boolean HaveNV2Ext()
    return (HasArchVersion(ARMv8p4) && HaveNVExt()
            && boolean IMPLEMENTATION_DEFINED "Has support for Enhanced Nested Virtualization");
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/functions/extension/HaveNVExt

```
// HaveNVExt()
// ===========
// Returns TRUE if Nested Virtualization is implemented.

boolean HaveNVExt()
    return HasArchVersion(ARMv8p3) && boolean IMPLEMENTATION_DEFINED "Has Nested Virtualization";
```

shared/functions/extension/HaveNoSecurePMUDisableOverride

```
// HaveNoSecurePMUDisableOverride()
// ================================

boolean HaveNoSecurePMUDisableOverride()
    return HasArchVersion(ARMv8p2);
```

shared/functions/extension/HaveNoninvasiveDebugAuth

```
// HaveNoninvasiveDebugAuth()
// ==========================
// Returns TRUE if the Non-invasive debug controls are implemented.

boolean HaveNoninvasiveDebugAuth()
    return !HasArchVersion(ARMv8p4);
```

shared/functions/extension/HavePANExt

```
// HavePANExt()
// ============

boolean HavePANExt()
    return HasArchVersion(ARMv8p1);
```

shared/functions/extension/HavePageBasedHardwareAttributes

```
// HavePageBasedHardwareAttributes()
// =================================

boolean HavePageBasedHardwareAttributes()
    return HasArchVersion(ARMv8p2);
```

shared/functions/extension/HavePrivATExt

```
// HavePrivATExt()
// ===============

boolean HavePrivATExt()
    return HasArchVersion(ARMv8p2);
```

shared/functions/extension/HaveQRDMLAHExt

```
// HaveQRDMLAHExt()
// ================

boolean HaveQRDMLAHExt()
    return HasArchVersion(ARMv8p1);

boolean HaveAccessFlagUpdateExt()
    return HasArchVersion(ARMv8p1);

boolean HaveDirtyBitModifierExt()
    return HasArchVersion(ARMv8p1);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/functions/extension/HaveRASExt

```
// HaveRASExt()
// ============

boolean HaveRASExt()
    return (HasArchVersion(ARMv8p2) ||
            boolean IMPLEMENTATION_DEFINED "Has RAS extension");
```

shared/functions/extension/HaveRNG

```
// HaveRNG()
// =========
// Returns TRUE if Random Number Generator extension
// support is implemented and FALSE otherwise.

boolean HaveRNG()
    return HasArchVersion(ARMv8p5) && boolean IMPLEMENTATION_DEFINED "Has RNG extension";
```

shared/functions/extension/HaveSBExt

```
// HaveSBExt()
// ===========
// Returns TRUE if support for SB is implemented, and FALSE otherwise.

boolean HaveSBExt()
    return HasArchVersion(ARMv8p5) || boolean IMPLEMENTATION_DEFINED "Has SB extension";
```

shared/functions/extension/HaveSSBSExt

```
// HaveSSBSExt()
// =============
// Returns TRUE if support for SSBS is implemented, and FALSE otherwise.

boolean HaveSSBSExt()
    return HasArchVersion(ARMv8p5) || boolean IMPLEMENTATION_DEFINED "Has SSBS extension";
```

shared/functions/extension/HaveSecureEL2Ext

```
// HaveSecureEL2Ext()
// ==================
// Returns TRUE if Secure EL2 is implemented.

boolean HaveSecureEL2Ext()
    return HasArchVersion(ARMv8p4);
```

shared/functions/extension/HaveSecureExtDebugView

```
// HaveSecureExtDebugView()
// ========================
// Returns TRUE if support for Secure and Non-secure views of debug peripherals is implemented.

boolean HaveSecureExtDebugView()
    return HasArchVersion(ARMv8p4);
```

shared/functions/extension/HaveSelfHostedTrace

```
// HaveSelfHostedTrace()
// =====================

boolean HaveSelfHostedTrace()
    return HasArchVersion(ARMv8p4);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/functions/extension/HaveSmallPageTblExt

```
// HaveSmallPageTblExt()
// =====================
// Returns TRUE if Small Page Table Support is implemented.

boolean HaveSmallPageTblExt()
    return HasArchVersion(ARMv8p4) && boolean IMPLEMENTATION_DEFINED "Has Small Page Table extension";
```

shared/functions/extension/HaveStage2MemAttrControl

```
// HaveStage2MemAttrControl()
// ==========================
// Returns TRUE if support for Stage2 control of memory types and cacheability attributes is
implemented.

boolean HaveStage2MemAttrControl()
    return HasArchVersion(ARMv8p4);
```

shared/functions/extension/HaveStatisticalProfiling

```
// HaveStatisticalProfiling()
// ==========================

boolean HaveStatisticalProfiling()
    return HasArchVersion(ARMv8p2);
```

shared/functions/extension/HaveTraceExt

```
// HaveTraceExt()
// ==============
// Returns TRUE if Trace functionality as described by the Trace Architecture
// is implemented.

boolean HaveTraceExt()
    return boolean IMPLEMENTATION_DEFINED "Has Trace Architecture functionality";
```

shared/functions/extension/HaveTrapLoadStoreMultipleDeviceExt

```
// HaveTrapLoadStoreMultipleDeviceExt()
// ====================================

boolean HaveTrapLoadStoreMultipleDeviceExt()
    return HasArchVersion(ARMv8p2);
```

shared/functions/extension/HaveUA16Ext

```
// HaveUA16Ext()
// =============
// Returns TRUE if extended unaligned memory access support is implemented, and FALSE otherwise.

boolean HaveUA16Ext()
    return HasArchVersion(ARMv8p4);
```

shared/functions/extension/HaveUAOExt

```
// HaveUAOExt()
// ============

boolean HaveUAOExt()
    return HasArchVersion(ARMv8p2);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/functions/extension/HaveVirtHostExt

```
// HaveVirtHostExt()
// =================

boolean HaveVirtHostExt()
    return HasArchVersion(ARMv8p1);
```

shared/functions/extension/InsertIESBBeforeException

```
// If SCTLR_ELx.IESB is 1 when an exception is generated to ELx, any pending Unrecoverable
// SError interrupt must be taken before executing any instructions in the exception handler.
// However, this can be before the branch to the exception handler is made.
boolean InsertIESBBeforeException(bits(2) el);
```

shared/functions/float/bfloat/BFAdd

```
// BFAdd()
// =======
// Single-precision add following BFloat16 computation behaviors.

bits(32) BFAdd(bits(32) op1, bits(32) op2)
    bits(32) result;

    (type1,sign1,value1) = BFUnpack(op1);
    (type2,sign2,value2) = BFUnpack(op2);
    if type1 == FPType_QNaN || type2 == FPType_QNaN then
        result = FPDefaultNaN();
    else
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN();
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1');
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1);
        else
            result_value = value1 + value2;
            if result_value == 0.0 then
                result = FPZero('0');    // Positive sign when Round to Odd
            else
                result = BFRound(result_value);

    return result;
```

shared/functions/float/bfloat/BFMatMulAdd

```
// BFMatMulAdd()
// =============
// BFloat16 matrix multiply and add to single-precision matrix
// result[2, 2] = addend[2, 2] + (op1[2, 4] * op2[4, 2])

bits(N) BFMatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2)
    assert N == 128;

    bits(N) result;
    bits(32) sum, prod0, prod1;

    for i = 0 to 1
        for j = 0 to 1
            sum = Elem[addend, 2*i + j, 32];
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            for k = 0 to 1
                prod0 = BFMul(Elem[op1, 4*i + 2*k + 0, 16], Elem[op2, 4*j + 2*k + 0, 16]);
                prod1 = BFMul(Elem[op1, 4*i + 2*k + 1, 16], Elem[op2, 4*j + 2*k + 1, 16]);
                sum   = BFAdd(sum, BFAdd(prod0, prod1));
            Elem[result, 2*i + j, 32] = sum;

    return result;
```

shared/functions/float/bfloat/BFMul

```
// BFMul()
// =======
// BFloat16 widening multiply to single-precision following BFloat16
// computation behaviors.

bits(32) BFMul(bits(16) op1, bits(16) op2)
    bits(32) result;

    (type1,sign1,value1) = BFUnpack(op1);
    (type2,sign2,value2) = BFUnpack(op2);
    if type1 == FPType_QNaN || type2 == FPType_QNaN then
        result = FPDefaultNaN();
    else
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN();
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = BFRound(value1*value2);

    return result;
```

shared/functions/float/bfloat/BFRound

```
// BFRound()
// =========
// Converts a real number OP into a single-precision value using the
// Round to Odd rounding mode and following BFloat16 computation behaviors.

bits(32) BFRound(real op)
    assert op != 0.0;
    bits(32) result;

    // Format parameters - minimum exponent, numbers of exponent and fraction bits.
    minimum_exp = -126;  E = 8;  F = 23;

    // Split value into sign, unrounded mantissa and exponent.
    if op < 0.0 then
        sign = '1';  mantissa = -op;
    else
        sign = '0';  mantissa = op;
    exponent = 0;
    while mantissa < 1.0 do
        mantissa = mantissa * 2.0;  exponent = exponent - 1;
    while mantissa >= 2.0 do
        mantissa = mantissa / 2.0;  exponent = exponent + 1;

    // Fixed Flush-to-zero.
    if exponent < minimum_exp then
        return FPZero(sign);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    // Start creating the exponent value for the result. Start by biasing the actual exponent
    // so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
    biased_exp = Max(exponent - minimum_exp + 1, 0);
    if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

    // Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
    int_mant = RoundDown(mantissa * 2.0^F);  // < 2.0^F if biased_exp == 0, >= 2.0^F if not
    error = mantissa * 2.0^F - Real(int_mant);

    // Round to Odd
    if error != 0.0 then
        int_mant<0> = '1';

    // Deal with overflow and generate result.
    if biased_exp >= 2^E - 1 then
        result = FPInfinity(sign);      // Overflows generate appropriately-signed Infinity
    else
        result = sign : biased_exp<30-F:0> : int_mant<F-1:0>;

    return result;
```

shared/functions/float/bfloat/BFUnpack

```
// BFUnpack()
// ==========
// Unpacks a BFloat16 or single-precision value into its type,
// sign bit and real number that it represents.
// The real number result has the correct sign for numbers and infinities,
// is very large in magnitude for infinities, and is 0.0 for NaNs.
// (These values are chosen to simplify the description of
// comparisons and conversions.)

(FPType, bit, real) BFUnpack(bits(N) fpval)
    assert N IN {16,32};

    if N == 16 then
        sign   = fpval<15>;
        exp    = fpval<14:7>;
        frac   = fpval<6:0> : Zeros(16);
    else  // N == 32
        sign   = fpval<31>;
        exp    = fpval<30:23>;
        frac   = fpval<22:0>;

    if IsZero(exp) then
        fptype = FPType_Zero;  value = 0.0;    // Fixed Flush to Zero
    elsif IsOnes(exp) then
        if IsZero(frac) then
            fptype = FPType_Infinity;  value = 2.0^1000000;
        else    // no SNaN for BF16 arithmetic
            fptype = FPType_QNaN; value = 0.0;
    else
        fptype = FPType_Nonzero;
        value = 2.0^(UInt(exp)-127) * (1.0 + Real(UInt(frac)) * 2.0^-23);

    if sign == '1' then value = -value;

    return (fptype, sign, value);
```

shared/functions/float/bfloat/FPConvertBF

```
// FPConvertBF()
// =============
// Converts a single-precision OP to BFloat16 value with rounding controlled by ROUNDING.
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(16) FPConvertBF(bits(32) op, FPCRType fpcr, FPRounding rounding)
    bits(32) result;    // BF16 value in top 16 bits

    // Unpack floating-point operand optionally with flush-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr);

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        if fpcr.DN == '1' then
            result = FPDefaultNaN();
        else
            result = FPConvertNaN(op);
        if fptype == FPType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpcr);
    elsif fptype == FPType_Infinity then
        result = FPInfinity(sign);
    elsif fptype == FPType_Zero then
        result = FPZero(sign);
    else
        result = FPRoundCVBF(value, fpcr, rounding);

    // Returns correctly rounded BF16 value from top 16 bits
    return result<31:16>;

// FPConvertBF()
// =============
// Converts a single-precision operand to BFloat16 value.

bits(16) FPConvertBF(bits(32) op, FPCRType fpcr)
    return FPConvertBF(op, fpcr, FPRoundingMode(fpcr));
```

shared/functions/float/bfloat/FPRoundCVBF

```
// FPRoundCVBF()
// =============
// Converts a real number OP into a BFloat16 value using the supplied rounding mode RMODE.

bits(32) FPRoundCVBF(real op, FPCRType fpcr, FPRounding rounding)
    boolean isbfloat = TRUE;
    return FPRoundBase(op, fpcr, rounding, isbfloat);
```

shared/functions/float/fixedtofp/FixedToFP

```
// FixedToFP()
// ===========

// Convert M-bit fixed point OP with FBITS fractional bits to
// N-bit precision floating point, controlled by UNSIGNED and ROUNDING.

bits(N) FixedToFP(bits(M) op, integer fbits, boolean unsigned, FPCRType fpcr, FPRounding rounding)
    assert N IN {16,32,64};
    assert M IN {16,32,64};
    bits(N) result;
    assert fbits >= 0;
    assert rounding != FPRounding_ODD;

    // Correct signed-ness
    int_operand = Int(op, unsigned);

    // Scale by fractional bits and generate a real value
    real_operand = Real(int_operand) / 2.0^fbits;

    if real_operand == 0.0 then
        result = FPZero('0');
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        result = FPRound(real_operand, fpcr, rounding);

    return result;
```

shared/functions/float/fpabs/FPAbs

```
// FPAbs()
// =======

bits(N) FPAbs(bits(N) op)
    assert N IN {16,32,64};
    return '0' : op<N-2:0>;
```

shared/functions/float/fpadd/FPAdd

```
// FPAdd()
// =======

bits(N) FPAdd(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1');
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1);
        else
            result_value = value1 + value2;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr, rounding);
    return result;
```

shared/functions/float/fpcompare/FPCompare

```
// FPCompare()
// ===========

bits(4) FPCompare(bits(N) op1, bits(N) op2, boolean signal_nans, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
        result = '0011';
        if type1==FPType_SNaN || type2==FPType_SNaN || signal_nans then
            FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        if value1 == value2 then
            result = '0110';
        elsif value1 < value2 then
            result = '1000';
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        else  // value1 > value2
            result = '0010';
    return result;
```

shared/functions/float/fpcompareeq/FPCompareEQ

```
// FPCompareEQ()
// =============

boolean FPCompareEQ(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
        result = FALSE;
        if type1==FPType_SNaN || type2==FPType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 == value2);
    return result;
```

shared/functions/float/fpcomparege/FPCompareGE

```
// FPCompareGE()
// =============

boolean FPCompareGE(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
        result = FALSE;
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 >= value2);
    return result;
```

shared/functions/float/fpcomparegt/FPCompareGT

```
// FPCompareGT()
// =============

boolean FPCompareGT(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
        result = FALSE;
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 > value2);
    return result;
```

shared/functions/float/fpconvert/FPConvert

```
// FPConvert()
// ===========

// Convert floating point OP with N-bit precision to M-bit precision,
// with rounding controlled by ROUNDING.
// This is used by the FP-to-FP conversion instructions and so for
// half-precision data ignores FZ16, but observes AHP.
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(M) FPConvert(bits(N) op, FPCRType fpcr, FPRounding rounding)
    assert M IN {16,32,64};
    assert N IN {16,32,64};
    bits(M) result;

    // Unpack floating-point operand optionally with flush-to-zero.
    (fptype,sign,value) = FPUnpackCV(op, fpcr);

    alt_hp = (M == 16) && (fpcr.AHP == '1');

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        if alt_hp then
            result = FPZero(sign);
        elsif fpcr.DN == '1' then
            result = FPDefaultNaN();
        else
            result = FPConvertNaN(op);
        if fptype == FPType_SNaN || alt_hp then
            FPProcessException(FPExc_InvalidOp,fpcr);
    elsif fptype == FPType_Infinity then
        if alt_hp then
            result = sign:Ones(M-1);
            FPProcessException(FPExc_InvalidOp, fpcr);
        else
            result = FPInfinity(sign);
    elsif fptype == FPType_Zero then
        result = FPZero(sign);
    else
        result = FPRoundCV(value, fpcr, rounding);
    return result;

// FPConvert()
// ===========

bits(M) FPConvert(bits(N) op, FPCRType fpcr)
    return FPConvert(op, fpcr, FPRoundingMode(fpcr));
```

shared/functions/float/fpconvertnan/FPConvertNaN

```
// FPConvertNaN()
// ==============
// Converts a NaN of one floating-point type to another

bits(M) FPConvertNaN(bits(N) op)
    assert N IN {16,32,64};
    assert M IN {16,32,64};
    bits(M) result;
    bits(51) frac;

    sign = op<N-1>;

    // Unpack payload from input NaN
    case N of
        when 64 frac = op<50:0>;
        when 32 frac = op<21:0>:Zeros(29);
        when 16 frac = op<8:0>:Zeros(42);

    // Repack payload into output NaN, while
    // converting an SNaN to a QNaN.
    case M of
        when 64 result = sign:Ones(M-52):frac;
        when 32 result = sign:Ones(M-23):frac<50:29>;
        when 16 result = sign:Ones(M-10):frac<50:42>;

    return result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/functions/float/fpcrtype/FPCRType

```
type FPCRType;
```

shared/functions/float/fpdecoderm/FPDecodeRM

```
// FPDecodeRM()
// ============

// Decode most common AArch32 floating-point rounding encoding.

FPRounding FPDecodeRM(bits(2) rm)
    case rm of
        when '00' return FPRounding_TIEAWAY; // A
        when '01' return FPRounding_TIEEVEN; // N
        when '10' return FPRounding_POSINF;  // P
        when '11' return FPRounding_NEGINF;  // M
```

shared/functions/float/fpdecoderounding/FPDecodeRounding

```
// FPDecodeRounding()
// ==================

// Decode floating-point rounding mode and common AArch64 encoding.

FPRounding FPDecodeRounding(bits(2) rmode)
    case rmode of
        when '00' return FPRounding_TIEEVEN; // N
        when '01' return FPRounding_POSINF;  // P
        when '10' return FPRounding_NEGINF;  // M
        when '11' return FPRounding_ZERO;    // Z
```

shared/functions/float/fpdefaultnan/FPDefaultNaN

```
// FPDefaultNaN()
// ==============

bits(N) FPDefaultNaN()
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    sign = '0';
    bits(E) exp  = Ones(E);
    bits(F) frac = '1':Zeros(F-1);
    return sign : exp : frac;
```

shared/functions/float/fpdiv/FPDiv

```
// FPDiv()
// =======

bits(N) FPDiv(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && inf2) || (zero1 && zero2) then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif inf1 || zero2 then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            result = FPInfinity(sign1 EOR sign2);
            if !inf1 then FPProcessException(FPExc_DivideByZero, fpcr);
        elsif zero1 || inf2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = FPRound(value1/value2, fpcr);
    return result;
```

shared/functions/float/fpexc/FPExc

```
enumeration FPExc       {FPExc_InvalidOp, FPExc_DivideByZero, FPExc_Overflow,
                         FPExc_Underflow, FPExc_Inexact, FPExc_InputDenorm};
```

shared/functions/float/fpinfinity/FPInfinity

```
// FPInfinity()
// ============

bits(N) FPInfinity(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    bits(E) exp  = Ones(E);
    bits(F) frac = Zeros(F);
    return sign : exp : frac;
```

shared/functions/float/fpmatmul/FPMatMulAdd

```
// FPMatMulAdd()
// =============
//
// Floating point matrix multiply and add to same precision matrix
// result[2, 2] = addend[2, 2] + (op1[2, 2] * op2[2, 2])

bits(N) FPMatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, integer esize, FPCRType fpcr)
    assert N == esize * 2 * 2;
    bits(N)  result;
    bits(esize) prod0, prod1, sum;

    for i = 0 to 1
        for j = 0 to 1
            sum   = Elem[addend, 2*i + j, esize];
            prod0 = FPMul(Elem[op1, 2*i + 0, esize],
                          Elem[op2, 2*j + 0, esize], fpcr);
            prod1 = FPMul(Elem[op1, 2*i + 1, esize],
                          Elem[op2, 2*j + 1, esize], fpcr);
            sum   = FPAdd(sum, FPAdd(prod0, prod1, fpcr), fpcr);
            Elem[result, 2*i + j, esize] = sum;

    return result;
```

shared/functions/float/fpmax/FPMax

```
// FPMax()
// =======

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        if value1 > value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            (fptype,sign,value) = (type2,sign2,value2);
        if fptype == FPType_Infinity then
            result = FPInfinity(sign);
        elsif fptype == FPType_Zero then
            sign = sign1 AND sign2; // Use most positive sign
            result = FPZero(sign);
        else
            // The use of FPRound() covers the case where there is a trapped underflow exception
            // for a denormalized number even though the result is exact.
            result = FPRound(value, fpcr);
    return result;
```

shared/functions/float/fpmaxnormal/FPMaxNormal

```
// FPMaxNormal()
// =============

bits(N) FPMaxNormal(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp  = Ones(E-1):'0';
    frac = Ones(F);
    return sign : exp : frac;
```

shared/functions/float/fpmaxnum/FPMaxNum

```
// FPMaxNum()
// ==========

bits(N) FPMaxNum(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,-,-) = FPUnpack(op1, fpcr);
    (type2,-,-) = FPUnpack(op2, fpcr);

    // treat a single quiet-NaN as -Infinity
    if type1 == FPType_QNaN && type2 != FPType_QNaN then
        op1 = FPInfinity('1');
    elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
        op2 = FPInfinity('1');

    return FPMax(op1, op2, fpcr);
```

shared/functions/float/fpmin/FPMin

```
// FPMin()
// =======

bits(N) FPMin(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        if value1 < value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        if fptype == FPType_Infinity then
            result = FPInfinity(sign);
        elsif fptype == FPType_Zero then
            sign = sign1 OR sign2; // Use most negative sign
            result = FPZero(sign);
        else
            // The use of FPRound() covers the case where there is a trapped underflow exception
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            // for a denormalized number even though the result is exact.
            result = FPRound(value, fpcr);
    return result;
```

shared/functions/float/fpminnum/FPMinNum

```
// FPMinNum()
// ==========

bits(N) FPMinNum(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,-,-) = FPUnpack(op1, fpcr);
    (type2,-,-) = FPUnpack(op2, fpcr);

    // Treat a single quiet-NaN as +Infinity
    if type1 == FPType_QNaN && type2 != FPType_QNaN then
        op1 = FPInfinity('0');
    elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
        op2 = FPInfinity('0');

    return FPMin(op1, op2, fpcr);
```

shared/functions/float/fpmul/FPMul

```
// FPMul()
// =======

bits(N) FPMul(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = FPRound(value1*value2, fpcr);
    return result;
```

shared/functions/float/fpmuladd/FPMulAdd

```
// FPMulAdd()
// ==========
//
// Calculates addend + op1*op2 with a single rounding.

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);
    (typeA,signA,valueA) = FPUnpack(addend, fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    inf1 = (type1 == FPType_Infinity); zero1 = (type1 == FPType_Zero);
    inf2 = (type2 == FPType_Infinity); zero2 = (type2 == FPType_Zero);
    (done,result) = FPProcessNaNs3(typeA, type1, type2, addend, op1, op2, fpcr);

    if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        result = FPDefaultNaN();
        FPProcessException(FPExc_InvalidOp, fpcr);

    if !done then
        infA = (typeA == FPType_Infinity);  zeroA = (typeA == FPType_Zero);

        // Determine sign and type product will have if it does not cause an Invalid
        // Operation.
        signP = sign1 EOR sign2;
        infP  = inf1 || inf2;
        zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero by infinity and
        // additions of opposite-signed infinities.
        if (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP) then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);

        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity('0');
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1');

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero(signA);

        // Otherwise calculate numerical result and round it.
        else
            result_value = valueA + (value1 * value2);
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr);

    return result;
```

shared/functions/float/fpmuladdh/FPMulAddH

```
// FPMulAddH()
// ===========

bits(N) FPMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2, FPCRType fpcr)
    assert N IN {32,64};
    rounding = FPRoundingMode(fpcr);
    (typeA,signA,valueA) = FPUnpack(addend, fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    inf1 = (type1 == FPType_Infinity); zero1 = (type1 == FPType_Zero);
    inf2 = (type2 == FPType_Infinity); zero2 = (type2 == FPType_Zero);
    (done,result) = FPProcessNaNs3H(typeA, type1, type2, addend, op1, op2, fpcr);
    if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
        result = FPDefaultNaN();
        FPProcessException(FPExc_InvalidOp, fpcr);
    if !done then
        infA = (typeA == FPType_Infinity); zeroA = (typeA == FPType_Zero);
        // Determine sign and type product will have if it does not cause an Invalid
        // Operation.
        signP = sign1 EOR sign2;
        infP = inf1 || inf2;
        zeroP = zero1 || zero2;
        // Non SNaN-generated Invalid Operation cases are multiplies of zero by infinity and
        // additions of opposite-signed infinities.
        if (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP) then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity('0');
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1');
        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero(signA);
        // Otherwise calculate numerical result and round it.
        else
            result_value = valueA + (value1 * value2);
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr);
    return result;
```

shared/functions/float/fpmuladdh/FPProcessNaNs3H

```
// FPProcessNaNs3H()
// =================

(boolean, bits(N)) FPProcessNaNs3H(FPType type1, FPType type2, FPType type3, bits(N) op1, bits(N DIV 2)
op2, bits(N DIV 2) op3, FPCRType fpcr)
    assert N IN {32,64};
    bits(N) result;
    if type1 == FPType_SNaN then
        done = TRUE; result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_SNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr));
    elsif type3 == FPType_SNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr));
    elsif type1 == FPType_QNaN then
        done = TRUE; result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_QNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr));
    elsif type3 == FPType_QNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr));
    else
        done = FALSE; result = Zeros(); // 'Don't care' result
    return (done, result);
```

shared/functions/float/fpmulx/FPMulX

```
// FPMulX()
// ========

bits(N) FPMulX(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    bits(N) result;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo(sign1 EOR sign2);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = FPRound(value1*value2, fpcr);
    return result;
```

shared/functions/float/fpneg/FPNeg

```
// FPNeg()
// =======

bits(N) FPNeg(bits(N) op)
    assert N IN {16,32,64};
    return NOT(op<N-1>) : op<N-2:0>;
```

shared/functions/float/fponepointfive/FPOnePointFive

```
// FPOnePointFive()
// ================

bits(N) FPOnePointFive(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp  = '0':Ones(E-1);
    frac = '1':Zeros(F-1);
    return sign : exp : frac;
```

shared/functions/float/fpprocessexception/FPProcessException

```
// FPProcessException()
// ====================
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

FPProcessException(FPExc exception, FPCRType fpcr)
    // Determine the cumulative exception bit number
    case exception of
        when FPExc_InvalidOp     cumul = 0;
        when FPExc_DivideByZero  cumul = 1;
        when FPExc_Overflow      cumul = 2;
        when FPExc_Underflow     cumul = 3;
        when FPExc_Inexact       cumul = 4;
        when FPExc_InputDenorm   cumul = 7;
    enable = cumul + 8;
    if fpcr<enable> == '1' then
        // Trapping of the exception enabled.
        // It is IMPLEMENTATION DEFINED whether the enable bit may be set at all, and
        // if so then how exceptions may be accumulated before calling FPTrappedException()
        IMPLEMENTATION_DEFINED "floating-point trap handling";
    elsif UsingAArch32() then
        // Set the cumulative exception bit
        FPSCR<cumul> = '1';
    else
        // Set the cumulative exception bit
        FPSR<cumul> = '1';
    return;
```

shared/functions/float/fpprocessnan/FPProcessNaN

```
// FPProcessNaN()
// ==============

bits(N) FPProcessNaN(FPType fptype, bits(N) op, FPCRType fpcr)
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    assert N IN {16,32,64};
    assert fptype IN {FPType_QNaN, FPType_SNaN};

    case N of
        when 16 topfrac =  9;
        when 32 topfrac = 22;
        when 64 topfrac = 51;

    result = op;
    if fptype == FPType_SNaN then
        result<topfrac> = '1';
        FPProcessException(FPExc_InvalidOp, fpcr);
    if fpcr.DN == '1' then  // DefaultNaN requested
        result = FPDefaultNaN();
    return result;
```

shared/functions/float/fpprocessnans/FPProcessNaNs

```
// FPProcessNaNs()
// ===============
//
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2,
                                 bits(N) op1, bits(N) op2,
                                 FPCRType fpcr)
    assert N IN {16,32,64};
    if type1 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr);
    elsif type1 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr);
    else
        done = FALSE;  result = Zeros();  // 'Don't care' result
    return (done, result);
```

shared/functions/float/fpprocessnans3/FPProcessNaNs3

```
// FPProcessNaNs3()
// ================
//
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs3(FPType type1, FPType type2, FPType type3,
                                  bits(N) op1, bits(N) op2, bits(N) op3,
                                  FPCRType fpcr)
    assert N IN {16,32,64};
    if type1 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr);
    elsif type3 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type3, op3, fpcr);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif type1 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr);
    elsif type3 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type3, op3, fpcr);
    else
        done = FALSE;  result = Zeros();  // 'Don't care' result
    return (done, result);
```

shared/functions/float/fprecipestimate/FPRecipEstimate

```
// FPRecipEstimate()
// =================

bits(N) FPRecipEstimate(bits(N) operand, FPCRType fpcr)
    assert N IN {16,32,64};
    (fptype,sign,value) = FPUnpack(operand, fpcr);
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, operand, fpcr);
    elsif fptype == FPType_Infinity then
        result = FPZero(sign);
    elsif fptype == FPType_Zero then
        result = FPInfinity(sign);
        FPProcessException(FPExc_DivideByZero, fpcr);
    elsif (
            (N == 16 && Abs(value) < 2.0^-16) ||
            (N == 32 && Abs(value) < 2.0^-128) ||
            (N == 64 && Abs(value) < 2.0^-1024)
          ) then
        case FPRoundingMode(fpcr) of
            when FPRounding_TIEEVEN
                overflow_to_inf = TRUE;
            when FPRounding_POSINF
                overflow_to_inf = (sign == '0');
            when FPRounding_NEGINF
                overflow_to_inf = (sign == '1');
            when FPRounding_ZERO
                overflow_to_inf = FALSE;
        result = if overflow_to_inf then FPInfinity(sign) else FPMaxNormal(sign);
        FPProcessException(FPExc_Overflow, fpcr);
        FPProcessException(FPExc_Inexact, fpcr);
    elsif ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16))
          && (
               (N == 16 && Abs(value) >= 2.0^14) ||
               (N == 32 && Abs(value) >= 2.0^126) ||
               (N == 64 && Abs(value) >= 2.0^1022)
             ) then
        // Result flushed to zero of correct sign
        result = FPZero(sign);
        if UsingAArch32() then
            FPSCR.UFC = '1';
        else
            FPSR.UFC = '1';
    else
        // Scale to a fixed point value in the range 0.5 <= x < 1.0 in steps of 1/512, and
        // calculate result exponent. Scaled value has copied sign bit,
        // exponent = 1022 = double-precision biased version of -1,
        // fraction = original fraction
        case N of
            when 16
                fraction = operand<9:0> : Zeros(42);
                exp = UInt(operand<14:10>);
            when 32
                fraction = operand<22:0> : Zeros(29);
                exp = UInt(operand<30:23>);
            when 64
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            fraction = operand<51:0>;
            exp = UInt(operand<62:52>);

        if exp == 0 then
            if fraction<51> == '0' then
                exp = -1;
                fraction = fraction<49:0>:'00';
            else
                fraction = fraction<50:0>:'0';

        integer scaled = UInt('1':fraction<51:44>);

        case N of
            when 16 result_exp =   29 - exp; // In range 29-30 = -1 to 29+1 = 30
            when 32 result_exp =  253 - exp; // In range 253-254 = -1 to 253+1 = 254
            when 64 result_exp = 2045 - exp; // In range 2045-2046 = -1 to 2045+1 = 2046

        // scaled is in range 256..511 representing a fixed-point number in range [0.5..1.0)
        estimate = RecipEstimate(scaled);

        // estimate is in the range 256..511 representing a fixed point result in the range [1.0..2.0)
        // Convert to scaled floating point result with copied sign bit,
        // high-order bits from estimate, and exponent calculated above.

        fraction = estimate<7:0> : Zeros(44);
        if result_exp == 0 then
            fraction = '1' : fraction<51:1>;
        elsif result_exp == -1 then
            fraction = '01' : fraction<51:2>;
            result_exp = 0;

        case N of
            when 16 result = sign : result_exp<N-12:0> : fraction<51:42>;
            when 32 result = sign : result_exp<N-25:0> : fraction<51:29>;
            when 64 result = sign : result_exp<N-54:0> : fraction<51:0>;

    return result;
```

shared/functions/float/fprecipestimate/RecipEstimate

```
// Compute estimate of reciprocal of 9-bit fixed-point number
//
// a is in range 256 .. 511 representing a number in the range 0.5 <= x < 1.0.
// result is in the range 256 .. 511 representing a number in the range in the range 1.0 to 511/256.

integer RecipEstimate(integer a)
    assert 256 <= a && a < 512;
    a = a*2+1; // round to nearest
    integer b = (2 ^ 19) DIV a;
    r = (b+1) DIV 2; // round to nearest
    assert 256 <= r && r < 512;
    return r;
```

shared/functions/float/fprecpx/FPRecpX

```
// FPRecpX()
// =========

bits(N) FPRecpX(bits(N) op, FPCRType fpcr)
    assert N IN {16,32,64};

    case N of
        when 16 esize =  5;
        when 32 esize =  8;
        when 64 esize = 11;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    bits(N)           result;
    bits(esize)       exp;
    bits(esize)       max_exp;
    bits(N-(esize+1)) frac = Zeros();

    case N of
        when 16 exp = op<10+esize-1:10>;
        when 32 exp = op<23+esize-1:23>;
        when 64 exp = op<52+esize-1:52>;

    max_exp = Ones(esize) - 1;

    (fptype,sign,value) = FPUnpack(op, fpcr);
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, op, fpcr);
    else
        if IsZero(exp) then // Zero and denormals
            result = sign:max_exp:frac;
        else // Infinities and normals
            result = sign:NOT(exp):frac;

    return result;
```

shared/functions/float/fpround/FPRound

```
// FPRound()
// =========
// Used by data processing and int/fixed <-> FP conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.

bits(N) FPRound(real op, FPCRType fpcr, FPRounding rounding)
    fpcr.AHP = '0';
    boolean isbfloat = FALSE;
    return FPRoundBase(op, fpcr, rounding, isbfloat);

// Convert a real number OP into an N-bit floating-point value using the
// supplied rounding mode RMODE.

bits(N) FPRoundBase(real op, FPCRType fpcr, FPRounding rounding, boolean isbfloat)
    assert N IN {16,32,64};
    assert op != 0.0;
    assert rounding != FPRounding_TIEAWAY;
    bits(N) result;

    // Obtain format parameters - minimum exponent, numbers of exponent and fraction bits.
    if N == 16 then
        minimum_exp = -14;  E = 5;  F = 10;
    elsif N == 32 && isbfloat then
        minimum_exp = -126;  E = 8;  F = 7;
    elsif N == 32 then
        minimum_exp = -126;  E = 8;  F = 23;
    else  // N == 64
        minimum_exp = -1022;  E = 11;  F = 52;

    // Split value into sign, unrounded mantissa and exponent.
    if op < 0.0 then
        sign = '1';  mantissa = -op;
    else
        sign = '0';  mantissa = op;
    exponent = 0;
    while mantissa < 1.0 do
        mantissa = mantissa * 2.0;  exponent = exponent - 1;
    while mantissa >= 2.0 do
        mantissa = mantissa / 2.0;  exponent = exponent + 1;

    // Deal with flush-to-zero.
    if ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16)) && exponent < minimum_exp then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    // Flush-to-zero never generates a trapped exception
    if UsingAArch32() then
        FPSCR.UFC = '1';
    else
        FPSR.UFC = '1';
    return FPZero(sign);

// Start creating the exponent value for the result. Start by biasing the actual exponent
// so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
biased_exp = Max(exponent - minimum_exp + 1, 0);
if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

// Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
int_mant = RoundDown(mantissa * 2.0^F);  // < 2.0^F if biased_exp == 0, >= 2.0^F if not
error = mantissa * 2.0^F - Real(int_mant);

// Underflow occurs if exponent is too small before rounding, and result is inexact or
// the Underflow exception is trapped.
if biased_exp == 0 && (error != 0.0 || fpcr.UFE == '1') then
    FPProcessException(FPExc_Underflow, fpcr);

// Round result according to rounding mode.
case rounding of
    when FPRounding_TIEEVEN
        round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
        overflow_to_inf = TRUE;
    when FPRounding_POSINF
        round_up = (error != 0.0 && sign == '0');
        overflow_to_inf = (sign == '0');
    when FPRounding_NEGINF
        round_up = (error != 0.0 && sign == '1');
        overflow_to_inf = (sign == '1');
    when FPRounding_ZERO, FPRounding_ODD
        round_up = FALSE;
        overflow_to_inf = FALSE;

if round_up then
    int_mant = int_mant + 1;
    if int_mant == 2^F then      // Rounded up from denormalized to normalized
        biased_exp = 1;
    if int_mant == 2^(F+1) then  // Rounded up to next exponent
        biased_exp = biased_exp + 1;  int_mant = int_mant DIV 2;

// Handle rounding to odd aka Von Neumann rounding
if error != 0.0 && rounding == FPRounding_ODD then
    int_mant<0> = '1';

// Deal with overflow and generate result.
if N != 16 || fpcr.AHP == '0' then  // Single, double or IEEE half precision
    if biased_exp >= 2^E - 1 then
        result = if overflow_to_inf then FPInfinity(sign) else FPMaxNormal(sign);
        FPProcessException(FPExc_Overflow, fpcr);
        error = 1.0;  // Ensure that an Inexact exception occurs
    else
        result = sign : biased_exp<E-1:0> : int_mant<F-1:0> : Zeros(N-(E+F+1));
else                                    // Alternative half precision
    if biased_exp >= 2^E then
        result = sign : Ones(N-1);
        FPProcessException(FPExc_InvalidOp, fpcr);
        error = 0.0;  // Ensure that an Inexact exception does not occur
    else
        result = sign : biased_exp<E-1:0> : int_mant<F-1:0> : Zeros(N-(E+F+1));

// Deal with Inexact exception.
if error != 0.0 then
    FPProcessException(FPExc_Inexact, fpcr);

return result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
// FPRound()
// =========

bits(N) FPRound(real op, FPCRType fpcr)
    return FPRound(op, fpcr, FPRoundingMode(fpcr));
```

shared/functions/float/fpround/FPRoundCV

```
// FPRoundCV()
// ===========
// Used for FP <-> FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

bits(N) FPRoundCV(real op, FPCRType fpcr, FPRounding rounding)
    fpcr.FZ16 = '0';
    boolean isbfloat = FALSE;
    return FPRoundBase(op, fpcr, rounding, isbfloat);
```

shared/functions/float/fprounding/FPRounding

```
enumeration FPRounding  {FPRounding_TIEEVEN, FPRounding_POSINF,
                         FPRounding_NEGINF,  FPRounding_ZERO,
                         FPRounding_TIEAWAY, FPRounding_ODD};
```

shared/functions/float/fproundingmode/FPRoundingMode

```
// FPRoundingMode()
// ================

// Return the current floating-point rounding mode.

FPRounding FPRoundingMode(FPCRType fpcr)
    return FPDecodeRounding(fpcr.RMode);
```

shared/functions/float/fproundint/FPRoundInt

```
// FPRoundInt()
// ============

// Round OP to nearest integral floating point value using rounding mode ROUNDING.
// If EXACT is TRUE, set FPSR.IXC if result is not numerically equal to OP.

bits(N) FPRoundInt(bits(N) op, FPCRType fpcr, FPRounding rounding, boolean exact)
    assert rounding != FPRounding_ODD;
    assert N IN {16,32,64};

    // Unpack using FPCR to determine if subnormals are flushed-to-zero
    (fptype,sign,value) = FPUnpack(op, fpcr);

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, op, fpcr);
    elsif fptype == FPType_Infinity then
        result = FPInfinity(sign);
    elsif fptype == FPType_Zero then
        result = FPZero(sign);
    else
        // extract integer component
        int_result = RoundDown(value);
        error = value - Real(int_result);

        // Determine whether supplied rounding mode requires an increment
        case rounding of
            when FPRounding_TIEEVEN
                round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            when FPRounding_POSINF
                round_up = (error != 0.0);
            when FPRounding_NEGINF
                round_up = FALSE;
            when FPRounding_ZERO
                round_up = (error != 0.0 && int_result < 0);
            when FPRounding_TIEAWAY
                round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

        if round_up then int_result = int_result + 1;

        // Convert integer value into an equivalent real value
        real_result = Real(int_result);

        // Re-encode as a floating-point value, result is always exact
        if real_result == 0.0 then
            result = FPZero(sign);
        else
            result = FPRound(real_result, fpcr, FPRounding_ZERO);

        // Generate inexact exceptions
        if error != 0.0 && exact then
            FPProcessException(FPExc_Inexact, fpcr);

    return result;
```

shared/functions/float/fproundintn/FPRoundIntN

```
// FPRoundIntN()
// =============

bits(N) FPRoundIntN(bits(N) op, FPCRType fpcr, FPRounding rounding, integer intsize)
    assert rounding != FPRounding_ODD;
    assert N IN {32,64};
    assert intsize IN {32, 64};
    integer exp;
    constant integer E = (if N == 32 then 8 else 11);
    constant integer F = N - (E + 1);

    // Unpack using FPCR to determine if subnormals are flushed-to-zero
    (fptype,sign,value) = FPUnpack(op, fpcr);

    if fptype IN {FPType_SNaN, FPType_QNaN, FPType_Infinity} then
        if N == 32 then
            exp = 126 + intsize;
            result = '1':exp<(E-1):0>:Zeros(F);
        else
            exp = 1022+intsize;
            result = '1':exp<(E-1):0>:Zeros(F);
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif fptype == FPType_Zero then
        result = FPZero(sign);
    else
        // Extract integer component
        int_result = RoundDown(value);
        error = value - Real(int_result);

        // Determine whether supplied rounding mode requires an increment
        case rounding of
            when FPRounding_TIEEVEN
                round_up = error > 0.5 || (error == 0.5 && int_result<0> == '1');
            when FPRounding_POSINF
                round_up = error != 0.0;
            when FPRounding_NEGINF
                round_up = FALSE;
            when FPRounding_ZERO
                round_up = error != 0.0 && int_result < 0;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            when FPRounding_TIEAWAY
                round_up = error > 0.5 || (error == 0.5 && int_result >= 0);

        if round_up then int_result = int_result + 1;

        if int_result > 2^(intsize-1)-1 || int_result < -1*2^(intsize-1) then
            if N == 32 then
                exp = 126 + intsize;
                result = '1':exp<(E-1):0>:Zeros(F);
            else
                exp = 1022 + intsize;
                result = '1':exp<(E-1):0>:Zeros(F);
            FPProcessException(FPExc_InvalidOp, fpcr);
            // this case shouldn't set Inexact
            error = 0.0;

        else
            // Convert integer value into an equivalent real value
            real_result = Real(int_result);

            // Re-encode as a floating-point value, result is always exact
            if real_result == 0.0 then
                result = FPZero(sign);
            else
                result = FPRound(real_result, fpcr, FPRounding_ZERO);

        // Generate inexact exceptions
        if error != 0.0 then
            FPProcessException(FPExc_Inexact, fpcr);

    return result;
```

shared/functions/float/fprsqrtestimate/FPRSqrtEstimate

```
// FPRSqrtEstimate()
// =================

bits(N) FPRSqrtEstimate(bits(N) operand, FPCRType fpcr)
    assert N IN {16,32,64};
    (fptype,sign,value) = FPUnpack(operand, fpcr);
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, operand, fpcr);
    elsif fptype == FPType_Zero then
        result = FPInfinity(sign);
        FPProcessException(FPExc_DivideByZero, fpcr);
    elsif sign == '1' then
        result = FPDefaultNaN();
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif fptype == FPType_Infinity then
        result = FPZero('0');
    else
        // Scale to a fixed-point value in the range 0.25 <= x < 1.0 in steps of 512, with the
        // evenness or oddness of the exponent unchanged, and calculate result exponent.
        // Scaled value has copied sign bit, exponent = 1022 or 1021 = double-precision
        // biased version of -1 or -2, fraction = original fraction extended with zeros.

        case N of
            when 16
                fraction = operand<9:0> : Zeros(42);
                exp = UInt(operand<14:10>);
            when 32
                fraction = operand<22:0> : Zeros(29);
                exp = UInt(operand<30:23>);
            when 64
                fraction = operand<51:0>;
                exp = UInt(operand<62:52>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        if exp == 0 then
            while fraction<51> == '0' do
                fraction = fraction<50:0> : '0';
                exp = exp - 1;
            fraction = fraction<50:0> : '0';

        if exp<0> == '0' then
            scaled = UInt('1':fraction<51:44>);
        else
            scaled = UInt('01':fraction<51:45>);

        case N of
            when 16 result_exp = (  44 - exp) DIV 2;
            when 32 result_exp = ( 380 - exp) DIV 2;
            when 64 result_exp = (3068 - exp) DIV 2;

        estimate = RecipSqrtEstimate(scaled);

        // estimate is in the range 256..511 representing a fixed point result in the range [1.0..2.0)
        // Convert to scaled floating point result with copied sign bit and high-order
        // fraction bits, and exponent calculated above.
        case N of
            when 16 result = '0' : result_exp<N-12:0> : estimate<7:0>:Zeros( 2);
            when 32 result = '0' : result_exp<N-25:0> : estimate<7:0>:Zeros(15);
            when 64 result = '0' : result_exp<N-54:0> : estimate<7:0>:Zeros(44);
    return result;
```

shared/functions/float/fprsqrtestimate/RecipSqrtEstimate

```
// Compute estimate of reciprocal square root of 9-bit fixed-point number
//
// a is in range 128 .. 511 representing a number in the range 0.25 <= x < 1.0.
// result is in the range 256 .. 511 representing a number in the range in the range 1.0 to 511/256.

integer RecipSqrtEstimate(integer a)
    assert 128 <= a && a < 512;
    if a < 256 then // 0.25 .. 0.5
        a = a*2+1;     // a in units of 1/512 rounded to nearest
    else // 0.5 .. 1.0
        a = (a >> 1) << 1;   // discard bottom bit
        a = (a+1)*2;  // a in units of 1/256 rounded to nearest
    integer b = 512;
    while a*(b+1)*(b+1) < 2^28 do
        b = b+1;
    // b = largest b such that b < 2^14 / sqrt(a) do
    r = (b+1) DIV 2; // round to nearest
    assert 256 <= r && r < 512;
    return r;
```

shared/functions/float/fpsqrt/FPSqrt

```
// FPSqrt()
// ========

bits(N) FPSqrt(bits(N) op, FPCRType fpcr)
    assert N IN {16,32,64};
    (fptype,sign,value) = FPUnpack(op, fpcr);
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, op, fpcr);
    elsif fptype == FPType_Zero then
        result = FPZero(sign);
    elsif fptype == FPType_Infinity && sign == '0' then
        result = FPInfinity(sign);
    elsif sign == '1' then
        result = FPDefaultNaN();
        FPProcessException(FPExc_InvalidOp, fpcr);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        result = FPRound(Sqrt(value), fpcr);
    return result;
```

shared/functions/float/fpsub/FPSub

```
// FPSub()
// =======

bits(N) FPSub(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1');
        elsif zero1 && zero2 && sign1 == NOT(sign2) then
            result = FPZero(sign1);
        else
            result_value = value1 - value2;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr, rounding);
    return result;
```

shared/functions/float/fpthree/FPThree

```
// FPThree()
// =========

bits(N) FPThree(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp  = '1':Zeros(E-1);
    frac = '1':Zeros(F-1);
    return sign : exp : frac;
```

shared/functions/float/fptofixed/FPToFixed

```
// FPToFixed()
// ===========

// Convert N-bit precision floating point OP to M-bit fixed point with
// FBITS fractional bits, controlled by UNSIGNED and ROUNDING.

bits(M) FPToFixed(bits(N) op, integer fbits, boolean unsigned, FPCRType fpcr, FPRounding rounding)
    assert N IN {16,32,64};
    assert M IN {16,32,64};
    assert fbits >= 0;
    assert rounding != FPRounding_ODD;

    // Unpack using fpcr to determine if subnormals are flushed-to-zero
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    (fptype,sign,value) = FPUnpack(op, fpcr);

    // If NaN, set cumulative flag or take exception
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);

    // Scale by fractional bits and produce integer rounded towards minus-infinity
    value = value * 2.0^fbits;
    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment
    case rounding of
        when FPRounding_TIEEVEN
            round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
        when FPRounding_POSINF
            round_up = (error != 0.0);
        when FPRounding_NEGINF
            round_up = FALSE;
        when FPRounding_ZERO
            round_up = (error != 0.0 && int_result < 0);
        when FPRounding_TIEAWAY
            round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

    if round_up then int_result = int_result + 1;

    // Generate saturated result and exceptions
    (result, overflow) = SatQ(int_result, M, unsigned);
    if overflow then
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif error != 0.0 then
        FPProcessException(FPExc_Inexact, fpcr);

    return result;
```

shared/functions/float/fptofixedjs/FPToFixedJS

```
// FPToFixedJS()
// =============

// Converts a double precision floating point input value
// to a signed integer, with rounding to zero.

(bits(N), bit) FPToFixedJS(bits(M) op, FPCRType fpcr, boolean Is64)

    assert M == 64 && N == 32;

    // Unpack using fpcr to determine if subnormals are flushed-to-zero
    (fptype,sign,value) = FPUnpack(op, fpcr);

    Z = '1';
    // If NaN, set cumulative flag or take exception
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);
        Z = '0';

    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment

    round_it_up = (error != 0.0 && int_result < 0);
    if round_it_up then int_result = int_result + 1;

    if int_result < 0 then
        result = int_result - 2^32*RoundUp(Real(int_result)/Real(2^32));
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        result = int_result - 2^32*RoundDown(Real(int_result)/Real(2^32));

    // Generate exceptions
    if int_result < -(2^31) || int_result > (2^31)-1 then
        FPProcessException(FPExc_InvalidOp, fpcr);
        Z = '0';
    elsif error != 0.0 then
        FPProcessException(FPExc_Inexact, fpcr);
        Z = '0';
    elsif sign == '1' && value == 0.0 then
        Z = '0';
    elsif sign == '0' && value == 0.0 && !IsZero(op<51:0>) then
        Z = '0';

    if fptype == FPType_Infinity then result = 0;

    return (result<N-1:0>, Z);
```

shared/functions/float/fptwo/FPTwo

```
// FPTwo()
// =======

bits(N) FPTwo(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp  = '1':Zeros(E-1);
    frac = Zeros(F);
    return sign : exp : frac;
```

shared/functions/float/fptype/FPType

```
enumeration FPType      {FPType_Nonzero, FPType_Zero, FPType_Infinity,
                         FPType_QNaN, FPType_SNaN};
```

shared/functions/float/fpunpack/FPUnpack

```
// FPUnpack()
// ==========
//
// Used by data processing and int/fixed <-> FP conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.

(FPType, bit, real) FPUnpack(bits(N) fpval, FPCRType fpcr)
    fpcr.AHP = '0';
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr);
    return (fp_type, sign, value);
```

shared/functions/float/fpunpack/FPUnpackBase

```
// FPUnpackBase()
// ==============
//
// Unpack a floating-point number into its type, sign bit and the real number
// that it represents. The real number result has the correct sign for numbers
// and infinities, is very large in magnitude for infinities, and is 0.0 for
// NaNs. (These values are chosen to simplify the description of comparisons
// and conversions.)
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

(FPType, bit, real) FPUnpackBase(bits(N) fpval, FPCRType fpcr)
    assert N IN {16,32,64};
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if N == 16 then
    sign   = fpval<15>;
    exp16  = fpval<14:10>;
    frac16 = fpval<9:0>;
    if IsZero(exp16) then
        // Produce zero if value is zero or flush-to-zero is selected
        if IsZero(frac16) || fpcr.FZ16 == '1' then
            fptype = FPType_Zero;  value = 0.0;
        else
            fptype = FPType_Nonzero;  value = 2.0^-14 * (Real(UInt(frac16)) * 2.0^-10);
    elsif IsOnes(exp16) && fpcr.AHP == '0' then  // Infinity or NaN in IEEE format
        if IsZero(frac16) then
            fptype = FPType_Infinity;  value = 2.0^1000000;
        else
            fptype = if frac16<9> == '1' then FPType_QNaN else FPType_SNaN;
            value = 0.0;
    else
        fptype = FPType_Nonzero;
        value = 2.0^(UInt(exp16)-15) * (1.0 + Real(UInt(frac16)) * 2.0^-10);

elsif N == 32 then

    sign   = fpval<31>;
    exp32  = fpval<30:23>;
    frac32 = fpval<22:0>;
    if IsZero(exp32) then
        // Produce zero if value is zero or flush-to-zero is selected.
        if IsZero(frac32) || fpcr.FZ == '1' then
            fptype = FPType_Zero;  value = 0.0;
            if !IsZero(frac32) then  // Denormalized input flushed to zero
                FPProcessException(FPExc_InputDenorm, fpcr);
        else
            fptype = FPType_Nonzero;  value = 2.0^-126 * (Real(UInt(frac32)) * 2.0^-23);
    elsif IsOnes(exp32) then
        if IsZero(frac32) then
            fptype = FPType_Infinity;  value = 2.0^1000000;
        else
            fptype = if frac32<22> == '1' then FPType_QNaN else FPType_SNaN;
            value = 0.0;
    else
        fptype = FPType_Nonzero;
        value = 2.0^(UInt(exp32)-127) * (1.0 + Real(UInt(frac32)) * 2.0^-23);

else // N == 64

    sign   = fpval<63>;
    exp64  = fpval<62:52>;
    frac64 = fpval<51:0>;
    if IsZero(exp64) then
        // Produce zero if value is zero or flush-to-zero is selected.
        if IsZero(frac64) || fpcr.FZ == '1' then
            fptype = FPType_Zero;  value = 0.0;
            if !IsZero(frac64) then  // Denormalized input flushed to zero
                FPProcessException(FPExc_InputDenorm, fpcr);
        else
            fptype = FPType_Nonzero;  value = 2.0^-1022 * (Real(UInt(frac64)) * 2.0^-52);
    elsif IsOnes(exp64) then
        if IsZero(frac64) then
            fptype = FPType_Infinity;  value = 2.0^1000000;
        else
            fptype = if frac64<51> == '1' then FPType_QNaN else FPType_SNaN;
            value = 0.0;
    else
        fptype = FPType_Nonzero;
        value = 2.0^(UInt(exp64)-1023) * (1.0 + Real(UInt(frac64)) * 2.0^-52);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if sign == '1' then value = -value;
    return (fptype, sign, value);
```

shared/functions/float/fpunpack/FPUnpackCV

```
// FPUnpackCV()
// ============
//
// Used for FP <-> FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

(FPType, bit, real) FPUnpackCV(bits(N) fpval, FPCRType fpcr)
    fpcr.FZ16 = '0';
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr);
    return (fp_type, sign, value);
```

shared/functions/float/fpzero/FPZero

```
// FPZero()
// ========

bits(N) FPZero(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp  = Zeros(E);
    frac = Zeros(F);
    return sign : exp : frac;
```

shared/functions/float/vfpexpandimm/VFPExpandImm

```
// VFPExpandImm()
// ==============

bits(N) VFPExpandImm(bits(8) imm8)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    sign = imm8<7>;
    exp  = NOT(imm8<6>):Replicate(imm8<6>,E-3):imm8<5:4>;
    frac = imm8<3:0>:Zeros(F-4);
    return sign : exp : frac;
```

shared/functions/integer/AddWithCarry

```
// AddWithCarry()
// ==============
// Integer addition with carry input, returning result and NZCV flags

(bits(N), bits(4)) AddWithCarry(bits(N) x, bits(N) y, bit carry_in)
    integer unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
    integer signed_sum = SInt(x) + SInt(y) + UInt(carry_in);
    bits(N) result = unsigned_sum<N-1:0>; // same value as signed_sum<N-1:0>
    bit n = result<N-1>;
    bit z = if IsZero(result) then '1' else '0';
    bit c = if UInt(result) == unsigned_sum then '0' else '1';
    bit v = if SInt(result) == signed_sum then '0' else '1';
    return (result, n:z:c:v);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/functions/memory/AArch64.BranchAddr

```
// AArch64.BranchAddr()
// ====================
// Return the virtual address with tag bits removed for storing to the program counter.

bits(64) AArch64.BranchAddr(bits(64) vaddress)
    assert !UsingAArch32();
    msbit = AddrTop(vaddress, TRUE, PSTATE.EL);
    if msbit == 63 then
        return vaddress;
    elsif (PSTATE.EL IN {EL0, EL1} || IsInHost()) && vaddress<msbit> == '1' then
        return SignExtend(vaddress<msbit:0>);
    else
        return ZeroExtend(vaddress<msbit:0>);
```

shared/functions/memory/AccType

```
enumeration AccType {AccType_NORMAL, AccType_VEC,        // Normal loads and stores
                     AccType_STREAM, AccType_VECSTREAM,  // Streaming loads and stores
                     AccType_ATOMIC, AccType_ATOMICRW,   // Atomic loads and stores
                     AccType_ORDERED, AccType_ORDEREDRW, // Load-Acquire and Store-Release
                     AccType_ORDEREDATOMIC,              // Load-Acquire and Store-Release with atomic
access
                     AccType_ORDEREDATOMICRW,
                     AccType_LIMITEDORDERED,             // Load-LOAcquire and Store-LORelease
                     AccType_UNPRIV,                     // Load and store unprivileged
                     AccType_IFETCH,                     // Instruction fetch
                     AccType_PTW,                        // Page table walk
                     AccType_NONFAULT,                   // Non-faulting loads
                     AccType_CNOTFIRST,                  // Contiguous FF load, not first element
                     AccType_NV2REGISTER,                // MRS/MSR instruction used at EL1 and which is
converted
                                                         // to a memory access that uses the EL2
translation regime
                     // Other operations
                     AccType_DC,                         // Data cache maintenance
                     AccType_DC_UNPRIV,                  // Data cache maintenance instruction used at
EL0
                     AccType_IC,                         // Instruction cache maintenance
                     AccType_DCZVA,                      // DC ZVA instructions
                     AccType_AT};                        // Address translation
```

shared/functions/memory/AccessDescriptor

```
type AccessDescriptor is (
    AccType acctype,
    MPAMinfo mpam,
    boolean page_table_walk,
    boolean secondstage,
    boolean s2fs1walk,
    integer level
)
```

shared/functions/memory/AddrTop

```
// AddrTop()
// =========
// Return the MSB number of a virtual address in the stage 1 translation regime for "el".
// If EL1 is using AArch64 then addresses from EL0 using AArch32 are zero-extended to 64 bits.

integer AddrTop(bits(64) address, boolean IsInstr, bits(2) el)
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    if ELUsingAArch32(regime) then
        // AArch32 translation regime.
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        return 31;
    else
        // AArch64 translation regime.
        case regime of
            when EL1
                tbi = (if address<55> == '1' then TCR_EL1.TBI1 else TCR_EL1.TBI0);
                if HavePACExt() then
                    tbid = if address<55> == '1' then TCR_EL1.TBID1 else TCR_EL1.TBID0;
            when EL2
                if HaveVirtHostExt() && ELIsInHost(el) then
                    tbi = (if address<55> == '1' then TCR_EL2.TBI1 else TCR_EL2.TBI0);
                    if HavePACExt() then
                        tbid = if address<55> == '1' then TCR_EL2.TBID1 else TCR_EL2.TBID0;
                else
                    tbi = TCR_EL2.TBI;
                    if HavePACExt() then tbid = TCR_EL2.TBID;
            when EL3
                tbi = TCR_EL3.TBI;
                if HavePACExt() then tbid = TCR_EL3.TBID;

    return (if tbi == '1' && (!HavePACExt() || tbid == '0' || !IsInstr ) then 55 else 63);
```

shared/functions/memory/AddressDescriptor

```
type AddressDescriptor is (
    FaultRecord      fault,    // fault.statuscode indicates whether the address is valid
    MemoryAttributes memattrs,
    FullAddress      paddress,
    bits(64)         vaddress
)
```

shared/functions/memory/Allocation

```
constant bits(2) MemHint_No = '00';     // No Read-Allocate, No Write-Allocate
constant bits(2) MemHint_WA = '01';     // No Read-Allocate, Write-Allocate
constant bits(2) MemHint_RA = '10';     // Read-Allocate, No Write-Allocate
constant bits(2) MemHint_RWA = '11';    // Read-Allocate, Write-Allocate
```

shared/functions/memory/BigEndian

```
// BigEndian()
// ===========

boolean BigEndian()
    boolean bigend;
    if UsingAArch32() then
        bigend = (PSTATE.E != '0');
    elsif PSTATE.EL == EL0 then
        bigend = (SCTLR[].E0E != '0');
    else
        bigend = (SCTLR[].EE != '0');
    return bigend;
```

shared/functions/memory/BigEndianReverse

```
// BigEndianReverse()
// ==================

bits(width) BigEndianReverse (bits(width) value)
    assert width IN {8, 16, 32, 64, 128};
    integer half = width DIV 2;
    if width == 8 then return value;
    return BigEndianReverse(value<half-1:0>) : BigEndianReverse(value<width-1:half>);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/functions/memory/Cacheability

```
constant bits(2) MemAttr_NC = '00';     // Non-cacheable
constant bits(2) MemAttr_WT = '10';     // Write-through
constant bits(2) MemAttr_WB = '11';     // Write-back
```

shared/functions/memory/CreateAccessDescriptor

```
// CreateAccessDescriptor()
// ========================

AccessDescriptor CreateAccessDescriptor(AccType acctype)
    AccessDescriptor accdesc;
    accdesc.acctype = acctype;
    accdesc.mpam = GenMPAMcurEL(acctype IN {AccType_IFETCH, AccType_IC});
    accdesc.page_table_walk = FALSE;
    return accdesc;
```

shared/functions/memory/CreateAccessDescriptorPTW

```
// CreateAccessDescriptorPTW()
// ===========================

AccessDescriptor CreateAccessDescriptorPTW(AccType acctype, boolean secondstage,
                                           boolean s2fs1walk, integer level)
    AccessDescriptor accdesc;
    accdesc.acctype = acctype;
    accdesc.mpam = GenMPAMcurEL(acctype IN {AccType_IFETCH, AccType_IC});
    accdesc.page_table_walk = TRUE;
    accdesc.s2fs1walk = s2fs1walk;
    accdesc.secondstage = secondstage;
    accdesc.level = level;
    return accdesc;
```

shared/functions/memory/DataMemoryBarrier

```
DataMemoryBarrier(MBReqDomain domain, MBReqTypes types);
```

shared/functions/memory/DataSynchronizationBarrier

```
DataSynchronizationBarrier(MBReqDomain domain, MBReqTypes types);
```

shared/functions/memory/DescriptorUpdate

```
type DescriptorUpdate is (
    boolean AF,                 // AF needs to be set
    boolean AP,                 // AP[2] / S2AP[2] will be modified
    AddressDescriptor descaddr  // Descriptor to be updated
)
```

shared/functions/memory/DeviceType

```
enumeration DeviceType {DeviceType_GRE, DeviceType_nGRE, DeviceType_nGnRE, DeviceType_nGnRnE};
```

shared/functions/memory/EffectiveTBI

```
// EffectiveTBI()
// ==============
// Returns the effective TBI in the AArch64 stage 1 translation regime for "el".

bit EffectiveTBI(bits(64) address, boolean IsInstr, bits(2) el)
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    assert(!ELUsingAArch32(regime));

    case regime of
        when EL1
            tbi = if address<55> == '1' then TCR_EL1.TBI1 else TCR_EL1.TBI0;
            if HavePACExt() then
                tbid = if address<55> == '1' then TCR_EL1.TBID1 else TCR_EL1.TBID0;
        when EL2
            if HaveVirtHostExt() && ELIsInHost(el) then
                tbi = if address<55> == '1' then TCR_EL2.TBI1 else TCR_EL2.TBI0;
                if HavePACExt() then
                    tbid = if address<55> == '1' then TCR_EL2.TBID1 else TCR_EL2.TBID0;
            else
                tbi = TCR_EL2.TBI;
                if HavePACExt() then tbid = TCR_EL2.TBID;
        when EL3
            tbi = TCR_EL3.TBI;
            if HavePACExt() then tbid = TCR_EL3.TBID;

    return (if tbi == '1' && (!HavePACExt() || tbid == '0' || !IsInstr) then '1' else '0');
```

shared/functions/memory/EffectiveTCMA

```
// EffectiveTCMA()
// ===============
// Returns the effective TCMA of a virtual address in the stage 1 translation regime for "el".

bit EffectiveTCMA(bits(64) address, bits(2) el)
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));

    case regime of
        when EL1
            tcma = if address<55> == '1' then TCR_EL1.TCMA1 else TCR_EL1.TCMA0;
        when EL2
            if HaveVirtHostExt() && ELIsInHost(el) then
                tcma = if address<55> == '1' then TCR_EL2.TCMA1 else TCR_EL2.TCMA0;
            else
                tcma = TCR_EL2.TCMA;
        when EL3
            tcma = TCR_EL3.TCMA;

    return tcma;
```

shared/functions/memory/Fault

```
enumeration Fault {Fault_None,
                   Fault_AccessFlag,
                   Fault_Alignment,
                   Fault_Background,
                   Fault_Domain,
                   Fault_Permission,
                   Fault_Translation,
                   Fault_AddressSize,
                   Fault_SyncExternal,
                   Fault_SyncExternalOnWalk,
                   Fault_SyncParity,
                   Fault_SyncParityOnWalk,
                   Fault_AsyncParity,
                   Fault_AsyncExternal,
                   Fault_Debug,
                   Fault_TLBConflict,
                   Fault_BranchTarget,
                   Fault_HWUpdateAccessFlag,
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
                    Fault_Lockdown,
                    Fault_Exclusive,
                    Fault_ICacheMaint};
```

shared/functions/memory/FaultRecord

```
type FaultRecord is (Fault    statuscode,  // Fault Status
                     AccType  acctype,      // Type of access that faulted
                     FullAddress ipaddress, // Intermediate physical address
                     boolean  s2fs1walk,    // Is on a Stage 1 page table walk
                     boolean  write,        // TRUE for a write, FALSE for a read
                     integer  level,        // For translation, access flag and permission faults
                     bit      extflag,      // IMPLEMENTATION DEFINED syndrome for external aborts
                     boolean  secondstage,  // Is a Stage 2 abort
                     bits(4)  domain,       // Domain number, AArch32 only
                     bits(2)  errortype,    // [Armv8.2 RAS] AArch32 AET or AArch64 SET
                     bits(4)  debugmoe)     // Debug method of entry, from AArch32 only

type PARTIDtype = bits(16);
type PMGtype = bits(8);

type MPAMinfo is (
     bit mpam_ns,
     PARTIDtype partid,
     PMGtype pmg
)
```

shared/functions/memory/FullAddress

```
type FullAddress is (
    bits(52) address,
    bit      NS                  // '0' = Secure, '1' = Non-secure
)
```

shared/functions/memory/Hint_Prefetch

```
// Signals the memory system that memory accesses of type HINT to or from the specified address are
// likely in the near future. The memory system may take some action to speed up the memory
// accesses when they do occur, such as pre-loading the the specified address into one or more
// caches as indicated by the innermost cache level target (0=L1, 1=L2, etc) and non-temporal hint
// stream. Any or all prefetch hints may be treated as a NOP. A prefetch hint must not cause a
// synchronous abort due to Alignment or Translation faults and the like. Its only effect on
// software-visible state should be on caches and TLBs associated with address, which must be
// accessible by reads, writes or execution, as defined in the translation regime of the current
// Exception level. It is guaranteed not to access Device memory.
// A Prefetch_EXEC hint must not result in an access that could not be performed by a speculative
// instruction fetch, therefore if all associated MMUs are disabled, then it cannot access any
// memory location that cannot be accessed by instruction fetches.
Hint_Prefetch(bits(64) address, PrefetchHint hint, integer target, boolean stream);
```

shared/functions/memory/MBReqDomain

```
enumeration MBReqDomain    {MBReqDomain_Nonshareable, MBReqDomain_InnerShareable,
                            MBReqDomain_OuterShareable, MBReqDomain_FullSystem};
```

shared/functions/memory/MBReqTypes

```
enumeration MBReqTypes     {MBReqTypes_Reads, MBReqTypes_Writes, MBReqTypes_All};
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/functions/memory/MemAttrHints

```
type MemAttrHints is (
    bits(2) attrs,  // See MemAttr_*, Cacheability attributes
    bits(2) hints,  // See MemHint_*, Allocation hints
    boolean transient
)
```

shared/functions/memory/MemType

```
enumeration MemType {MemType_Normal, MemType_Device};
```

shared/functions/memory/MemoryAttributes

```
type MemoryAttributes is (
    MemType      memtype,

    DeviceType   device,     // For Device memory types
    MemAttrHints inner,      // Inner hints and attributes
    MemAttrHints outer,      // Outer hints and attributes
    boolean      tagged,     // Tagged access
    boolean      shareable,
    boolean      outershareable
)
```

shared/functions/memory/Permissions

```
type Permissions is (
    bits(3) ap,   // Access permission bits
    bit     xn,   // Execute-never bit
    bit     xxn,  // [Armv8.2] Extended execute-never bit for stage 2
    bit     pxn   // Privileged execute-never bit
)
```

shared/functions/memory/PrefetchHint

```
enumeration PrefetchHint {Prefetch_READ, Prefetch_WRITE, Prefetch_EXEC};
```

shared/functions/memory/SpeculativeStoreBypassBarrierToPA

```
SpeculativeStoreBypassBarrierToPA();
```

shared/functions/memory/SpeculativeStoreBypassBarrierToVA

```
SpeculativeStoreBypassBarrierToVA();
```

shared/functions/memory/TLBRecord

```
type TLBRecord is (
    Permissions       perms,
    bit               nG,             // '0' = Global, '1' = not Global
    bits(4)           domain,         // AArch32 only
    bit               GP,             // Guarded Page
    boolean           contiguous,     // Contiguous bit from page table
    integer           level,          // AArch32 Short-descriptor format: Indicates Section/Page
    integer           blocksize,      // Describes size of memory translated in KBytes
    DescriptorUpdate  descupdate,     // [Armv8.1] Context for h/w update of table descriptor
    bit               CnP,            // [Armv8.2] TLB entry can be shared between different PEs
    AddressDescriptor addrdesc
)
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/functions/memory/Tag

```
constant integer LOG2_TAG_GRANULE = 4;

constant integer TAG_GRANULE = 1 << LOG2_TAG_GRANULE;
```

shared/functions/memory/_Mem

```
// These two _Mem[] accessors are the hardware operations which perform single-copy atomic,
// aligned, little-endian memory accesses of size bytes from/to the underlying physical
// memory array of bytes.
//
// The functions address the array using desc.paddress which supplies:
// * A 52-bit physical address
// * A single NS bit to select between Secure and Non-secure parts of the array.
//
// The accdesc descriptor describes the access type: normal, exclusive, ordered, streaming,
// etc and other parameters required to access the physical memory or for setting syndrome
// register in the event of an external abort.
bits(8*size) _Mem[AddressDescriptor desc, integer size, AccessDescriptor accdesc];

_Mem[AddressDescriptor desc, integer size, AccessDescriptor accdesc] = bits(8*size) value;
```

shared/functions/mpam/DefaultMPAMinfo

```
// DefaultMPAMinfo
// ===============
// Returns default MPAM info.  If secure is TRUE return default Secure
// MPAMinfo, otherwise return default Non-secure MPAMinfo.

MPAMinfo DefaultMPAMinfo(boolean secure)
    MPAMinfo DefaultInfo;
    DefaultInfo.mpam_ns = if secure then '0' else '1';
    DefaultInfo.partid  = DefaultPARTID;
    DefaultInfo.pmg     = DefaultPMG;
    return DefaultInfo;
```

shared/functions/mpam/DefaultPARTID

```
constant PARTIDtype DefaultPARTID = 0<15:0>;
```

shared/functions/mpam/DefaultPMG

```
constant PMGtype    DefaultPMG = 0<7:0>;
```

shared/functions/mpam/GenMPAMcurEL

```
// GenMPAMcurEL
// ============
// Returns MPAMinfo for the current EL and security state.
// InD is TRUE instruction access and FALSE otherwise.
// May be called if MPAM is not implemented (but in an version that supports
// MPAM), MPAM is disabled, or in AArch32.  In AArch32, convert the mode to
// EL if can and use that to drive MPAM information generation.  If mode
// cannot be converted, MPAM is not implemented, or MPAM is disabled return
// default MPAM information for the current security state.

MPAMinfo GenMPAMcurEL(boolean InD)
    bits(2) mpamel;
    boolean validEL;
    boolean securempam;
    if HaveEMPAMExt() then
        boolean secure = IsSecure();
        securempam = MPAM3_EL3.FORCE_NS == '0' && secure;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        if MPAMisEnabled() && (!secure || MPAM3_EL3.SDEFLT == '0') then
            if UsingAArch32() then
                (validEL, mpamel) = ELFromM32(PSTATE.M);
            else
                validEL = TRUE;
                mpamel = PSTATE.EL;
            if validEL then
                return genMPAM(UInt(mpamel), InD, securempam);
    else
        securempam = IsSecure();
        if HaveMPAMExt() && MPAMisEnabled() then
            if UsingAArch32() then
                (validEL, mpamel) = ELFromM32(PSTATE.M);
            else
                validEL = TRUE;
                mpamel = PSTATE.EL;
            if validEL then
                return genMPAM(UInt(mpamel), InD, securempam);
    return DefaultMPAMinfo(securempam);
```

shared/functions/mpam/MAP_vPARTID

```
// MAP_vPARTID
// ===========
// Performs conversion of virtual PARTID into physical PARTID
// Contains all of the error checking and implementation
// choices for the conversion.

(PARTIDtype, boolean) MAP_vPARTID(PARTIDtype vpartid)
    // should not ever be called if EL2 is not implemented
    // or is implemented but not enabled in the current
    // security state.
    PARTIDtype ret;
    boolean err;
    integer virt    = UInt( vpartid );
    integer vpmrmax = UInt( MPAMIDR_EL1.VPMR_MAX );

    // vpartid_max is largest vpartid supported
    integer vpartid_max = (4 * vpmrmax) + 3;

    // One of many ways to reduce vpartid to value less than vpartid_max.
    if virt > vpartid_max then
        virt = virt MOD (vpartid_max+1);

    // Check for valid mapping entry.
    if MPAMVPMV_EL2<virt> == '1' then
        // vpartid has a valid mapping so access the map.
        ret = mapvpmw(virt);
        err = FALSE;

    // Is the default virtual PARTID valid?
    elsif MPAMVPMV_EL2<0> == '1' then
        // Yes, so use default mapping for vpartid == 0.
        ret = MPAMVPM0_EL2<0 +: 16>;
        err = FALSE;

    // Neither is valid so use default physical PARTID.
    else
        ret = DefaultPARTID;
        err = TRUE;

    // Check that the physical PARTID is in-range.
    // This physical PARTID came from a virtual mapping entry.
    integer partid_max = UInt( MPAMIDR_EL1.PARTID_MAX );
    if UInt(ret) > partid_max then
        // Out of range, so return default physical PARTID
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        ret = DefaultPARTID;
        err = TRUE;
    return (ret, err);
```

shared/functions/mpam/MPAMisEnabled

```
// MPAMisEnabled
// =============
// Returns TRUE if MPAMisEnabled.

boolean MPAMisEnabled()
    el = HighestEL();
    case el of
        when EL3 return MPAM3_EL3.MPAMEN == '1';
        when EL2 return MPAM2_EL2.MPAMEN == '1';
        when EL1 return MPAM1_EL1.MPAMEN == '1';
```

shared/functions/mpam/MPAMisVirtual

```
// MPAMisVirtual
// =============
// Returns TRUE if MPAM is configured to be virtual at EL.

boolean MPAMisVirtual(integer el)
    return ( MPAMIDR_EL1.HAS_HCR == '1' && EL2Enabled() &&
            ( HCR_EL2.E2H == '0' || HCR_EL2.TGE == '0' ) &&
            (( el == 0 && MPAMHCR_EL2.EL0_VPMEN == '1' ) ||
             ( el == 1 && MPAMHCR_EL2.EL1_VPMEN == '1')));
```

shared/functions/mpam/genMPAM

```
// genMPAM
// =======
// Returns MPAMinfo for exception level el.
// If InD is TRUE returns MPAM information using PARTID_I and PMG_I fields
// of MPAMel_ELx register and otherwise using PARTID_D and PMG_D fields.
// Produces a Secure PARTID if Secure is TRUE and a Non-secure PARTID otherwise.

MPAMinfo genMPAM(integer el, boolean InD, boolean secure)
    MPAMinfo returnInfo;
    PARTIDtype partidel;
    boolean perr;
    boolean gstplk = (el == 0 && EL2Enabled() &&
                      MPAMHCR_EL2.GSTAPP_PLK == '1' && HCR_EL2.TGE == '0');
    integer eff_el = if gstplk then 1 else el;
    (partidel, perr) = genPARTID(eff_el, InD);
    PMGtype groupel  = genPMG(eff_el, InD, perr);
    returnInfo.mpam_ns = if secure then '0' else '1';
    returnInfo.partid  = partidel;
    returnInfo.pmg     = groupel;
    return returnInfo;
```

shared/functions/mpam/genMPAMel

```
// genMPAMel
// =========
// Returns MPAMinfo for specified EL in the current security state.
// InD is TRUE for instruction access and FALSE otherwise.

MPAMinfo genMPAMel(bits(2) el, boolean InD)
    boolean secure = IsSecure();
    boolean securempam = secure;
    if HaveEMPAMExt() then
        securempam = MPAM3_EL3.FORCE_NS == '0' && secure;
        if HaveMPAMExt() && MPAMisEnabled() && (!secure || MPAM3_EL3.SDEFLT == '0') then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            return genMPAM(UInt(el), InD, securempam);
    else
        if HaveMPAMExt() && MPAMisEnabled() then
            return genMPAM(UInt(el), InD, securempam);
    return DefaultMPAMinfo(securempam);
```

shared/functions/mpam/genPARTID

```
// genPARTID
// =========
// Returns physical PARTID and error boolean for exception level el.
// If InD is TRUE then PARTID is from MPAMel_ELx.PARTID_I and
// otherwise from MPAMel_ELx.PARTID_D.

(PARTIDtype, boolean) genPARTID(integer el, boolean InD)
    PARTIDtype partidel = getMPAM_PARTID(el, InD);

    integer partid_max = UInt(MPAMIDR_EL1.PARTID_MAX);
    if UInt(partidel) > partid_max then
        return (DefaultPARTID, TRUE);

    if MPAMisVirtual(el) then
        return MAP_vPARTID(partidel);
    else
        return (partidel, FALSE);
```

shared/functions/mpam/genPMG

```
// genPMG
// ======
// Returns PMG for exception level el and I- or D-side (InD).
// If PARTID generation (genPARTID) encountered an error, genPMG() should be
// called with partid_err as TRUE.

PMGtype genPMG(integer el, boolean InD, boolean partid_err)
    integer pmg_max = UInt(MPAMIDR_EL1.PMG_MAX);

    // It is CONSTRAINED UNPREDICTABLE whether partid_err forces PMG to
    // use the default or if it uses the PMG from getMPAM_PMG.
    if partid_err then
        return DefaultPMG;
    PMGtype groupel = getMPAM_PMG(el, InD);
    if UInt(groupel) <= pmg_max then
        return groupel;
    return DefaultPMG;
```

shared/functions/mpam/getMPAM_PARTID

```
// getMPAM_PARTID
// ==============
// Returns a PARTID from one of the MPAMn_ELx registers.
// MPAMn selects the MPAMn_ELx register used.
// If InD is TRUE, selects the PARTID_I field of that
// register.  Otherwise, selects the PARTID_D field.

PARTIDtype getMPAM_PARTID(integer MPAMn, boolean InD)
    PARTIDtype partid;
    boolean el2avail = EL2Enabled();

    if InD then
        case MPAMn of
            when 3 partid = MPAM3_EL3.PARTID_I;
            when 2 partid = if el2avail then MPAM2_EL2.PARTID_I else Zeros();
            when 1 partid = MPAM1_EL1.PARTID_I;
            when 0 partid = MPAM0_EL1.PARTID_I;
            otherwise partid = PARTIDtype UNKNOWN;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        case MPAMn of
            when 3 partid = MPAM3_EL3.PARTID_D;
            when 2 partid = if el2avail then MPAM2_EL2.PARTID_D else Zeros();
            when 1 partid = MPAM1_EL1.PARTID_D;
            when 0 partid = MPAM0_EL1.PARTID_D;
            otherwise partid = PARTIDtype UNKNOWN;
    return partid;
```

shared/functions/mpam/getMPAM_PMG

```
// getMPAM_PMG
// ===========
// Returns a PMG from one of the MPAMn_ELx registers.
// MPAMn selects the MPAMn_ELx register used.
// If InD is TRUE, selects the PMG_I field of that
// register.  Otherwise, selects the PMG_D field.

PMGtype getMPAM_PMG(integer MPAMn, boolean InD)
    PMGtype pmg;
    boolean el2avail = EL2Enabled();

    if InD then
        case MPAMn of
            when 3 pmg = MPAM3_EL3.PMG_I;
            when 2 pmg = if el2avail then MPAM2_EL2.PMG_I else Zeros();
            when 1 pmg = MPAM1_EL1.PMG_I;
            when 0 pmg = MPAM0_EL1.PMG_I;
            otherwise pmg = PMGtype UNKNOWN;
    else
        case MPAMn of
            when 3 pmg = MPAM3_EL3.PMG_D;
            when 2 pmg = if el2avail then MPAM2_EL2.PMG_D else Zeros();
            when 1 pmg = MPAM1_EL1.PMG_D;
            when 0 pmg = MPAM0_EL1.PMG_D;
            otherwise pmg = PMGtype UNKNOWN;
    return pmg;
```

shared/functions/mpam/mapvpmw

```
// mapvpmw
// =======
// Map a virtual PARTID into a physical PARTID using
// the MPAMVPMn_EL2 registers.
// vpartid is now assumed in-range and valid (checked by caller)
// returns physical PARTID from mapping entry.

PARTIDtype mapvpmw(integer vpartid)
    bits(64) vpmw;
    integer wd = vpartid DIV 4;
    case wd of
        when 0 vpmw = MPAMVPM0_EL2;
        when 1 vpmw = MPAMVPM1_EL2;
        when 2 vpmw = MPAMVPM2_EL2;
        when 3 vpmw = MPAMVPM3_EL2;
        when 4 vpmw = MPAMVPM4_EL2;
        when 5 vpmw = MPAMVPM5_EL2;
        when 6 vpmw = MPAMVPM6_EL2;
        when 7 vpmw = MPAMVPM7_EL2;
        otherwise vpmw = Zeros(64);
    // vpme_lsb selects LSB of field within register
    integer vpme_lsb = (vpartid MOD 4) * 16;
    return vpmw<vpme_lsb +: 16>;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/functions/registers/BranchTo

```
// BranchTo()
// ==========

// Set program counter to a new address, with a branch type
// In AArch64 state the address might include a tag in the top eight bits.

BranchTo(bits(N) target, BranchType branch_type)
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target);
    else
        assert N == 64 && !UsingAArch32();
        _PC = AArch64.BranchAddr(target<63:0>);
    return;
```

shared/functions/registers/BranchToAddr

```
// BranchToAddr()
// ==============

// Set program counter to a new address, with a branch type
// In AArch64 state the address does not include a tag in the top eight bits.

BranchToAddr(bits(N) target, BranchType branch_type)
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target);
    else
        assert N == 64 && !UsingAArch32();
        _PC = target<63:0>;
    return;
```

shared/functions/registers/BranchType

```
enumeration BranchType {
    BranchType_DIRCALL,     // Direct Branch with link
    BranchType_INDCALL,     // Indirect Branch with link
    BranchType_ERET,        // Exception return (indirect)
    BranchType_DBGEXIT,     // Exit from Debug state
    BranchType_RET,         // Indirect branch with function return hint
    BranchType_DIR,         // Direct branch
    BranchType_INDIR,       // Indirect branch
    BranchType_EXCEPTION,   // Exception entry
    BranchType_RESET,       // Reset
    BranchType_UNKNOWN};    // Other
```

shared/functions/registers/Hint_Branch

```
// Report the hint passed to BranchTo() and BranchToAddr(), for consideration when processing
// the next instruction.
Hint_Branch(BranchType hint);
```

shared/functions/registers/NextInstrAddr

```
// Return address of the sequentially next instruction.
bits(N) NextInstrAddr();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/functions/registers/ResetExternalDebugRegisters

```
// Reset the External Debug registers in the Core power domain.
ResetExternalDebugRegisters(boolean cold_reset);
```

shared/functions/registers/ThisInstrAddr

```
// ThisInstrAddr()
// ===============
// Return address of the current instruction.

bits(N) ThisInstrAddr()
    assert N == 64 || (N == 32 && UsingAArch32());
    return _PC<N-1:0>;
```

shared/functions/registers/_PC

```
bits(64) _PC;
```

shared/functions/registers/_R

```
array bits(64) _R[0..30];
```

shared/functions/sysregisters/SPSR

```
// SPSR[] - non-assignment form
// ============================

bits(32) SPSR[]
    bits(32) result;
    if UsingAArch32() then
        case PSTATE.M of
            when M32_FIQ      result = SPSR_fiq;
            when M32_IRQ      result = SPSR_irq;
            when M32_Svc      result = SPSR_svc;
            when M32_Monitor  result = SPSR_mon;
            when M32_Abort    result = SPSR_abt;
            when M32_Hyp      result = SPSR_hyp;
            when M32_Undef    result = SPSR_und;
            otherwise         Unreachable();
    else
        case PSTATE.EL of
            when EL1          result = SPSR_EL1;
            when EL2          result = SPSR_EL2;
            when EL3          result = SPSR_EL3;
            otherwise         Unreachable();
    return result;

// SPSR[] - assignment form
// ========================

SPSR[] = bits(32) value
    if UsingAArch32() then
        case PSTATE.M of
            when M32_FIQ      SPSR_fiq = value;
            when M32_IRQ      SPSR_irq = value;
            when M32_Svc      SPSR_svc = value;
            when M32_Monitor  SPSR_mon = value;
            when M32_Abort    SPSR_abt = value;
            when M32_Hyp      SPSR_hyp = value;
            when M32_Undef    SPSR_und = value;
            otherwise         Unreachable();
    else
        case PSTATE.EL of
            when EL1          SPSR_EL1 = value;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            when EL2          SPSR_EL2 = value;
            when EL3          SPSR_EL3 = value;
            otherwise         Unreachable();
    return;
```

shared/functions/system/ArchVersion

```
enumeration ArchVersion {
    ARMv8p0
    , ARMv8p1
    , ARMv8p2
    , ARMv8p3
    , ARMv8p4
    , ARMv8p5
    , ARMv8p6
};
```

shared/functions/system/BranchTargetCheck

```
// BranchTargetCheck()
// ===================
// This function is executed checks if the current instruction is a valid target for a branch
// taken into, or inside, a guarded page. It is executed on every cycle once the current
// instruction has been decoded and the values of InGuardedPage and BTypeCompatible have been
// determined for the current instruction.

BranchTargetCheck()
    assert HaveBTIExt() && !UsingAArch32();

    // The branch target check considers two state variables:
    // * InGuardedPage, which is evaluated during instruction fetch.
    // * BTypeCompatible, which is evaluated during instruction decode.
    if InGuardedPage && PSTATE.BTYPE != '00' && !BTypeCompatible && !Halted() then
        bits(64) pc = ThisInstrAddr();
        AArch64.BranchTargetException(pc<51:0>);

    boolean branch_instr = AArch64.ExecutingBROrBLROrRetInstr();
    boolean bti_instr    = AArch64.ExecutingBTIInstr();

    // PSTATE.BTYPE defaults to 00 for instructions that do not explictly set BTYPE.
    if !(branch_instr || bti_instr) then
        BTypeNext = '00';
```

shared/functions/system/ClearEventRegister

```
// ClearEventRegister()
// ====================
// Clear the Event Register of this PE

ClearEventRegister()
    EventRegister = '0';
    return;
```

shared/functions/system/ClearPendingPhysicalSError

```
// Clear a pending physical SError interrupt
ClearPendingPhysicalSError();
```

shared/functions/system/ClearPendingVirtualSError

```
// Clear a pending virtual SError interrupt
ClearPendingVirtualSError();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/functions/system/ConditionHolds

```
// ConditionHolds()
// ================
// Return TRUE iff COND currently holds

boolean ConditionHolds(bits(4) cond)
    // Evaluate base condition.
    case cond<3:1> of
        when '000' result = (PSTATE.Z == '1');                          // EQ or NE
        when '001' result = (PSTATE.C == '1');                          // CS or CC
        when '010' result = (PSTATE.N == '1');                          // MI or PL
        when '011' result = (PSTATE.V == '1');                          // VS or VC
        when '100' result = (PSTATE.C == '1' && PSTATE.Z == '0');       // HI or LS
        when '101' result = (PSTATE.N == PSTATE.V);                     // GE or LT
        when '110' result = (PSTATE.N == PSTATE.V && PSTATE.Z == '0');  // GT or LE
        when '111' result = TRUE;                                       // AL

    // Condition flag values in the set '111x' indicate always true
    // Otherwise, invert condition if necessary.
    if cond<0> == '1' && cond != '1111' then
        result = !result;

    return result;
```

shared/functions/system/ConsumptionOfSpeculativeDataBarrier

```
ConsumptionOfSpeculativeDataBarrier();
```

shared/functions/system/CurrentInstrSet

```
// CurrentInstrSet()
// =================

InstrSet CurrentInstrSet()

    if UsingAArch32() then
        result = if PSTATE.T == '0' then InstrSet_A32 else InstrSet_T32;
        // PSTATE.J is RES0. Implementation of T32EE or Jazelle state not permitted.
    else
        result = InstrSet_A64;
    return result;
```

shared/functions/system/CurrentPL

```
// CurrentPL()
// ===========

PrivilegeLevel CurrentPL()
    return PLOfEL(PSTATE.EL);
```

shared/functions/system/EL0

```
constant bits(2) EL3 = '11';
constant bits(2) EL2 = '10';
constant bits(2) EL1 = '01';
constant bits(2) EL0 = '00';
```

shared/functions/system/EL2Enabled

```
// EL2Enabled()
// ============
// Returns TRUE if EL2 is present and executing
// - with SCR_EL3.NS==1 when Non-secure EL2 is implemented, or
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
// - with SCR_EL3.NS==0 when Secure EL2 is implemented and enabled, or
// - when EL3 is not implemented.

boolean EL2Enabled()
    return HaveEL(EL2) && (!HaveEL(EL3) || SCR_EL3.NS == '1' || IsSecureEL2Enabled());
```

shared/functions/system/ELFromM32

```
// ELFromM32()
// ===========

(boolean,bits(2)) ELFromM32(bits(5) mode)
    // Convert an AArch32 mode encoding to an Exception level.
    // Returns (valid,EL):
    //   'valid' is TRUE if 'mode<4:0>' encodes a mode that is both valid for this implementation
    //           and the current value of SCR.NS/SCR_EL3.NS.
    //   'EL'    is the Exception level decoded from 'mode'.
    bits(2) el;
    boolean valid = !BadMode(mode);  // Check for modes that are not valid for this implementation
    case mode of
        when M32_Monitor
            el = EL3;
        when M32_Hyp
            el = EL2;
            valid = valid && (!HaveEL(EL3) || SCR_GEN[].NS == '1');
        when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32_Undef, M32_System
            // If EL3 is implemented and using AArch32, then these modes are EL3 modes in Secure
            // state, and EL1 modes in Non-secure state. If EL3 is not implemented or is using
            // AArch64, then these modes are EL1 modes.
            el = (if HaveEL(EL3) && HighestELUsingAArch32() && SCR.NS == '0' then EL3 else EL1);
        when M32_User
            el = EL0;
        otherwise
            valid = FALSE;           // Passed an illegal mode value
    if !valid then el = bits(2) UNKNOWN;
    return (valid, el);
```

shared/functions/system/ELFromSPSR

```
// ELFromSPSR()
// ============

// Convert an SPSR value encoding to an Exception level.
// Returns (valid,EL):
//   'valid' is TRUE if 'spsr<4:0>' encodes a valid mode for the current state.
//   'EL'    is the Exception level decoded from 'spsr'.

(boolean,bits(2)) ELFromSPSR(bits(32) spsr)
    if spsr<4> == '0' then                      // AArch64 state
        el = spsr<3:2>;
        if HighestELUsingAArch32() then         // No AArch64 support
            valid = FALSE;
        elsif !HaveEL(el) then                  // Exception level not implemented
            valid = FALSE;
        elsif spsr<1> == '1' then               // M[1] must be 0
            valid = FALSE;
        elsif el == EL0 && spsr<0> == '1' then  // for EL0, M[0] must be 0
            valid = FALSE;
        elsif el == EL2 && HaveEL(EL3) && !IsSecureEL2Enabled() && SCR_EL3.NS == '0' then
            valid = FALSE;                      // Unless Secure EL2 is enabled, EL2 only valid in
Non-secure state
        else
            valid = TRUE;
    elsif HaveAnyAArch32() then                // AArch32 state
        (valid, el) = ELFromM32(spsr<4:0>);
    else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        valid = FALSE;

    if !valid then el = bits(2) UNKNOWN;
    return (valid,el);
```

shared/functions/system/ELIsInHost

```
// ELIsInHost()
// ============

boolean ELIsInHost(bits(2) el)
    return ((IsSecureEL2Enabled() || !IsSecureBelowEL3()) && HaveVirtHostExt() && !ELUsingAArch32(EL2)
&&
            HCR_EL2.E2H == '1' && (el == EL2 || (el == EL0 && HCR_EL2.TGE == '1')));
```

shared/functions/system/ELStateUsingAArch32

```
// ELStateUsingAArch32()
// =====================

boolean ELStateUsingAArch32(bits(2) el, boolean secure)
    // See ELStateUsingAArch32K() for description. Must only be called in circumstances where
    // result is valid (typically, that means 'el IN {EL1,EL2,EL3}').
    (known, aarch32) = ELStateUsingAArch32K(el, secure);
    assert known;
    return aarch32;
```

shared/functions/system/ELStateUsingAArch32K

```
// ELStateUsingAArch32K()
// ======================

(boolean,boolean) ELStateUsingAArch32K(bits(2) el, boolean secure)
    // Returns (known, aarch32):
    //   'known'   is FALSE for EL0 if the current Exception level is not EL0 and EL1 is
    //             using AArch64, since it cannot determine the state of EL0; TRUE otherwise.
    //   'aarch32' is TRUE if the specified Exception level is using AArch32; FALSE otherwise.
    if !HaveAArch32EL(el) then
        return (TRUE, FALSE);                     // Exception level is using AArch64
    elsif secure && el == EL2 then
        return (TRUE, FALSE);                     // Secure EL2 is using AArch64
    elsif HighestELUsingAArch32() then
        return (TRUE, TRUE);                      // Highest Exception level, and therefore all levels
are using AArch32
    elsif el == HighestEL() then
        return (TRUE, FALSE);                     // This is highest Exception level, so is using
AArch64

    // Remainder of function deals with the interprocessing cases when highest Exception level is using
AArch64

    boolean aarch32 = boolean UNKNOWN;
    boolean known = TRUE;

    aarch32_below_el3 = HaveEL(EL3) && SCR_EL3.RW == '0' && (!secure || !HaveSecureEL2Ext() ||
SCR_EL3.EEL2 == '0');
    aarch32_at_el1 = (aarch32_below_el3 || (HaveEL(EL2) &&
                                        ((HaveSecureEL2Ext() && SCR_EL3.EEL2 == '1') || !secure) &&
HCR_EL2.RW == '0' &&
                                        !(HCR_EL2.E2H == '1' && HCR_EL2.TGE == '1' &&
HaveVirtHostExt())));
    if el == EL0 && !aarch32_at_el1 then       // Only know if EL0 using AArch32 from PSTATE
        if PSTATE.EL == EL0 then
            aarch32 = PSTATE.nRW == '1';       // EL0 controlled by PSTATE
        else
            known = FALSE;                     // EL0 state is UNKNOWN
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        aarch32 = (aarch32_below_el3 && el != EL3) || (aarch32_at_el1 && el IN {EL1,EL0});

    if !known then aarch32 = boolean UNKNOWN;
    return (known, aarch32);
```

shared/functions/system/ELUsingAArch32

```
// ELUsingAArch32()
// ================

boolean ELUsingAArch32(bits(2) el)
    return ELStateUsingAArch32(el, IsSecureBelowEL3());
```

shared/functions/system/ELUsingAArch32K

```
// ELUsingAArch32K()
// =================

(boolean,boolean) ELUsingAArch32K(bits(2) el)
    return ELStateUsingAArch32K(el, IsSecureBelowEL3());
```

shared/functions/system/EndOfInstruction

```
// Terminate processing of the current instruction.
EndOfInstruction();
```

shared/functions/system/EnterLowPowerState

```
// PE enters a low-power state
EnterLowPowerState();
```

shared/functions/system/EventRegister

```
bits(1) EventRegister;
```

shared/functions/system/GetPSRFromPSTATE

```
// GetPSRFromPSTATE()
// ==================
// Return a PSR value which represents the current PSTATE

bits(32) GetPSRFromPSTATE()
    bits(32) spsr = Zeros();
    spsr<31:28> = PSTATE.<N,Z,C,V>;
    if HavePANExt() then spsr<22> = PSTATE.PAN;
    spsr<20>    = PSTATE.IL;
    if PSTATE.nRW == '1' then // AArch32 state
        spsr<27>    = PSTATE.Q;
        spsr<26:25> = PSTATE.IT<1:0>;
        if HaveSSBSExt() then spsr<23> = PSTATE.SSBS;
        if HaveDITExt() then spsr<21> = PSTATE.DIT;
        spsr<19:16> = PSTATE.GE;
        spsr<15:10> = PSTATE.IT<7:2>;
        spsr<9>     = PSTATE.E;
        spsr<8:6>   = PSTATE.<A,I,F>;                   // No PSTATE.D in AArch32 state
        spsr<5>     = PSTATE.T;
        assert PSTATE.M<4> == PSTATE.nRW;               // bit [4] is the discriminator
        spsr<4:0>   = PSTATE.M;
    else // AArch64 state
        if HaveMTEExt() then spsr<25> = PSTATE.TCO;
        if HaveDITExt() then spsr<24> = PSTATE.DIT;
        if HaveUAOExt() then spsr<23> = PSTATE.UAO;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        spsr<21>    = PSTATE.SS;
        if HaveSSBSExt() then spsr<12> = PSTATE.SSBS;
        if HaveBTIExt() then spsr<11:10> = PSTATE.BTYPE;
        spsr<9:6>   = PSTATE.<D,A,I,F>;
        spsr<4>     = PSTATE.nRW;
        spsr<3:2>   = PSTATE.EL;
        spsr<0>     = PSTATE.SP;
    return spsr;
```

shared/functions/system/HasArchVersion

```
// HasArchVersion()
// ================
// Return TRUE if the implemented architecture includes the extensions defined in the specified
// architecture version.

boolean HasArchVersion(ArchVersion version)
    return version == ARMv8p0 || boolean IMPLEMENTATION_DEFINED;
```

shared/functions/system/HaveAArch32EL

```
// HaveAArch32EL()
// ===============

boolean HaveAArch32EL(bits(2) el)
    // Return TRUE if Exception level 'el' supports AArch32 in this implementation
    if !HaveEL(el) then
        return FALSE;                   // The Exception level is not implemented
    elsif !HaveAnyAArch32() then
        return FALSE;                   // No Exception level can use AArch32
    elsif HighestELUsingAArch32() then
        return TRUE;                    // All Exception levels are using AArch32
    elsif el == HighestEL() then
        return FALSE;                   // The highest Exception level is using AArch64
    elsif el == EL0 then
        return TRUE;                    // EL0 must support using AArch32 if any AArch32
    return boolean IMPLEMENTATION_DEFINED;
```

shared/functions/system/HaveAnyAArch32

```
// HaveAnyAArch32()
// ================
// Return TRUE if AArch32 state is supported at any Exception level

boolean HaveAnyAArch32()
    return boolean IMPLEMENTATION_DEFINED;
```

shared/functions/system/HaveAnyAArch64

```
// HaveAnyAArch64()
// ================
// Return TRUE if AArch64 state is supported at any Exception level

boolean HaveAnyAArch64()
    return !HighestELUsingAArch32();
```

shared/functions/system/HaveEL

```
// HaveEL()
// ========
// Return TRUE if Exception level 'el' is supported

boolean HaveEL(bits(2) el)
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if el IN {EL1,EL0} then
        return TRUE;                            // EL1 and EL0 must exist
    return boolean IMPLEMENTATION_DEFINED;
```

shared/functions/system/HaveELUsingSecurityState

```
// HaveELUsingSecurityState()
// ==========================
// Returns TRUE if Exception level 'el' with Security state 'secure' is supported,
// FALSE otherwise.

boolean HaveELUsingSecurityState(bits(2) el, boolean secure)

    case el of
        when EL3
            assert secure;
            return HaveEL(EL3);
        when EL2
            if secure then
                return HaveEL(EL2) && HaveSecureEL2Ext();
            else
                return HaveEL(EL2);
        otherwise
            return (HaveEL(EL3) ||
                    (secure == boolean IMPLEMENTATION_DEFINED "Secure-only implementation"));
```

shared/functions/system/HaveFP16Ext

```
// HaveFP16Ext()
// =============
// Return TRUE if FP16 extension is supported

boolean HaveFP16Ext()
    return boolean IMPLEMENTATION_DEFINED;
```

shared/functions/system/HighestEL

```
// HighestEL()
// ===========
// Returns the highest implemented Exception level.

bits(2) HighestEL()
    if HaveEL(EL3) then
        return EL3;
    elsif HaveEL(EL2) then
        return EL2;
    else
        return EL1;
```

shared/functions/system/HighestELUsingAArch32

```
// HighestELUsingAArch32()
// =======================
// Return TRUE if configured to boot into AArch32 operation

boolean HighestELUsingAArch32()
    if !HaveAnyAArch32() then return FALSE;
    return boolean IMPLEMENTATION_DEFINED;       // e.g. CFG32SIGNAL == HIGH
```

shared/functions/system/Hint_DGH

```
// Provides a hint to close any gathering occurring within the micro-architecture.
Hint_DGH();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/functions/system/Hint_Yield

```
// Provides a hint that the task performed by a thread is of low
// importance so that it could yield to improve overall performance.
Hint_Yield();
```

shared/functions/system/IllegalExceptionReturn

```
// IllegalExceptionReturn()
// ========================

boolean IllegalExceptionReturn(bits(32) spsr)

    // Check for illegal return:
    //   * To an unimplemented Exception level.
    //   * To EL2 in Secure state, when SecureEL2 is not enabled.
    //   * To EL0 using AArch64 state, with SPSR.M[0]==1.
    //   * To AArch64 state with SPSR.M[1]==1.
    //   * To AArch32 state with an illegal value of SPSR.M.
    (valid, target) = ELFromSPSR(spsr);
    if !valid then return TRUE;

    // Check for return to higher Exception level
    if UInt(target) > UInt(PSTATE.EL) then return TRUE;

    spsr_mode_is_aarch32 = (spsr<4> == '1');

    // Check for illegal return:
    //   * To EL1, EL2 or EL3 with register width specified in the SPSR different from the
    //     Execution state used in the Exception level being returned to, as determined by
    //     the SCR_EL3.RW or HCR_EL2.RW bits, or as configured from reset.
    //   * To EL0 using AArch64 state when EL1 is using AArch32 state as determined by the
    //     SCR_EL3.RW or HCR_EL2.RW bits or as configured from reset.
    //   * To AArch64 state from AArch32 state (should be caught by above)
    (known, target_el_is_aarch32) = ELUsingAArch32K(target);
    assert known || (target == EL0 && !ELUsingAArch32(EL1));
    if known && spsr_mode_is_aarch32 != target_el_is_aarch32 then return TRUE;

    // Check for illegal return from AArch32 to AArch64
    if UsingAArch32() && !spsr_mode_is_aarch32 then return TRUE;

    // Check for illegal return to EL1 when HCR.TGE is set and when either of
    // * SecureEL2 is enabled.
    // * SecureEL2 is not enabled and EL1 is in Non-secure state.
    if HaveEL(EL2) && target == EL1 && HCR_EL2.TGE == '1' then
        if (!IsSecureBelowEL3() || IsSecureEL2Enabled()) then return TRUE;
    return FALSE;
```

shared/functions/system/InstrSet

```
enumeration InstrSet {InstrSet_A64, InstrSet_A32, InstrSet_T32};
```

shared/functions/system/InstructionSynchronizationBarrier

```
InstructionSynchronizationBarrier();
```

shared/functions/system/InterruptPending

```
// InterruptPending()
// ==================
// Return TRUE if there are any pending physical or virtual
// interrupts, and FALSE otherwise.

boolean InterruptPending()
    pending_physical_interrupt = (IRQPending() || FIQPending() ||
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
                                IsPhysicalSErrorPending());
    pending_virtual_interrupt  = !IsInHost() && ((HCR_EL2.<VSE,VI,VF> AND
                                HCR_EL2.<AMO,IMO,FMO>) != '000');
    return pending_physical_interrupt || pending_virtual_interrupt;
```

shared/functions/system/IsEventRegisterSet

```
// IsEventRegisterSet()
// ====================
// Return TRUE if the Event Register of this PE is set, and FALSE otherwise

boolean IsEventRegisterSet()
    return EventRegister == '1';
```

shared/functions/system/IsHighestEL

```
// IsHighestEL()
// =============
// Returns TRUE if given exception level is the highest exception level implemented

boolean IsHighestEL(bits(2) el)
    return HighestEL() == el;
```

shared/functions/system/IsInHost

```
// IsInHost()
// ==========

boolean IsInHost()
    return ELIsInHost(PSTATE.EL);
```

shared/functions/system/IsPhysicalSErrorPending

```
// Return TRUE if a physical SError interrupt is pending
boolean IsPhysicalSErrorPending();
```

shared/functions/system/IsSecure

```
// IsSecure()
// ==========
// Returns TRUE if current Exception level is in Secure state.

boolean IsSecure()
    if HaveEL(EL3) && !UsingAArch32() && PSTATE.EL == EL3 then
        return TRUE;
    elsif HaveEL(EL3) && UsingAArch32() && PSTATE.M == M32_Monitor then
        return TRUE;
    return IsSecureBelowEL3();
```

shared/functions/system/IsSecureBelowEL3

```
// IsSecureBelowEL3()
// ==================
// Return TRUE if an Exception level below EL3 is in Secure state
// or would be following an exception return to that level.
//
// Differs from IsSecure in that it ignores the current EL or Mode
// in considering security state.
// That is, if at AArch64 EL3 or in AArch32 Monitor mode, whether an
// exception return would pass to Secure or Non-secure state.

boolean IsSecureBelowEL3()
    if HaveEL(EL3) then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        return SCR_GEN[].NS == '0';
    elsif HaveEL(EL2) && (!HaveSecureEL2Ext() || HighestELUsingAArch32()) then
        // If Secure EL2 is not an architecture option then we must be Non-secure.
        return FALSE;
    else
        // TRUE if processor is Secure or FALSE if Non-secure.
        return boolean IMPLEMENTATION_DEFINED "Secure-only implementation";
```

shared/functions/system/IsSecureEL2Enabled

```
// IsSecureEL2Enabled()
// ====================
// Returns TRUE if Secure EL2 is enabled, FALSE otherwise.

boolean IsSecureEL2Enabled()
    if HaveEL(EL2) && HaveSecureEL2Ext() then
        if HaveEL(EL3) then
            if !ELUsingAArch32(EL3) && SCR_EL3.EEL2 == '1' then
                return TRUE;
            else
                return FALSE;
        else
            return IsSecure();
    else
        return FALSE;
```

shared/functions/system/IsVirtualSErrorPending

```
// Return TRUE if a virtual SError interrupt is pending
boolean IsVirtualSErrorPending();
```

shared/functions/system/Mode_Bits

```
constant bits(5) M32_User    = '10000';
constant bits(5) M32_FIQ     = '10001';
constant bits(5) M32_IRQ     = '10010';
constant bits(5) M32_Svc     = '10011';
constant bits(5) M32_Monitor = '10110';
constant bits(5) M32_Abort   = '10111';
constant bits(5) M32_Hyp     = '11010';
constant bits(5) M32_Undef   = '11011';
constant bits(5) M32_System  = '11111';
```

shared/functions/system/PLOfEL

```
// PLOfEL()
// ========

PrivilegeLevel PLOfEL(bits(2) el)
    case el of
        when EL3  return if HighestELUsingAArch32() then PL1 else PL3;
        when EL2  return PL2;
        when EL1  return PL1;
        when EL0  return PL0;
```

shared/functions/system/PSTATE

```
ProcState PSTATE;
```

shared/functions/system/PrivilegeLevel

```
enumeration PrivilegeLevel {PL3, PL2, PL1, PL0};
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/functions/system/ProcState

```
type ProcState is (
    bits (1) N,        // Negative condition flag
    bits (1) Z,        // Zero condition flag
    bits (1) C,        // Carry condition flag
    bits (1) V,        // oVerflow condition flag
    bits (1) D,        // Debug mask bit                    [AArch64 only]
    bits (1) A,        // SError interrupt mask bit
    bits (1) I,        // IRQ mask bit
    bits (1) F,        // FIQ mask bit
    bits (1) PAN,      // Privileged Access Never Bit       [v8.1]
    bits (1) UAO,      // User Access Override              [v8.2]
    bits (1) DIT,      // Data Independent Timing           [v8.4]
    bits (1) TCO,      // Tag Check Override                [v8.5, AArch64 only]
    bits (2) BTYPE,    // Branch Type                       [v8.5]
    bits (1) SS,       // Software step bit
    bits (1) IL,       // Illegal Execution state bit
    bits (2) EL,       // Exception Level
    bits (1) nRW,      // not Register Width: 0=64, 1=32
    bits (1) SP,       // Stack pointer select: 0=SP0, 1=SPx [AArch64 only]
    bits (1) Q,        // Cumulative saturation flag        [AArch32 only]
    bits (4) GE,       // Greater than or Equal flags       [AArch32 only]
    bits (1) SSBS,     // Speculative Store Bypass Safe
    bits (8) IT,       // If-then bits, RES0 in CPSR        [AArch32 only]
    bits (1) J,        // J bit, RES0                       [AArch32 only, RES0 in SPSR and CPSR]
    bits (1) T,        // T32 bit, RES0 in CPSR             [AArch32 only]
    bits (1) E,        // Endianness bit                    [AArch32 only]
    bits (5) M         // Mode field                        [AArch32 only]
)
```

shared/functions/system/RestoredITBits

```
// RestoredITBits()
// ================
// Get the value of PSTATE.IT to be restored on this exception return.

bits(8) RestoredITBits(bits(32) spsr)
    it = spsr<15:10,26:25>;

    // When PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the IT bits are each set
    // to zero or copied from the SPSR.
    if PSTATE.IL == '1' then
        if ConstrainUnpredictableBool() then return '00000000';
        else return it;

    // The IT bits are forced to zero when they are set to a reserved value.
    if !IsZero(it<7:4>) && IsZero(it<3:0>) then
        return '00000000';

    // The IT bits are forced to zero when returning to A32 state, or when returning to an EL
    // with the ITD bit set to 1, and the IT bits are describing a multi-instruction block.
    itd = if PSTATE.EL == EL2 then HSCTLR.ITD else SCTLR.ITD;
    if (spsr<5> == '0' && !IsZero(it)) || (itd == '1' && !IsZero(it<2:0>)) then
        return '00000000';
    else
        return it;
```

shared/functions/system/SCRType

```
type SCRType;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/functions/system/SCR_GEN

```
// SCR_GEN[]
// =========

SCRType SCR_GEN[]
    // AArch32 secure & AArch64 EL3 registers are not architecturally mapped
    assert HaveEL(EL3);
    bits(64) r;
    if HighestELUsingAArch32() then
        r = ZeroExtend(SCR);
    else
        r = ZeroExtend(SCR_EL3);
    return r;
```

shared/functions/system/SendEvent

```
// Signal an event to all PEs in a multiprocessor system to set their Event Registers.
// When a PE executes the SEV instruction, it causes this function to be executed
SendEvent();
```

shared/functions/system/SendEventLocal

```
// SendEventLocal()
// ================
// Set the local Event Register of this PE.
// When a PE executes the SEVL instruction, it causes this function to be executed

SendEventLocal()
    EventRegister = '1';
    return;
```

shared/functions/system/SetPSTATEFromPSR

```
// SetPSTATEFromPSR()
// ==================
// Set PSTATE based on a PSR value

SetPSTATEFromPSR(bits(32) spsr)
    PSTATE.SS = DebugExceptionReturnSS(spsr);
    if IllegalExceptionReturn(spsr) then
        PSTATE.IL = '1';
        if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
    else
        // State that is reinstated only on a legal exception return
        PSTATE.IL = spsr<20>;
        if spsr<4> == '1' then                    // AArch32 state
            AArch32.WriteMode(spsr<4:0>);         // Sets PSTATE.EL correctly
            if HaveSSBSExt() then PSTATE.SSBS = spsr<23>;
        else                                      // AArch64 state
            PSTATE.nRW = '0';
            PSTATE.EL  = spsr<3:2>;
            PSTATE.SP  = spsr<0>;
            if HaveSSBSExt() then PSTATE.SSBS = spsr<12>;

    // If PSTATE.IL is set and returning to AArch32 state, it is CONSTRAINED UNPREDICTABLE whether
    // the T bit is set to zero or copied from SPSR.
    if PSTATE.IL == '1' && PSTATE.nRW == '1' then
        if ConstrainUnpredictableBool() then spsr<5> = '0';

    // State that is reinstated regardless of illegal exception return
    PSTATE.<N,Z,C,V> = spsr<31:28>;
    if HavePANExt() then PSTATE.PAN = spsr<22>;
    if PSTATE.nRW == '1' then                     // AArch32 state
        PSTATE.Q         = spsr<27>;
        PSTATE.IT        = RestoredITBits(spsr);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        ShouldAdvanceIT  = FALSE;
        if HaveDITExt() then PSTATE.DIT = (if Restarting() then spsr<24> else spsr<21>);
        PSTATE.GE        = spsr<19:16>;
        PSTATE.E         = spsr<9>;
        PSTATE.<A,I,F>   = spsr<8:6>;             // No PSTATE.D in AArch32 state
        PSTATE.T         = spsr<5>;               // PSTATE.J is RES0
    else                                          // AArch64 state
        if HaveMTEExt() then PSTATE.TCO = spsr<25>;
        if HaveDITExt() then PSTATE.DIT = spsr<24>;
        if HaveUAOExt() then PSTATE.UAO = spsr<23>;
        if HaveBTIExt() then PSTATE.BTYPE = spsr<11:10>;
        PSTATE.<D,A,I,F> = spsr<9:6>;             // No PSTATE.<Q,IT,GE,E,T> in AArch64 state
    return;
```

shared/functions/system/ShouldAdvanceIT

```
boolean ShouldAdvanceIT;
```

shared/functions/system/SpeculationBarrier

```
SpeculationBarrier();
```

shared/functions/system/SynchronizeContext

```
SynchronizeContext();
```

shared/functions/system/SynchronizeErrors

```
// Implements the error synchronization event.
SynchronizeErrors();
```

shared/functions/system/TakeUnmaskedPhysicalSErrorInterrupts

```
// Take any pending unmasked physical SError interrupt
TakeUnmaskedPhysicalSErrorInterrupts(boolean iesb_req);
```

shared/functions/system/TakeUnmaskedSErrorInterrupts

```
// Take any pending unmasked physical SError interrupt or unmasked virtual SError
// interrupt.
TakeUnmaskedSErrorInterrupts();
```

shared/functions/system/ThisInstr

```
bits(32) ThisInstr();
```

shared/functions/system/ThisInstrLength

```
integer ThisInstrLength();
```

shared/functions/system/Unreachable

```
Unreachable()
    assert FALSE;
```

shared/functions/system/UsingAArch32

```
// UsingAArch32()
// ==============
// Return TRUE if the current Exception level is using AArch32, FALSE if using AArch64.
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
boolean UsingAArch32()
    boolean aarch32 = (PSTATE.nRW == '1');
    if !HaveAnyAArch32() then assert !aarch32;
    if HighestELUsingAArch32() then assert aarch32;
    return aarch32;
```

shared/functions/system/WaitForEvent

```
// WaitForEvent()
// ==============
// PE suspends its operation and enters a low-power state
// if the Event Register is clear when the WFE is executed

WaitForEvent()
    if EventRegister == '0' then
        EnterLowPowerState();
    return;
```

shared/functions/system/WaitForInterrupt

```
// WaitForInterrupt()
// ==================
// PE suspends its operation to enter a low-power state
// until a WFI wake-up event occurs or the PE is reset

WaitForInterrupt()
    EnterLowPowerState();
    return;
```

shared/functions/unpredictable/ConstrainUnpredictable

```
// Return the appropriate Constraint result to control the caller's behavior. The return value
// is IMPLEMENTATION DEFINED within a permitted list for each UNPREDICTABLE case.
// (The permitted list is determined by an assert or case statement at the call site.)
Constraint ConstrainUnpredictable();
```

shared/functions/unpredictable/ConstrainUnpredictableBits

```
// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN.
// If the result is Constraint_UNKNOWN then the function also returns UNKNOWN value, but that
// value is always an allocated value; that is, one for which the behavior is not itself
// CONSTRAINED.
(Constraint,bits(width)) ConstrainUnpredictableBits();
```

shared/functions/unpredictable/ConstrainUnpredictableBool

```
// ConstrainUnpredictableBool()
// ============================

// This is a simple wrapper function for cases where the constrained result is either TRUE or FALSE.

boolean ConstrainUnpredictableBool()

    c = ConstrainUnpredictable();
    assert c IN {Constraint_TRUE, Constraint_FALSE};
    return (c == Constraint_TRUE);
```

shared/functions/unpredictable/ConstrainUnpredictableInteger

```
// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN. If
// the result is Constraint_UNKNOWN then the function also returns an UNKNOWN value in the range
// low to high, inclusive.
(Constraint,integer) ConstrainUnpredictableInteger(integer low, integer high);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/functions/unpredictable/Constraint

```
enumeration Constraint    {// General
                           Constraint_NONE,              // Instruction executes with
                                                         //   no change or side-effect to its described
behavior
                           Constraint_UNKNOWN,           // Destination register has UNKNOWN value
                           Constraint_UNDEF,             // Instruction is UNDEFINED
                           Constraint_UNDEFEL0,          // Instruction is UNDEFINED at EL0 only
                           Constraint_NOP,               // Instruction executes as NOP
                           Constraint_TRUE,
                           Constraint_FALSE,
                           Constraint_DISABLED,
                           Constraint_UNCOND,            // Instruction executes unconditionally
                           Constraint_COND,              // Instruction executes conditionally
                           Constraint_ADDITIONAL_DECODE, // Instruction executes with additional decode
                           // Load-store
                           Constraint_WBSUPPRESS, Constraint_FAULT,
                           // IPA too large
                           Constraint_FORCE, Constraint_FORCENOSLCHECK};
```

shared/functions/vector/AdvSIMDExpandImm

```
// AdvSIMDExpandImm()
// ==================

bits(64) AdvSIMDExpandImm(bit op, bits(4) cmode, bits(8) imm8)
    case cmode<3:1> of
        when '000'
            imm64 = Replicate(Zeros(24):imm8, 2);
        when '001'
            imm64 = Replicate(Zeros(16):imm8:Zeros(8), 2);
        when '010'
            imm64 = Replicate(Zeros(8):imm8:Zeros(16), 2);
        when '011'
            imm64 = Replicate(imm8:Zeros(24), 2);
        when '100'
            imm64 = Replicate(Zeros(8):imm8, 4);
        when '101'
            imm64 = Replicate(imm8:Zeros(8), 4);
        when '110'
            if cmode<0> == '0' then
                imm64 = Replicate(Zeros(16):imm8:Ones(8), 2);
            else
                imm64 = Replicate(Zeros(8):imm8:Ones(16), 2);
        when '111'
            if cmode<0> == '0' && op == '0' then
                imm64 = Replicate(imm8, 8);
            if cmode<0> == '0' && op == '1' then
                imm8a = Replicate(imm8<7>, 8); imm8b = Replicate(imm8<6>, 8);
                imm8c = Replicate(imm8<5>, 8); imm8d = Replicate(imm8<4>, 8);
                imm8e = Replicate(imm8<3>, 8); imm8f = Replicate(imm8<2>, 8);
                imm8g = Replicate(imm8<1>, 8); imm8h = Replicate(imm8<0>, 8);
                imm64 = imm8a:imm8b:imm8c:imm8d:imm8e:imm8f:imm8g:imm8h;
            if cmode<0> == '1' && op == '0' then
                imm32 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,5):imm8<5:0>:Zeros(19);
                imm64 = Replicate(imm32, 2);
            if cmode<0> == '1' && op == '1' then
                if UsingAArch32() then ReservedEncoding();
                imm64 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,8):imm8<5:0>:Zeros(48);

    return imm64;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/functions/vector/MatMulAdd

```
// MatMulAdd()
// ===========
//
// Signed or unsigned 8-bit integer matrix multiply and add to 32-bit integer matrix
// result[2, 2] = addend[2, 2] + (op1[2, 8] * op2[8, 2])

bits(N) MatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, boolean op1_unsigned, boolean op2_unsigned)
    assert N == 128;

    bits(N)  result;
    bits(32) sum;
    integer  prod;

    for i = 0 to 1
        for j = 0 to 1
            sum = Elem[addend, 2*i + j, 32];
            for k = 0 to 7
                prod = Int(Elem[op1, 8*i + k, 8], op1_unsigned) * Int(Elem[op2, 8*j + k, 8],
op2_unsigned);
                sum  = sum + prod;
            Elem[result, 2*i + j, 32] = sum;

    return result;
```

shared/functions/vector/PolynomialMult

```
// PolynomialMult()
// ================

bits(M+N) PolynomialMult(bits(M) op1, bits(N) op2)
    result = Zeros(M+N);
    extended_op2 = ZeroExtend(op2, M+N);
    for i=0 to M-1
        if op1<i> == '1' then
            result = result EOR LSL(extended_op2, i);
    return result;
```

shared/functions/vector/SatQ

```
// SatQ()
// ======

(bits(N), boolean) SatQ(integer i, integer N, boolean unsigned)
    (result, sat) = if unsigned then UnsignedSatQ(i, N) else SignedSatQ(i, N);
    return (result, sat);
```

shared/functions/vector/SignedSatQ

```
// SignedSatQ()
// ============

(bits(N), boolean) SignedSatQ(integer i, integer N)
    if i > 2^(N-1) - 1 then
        result = 2^(N-1) - 1;  saturated = TRUE;
    elsif i < -(2^(N-1)) then
        result = -(2^(N-1));  saturated = TRUE;
    else
        result = i;  saturated = FALSE;
    return (result<N-1:0>, saturated);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/functions/vector/UnsignedRSqrtEstimate

```
// UnsignedRSqrtEstimate()
// =======================

bits(N) UnsignedRSqrtEstimate(bits(N) operand)
    assert N IN {16,32};
    if operand<N-1:N-2> == '00' then  // Operands <= 0x3FFFFFFF produce 0xFFFFFFFF
        result = Ones(N);
    else
        // input is in the range 0x40000000 .. 0xffffffff representing [0.25 .. 1.0)

        // estimate is in the range 256 .. 511 representing [1.0 .. 2.0)
        case N of
            when 16 estimate = RecipSqrtEstimate(UInt(operand<15:7>));
            when 32 estimate = RecipSqrtEstimate(UInt(operand<31:23>));

        // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
        result = estimate<8:0> : Zeros(N-9);

    return result;
```

shared/functions/vector/UnsignedRecipEstimate

```
// UnsignedRecipEstimate()
// =======================

bits(N) UnsignedRecipEstimate(bits(N) operand)
    assert N IN {16,32};
    if operand<N-1> == '0' then  // Operands <= 0x7FFFFFFF produce 0xFFFFFFFF
        result = Ones(N);
    else
        // input is in the range 0x80000000 .. 0xffffffff representing [0.5 .. 1.0)

        // estimate is in the range 256 to 511 representing [1.0 .. 2.0)
        case N of
            when 16 estimate = RecipEstimate(UInt(operand<15:7>));
            when 32 estimate = RecipEstimate(UInt(operand<31:23>));

        // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
        result = estimate<8:0> : Zeros(N-9);

    return result;
```

shared/functions/vector/UnsignedSatQ

```
// UnsignedSatQ()
// ==============

(bits(N), boolean) UnsignedSatQ(integer i, integer N)
    if i > 2^N - 1 then
        result = 2^N - 1;  saturated = TRUE;
    elsif i < 0 then
        result = 0;  saturated = TRUE;
    else
        result = i;  saturated = FALSE;
    return (result<N-1:0>, saturated);
```

J1.3.4 shared/trace

This section includes the following pseudocode functions:

- *shared/trace/selfhosted/SelfHostedTraceEnabled* on page J1-7819.
- *shared/trace/selfhosted/TraceAllowed* on page J1-7819.
- *shared/trace/selfhosted/TraceContextIDR2* on page J1-7819.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *shared/trace/selfhosted/TraceSynchronizationBarrier*.
- *shared/trace/selfhosted/TraceTimeStamp*.

shared/trace/selfhosted/SelfHostedTraceEnabled

```
// SelfHostedTraceEnabled()
// ========================
// Returns TRUE if Self-hosted Trace is enabled.

boolean SelfHostedTraceEnabled()
    if !HaveTraceExt() || !HaveSelfHostedTrace() then return FALSE;
    if HaveEL(EL3) then
        secure_trace_enable = (if ELUsingAArch32(EL3) then SDCR.STE else MDCR_EL3.STE);
        niden = (secure_trace_enable == '0' || ExternalSecureNoninvasiveDebugEnabled());
    else
        // If no EL3, IsSecure() returns the Effective value of (SCR_EL3.NS == '0')
        niden = (!IsSecure() || ExternalSecureNoninvasiveDebugEnabled());
    return (EDSCR.TFO == '0' || !niden);
```

shared/trace/selfhosted/TraceAllowed

```
// TraceAllowed()
// ==============
// Returns TRUE if Self-hosted Trace is allowed in the current Security state and Exception Level

boolean TraceAllowed()
    if !HaveTraceExt() then return FALSE;
    if SelfHostedTraceEnabled() then
        if IsSecure() && HaveEL(EL3) then
            secure_trace_enable = (if ELUsingAArch32(EL3) then SDCR.STE else MDCR_EL3.STE);
            if secure_trace_enable == '0' then return FALSE;
        TGE_bit = if EL2Enabled() then HCR_EL2.TGE else '0';
        case PSTATE.EL of
            when EL3  TRE_bit = if HighestELUsingAArch32() then TRFCR.E1TRE else '0';
            when EL2  TRE_bit = TRFCR_EL2.E2TRE;
            when EL1  TRE_bit = TRFCR_EL1.E1TRE;
            when EL0  TRE_bit = if TGE_bit == '1' then TRFCR_EL2.E0HTRE else TRFCR_EL1.E0TRE;
        return TRE_bit == '1';
    else
        return (!IsSecure() || ExternalSecureNoninvasiveDebugEnabled());
```

shared/trace/selfhosted/TraceContextIDR2

```
// TraceContextIDR2()
// ==================

boolean TraceContextIDR2()
    if !TraceAllowed()|| !HaveEL(EL2) then return FALSE;
    return (!SelfHostedTraceEnabled() || TRFCR_EL2.CX == '1');
```

shared/trace/selfhosted/TraceSynchronizationBarrier

```
// Memory barrier instruction that preserves the relative order of memory accesses to System
// registers due to trace operations and other memory accesses to the same registers
TraceSynchronizationBarrier();
```

shared/trace/selfhosted/TraceTimeStamp

```
// TraceTimeStamp()
// ================

TimeStamp TraceTimeStamp()
    if SelfHostedTraceEnabled() then
        if HaveEL(EL2) then
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            TS_el2 = TRFCR_EL2.TS;
            if TS_el2 == '10' then (-, TS_el2) = ConstrainUnpredictableBits();  // Reserved value
            case TS_el2 of
                when '00'  /* falls through to check TRFCR_EL1.TS */
                when '01'  return TimeStamp_Virtual;
                when '11'  return TimeStamp_Physical;
                otherwise  Unreachable();          // ConstrainUnpredictableBits removes this case
        TS_el1 = TRFCR_EL1.TS;
        if TS_el1 == 'x0' then (-, TS_el1) = ConstrainUnpredictableBits();      // Reserved values
        case TS_el1 of
            when '01'  return TimeStamp_Virtual;
            when '11'  return TimeStamp_Physical;
            otherwise  Unreachable();              // ConstrainUnpredictableBits removes this case
    else
        return TimeStamp_CoreSight;
```

J1.3.5 shared/translation

This section includes the following pseudocode functions:

- *shared/translation/attrs/CombineS1S2AttrHints*.
- *shared/translation/attrs/CombineS1S2Device* on page J1-7821.
- *shared/translation/attrs/LongConvertAttrsHints* on page J1-7821.
- *shared/translation/attrs/MemAttrDefaults* on page J1-7822.
- *shared/translation/attrs/S1CacheDisabled* on page J1-7822.
- *shared/translation/attrs/S2AttrDecode* on page J1-7822.
- *shared/translation/attrs/S2CacheDisabled* on page J1-7823.
- *shared/translation/attrs/S2ConvertAttrsHints* on page J1-7823.
- *shared/translation/attrs/ShortConvertAttrsHints* on page J1-7823.
- *shared/translation/attrs/WalkAttrDecode* on page J1-7824.
- *shared/translation/translation/HasS2Translation* on page J1-7824.
- *shared/translation/translation/Have16bitVMID* on page J1-7824.
- *shared/translation/translation/PAMax* on page J1-7824.
- *shared/translation/translation/S1TranslationRegime* on page J1-7825.
- *shared/translation/translation/VAMax* on page J1-7825.

shared/translation/attrs/CombineS1S2AttrHints

```
// CombineS1S2AttrHints()
// ======================
// Combines cacheability attributes and allocation hints from stage 1 and stage 2

MemAttrHints CombineS1S2AttrHints(MemAttrHints s1desc, MemAttrHints s2desc)

    MemAttrHints result;

    apply_force_writeback = HaveStage2MemAttrControl() && HCR_EL2.FWB == '1';
    if apply_force_writeback then
        if s2desc.attrs == '11' then
            result.attrs = s1desc.attrs;
        elsif s2desc.attrs == '10' then
            result.attrs = MemAttr_WB;    // force Write-back
        else
            result.attrs = MemAttr_NC;
    else
        if s2desc.attrs == '01' || s1desc.attrs == '01' then
            result.attrs = bits(2) UNKNOWN;   // Reserved
        elsif s2desc.attrs == MemAttr_NC || s1desc.attrs == MemAttr_NC then
            result.attrs = MemAttr_NC;        // Non-cacheable
        elsif s2desc.attrs == MemAttr_WT || s1desc.attrs == MemAttr_WT then
            result.attrs = MemAttr_WT;        // Write-through
        else
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            result.attrs = MemAttr_WB;        // Write-back

    if result.attrs == MemAttr_NC then
        result.hints = MemHint_No;
    elsif apply_force_writeback then
        if s1desc.attrs != MemAttr_NC then
            result.hints = s1desc.hints;
        else
            result.hints = MemHint_RWA;
    else
        result.hints = s1desc.hints;
    result.transient = s1desc.transient;

    return result;
```

shared/translation/attrs/CombineS1S2Device

```
// CombineS1S2Device()
// ===================
// Combines device types from stage 1 and stage 2

DeviceType CombineS1S2Device(DeviceType s1device, DeviceType s2device)

    if s2device == DeviceType_nGnRnE || s1device == DeviceType_nGnRnE then
        result = DeviceType_nGnRnE;
    elsif s2device == DeviceType_nGnRE || s1device == DeviceType_nGnRE then
        result = DeviceType_nGnRE;
    elsif s2device == DeviceType_nGRE || s1device == DeviceType_nGRE then
        result = DeviceType_nGRE;
    else
        result = DeviceType_GRE;

    return result;
```

shared/translation/attrs/LongConvertAttrsHints

```
// LongConvertAttrsHints()
// =======================
// Convert the long attribute fields for Normal memory as used in the MAIR fields
// to orthogonal attributes and hints

MemAttrHints LongConvertAttrsHints(bits(4) attrfield, AccType acctype)
    assert !IsZero(attrfield);
    MemAttrHints result;
    if S1CacheDisabled(acctype) then             // Force Non-cacheable
        result.attrs = MemAttr_NC;
        result.hints = MemHint_No;
    else
        if attrfield<3:2> == '00' then          // Write-through transient
            result.attrs = MemAttr_WT;
            result.hints = attrfield<1:0>;
            result.transient = TRUE;
        elsif attrfield<3:0> == '0100' then     // Non-cacheable (no allocate)
            result.attrs = MemAttr_NC;
            result.hints = MemHint_No;
            result.transient = FALSE;
        elsif attrfield<3:2> == '01' then       // Write-back transient
            result.attrs = MemAttr_WB;
            result.hints = attrfield<1:0>;
            result.transient = TRUE;
        else                                    // Write-through/Write-back non-transient
            result.attrs = attrfield<3:2>;
            result.hints = attrfield<1:0>;
            result.transient = FALSE;

    return result;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/translation/attrs/MemAttrDefaults

```
// MemAttrDefaults()
// =================
// Supply default values for memory attributes, including overriding the shareability attributes
// for Device and Non-cacheable memory types.

MemoryAttributes MemAttrDefaults(MemoryAttributes memattrs)

    if memattrs.memtype == MemType_Device then
        memattrs.inner = MemAttrHints UNKNOWN;
        memattrs.outer = MemAttrHints UNKNOWN;
        memattrs.shareable = TRUE;
        memattrs.outershareable = TRUE;
    else
        memattrs.device = DeviceType UNKNOWN;
        if memattrs.inner.attrs == MemAttr_NC && memattrs.outer.attrs == MemAttr_NC then
            memattrs.shareable = TRUE;
            memattrs.outershareable = TRUE;

    return memattrs;
```

shared/translation/attrs/S1CacheDisabled

```
// S1CacheDisabled()
// =================

boolean S1CacheDisabled(AccType acctype)
    if ELUsingAArch32(S1TranslationRegime()) then
        if PSTATE.EL == EL2 then
            enable = if acctype == AccType_IFETCH then HSCTLR.I else HSCTLR.C;
        else
            enable = if acctype == AccType_IFETCH then SCTLR.I else SCTLR.C;
    else
        enable = if acctype == AccType_IFETCH then SCTLR[].I else SCTLR[].C;
    return enable == '0';
```

shared/translation/attrs/S2AttrDecode

```
// S2AttrDecode()
// ==============
// Converts the Stage 2 attribute fields into orthogonal attributes and hints

MemoryAttributes S2AttrDecode(bits(2) SH, bits(4) attr, AccType acctype)

    MemoryAttributes memattrs;

    apply_force_writeback = HaveStage2MemAttrControl() && HCR_EL2.FWB == '1';

    // Device memory
    if (apply_force_writeback && attr<2> == '0') || attr<3:2> == '00' then
        memattrs.memtype = MemType_Device;
        case attr<1:0> of
            when '00'  memattrs.device = DeviceType_nGnRnE;
            when '01'  memattrs.device = DeviceType_nGnRE;
            when '10'  memattrs.device = DeviceType_nGRE;
            when '11'  memattrs.device = DeviceType_GRE;

    // Normal memory
    elsif apply_force_writeback then
        if attr<2> == '1' then
            memattrs.memtype = MemType_Normal;
            memattrs.inner.attrs = attr<1:0>;
            memattrs.outer.attrs = attr<1:0>;
    elsif attr<1:0> != '00' then
        memattrs.memtype = MemType_Normal;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        memattrs.outer = S2ConvertAttrsHints(attr<3:2>, acctype);
        memattrs.inner = S2ConvertAttrsHints(attr<1:0>, acctype);
        memattrs.shareable = SH<1> == '1';
        memattrs.outershareable = SH == '10';
    else
        memattrs = MemoryAttributes UNKNOWN;    // Reserved

    return MemAttrDefaults(memattrs);
```

shared/translation/attrs/S2CacheDisabled

```
// S2CacheDisabled()
// =================

boolean S2CacheDisabled(AccType acctype)
    if ELUsingAArch32(EL2) then
        disable = if acctype == AccType_IFETCH then HCR2.ID else HCR2.CD;
    else
        disable = if acctype == AccType_IFETCH then HCR_EL2.ID else HCR_EL2.CD;
    return disable == '1';
```

shared/translation/attrs/S2ConvertAttrsHints

```
// S2ConvertAttrsHints()
// =====================
// Converts the attribute fields for Normal memory as used in stage 2
// descriptors to orthogonal attributes and hints

MemAttrHints S2ConvertAttrsHints(bits(2) attr, AccType acctype)
    assert !IsZero(attr);

    MemAttrHints result;

    if HCR_EL2.FWB=='0' && S2CacheDisabled(acctype) then                // Force Non-cacheable
        result.attrs = MemAttr_NC;
        result.hints = MemHint_No;
    else
        case attr of
            when '01'                               // Non-cacheable (no allocate)
                result.attrs = MemAttr_NC;
                result.hints = MemHint_No;
            when '10'                               // Write-through
                result.attrs = MemAttr_WT;
                result.hints = MemHint_RWA;
            when '11'                               // Write-back
                result.attrs = MemAttr_WB;
                result.hints = MemHint_RWA;

    result.transient = FALSE;

    return result;
```

shared/translation/attrs/ShortConvertAttrsHints

```
// ShortConvertAttrsHints()
// ========================
// Converts the short attribute fields for Normal memory as used in the TTBR and
// TEX fields to orthogonal attributes and hints

MemAttrHints ShortConvertAttrsHints(bits(2) RGN, AccType acctype, boolean secondstage)

    MemAttrHints result;

    if (!secondstage && S1CacheDisabled(acctype)) || (secondstage && S2CacheDisabled(acctype)) then
       // Force Non-cacheable
        result.attrs = MemAttr_NC;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        result.hints = MemHint_No;
    else
        case RGN of
            when '00'                   // Non-cacheable (no allocate)
                result.attrs = MemAttr_NC;
                result.hints = MemHint_No;
            when '01'                   // Write-back, Read and Write allocate
                result.attrs = MemAttr_WB;
                result.hints = MemHint_RWA;
            when '10'                   // Write-through, Read allocate
                result.attrs = MemAttr_WT;
                result.hints = MemHint_RA;
            when '11'                   // Write-back, Read allocate
                result.attrs = MemAttr_WB;
                result.hints = MemHint_RA;

    result.transient = FALSE;

    return result;
```

shared/translation/attrs/WalkAttrDecode

```
// WalkAttrDecode()
// ================

MemoryAttributes WalkAttrDecode(bits(2) SH, bits(2) ORGN, bits(2) IRGN, boolean secondstage)

    MemoryAttributes mematrs;

    AccType acctype = AccType_NORMAL;

    mematrs.memtype = MemType_Normal;
    mematrs.inner = ShortConvertAttrsHints(IRGN, acctype, secondstage);
    mematrs.outer = ShortConvertAttrsHints(ORGN, acctype, secondstage);
    mematrs.shareable = SH<1> == '1';
    mematrs.outershareable = SH == '10';
    mematrs.tagged = FALSE;

    return MemAttrDefaults(mematrs);
```

shared/translation/translation/HasS2Translation

```
// HasS2Translation()
// ==================
// Returns TRUE if stage 2 translation is present for the current translation regime

boolean HasS2Translation()
    return (EL2Enabled() && !IsInHost() && PSTATE.EL IN {EL0,EL1});
```

shared/translation/translation/Have16bitVMID

```
// Returns TRUE if EL2 and support for a 16-bit VMID are implemented.
boolean Have16bitVMID();
```

shared/translation/translation/PAMax

```
// PAMax()
// =======
// Returns the IMPLEMENTATION DEFINED upper limit on the physical address
// size for this processor, as log2().

integer PAMax()
    return integer IMPLEMENTATION_DEFINED "Maximum Physical Address Size";
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

shared/translation/translation/S1TranslationRegime

```
// S1TranslationRegime()
// =====================
// Stage 1 translation regime for the given Exception level

bits(2) S1TranslationRegime(bits(2) el)
    if el != EL0 then
        return el;
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.NS == '0' then
        return EL3;
    elsif HaveVirtHostExt() && ELIsInHost(el) then
        return EL2;
    else
        return EL1;

// S1TranslationRegime()
// =====================
// Returns the Exception level controlling the current Stage 1 translation regime. For the most
// part this is unused in code because the system register accessors (SCTLR[], etc.) implicitly
// return the correct value.

bits(2) S1TranslationRegime()
    return S1TranslationRegime(PSTATE.EL);
```

shared/translation/translation/VAMax

```
// VAMax()
// =======
// Returns the IMPLEMENTATION DEFINED upper limit on the virtual address
// size for this processor, as log2().

integer VAMax()
    return integer IMPLEMENTATION_DEFINED "Maximum Virtual Address Size";
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Part K

Appendixes

Appendix K1
Architectural Constraints on UNPREDICTABLE Behaviors

This chapter describes the architectural constraints on UNPREDICTABLE behaviors in the Armv8 architecture. It contains the following sections:

- *AArch32 CONSTRAINED UNPREDICTABLE behaviors* on page K1-7830.
- *AArch64 CONSTRAINED UNPREDICTABLE behaviors* on page K1-7855.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K1.1 AArch32 CONSTRAINED UNPREDICTABLE behaviors

Armv8 defines architecturally-required constraints on many behaviors that are UNPREDICTABLE in Armv7. The following sections define those constraints:

- *Overview of the constraints on Armv7 UNPREDICTABLE behaviors* on page K1-7831.
- *Using R13* on page K1-7831.
- *Using R15* on page K1-7831.
- *Branching into an IT block* on page K1-7832.
- *Branching to an unaligned PC* on page K1-7832.
- *Loads and Stores to unaligned locations* on page K1-7832.
- *CONSTRAINED UNPREDICTABLE behavior associated with IT instructions and PSTATE.IT* on page K1-7832.
- *Unallocated System register access instructions* on page K1-7834.
- *SBZ or SBO fields T32 and A32 in instructions* on page K1-7834.
- *UNPREDICTABLE cases in immediate constants in T32 data-processing instructions* on page K1-7834.
- *UNPREDICTABLE cases in immediate constants in Advanced SIMD instructions* on page K1-7835.
- *CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values* on page K1-7835.
- *CONSTRAINED UNPREDICTABLE behavior due to inadequate context synchronization* on page K1-7836
- *Translation Table Base Address alignment* on page K1-7836.
- *Handling of System register control fields for Advanced SIMD and floating-point operation* on page K1-7836.
- *Mapping of non-idempotent memory locations using the Normal memory type* on page K1-7837.
- *The Performance Monitors Extension* on page K1-7837.
- *The Activity Monitors Extension* on page K1-7839.
- *Syndrome register handling for CONSTRAINED UNPREDICTABLE instructions treated as UNDEFINED* on page K1-7840.
- *Out of range VA* on page K1-7840.
- *Instruction fetches from Device memory* on page K1-7840.
- *Multi-access instructions that load the PC from Device memory* on page K1-7840.
- *Programming CSSELR.Level for a cache level that is not implemented* on page K1-7840.
- *Crossing a page boundary with different memory types or Shareability attributes* on page K1-7841.
- *Crossing a 4KB boundary with a Device access* on page K1-7841.
- *UNPREDICTABLE behaviors with Load-Exclusive/Store-Exclusive pairs* on page K1-7842.
- *CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings* on page K1-7842.
- *Out of range values of the Set/Way/Index fields in cache maintenance instructions* on page K1-7843.
- *CONSTRAINED UNPREDICTABLE behavior for A32 and T32 System instructions in the base instruction set* on page K1-7843.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *CONSTRAINED UNPREDICTABLE behavior, A32 and T32 Advanced SIMD and floating-point instructions* on page K1-7846.
- *CONSTRAINED UNPREDICTABLE behaviors associated with the VTCR* on page K1-7850.
- *CONSTRAINED UNPREDICTABLE behavior of EL2 features* on page K1-7850.
- *Reserved values in System and memory-mapped registers and translation table entries* on page K1-7853.
- *CONSTRAINED UNPREDICTABLE behavior in Debug state* on page K1-7854.

K1.1.1 Overview of the constraints on Armv7 UNPREDICTABLE behaviors

The term UNPREDICTABLE describes a number of cases where the architecture has a feature that software must not use. For execution in AArch32 state, where previous versions of the architecture define behavior as UNPREDICTABLE, the Armv8-A architecture specifies a narrow range of permitted behaviors. This range is the range of CONSTRAINED UNPREDICTABLE behavior. All implementations that are compliant with the architecture must follow the CONSTRAINED UNPREDICTABLE behavior.

Note

Software designed to be compatible with the Armv8-A architecture must not rely on these CONSTRAINED UNPREDICTABLE cases.

K1.1.2 Using R13

In prior versions of the architecture, the use of R13 as a named register specifier was described as UNPREDICTABLE in the pseudocode. In the Armv8-A architecture, the use of R13 as a named register specifier is not UNPREDICTABLE, unless this is specifically stated, and R13 can be used in the regular form. Bits[1:0] of R13 are not treated as RES0, but can hold any values programmed into them.

K1.1.3 Using R15

All uses of R15 as a named register specifier for a source register that are described as CONSTRAINED UNPREDICTABLE in the pseudocode or in other places in this Manual must do one of the following:

- Cause the instruction to be treated as UNDEFINED.
- Cause the instruction to execute as a `NOP`.
- Read or return an UNKNOWN value for the source register specified as R15.

All uses of R15 as a named register specifier for a destination register that are described as CONSTRAINED UNPREDICTABLE in the pseudocode or in other places in this reference manual must do one of the following:

- Cause the instruction to be treated as UNDEFINED.
- Cause the instruction to execute as a `NOP`.
- Ignore the write.
- Branch to an UNKNOWN location in either A32 or T32 state.

The choice between these behaviors might in some implementations vary from instruction to instruction, or between different instances of the same instruction.

Instructions that are CONSTRAINED UNPREDICTABLE when the base register is R15 and the instruction specifies a writeback of the base register, are treated as having R15 as both a source register and a destination register.

For instructions that have two destination registers, for example `LDRD`, `MRRC`, and many of the multiply instructions, if Rt, Rt2, RdLo, or RdHi is R15, then the other destination register of the pair is UNKNOWN, if the CONSTRAINED UNPREDICTABLE behavior for the write to R15 is either to ignore the write or to branch to an UNKNOWN location.

For instructions that affect any or all of PSTATE.{N, Z, C, V}, PSTATE.Q, and PSTATE.GE when the register specifier is not R15, any flags affected by an instruction that is CONSTRAINED UNPREDICTABLE when the register specifier is R15 become UNKNOWN,

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In addition, for MRC instructions that use R15 as the destination register descriptor, and therefore target APSR_nzcv where these are described as being CONSTRAINED UNPREDICTABLE, PSTATE.{N, Z, C, V} becomes UNKNOWN.

K1.1.4 Branching into an IT block

Branching into an IT block leads to CONSTRAINED UNPREDICTABLE behavior. Execution starts from the address determined by the branch, but each instruction in the IT block is:

- Executed as if it were not in an IT block. This means that it is executed unconditionally.
- Executed as if it had passed its Condition code check within an IT block.
- Executed as a NOP. That is, it behaves as if it had failed the Condition code check.

K1.1.5 Branching to an unaligned PC

In A32 state, when branching to an address that is not word aligned and is defined to be CONSTRAINED UNPREDICTABLE, one of the following behaviors must occur:

- The unaligned location is forced to be aligned.
- The unaligned address generates an exception on the first instruction using the unaligned PC value.

  If that instruction is executed at EL0 and either of the following applies, the exception is taken to EL2:
  - EL2 is using AArch32 and the value of HCR.TGE is 1.
  - EL2 is using AArch64 and the value of HCR_EL2.TGE is 1.

  If the instruction is executed at EL0 when the applicable TGE bit is 0 the exception is taken to EL1.

  If the instruction is executed at an Exception level that is higher than EL0 the exception is taken to the Exception level at which the instruction was executed.

  In all cases, the exception is generated only if the first instruction using the unaligned PC value is architecturally executed.

If the exception that results from a branch to an unaligned PC value:

- Is taken to an Exception level that is using AArch64, it is reported as a PC alignment fault exception, see *ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault* on page D13-2984.
- Is taken to an Exception level that is using AArch32, it is reported as a Prefetch Abort exception, see *Prefetch Abort exception reporting a PC alignment fault exception* on page G1-5746.

Note

Because bit[0] is used for interworking, it is impossible to specify a branch to A32 state when the bottom bit of the target address is 1. Therefore the bottom bit of IFAR, HIFAR, or FAR_ELx is 0 for all these cases.

K1.1.6 Loads and Stores to unaligned locations

Some unaligned loads and stores in the Armv7 architecture are described as UNPREDICTABLE. These are defined in the Armv8-A architecture to do one of the following:

- Take an alignment fault.
- Perform the specified load or store to the unaligned memory location.

K1.1.7 CONSTRAINED UNPREDICTABLE behavior associated with IT instructions and PSTATE.IT

A number of instructions in the architecture are described as being CONSTRAINED UNPREDICTABLE either:

- Anywhere within an IT block.
- As an instruction within an IT block, other than the last instruction within an IT block.

Unless otherwise stated in this manual, when these instructions are committed for execution, one of the following occurs:

- An UNDEFINED exception results.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The instructions are executed as if they had passed the Condition code check.
- The instructions execute as NOPs. This means that they behave as if they had failed the Condition code check.

The behavior might in some implementations vary from instruction to instruction, or between different instances of the same instruction.

Many instructions that are CONSTRAINED UNPREDICTABLE in an IT block are branch instructions or other non-sequential instructions that change the PC. Where these instructions are not treated as UNDEFINED within an IT block, the remaining iterations of the PSTATE.IT state machine must be treated in one of the following ways:

- PSTATE.IT is cleared to 0.
- PSTATE.IT advances for either a sequential or a nonsequential change of the PC in the same way as it does for instructions that are not CONSTRAINED UNPREDICTABLE that cause a sequential change of the PC.

Note

This does not apply to an instruction that is the last instruction in an IT block.

The instructions addressed by the updated PC must do one of the following:

- Execute as if they had passed the Condition code check for the remaining iterations of the PSTATE.IT state machine.
- Execute as NOPs. That is, they behave as if they had failed the Condition code check for the remaining iterations of the PSTATE.IT state machine.
- Execute as if they were unconditional, or, if the instructions are part of another IT block, in accordance with the behavior described in *Branching into an IT block* on page K1-7832.

The behavior might in some implementations vary from instruction to instruction, or between different instances of the same instruction.

For exception returns or Debug state exits that cause PSTATE.IT to be set to a reserved value in T32 state or that return to A32 state with a nonzero value in PSTATE.IT, the PSTATE.IT bits are forced to '00000000'. The reserved values are:

```
PSTATE.IT[7:4] != '0000' && PSTATE.IT[3:0] = '0000'
PSTATE.IT[2:0] != '000' when SCTLR/SCTLR_EL_1.ITD == '1'
```

Exception returns or Debug state exits that set PSTATE.IT to a non-reserved value in T32 state can occur when the flow of execution returns to a point:

- Outside an IT block, but with the PSTATE.IT bits set to a value other than '00000000'.
- Inside an IT block, but with a different value of the PSTATE.IT bits than if the IT block had been executed without an exception return or Debug state exit.

In this case the instructions at the target of the exception return or Debug state exit must do one of the following:

- Execute as if they passed the Condition code check for the remaining iterations of the PSTATE.IT state machine.
- Execute as NOPs. That is, they behave as if they failed the Condition code check for the remaining iterations of the PSTATE.IT state machine.
- Execute as if they were unconditional, or as if the instruction were part of another IT block, in accordance with the behavior in *Branching into an IT block* on page K1-7832.

The remaining iterations of the PSTATE.IT state machine must behave in one of the following ways:

- The PSTATE.IT state machine advances as if it were in an IT block.
- The PSTATE.IT bits are ignored.
- The PSTATE.IT bits are forced to '00000000'.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K1.1.8 Unallocated System register access instructions

In Armv8-A, accesses to unallocated System register encodings are UNDEFINED.

This includes:

- Reads using encodings that are defined as WO.
- Writes using encodings that are defined as RO.
- `MCR` or `MRC` accesses to using a set of {`coproc`, `CRn`, `opc1`, `CRm`, `opc2`} values that the Armv7 architecture defined as UNPREDICTABLE.
- Accesses to System registers in the (`coproc`==`0b111x`) encoding space that the Armv7 architecture defined as UNPREDICTABLE when particular functionality was not implemented, when an Armv8 implementation does not include the Exception level that provides that functionality.

K1.1.9 SBZ or SBO fields T32 and A32 in instructions

Many of the A32 and T32 instructions have (0) or (1) in the instruction decode to indicate *should-be-zero*, SBZ, or *should-be-one*, SBO. If the instruction bit pattern of an instruction is executed with these fields not having the *should be* values, one of the following must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction operates as if the bit had the *should-be* value.
- Any destination registers of the instruction become UNKNOWN.

The exceptions to this rule are:

- *LDM, LDMIA, LDMFD* on page F5-4398.
- *LDMDB, LDMEA* on page F5-4408.
- *LDR (literal)* on page F5-4417.
- *LDRB (literal)* on page F5-4427.
- *LDRD (immediate)* on page F5-4435.
- *LDRD (register)* on page F5-4441.
- *LDRD (literal)* on page F5-4438.
- *LDRH (literal)* on page F5-4456.
- *LDRSB (literal)* on page F5-4467.
- *LDRSH (literal)* on page F5-4478.
- *POP* on page F5-4585.
- *PUSH* on page F5-4592.
- *SDIV* on page F5-4672.
- *STM, STMIA, STMEA* on page F5-4765.
- *STMDB, STMFD* on page F5-4773.
- *UDIV* on page F5-4887.

K1.1.10 UNPREDICTABLE cases in immediate constants in T32 data-processing instructions

The description of immediate constants in T32 data processing *Modified immediate constants in T32 instructions* on page F2-4095 include constant values that were UNPREDICTABLE in Armv7. *Instruction encodings* on page F2-4076 describes 32-bit T32 instructions as {`hw1`, `hw2`}, where `hw1` is the left-hand halfword in the 32-bit encoding diagram for the instruction. The UNPREDICTABLE cases are those where both:

- `hw2`[7:0] == `0b0000000`.
- `hw1`[10] == `0` and either:
  - `hw2`[14:12] == `0b001`.
  - `hw2`[14:12] == `0b010`.
  - `hw2`[14:12] == `0b011`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In Armv8 the CONSTRAINED UNPREDICTABLE behavior is that these encodings produce the value `0b0000000`.

K1.1.11 UNPREDICTABLE cases in immediate constants in Advanced SIMD instructions

The description of immediate constants in *Modified immediate constants in T32 and A32 Advanced SIMD instructions* on page F2-4097 include constant values that were UNPREDICTABLE in Armv7. The UNPREDICTABLE cases are those where:

- The bits that the encoding diagram shows as `abcd` are all 0.

  In the A32 encoding these are bits[24, 18:6, 3:0]. In the T32 encoding they are bits {`hw1`[12, 2:0], `hw2`[3:0]}.

- The bits that the encoding diagram shows as `cmode[3:1]` are one of {`0b001`, `0b010`, `0b011`, `0b101`, `0b110`}.

  In the A32 encoding these are bits[11:9]. In the T32 encoding they are bits `hw2`[11:9].

In Armv8 the CONSTRAINED UNPREDICTABLE behavior is that these encodings produce an immediate constant value of zero.

K1.1.12 CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values

The Arm architecture allows copies of control values or data values to be cached in a cache or TLB. This can lead to CONSTRAINED UNPREDICTABLE behavior if the cache or TLB has not been correctly invalidated following a change of the control or data values.

Unless explicitly stated otherwise, the behavior of the PE is consistent with one of:

- The old data or control value.
- The new data or control value.
- An amalgamation of the old and new data or control values.

In an implementation that includes ARMv8.2-TTCNP, this CONSTRAINED UNPREDICTABLE case can arise from misprogramming when setting TTBR.CnP to 1, as identified in the descriptions of the TTBR.CnP field. In this case, for a particular TTBR, the behavior of the PE is consistent with one of:

- The value of the translation table entry pointed to by that TTBR on one of the PEs within the Inner Shareable domain for which both the value of TTBR.CnP is 1 and the other conditions for sharing translation table entries pointed to by that TTBR are met.

- An amalgamation of the values of the translation table entries pointed to by that TTBR on two or more of the PEs within the Inner Shareable domain for which both the value of TTBR.CnP is 1 and the other conditions for sharing translation table entries pointed to by that TTBR are met.

Note

If the *Effective value* of a control or data value that determines the behavior of the PE results from the amalgamation of two or more values then that *Effective value* must not generate a privilege violation. So, for example:

- Where the CONSTRAINED UNPREDICTABLE behavior occurs because inadequate invalidation of the TLB causes multiple hits in the TLB, the failure to invalidate the TLB by software executing at a given Exception level and Security state must not make it possible to access regions of memory with permissions or attributes that could not be accessed at that Exception level and Security state.

- Where the CONSTRAINED UNPREDICTABLE behavior occurs because of a programming error, on one or more PEs in the Inner Shareable domain, when using a TTBR.CnP value of 1 to share translation table entries, the misprogramming must not make it possible to access regions of memory with permissions or attributes that could not be accessed at the Exception level of that TTBR and the Security state corresponding to the translation table entries being shared.

Alternatively to this CONSTRAINED UNPREDICTABLE behavior, an implementation detecting multiple hits within a TLB might generate an exception, reporting the exception using the TLB Conflict fault code, see *TLB conflict aborts* on page G5-5993.

The choice between the behaviors might, in some implementations, vary for each use of a control or data value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K1.1.13 CONSTRAINED UNPREDICTABLE behavior due to inadequate context synchronization

The Arm architecture requires that changes to System registers must be synchronized before they take effect. This can lead to CONSTRAINED UNPREDICTABLE behavior if the synchronization has not been performed.

In these cases, the behavior of the PE is consistent with the unsynchronized control value being either the old value or the new value.

Where multiple control values are updated but not yet synchronized, each control value might independently be the old value or the new value.

In addition, where the unsynchronized control value applies to different areas of functionality, or what an implementation has constructed as different areas of functionality, those areas might independently treat the control value as being either the old value or the new value.

The choice between these behaviors might, in some implementations, vary for each use of a control value.

K1.1.14 Translation Table Base Address alignment

A misaligned Translation Table Base Address can occur if:

- The VMSAv8-32 Short-descriptor translation table format is enabled and TTBR0[13-N:7], which is defined to be RES0, contains one or more nonzero values.
- The VMSAv8-32 Long-descriptor translation table format is enabled, and TTBR0[x-1:3], TTBR1[x-1:3], HTTBR[x-1:3], or VTTBR[x-1:3], which are defined to be RES0, contain one or more nonzero values.

In the event of a misaligned Translation Table Base Address, one of the following behaviors must occur:

- The field that is defined to be RES0 is treated as if all bits were zero:
  - The value that is read back might be the value written or it might be zero.
- The calculation of an address for a translation table walk using that register might be corrupted in those bits that are nonzero.

K1.1.15 Handling of System register control fields for Advanced SIMD and floating-point operation

For historical reasons described in *Background to the System register interface* on page G1-5770, each of the CPACR, HCPTR, and NSACR has a pair of control fields that were defined to have identical functionality for controlling Advanced SIMD and floating-point operation. These fields are:

- CPACR.{cp10, cp11}.
- HCPTR.{TCP10, TCP11}.
- NSACR.{cp10, cp11}.

The architecture requires that both fields in one of these pairs are programmed to the same value. If this is not done, then the CONSTRAINED UNPREDICTABLE behavior is that behavior is the same as if the cp11, or TCP11, control field was equal to the cp10, or TCP10, field in all respects other than the value read back by a direct read of the register. After a register write that writes different values to the two fields of a pair, a direct read of the register might return an UNKNOWN value for the cp11 or TCP11 field.

Note

This means that, when different values are written to the {cp10, cp11} fields in a single register, the architecture permits but does not require that a read of that register returns the value written to the cp11 field.

CONSTRAINED UNPREDICTABLE CPACR and NSACR settings

If CPACR.cp<n> contains the encoding `'10'`, then one of the following behaviors must occur:

- The encoding maps onto any of the allocated values, but otherwise does not cause UNPREDICTABLE behavior.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The encoding causes effects that could be achieved by a combination of more than one of the allocated encodings.

Note

In Armv7, CPACR had a D32DIS bit, and NSACR had an NSD32DIS bit. There is no CPACR.D32DIS or NSACR.NSD32DIS in Armv8-A, and the corresponding bits in the two registers are RES0.

K1.1.16 Mapping of non-idempotent memory locations using the Normal memory type

If non-idempotent memory locations are mapped using the Normal memory type, the state of the non-idempotent memory location may become corrupted in the following circumstances:

- Speculative read accesses may cause accesses to the non-idempotent memory locations that would not occur as part of a simple sequential execution.
- Writes to non-idempotent memory locations might be merged or split. In this case, the number and size of writes seen by the memory location might not be the number and size that occur as part of a simple sequential execution.

K1.1.17 The Performance Monitors Extension

The following subsections describe CONSTRAINED UNPREDICTABLE behaviors when accessing the Performance Monitors Extension in AArch32 state:

- *CONSTRAINED UNPREDICTABLE accesses to PMXEVTYPER or PMXEVCNTR*.
- *CONSTRAINED UNPREDICTABLE accesses to PMEVCNTR<n> and PMEVTYPER<n>* on page K1-7838.
- *CONSTRAINED UNPREDICTABLE behavior caused by HDCR.HPMN* on page K1-7839.

CONSTRAINED UNPREDICTABLE accesses to PMXEVTYPER or PMXEVCNTR

If ARMv8.6-FGT is implemented, and EL2 is implemented in the current Security state, and EL1 is using AArch64, permitted access to PMXEVCNTR and PMXEVTYPER are not CONSTRAINED UNPREDICTABLE.

Otherwise, if PMSELR.SEL is greater than the number of counters accessible at this Exception level, accesses to PMXEVTYPER or PMXEVCNTR can cause CONSTRAINED UNPREDICTABLE behavior. This occurs when one of the following is true:

- If PMSELR.SEL is not equal to 31, and PMSELR.SEL is greater than or equal to PMCR.N, and the PE is executing in EL2 or EL3.
- If ARMv8.4-SecEL2 is disabled or is not implemented, PMSELR.SEL is not 31, and PMSELR.SEL is greater than or equal to PMCR.N, and the PE is executing in Secure EL1 or Secure EL0.
- If PMSELR.SEL is not 31, and PMSELR.SEL is greater than or equal to HDCR.HPMN, and the PE is executing in EL1 or EL0.

In these UNPREDICTABLE cases, one of the following behaviors must occur:

- Accesses to PMXEVTYPER or PMXEVCNTR from that mode are UNDEFINED.
- Accesses to PMXEVTYPER or PMXEVCNTR from that mode behave as RAZ/WI.
- Accesses to PMXEVTYPER or PMXEVCNTR from that mode execute as NOPs.
- Accesses to PMXEVTYPER or PMXEVCNTR from that mode behave as if PMSELR.SEL contains an UNKNOWN value that is less than the number of counters accessible at the current Exception level and Security state.
- Accesses to PMXEVTYPER or PMXEVCNTR behave as if PMSELR.SEL is 31.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If EL2 is implemented and enabled in the current Security state, and PMSELR.SEL is less than the number of implemented counters but greater than the number of accessible counters at this Exception level, access to PMXEVTYPER or PMXEVCNTR from EL1 or permitted access from EL0 is trapped to EL2.

If PMSELR.SEL is equal to 31, then one of the following behaviors must occur:

- Accesses to PMXEVCNTR are UNDEFINED.
- Accesses to PMXEVCNTR behave as RAZ/WI.
- Accesses to PMXEVCNTR execute as NOPs.
- Accesses to PMXEVCNTR behave as if PMSELR.SEL contains an UNKNOWN value that is less than the number of counters accessible at the current Exception level and Security state.
- If EL2 is implemented and enabled in the current Security state, for an access to PMXEVCNTR from EL1 or a permitted access from EL0, if the counter is implemented but not accessible at the current Exception level, the register access is trapped to EL2.

Note

If EL2 is implemented and enabled in the current Security state, HDCR.HPMN, or MDCR_EL2.HPMN, identifies the number of accessible counters at EL0 or EL1. Otherwise, the number of accessible counters is the number of implemented counters.

Accesses from EL0 to PMXEVCNTR are permitted when:

- EL1 is using AArch32 and the values of PMUSERENR.{ER, EN} are both 1.
- EL1 is using AArch64 and the values of PMUSERENR_EL0.{ER, EN} are both 1.

Accesses from EL0 to PMXEVTYPER are permitted when:

- EL1 is using AArch32 and the value of PMUSERENR.EN is 1.
- EL1 is using AArch64 and the value of PMUSERENR_EL0.EN is 1.

CONSTRAINED UNPREDICTABLE accesses to PMEVCNTR<n> and PMEVTYPER<n>

If ARMv8.6-FGT is implemented, and EL2 is implemented in the current Security state, and EL1 is using AArch64, permitted access to PMEVCNTR<n> and PMEVTYPER<n> are not CONSTRAINED UNPREDICTABLE.

Otherwise, if <n> is greater than the number of counters available in the current Exception level and state, reads and writes of PMEVCNTR<n> and PMEVTYPER<n> are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.
- If EL2 is implemented and enabled in the current Security state, for an access to PMEVCNTR<n> or PMEVTYPER<n> from EL1 or a permitted access from EL0, if the counter is implemented but not accessible at the current Exception level, the register access is trapped to EL2.

  Accesses from EL0 are permitted to PMEVCNTR<n> when:

  — EL1 is using AArch32 and the values of PMUSERENR.{ER, EN} are both 1.

  — EL1 is using AArch64 and the values of PMUSERENR_EL0.{ER, EN} are both 1.

  Accesses from EL0 are permitted to PMEVTYPER<n> when:

  — EL1 is using AArch32 and the value of PMUSERENR.EN is 1.

  — EL1 is using AArch64 and the value of PMUSERENR_EL0.EN is 1.

Note

If EL2 is implemented and enabled in the current Security state, at EL0 and EL1, HDCR.HPMN, or MDCR_EL2.HPMN, identifies the number of accessible counters. Otherwise, the number of accessible counters is the number of implemented counters.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CONSTRAINED UNPREDICTABLE behavior caused by HDCR.HPMN

If HDCR.HPMN is set to 0 or to a value greater than PMCR.N, then the CONSTRAINED UNPREDICTABLE behavior is:

- The value returned by a direct read of HDCR.HPMN is UNKNOWN.
- Either:
  - An UNKNOWN number of counters are reserved for EL2 use. That is, the PE behaves as if HDCR.HPMN is set to an UNKNOWN non-zero value less than PMCR.N.
  - All counters are reserved for EL2 and EL3 use, meaning no counters are accessible from EL1 and EL0.

K1.1.18 The Activity Monitors Extension

The following subsections describe CONSTRAINED UNPREDICTABLE behaviors when accessing the Activity Monitors registers in AArch32 state:

- *CONSTRAINED UNPREDICTABLE accesses to AMEVCNTR0<n> and AMEVTYPER0<n>*.
- *CONSTRAINED UNPREDICTABLE accesses to AMEVCNTR1<n> and AMEVTYPER1<n>*.
- *CONSTRAINED UNPREDICTABLE accesses to AMCNTENCLR1 and AMCNTENSET1*.

CONSTRAINED UNPREDICTABLE accesses to AMEVCNTR0<n> and AMEVTYPER0<n>

If <n> is greater than the number of architected activity monitor event counters, reads and writes of AMEVCNTR0<n> and AMEVTYPER0<n> are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a `NOP`.

Note

AMCGCR.CG0NC identifies the number of architected activity monitor event counters.

CONSTRAINED UNPREDICTABLE accesses to AMEVCNTR1<n> and AMEVTYPER1<n>

If <n> is greater than the number of auxiliary activity monitor event counters, reads and writes of AMEVCNTR1<n> and AMEVTYPER1<n> are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a `NOP`.

Note

AMCGCR.CG1NC identifies the number of auxiliary activity monitor event counters.

CONSTRAINED UNPREDICTABLE accesses to AMCNTENCLR1 and AMCNTENSET1

If the number of auxiliary activity monitor event counters that are implemented is zero, reads and writes of AMCNTENCLR1 and AMCNTENSET1 are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a `NOP`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

The number of auxiliary activity monitor event counters that are implemented is zero exactly when AMCFGR.NCG == `0b0000`.

K1.1.19 Syndrome register handling for CONSTRAINED UNPREDICTABLE instructions treated as UNDEFINED

When a CONSTRAINED UNPREDICTABLE instruction is treated as UNDEFINED, this generates an exception:

- If this exception is taken to an Exception level that is using AArch64 then ESR_ELx is UNKNOWN.
- If this exception is taken to EL2 and EL2 is using AArch32, then the HSR is unknown.

Note

The value written to ESR or HSR must be consistent with a value that could be created as the result of an exception from the same Exception level that generated the exception, but resulted from a situation that is not CONSTRAINED UNPREDICTABLE at that Exception level. This is to avoid a possible privilege violation.

K1.1.20 Out of range VA

If the PE executes an instruction for which the instruction address, size, and alignment mean it contains the bytes `0xFFFF FFFF` and `0x0000 0000`, then the bytes that wrap around and appear to be from `0x0000 0000` onwards come from an UNKNOWN address.

If the PE executes a load or store instruction for which the computed address, total access size, and alignment mean it accesses bytes `0xFFFF FFFF` and `0x0000 0000`, then the bytes that wrap around and appear to be from `0x0000 0000` onwards come from an UNKNOWN address.

K1.1.21 Instruction fetches from Device memory

Instruction fetches from Device memory are CONSTRAINED UNPREDICTABLE.

If a location in memory has the Device attribute and is not marked as execute-never, then an implementation might perform speculative instruction accesses to this memory location when address translation is enabled.

If a branch causes the program counter to point to a location in memory with the Device attribute that is not marked as execute-never for the current Exception level for instruction fetches, then an implementation must perform one of the following behaviors:

- It treats the instruction fetch as if it were to a memory location with the Normal, Non-cacheable attribute.
- It generates a Permission fault.

K1.1.22 Multi-access instructions that load the PC from Device memory

Multi-access instructions that load the PC from Device memory when address translation is enabled are UNPREDICTABLE in AArch32 state. In the Armv8-A architecture in AArch32 state an implementation must perform one of the following behaviors:

- It loads the PC from the memory location as if the memory location had the Normal Non-cacheable attribute.
- It generates a permission fault.

K1.1.23 Programming CSSELR.Level for a cache level that is not implemented

If CSSELR.Level is programmed to a cache level that is not implemented, then a read of CSSELR returns an UNKNOWN value in CSSELR.Level.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If CSSELR.Level is programmed to a cache level that is not implemented, then on a read of CCSIDR an implementation must perform one of the following behaviors:

- The CCSIDR read is treated as a NOP.
- The CCSIDR read is UNDEFINED.
- The CCSIDR read returns an UNKNOWN value.

When ARMv8.3-CCIDX is implemented, CCSIDR2 is implemented. If CSSELR.Level is programmed to a cache level that is not implemented, then on a read of CCSIDR2 an implementation must perform one of the following behaviors:

- The CCSIDR2 read is treated as a NOP.
- The CCSIDR2 read is UNDEFINED.
- The CCSIDR2 read returns an UNKNOWN value.

K1.1.24 Crossing a page boundary with different memory types or Shareability attributes

A memory access from a load or store instruction that crosses a page boundary to a memory location that has a different memory type or Shareability attribute results in CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation must perform one of the following behaviors:

- Each memory access generated by the instruction uses the memory type and Shareability attribute associated with its own address.
- The instruction generates an alignment fault caused by the memory type.

  For the Non-secure PL1&0 translation regime:

  — If the stage 1 translation causes the mismatch, the resulting exception is taken to PL1.

  — If the stage 2 translation causes the mismatch, the resulting exception is taken to PL2.

  — If both stages of translation cause the mismatch, the resulting exception can be taken to either PL1 or PL2.
- The instruction executes as a NOP.

K1.1.25 Crossing a 4KB boundary with a Device access

A memory access from a load or store instruction to Device memory that crosses a 4KB boundary results in CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation must perform one of the following behaviors:

- All memory accesses generated by the instruction are performed as if the presence of the boundary had no effect on the memory accesses.
- All memory accesses generated by the instruction are performed as if the presence of the boundary had no effect on the memory accesses, except that there is no guarantee of ordering between memory accesses.
- The instruction generates an Alignment fault caused by the memory type.

  For the Non-secure PL1&0 translation regime:

  — If the stage 1 translation causes the boundary to be crossed then the resulting exception is taken to PL1.

  — If the stage 2 translation causes the boundary to be crossed then the resulting exception is taken to PL2.

  — If both stages of translation cause the boundary to be crossed then the resulting exception can be taken to either PL1 or PL2.
- The instruction executes as a NOP.

Note

The boundary referred to is between two Device memory regions that are both of 4KB and aligned to 4KB.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K1.1.26 UNPREDICTABLE behaviors with Load-Exclusive/Store-Exclusive pairs

Load-Exclusive and Store-Exclusive instruction usage restrictions on page E2-4029 defines a Load-Exclusive/Store-Exclusive pair, and identifies various CONSTRAINED UNPREDICTABLE behaviors associated with using Load-Exclusive/Store-Exclusive pairs. These cases were UNPREDICTABLE in Armv7. In summary, these cases are:

- The target virtual address of a `StoreExcl` instruction is different from the virtual address of the preceding `LoadExcl` instruction in the same thread of execution.
- The transaction size of a `StoreExcl` instruction is different from the transaction size of the preceding `LoadExcl` instruction in the same thread of execution.
- The memory attributes for a `StoreExcl` instruction are different from the memory attributes for the preceding `LoadExcl` instruction in the same thread of execution, either:
  - Because the translation of the accessed address changes between the `LoadExcl` instruction and the `StoreExcl` instruction.
  - Because the `LoadExcl` instruction and the `StoreExcl` instruction use different virtual addresses, with different attributes, that point to the same physical address.

In addition, the effect of a data or unified cache invalidate, clean, or clean and invalidate instruction on a local or global Exclusives monitor that is in the Exclusive Access state is CONSTRAINED UNPREDICTABLE.

See the descriptions in *Load-Exclusive and Store-Exclusive instruction usage restrictions* on page E2-4029 for the permitted behavior in each of these cases, and any constraints that might apply to whether the case is CONSTRAINED UNPREDICTABLE.

Note

Additional CONSTRAINED UNPREDICTABLE cases can apply to Load-Exclusive and Store-Exclusive instructions, see *CONSTRAINED UNPREDICTABLE behavior for A32 and T32 System instructions in the base instruction set* on page K1-7843.

K1.1.27 CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings

The A32 and T32 instruction sets include encodings that result in CONSTRAINED UNPREDICTABLE behavior when they are decoded.

CONSTRAINED UNPREDICTABLE behavior of CRC32 instruction encodings

In the A32 and T32 instruction sets, there are encodings of the CRC32 and CRC32C instructions that result in CONSTRAINED UNPREDICTABLE behavior. These encodings are listed in the following places in the A32 and T32 instruction sets:

- *Cyclic Redundancy Check* on page F4-4188 for the A32 instruction set, with `sz = 11`.
- *Data-processing (two source registers)* on page F3-4174 for the T32 instruction set, with `op1 = 10x` and `op2 = 11`.

The CONSTRAINED UNPREDICTABLE behavior for these encodings is described in *CRC32* on page F5-4339 and *CRC32C* on page F5-4342.

CONSTRAINED UNPREDICTABLE behavior of other A32 instruction encodings

In the A32 instruction set, there are encodings that result in CONSTRAINED UNPREDICTABLE behavior. These encodings are listed in:

- *Miscellaneous* on page F4-4224.
- *Memory hints and barriers* on page F4-4234.
- *Barriers* on page F4-4235.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The CONSTRAINED UNPREDICTABLE behavior is that an implementation must treat the encodings in one of the following ways:

- The instruction encoding is UNDEFINED.
- The instruction encoding executes as NOP.

K1.1.28 Out of range values of the Set/Way/Index fields in cache maintenance instructions

In the cache maintenance by set/way instructions DCCISW, DCCSW, and DCISW, if any set/way/index argument is larger than the value supported by the implementation, then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

Note

This CONSTRAINED UNPREDICTABLE behavior applies, also, to the A64 cache maintenance by set/way instructions DC CISW, DC CSW, and DC ISW.

K1.1.29 CONSTRAINED UNPREDICTABLE behavior for A32 and T32 System instructions in the base instruction set

This section lists the CONSTRAINED UNPREDICTABLE behavior for the different A32 and T32 System instructions.

Note

If an instruction can result in CONSTRAINED UNPREDICTABLE behavior that is not specific to that particular instruction, see the relevant section in this appendix for a description of the CONSTRAINED UNPREDICTABLE behavior.

SRS (T32)

For a description of this instruction and the encoding, see *SRS, SRSDA, SRSDB, SRSIA, SRSIB* on page F5-4729.

CONSTRAINED UNPREDICTABLE behavior

For all encodings:

- If the instruction specifies an illegal mode field, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
  - R13 of the current mode is used.
  - The store occurs to an UNKNOWN address, and if the instruction specifies writeback, any general-purpose register that can be accessed without privilege violation from the current Exception level become UNKNOWN.

SRS (A32)

For a description of this instruction and the encoding, see *SRS, SRSDA, SRSDB, SRSIA, SRSIB* on page F5-4729.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CONSTRAINED UNPREDICTABLE behavior

For all encodings:

- If the instruction specifies an illegal mode field, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - R13 of the current mode is used.
  - The store occurs to an UNKNOWN address, and if the instruction specifies writeback, any general-purpose register that can be accessed without privilege violation from the current Exception level become UNKNOWN.

SUBS PC, LR and related instructions (T32)

For a description of this instruction and the encoding, see the exception return form of *SUB, SUBS (immediate)* on page F5-4831.

CONSTRAINED UNPREDICTABLE behavior

For all encodings:

- If this instruction is executed in User mode or in System mode, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
- If the instruction transfers an illegal mode encoding to PSTATE.M, then this invokes the illegal exception return.

  Note

  An illegal mode encoding is either an unallocated mode encoding or one that is not accessible at the current Exception level.

For encoding T5:

- If `hw2`[3:0] are `0b1110`, and the instruction is executed when not in Hyp mode, System mode, or User mode, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction is treated as a `NOP`.
  - The instruction is treated as if `hw2`[3:0] are `0b1110`.
  - The program counter is set using the value in the register specified by `hw2`[3:0].

SUBS PC. LR and related instructions (A32)

For a description of this instruction and the encoding, see the exception return forms of *MOV, MOVS (register)* on page F5-4517 and *SUB, SUBS (immediate)* on page F5-4831.

CONSTRAINED UNPREDICTABLE behavior

For all encodings:

- If this instruction is executed in User mode or in System mode, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
- If the instruction transfers an illegal mode encoding to PSTATE.M, then this invokes the illegal exception return.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

An illegal mode encoding is either an unallocated mode encoding or one that is not accessible at the current Exception level.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K1.1.30 CONSTRAINED UNPREDICTABLE behavior, A32 and T32 Advanced SIMD and floating-point instructions

This section lists the CONSTRAINED UNPREDICTABLE behavior for the different A32 and T32 Advanced SIMD and floating-point instructions listed in *Alphabetical list of Advanced SIMD and floating-point instructions* on page F6-4956.

Note

- The pseudocode used in this section to describe cases that can result in CONSTRAINED UNPREDICTABLE behavior does not necessarily match the encoding specific pseudocode for a specific instruction.
- If an instruction can result in CONSTRAINED UNPREDICTABLE behavior that is not specific to that particular instruction, see the relevant section in this appendix for a description of the CONSTRAINED UNPREDICTABLE behavior.

VCVT (between floating-point and fixed-point)

For a description of this instruction and the encoding, see *VCVT (between floating-point and fixed-point, floating-point)* on page F6-5115.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

VLD1 (multiple single elements)

For a description of this instruction and the encoding, see *VLD1 (multiple single elements)* on page F6-5213.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

VLD1 (single element to all lanes)

For a description of this instruction and the encoding, see *VLD1 (single element to all lanes)* on page F6-5210.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

VLD2 (multiple 2-element structures)

For a description of this instruction and the encoding, see *VLD2 (multiple 2-element structures)* on page F6-5230.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

VLD2 (single 2-element structure to one lane)

For a description of this instruction and the encoding, see *VLD2 (single 2-element structure to one lane)* on page F6-5221.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

VLD2 (single 2-element structure to all lanes)

For a description of this instruction and the encoding, see *VLD2 (single 2-element structure to all lanes)* on page F6-5227.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

VLD3 (multiple 3-element structures)

For a description of this instruction and the encoding, see *VLD3 (multiple 3-element structures)* on page F6-5244.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

VLD3 (single 3-element structure to one lane)

For a description of this instruction and the encoding, see *VLD3 (single 3-element structure to one lane)* on page F6-5235.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

VLD3 (single 3-element structure to all lanes)

For a description of this instruction and the encoding, see *VLD3 (single 3-element structure to all lanes)* on page F6-5241.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

VLD4 (multiple 4-element structures)

For a description of this instruction and the encoding, see *VLD4 (multiple 4-element structures)* on page F6-5256.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

VLD4 (single 4-element structure to one lane)

For a description of this instruction and the encoding, see *VLD4 (single 4-element structure to one lane)* on page F6-5247.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

VLD4 (single 4-element structure to all lanes)

For a description of this instruction and the encoding, see *VLD4 (single 4-element structure to all lanes)* on page F6-5253.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

VLDM

For a description of this instruction and the encoding, see *VLDM, VLDMDB, VLDMIA* on page F6-5259.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

VMOV (between two general-purpose registers and two single-precision registers)

For a description of this instruction and the encoding, see *VMOV (between two general-purpose registers and two single-precision registers)* on page F6-5337.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

VMOV (between two general-purpose registers and a doubleword floating-point register)

For a description of this instruction and the encoding, see *VMOV (between two general-purpose registers and a doubleword floating-point register)* on page F6-5316.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

VST1 (multiple single elements)

For a description of this instruction and the encoding, see *VST1 (multiple single elements)* on page F6-5580.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

VST2 (multiple 2-element structures)

For a description of this instruction and the encoding, see *VST2 (multiple 2-element structures)* on page F6-5594.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

VST2 (single 2-element structure from one lane)

For a description of this instruction and the encoding, see *VST2 (single 2-element structure from one lane)* on page F6-5588.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

VST3 (multiple 3-element structures)

For a description of this instruction and the encoding, see *VST3 (multiple 3-element structures)* on page F6-5605.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

VST3 (single 3-element structure from one lane)

For a description of this instruction and the encoding, see *VST3 (single 3-element structure from one lane)* on page F6-5599.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

VST4 (multiple 4-element structures)

For a description of this instruction and the encoding, see *VST4 (multiple 4-element structures)* on page F6-5614.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

VST4 (single 4-element structure from one lane)

For a description of this instruction and the encoding, see *VST4 (single 4-element structure from one lane)* on page F6-5608.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

VSTM

For a description of this instruction and the encoding, see *VSTM, VSTMDB, VSTMIA* on page F6-5617.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K1.1.31 CONSTRAINED UNPREDICTABLE behaviors associated with the VTCR

The following subsections describe the CONSTRAINED UNPREDICTABLE behavior associated with programming the VTCR:

- *Misprogramming VTCR.S*.
- *Misprogramming VTCR.{SL0, T0SZ}*.

Misprogramming VTCR.S

VTCR.S must be programmed to the value of T0SZ[3], or the effect is CONSTRAINED UNPREDICTABLE. For the Armv8-A architecture, if VTCR.S is not programmed correctly, then the VTCR.T0SZ value is treated as an UNKNOWN value.

Note

The CONSTRAINED UNPREDICTABLE behavior described in *Misprogramming VTCR.{SL0, T0SZ}* means the UNKNOWN VTCR.T0SZ value might generate a Translation fault.

Misprogramming VTCR.{SL0, T0SZ}

If the stage 2 input address size, as programmed in VTCR.T0SZ, is out of range with respect to the starting level, as programmed in the VTCR.SL0 field, or the VTCR.SL0 field is programmed to a reserved value, then at the time of a translation walk that uses the stage 2 translation, a stage 2 level 1 Translation Fault is generated.

K1.1.32 CONSTRAINED UNPREDICTABLE behavior of EL2 features

The following sections describe CONSTRAINED UNPREDICTABLE behavior that can occur in an implementation that includes EL2 where EL2 can use AArch32:

- *ERET in User mode or System mode*.
- *Accessing Hyp mode from outside Hyp mode*.
- *Modifying PSTATE.M when in Hyp mode* on page K1-7851
- *Use of Hyp mode in Secure state* on page K1-7851.
- *Execution of Load/Store unprivileged instructions in Hyp mode* on page K1-7851.
- *Exception return to Hyp mode* on page K1-7851.
- *Accessing registers that cannot be accessed using MSR/MRS instructions* on page K1-7851.
- *Memory type handling* on page K1-7852.
- *Hyp mode TLB maintenance instructions* on page K1-7852.
- *Hyp mode VA to PA address translation instructions* on page K1-7852.
- *Stage 1 default memory type* on page K1-7852.
- *Trapping of general exceptions to Hyp mode* on page K1-7852.
- *Prevention of rootkits using Hyp mode or Secure state* on page K1-7853.
- *HVC* on page K1-7853.
- *MSR (banked register) and MRS (banked register)* on page K1-7853.

ERET in User mode or System mode

If ERET is executed in User mode or System mode, it behaves as described in *SUBS PC, LR and related instructions (T32)* on page K1-7844.

Accessing Hyp mode from outside Hyp mode

Attempting to change into Hyp mode or out of Hyp mode using the `MSR` or `CPS` instruction invokes the Armv8 illegal exception return by not changing the mode, and setting PSTATE.IL to 1.

`SRS` using the Hyp mode SP from Non-secure modes other than Hyp mode, or from Secure state, is handled as described in *SRS (T32)* on page K1-7843 and *SRS (A32)* on page K1-7843.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Modifying PSTATE.M when in Hyp mode

Attempting to change into Hyp mode or out of Hyp mode using the `MSR` or `CPS` instruction invokes the Armv8 illegal exception return by not changing the mode, and setting PSTATE.IL to 1.

`SRS` using the Hyp mode SP from Non-secure modes other than Hyp mode, or from Secure state, is handled as described in *SRS (T32)* on page K1-7843 and *SRS (A32)* on page K1-7843.

Use of Hyp mode in Secure state

Attempting to change into Hyp mode or out of Hyp mode using the `MSR` or `CPS` instruction invokes the Armv8 illegal exception return by not changing the mode, and setting PSTATE.IL to 1.

`SRS` using the Hyp mode SP from Non-secure modes other than Hyp mode, or from Secure state, is handled as described in *SRS (T32)* on page K1-7843 and *SRS (A32)* on page K1-7843.

Execution of Load/Store unprivileged instructions in Hyp mode

If `LDRT`, `LDRSHT`, `LDRHT`, `LDRSBT`, `LDRBT`, `STRT`, `STRHT` or `STRBT` are executed in Hyp mode, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the equivalent, corresponding `LDR`, `LDRSH`, `LDRH`, `LDRSB`, `LDRB`, `STR`, `STRH` or `STRB` instruction in Hyp mode.

Exception return to Hyp mode

Exception returns to Hyp mode when SCR.NS == 0 or from a Non-secure PL1 mode invokes the Armv8 illegal exception return.

Accessing registers that cannot be accessed using MSR/MRS instructions

The following `MSR` and `MRS` instructions can lead to CONSTRAINED UNPREDICTABLE behavior:

```
MSR <Rm>_<mode>, <Rn>
MSR SPSR_<mode>, <Rn>
MSR ELR_hyp, <Rn>
MRS <Rn>, <Rm>_<mode>
MRS <Rn>, SPSR_<mode>
MRS <Rn>, ELR_hyp
```

If these instructions are executed in either Secure or Non-secure User mode, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.

If the `MSR` and `MRS` instructions attempt to access a register that cannot be legally accessed, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- For `MRS` instructions, the destination general-purpose register becomes UNKNOWN.
- For `MSR` instructions, if the register specified could be accessed from the current mode by other mechanisms, then this register is UNKNOWN. Otherwise the instruction executes as a `NOP`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Memory type handling

If the attributes for a memory location after combining stage 1 and stage 2 of a translation regime is Normal Inner Non-cacheable, Outer Non-cacheable, then the shareability attributes after combining the two stages of translation is Outer Shareable.

Hyp mode TLB maintenance instructions

If a TLBIMVAH, TLBIMVALH, TLBIMVAHIS, TLBIMVALHIS, TLBIALLH, TLBIALLHIS, TLBIALLNSNH, TLBIALLNSNHIS, TLBIIPAS2, TLBIIPAS2L, TLBIIPAS2IS, or TLBIIPAS2LIS instruction is executed in a Secure Privileged mode other than Monitor mode, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a `NOP`.
- The instruction is executes as if it had been executed in Monitor mode.

For more information about these instructions see *The scope of TLB maintenance instructions* on page G5-6004.

Hyp mode VA to PA address translation instructions

If an ATS1HR or ATS1HW instruction is executed in a Secure Privileged mode other than Monitor mode, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a `NOP`.
- The instruction is executes as if it had been executed in Monitor mode.

For more information about these instructions see *Address translation instruction naming and operation summary* on page G5-6043.

Stage 1 default memory type

If HCR.DC == 1, then the behavior of the PE when executing in a Non-secure mode other than Hyp mode is consistent with:

- SCTLR.M == 0, regardless of the actual value of SCTLR.M, other than for the value returned by an explicit read of SCTLR.M.
- HCR.VM == 1, regardless of the actual value of HCR.VM, other than for an explicit read of this bit.

Trapping of general exceptions to Hyp mode

Attempting to perform an exception return to a Non-secure PL1 mode when HCR.TGE == 1 invokes an illegal exception return.

Attempting to change from Monitor mode to a Non-secure PL1 mode when HCR.TGE == 1 by executing a `CPS` or `MSR` instruction generates an Illegal Execution state exception, by not changing the mode, and setting PSTATE.IL to 1.

When EL3 is using AArch32, attempting to change from a Secure PL1 mode to a Non-secure PL1 mode when HCR.TGE is set, by changing SCR.NS from 0 to 1, results in no change of SCR.NS.

Because taking an exception into Non-secure PL1 modes leads to a CONSTRAINED UNPREDICTABLE situation, the following additional properties apply when HCR.TGE == 1:

- All exceptions that would be routed to EL1 are routed to EL2.
- Non-secure SCTLR.M is treated as being 0, regardless of its actual value, other than for an explicit read of of this bit.
- HCR.FMO, HCR.IMO, and HCR.AMO are treated as being 1, regardless of their actual value, other than for an explicit read of these bits.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- All virtual interrupts are disabled.
- Any IMPLEMENTATION DEFINED mechanisms for signaling virtual interrupts are disabled.

Prevention of rootkits using Hyp mode or Secure state

If an HVC instruction is executed in Hyp mode when SCR.HCE == 0, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.

If an SMC instruction is executed in a Secure privileged mode when SCR.SCD == 1, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.

HVC

For a description of this instruction and the encoding, see *HVC* on page F5-4375.

For the A1 encoding, if `cond field !=1110`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction executes unconditionally.
- The instruction executes conditionally.

MSR (banked register) and MRS (banked register)

Encoding and use of banked register transfer instructions on page F5-4951 identifies cases where attempted execution of an `MRS` (banked register) or `MSR` (banked register) was UNPREDICTABLE in Armv7 and becomes CONSTRAINED UNPREDICTABLE in Armv8. This includes cases where:

- The target register specified by the {`R`, `SYSm`} fields of the instruction encoding is not accessible from the PE mode in which the instruction was executed, see *Usage restrictions on the banked register transfer instructions* on page F5-4952.
- The instruction was executed specifying unallocated {`R`, `SYSm`} field values, see *Encoding the register argument in the banked register transfer instructions* on page F5-4953.

If one of these encodings for an `MSR` (banked register) or `MRS` (banked register) instruction is executed, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- An allocated `MSR` (banked register) or `MRS` (banked register) instruction is executed.

K1.1.33 Reserved values in System and memory-mapped registers and translation table entries

Unless otherwise stated, all unallocated or reserved values of fields with allocated values within the AArch32 System registers, memory-mapped registers, and translation table entries behave in one of the following ways:

- The encoding maps onto any of the allocated values, but otherwise does not cause CONSTRAINED UNPREDICTABLE behavior.
- The encoding causes effects that could be achieved by a combination of more than one of the allocated encodings.
- The encoding causes the field to have no functional effect.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

These constraints are identical to those for the equivalent AArch64 definitions, as given in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-7871.

K1.1.34 CONSTRAINED UNPREDICTABLE behavior in Debug state

Behavior in Debug state on page H2-6936 of this manual describes the CONSTRAINED UNPREDICTABLE behaviors that are specifically associated with Debug state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K1.2 AArch64 CONSTRAINED UNPREDICTABLE behaviors

It contains the following sections:

- *Overview of the constraints on AArch64 UNPREDICTABLE behaviors*.
- *SBZ or SBO fields in A64 instructions*.
- *CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values* on page K1-7856.
- *CONSTRAINED UNPREDICTABLE behavior due to inadequate context synchronization* on page K1-7856.
- *Translation table base address alignment* on page K1-7857.
- *The Performance Monitors Extension* on page K1-7857.
- *The Activity Monitors Extension* on page K1-7859.
- *Syndrome register handling for CONSTRAINED UNPREDICTABLE instructions treated as UNDEFINED* on page K1-7859.
- *Out of range virtual address* on page K1-7860.
- *Mapping of non-idempotent memory locations using the Normal memory type* on page K1-7860.
- *Instruction fetches from Device memory* on page K1-7860.
- *Programming the CSSELR_EL1.Level for a cache level that is not implemented* on page K1-7860.
- *Crossing a page boundary with different memory types or Shareability attributes* on page K1-7861.
- *Crossing a peripheral boundary with a Device access* on page K1-7861.
- *CONSTRAINED UNPREDICTABLE behaviors with Load-Exclusive/Store-Exclusive pairs* on page K1-7862.
- *CONSTRAINED UNPREDICTABLE behavior for A64 instructions* on page K1-7862.
- *Out of range values of the Set/Way/Index fields in cache maintenance instructions* on page K1-7871.
- *Reserved values in System and memory-mapped registers and translation table entries* on page K1-7871.
- *CONSTRAINED UNPREDICTABLE behavior in Debug state* on page K1-7871.

K1.2.1 Overview of the constraints on AArch64 UNPREDICTABLE behaviors

The term UNPREDICTABLE describes a number of cases where the architecture has a feature that software must not use. For execution in AArch64 state, the Armv8-A architecture specifies a narrow range of permitted behaviors. This range is the range of CONSTRAINED UNPREDICTABLE behavior. All implementations that are compliant with the architecture must follow the CONSTRAINED UNPREDICTABLE behavior.

Note

Software designed to be compatible with the Armv8-A architecture must not rely on these CONSTRAINED UNPREDICTABLE cases being handled in any way other than those listed under the heading CONSTRAINED UNPREDICTABLE.

K1.2.2 SBZ or SBO fields in A64 instructions

Some A64 instructions have (0) or (1) in the instruction decode to indicate *should-be-zero*, SBZ, or *should-be-one*, SBO, as described in *Fixed values in AArch64 instruction and System register descriptions* on page C2-197. Except for specific cases identified in *CONSTRAINED UNPREDICTABLE behaviors with Load-Exclusive/Store-Exclusive pairs* on page K1-7862, if the instruction bit pattern of an instruction is executed with these fields not having the *should be* values, one of the following must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction operates as if the bit had the *should-be* value.
- Any destination registers of the instruction become UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K1.2.3 CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values

The Arm architecture allows copies of control values or data values to be cached in a cache or TLB. This can lead to UNPREDICTABLE behavior if the cache or TLB has not been correctly invalidated following a change of the control or data values.

Unless explicitly stated otherwise, the behavior of the PE is consistent with one of:

- The old data or control value.
- The new data or control value.
- An amalgamation of the old and new data or control values.

In an implementation that includes ARMv8.2-TTCNP, this CONSTRAINED UNPREDICTABLE case can arise from misprogramming when setting TTBR.CnP to 1, as identified in the descriptions of the TTBR.CnP field. In this case, for a particular TTBR, the behavior of the PE is consistent with one of:

- The value of the translation table entry pointed to by that TTBR on one of the PEs within the Inner Shareable domain for which both the value of TTBR.CnP is 1 and the other conditions for sharing translation table entries pointed to by that TTBR are met.
- An amalgamation of the values of the translation table entries pointed to by that TTBR on two or more of the PEs within the Inner Shareable domain for which both the value of TTBR.CnP is 1 and the other conditions for sharing translation table entries pointed to by that TTBR are met.

Note

If the *Effective value* of a control or data value that determines the behavior of the PE results from the amalgamation of two or more values then that *Effective value* must not generate a privilege violation. So, for example:

- Where the CONSTRAINED UNPREDICTABLE behavior occurs because inadequate invalidation of the TLB causes multiple hits in the TLB, the failure to invalidate the TLB by software executing at a given Exception level and Security state must not make it possible to access regions of memory with permissions or attributes that could not be accessed at that Exception level and Security state.
- Where the CONSTRAINED UNPREDICTABLE behavior occurs because of a programming error, on one or more PEs in the Inner Shareable domain, when using a TTBR.CnP value of 1 to share translation table entries, the misprogramming must not make it possible to access regions of memory with permissions or attributes that could not be accessed at the Exception level of that TTBR and the Security state corresponding to the translation table entries being shared.

Alternatively to this CONSTRAINED UNPREDICTABLE behavior, an implementation detecting multiple hits in a TLB might generate an exception, reporting the exception using the TLB conflict fault code, see *TLB conflict aborts* on page D5-2667.

The choice between the behaviors might, in some implementations, vary for each use of a control or data value.

K1.2.4 CONSTRAINED UNPREDICTABLE behavior due to inadequate context synchronization

The Arm architecture requires that changes to System registers must be synchronized before they take effect. This can lead to UNPREDICTABLE behavior if the synchronization has not been performed.

In these cases, the behavior of the PE is consistent with the unsynchronized control value being either the old value or the new value.

Where multiple control values are updated but not yet synchronized, each control value might independently be the old value or the new value.

In addition, where the unsynchronized control value applies to different areas of functionality, or what an implementation has constructed as different areas of functionality, those areas might independently treat the control value as being either the old value or the new value.

The choice between these behaviors might, in some implementations, vary for each use of a control value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K1.2.5 Translation table base address alignment

In the translation table base registers TTBR0_EL1, TTBR1_EL1, TTBR0_EL2, VTTBR_EL2, and TTBR0_EL3, register bits[48:*x*] hold the translation table base address, where *x* depends on the translation table granule size and the size of the addressed translation table, as described in *Memory translation granule size* on page D5-2554. Register bits[(*x*-1):0], unless redefined for another purpose, correspond to bits[(*x*-1):0] of the translation table base address and therefore are RES0.

Note

- When ARMv8.2-LPA is implemented and the 64KB granule size is used, register bits[5:2] are redefined to hold bits[51:48] of the translation table base address.
- When ARMv8.2-TTCNP is implemented register bit[0] is redefined as the CnP bit.

For these registers, if one or more RES0 bits in register bits [(x-1):0] does not have a value of 0, this can result in a misaligned translation table base address. In this case, one of the following behaviors must occur:

- The field that is defined to be RES0 is treated as if all the bits had a value of 0:
  - The value read back might be the value written or it might be zero.
- The calculation of an address for a translation table walk using those registers might be corrupted in those bits that are nonzero.

For more information, see the appropriate TTBR.BADDR field description.

K1.2.6 The Performance Monitors Extension

The following subsections describe CONSTRAINED UNPREDICTABLE behaviors when accessing the Performance Monitors Extension in AArch64 state:

- *CONSTRAINED UNPREDICTABLE accesses to PMXEVTYPER_EL0 or PMXEVETYPER_EL0*.
- *CONSTRAINED UNPREDICTABLE accesses to PMEVCNTR<n>_EL0 and PMEVTYPER<n>_EL0* on page K1-7858.
- *CONSTRAINED UNPREDICTABLE behavior caused by MDCR_EL2.HPMN* on page K1-7859.

CONSTRAINED UNPREDICTABLE accesses to PMXEVTYPER_EL0 or PMXEVETYPER_EL0

If ARMv8.6-FGT is implemented, and EL2 is implemented in the current Security state, and EL1 is using AArch64, permitted access to PMXEVTYPER_EL0 and PMXEVCNTR_EL0 are not CONSTRAINED UNPREDICTABLE.

Otherwise, if PMSELR_EL0.SEL is greater than the number of counters accessible at this Exception level, accesses to PMXEVTYPER_EL0 and PMXEVCNTR_EL0 can cause CONSTRAINED UNPREDICTABLE behavior. This occurs when one of the following is true:

- If PMSELR_EL0.SEL is not equal to 31, and PMSELR_EL0.SEL is greater than or equal to PMCR_EL0.N, and the PE is executing in EL2 or EL3.
- If ARMv8.4-SecEL2 is disabled or is not implemented, PMSELR_EL0.SEL is not 31, and PMSELR_EL0.SEL is greater than or equal to PMCR_EL0.N, and the PE is executing in Secure EL1 or Secure EL0.
- If PMSELR_EL0.SEL is not 31, and PMSELR_EL0.SEL is greater than or equal to MDCR_EL2.HPMN, and the PE is executing in EL0 or EL1.

In these cases, one of the following behaviors must occur:

- Accesses to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 from that state are UNDEFINED.
- Accesses to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 from that state behave as RAZ/WI.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Accesses to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 from that state execute as `NOP`s.
- Accesses to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 from that state behave as if PMSELR_EL0.SEL contains an UNKNOWN value that is less than the number of counters accessible at the current Exception level and Security state.
- Accesses to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 from that state behave as if PMSELR_EL0.SEL is 31.
- If EL2 is implemented and enabled in the current Security state, and PMSELR_EL0.SEL is less than the number of implemented counters but greater than or equal to the number of accessible counters at this Exception level, access to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 from EL1 or a permitted access from EL0 is trapped to EL2.

  Note

  If EL2 is implemented and enabled in the current Security state, MDCR_EL2.HPMN identifies the number of accessible counters at EL0 or EL1. Otherwise, the number of accessible counters is the number of implemented counters.

  Accesses from EL0 to PMXEVCNTR_EL0 are permitted when:

  — EL1 is using AArch32 and the values of PMUSERENR.{ER, EN} are both 1.

  — EL1 is using AArch64 and the values of PMUSERENR_EL0.{ER, EN} are both 1.

  Accesses from EL0 to PMXEVTYPER_EL0 are permitted when:

  — EL1 is using AArch32 and the value of PMUSERENR.EN is 1.

  — EL1 is using AArch64 and the value of PMUSERENR_EL0.EN is 1.

If PMSELR_EL0.SEL is equal to 31, then one of the following behaviors must occur:

- Accesses to PMXEVCNTR_EL0 are UNDEFINED.
- Accesses to PMXEVCNTR_EL0 behave as RAZ/WI.
- Accesses to PMXEVCNTR_EL0 execute as `NOP`s.
- Accesses to PMXEVCNTR_EL0 behave as if PMSELR_EL0.SEL contains an unknown value that is less than the number of counters accessible at the current Exception level and Security state.

CONSTRAINED UNPREDICTABLE accesses to PMEVCNTR<n>_EL0 and PMEVTYPER<n>_EL0

If ARMv8.6-FGT is implemented, and EL2 is implemented in the current Security state, and EL1 is using AArch64, permitted access to PMEVCNTR<n>_EL0 and PMEVTYPER<n>_EL0 are not CONSTRAINED UNPREDICTABLE.

Otherwise, if <n> is greater than the number of counters available in the current Exception level and state, reads and writes of PMEVCNTR<n>_EL0 and PMEVTYPER<n>_EL0 are CONSTRAINED UNPREDICTABLE, the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a `NOP`.
- If EL2 is implemented and enabled in the current Security state, for an access to PMEVCNTR<n>_EL0 or PMEVTYPER<n>_EL0 from EL1 or a permitted access from EL0, if the counter is implemented but not accessible at the current Exception level, the register access is trapped to EL2.

  Accesses from EL0 to PMEVCNTR<n>_EL0 are permitted when:

  — EL1 is using AArch32 and the value of PMUSERENR.{ER, EN} are both 1.

  — EL1 is using AArch64 and the value of PMUSERENR_EL0.{ER, EN} are both 1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Accesses from EL0 to PMEVTYPER<n>_EL0 are permitted when:

— EL1 is using AArch32 and the value of PMUSERENR.EN is 1.

— EL1 is using AArch64 and the value of PMUSERENR_EL0.EN is 1.

CONSTRAINED UNPREDICTABLE behavior caused by MDCR_EL2.HPMN

If MDCR_EL2.HPMN is set to 0, or to a value larger than PMCR_EL0.N, then the following CONSTRAINED UNPREDICTABLE behavior applies:

- The value returned by a direct read of MDCR_EL2.HPMN is UNKNOWN.
- Either:
  - — An UNKNOWN number of counters are reserved for EL2 use. That is, the PE behaves as if MDCR_EL2.HPMN is set to an UNKNOWN non-zero value less than PMCR_EL0.N.
  - — All counters are reserved for EL2 and EL3 use, meaning no counters are accessible from EL1 and EL0.

K1.2.7 The Activity Monitors Extension

If <n> is greater than the number of architected activity monitor event counters, reads and writes of AMEVCNTR0<n>_EL0 and AMEVTYPER0<n>_EL0 are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a `NOP`.

Note

AMCGCR_EL0.CG0NC identifies the number of architected activity monitor event counters.

If <n> is greater than the number of auxiliary activity monitor event counters, reads and writes of AMEVCNTR1<n>_EL0and AMEVTYPER1<n>_EL0 are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a `NOP`.

Note

AMCGCR_EL0.CG1NC identifies the number of auxiliary activity monitor event counters.

If the number of auxiliary activity monitor event counters that are implemented is zero, reads and writes of AMCNTENCLR1_EL0 and AMCNTENSET0_EL0 are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a `NOP`.

Note

The number of auxiliary activity monitor event counters that are implemented is zero exactly when AMCFGR_EL0.NCG == 0b0000.

K1.2.8 Syndrome register handling for CONSTRAINED UNPREDICTABLE instructions treated as UNDEFINED

When a CONSTRAINED UNPREDICTABLE instruction is treated as UNDEFINED, ESR_ELx is UNKNOWN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

The value written to ESR_ELx must be consistent with a value that could be created as the result of an exception from the same Exception level that generated the exception, but was the result of a situation that is not CONSTRAINED UNPREDICTABLE at that Exception level. This is to avoid a possible privilege violation.

K1.2.9 Out of range virtual address

If the PE executes a load or store instruction with tagged addressing disabled in the current translation regime, and where the computed virtual address, total access size, and alignment mean that it accesses the bytes at `0xFFFF FFFF FFFF FFFF` and `0x0000 0000 0000 0000`, then the bytes that appear to be from `0x0000 0000 0000 0000` onwards are accessed at an UNKNOWN address.

If the PE executes a load or store instruction with tagged addressing enabled in the current translation regime, and where the computed address, total access size, and alignment mean that it accesses the bytes at `0xFFFF FFFF FFFF FFFF` and `0x0000 0000 0000 0000`, then the bytes that appear to be from `0x0000 0000 0000 0000` onwards are accessed at an unknown address and the tags associated with address also become unknown.

Note

Because of program counter alignment constraints, it is impossible for a PE to fetch an A64 instruction that includes both the byte at virtual address `0xFFFF FFFF FFFF FFFF` and the byte at virtual address `0x0000 0000 0000 0000`.

K1.2.10 Mapping of non-idempotent memory locations using the Normal memory type

If non-idempotent memory locations are mapped using the Normal memory type, the state of the non-idempotent memory location may become corrupted north following circumstances:

- Speculative read accesses may cause accesses to the non-idempotent memory locations that would not occur as part of a simple sequential execution.
- Writes to non-idempotent memory locations might be merged or split. In this case, the number and size of writes seen by the memory location might not be the number and size that occur as part of a simple sequential execution.

K1.2.11 Instruction fetches from Device memory

Instruction fetches from Device memory are CONSTRAINED UNPREDICTABLE.

If a location in memory has the Device attribute and is not marked as execute-never, then an implementation might perform speculative instruction accesses to this memory location at times when address translation is enabled.

If a branch causes the program counter to point to an area of memory with the Device attribute that is not marked as execute-never for the current Exception level for instruction fetches, then an implementation must perform one of the following behaviors:

- It treats the instruction fetch as if it were to a memory location with the Normal, Non-cacheable attribute.
- It generates a Permission fault.

K1.2.12 Programming the CSSELR_EL1.Level for a cache level that is not implemented

If the CSSELR_EL1.Level is programmed to a cache level that is not implemented, then a read of CSSELR_EL1 returns an UNKNOWN value in CSSELR_EL1.Level.

If CSSELR_EL1.Level is programmed to a cache level that is not implemented, then on a read of CCSIDR_EL1 an implementation must perform one of the following behaviors:

- The CCSIDR_EL1 read is treated as a `NOP`.
- The CCSIDR_EL1 read is UNDEFINED.
- The CCSIDR_EL1 read returns an UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When ARMv8.3-CCIDX is implemented, CCSIDR2_EL1 is implemented. If CSSELR_EL1.Level is programmed to a cache level that is not implemented, then on a read of CCSIDR2_EL1 an implementation must perform one of the following behaviors:

- The CCSIDR2_EL1 read is treated as a NOP.
- The CCSIDR2_EL1 read is UNDEFINED.
- The CCSIDR2_EL1 read returns an UNKNOWN value.

K1.2.13 Crossing a page boundary with different memory types or Shareability attributes

A memory access from a load or store instruction that crosses a page boundary to a memory location that has a different memory type or Shareability attribute results in CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation must perform one of the following behaviors:

- Each memory access generated by the instruction uses the memory type and Shareability attribute associated with its own address.
- The instruction generates an Alignment fault caused by the memory type.

  For the EL1&0 translation regime, when EL2 is enabled in the current Security state:
  - If the stage 1 translation generated the mismatch, the resulting exception is taken to EL1.
  - If the stage 2 translation generated the mismatch, the resulting exception is taken to EL2.
  - If both stages of translation generate the mismatch, the exception can be taken to either EL1 or EL2.
- The instruction executes as a NOP.

K1.2.14 Crossing a peripheral boundary with a Device access

Performing memory accesses from one load or store instruction to Device memory that crosses a boundary corresponding to the smallest translation granule size of the implementation causes CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation performs one of the following behaviors:

- All memory accesses generated by the instruction are performed as if the boundary has no effect on the memory accesses.
- All memory accesses generated by the instruction are performed as if the boundary has no effect on the memory accesses except that there is no guarantee of ordering between memory accesses.
- The instruction generates an alignment fault caused by the memory type.

  For the EL1&0 translation regime, when EL2 is enabled in the current Security state:
  - If the stage 1 translation causes the boundary to be crossed then the resulting exception is taken to EL1.
  - If the stage 2 translation causes the boundary to be crossed then the resulting exception is taken to EL2.
  - If both stages of translation cause the boundary to be crossed then the resulting exception can be taken to either EL1 or EL2.
- The instruction executes as a NOP.

Note

The boundary referred to is between two Device memory regions that are both:

- Of the size of the smallest implemented translation granule.
- Aligned to the size of the smallest implemented translation granule.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K1.2.15 CONSTRAINED UNPREDICTABLE behaviors with Load-Exclusive/Store-Exclusive pairs

Load-Exclusive and Store-Exclusive instruction usage restrictions on page B2-174 defines a Load-Exclusive/Store-Exclusive pair, and identifies various CONSTRAINED UNPREDICTABLE behaviors associated with using Load-Exclusive/Store-Exclusive pairs. In summary, these cases are:

- The target virtual address of a `StoreExcl` instruction is different from the virtual address of the preceding `LoadExcl` instruction in the same thread of execution.
- The transaction size of a `StoreExcl` instruction is different from the transaction size of the preceding `LoadExcl` instruction in the same thread of execution.
- The `StoreExcl` instruction accesses a different number of registers than the preceding `LoadExcl` instruction in the same thread of execution.
- The memory attributes for a `StoreExcl` instruction are different from the memory attributes for the preceding `LoadExcl` instruction in the same thread of execution, either:
  - — Because the translation of the accessed address changes between the `LoadExcl` instruction and the `StoreExcl` instruction.
  - — Because the `LoadExcl` instruction and the `StoreExcl` instruction use different virtual addresses, with different attributes, that point to the same physical address.

In addition, the effect of a data or unified cache invalidate, clean, or clean and invalidate instruction on a local or global Exclusives monitor that is in the Exclusive Access state is CONSTRAINED UNPREDICTABLE.

See the descriptions in *Load-Exclusive and Store-Exclusive instruction usage restrictions* on page B2-174 for the permitted behavior in each of these cases, and any constraints that might apply to whether the case is CONSTRAINED UNPREDICTABLE.

K1.2.16 CONSTRAINED UNPREDICTABLE behavior for A64 instructions

This section lists the CONSTRAINED UNPREDICTABLE behavior for the different A64 instructions listed in Chapter C6 *A64 Base Instruction Descriptions* and Chapter C7 *A64 Advanced SIMD and Floating-point Instruction Descriptions*.

LDAXP

For a description of this instruction and the encoding, see *LDAXP* on page C6-966.

CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs a load using the specified addressing mode, and the transfer register is set to an UNKNOWN value.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

LDNP

For a description of this instruction and the encoding, see *LDNP* on page C6-992.

CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs a load using the specified addressing mode, and the transfer register is set to an UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

LDNP (SIMD&FP)

For a description of this instruction and the encoding, see *LDNP (SIMD&FP)* on page C7-1831.

CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs a load using the specified addressing mode, and the transfer register is set to an UNKNOWN value.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

LDP

For a description of this instruction and the encoding, see *LDP* on page C6-994.

CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `(t == n || t2 == n) && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs a load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

If t == t2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs all of the loads using the specified addressing mode, and the transfer register is set to an UNKNOWN value.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

LDP (SIMD&FP)

For a description of this instruction and the encoding, see *LDP (SIMD&FP)* on page C7-1833.

CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs a load using the specified addressing mode, and the transfer register is set to an UNKNOWN value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

LDPSW

For a description of this instruction and the encoding, see *LDPSW* on page C6-997.

CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `(t == n || t2 == n) && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs a load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs all of the loads using the specified addressing mode, and the register loaded is set to an UNKNOWN value.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

LDR (immediate)

For a description of this instruction and the encoding, see *LDR (immediate)* on page C6-1000.

CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

LDRB (immediate)

For a description of this instruction and the encoding, see *LDRB (immediate)* on page C6-1009.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

LDRH (immediate)

For a description of this instruction and the encoding, see *LDRH (immediate)* on page C6-1014.

CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

LDRSB (immediate)

For a description of this instruction and the encoding, see *LDRSB (immediate)* on page C6-1019.

CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

LDRSH (immediate)

For a description of this instruction and the encoding, see *LDRSH (immediate)* on page C6-1024.

CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

LDRSW (immediate)

For a description of this instruction and the encoding, see *LDRSW (immediate)* on page C6-1029.

CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

LDXP

For a description of this instruction and the encoding, see *LDXP* on page C6-1091.

CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs a load using the specified addressing mode, and the transfer register is set to an UNKNOWN value.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

STP

For a description of this instruction and the encoding, see *STP* on page C6-1261.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `(t == n || t2 == n) && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs a store using the specified addressing mode but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

STLXP

For a description of this instruction and the encoding, see *STLXP* on page C6-1250.

CONSTRAINED UNPREDICTABLE behavior

If `s == t || (s == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the store to an UNKNOWN address.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

STLXR

For a description of this instruction and the encoding, see *STLXR* on page C6-1253.

CONSTRAINED UNPREDICTABLE behavior

If `s == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the store to an UNKNOWN address.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

STLXRB

For a description of this instruction and the encoding, see *STLXRB* on page C6-1255.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CONSTRAINED UNPREDICTABLE behavior

If `s == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the store to an UNKNOWN address.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

STLXRH

For a description of this instruction and the encoding, see *STLXRH* on page C6-1257.

CONSTRAINED UNPREDICTABLE behavior

If `s == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the store to an UNKNOWN address.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

STR (immediate)

For a description of this instruction and the encoding, see *STR (immediate)* on page C6-1264.

CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs a store using the specified addressing mode but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

STRB (immediate)

For a description of this instruction and the encoding, see *STRB (immediate)* on page C6-1269.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs a store using the specified addressing mode but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

STRH (immediate)

For a description of this instruction and the encoding, see *STRH (immediate)* on page C6-1274.

CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs a store using the specified addressing mode but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

STXP

For a description of this instruction and the encoding, see *STXP* on page C6-1319.

CONSTRAINED UNPREDICTABLE behavior

If `s == t || (s == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the store to an UNKNOWN address.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

STXR

For a description of this instruction and the encoding, see *STXR* on page C6-1322.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CONSTRAINED UNPREDICTABLE behavior

If `s == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the store to an UNKNOWN address.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

STXRB

For a description of this instruction and the encoding, see *STXRB* on page C6-1324.

CONSTRAINED UNPREDICTABLE behavior

If `s == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the store to an UNKNOWN address.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

STXRH

For a description of this instruction and the encoding, see *STXRH* on page C6-1326.

CONSTRAINED UNPREDICTABLE behavior

If `s == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the store to an UNKNOWN address.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K1.2.17 Out of range values of the Set/Way/Index fields in cache maintenance instructions

In the cache maintenance by set/way instructions DC CISW, DC CSW, and DC ISW, if any set/way/index argument is larger than the value supported by the implementation, then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

Note

This CONSTRAINED UNPREDICTABLE behavior applies, also, to the AArch32 cache maintenance by set/way instructions DCCISW, DCCSW, and DCISW.

K1.2.18 Reserved values in System and memory-mapped registers and translation table entries

Unless otherwise stated in this manual, all unallocated or reserved values of fields with allocated values within AArch64 System registers, memory-mapped registers, and translation table entries behave in one of the following ways:

- The unallocated value maps onto any of the allocated values, but otherwise does not cause CONSTRAINED UNPREDICTABLE behavior.
- The unallocated value causes effects that could be achieved by a combination of more than one of the allocated values.
- The unallocated value causes the field to have no functional effect.

Note

These constraints are identical to those for the equivalent AArch32 definitions, as given in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-7853.

K1.2.19 CONSTRAINED UNPREDICTABLE behavior in Debug state

Behavior in Debug state on page H2-6936 of this manual describes the CONSTRAINED UNPREDICTABLE behaviors that are specifically associated with Debug state.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Appendix K2
Recommended External Debug Interface

This appendix describes the recommended external debug interface. It contains the following sections:

- *About the recommended external debug interface* on page K2-7874.
- *PMUEVENT bus* on page K2-7878.
- *Recommended authentication interface* on page K2-7879.
- *Management registers and CoreSight compliance* on page K2-7881.

Note

This recommended external debug interface specification is not part of the Arm architecture specification. Implementers and users of the Armv8 architecture must not consider this appendix as a requirement of the architecture. It is included as an appendix to this manual only:

- As reference material for users of Arm products that implement this interface.
- As an example of how an external debug interface might be implemented.

The inclusion of this appendix is no indication of whether any Arm products might, or might not, implement this external debug interface. For details of the implemented external debug interface you must always see the appropriate product documentation.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K2.1 About the recommended external debug interface

See the *Note* on the first page of this appendix for information about the architectural status of this recommended debug interface.

This specification provides a recommended external debug interface for Armv8 to define a standard set of connections for validation environments. In general, the connection between components, such as between the PE and Trace extension, is not described here, although the table does include the signals for the CTI connection. Table K2-1 shows the signals in the recommended interface.

Table K2-1 Recommended debug interface signals

Name	Direction	Description	Notes
DBGEN	In	External debug enable	-
SPIDEN	In	Secure privileged external debug enable	-
		Secure privileged self-hosted debug enable	Only in Secure AArch32 modes when enabled by MDCR_EL3.SPD32
NIDEN	In	External profiling and trace enable	If ARMv8.4-Debug is implemented, this signal is not implemented.
SPNIDEN	In	Secure external profiling and trace enable	If ARMv8.4-Debug is implemented, this signal is not implemented.
EDBGRQ	In	External halt request	IMPLEMENTATION DEFINED mechanism to halt the PE. See *EDBGRQ and DBGACK* on page K2-7877.
DBGACK	Out	Debug Acknowledge	Indicate to the system that a PE is in Debug state. See *EDBGRQ and DBGACK* on page K2-7877.
COMMIRQ	Out	DCC interrupt	Interface to an interrupt controller. See *Interrupt-driven use of the DCC* on page H4-7008 and the pseudocode for function `CheckForDCCInterrupts()`.
PMUIRQ	Out	Performance Monitor overflow	Interface to an interrupt controller. See *Behavior on overflow* on page D7-2705.
COMMRX	Out	DTRRX is full	Provided for legacy connection to an interrupt controller only. See *Interrupt-driven use of the DCC* on page H4-7008 and the pseudocode for function `CheckForDCCInterrupts()`.
COMMTX	Out	DTRTX is empty	
PMUEVENT[n:0]	Out	Performance Monitors event bus	See *PMUEVENT bus* on page K2-7878.
DBGNOPWRDWN	Out	Emulate low-power state request	Interface to a power controller. See *Emulating low-power states* on page H6-7033.
DBGPWRUPREQ	Out	Core powerup request	Interface to a power controller. See *Powerup request mechanism* on page H6-7032.
DBGRSTREQ	Out	Warm reset request	Interface to a power controller. See EDPRCR.CWRR.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K2-1 Recommended debug interface signals (continued)

Name	Direction	Description	Notes
DBGBUSCANCELREQ	Out	Allow asynchronous entry to Debug state	Extension to the bus interface. See EDRCR.CBRRQ.
DBGPWRDUP	In	Core powerup status	Interface to a power controller. See EDPRSR.PU.
DBGROMADDR[n:12]	In	MDRAR_EL1.ROMADDR	*n* depends on the size of the physical address space.
DBGROMADDRV	In	MDRAR_EL1.Valid	-
PRESETDBG	In	External debug reset	-
CPUPORESET	In	Cold reset	-
CORERESET	In	Warm reset	-
PSELDBG	In	Debug APB slave port	For details see *AMBA APB3*. Arm recommends a single slave port for all integrated debug components. **PADDRDBG31** distinguishes memory-mapped and Debug Access Port accesses: **0** Memory-mapped access **1** Debug Access Port access If ARMv8.4-Debug is implemented, **PPROTDBG[1]** distinguishes between Secure and Non-secure accesses.
PENABLEDBG	In		
PWRITEDBG	In		
PRDATADBG[31:0]	Out		
PWDATADBG[31:0]	In		
PADDRDBG[n:2][a]	In		
PREADYDBG	Out		
PSLVERRDBG	Out		
PCLKDBG	In		
PCLKENDBG	In		
PPROTDBG[1]	In		
CTICHIN	In	CoreSight channel interface	For details, see the *Arm® CoreSight™ Architecture Specification*. The ACK signals are not required if the channel interface is synchronous.
CTICHOUTACK	In		
CTICHOUT	Out		
CTICHINACK	Out		
CTIIRQ	Out	CTI interrupt, see *Description and allocation of CTI triggers* on page H5-7019	Implements a handshake for an edge-sensitive interrupt.
CTIIRQACK	In		

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K2-1 Recommended debug interface signals (continued)

Name	Direction	Description	Notes
ATDATA[nx8-1:0]	Out	AMBA 4 ATB interface	For details, see the *AMBA 4 ATB Protocol Specification, ATBv1.0 and ATBv1.1*. Only available if the OPTIONAL Trace extension is implemented.
ATBYTES[n-1:0]	Out		
ATID[6:0]	Out		
ATREADY	In		
ATVALID	Out		
AFREADY	Out		
AFVALID	Out		
SYNCREQ	In		
ATCLK	In		
ATCLKEN	In		
ATRESET	In		

a. The value of *n* depends on the size of the address space occupied by the Debug port.

Figure K2-1 shows the recommended debug interface.

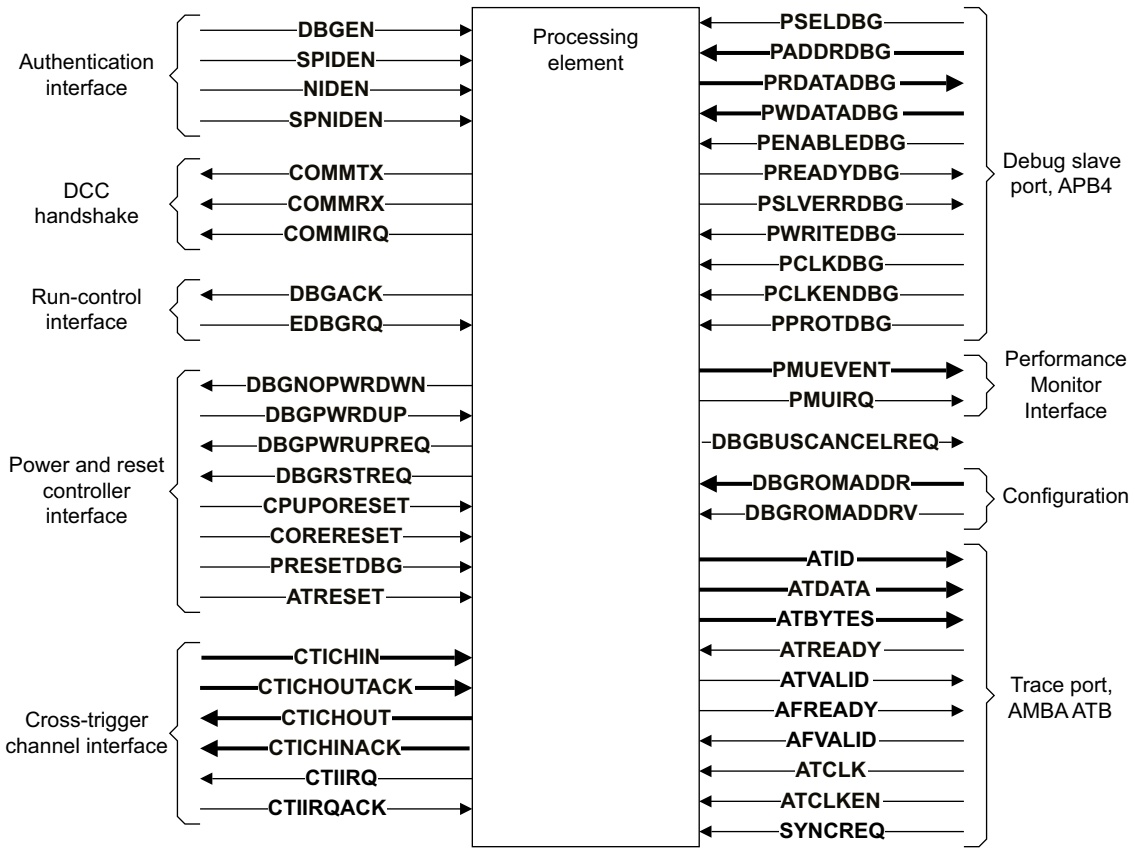


Figure K2-1 Recommended external debug interface, including the APB3 slave port

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K2.1.1 EDBGRQ and DBGACK

EDBGRQ is an IMPLEMENTATION DEFINED means of generating the External Debug Request debug event described in *External Debug Request debug event* on page H3-6983.

The PE asserts **DBGACK** when the PE is in Debug state. The PE might also include variants of this signal:

DBGTRIGGER

Asserted by the PE when it commits to entering Debug state.

DBGCPUDONE

Asserted by the PE when it has completed all Non-debug state memory accesses and Debug state entry is complete. **DBGCPUDONE** indicates that memory accesses issued by the PE result from operations originating from debugger commands.

In previous architecture versions, these signals provide an interface between the PE and cross-trigger logic. In Armv8, the architectural Cross-Trigger Interface provides this functionality for external debuggers.

K2.1.2 Secure and Non-secure views of the debug registers

If ARMv8.4-Debug is implemented, the external debug interface has views of Secure and Non-secure debug registers. The DAP is must ensure that accesses are made only when permitted. The Arm debug interface describes a standard APB-AP programmers model for APB4 which signals Secure and Non-secure accesses on the **PPROTDBG[1]** signal, and is recommended for new designs.

If ARMv8.4-Debug is implemented, and an APB-AP implements an APB3 master port, which does not support Secure and Non-secure views, Arm recommends that the following is implemented:

- If **SPIDEN** is HIGH and **DBGEN** is HIGH, all external debug accesses area treated as Secure.
- If either **SPIDEN** is LOW or **DBGEN** is LOW, all external debug accesses are treated as Non-secure.

If the PE APB slave port is APB4, this might be implemented by, for example, fixing **PPROTDBG[1]** to the inverse of (**SPIDEN** & **DBGEN**) when bridging from APB3 to APB4.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K2.2 PMUEVENT bus

The **PMUEVENT** bus exports Performance Monitor events from the PE to an on-chip agent. Arm recommends that it has the following characteristics:

- The bus is synchronous.
- The width of the bus is IMPLEMENTATION DEFINED.
- It is IMPLEMENTATION DEFINED which events are exported on the bus.
- Each exported event occupies a contiguous sub-field of the bus. Arm recommends that the sub-fields of the bus are occupied in the same order as the event numbers.
- If the event can only occur once per cycle, it occupies a single bit. If the event can occur more than once per cycle, it is IMPLEMENTATION DEFINED how the event is encoded. The encoding depends on constraints such as the designated use of the event bus and the number of pins available. For example, the event can be encoded:
  - As a count, using a plain binary number. This is the most useful encoding when exporting to an external counter. It is not a useful encoding for exporting to a Trace extension external input.
  - As a count, using thermometer encoding. This is the most useful encoding when exporting to a Trace extension.
  - Using a single bit encoding to indicate whether the event count is zero or nonzero. This is useful for exporting to an activity monitor where the number of pins is constrained.

If a Trace extension is implemented, the **PMUEVENT** bus is normally connected to the Trace extension using the external inputs. TRCEXTINSELR multiplexes a wide **PMUEVENT** bus to a narrow set of inputs. An external **PMUEVENT** bus might also be provided. For more information, contact Arm.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K2.3 Recommended authentication interface

An implementation of the Armv8 architecture must support debug authentication described in *Required debug authentication* on page H1-6924.

The details of the debug authentication interface are IMPLEMENTATION DEFINED, but Arm recommends the use of the CoreSight interface, which includes the following signals for external debug authentication:

- **DBGEN**.
- **SPIDEN**.

If ARMv8.4-Debug is not implemented, Arm also recommends using the following signals:

- **NIDEN**.
- **SPNIDEN**.

Arm recommends an interface in which **DBGEN** and **SPIDEN** are also used for self-hosted Secure debug authentication if either:

- EL3 is using AArch32 and SDCR.SPD == `0b00`.
- Secure EL1 is using AArch32 and MDCR_EL3.SPD32 == `0b00`.

If EL3 is not implemented and the PE is in Non-secure state, **SPIDEN** and **SPNIDEN** are not implemented, and the PE behaves as if these signals were tied LOW.

If EL3 is not implemented and the PE is in Secure state, **SPIDEN** is usually connected to **DBGEN** and **SPNIDEN** is connected to **NIDEN**, but this is not required. The recommended interface is defined as if all four signals are implemented.

How the authentication signals are driven is IMPLEMENTATION DEFINED. For example, the signals might be hard-wired, connected to fuses, or to an authentication module. The architecture permits PEs within a cluster to have independent authentication interfaces, but this is not required. Arm recommends that any Trace extension has the same authentication interface as the PE it is connected to.

If ARMv8.4-Debug and CoreSight ETR are both implemented, the ETR has an independent **DBGEN** signal that must be tied HIGH to enable self-hosted use of trace.

Table K2-2 shows the debug authentication pseudocode functions and the recommended implementations.

Table K2-2 Recommended implementation of debug enable pseudocode functions

Pseudocode function	Description	Implementation
`AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()`	Secure invasive self-hosted debug enabled in AArch32 state (legacy)	(**DBGEN** AND **SPIDEN**)
`ExternalSecureNoninvasiveDebugEnabled()`[a]	Secure non-invasive debug enabled	(**DBGEN** OR **NIDEN**[b]) AND (**SPIDEN** OR **SPNIDEN**[c])
`ExternalSecureInvasiveDebugEnabled()`	Secure invasive debug enabled	(**DBGEN** AND **SPIDEN**)
`ExternalNoninvasiveDebugEnabled()`[d]	Non-secure non-invasive debug enabled	(**DBGEN** OR **NIDEN**[b])
`ExternalInvasiveDebugEnabled()`	Non-secure invasive debug enabled	**DBGEN**

a. If ARMv8.4-Debug is implemented, `ExternalSecureNoninvasiveDebugEnabled()` == `ExternalSecureInvasiveDebugEnabled()`.
b. If ARMv8.4-Debug is implemented, the **NIDEN** signal is not implemented.
c. If ARMv8.4-Debug is implemented, the **SPNIDEN** signal is not implemented.
d. If ARMv8.4-Debug is implemented, `ExternalNoninvasiveDebugEnabled()` == TRUE.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The `Debug_authentication()` pseudocode function on *shared/debug* on page J1-7705 defines the authentication signals DBGEN, SPIDEN, NIDEN and SPNIDEN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K2.4 Management registers and CoreSight compliance

The CoreSight architecture requires the implementation of a set of management registers that occupy the memory map from `0xF00` upwards in each of the debug components.

CoreSight compliance and complete implementation of the management registers is OPTIONAL, but Arm recommends that the registers are implemented.

The CoreSight architecture specification recommends that any integration test registers are implemented starting from `0xEFC` downwards. Each of the debug components has an IMPLEMENTATION DEFINED region from `0xE80` to `0xEFC` for this purpose.

K2.4.1 CoreSight interface register map

Table K2-3 shows the external management register maps for the following registers:

ED These are the external debug register.

CTI These are the Cross-trigger interface registers.

PMU These are the Performance Monitors registers.

Table K2-3 CoreSight interface register map

Offset	Mnemonic			Name
	ED	CTI	PMU	
0xF00	EDITCTRL	CTIITCTRL	PMITCTRL	Integration Model Control registers
0xF04-0xF9C	-	-	-	Reserved, RES0
0xFA0	DBGCLAIMSET_EL1[a]	CTICLAIMSET[b]	-	CLAIM Tag Set registers
0xFA4	DBGCLAIMCLR_EL1[a]	CTICLAIMCLR[b]	-	CLAIM Tag Clear registers
0xFA8	EDDEVAFF0[a]	CTIDEVAFF0[c]	PMDEVAFF0	Device Affinity registers
0xFAC	EDDEVAFF1[a]	CTIDEVAFF1[c]	PMDEVAFF1	
0xFB0	EDLAR[d]	CTILAR[d]	PMLAR[d]	Lock Access register
0xFB4	EDLSR[d]	CTILSR[d]	PMLSR[d]	Lock Status register
0XFB8	DBGAUTHSTATUS_EL1[a]	CTIAUTHSTATUS	PMAUTHSTATUS	Authentication Status register
0xFBC	EDDEVARCH	CTIDEVARCH	PMDEVARCH	Device Architecture register
0xFC0	EDDEVID2[a]	CTIDEVID2[a]	-	Device ID register
0xFC4	EDDEVID1[a]	CTIDEVID1[a]	-	
0xFC8	EDDEVID[a]	CTIDEVID[a]	PMDEVID[a, e]	
0xFCC	EDDEVTYPE	CTIDEVTYPE	PMDEVTYPE	Device Type register
0xFD0	EDPIDR4	CTIPIDR4	PMPIDR4	Peripheral ID registers
0xFD4-0xFDC	-	-	-	Reserved, RES0
0xFE0	EDPIDR0	CTIPIDR0	PMPIDR0	Peripheral ID registers
0xFE4	EDPIDR1	CTIPIDR1	PMPIDR1	
0xFE8	EDPIDR2	CTIPIDR2	PMPIDR2	
0xFEC	EDPIDR3	CTIPIDR3	PMPIDR3	

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K2-3 CoreSight interface register map (continued)

Offset	Mnemonic			Name
	ED	CTI	PMU	
0xFF0	EDCIDR0	CTICIDR0	PMCIDR0	Component ID registers
0xFF4	EDCIDR1	CTICIDR1	PMCIDR1	
0xFF8	EDCIDR2	CTICIDR2	PMCIDR2	
0xFFC	EDCIDR3	CTICIDR3	PMCIDR3	

a. This register must always be implemented, regardless of whether the component is CoreSight compliant.
b. If implemented, the number of CLAIM bits is IMPLEMENTATION DEFINED and can be discovered by reading CLAIMSET.
c. If the CTI implements CTIv1, this register is not implemented. See the register description for details.
d. The Software lock registers are defined as part of CoreSight compliance, but their contents depend on the type of access that is made and whether the OPTIONAL Software lock is implemented. See the register description for details.
e. PMDEVID is implemented only from Armv8.2 or if ARMv8.2-PCSample is implemented, otherwise its offset is RES0.

K2.4.2 Management register access permissions

Access to the OPTIONAL Integration Control register (ITCTRL) is IMPLEMENTATION DEFINED.

Table K2-4 on page K2-7883, Table K2-5 on page K2-7884, Table K2-6 on page K2-7885, Table K2-7 on page K2-7886, and Table K2-8 on page K2-7887 show the response to accesses by the external debug interface to the CoreSight management registers.

Note

Access to the CoreSight management registers is not affected by the values of EDAD and EPMAD.

Table K2-4 on page K2-7883, Table K2-5 on page K2-7884, Table K2-6 on page K2-7885, Table K2-7 on page K2-7886, and Table K2-8 on page K2-7887 include reserved management registers, because the CoreSight architecture requires that these registers are always RES0. The descriptions in *Reserved and unallocated registers* on page H8-7059 do not apply to reserved management registers if the implementation is CoreSight compliant.

If OPTIONAL memory-mapped access to the external debug interface is supported, there are additional constraints on memory-mapped accesses. See *Register access permissions for memory-mapped accesses* on page H8-7054.

When `HaveSecureExtDebugView`() == TRUE, each debug component has a Secure and Non-secure view. The Secure view of a debug component is mapped into Secure physical memory and the Non-secure view of a debug component is mapped into Non-secure memory. Apart from access conditions, the Non-secure and Secure views of the debug components are identical.

The terms in Table K2-4 on page K2-7883, Table K2-5 on page K2-7884, Table K2-6 on page K2-7885, Table K2-7 on page K2-7886, and Table K2-8 on page K2-7887 are defined as follows:

Domain This describes the power domain in which the register is logically implemented. Registers described as implemented in the Core power domain might be implemented in the Debug power domain, as long as they exhibit the required behavior.

If ARMv8.3-DoPD is implemented, most External debug interface registers are in the Core power domain, as shown in Table K2-4 on page K2-7883 and Table K2-7 on page K2-7886.

If ARMv8.3-DoPD is not implemented, most of the registers are in the Debug Power Domain, as shown in Table K2-5 on page K2-7884 and Table K2-8 on page K2-7887.

Conditions This lists the conditions under which the access is attempted.

To determine the access permissions for a register, read these columns from left to right, and stop at first column that lists the condition as being true.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The conditions are:

Off EDPRSR.PU == 0. The Core power domain is completely off, or in low-power state. In these cases, the Core power domain registers cannot be accessed.

Note

When the Debug power domain is off, all accesses to the registers in the external Debug power domain return an error.

DLK If the OS Double Lock is implemented and `DoubleLockStatus()` == TRUE. The OS Double Lock is locked.

OSLK OSLSR.OSLK == 1. The OS Lock is locked.

Default This provides the default access permissions, if there are no conditions that prevent access to the register.

SLK This provides the modified default access permissions for OPTIONAL memory-mapped accesses to the external debug interface if the OPTIONAL Software Lock is locked. See *Register access permissions for memory-mapped accesses* on page H8-7054. If ARMv8.3-DoPD is implemented, the Software Lock is not locked, or not implemented, this column is ignored.

The access permissions are:

- This means that the default access permission applies. See the Default column, or the SLK column, if applicable.

RO This means that the register or field is read-only.

RW This means that the register or field is read/write. Individual fields within the register might be RO. See the relevant register description for details.

RC This means that the bit clears to 0 after a read.

(SE) This means that accesses to this register have indirect write side effects. A side effect occurs when a direct read or a direct write of a register creates an indirect write to the same register or to another register.

WO This means that the register or field is write-only.

WI This means that the register or field ignores writes.

IMP DEF This means that the access permissions are IMPLEMENTATION DEFINED.

Table K2-4 External debug interface access permissions, CoreSight registers (debug) if ARMv8.3-DoPD is implemented

Offset	Register	Domain	Conditions (priority left to right)			Default
			Off	DLK	OSLK	
0xF00	EDITCTRL	IMP DEF	IMPLEMENTATION DEFINED			IMP DEF
0xF04-0xF8C	Reserved		-	-	-	RES0
0xFA0	DBGCLAIMSET_EL1	Core	Error	Error	Error	RW (SE)
0xFA4	DBGCLAIMCLR_EL1	Core	Error	Error	Error	RW (SE)
0xFA8	EDDEVAFF0	Core	Error	-	-	RO
0xFAC	EDDEVAFF1	Core	Error	-	-	RO
0xFB0	EDLAR	Core	Error	-	-	WO (SE)

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K2-4 External debug interface access permissions, CoreSight registers (debug) if ARMv8.3-DoPD is implemented (continued)

Offset	Register	Domain	Conditions (priority left to right)			Default
			Off	DLK	OSLK	
0xFB4	EDLSR	Core	Error	-	-	RO
0xFB8	DBGAUTHSTATUS_EL1	Core	Error	-	-	RO
0xFBC	EDDEVARCH	Core	Error	-	-	RO
0xFC0	EDDEVID2	Core	Error	-	-	RO
0xFC4	EDDEVID1	Core	Error	-	-	RO
0xFC8	EDDEVID	Core	Error	-	-	RO
0xFCC	EDDEVTYPE	Core	Error	-	-	RO
0xFD0	EDPIDR4	Core	Error	-	-	RO
0xFD4-0xFDC	Reserved		-	-	-	RES0
0xFE0-0xFEC	EDPIDR0	Core	Error	-	-	RO
0xFE4	EDPIDR1	Core	Error	-	-	RO
0xFE8	EDPIDR2	Core	Error	-	-	RO
0xFEC	EDPIDR3	Core	Error	-	-	RO
0xFF0	EDCIDR0	Core	Error	-	-	RO
0xFF4	EDCIDR1	Core	Error	-	-	RO
0xFF8	EDCIDR2	Core	Error	-	-	RO
0xFFC	EDCIDR3	Core	Error	-	-	RO

Table K2-5 External debug interface access permissions, CoreSight registers (debug) if ARMv8.3-DoPD is not implemented

Offset	Register	Domain	Conditions (priority left to right)			Default	SLK
			Off	DLK	OSLK		
0xF00	EDITCTRL	IMP DEF	IMPLEMENTATION DEFINED			IMP DEF	RO/WI
0xF04-0xF8C	Reserved	Debug	-	-	-	RES0	-
0xFA0	DBGCLAIMSET_EL1	Core	Error	Error	Error	RW (SE)	RO
0xFA4	DBGCLAIMCLR_EL1	Core	Error	Error	Error	RW (SE)	RO
0xFA8	EDDEVAFF0	Debug	-	-	-	RO	-
0xFAC	EDDEVAFF1	Debug	-	-	-	RO	-
0xFB0	EDLAR	Debug	-	-	-	WO (SE)	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K2-5 External debug interface access permissions, CoreSight registers (debug) if ARMv8.3-DoPD is not implemented (continued)

Offset	Register	Domain	Conditions (priority left to right)			Default	SLK
			Off	DLK	OSLK		
0xFB4	EDLSR	Debug	-	-	-	RO	-
0xFB8	DBGAUTHSTATUS_EL1	Debug	-	-	-	RO	-
0xFBC	EDDEVARCH	Debug	-	-	-	RO	-
0xFC0	EDDEVID2	Debug	-	-	-	RO	-
0xFC4	EDDEVID1	Debug	-	-	-	RO	-
0xFC8	EDDEVID	Debug	-	-	-	RO	-
0xFCC	EDDEVTYPE	Debug	-	-	-	RO	-
0xFD0	EDPIDR4	Debug	-	-	-	RO	-
0xFD4-0xFDC	Reserved	Debug	-	-	-	RES0	-
0xFE0-0xFEC	EDPIDR0	Debug	-	-	-	RO	-
0xFE4	EDPIDR1	Debug	-	-	-	RO	-
0xFE8	EDPIDR2	Debug	-	-	-	RO	-
0xFEC	EDPIDR3	Debug	-	-	-	RO	-
0xFF0	EDCIDR0	Debug	-	-	-	RO	-
0xFF4	EDCIDR1	Debug	-	-	-	RO	-
0xFF8	EDCIDR2	Debug	-	-	-	RO	-
0xFFC	EDCIDR3	Debug	-	-	-	RO	-

Table K2-6 External debug interface access permissions, CoreSight registers (CTI)

Offset	Register	Domain	Conditions (priority left to right)			Default	SLK
			Off	DLK	OSLK		
0xF00	CTIITCTRL	IMP DEF	IMPLEMENTATION DEFINED			IMP DEF	RO/WI
0xF04-0xF8C	Reserved	Debug	-	-	-	RES0	-
0xFA0	CTICLAIMSET	Debug	-	-	-	RW (SE)	RO
0xFA4	CTICLAIMCLR	Debug	-	-	-	RW (SE)	RO
0xFA8	CTIDEVAFF0	Debug	-	-	-	RO	-
0xFAC	CTIDEVAFF1	Debug	-	-	-	RO	-
0xFB0	CTILAR	Debug	-	-	-	WO (SE)	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K2-6 External debug interface access permissions, CoreSight registers (CTI) (continued)

Offset	Register	Domain	Conditions (priority left to right)			Default	SLK
			Off	DLK	OSLK		
0xFB4	CTILSR	Debug	-	-	-	RO	-
0xFB8	CTIAUTHSTATUS	Debug	-	-	-	RO	-
0xFBC	CTIDEVARCH	Debug	-	-	-	RO	-
0xFC0	CTIDEVID2	Debug	-	-	-	RO	-
0xFC4	CTIDEVID1	Debug	-	-	-	RO	-
0xFC8	CTIDEVID	Debug	-	-	-	RO	-
0xFCC	CTIDEVTYPE	Debug	-	-	-	RO	-
0xFD0	CTIPIDR4	Debug	-	-	-	RO	-
0xFD4-0xFDC	Reserved	Debug	-	-	-	RES0	-
0xFE0	CTIPIDR0	Debug	-	-	-	RO	-
0xFE4	CTIPIDR1	Debug	-	-	-	RO	-
0xFE8	CTIPIDR2	Debug	-	-	-	RO	-
0xFEC	CTIPIDR3	Debug	-	-	-	RO	-
0xFF0	CTICIDR0	Debug	-	-	-	RO	-
0xFF4	CTICIDR1	Debug	-	-	-	RO	-
0xFF8	CTICIDR2	Debug	-	-	-	RO	-
0xFFC	CTICIDR3	Debug	-	-	-	RO	-

Table K2-7 External debug interface access permissions, CoreSight registers (PMU) if ARMv8.3-DoPD is implemented

Offset	Register	Domain	Conditions (priority left to right)			Default
			Off	DLK	OSLK	
0xF00	PMITCTRL	IMP DEF	IMPLEMENTATION DEFINED			IMP DEF
0xF04-0xFA4	Reserved		-	-	-	RES0
0xFA8	PMDEVAFF0	Core	Error	-	-	RO
0xFAC	PMDEVAFF1	Core	Error	-	-	RO
0xFB0	PMLAR	Core	Error	-	-	WO (SE)
0xFB4	PMLSR	Core	Error	-	-	RO
0xFB8	PMAUTHSTATUS	Core	Error	-	-	RO

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K2-7 External debug interface access permissions, CoreSight registers (PMU) if ARMv8.3-DoPD is implemented (continued)

Offset	Register	Domain	Conditions (priority left to right)			Default
			Off	DLK	OSLK	
0xFBC	PMDEVARCH	Core	Error	-	-	RO
0xFC0-0xFC4	Reserved		-	-	-	RES0
0xFC8	PMDEVID[a]	Core	Error	-	-	RO
0xFCC	PMDEVTYPE	Core	Error	-	-	RO
0xFD0	PMPIDR4	Core	Error	-	-	RO
0xFD4-0xFDC	Reserved		-	-	-	RES0
0xFE0	PMPIDR0	Core	Error	-	-	RO
0xFE4	PMPIDR1	Core	Error	-	-	RO
0xFE8	PMPIDR2	Core	Error	-	-	RO
0xFEC	PMPIDR3	Core	Error	-	-	RO
0xFF0	PMCIDR0	Core	Error	-	-	RO
0xFF4	PMCIDR1	Core	Error	-	-	RO
0xFF8	PMCIDR2	Core	Error	-	-	RO
0xFFC	PMCIDR3	Core	Error	-	-	RO

a. Implemented from Armv8.2, or if ARMv8.2-PCSample is implemented. Otherwise this location is RES0.

Table K2-8 External debug interface access permissions, CoreSight registers (PMU) if ARMv8.3-DoPD is not implemented

Offset	Register	Domain	Conditions (priority left to right)			Default	SLK
			Off	DLK	OSLK		
0xF00	PMITCTRL	IMP DEF	IMPLEMENTATION DEFINED			IMP DEF	RO/WI
0xF04-0xFA4	Reserved	Debug	-	-	-	RES0	-
0xFA8	PMDEVAFF0	Debug	-	-	-	RO	-
0xFAC	PMDEVAFF1	Debug	-	-	-	RO	-
0xFB0	PMLAR	Debug	-	-	-	WO (SE)	-
0xFB4	PMLSR	Debug	-	-	-	RO	-
0xFB8	PMAUTHSTATUS	Debug	-	-	-	RO	-
0xFBC	PMDEVARCH	Debug	-	-	-	RO	-
0xFC0-0xFC4	Reserved	Debug	-	-	-	RES0	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K2-8 External debug interface access permissions, CoreSight registers (PMU) if ARMv8.3-DoPD is not implemented (continued)

Offset	Register	Domain	Conditions (priority left to right)			Default	SLK
			Off	DLK	OSLK		
0xFC8	PMDEVID[a]	Debug	-	-	-	RO	-
0xFCC	PMDEVTYPE	Debug	-	-	-	RO	-
0xFD0	PMPIDR4	Debug	-	-	-	RO	-
0xFD4-0xFDC	Reserved	Debug	-	-	-	RES0	-
0xFE0	PMPIDR0	Debug	-	-	-	RO	-
0xFE4	PMPIDR1	Debug	-	-	-	RO	-
0xFE8	PMPIDR2	Debug	-	-	-	RO	-
0xFEC	PMPIDR3	Debug	-	-	-	RO	-
0xFF0	PMCIDR0	Debug	-	-	-	RO	-
0xFF4	PMCIDR1	Debug	-	-	-	RO	-
0xFF8	PMCIDR2	Debug	-	-	-	RO	-
0xFFC	PMCIDR3	Debug	-	-	-	RO	-

a. Implemented from Armv8.2, or if ARMv8.2-PCSample is implemented. Otherwise this location is RES0.

K2.4.3 Management register resets

Table K2-9 shows the management register resets. This table does not include:

- Read-only identification registers that have a fixed value from reset. These registers include those with the DEVAFFn, DEVARCH, DEVID{n}, DEVTYPE, PIDRn, and CIDRn mnemonics.
- Registers that have the AUTHSTATUS mnemonic. This is a read-only status register that reflects the status outside of the reset domain of the register.
- Registers that have the LAR mnemonic. These are write-only registers that only have an effect on writes.

All other fields in the management registers are reset to an IMPLEMENTATION DEFINED value which can be UNKNOWN. The registers are in the reset domain specified in the table.

Table K2-9 shows a summary of the management register resets.

Table K2-9 Management register resets

Register	Reset domain	Field	Value	Description
CTIITCTRL EDITCTRL PMITCTRL	IMPLEMENTATION DEFINED	IME	0	Integration mode enable

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K2-9 Management register resets (continued)

Register	Reset domain	Field	Value	Description
DBGCLAIMCLR_EL1	Cold reset	CLAIM	0x00	CLAIM tags
CTICLAIMCLR	External debug	CLAIM	0x00000000	
CTILSR[a] EDLSR[a] PMLSR[a]	If ARMv8.3-DoPD is implemented, reset by Cold reset, otherwise External debug.	SLK	1	Software Lock

a. Only if the OPTIONAL Software Lock is implemented

K2.4.4 About the Peripheral identification scheme

The Peripheral Identification scheme provides the standard information required by all components that conform to the *Arm® Debug Interface Architecture Specification, ADIv5.0 to ADIv5.2*, that implements the CoreSight identification scheme. They identify a peripheral in a particular namespace. For more information, see the *Arm® CoreSight™ Architecture Specification*.

Table K2-10 lists the Peripheral ID Registers that make up the Peripheral Identification scheme for each component.

Table K2-10 Peripheral Identification Registers

		Reference		
Register offset	**Description**	**External Debug**	**CTI**	**Performance Monitors**
0xFD0	Peripheral ID4	EDPIDR4	CTIPIDR4	PMPIDR4
0xFD4	Reserved for Peripheral ID5	-	-	-
0xFD8	Reserved for Peripheral ID6	-	-	-
0xFDC	Reserved for Peripheral ID7	-	-	-
0xFE0	Peripheral ID0	EDPIDR0	CTIPIDR0	PMPIDR0
0xFE4	Peripheral ID1	EDPIDR1	CTIPIDR1	PMPIDR1
0xFE8	Peripheral ID2	EDPIDR2	CTIPIDR2	PMPIDR2
0xFEC	Peripheral ID3	EDPIDR3	CTIPIDR3	PMPIDR3

Figure K2-2 shows the register field allocation scheme for the Peripheral ID Registers.

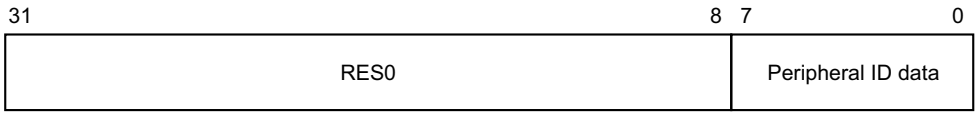


Figure K2-2 Peripheral ID register format

Software can consider the eight Peripheral ID Registers as defining a single 64-bit Peripheral ID, as shown in Figure K2-3 on page K2-7890.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

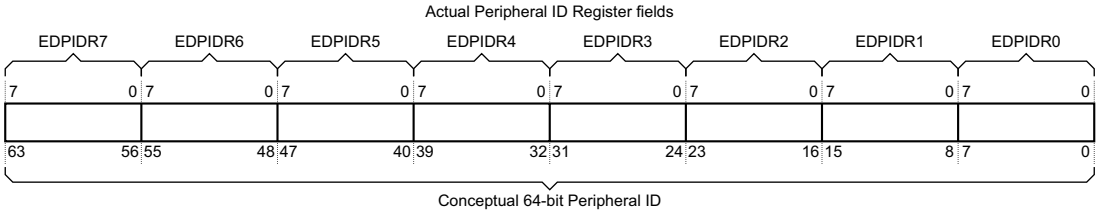


Figure K2-3 Mapping between Peripheral ID Registers and a 64-bit Peripheral ID Value

Figure K2-3 shows the fields in the 64-bit Peripheral ID value, and includes the field values for fields that:

- Have fixed values, including the bits that are reserved.
- Have fixed values in an implementation that is designed by Arm.

For more information about the fields and their values see Table K2-11.

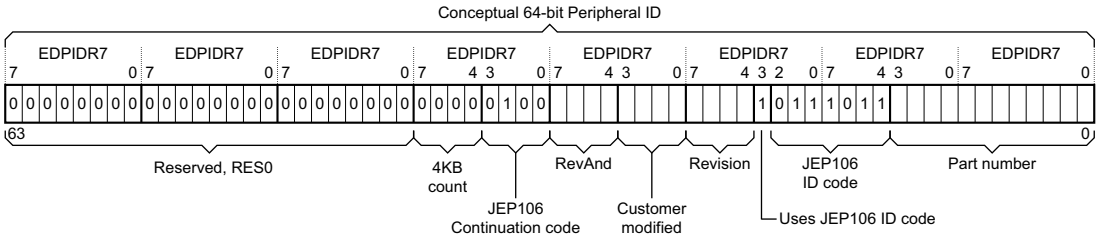


Figure K2-4 Peripheral ID fields, with values for a implementation designed by ARM

Table K2-11 shows the fields in the Peripheral ID.

Table K2-11 Fields in the Peripheral Identification Registers

Name	Size	Description	Registers
4KB count	4 bits	Log2 of the number of 4KB blocks occupied by the implementation.	EDPIDR4 CTIPIDR4 PMPIDR4
JEP106 code	4+7 bits	Identifies the designer of the implementation. This value consists of: • A 4-bit continuation code, also described as the bank number. • A 7-bit identification code. For implementations designed by Arm, the continuation code is 0x4, indicating bank 5, and the identity code is 0x3B.	EDPIDR1, EDPIDR2, EDPIDR4 CTIPIDR1, CTIPIDR2, CTIPIDR4 PMPIDR1, PMPIDR2, PMPIDR4
RevAnd	4 bits	Manufacturing revision number. Indicates a late modification to the implementation, usually as a result of an *Engineering Change Order* (ECO). This field starts at `0x0` and is incremented by the integrated circuit manufacturer on metal fixes.	EDPIDR3 CTIPIDR3 PMPIDR3
Customer modified	4 bits	Indicates an endorsed modification to the implementation. If the system designer cannot modify the implementation supplied by the implementation designer then this field is RES0.	EDPIDR3 CTIPIDR3 PMPIDR3

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K2-11 Fields in the Peripheral Identification Registers (continued)

Name	Size	Description	Registers
Revision	4 bits	Revision number for the implementation. Starts at `0x0` and increments by 1 at both major and minor revisions.	EDPIDR2 CTIPIDR2 PMPIDR2
Uses JEP106 ID code	1 bit	This bit is set to 1 when a JEP106 identification code is used. This bit must be 1 on all Armv8 implementations.	EDPIDR2 CTIPIDR2 PMPIDR2
Part number	12 bits	Part number for the implementation. Each organization designing to the Arm Debug architecture specification keeps its own part number list.	EDPIDR0, EDPIDR1 CTIPIDR0, CTIPIDR1 PMPIDR0, PMPIDR1

A component is identified uniquely by the combination of the following fields:

- JEP106 continuation code.
- JEP106 identity code.
- Part number.
- Revision.
- Customer Modified.
- RevAnd.

For components with a *Component class* of `0x9`, Debug component, indicated by the Component Identification Registers, multiple components can have the same Part number, provided each component has a different CoreSight *Device type*. However, Arm strongly recommends that each device has a unique Part number. For more information:

- About the Component Identification Registers, see *About the Component Identification scheme* on page K2-7892.
- About the CoreSight Device type, see EDDEVTYPE, CTIDEVTYPE, or PMDEVTYPE.
- About CoreSight components and their identification, see the *Arm® Debug Interface Architecture Specification*.

Allocating revisions and part numbers

Within the Peripheral Identification registers, the allocation of major and minor revisions, part numbers, and customer-modified fields is IMPLEMENTATION DEFINED, with the following set of restrictions so that:

- The REVISION field must increase monotonically with revisions.

  Note

  Arm recommends that the REVISION field is updated for each update to the RTL, regardless of whether this is a major or minor update.

- The REVAND field should increase monotonically with revisions.

  Note

  Arm recommends that the REVAND field is used only for post-release changes. For example, those due to engineering change order (ECO) fixes related to the debug component of the processor.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The PART field must have a degree of uniqueness:
  - Two component designs can have the same part number so long as they are sub-components of the same part and the programmers' model for the part has the means to disambiguate sub-components.
  - Otherwise, two component designs must have unique part numbers.

The DEVARCH (if implemented) or DEVTYPE (otherwise) register provides the means to disambiguate sub-components of the Debug Architecture.

A ROM table has no DEVTYPE or DEVARCH register. However, if it is the only CLASS `0x1` component in a processor cluster, it can still be disambiguated.

Multiple instances of the same component design have the same part number.

K2.4.5 About the Component Identification scheme

The Component Identification Registers identify the processor as an Arm Debug Interface v5 component. For more information, see the *Arm® Debug Interface Architecture Specification* and the *Arm® CoreSight™ Architecture Specification*.

The Component Identification Registers occupy the last four words of the 4KB block of debug registers.

Table K2-12 Component Identification Registers

Register offset	Description	External debug	CTI	Performance Monitors
`0xFF0`	Component ID0	EDCIDR0	CTICIDR0	PMCIDR0
`0xFF0`	Component ID1	EDCIDR1	CTICIDR1	PMCIDR1
`0xFF0`	Component ID2	EDCIDR2	CTICIDR2	PMCIDR2
`0xFF0`	Component ID3	EDCIDR3	CTICIDR3	PMCIDR3

Figure K2-5 shows the register field allocation scheme for the Component ID Registers.

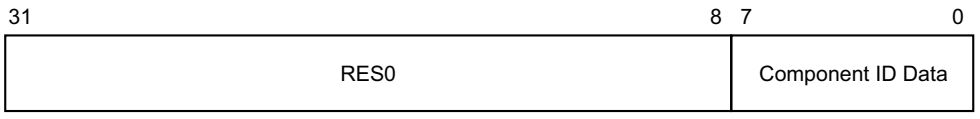


Figure K2-5 Component ID Register format

Software can consider the eight Component ID Registers as defining a single 32-bit Component ID, as shown in Figure K2-6.

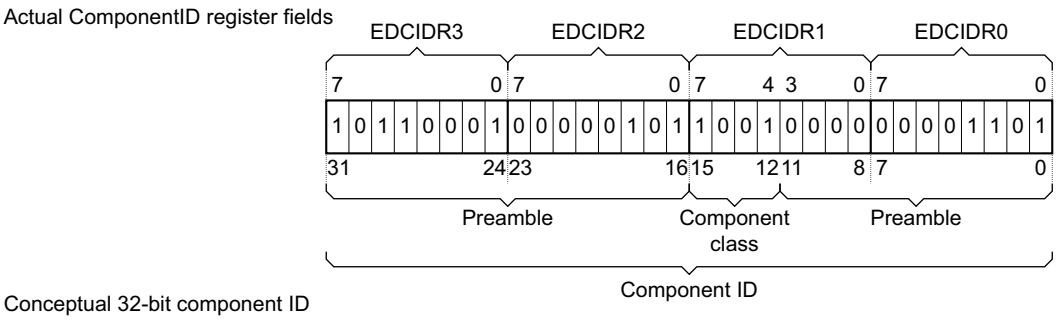


Figure K2-6 Mapping between Component ID Registers and a 32-bit Component ID Value

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Appendix K3
Recommendations for Performance Monitors Event Numbers for IMPLEMENTATION DEFINED Events

This appendix describes the Arm recommendations for the use of the IMPLEMENTATION DEFINED event numbers. It contains the following sections:

- *Arm recommendations for IMPLEMENTATION DEFINED event numbers* on page K3-7894.
- *Summary of events for exceptions taken to an Exception level using AArch64* on page K3-7909.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K3.1 Arm recommendations for IMPLEMENTATION DEFINED event numbers

These are the Arm recommendations for the use of the IMPLEMENTATION DEFINED event numbers. Arm does not define these events as rigorously as those in the architectural and microarchitectural event lists, and an implementation might:

- Modify the definition of an event to better correspond to the implementation.
- Not use some, or many, of these event numbers.

Table K3-1 lists the PMU IMPLEMENTATION DEFINED event numbers in event number order.

Table K3-1 PMU IMPLEMENTATION DEFINED event numbers

Event number	Event mnemonic	Description
0x0040	L1D_CACHE_RD	Attributable Level 1 data cache access, read
0x0041	L1D_CACHE_WR	Attributable Level 1 data cache access, write
0x0042	L1D_CACHE_REFILL_RD[a]	Attributable Level 1 data cache refill, read
0x0043	L1D_CACHE_REFILL_WR[a]	Attributable Level 1 data cache refill, write
0x0044	L1D_CACHE_REFILL_INNER	Attributable Level 1 data cache refill, inner
0x0045	L1D_CACHE_REFILL_OUTER	Attributable Level 1 data cache refill, outer
0x0046	L1D_CACHE_WB_VICTIM	Attributable Level 1 data cache Write-Back, victim
0x0047	L1D_CACHE_WB_CLEAN	Level 1 data cache Write-Back, cleaning and coherency
0x0048	L1D_CACHE_INVAL	Attributable Level 1 data cache invalidate
0x0049-0x004B	-	Reserved
0x004C	L1D_TLB_REFILL_RD[a]	Attributable Level 1 data TLB refill, read
0x004D	L1D_TLB_REFILL_WR[a]	Attributable Level 1 data TLB refill, write
0x004E	L1D_TLB_RD	Attributable Level 1 data or unified TLB access, read
0x004F	L1D_TLB_WR	Attributable Level 1 data or unified TLB access, write
0x0050	L2D_CACHE_RD	Attributable Level 2 data cache access, read
0x0051	L2D_CACHE_WR	Attributable Level 2 data cache access, write
0x0052	L2D_CACHE_REFILL_RD[a]	Attributable Level 2 data cache refill, read
0x0053	L2D_CACHE_REFILL_WR[a]	Attributable Level 2 data cache refill, write
0x0054-0x0055	-	Reserved
0x0056	L2D_CACHE_WB_VICTIM	Attributable Level 2 data cache Write-Back, victim
0x0057	L2D_CACHE_WB_CLEAN	Level 2 data cache Write-Back, cleaning and coherency
0x0058	L2D_CACHE_INVAL	Attributable Level 2 data cache invalidate
0x0059-0x005B	-	Reserved
0x005C	L2D_TLB_REFILL_RD[a]	Attributable Level 2 data or unified TLB refill, read
0x005D	L2D_TLB_REFILL_WR[a]	Attributable Level 2 data or unified TLB refill, write

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K3-1 PMU IMPLEMENTATION DEFINED event numbers (continued)

Event number	Event mnemonic	Description
0x005E	L2D_TLB_RD	Attributable Level 2 data or unified TLB access, read
0x005F	L2D_TLB_WR	Attributable Level 2 data or unified TLB access, write
0x0060	BUS_ACCESS_RD	Bus access, read
0x0061	BUS_ACCESS_WR	Bus access, write
0x0062	BUS_ACCESS_SHARED	Bus access, Normal, Cacheable, Shareable
0x0063	BUS_ACCESS_NOT_SHARED	Bus access, not Normal, Cacheable, Shareable
0x0064	BUS_ACCESS_NORMAL	Bus access, normal
0x0065	BUS_ACCESS_PERIPH	Bus access, peripheral
0x0066	MEM_ACCESS_RD	Data memory access, read
0x0067	MEM_ACCESS_WR	Data memory access, write
0x0068	UNALIGNED_LD_SPEC	Unaligned access, read
0x0069	UNALIGNED_ST_SPEC	Unaligned access, write
0x006A	UNALIGNED_LDST_SPEC	Unaligned access
0x006B	-	Reserved
0x006C	LDREX_SPEC	Exclusive operation speculatively executed, `LDREX` or `LDX`
0x006D	STREX_PASS_SPEC	Exclusive operation speculatively executed, `STREX` or `STX` pass
0x006E	STREX_FAIL_SPEC	Exclusive operation speculatively executed, `STREX` or `STX` fail
0x006F	STREX_SPEC	Exclusive operation speculatively executed, `STREX` or `STX`
0x0070	LD_SPEC	Operation speculatively executed, load
0x0071	ST_SPEC	Operation speculatively executed, store
0x0072	LDST_SPEC	Operation speculatively executed, load or store
0x0073	DP_SPEC	Operation speculatively executed, integer data processing
0x0074	ASE_SPEC	Operation speculatively executed, Advanced SIMD instruction
0x0075	VFP_SPEC	Operation speculatively executed, floating-point instruction
0x0076	PC_WRITE_SPEC	Operation speculatively executed, software change of the PC
0x0077	CRYPTO_SPEC	Operation speculatively executed, Cryptographic instruction
0x0078	BR_IMMED_SPEC	Branch speculatively executed, immediate branch
0x0079	BR_RETURN_SPEC	Branch speculatively executed, procedure return
0x007A	BR_INDIRECT_SPEC	Branch speculatively executed, indirect branch
0x007B	-	Reserved
0x007C	ISB_SPEC	Barrier speculatively executed, `ISB`
0x007D	DSB_SPEC	Barrier speculatively executed, `DSB`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K3-1 PMU IMPLEMENTATION DEFINED event numbers (continued)

Event number	Event mnemonic	Description
0x007E	DMB_SPEC	Barrier speculatively executed, DMB
0x007F	CSDB_SPEC	Barrier speculatively executed, CSDB
0x0080	-	Reserved
0x0081	EXC_UNDEF	Exception taken, Other synchronous
0x0082	EXC_SVC	Exception taken, Supervisor Call
0x0083	EXC_PABORT	Exception taken, Instruction Abort
0x0084	EXC_DABORT	Exception taken, Data Abort and SError
0x0085	-	Reserved
0x0086	EXC_IRQ	Exception taken, IRQ
0x0087	EXC_FIQ	Exception taken, FIQ
0x0088	EXC_SMC	Exception taken, Secure Monitor Call
0x0089	-	Reserved
0x008A	EXC_HVC	Exception taken, Hypervisor Call
0x008B	EXC_TRAP_PABORT	Exception taken, Instruction Abort not *Taken locally*[b]
0x008C	EXC_TRAP_DABORT	Exception taken, Data Abort or SError not *Taken locally*[b]
0x008D	EXC_TRAP_OTHER	Exception taken, Other traps not *Taken locally*[b]
0x008E	EXC_TRAP_IRQ	Exception taken, IRQ not *Taken locally*[b]
0x008F	EXC_TRAP_FIQ	Exception taken, FIQ not *Taken locally*[b]
0x0090	RC_LD_SPEC	Release consistency operation speculatively executed, Load-Acquire
0x0091	RC_ST_SPEC	Release consistency operation speculatively executed, Store-Release
0x0092-0x009F	-	Reserved
0x00A0	L3D_CACHE_RD	Attributable Level 3 data or unified cache access, read
0x00A1	L3D_CACHE_WR	Attributable Level 3 data or unified cache access, write
0x00A2	L3D_CACHE_REFILL_RD[a]	Attributable Level 3 data or unified cache refill, read
0x00A3	L3D_CACHE_REFILL_WR[a]	Attributable Level 3 data or unified cache refill, write
0x00A4-0x00A5	-	Reserved
0x00A6	L3D_CACHE_WB_VICTIM	Attributable Level 3 data or unified cache Write-Back, victim
0x00A7	L3D_CACHE_WB_CLEAN	Attributable Level 3 data or unified cache Write-Back, cache clean
0x00A8	L3D_CACHE_INVAL	Attributable Level 3 data or unified cache access, invalidate

a. For more information, see *Relationship between REFILL events and associated access events* on page K3-7907.
b. The Glossary defines the term *Taken locally*. See also *Exception levels* on page D1-2314 for more information.

0x0040, L1D_CACHE_RD, Attributable Level 1 data cache access, read

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This event is similar to Level 1 data cache access, L1D_CACHE, but the counter counts only memory-read operations that access at least the Level 1 data or unified cache.

If the cache is shared, only events Attributable to this PE are counted. If the cache is not shared, all events are counted.

See also:

- *Attributability* on page D7-2707.
- *Meaningful ratios between common microarchitectural events* on page D7-2769.

`0x0041`, **L1D_CACHE_WR, Attributable Level 1 data cache access, write**

This event is similar to Level 1 data cache access, L1D_CACHE, but the counter counts only memory-write operations that access at least the Level 1 data or unified cache.

The counter counts DC ZVA as a store instruction.

`0x0042`, **L1D_CACHE_REFILL_RD, Attributable Level 1 data cache refill, read**

This event is similar to Level 1 data cache refill, L1D_CACHE_REFILL, but the counter counts only memory-read operations that cause a refill of at least the Level 1 data or unified cache.

See also *Relationship between REFILL events and associated access events.* on page K3-7907.

`0x0043`, **L1D_CACHE_REFILL_WR, Attributable Level 1 data cache refill, write**

This event is similar to Level 1 data cache refill, L1D_CACHE_REFILL, but the counter counts only memory-write operations that cause a refill of at least the Level 1 data or unified cache.

The counter counts DC ZVA as a store instruction.

See also *Relationship between REFILL events and associated access events.* on page K3-7907.

`0x0044`, **L1D_CACHE_REFILL_INNER, Attributable Level 1 data cache refill, inner**

This event is similar to Level 1 data cache refill, L1D_CACHE_REFILL, but the counter counts only memory-read and memory-write operations that generate refills satisfied by transfer from another cache inside of the immediate cluster.

Note

The boundary between *inner* and *outer* is IMPLEMENTATION DEFINED, and it is not necessarily linked to other similar boundaries, such as the boundary between Inner Cacheable and Outer Cacheable or the boundary between Inner Shareable and Outer Shareable.

`0x0045`, **L1D_CACHE_REFILL_OUTER, Attributable Level 1 data cache refill, outer**

This event is similar to Level 1 data cache refill, L1D_CACHE_REFILL, but the counter counts only memory-read and memory-write operations that generate refills satisfied from outside of the immediate cluster.

`0x0046`, **L1D_CACHE_WB_VICTIM, Attributable Level 1 data cache Write-Back, victim**

This event is similar to Level 1 data cache Write-Back, L1D_CACHE_WB, but the counter counts only Write-Backs that are a result of the line being allocated for an access made by the PE.

If ARMv8.4-PMU is not implemented, Write-Backs caused by the execution of a cache maintenance instruction are not counted. If ARMv8.4-PMU is implemented, it is IMPLEMENTATION DEFINED whether Write-Backs caused by the execution of a cache maintenance instruction are counted.

It is IMPLEMENTATION DEFINED whether a write of a whole cache line that is not the result of the eviction of a line from the cache is counted. For example, this might occur if the PE detects streaming writes to memory and does not allocate lines to the cache, or as the result of a DC ZVA.

`0x0047`, **L1D_CACHE_WB_CLEAN, Level 1 data cache Write-Back, cleaning and coherency**

This event is similar to Attributable Level 1 data cache Write-Back, L1D_CACHE_WB, but the counter counts only Write-Backs that are a result of a coherency operation made by another PE or are caused by the execution of a cache maintenance instruction. Whether Write-Backs caused by the execution of a cache maintenance instruction are counted is IMPLEMENTATION DEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If a coherency request from a requestor outside the PE results in a Write-Back, it is an Unattributable event.

Note

The transfer of a dirty cache line from the Level 1 data cache of this PE to the Level 1 data cache of another PE due to a hardware coherency operation is not counted unless the dirty cache line is also written back to a Level 2 cache or memory.

`0x0048`, **L1D_CACHE_INVAL, Attributable Level 1 data cache invalidate**

The counter counts each invalidation of a cache line in the Level 1 data or unified cache.

The counter does not count events if a cache refill invalidates a line.

If ARMv8.4-PMU is not implemented, the counter does not count locally-executed cache maintenance instructions that operate by set/way. If ARMv8.4-PMU is implemented, it is IMPLEMENTATION DEFINED whether the counter counts locally-executed cache maintenance instructions that operate by set/way.

If a coherency request from a requestor outside the PE results in a Write-Back, it is an Unattributable event.

`0x004C`, **L1D_TLB_REFILL_RD, Attributable Level 1 data TLB refill, read**

This event is similar to Level 1 data TLB refill, L1D_TLB_REFILL, but the counter counts only memory-read operations that cause a data TLB refill of a least the Level 1 data or unified TLB.

See also *Relationship between REFILL events and associated access events.* on page K3-7907.

`0x004D`, **L1D_TLB_REFILL_WR, Attributable Level 1 data TLB refill, write**

This event is similar to Level 1 data TLB refill, L1D_TLB_REFILL, but the counter counts only memory-write operations that cause a data TLB refill of a least the Level 1 data or unified TLB.

The counter counts DC ZVA as a store instruction.

See also *Relationship between REFILL events and associated access events.* on page K3-7907.

`0x004E`, **L1D_TLB_RD, Attributable Level 1 data or unified TLB access, read**

This event is similar to Level 1 data or unified TLB access, L1D_TLB, but the counter counts only memory-read operations that cause a TLB access to at least the Level 1 data or unified TLB.

`0x004F`, **L1D_TLB_WR, Attributable Level 1 data or unified TLB access, write**

This event is similar to Level 1 data or unified TLB access, L1D_TLB, but the counter counts only memory-write operations that cause a TLB access to at least the Level 1 data or unified TLB.

`0x0050`, **L2D_CACHE_RD, Attributable Level 2 data cache access, read**

This event is similar to Attributable Level 2 data cache access, L2D_CACHE, but the counter counts only memory-read operations that access at least the Level 2 data or unified cache.

`0x0051`, **L2D_CACHE_WR, Attributable Level 2 data cache access, write**

This event is similar to Attributable Level 2 data cache access, L2D_CACHE, but the counter counts only memory-write operations that access at least the Level 2 data or unified cache.

The counter counts DC ZVA as a store instruction.

`0x0052`, **L2D_CACHE_REFILL_RD, Attributable Level 2 data cache refill, read**

This event is similar to Attributable Level 2 data cache refill, L2D_CACHE_REFILL, but the counter counts only memory-read operations that cause a refill of at least the Level 2 data or unified cache.

See also *Relationship between REFILL events and associated access events.* on page K3-7907.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0x0053`, L2D_CACHE_REFILL_WR, Attributable Level 2 data cache refill, write

This event is similar to Attributable Level 2 data cache refill, L2D_CACHE_REFILL, but the counter counts only memory-write operations that cause a refill of at least the Level 2 data or unified cache.

The counter counts DC ZVA as a store instruction.

See also *Relationship between REFILL events and associated access events.* on page K3-7907.

`0x0056`, L2D_CACHE_WB_VICTIM, Attributable Level 2 data cache Write-Back, victim

This event is similar to Attributable Level 2 data cache Write-Back, L2D_CACHE_WB, but the counter counts only Write-Backs that are a result of the line being allocated for an access made by the PE.

If ARMv8.4-PMU is not implemented, Write-Backs caused by the execution of a cache maintenance instruction are not counted. If ARMv8.4-PMU is implemented, it is IMPLEMENTATION DEFINED whether Write-Backs caused by the execution of a cache maintenance instruction are counted.

It is IMPLEMENTATION DEFINED whether a write of a whole cache line that is not the result of the eviction of a line from the cache is counted. For example, this might occur if the PE detects streaming writes to memory and does not allocate lines to the cache, or as the result of a DC ZVA.

`0x0057`, L2D_CACHE_WB_CLEAN, Level 2 data cache Write-Back, cleaning and coherency

This event is similar to Attributable Level 2 data cache Write-Back, L2D_CACHE_WB, but the counter counts only Write-Backs that are a result of a coherency operation made by another PE or are caused by the execution of a cache maintenance instruction. Whether Write-Backs caused by the execution of a cache maintenance instruction are counted as IMPLEMENTATION DEFINED.

Note

The transfer of a dirty cache line from the Level 2 data cache of this PE to the Level 2 data cache of another PE due to a hardware coherency operation is not counted unless the dirty cache line is also written back to a Level 3 cache or memory.

If a coherency request from a requestor outside the PE results in a Write-Back, it is an Unattributable event.

`0x0058`, L2D_CACHE_INVAL, Attributable Level 2 data cache invalidate

The counter counts each invalidation of a cache line in the Level 2 data or unified cache.

The counter does not count events if a cache refill invalidates a line.

If ARMv8.4-PMU is not implemented, the counter does not count locally-executed cache maintenance instructions that operate by set/way. If ARMv8.4-PMU is implemented, it is IMPLEMENTATION DEFINED whether the counter counts locally-executed cache maintenance instructions that operate by set/way.

Note

Software that uses this event must know whether the Level 2 data cache is shared with other PEs. This event does not follow the general rule of Level 2 data cache events of only counting events that directly affect this PE.

If a coherency request from a requestor outside the PE results in a Write-Back, it is an Unattributable event.

`0x005`C, L2D_TLB_REFILL_RD, Attributable Level 2 data or unified TLB refill, read

This event is similar to Attributable Level 2 data or unified TLB refill, L2D_TLB_REFILL, but the counter counts only Attributable memory read operations that cause a TLB refill of at least the Level 2 data or unified TLB. See also *Relationship between REFILL events and associated access events.* on page K3-7907.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0x005D`, **L2D_TLB_REFILL_WR, Attributable Level 2 data or unified TLB refill, write**

This event is similar to Attributable Level 2 data or unified TLB refill, L2D_TLB_REFILL, but the counter counts only Attributable memory write operations that cause a TLB refill of at least the Level 2 data or unified TLB. See also *Relationship between REFILL events and associated access events.* on page K3-7907.

`0x005E`, **L2D_TLB_RD, Attributable Level 2 data or unified TLB access, read**

This event is similar to Attributable Level 2 data or unified TLB access, L2D_TLB, but the counter counts only Attributable memory read operations that cause a TLB access to at least the Level 2 data or unified TLB.

`0x005F`, **L2D_TLB_WR, Attributable Level 2 data or unified TLB access, write**

This event is similar to Attributable Level 2 data or unified TLB access, L2D_TLB, but the counter counts only Attributable memory write operations that cause a TLB access to at least the Level 2 data or unified TLB.

`0x0060`, **BUS_ACCESS_RD, Bus access, read**

This event is similar to Bus access, BUS_ACCESS, but the counter counts only memory-read operations that access outside the boundary of the PE and its closely-coupled caches.

`0x0061`, **BUS_ACCESS_WR, Bus access, write**

This event is similar to Bus access, BUS_ACCESS, but the counter counts only memory-write operations that access outside the boundary of the PE and its closely-coupled caches.

`0x0062`, **BUS_ACCESS_SHARED, Bus access, Normal, Cacheable, Shareable**

This event is similar to Bus access, BUS_ACCESS, but the counter counts only memory-read and memory-write operations that make Normal, Cacheable, Shareable accesses outside the boundary of the PE and its closely-coupled caches.

Note

It is IMPLEMENTATION DEFINED how the PE translates the attributes from the translation table entry for a region to the attributes on the bus.

In particular, a region of memory designated as Normal, Cacheable, Inner Shareable, Not Outer Shareable by a translation table entry, might be marked as either shareable or Non-shareable at the boundary of the PE and its closely-coupled caches. This depends on where the IMPLEMENTATION DEFINED boundary lies, between Inner and Outer Shareable.

If the Inner Shareable extends beyond the PE boundary, and the bus indicates the distinction between Inner and Outer Shareable, then either is counted as shareable for the purposes of defining this event.

`0x0063`, **BUS_ACCESS_NOT_SHARED, Bus access, not Normal, Cacheable, Shareable**

This event is similar to Bus access, BUS_ACCESS, but the counter counts only memory-read and memory-write operations that make accesses outside the boundary of the PE and its closely-coupled caches that are not Normal, Cacheable, Shareable. For example, the counter counts accesses marked as:

- Normal, Cacheable, Non-shareable.
- Normal, Non-cacheable.
- Device.

Note

It is IMPLEMENTATION DEFINED, how the PE translates the attributes from the translation table entries for a region to the attributes on the bus.

In particular, a region of memory designated as Normal, Cacheable, Inner Shareable, Not Outer Shareable by a translation table entry, might be marked as either shareable or Non-shareable at the boundary of the PE and its closely-coupled caches. This depends on where the IMPLEMENTATION DEFINED boundary lies, between Inner and Outer Shareable.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the Inner Shareable extends beyond the PE boundary, and the bus indicates the distinction between Inner and Outer Shareable, then either is counted as shareable for the purposes of defining this event.

`0x0064`, **BUS_ACCESS_NORMAL, Bus access, normal**

This event is similar to Bus access, BUS_ACCESS, but the counter counts only memory-read and memory-write operations that make Normal accesses outside the boundary of the PE and its closely-coupled caches. For example, the counter counts Normal, Cacheable and Normal, Non-cacheable accesses but does not count Device accesses.

`0x0065`, **BUS_ACCESS_PERIPH, Bus access, peripheral**

This event is similar to Bus access, BUS_ACCESS, but the counter counts only memory-read and memory-write operations that make Device accesses outside the boundary of the PE and its closely-coupled caches.

`0x0066`, **MEM_ACCESS_RD, Data memory access, read**

This event is similar to Data memory access, MEM_ACCESS, but the counter counts only memory-read operations and SVE memory-read operations that the PE made.

`0x0067`, **MEM_ACCESS_WR, Data memory access, write**

This event is similar to Data memory access, MEM_ACCESS, but the counter counts only memory-write operations and SVE memory-write operations made by the PE.

`0x0068`, **UNALIGNED_LD_SPEC. Unaligned access, read**

This event is similar to Data memory access, MEM_ACCESS, but the counter counts only unaligned memory-read operations and unaligned memory-read SVE operations that the PE made. It also counts unaligned accesses if they are subsequently transposed into multiple aligned accesses.

`0x0069`, **UNALIGNED_ST_SPEC, Unaligned access, write**

This event is similar to Data memory access, MEM_ACCESS, but the counter counts only unaligned memory-write operations and unaligned SVE memory-write operations that the PE made. It also counts unaligned accesses if they are subsequently transposed into multiple aligned accesses.

`0x006A`, **UNALIGNED_LDST_SPEC, Unaligned access**

This event is similar to Data memory access, MEM_ACCESS, but the counter counts only unaligned memory-read operations, unaligned memory-write operations, unaligned memory-read SVE operations, and unaligned memory-write SVE operations that the PE made. It also counts unaligned accesses if they are subsequently transposed into multiple aligned accesses.

`0x006C`, **LDREX_SPEC, Exclusive operation speculatively executed, Load-Exclusive**

The counter counts Load-Exclusive instructions speculatively executed.

The definition of speculatively executed is IMPLEMENTATION DEFINED.

`0x006D`, **STREX_PASS_SPEC, Exclusive operation speculatively executed, Store-Exclusive pass**

The counter counts Store-Exclusive instructions speculatively executed that completed a write.

The definition of speculatively executed is IMPLEMENTATION DEFINED but must be the same as for the LDREX_SPEC event.

`0x006E`, **STREX_FAIL_SPEC, Exclusive operation speculatively executed, Store-Exclusive fail**

The counter counts Store-Exclusive instructions speculatively executed that fail to complete a write. It is within the IMPLEMENTATION DEFINED definition of speculatively executed whether this includes conditional instructions that fail the condition code check.

The definition of speculatively executed is IMPLEMENTATION DEFINED but must be the same as for the LDREX_SPEC event.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0x006F`, **STREX_SPEC, Exclusive operation speculatively executed, Store-Exclusive**

The counter counts Store-Exclusive instructions speculatively executed.

The definition of speculatively executed is IMPLEMENTATION DEFINED but it must be the same as for the LDREX_SPEC event.

Arm recommends that this event is implemented if it is not possible to implement the exclusive operation speculatively executed, Store-Exclusive pass, and exclusive operation speculatively executed, Store-Exclusive fail, events with the same degree of speculation as the LDREX_SPEC event.

`0x0070`, **LD_SPEC, Operation speculatively executed, load**

This event is similar to Operation speculatively executed, INST_SPEC, but the counter counts only operations due to memory-reading instructions and operations due to memory-reading SVE instructions, as defined by the LD_RETIRED event.

The definition of Speculatively executed is IMPLEMENTATION DEFINED but must be the same as for the INST_SPEC event.

`0x0071`, **ST_SPEC, Operation speculatively executed, store**

This event is similar to Operation speculatively executed, INST_SPEC, but the counter counts only operations due to memory-writing instructions and operations due to memory-writing SVE instructions, as defined by the ST_RETIRED event.

The counter counts DC ZVA as a store operation.

The definition of Speculatively executed is IMPLEMENTATION DEFINED but must be the same as for the INST_SPEC event.

`0x0072`, **LDST_SPEC, Operation speculatively executed, load or store**

This event is similar to Operation speculatively executed, INST_SPEC, but the counter counts only operations due to memory-reading instructions, operations due to memory-writing instructions, operations due to memory-reading SVE instructions and operations due to memory-writing SVE instructions as defined by the LD_RETIRED and ST_RETIRED events.

The definition of Speculatively executed is IMPLEMENTATION DEFINED but must be the same as for the INST_SPEC event.

`0x0073`, **DP_SPEC, Operation speculatively executed, integer data processing**

This event is similar to Operation speculatively executed, INST_SPEC, but counts only operations due to integer data-processing instructions. It counts the following operations that operate on the general-purpose registers:

- In AArch64 state, *Data processing - immediate* on page C3-226 and *Data processing - register* on page C3-231.
- In AArch32 state, *Data-processing instructions* on page F1-4040.

This includes `MOV` and `MVN` operations.

This event also counts the following miscellaneous instructions:

- In AArch64 state, *System register instructions* on page C3-204, *System instructions* on page C3-204, and *Hint instructions* on page C3-205.
- In AArch32 state, *PSTATE and banked register access instructions* on page F1-4048, *Banked register access instructions* on page F1-4048, *Miscellaneous instructions* on page F1-4053, other than `ISB` and preloads, and *System register access instructions* on page F1-4057, other than `LDC` and `STC` instructions.

If the preload instructions `PRFM, PLD`, `PLDW`, and `PLI`, do not count as memory-reading instructions then they must count as integer data-processing instructions.

If `ISB`s do not count as software change of the PC then they must count as integer data-processing instructions.

The definition of Speculatively executed is IMPLEMENTATION DEFINED, but must be the same as for the INST_SPEC event.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

It is IMPLEMENTATION DEFINED whether the following instructions are counted as integer data-processing operations, SIMD operations, or floating-point operations, but Arm recommends that the instructions are all counted as integer data-processing operations:

- For AArch64 state, from the A64 floating-point convert to integer class, operations that move a value between a general-purpose register and a SIMD and floating-point register without type conversion:
  - `FMOV` (general).
- For AArch64 state, from the SIMD Move group, operations that move a values between a general-purpose register and an element or elements in a SIMD and floating-point register:
  - `DUP` (general).
  - `SMOV`.
  - `UMOV`.
  - `INS` (general).
- For AArch32 state:
  - `VDUP` (general-purpose register) and all `VMOV` instructions that transfer data between a general-purpose register and a SIMD and floating-point register.
  - `VMRS`.
  - `VMSR`.

`0x0074`, ASE_SPEC, Operation speculatively executed, Advanced SIMD

This event is similar to Operation speculatively executed, INST_SPEC, but the counter counts only operations due to Advanced SIMD data-processing instructions, see:

- For AArch64 state, the SIMD operations listed in *Data processing - SIMD and floating-point* on page C3-239.
- For AArch32 state, *Advanced SIMD data-processing instructions* on page F1-4061.

This includes all operations that operate on the SIMD and floating-point registers, except those that are counted as:

- Integer data-processing operations.
- Floating-point data-processing operations.
- Memory-reading operations.
- Memory-writing operations.
- Cryptographic operations other than `PMULL`, in AArch64 state.
- `VMULL`, in AArch32 state.

Advanced SIMD scalar operations are counted as Advanced SIMD operations, including those which operate on floating-point values. In AArch64 state, `PMULL`, and in AArch32 state, `VMULL` are counted as Advanced SIMD operations.

The definition of Speculatively executed is IMPLEMENTATION DEFINED, but must be the same as for the INST_SPEC event.

`0x0075`, VFP_SPEC, Operation speculatively executed, floating-point

This event is similar to Operation speculatively executed, INST_SPEC, but the counter counts only operations due to floating-point data-processing instructions, see:

- In AArch64 state, only the scalar floating-point operations listed in *Data processing - SIMD and floating-point* on page C3-239.

  Note

  This event does not count the SIMD floating-point operations listed in *Data processing - SIMD and floating-point* on page C3-239.

- In AArch32 state, *Floating-point data-processing instructions* on page F1-4072.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This includes all operations that operate on the SIMD and floating-point registers as floating-point values, except for SIMD scalar operations and those that are counted as one of:

- Integer data processing.
- Memory-reading operations.
- Memory-writing operations.

The following instructions that take both an integer register and a floating-point register argument and perform a type conversion (to/from integer or to/from fixed-point), are counted as floating-point data-processing operations:

- In AArch64 state, `FCVT{<mode>}`, `UCVTF`, and `SCVTF`.
- In AArch32 state, `VCVT<mode>`(floating-point), `VCVT`, `VCVTT`, and `VCVTB`.

The definition of Speculatively executed is IMPLEMENTATION DEFINED, but must be the same as for the INST_SPEC event.

`0x0076`, PC_WRITE_SPEC, Operation speculatively executed, software change of the PC

This event is similar to Operation speculatively executed, INST_SPEC, but the counter counts only operations due to software changes of the PC. Defined by the instruction architecturally executed, condition code check pass, software change of the PC event, see *Common event numbers* on page D7-2722.

The definition of Speculatively executed is IMPLEMENTATION DEFINED but must be the same as for the INST_SPEC event.

See also PC_WRITE_RETIRED.

`0x0077`, CRYPTO_SPEC, Operation speculatively executed, Cryptographic instruction

This event is similar to Operation speculatively executed, INST_SPEC, but the counter counts only operations due to Cryptographic instructions, except `PMULL` and `VMULL`, see *The Cryptographic Extension* on page C3-261.

The definition of Speculatively executed is IMPLEMENTATION DEFINED but must be the same as for the INST_SPEC event.

`0x0078`, BR_IMMED_SPEC, Branch speculatively executed, immediate branch

The counter counts immediate branch instructions speculatively executed. Defined by the instruction architecturally executed, immediate branch event, see *Common event numbers* on page D7-2722.

The definition of Speculatively executed is IMPLEMENTATION DEFINED.

See also BR_IMMED_RETIRED.

`0x0079`, BR_RETURN_SPEC, Branch speculatively executed, procedure return

The counter counts procedure return instructions speculatively executed. Defined by the BR_RETURN_RETIRED event.

The definition of Speculatively executed is IMPLEMENTATION DEFINED.

See also BR_RETURN_RETIRED.

`0x007A`, BR_INDIRECT_SPEC, Branch speculatively executed, indirect branch

The counter counts indirect branch instructions speculatively executed. This includes software change of the PC other than exception-generating instructions and immediate branch instructions.

The definition of Speculatively executed is IMPLEMENTATION DEFINED.

`0x007C`, ISB_SPEC, Barrier speculatively executed, `ISB`

The counter counts Instruction Synchronization Barrier instructions speculatively executed, including CP15ISB.

The definition of Speculatively executed is IMPLEMENTATION DEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0x007D`, **DSB_SPEC, Barrier speculatively executed,** `DSB`

The counter counts data synchronization barrier instructions speculatively executed, including CP15DSB, SSBB and PSSBB.

The definition of Speculatively executed is IMPLEMENTATION DEFINED.

`0x007E`, **DMB_SPEC, Barrier speculatively executed,** `DMB`

The counter counts data memory barrier instructions speculatively executed, including CP15DSB. It does not include the implied barrier operations of load/store operations with release consistency semantics.

The definition of Speculatively executed is IMPLEMENTATION DEFINED.

`007F`, **CSDB_SPEC, Barrier speculatively executed, CSDB**

If ARMv8.5-PMU is implemented, the counter counts control speculation barrier instructions speculatively executed.

The definition of Speculatively executed is IMPLEMENTATION DEFINED.

`0x0081`, **EXC_UNDEF, Exception taken, other synchronous**

This event is similar to Exception taken, EXC_TAKEN, but the counter counts only those exceptions *Taken locally* that are not counted as:

- Exception taken, Supervisor Call (EXC_SVC).
- Exception taken, Secure Monitor Call (EXC_SMC).
- Exception taken, Hypervisor Call (EXC_HVC).
- Exception taken, Instruction Abort (EXC_PABORT).
- Exception taken, Data Abort or SError (EXC_DABORT).
- Exception taken, IRQ (EXC_IRQ).
- Exception taken, FIQ (EXC_FIQ).

`0x0082`, **EXC_SVC, Exception taken, Supervisor Call**

This event is similar to Exception taken, EXC_TAKEN, but the counter counts only Supervisor Call exceptions that are *Taken locally*.

`0x0083`, **EXC_PABORT, Exception taken, Instruction Abort**

This event is similar to Exception taken, EXC_TAKEN, but the counter counts only Instruction Abort exceptions that are *Taken locally*.

`0x0084`, **EXC_DABORT, Exception taken, Data Abort or SError**

This event is similar to Exception taken, EXC_TAKEN, but the counter counts only Data Abort or SError interrupt exceptions. The counter counts only exceptions *Taken locally*.

`0x0086`, **EXC_IRQ, Exception taken, IRQ**

This event is similar to Exception taken, EXC_TAKEN, but the counter counts only IRQ exceptions that are *Taken locally*, including Virtual IRQ exceptions.

`0x0087`, **EXC_FIQ, Exception taken, FIQ**

This event is similar to Exception taken, EXC_TAKEN, but the counter counts only FIQ exceptions that are *Taken locally*, including Virtual FIQ exceptions.

`0x0088`, **EXC_SMC, Exception taken, Secure Monitor Call**

This event is similar to Exception taken, EXC_TAKEN, but the counter counts only Secure Monitor Call exceptions. The counter does not increment on `SMC` instructions trapped as a Hyp Trap exception.

`0x008A`, **EXC_HVC, Exception taken, Hypervisor Call**

This event is similar to Exception taken, EXC_TAKEN, but the counter counts only Hypervisor Call exceptions. The counter counts for both Hypervisor Call exceptions *Taken locally* in the hypervisor and those taken as an exception from Non-secure EL1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0x008B`, **EXC_TRAP_PABORT, Exception taken, Instruction Abort not Taken locally**

This event is similar to Exception taken, EXC_TAKEN, but the counter counts only Instruction Abort exceptions not *Taken locally*.

`0x008C`, **EXC_TRAP_DABORT, Exception taken, Data Abort or SError not Taken locally**

This event is similar to Exception taken, EXC_TAKEN, but the counter counts only Data Abort or SError interrupt exceptions not *Taken locally*.

`0x008D`, **EXC_TRAP_OTHER, Exception taken, other traps not Taken locally**

This event is similar to Exception taken, EXC_TAKEN, but the counter counts only those traps that are not counted as:

- Exception taken, Secure Monitor Call (EXC_SMC).
- Exception taken, Hypervisor Call (EXC_HVC).
- Exception taken, Instruction Abort not *Taken locally* (EXC_TRAP_PABORT).
- Exception taken, Data Abort or SError not *Taken locally* (EXC_TRAP_DABORT).
- Exception taken, IRQ not *Taken locally* (EXC_TRAP_IRQ).
- Exception taken, FIQ not *Taken locally* (EXC_TRAP_FIQ).

`0x008E`, **EXC_TRAP_IRQ, Exception taken, IRQ not Taken locally**

This event is similar to Exception taken, EXC_TAKEN, but the counter counts only IRQ exceptions not *Taken locally*.

`0x008F`, **EXC_TRAP_FIQ, Exception taken, FIQ not Taken locally**

This event is similar to Exception taken, EXC_TAKEN, but the counter counts only FIQ exceptions not *Taken locally*.

`0x0090`, **RC_LD_SPEC, Release consistency operation speculatively executed, Load-Acquire**

The counter counts memory-read operations with acquire or acquirepc semantics that are speculatively executed.

`0x0091`, **RC_ST_SPEC, Release consistency operation speculatively executed, Store-Release**

The counter counts memory-write operations with release semantics that are speculatively executed.

`0x00A0`, **L3D_CACHE_RD, Attributable Level 3 data or unified cache access, read**

This event is similar to Attributable Level 3 data or unified cache access, L3D_CACHE, but the counter counts only attributable memory read operations that cause a cache access to at least the Level 3 data or unified cache.

`0x00A1`, **L3D_CACHE_WR, Attributable Level 3 data or unified cache access, write**

This event is similar to Attributable Level 3 data or unified cache access, L3D_CACHE, but the counter counts only attributable memory write operations that cause a cache access to at least the Level 3 data or unified cache.

`0x00A2`, **L3D_CACHE_REFILL_RD, Attributable Level 3 data or unified cache refill, read**

This event is similar to Attributable Level 3 data or unified cache refill, L3D_CACHE_REFILL, but the counter counts only attributable memory read operations that cause a refill of at least the Level 3 data or unified cache from outside the Level 3cache. See also *Relationship between REFILL events and associated access events.* on page K3-7907

`0x00A3`, **L3D_CACHE_REFILL_WR, Attributable Level 3 data or unified cache refill, write**

This event is similar to Attributable Level 3 data or unified cache refill, L3D_CACHE_REFILL, but the counter counts only attributable memory write operations that cause a refill of at least the Level 3 data or unified cache from outside the Level 3cache. See also *Relationship between REFILL events and associated access events.* on page K3-7907

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

0x00A6, L3D_CACHE_WB_VICTIM, Attributable Level 3 data or unified cache Write-Back, victim

This event is similar to Attributable Level 3 data cache Write-Back, L3D_CACHE_WB, but the counter counts only Write-Backs that are a result of the line being allocated for an access made by the PE.

If ARMv8.4-PMU is not implemented, Write-Backs caused by the execution of a cache maintenance instruction are not counted. If ARMv8.4-PMU is implemented, it is IMPLEMENTATION DEFINED whether Write-Backs caused by the execution of a cache maintenance instruction are counted.

It is IMPLEMENTATION DEFINED whether a write of a whole cache line that is not the result of the eviction of a line from the cache is counted. For example, this might occur if the PE detects streaming writes to memory and does not allocate lines to the cache, or as the result of a DC ZVA.

0x00A7, L3D_CACHE_WB_CLEAN, Level 3 data or unified cache Write-Back, cache clean

This event is similar to Attributable Level 3 data cache Write-Back, L3D_CACHE_WB, but the counter counts only Write-Backs that are a result of a coherency operation made by another PE or are caused by the execution of a cache maintenance instruction. Whether Write-Backs that are caused by the execution of a cache maintenance instruction are counted is IMPLEMENTATION DEFINED.

Note

The transfer of a dirty cache line from the Level 3 data cache of this PE to the Level 3 data cache of another PE due to a hardware coherency operation is not counted unless the dirty cache line is also written back to a Level 3 cache or memory.

If a coherency request from a requestor outside the PE results in a Write-Back, it is an Unattributable event.

0x00A8, L3D_CACHE_INVAL, Attributable Level 3 data or unified cache access, invalidate

The counter counts each invalidation of a cache line in the Level 3 data or unified cache.

The counter does not count events if a cache refill invalidates a line.

If ARMv8.4-PMU is not implemented, the counter does not count locally-executed cache maintenance instructions that operate by set/way. If ARMv8.4-PMU is implemented, it is IMPLEMENTATION DEFINED whether the counter counts locally-executed cache maintenance instructions that operate by set/way.

Note

Software that uses this event must know whether the Level 3 data cache is shared with other PEs. This event does not follow the general rule of Level 3 data cache events of only counting Attributable events.

K3.1.1 Relationship between REFILL events and associated access events.

CACHE_REFILL and TLB_REFILL events count the refills for accesses that are counted by the corresponding CACHE or TLB event. Table K3-2 shows this correspondence.

Table K3-2 Relationship between REFILL events and associated access events

REFILL event	Access event	Ratio REFILL/Access
0x0042 L1D_CACHE_REFILL_RD	0x0040 L1D_CACHE_RD	Attributable Level 1 cache refill rate, read
0x0043 L1D_CACHE_REFILL_WR	0x0041 L1D_CACHE_WR	Attributable Level 1 cache refill rate, write
0x004C L1D_TLB_REFILL_RD	0x004E L1D_TLB_RD	Attributable Level 1 TLB refill rate, read
0x004D L1D_TLB_REFILL_WR	0x004F L1D_TLB_WR	Attributable Level 1 TLB refill rate, write

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K3-2 Relationship between REFILL events and associated access events (continued)

REFILL event	Access event	Ratio REFILL/Access
`0x0052` L2D_CACHE_REFILL_RD	0x0050 L2D_CACHE_RD	Attributable Level 2 data cache refill rate, read
`0x0053` L2D_CACHE_REFILL_WR	0x0051 L2D_CACHE_WR	Attributable Level 2 data cache refill rate, write
`0x005C` L2D_TLB_REFILL_RD	0x005E L2D_TLB_RD	Attributable Level 2 data TLB refill rate, read
`0x005D` L2D_TLB_REFILL_WR	0x005F L2D_TLB_WR	Attributable Level 2 data TLB refill rate, write
`0x00A2` L3D_CACHE_REFILL_RD	0x00A0 L3D_CACHE_RD	Attributable Level 3 data cache refill rate, read
`0x00A3` L3D_CACHE_REFILL_WR	0x00A1 L3D_CACHE_WR	Attributable Level 3 data cache refill rate, write

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K3.2 Summary of events for exceptions taken to an Exception level using AArch64

Table K3-3 shows the events for exceptions taken to an Exception level using AArch64.

Table K3-3 Events for exceptions taken to an Exception level using AArch64

ESR.EC	Description		Event number and classification for exceptions	
			Taken locally	Not *Taken locally*
0x00	Unknown or uncategorized		0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x01	WFE/WFI traps		0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x03	AArch32 MCR/MRC traps on (coproc==0b1111) accesses		0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x04	AArch32 MCRR/MRRC traps on (coproc==0b1111) accesses		0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x05	AArch32 MCR/MRC traps on (coproc==0b1110) accesses		0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x06	AArch32 LDC/STC traps on (coproc==0b1110) accesses		0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x07	Advanced SIMD or FP traps		0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x08	AArch32 MVFR* and FPSID traps		-	0x008D, EXC_TRAP_OTHER
0x0C	AArch32 MCRR/MRRC traps on (coproc==0b1110) accesses		0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x0E	Illegal instruction set state		0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x11	AArch32 SVC		0x0082, EXC_SVC	0x008D, EXC_TRAP_OTHER
0x12	AArch32 HVC that is not disabled		-	0x008A, EXC_HVC
0x13	AArch32 SMC that is not disabled	to EL2	-	0x008D, EXC_TRAP_OTHER
		to EL3	-	0x0088, EXC_SMC
0x15	AArch64 SVC		0x0082, EXC_SVC	0x008D, EXC_TRAP_OTHER
0x16	AArch64 HVC that is not disabled		0x008A, EXC_HVC	0x008A, EXC_HVC
0x17	AArch64 SMC that is not disabled	to EL2	-	0x008D, EXC_TRAP_OTHER
		to EL3	0x0088, EXC_SMC	0x0088, EXC_SMC
0x18	AArch64 MSR, MRS and System instruction traps		0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x19	SVE traps		0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x1F	IMPLEMENTATION DEFINED exception taken to EL3		IMPLEMENTATION DEFINED[a]	IMPLEMENTATION DEFINED[a]
0x20	Instruction Abort from below		0x0083, EXC_PABORT	0x008B, EXC_TRAP_PABORT
0x21	Instruction Abort from current Exception level		0x0083, EXC_PABORT	-
0x22	PC alignment		0x0083, EXC_PABORT	0x008B, EXC_TRAP_PABORT

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K3-3 Events for exceptions taken to an Exception level using AArch64 (continued)

ESR.EC	Description	Event number and classification for exceptions	
		Taken locally	Not *Taken locally*
0x24	Data Abort from below	0x0084, EXC_DABORT	0x008C, EXC_TRAP_DABORT
0x25	Data Abort from current Exception level	0x0084, EXC_DABORT	-
0x26	SP alignment fault exception	0x0084, EXC_DABORT	0x008C, EXC_TRAP_DABORT
0x28	AArch32 FP exception	0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x2C	AArch64 FP exception	0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x2F	SError interrupt	0x0084, EXC_DABORT	0x008C, EXC_TRAP_DABORT
0x30	Breakpoint from below	0x0083, EXC_PABORT	0x008B, EXC_TRAP_PABORT
0x31	Breakpoint from current Exception level	0x0083, EXC_PABORT	-
0x32	Software step from below	0x0083, EXC_PABORT	0x008B, EXC_TRAP_PABORT
0x33	Software step from current Exception level	0x0083, EXC_PABORT	-
0x34	Watchpoint from below	0x0084, EXC_DABORT	0x008C, EXC_TRAP_DABORT
0x35	Watchpoint from current Exception level	0x0084, EXC_DABORT	-
0x38	AArch32 BKPT instruction	0x0083, EXC_PABORT	0x008B, EXC_TRAP_PABORT
0x3A	AArch32 Vector Catch debug event	0x0083, EXC_PABORT	0x008B, EXC_TRAP_PABORT
0x3C	AArch64 BRK instruction	0x0083, EXC_PABORT	0x008B, EXC_TRAP_PABORT
-	IRQ interrupt	0x0086, EXC_IRQ	0x008E, EXC_TRAP_IRQ
-	FIQ interrupt	0x0087, EXC_FIQ	0x008F, EXC_TRAP_FIQ
-	All other values	-	0x008D, EXC_TRAP_OTHER

a. The exception reported with EC 0x1F is IMPLEMENTATION DEFINED, and therefore it is IMPLEMENTATION DEFINED which event counts the exception, except that the event that counts the exception must correctly indicate whether the exception was *Taken locally*.

Note

The Glossary defines the term *Taken locally*, that is used in event definitions in this chapter. See also *Exception levels* on page D1-2314 for more information.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Appendix K4
Recommendations for Reporting Memory Attributes on an Interconnect

This appendix describes the Arm recommendations for reporting the memory attributes that are assigned by the PE. It contains the following section:

- *Arm recommendations for reporting memory attributes on an interconnect* on page K4-7912.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K4.1 Arm recommendations for reporting memory attributes on an interconnect

The Arm architecture defines the architectural interface between software and the PE hardware. This means the mechanisms by which different memory type and Cacheability attributes are presented on an interface to an interconnect fabric such as AMBA® AXI are, strictly, outside the scope of the architecture. This appendix describes an approach for the interface between a PE implementation and an interconnect fabric that Arm strongly recommends, but these recommendations do not form part of the Armv8 architecture.

K4.1.1 Effect of microarchitectural choices on memory attributes

Implementations of the Arm architecture permit considerable variability in the presentation of memory attributes on the interconnect fabric, particularly in cases where the PE implementation does not provide optimized support for a memory type. For example, an implementation might treat Write-Through locations as Non-cacheable at some level of cache, because functionally this is consistent with the definition of Write-Through, but for the particular implementation the performance trade-off does not merit the hardware directly providing Write-Through capability. However, in such implementations, the assigned memory attributes are not changed by the microarchitectural choices. The microarchitecture simply implements different ways of handling some memory attributes.

Therefore, Arm strongly recommended that where any or all of the following memory attributes are presented on the interface between a PE and an interconnect fabric, the attributes that are presented are completely consistent with the attributes defined by the translation system:

- The memory type, Normal or Device.
- The Early write acknowledgement attribute.
- The ordering requirements.
- The Shareability.
- The Cacheability, including where practicable, the allocation hints.

Effect when memory accesses are forced to be Non-cacheable

Arm also strongly recommends that the effects of forcing accesses to Normal memory to be Non-cacheable, as described in *Enabling and disabling the caching of memory accesses* on page D4-2503 for AArch64 and in *Enabling and disabling the caching of memory accesses in AArch32 state* on page G4-5893 for AArch32, are reflected on the interconnect by the memory type and attributes used for memory transactions generated while the cache is disabled.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Appendix K5
Additional Information for Implementations of the Generic Timer

This appendix gives additional information about implementations of the Generic Timer. It contains the following sections:

- *Providing a complete set of features in a system level implementation* on page K5-7914.
- *Gray-count scheme for timer distribution scheme* on page K5-7916.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K5.1 Providing a complete set of features in a system level implementation

As an example system design, using memory-mapped Generic Timer components as described in Chapter I2 *System Level Implementation of the Generic Timer*, the feature set of a System registers counter and timer, in an implementation that includes Non-secure EL2 and EL3, can be implemented using the following set of timer frames:

- A CNTCTLBase control frame.
- The following CNTBaseN*N* timer frames:

  Frame 0 Accessible by Non-secure accesses, with second view and virtual capability. This provides the Non-secure EL1&0 timers.

  Frame 1 Accessible by Non-secure accesses, with no second view and no virtual capability. This provides the Non-secure EL2 timers.

  Frame 2 Accessible only by Secure accesses, with a second view but no virtual capability. This provides the Secure PL1&0 timers, meaning:

  - Compared to a PE where EL3 is using AArch32, it provides the only Secure state timer.
  - Compared to a PE where EL3 is using AArch64, it provides the Secure EL1&0 timer.

  Frame 3 Accessible only by Secure accesses, with no second view and no virtual capability. This provides the Secure EL3 timers.

  Note

  This frame is not required for a memory-mapped timer that provides only the feature set of a PE for which EL3 is using AArch32.

In this implementation, the full set of implemented frames, and accessibility as memory pages in the different translation regimes, is as follows:

CNTCTLBase

The control frame. This frame is accessible in both the Secure and Non-secure memory maps, and:

- In the Secure EL1&0 translation regime, this frame is accessible only at EL1.
- In the Non-secure EL2 translation regime, this frame is accessible.
- In the Non-secure EL1&0 translation regime, this frame is not accessible.

CNTBase0 The first view of the Non-secure EL1&0 timers. This frame is accessible only in the Non-secure memory map, and:

- In the Secure EL1&0 translation regime, this frame is accessible only at EL1.
- In the Non-secure EL2 translation regime, this frame is accessible.
- In the Non-secure EL1&0 translation regime, this frame is accessible only at EL1.

CNTEL0Base0

The second view of CNTBase0, meaning it is the EL0 view of the Non-secure EL1&0 timers. This frame is accessible only in the Non-secure memory map, and:

- In the Secure EL1&0 translation regime, the architecture permits this frame to be accessible at EL1, or at EL1 and EL0, but does not require either of these options.
- In the Non-secure EL2 translation regime, this frame is accessible.
- In the Non-secure EL1&0 translation regime, this frame is accessible at EL1 and EL0.

CNTBase1 The first and only view of the Non-secure EL2 timers. This frame is accessible only in Non-secure memory map, and:

- When EL3 is using AArch64:
  - In the Secure EL1&0 translation regime, this frame is accessible only at EL1.
  - In the Secure EL3 translation regime, this frame is accessible.
- When EL3 is using AArch32, in the Secure PL1&0 translation regime, this frame is accessible only at PL1 (EL3).
- In the Non-secure EL2 translation regime, this frame is accessible.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- In the Non-secure EL1&0 translation regime, this frame is not accessible.

CNTBase2 The first view of the Secure EL1&0, or PL1&0 timers.

Note

In AArch64 state, these timers are always called the Secure EL1&0 timers. In AArch32 state they are usually called the Secure PL1&0 timers because, in AArch32 Secure state, whether some of the PE modes map to EL1 or to EL3 depends on whether EL3 is using AArch64 or is using AArch32, see *Security state, Exception levels, and AArch32 execution privilege* on page G1-5682.

This frame is accessible only in the Secure memory map, and:

- When EL3 is using AArch64:
  - In the Secure EL1&0 translation regime, this frame is accessible only at EL1.
  - In the Secure EL3 translation regime, this frame is accessible.
- When EL3 is using AArch32, in the Secure PL1&0 translation regime, this frame is accessible only at PL1 (EL3).
- Because the frame is in Secure memory, it is not accessible in any Non-secure translation regime.

CNTEL0Base2

The second view of CNTBase2, meaning it is the EL0 view of the Secure EL1&0, or PL1&0, timers.

Note

See the Note in the description of the CNTBase2 frame for more information about the naming of these timers.

This frame is accessible only in the Secure memory map, and:

- When EL3 is using AArch64:
  - In the Secure EL1&0 translation regime, this frame is accessible at EL1 and EL0.
  - In the Secure EL3 translation regime, this frame is accessible.
- When EL3 is using AArch32, in the Secure PL1&0 translation regime, this frame is accessible at PL1 (EL3) and EL0.
- Because the frame is in Secure memory, it is not accessible in any Non-secure translation regime.

CNTBase3 The first and only view of the EL3 timers. This frame is accessible only in the Secure memory map, and:

- When EL3 is using AArch64:
  - In the Secure EL1&0 translation regime, this frame is not accessible.
  - In the Secure EL3 translation regime, this frame is accessible.
- When EL3 is using AArch32, this frame is not accessible.
- Because the frame is in Secure memory, it is not accessible in any Non-secure translation regime.

Note

About the Virtual Memory System Architecture (VMSA) on page D5-2532 describes the VMSAv8-64 translation regimes, and *About VMSAv8-32* on page G5-5922 describes the VMSAv8-32 translation regimes.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K5.2 Gray-count scheme for timer distribution scheme

The distribution of the Counter value using a Gray-code provides a relatively simple mechanism to avoid any danger of the count being sampled with an intermediate value even if the clocking is asynchronous. It has a further advantage that the distribution is relatively low power, since only one bit changes on the main distribution wires for each clock tick.

A suitable Gray-coding scheme can be achieved with the following logic:

```
Gray[N] = Count[N]
Gray[i] = (XOR(Gray[N:i+1])) XOR Count[i] for N-1 >= i >= 0
Count[i] = XOR(Gray[N:i]) for N >= i >= 0
```

This is for an N+1 bit counter, where Count is a conventional binary count value, and Gray is the corresponding Gray count value.

Note

This scheme has the advantage of being relatively simple to switch, in either direction, between operating with low-frequency and low-precision, and operating with high-frequency and high-precision. To achieve this, the ratio of the frequencies must be 2^n, where n is an integer. A switch-over can occur only on the $2n+1$ boundary to avoid losing the Gray-coding property on a switch-over.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Appendix K6
Legacy Instruction Syntax for AArch32 Instruction Sets

This appendix describes the legacy instruction syntax in the Arm instruction sets, and their *Unified Assembler Language* (UAL) equivalents. It contains the following section:

- *Legacy Instruction Syntax* on page K6-7918.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K6.1 Legacy Instruction Syntax

Early versions of the Arm Architecture defined an assembly language for A32 (ARM) instructions, and a separate assembly language for T32 (Thumb) instructions. UAL is based on the A32 assembly language, with some changes to the instruction syntax. The appendix describes those changes. The pre-UAL mnemonics are compatible with UAL, and might be supported by an assembler.

The original T32 assembly language is not compatible with UAL, and is not described in the manual.

K6.1.1 Pre-UAL instruction syntax for the A32 base instructions

Table K6-1 lists the syntax for the A32 base instructions that have changed after UAL was introduced.

Table K6-1 Pre-UAL instruction syntax for the A32 base instructions

Pre-UAL syntax	UAL equivalent	See
ADC<c>S	ADCS<c>	*ADC, ADCS (immediate)* on page F5-4243, *ADC, ADCS (register)* on page F5-4246, *ADC, ADCS (register-shifted register)* on page F5-4250
ADD<c>S	ADDS<c>	*ADD, ADDS (immediate)* on page F5-4252, *ADD, ADDS (register)* on page F5-4256, *ADD, ADDS (register-shifted register)* on page F5-4260, *ADD, ADDS (SP plus immediate)* on page F5-4262, *ADD, ADDS (SP plus register)* on page F5-4265
AND<c>S	ANDS<c>	*AND, ANDS (immediate)* on page F5-4274, *AND, ANDS (register)* on page F5-4277, *AND, ANDS (register-shifted register)* on page F5-4281
BIC<c>S	BICS<c>	*BIC, BICS (immediate)* on page F5-4298, *BIC, BICS (register)* on page F5-4301, *BIC, BICS (register-shifted register)* on page F5-4305
EOR<c>S	EORS<c>	*EOR, EORS (immediate)* on page F5-4360, *EOR, EORS (register)* on page F5-4363, *EOR, EORS (register-shifted register)* on page F5-4367
LDC<c>L	LDCL<c>	*LDC (immediate)* on page F5-4394, *LDC (literal)* on page F5-4396
LDM<c>IA, LDM<c>FD	LDM<c>	*LDM, LDMIA, LDMFD* on page F5-4398
LDM<c>DA, LDM<c>FA	LDMDA<c>	*LDMDA, LDMFA* on page F5-4406
LDM<c>DB, LDM<c>EA	LDMDB<c>	*LDMDB, LDMEA* on page F5-4408
LDM<c>IB, LDM<c>ED	LDMIB<c>	*LDMIB, LDMED* on page F5-4411
LDR<c>B	LDRB<c>	*LDRB (immediate)* on page F5-4423, *LDRB (literal)* on page F5-4427, *LDRB (register)* on page F5-4429
LDR<c>BT	LDRBT<c>	*LDRBT* on page F5-4432
LDR<c>D	LDRD<c>	*LDRD (immediate)* on page F5-4435, *LDRD (literal)* on page F5-4438, *LDRD (register)* on page F5-4441

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K6-1 Pre-UAL instruction syntax for the A32 base instructions (continued)

Pre-UAL syntax	UAL equivalent	See
LDR<c>H	LDRH<c>	*LDRH (immediate)* on page F5-4452, *LDRH (literal)* on page F5-4456, *LDRH (register)* on page F5-4458
LDR<c>SB	LDRSB<c>	*LDRSB (immediate)* on page F5-4464. *LDRSB (literal)* on page F5-4467, *LDRSB (register)* on page F5-4469
LDR<c>SH	LDRSH<c>	*LDRSH (immediate)* on page F5-4475, *LDRSH (literal)* on page F5-4478, *LDRSH (register)* on page F5-4480
LDR<c>T	LDRT<c>	*LDRT* on page F5-4486
MLA<c>S	MLAS<c>	*MLA, MLAS* on page F5-4509
LSLS <Rd>, <Rn>, #0	MOVS <Rd>, <Rm>	*MOV, MOVS (immediate)* on page F5-4513, *MOV, MOVS (register)* on page F5-4517
MOV<c>S	MOVS<c>	
MUL<c>S	MULS<c>	*MUL, MULS* on page F5-4547
MVN<c>S	MVNS<c>	*MVN, MVNS (immediate)* on page F5-4549, *MVN, MVNS (register)* on page F5-4551, *MVN, MVNS (register-shifted register)* on page F5-4554
ORR<c>S	ORRS<C>	*ORR, ORRS (immediate)* on page F5-4562, *ORR, ORRS (register)* on page F5-4565, *ORR, ORRS (register-shifted register)* on page F5-4569
QADDSUBX	QASX	*QASX* on page F5-4602
QSUBADDX	QSAX	*QSAX* on page F5-4608
RSB<c>S	RSBS<c>	*RSB, RSBS (immediate)* on page F5-4639, *RSB, RSBS (register)* on page F5-4642, *RSB, RSBS (register-shifted register)* on page F5-4645
RSC<c>S	RSCS<c>	*RSC, RSCS (immediate)* on page F5-4647, *RSC, RSCS (register)* on page F5-4649, *RSC, RSCS (register-shifted register)* on page F5-4651
SADDSUBX	SASX	*SASX* on page F5-4657
SBC<c>S	SBCS<c>	*SBC, SBCS (immediate)* on page F5-4661, SBC, SBCS (register), *SBC, SBCS (register-shifted register)* on page F5-4668
SHADDSUBX	SHASX	*SHASX* on page F5-4686
SHSUBADDX	SHSAX	*SHSAX* on page F5-4688
SMI<c>	SMC<c>	*SMC* on page F5-4694
SMLAL<c>S	SMLALS<c>	*SMLAL, SMLALS* on page F5-4700
SMULL<c>S	SMULLS<c>	*SMULL, SMULLS* on page F5-4723

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K6-1 Pre-UAL instruction syntax for the A32 base instructions (continued)

Pre-UAL syntax	UAL equivalent	See
SSUBADDX<c>	SSAX<c>	*SSAX* on page F5-4737
STC<c>L	STCL<c>	*STC* on page F5-4745
STM<c>EA, STM<c>IA	STM<c>	*STM, STMIA, STMEA* on page F5-4765
STM<c>DA, STM<c>ED	STMDA<c>	*STMDA, STMED* on page F5-4771
STM<c>DB, STM<c>FD	STMDB<c>	*STMDB, STMFD* on page F5-4773
STM<c>IB, STM<c>FA	STMIB<c>	*STMIB, STMFA* on page F5-4776
STR<c>B	STRB<c>	*STRB (immediate)* on page F5-4786, *STRB (register)* on page F5-4790
STR<c>BT	STRBT<c>	*STRBT* on page F5-4793
STR<c>D	STRD<c>	*STRD (immediate)* on page F5-4797, *STRD (register)* on page F5-4801
STR<c>H	STRH<c>	*STRH (immediate)* on page F5-4815, *STRH (register)* on page F5-4819
STR<c>T	STRT<c>	*STRT* on page F5-4826
SUB<c>S	SUBS<c>	*SUB, SUBS (immediate)* on page F5-4831, *SUB, SUBS (register)* on page F5-4835, *SUB, SUBS (register-shifted register)* on page F5-4839, *SUB, SUBS (SP minus immediate)* on page F5-4841, *SUB, SUBS (SP minus register)* on page F5-4844
SWI	SVC	*SVC* on page F5-4847
UADDSUBX	UASX	*UASX* on page F5-4881
UHADDSUBX	UHASX	*UHASX* on page F5-4893
UHSUBADDX	UHSAX	*UHSAX* on page F5-4895
UMLAL<c>S	UMLALS<c>	*UMLAL, UMLALS* on page F5-4903
UMULL<c>S	UMULLS<c>	*UMULL, UMULLS* on page F5-4905
UQADDSUBX	UQASX	*UQASX* on page F5-4911
UQSUBADDX	UQSAX	*UQSAX* on page F5-4913
USUBADDX	USAX	*USAX* on page F5-4927
UEXT8	UXTB	*UXTB* on page F5-4939
UEXT16	UXTH	*UXTH* on page F5-4943

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K6.1.2 Pre-UAL instruction syntax for the A32 floating-point instructions

Table K6-2 lists the syntax for A32 floating-point instructions that have changed after UAL was introduced.

Table K6-2 Pre-UAL instruction syntax for A32 floating-point instructions

Pre-UAL syntax	UAL equivalent	See
FABSD	VABS.F64	*VABS* on page F6-5001
FABSS	VABS.F32	
FADDD	VADD.F64	*VADD (floating-point)* on page F6-5015
FADDS	VADD.F32	
FCMPEZD	VCMPE.F64	*VCMPE* on page F6-5090
FCMPEZS	VCMPE.F32	
FCMPZD	VCMP.F64	*VCMP* on page F6-5086,
FCMPZS	VCMP.F32	
FCONSTD <Dd>, #<imm8>	VMOV.F64 <Dd>, #<fpimm>	*VMOV (immediate)* on page F6-5320 For more information, see *FCONST* on page K6-7923.
FCONSTS <Sd>, #<imm8>	VMOV.F32 <Sd>, #<fpimm>	
FCPYD	VMOV.F64	*VMOV (register)* on page F6-5327
FCPYS	VMOV.F32	
FCVTDS	VCVT.F64.F32	*VCVT (between double-precision and single-precision)* on page F6-5098
FCVTSD	VCVT.F32.F64	
FDIVD	VDIV.F64	*VDIV* on page F6-5149
FDIVS	VDIV.F32	
FLDD	VLDR.F64	*VLDR (immediate)* on page F6-5264 *VLDR (literal)* on page F6-5267
FLDMD, FLDMIAD	VLDM.F64	*VLDM, VLDMDB, VLDMIA* on page F6-5259
FLDMS	VLDM.F32	
FLDS	VLDR.F32	*VLDR (immediate)* on page F6-5264 *VLDR (literal)* on page F6-5267
FMACD	VMLA.F64	*VMLA (floating-point)* on page F6-5288
FMACS	VMLA.F32	
FMDHR <Dd>, <Rt>	VMOV <Dd[1]>, <Rt>	*VMOV (general-purpose register to scalar)* on page F6-5331
FMDLR <Dd>, <Rt>	VMOV <Dd[0]>, <Rt>	
FMDRR	VMOV	*VMOV (between two general-purpose registers and a doubleword floating-point register)* on page F6-5316
FMRDH <Rt>, <Dd>	VMOV <Rt>, <Dd[1]>	*VMOV (scalar to general-purpose register)* on page F6-5335
FMRDL <Rt>, <Dd>	VMOV <Rt>, <Dd[0]>	

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K6-2 Pre-UAL instruction syntax for A32 floating-point instructions (continued)

Pre-UAL syntax	UAL equivalent	See
FMRRD	VMOV	*VMOV (between two general-purpose registers and a doubleword floating-point register)* on page F6-5316
FMRRS	VMOV	*VMOV (between two general-purpose registers and two single-precision registers)* on page F6-5337
FMRS	VMOV	*VMOV (between general-purpose register and single-precision)* on page F6-5333
FMRX	VMRS	*VMRS* on page F6-5346
FMSCD	VNMLS.F64	*VNMLS* on page F6-5379
FMSCS	VNMLS.F32	
FMSR	VMOV	*VMOV (between general-purpose register and single-precision)* on page F6-5333
FMSRR	VMOV	*VMOV (between two general-purpose registers and two single-precision registers)* on page F6-5337
FMSTAT	VMRS APSR_nzcv, FPSCR	*VMRS* on page F6-5346
FMULD	VMUL.F64	*VMUL (floating-point)* on page F6-5351
FMULS	VMUL.F32	
FMXR	VMSR	*VMSR* on page F6-5349
FNEGD	VNEG.F64	*VNEG* on page F6-5372
FNEGS	VNEG.F32	
FNMACD	VMLS.F64	*VNMLS* on page F6-5379
FNMACS	VMLS.F32	
FNMSCD	VNMLA.F64	*VNMLA* on page F6-5376
FNMSCS	VNMLA.F32	
FNMULD	VNMUL.F64	*VNMUL* on page F6-5382
FNMULS	VNMUL.F32	
FSHTOD	VCVT.F64.S16	*VCVT (between floating-point and fixed-point, floating-point)* on page F6-5115
FSHTOS	VCVT.F32.S16	
FSITOD	VCVT.F64.S32	*VCVT (between floating-point and integer, Advanced SIMD)* on page F6-5102, *VCVTR* on page F6-5140
FSITOS	VCVT.F32.S32	
FSLTOD	VCVT.F64.S32	*VCVT (between floating-point and fixed-point, floating-point)* on page F6-5115
FSLTOS	VCVT.F32.S32	
FSQRTD	VSQRT.F64	*VSQRT* on page F6-5567
FSQRTS	VSQRT.F32	
FSTD	VSTR	*VSTR* on page F6-5622

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K6-2 Pre-UAL instruction syntax for A32 floating-point instructions (continued)

Pre-UAL syntax	UAL equivalent	See
FSTMD, FSTMIAS	VSTM.F64	*VSTM, VSTMDB, VSTMIA* on page F6-5617
FSTMS	VSTM.F32	
FSTS	VSTR	*VSTR* on page F6-5622
FSUBD	VSUB.F64	*VSUB (floating-point)* on page F6-5625
FSUBS	VSUB.F32	
FTOSHD	VCVT.S16.F64	*VCVT (between floating-point and fixed-point, floating-point)* on page F6-5115
FTOSHS	VCVT.S16.F23	
FTOSID	VCVT.S32.F64	*VCVT (between floating-point and integer, Advanced SIMD)* on page F6-5102
FTOSIS	VCVT.S32.F32	
FTOSIZD	VCVTR.S32.F64	*VCVTR* on page F6-5140
FTOSIZS	VCVTR.S32.F32	
FTOSLD	VCVT.S32.F64	*VCVT (between floating-point and fixed-point, floating-point)* on page F6-5115
FTOSLS	VCVT.S32.F32	
FTOUHD	VCVT.U16.F64	
FTOUHS	VCVT.U16.F32	
FTOUID	VCVT.U32.F64	*VCVT (between floating-point and integer, Advanced SIMD)* on page F6-5102
FTOUIS	VCVT.U32.F32	
FTOUIZD	VCVTR.U32.F64	*VCVTR* on page F6-5140
FTOUIZS	VCVTR.U32.F32	
FTOULD	VCVT.U32.F64	*VCVT (between floating-point and fixed-point, floating-point)* on page F6-5115
FTOULS	VCVT.U32.F32	
FUHTOD,	VCVT.F64.U16	
FUHTOS	VCVT.F64.U16	
FUITOD	VCVT.F64.U32	*VCVT (between floating-point and integer, Advanced SIMD)* on page F6-5102
FUITOS	VCVT.F32.U32	
FULTOD	VCVT.F64.U32	*VCVT (between floating-point and fixed-point, floating-point)* on page F6-5115
FULTOS	VCVT.F32.U32	

K6.1.3 FCONST

The syntax of FCONST is

```
FCONST<dest>{<c>} <Fd>, #<imm8>
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

where:

`<dest>` Specifies the destination data type. It must be one of:

- `S` Single-precision floating-point.
- `D` Double-precision floating-point.

`<c>` This is an optional field. It specifies the condition under which the instruction is executed. See *Conditional execution* on page F2-4081 for the range of available conditions and their encoding. If <c> is omitted, it defaults to *always* (`AL`).

`<Fd>` Specifies the destination register. It must be one of:

- `<Dd>` 64-bit name of the SIMD&FP destination register.
- `<Sd>` 32-bit name of the SMID&FP destination register.

`<imm8>` Specifies the immediate value used to generate the floating-point constant.

`FCONSTD{<c>} <Dd>, #<imm8>` maps to `VMOV.F64 <Dd>, #<fpimm>`

`FCONSTS{<c>} <Sd>, #<imm8>` maps to `VMOV.F32 <Sd>, #<fpimm>`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Appendix K7
Address Translation Examples

This appendix gives examples of address translations using the translation regimes described in Chapter D5 *The AArch64 Virtual Memory System Architecture* and Chapter G5 *The AArch32 Virtual Memory System Architecture*. It contains the following sections:

- *AArch64 Address translation examples* on page K7-7926.
- *AArch32 Address translation examples* on page K7-7938.

Note

This chapter gives examples of translation table lookups for the Armv8 address translation stages. It does not define any part of the address translation mechanism. If any information in this appendix appears to contradict the information in Chapter D5 *The AArch64 Virtual Memory System Architecture* or Chapter G5 *The AArch32 Virtual Memory System Architecture* then the information in Chapter D5 or Chapter G5 must be taken as the definition of the required behavior.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K7.1 AArch64 Address translation examples

Figure D5-1 on page D5-2542 shows the VMSAv8 address translation stages that are controlled by an Exception level that is using AArch64. *The VMSAv8-64 address translation system* on page D5-2540 describes the VMSAv8-64 address translation scheme. This section gives examples of the use of that scheme, for common translation requirements.

System registers relevant to MMU operation on page D5-2547 specifies the relevant registers, including the TCR_ELx and TTBR_ELx, or TTBR_ELxs, for each stage of address translation.

For any stage of translation, a TCR_ELx.T*n*SZ field indicates the supported input address size. For a stage of address translation controlled from an Exception level using AArch64, the supported input address size is $2^{(64-T_nSZ)}$.

This section describes:

- Performing the initial lookup, for an address for which the initial lookup is either:
  - At the highest lookup level used for the appropriate translation granule size.
  - Because of the concatenation of translation tables at the initial lookup level, one level down from the highest level used for the translation granule size.

  These descriptions take account of the following cases:
  - The IA size is smaller than the largest size for the translation level, see *Reduced IA width* on page D5-2559.
  - For a stage 2 translation, translation tables are concatenated, to move the initial lookup level down by one level, see *Concatenated translation tables* on page D5-2560.

  For examples of performing the initial lookup, see *Examples of performing the initial lookup*.
- The full translation flow for resolving a page of memory. These examples describe resolving the largest IA size supported by the initial lookup level. For these examples, see *Full translation flows for VMSAv8-64 address translation* on page K7-7932.

K7.1.1 Examples of performing the initial lookup

The address ranges used for the initial translation table lookup depend on the translation granule, as described in:

- *Performing the initial lookup using the 4KB translation granule*.
- *Performing the initial lookup using the 16KB granule* on page K7-7928.
- *Performing the initial lookup using the 64KB translation granule* on page K7-7930.

Performing the initial lookup using the 4KB translation granule

This subsection describes examples of the initial lookup when using the 4KB translation granule that Table D5-13 on page D5-2566 shows as starting at level 0 or at level 1. It includes those stage 2 translations where concatenation of translation tables is required for the lookup to start at level 1. This means that it gives specific examples of the mechanisms described in *The VMSAv8-64 address translation system* on page D5-2540.

Note

For stage 2 translations, the same principles apply to an initial lookup that Table D5-13 on page D5-2566 shows as starting at level 1. In this case, for some IA sizes concatenation of translation tables means the lookup can, instead, start at level 2.

The following subsections describe these examples of the initial lookup:

- *Initial lookup at level 0, 4KB translation granule* on page K7-7927.
- *Initial lookup at level 1, 4KB translation granule* on page K7-7927.

In all cases, for a stage 2 translation, the VTCR_EL2.SL0 field must indicate the required initial lookup level, and this level must be consistent with the value of the VTCR_EL2.T0SZ field, see *Overview of stage 2 translations, 4KB granule* on page D5-2565.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Initial lookup at level 0, 4KB translation granule

This subsection describes initial lookups with an input address width of (n+1) bits, meaning the input address is IA[n:0]. As Table D5-13 on page D5-2566 shows, a stage 1 or stage 2 initial lookup at level 0 is required when $39 \le n \le 47$. For these lookups:

- TTBR_ELx[47:(n-35)] specify the translation table base address.
- Bits[n:39] of the input address are bits[(n-36):3] of the descriptor offset in the translation table.

Note

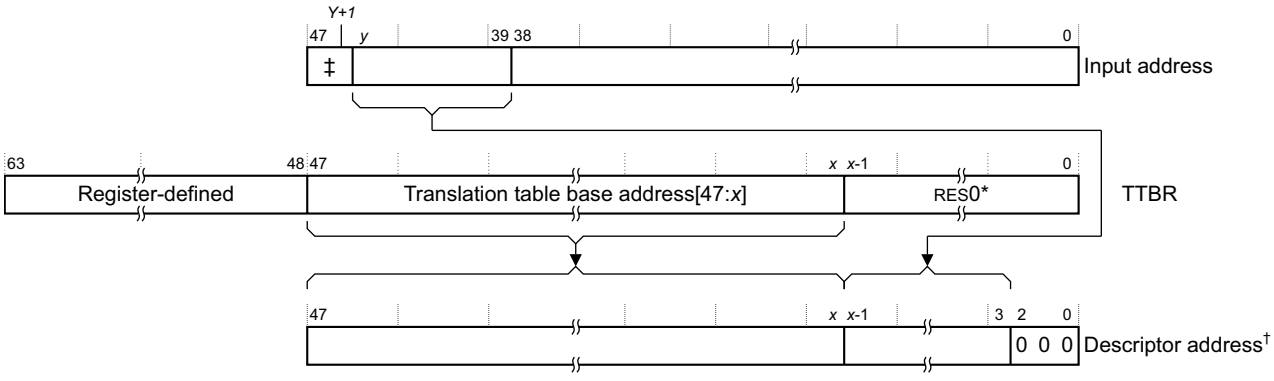
This means that, when the input address width is less than 48 bits

- The size of the translation table is reduced.
- More low-order bits of the TTBR_ELx are required to specify the translation table base address.
- Fewer input address bit are used to specify the descriptor offset in the translation table.

For example, if the input address width is 46 bits:

- The translation table size is 1KB,
- TTBR_ELx bits[47:10] specify the translation table base address.
- Input address bits[45:39] specify bits[9:3] of the descriptor offset.

Figure K7-1 shows this lookup.



Supported input address range is IA[y:0], $4 \le x \le 12$, $y = x + 35$. When y is 47 the field marked ‡ is absent.

† For an EL1&0 stage 1 translation, when EL2 is implemented and enabled in the current Security state, the IPA of the descriptor. Otherwise, the PA of the descriptor.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K7-1 Initial lookup for VMSAv8-64 using the 4KB granule, starting at level 0

Initial lookup at level 1, 4KB translation granule

This subsection describes initial lookups with an input address width of (n+1) bits, meaning the input address is IA[n:0].

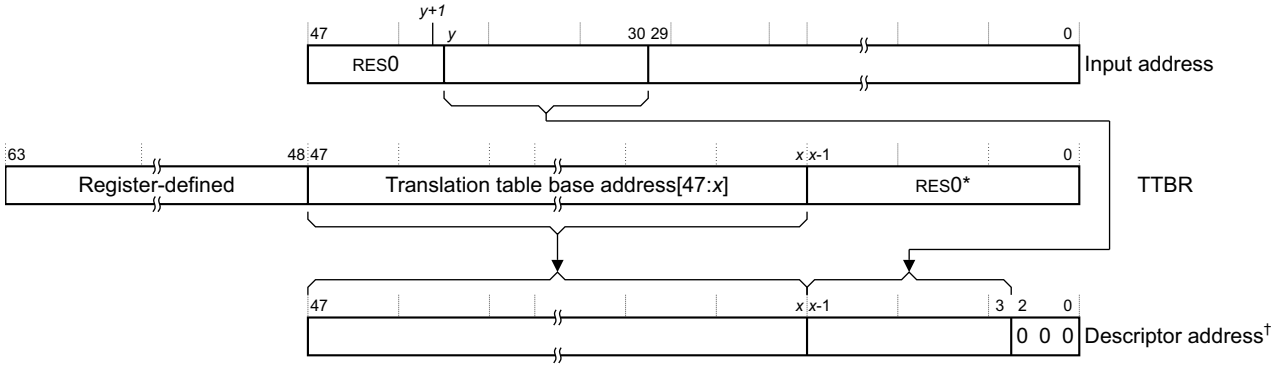
For a stage 1 or stage 2 initial lookup at level 1, without use of concatenated translation tables

As Table D5-13 on page D5-2566 shows, this applies to IA[n:0], where $30 \le n \le 38$. For these lookups:

- There is a single translation table at this level.
- TTBR_ELx[47:(n-26)] specify the translation table base address.
- Bits[n:30] of the input address are bits[(n-27):3] of the descriptor offset in the translation table.

Figure K7-2 on page K7-7928 shows this lookup.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta



Supported input address range is IA[y:0], $4 \le x \le 12$, $y = x + 26$.

† For an EL1&0 stage 1 translation, when EL2 is implemented and enabled in the current Security state, the IPA of the descriptor. Otherwise, the PA of the descriptor.

* Field has additional properties to the default RES0 definition, see the register description for more information.

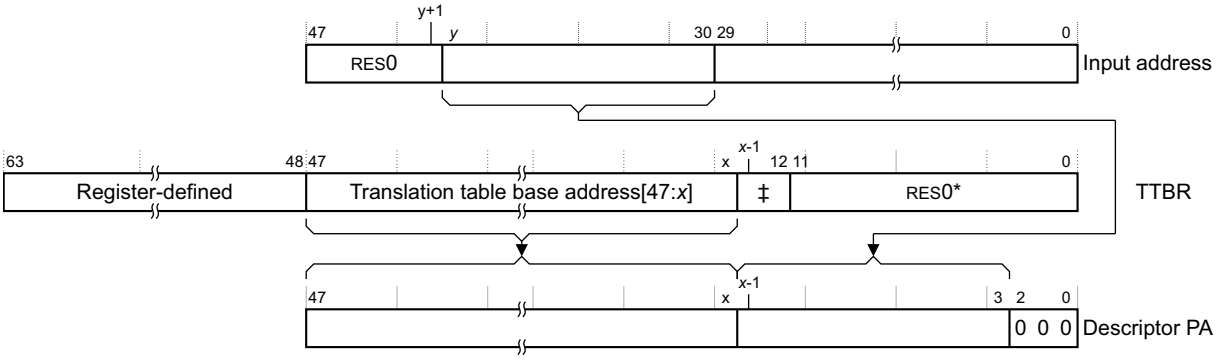
Figure K7-2 Initial lookup for VMSAv8-64 using the 4KB granule, starting at level 1, without concatenation

For a stage 2 initial lookup at level 1, with concatenated translation tables

As Table D5-13 on page D5-2566 shows, this applies to IA[n:0], where $39 \le n \le 42$. For these lookups:

- There are $2^{(n-38)}$ concatenated translation tables at this level.
- These concatenated translation tables must be aligned to $2^{(n-38)} \times$4KB. This means TTBR_ELx[(n-27):12] must be zero.
- TTBR_ELx[47:(n-26)] specify the base address of the block of concatenated translation tables.
- Bits[n:30] of the input address are bits[(n-27):3] of the descriptor offset from the base address of the block of concatenated translation tables.

Figure K7-3 shows this lookup.



Supported input address range is IPA[y:0], $4 \le x \le 16$, $y = x + 26$. The field marked ‡ must be zero.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K7-3 Initial lookup for VMSAv8-64 using the 4KB granule, starting at level 1, with concatenation

Performing the initial lookup using the 16KB granule

This subsection describes examples of the initial lookup when using the 16KB translation granule that Table D5-15 on page D5-2569 shows as starting at level 0 or at level 1. It includes those stage 2 translations where concatenation of translation tables is required for the lookup to start at level 1. This means that it gives specific examples of the mechanisms described in *The VMSAv8-64 address translation system* on page D5-2540.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

For stage 2 translations, the same principles apply to an initial lookup that Table D5-15 on page D5-2569 shows as starting at level 1. In this case, for some IA sizes concatenation of translation tables means the lookup can, instead, start at level 2.

The following subsections describe these examples of the initial lookup:

- *Initial lookup at level 0, 16KB translation granule*.
- *Initial lookup at level 1, 16KB translation granule*.

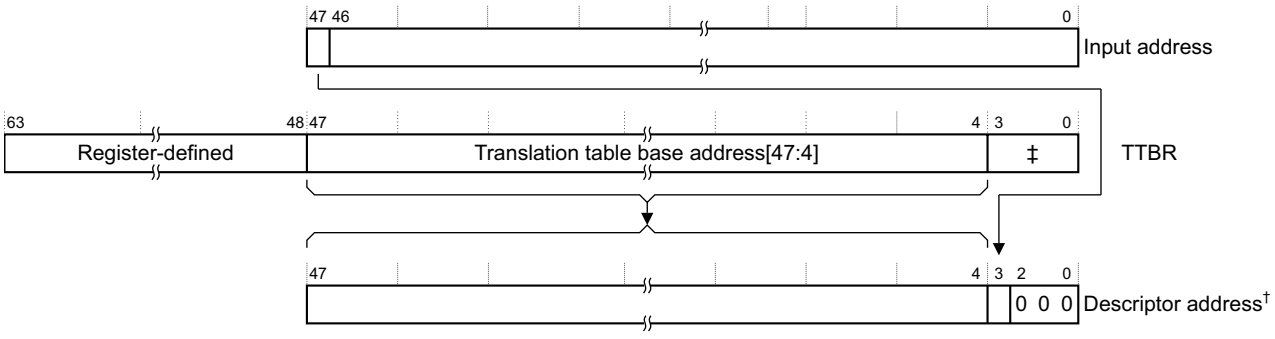
In all cases, for a stage 2 translation, the VTCR_EL2.SL0 field must indicate the required initial lookup level, and this level must be consistent with the value of the VTCR_EL2.T0SZ field, see *Overview of stage 2 translations, 16KB granule* on page D5-2569.

Initial lookup at level 0, 16KB translation granule

This subsection describes initial lookups with an input address width of (n+1) bits, meaning the input address is IA[n:0]. As Table D5-14 on page D5-2568 shows, the only case where an address translation using the 16KB granule starts at level 0 is a stage 1 translation of a 48-bit input address, IA[47:0]. For this lookup:

- The required translation table has only two entries, meaning its size is 16bytes, and it must be aligned to 16 bytes.
- TTBR_ELx[47:4] specify the translation table base address.
- Bit[47] of the input address is bits[3] of the descriptor offset in the translation table.

Figure K7-4 shows this lookup.



Supported input address range is IA[47:0]. The field marked ‡ is RES0*.

† For an EL1&0 stage 1 translation, when EL2 is implemented and enabled in the current Security state, the IPA of the descriptor. Otherwise, the PA of the descriptor.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K7-4 Initial lookup for VMSAv8-64 using the 16KB granule, starting at level 0

Initial lookup at level 1, 16KB translation granule

This subsection describes initial lookups with an input address width of (n+1) bits, meaning the input address is IA[n:0].

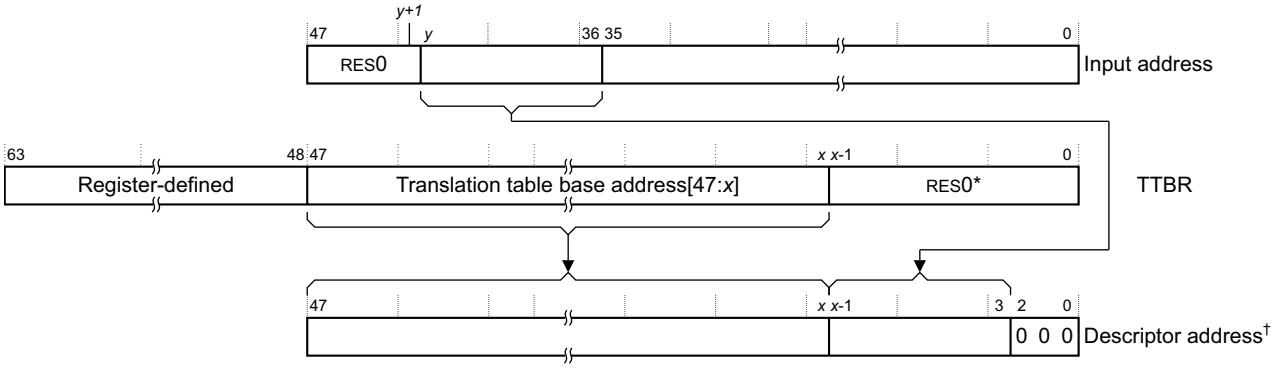
For a stage 1 or stage 2 initial lookup at level 1, without use of concatenated translation tables

As Table D5-15 on page D5-2569 shows, this applies to IA[n:0], where $36 \le n \le 46$. For these lookups:

- There is a single translation table at this level.
- TTBR_ELx[47:(n-32)] specify the translation table base address.
- Bits[n:36] of the input address are bits[(n-33):3] of the descriptor offset in the translation table.

Figure K7-5 on page K7-7930 shows this lookup.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta



Supported input address range is IA[*y*:0], 4 ≤ *x* ≤ 14, *y* = *x* + 32.

† For an EL1&0 stage 1 translation, when EL2 is implemented and enabled in the current Security state, the IPA of the descriptor. Otherwise, the PA of the descriptor.

* Field has additional properties to the default RES0 definition, see the register description for more information.

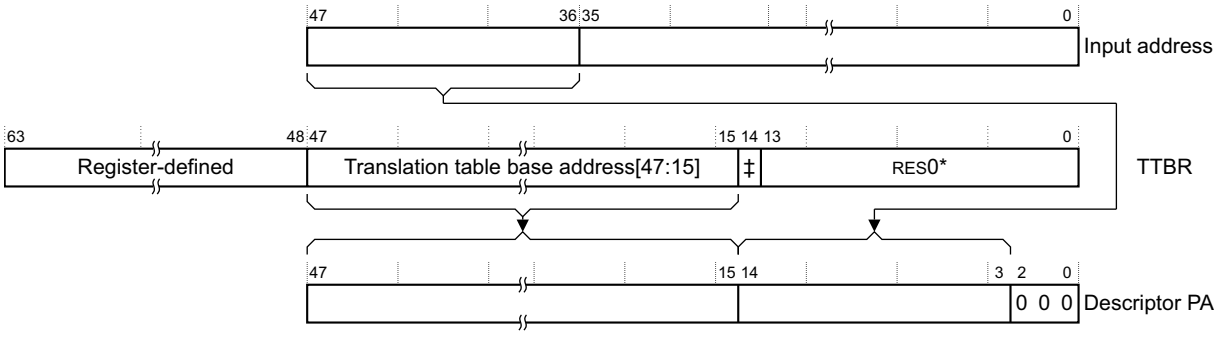
Figure K7-5 Initial lookup for VMSAv8-64 using the 16KB granule, starting at level 1, without concatenation

For a stage 2 initial lookup at level 1, with concatenated translation tables

As Table D5-15 on page D5-2569 shows, the only case where an address translation using the 16KB granule starts at level 1 because of concatenation of translation tables is a stage 2 translation of a 48-bit input address, IA[47:0]. For this lookup:

- There are two concatenated translation tables at this level.
- These concatenated translation tables must be aligned to 2×16KB. This means TTBR_ELx[14] must be zero.
- TTBR_ELx[47:15] specify the base address of the block of two concatenated translation tables.
- Bits[47:36] of the input address are bits[14:3] of the descriptor offset from the base address of the block of concatenated translation tables.

Figure K7-6 shows this lookup.



Supported input address range is IPA[47:0]. The bit marked ‡ must be zero.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K7-6 Initial lookup for VMSAv8-64 using the 16KB granule, starting at level 1, with concatenation

Performing the initial lookup using the 64KB translation granule

This subsection describes examples of the initial lookup when using the 64KB translation granule that Table D5-17 on page D5-2573 shows as starting at level 1 or at level 2. It includes those stage 2 translations where concatenation of translation tables is required for the lookup to start at level 2. This means that it gives specific examples of the mechanisms described in *The VMSAv8-64 address translation system* on page D5-2540.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

For stage 2 translations, the same principles apply to an initial lookup that Table D5-17 on page D5-2573 shows as starting at level 2. In this case, for some IA sizes concatenation of translation tables means the lookup can, instead, start at level 3.

The following subsections describe these examples of the initial lookup:

- *Initial lookup at level 1, 64KB translation granule*.
- *Initial lookup at level 2, 64KB translation granule*.

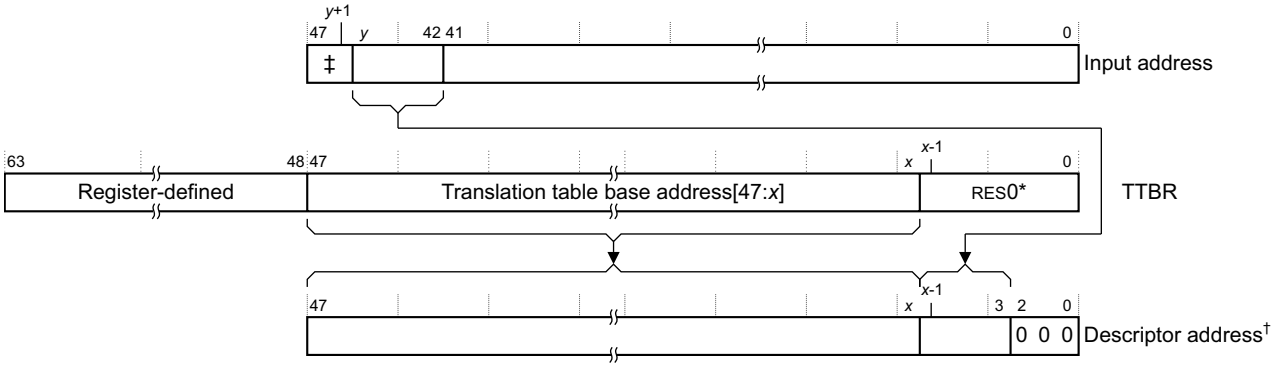
In all cases, for a stage 2 translation, the VTCR_EL2.SL0 field must indicate the required initial lookup level, and this level must be consistent with the value of the VTCR_EL2.T0SZ field, see *Overview of stage 2 translations, 64KB granule* on page D5-2573.

Initial lookup at level 1, 64KB translation granule

This subsection describes initial lookups with an input address width of (n+1) bits, meaning the input address is IA[n:0]. As Table D5-17 on page D5-2573 shows, a stage 1 or stage 2 initial lookup at level 1 is required when $42 \le n \le 47$. For these lookups:

- The size of the translation table is $2^{(n-39)}$ bytes. This means the size of the translation table, at this level, is always less than the granule size. The address of this translation table must align to the size of the table.
- Bits[n:42] of the input address are bits[(n-39):3] of the descriptor offset in the translation table.
- Bits[47:(n-38)] of the TTBR_ELx specify the translation table base address.

Figure K7-7 shows this lookup.



Supported input address range is IA[y:0], $4 \le x$ $y = x + 38$. When y is 47 the field marked ‡ is absent.

† For an EL1&0 stage 1 translation, when EL2 is implemented and enabled in the current Security state, the IPA of the descriptor. Otherwise, the PA of the descriptor.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K7-7 Initial lookup for VMSAv8-64 using the 64KB granule, starting at level 1

Initial lookup at level 2, 64KB translation granule

This subsection describes initial lookups with an input address width of (n+1) bits, meaning the input address is IA[n:0].

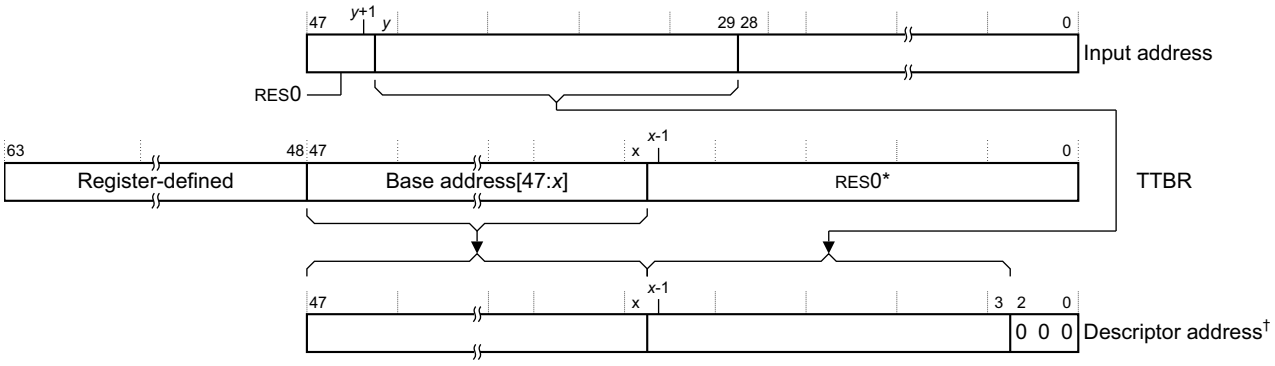
For a stage 1 or stage 2 initial lookup at level 2, without the use of concatenated translation tables

As Table D5-17 on page D5-2573 shows, this applies to IA[n:0], where $29 \le n \le 41$. For these lookups:

- There is a single translation table at this level.
- TTBR_ELx[47:(n-25)] of the specify the translation table base address.
- Bits[n:29] of the input address are bits[(n-26):3] of the descriptor offset in the translation table.

Figure K7-8 on page K7-7932 shows this lookup.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta



Supported input address range is IA[y:0]. $4 \le x \le 16$, $y = x + 25$.

† For an EL1&0 stage 1 translation, when EL2 is implemented and enabled in the current Security state, the IPA of the descriptor. Otherwise, the PA of the descriptor.

* Field has additional properties to the default RES0 definition, see the register description for more information.

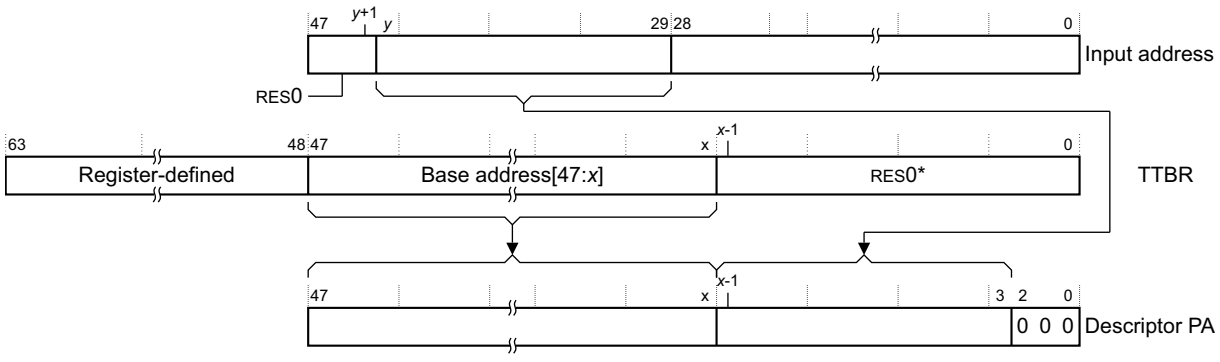
Figure K7-8 Initial lookup for VMSAv8-64 using the 64KB granule, starting at level 2, without concatenation

For a stage 2 initial lookup at level 2, with concatenated translation tables

As Table D5-17 on page D5-2573 shows, this applies to IA[n:0], where $42 \le n \le 45$. For these lookups:

- There are $2^{(m-41)}$ concatenated translation tables at this level.
- These concatenated translation tables must be aligned to $2^{(m-41)} \times 64$KB. This means TTBR_ELx[(n-26):16] must be zero.
- TTBR_ELx[47:(n-25)] specify the base address of the block of translation tables.
- Bits[n:42] of the input address are bits[(n-26):16] of the descriptor offset from the base address of the block of translation tables.

Figure K7-9 shows this lookup.



Supported input address range is IPA[y:0], $4 \le x \le 20$, $y = x + 25$.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K7-9 Initial lookup for VMSAv8-64 using the 64KB granule, starting at level 2, with concatenation

K7.1.2 Full translation flows for VMSAv8-64 address translation

In a translation table walk, only the first lookup uses the translation table base address from the appropriate TTBR_ELx. Subsequent lookups use a combination of address information from:

- The table descriptor read in the previous lookup.
- The input address.

This section describes example full translation flows, from the initial lookup to the address of a memory page. The example flows:

- Resolve the maximum-sized IA range supported by the initial lookup level.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Do not have any concatenation of translation tables.
- Cover only the 4KB and the 64KB translation granules.

Examples of performing the initial lookup on page K7-7926 described how either reducing the IA range or concatenating translation tables affects the initial lookup.

Note

Reducing the IA range or concatenating translation tables affects only the initial lookup.

The following sections describe full VMSAv8-64 translation flows, down to an entry for a memory page:

- *The address and properties fields shown in the translation flows*.
- *Full translation flow using the 4KB granule and starting at level 0*.
- *Full translation flow using the 4KB granule and starting at level 1* on page K7-7935.
- *Full translation flow using the 64KB granule and starting at level 1* on page K7-7936.
- *Full translation flow using the 64KB granule and starting at level 2* on page K7-7937.

The address and properties fields shown in the translation flows

For an EL1&0 stage 1 translation, when EL2 is implemented and enabled in the current Security state:

- Any descriptor address is the IPA of the required descriptor.
- The final output address is the IPA of the block or page.

In these cases, an EL1&0 stage 2 translation is performed to translate the IPA to the required PA.

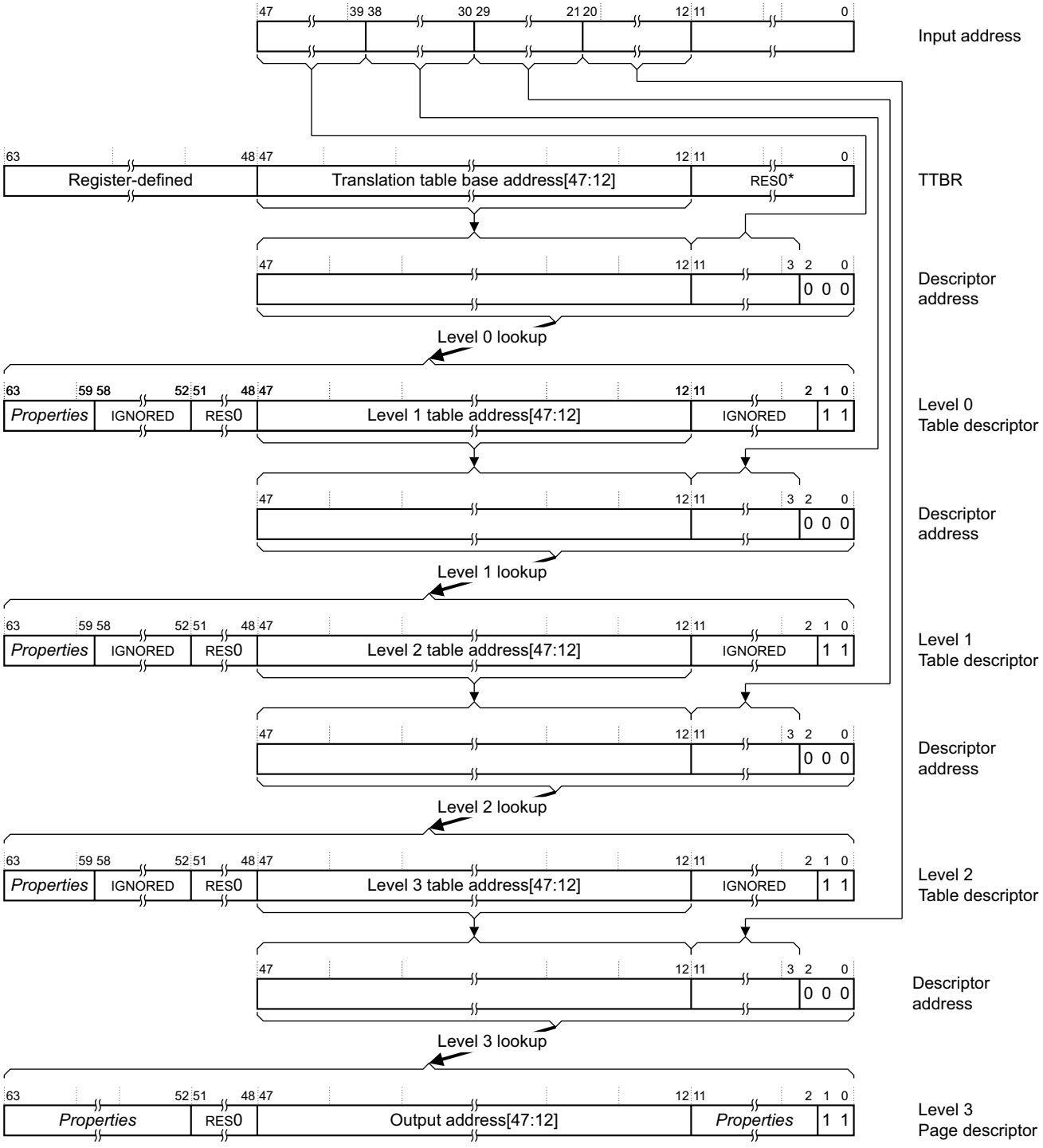
For all other translations, the final output address is the PA of the block or page, and any descriptor address is the PA of the descriptor.

Properties indicates register or translation table fields that return information, other than address information, about the translation or the targeted memory region. For more information see *Memory attribute fields in the VMSAv8-64 translation table format descriptors* on page D5-2599.

Full translation flow using the 4KB granule and starting at level 0

Figure K7-10 on page K7-7934 shows the complete translation flow for a stage 1 translation table walk for a 48-bit input address. This lookup must start with a level 0 lookup. For more information about the fields shown in the figure see *The address and properties fields shown in the translation flows*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta



For details of *Properties* fields, see the register or descriptor description.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K7-10 Complete stage 1 translation of a 48-bit address using the 4KB translation granule

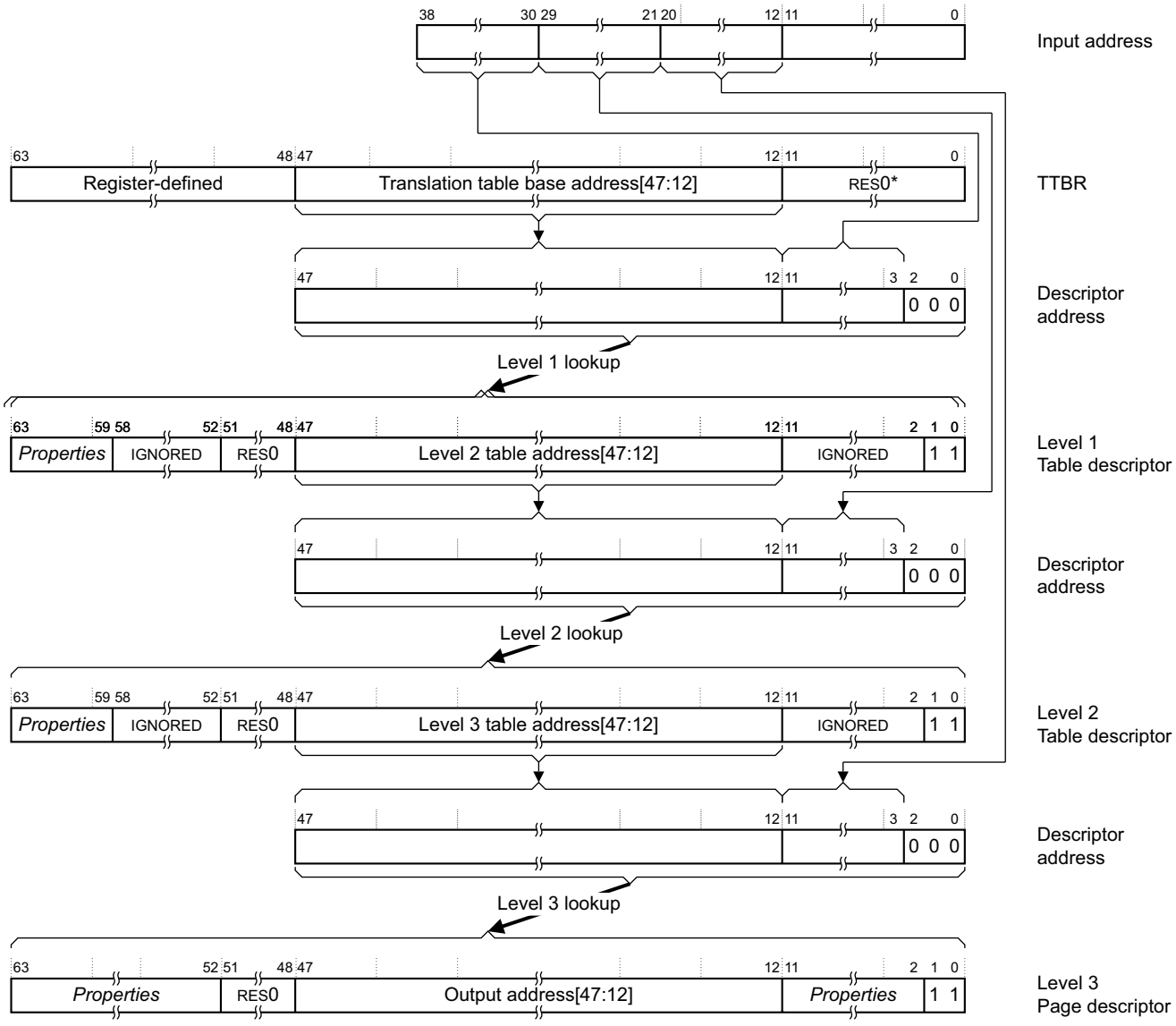
If the level 1 lookup or level 2 lookup returns a block descriptor then the translation table walk completes at that level.

Figure K7-10 shows a stage 1 translation. The only difference for a stage 2 translation is that bits[63:58] of the Table descriptors are SBZ.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Full translation flow using the 4KB granule and starting at level 1

Figure K7-11 shows the complete translation flow for a stage 1 translation table walk for a 39-bit input address. This lookup must start with a level 1 lookup. For more information about the fields shown in the figure see *The address and properties fields shown in the translation flows* on page K7-7933.



For details of *Properties* fields, see the register or descriptor description.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K7-11 Complete stage 1 translation of a 39-bit address using the 4KB translation granule

If the level 1 lookup or the level 2 lookup returns a block descriptor then the translation table walk completes at that level.

Figure K7-11 shows a stage 1 translation. The only difference for a stage 2 translation is that bits[63:58] of the Table descriptors are SBZ.

Comparing this translation with the translation for a 48-bit address, shown in Figure K7-10 on page K7-7934, shows how the translation for the 42-bit address start the same lookup process one stage later.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Full translation flow using the 64KB granule and starting at level 1

Figure K7-10 on page K7-7934 shows the complete translation flow for a stage 1 translation table walk for a 48-bit input address. This lookup must start with a level 0 lookup. For more information about the fields shown in the figure see *The address and properties fields shown in the translation flows* on page K7-7933.

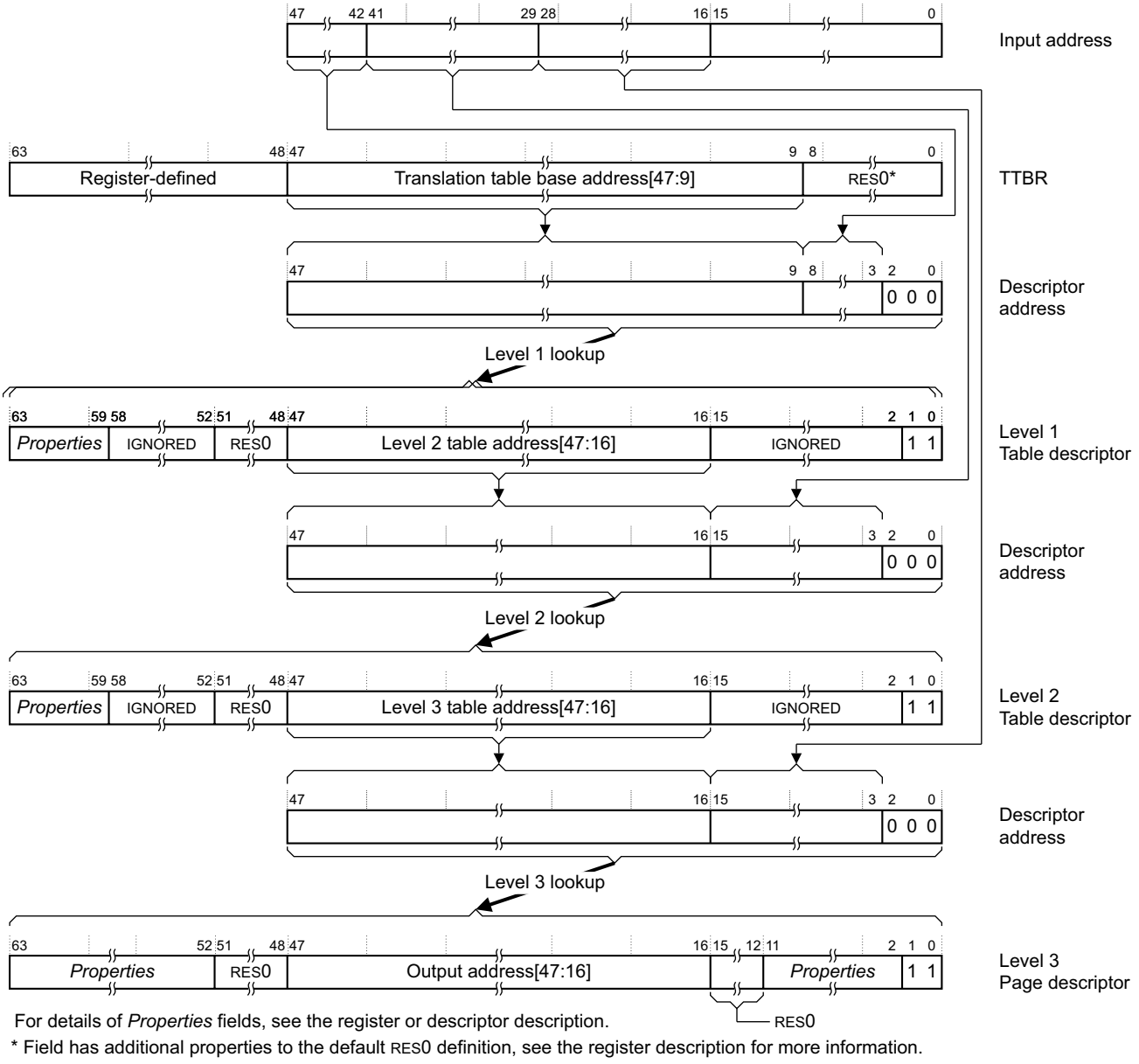


Figure K7-12 Complete stage 1 translation of a 48-bit address using the 64KB translation granule

If the level 2 lookup returns a block descriptor then the translation table walk completes at that level.

Figure K7-12 shows a stage 1 translation. The only difference for a stage 2 translation is that bits[63:58] of the Table descriptors are SBZ.

The level 1 lookup resolves only 6 bits of the input address. As described in *Performing the initial lookup using the 64KB translation granule* on page K7-7930, this means:

- The translation table size for this level is only 512 bytes.
- The required translation table alignment for this level is 512 bytes.
- The Base address field in the TTBR_ELx is extended, at the low-order end, to be bits[47:9].

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Full translation flow using the 64KB granule and starting at level 2

Figure K7-11 on page K7-7935 shows the complete translation flow for a stage 1 translation table walk for a 42-bit input address. This lookup must start with a level 2 lookup. For more information about the fields shown in the figure see *The address and properties fields shown in the translation flows* on page K7-7933.

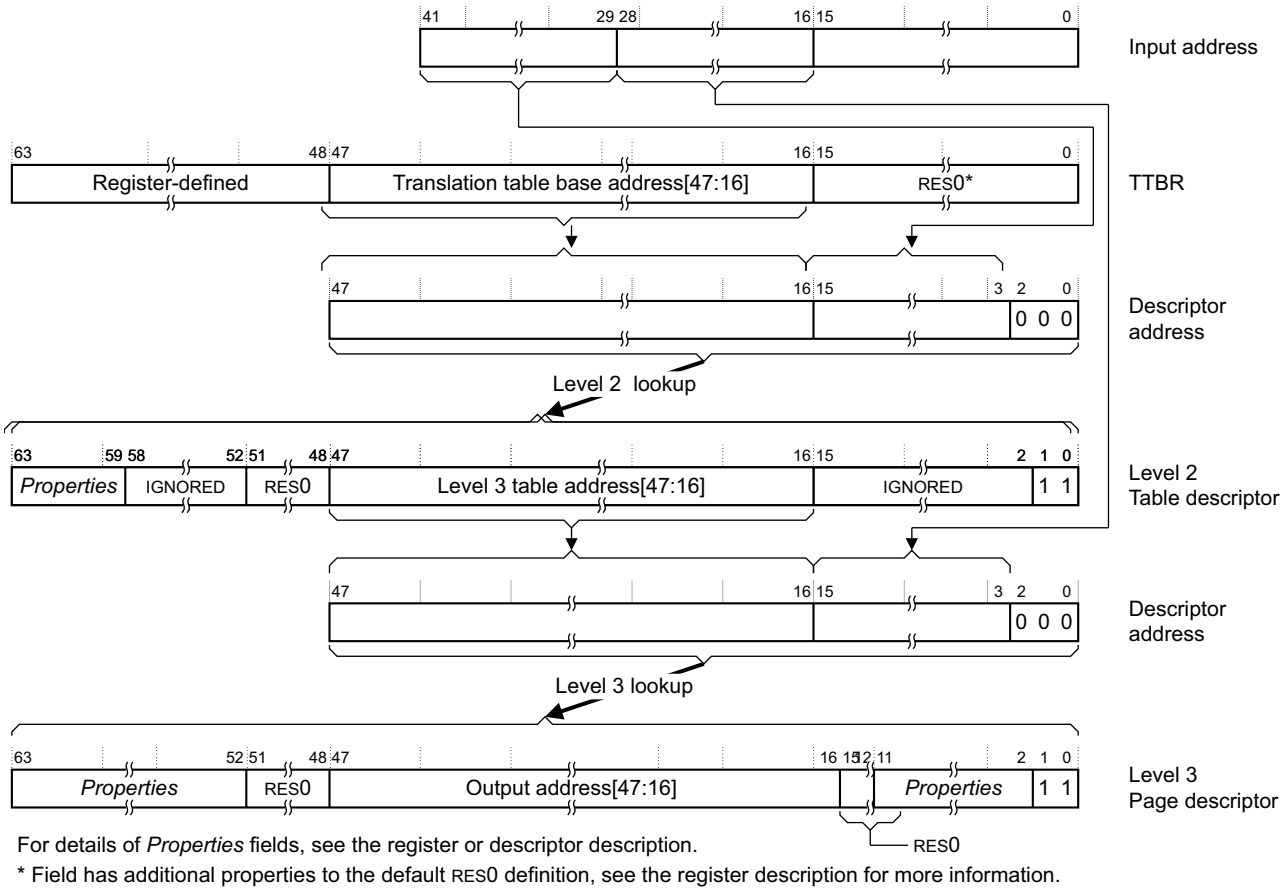


Figure K7-13 Complete stage 1 translation of a 42-bit address using the 64KB translation granule

If the level 2 lookup returns a block descriptor then the translation table walk completes at that level.

Figure K7-13 shows a stage 1 translation. The only difference for a stage 2 translation is that bits[63:58] of the Table descriptors are SBZ.

Comparing this translation with the translation for a 48-bit address, shown in Figure K7-12 on page K7-7936, shows:

- The translation for the 42-bit address starts the same lookup process one stage later.
- Because the initial lookup resolves 13 bits of address:
  - The translation table size for this level is 64KB.
  - The required translation table alignment for this level is 64KB.
  - The Base address field in the TTBR_ELx is bits[47:16].

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K7.2 AArch32 Address translation examples

The following sections give address translation examples for the VMSAv8-32 address translation formats:

- *Address translation examples using the VMSAv8-32 Short descriptor translation table format*.
- *Address translation examples using the VMSAv8-32 Long descriptor translation table format* on page K7-7943.

K7.2.1 Address translation examples using the VMSAv8-32 Short descriptor translation table format

VMSAv8-32 Short-descriptor translation table format descriptors on page G5-5940 describes the memory section and page option for a single VMSAv8-32 address translation. The following sections show the full translation flow for each of these options:

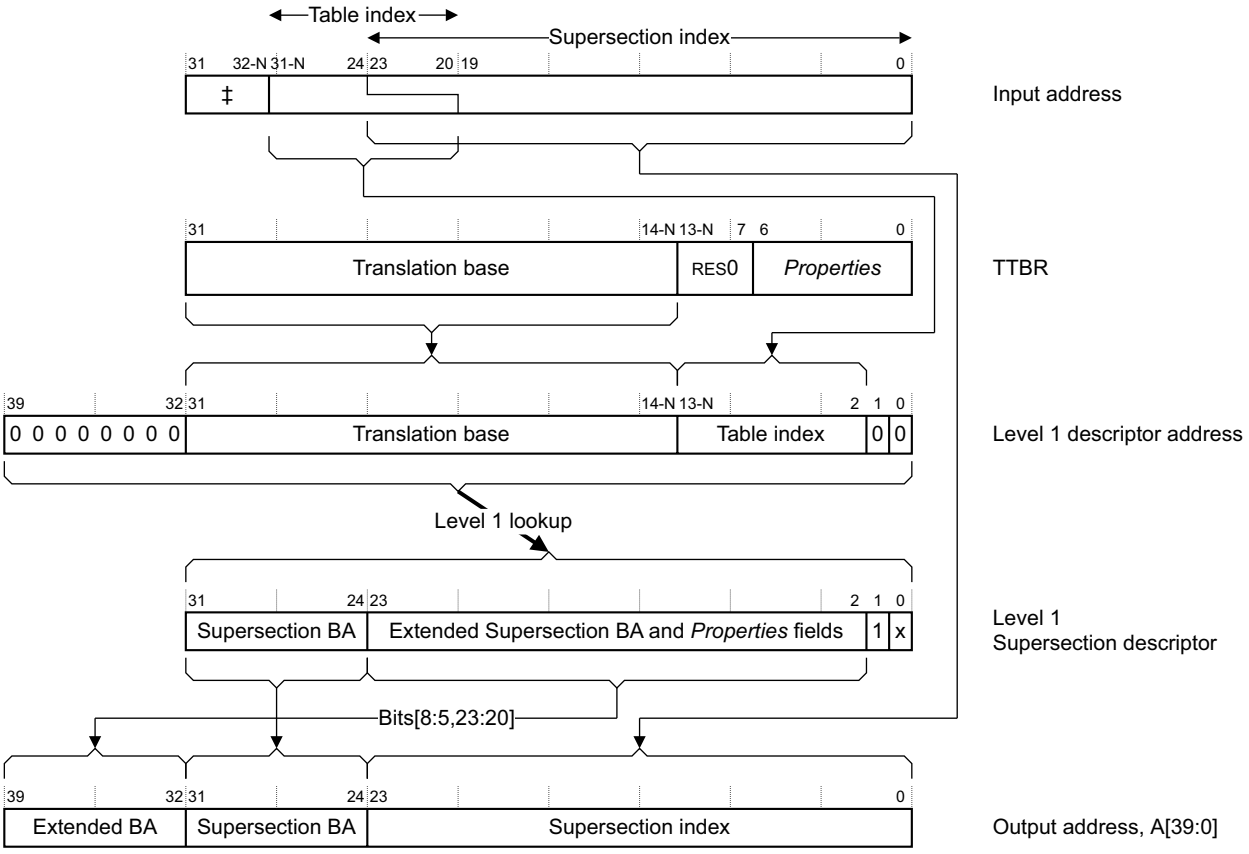
- *Translation flow for a Supersection*.
- *Translation flow for a Section* on page K7-7939.
- *Translation flow for a Large page* on page K7-7940.
- *Translation flow for a Small page* on page K7-7941.

The address and Properties fields shown in the translation flows on page K7-7942 summarizes the information returned by the lookup.

Translation flow for a Supersection

Figure K7-14 on page K7-7939 shows the complete translation flow for a Supersection. For more information about the fields shown in this figure see *The address and Properties fields shown in the translation flows* on page K7-7942.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta



‡ This field is absent if N is 0
BA = Base address
For a translation based on TTBR0, N is the value of TTBCR.N
For a translation based on TTBR1, N is 0
For details of *Properties* fields, see the register or descriptor description

Figure K7-14 VMSAv8-32 Short-descriptor Supersection address translation

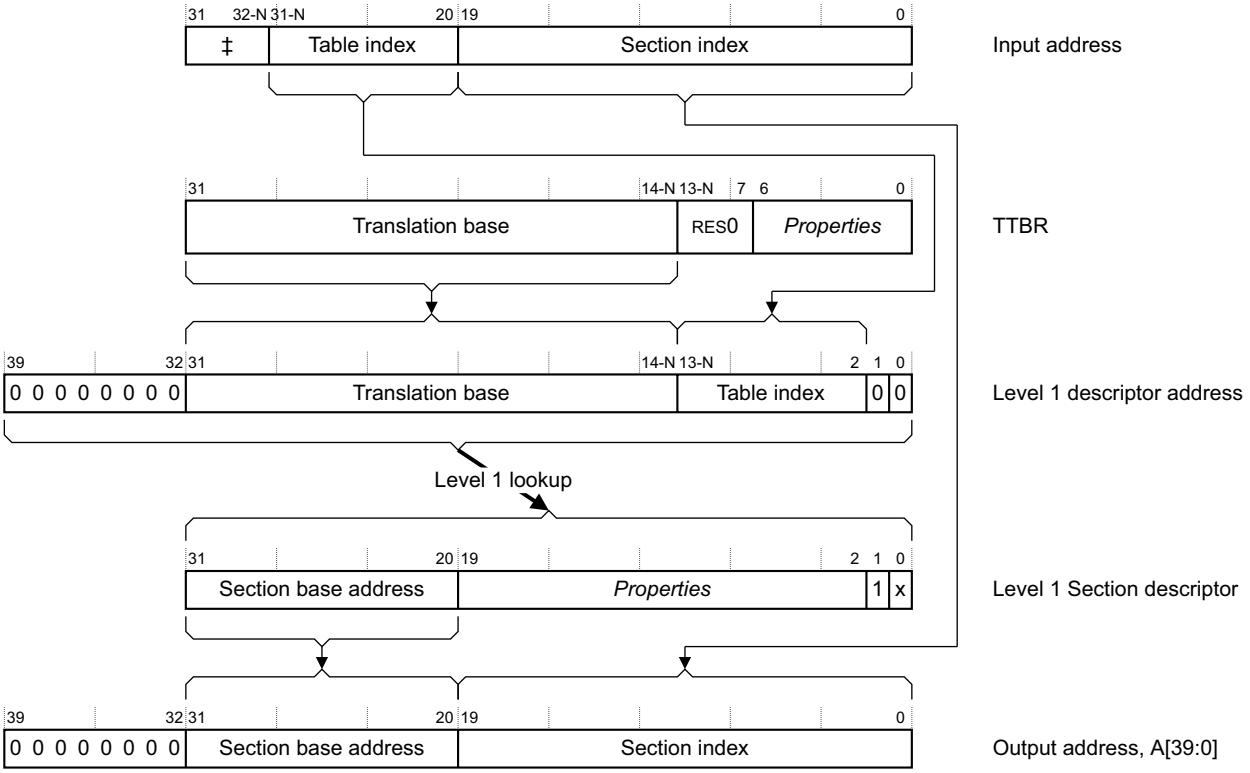
Note

Figure K7-14 shows how, when the input address, the VA, addresses a Supersection, the top four bits of the *Supersection index* bits of the address overlap the bottom four bits of the *Table index* bits. For more information, see *Additional requirements for Short-descriptor format translation tables* on page G5-5943.

Translation flow for a Section

Figure K7-15 on page K7-7940 shows the complete translation flow for a Section. For more information about the fields shown in this figure see *The address and Properties fields shown in the translation flows* on page K7-7942.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta



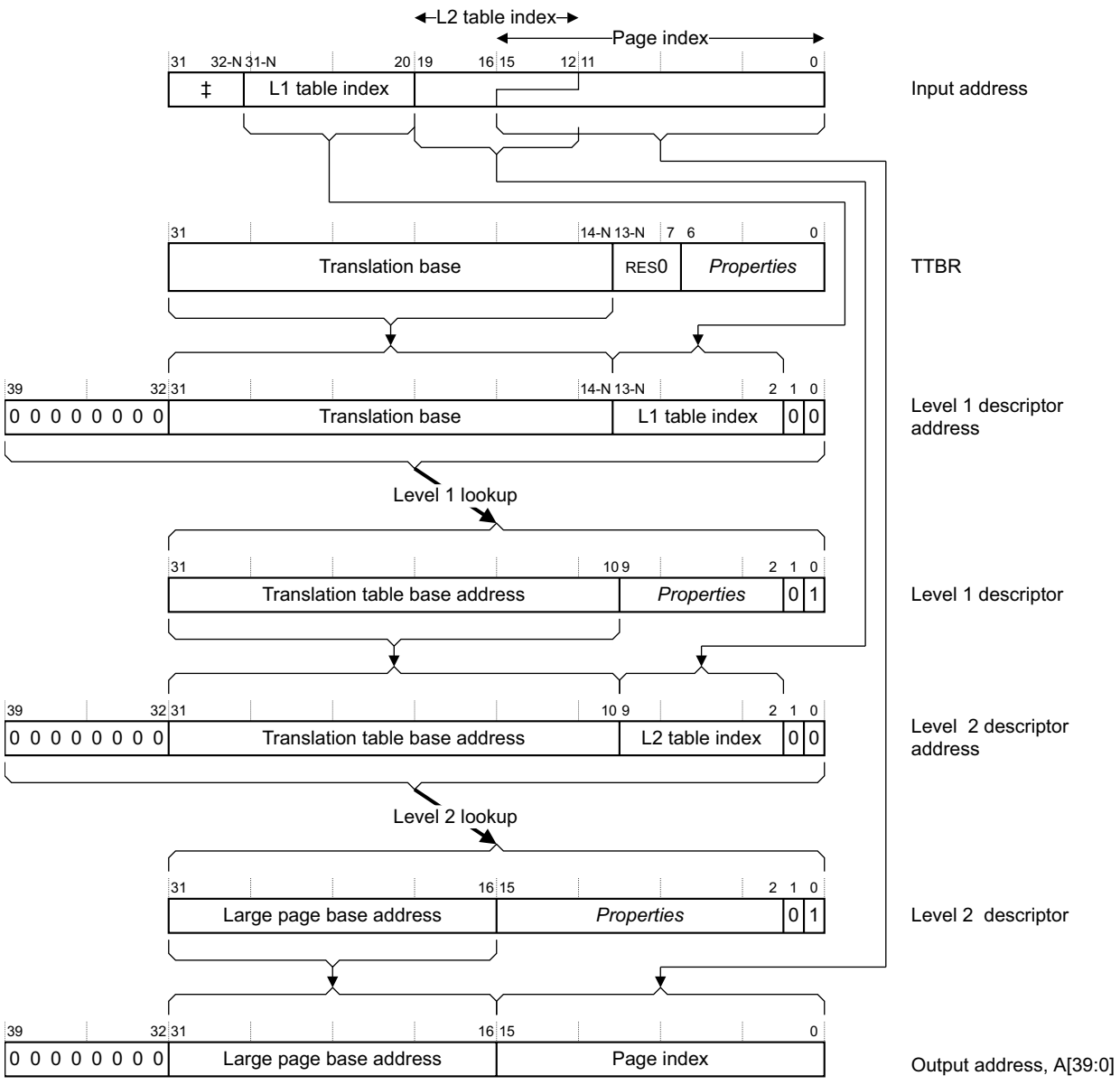
‡ This field is absent if N is 0
For a translation based on TTBR0, N is the value of TTBCR.N
For a translation based on TTBR1, N is 0
For details of *Properties* fields, see the register or descriptor description.

Figure K7-15 VMSAv8-32 Short-descriptor Section address translation

Translation flow for a Large page

Figure K7-16 on page K7-7941 shows the complete translation flow for a Large page. For more information about the fields shown in this figure see *The address and Properties fields shown in the translation flows* on page K7-7942.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta



‡ This field is absent if N is 0
L1 = Level 1, L2 = Level 2
For a translation based on TTBR0, N is the value of TTBCR.N
For a translation based on TTBR1, N is 0
For details of *Properties* fields, see the register or descriptor description

Figure K7-16 VMSAv8-32 Short-descriptor Large page address translation

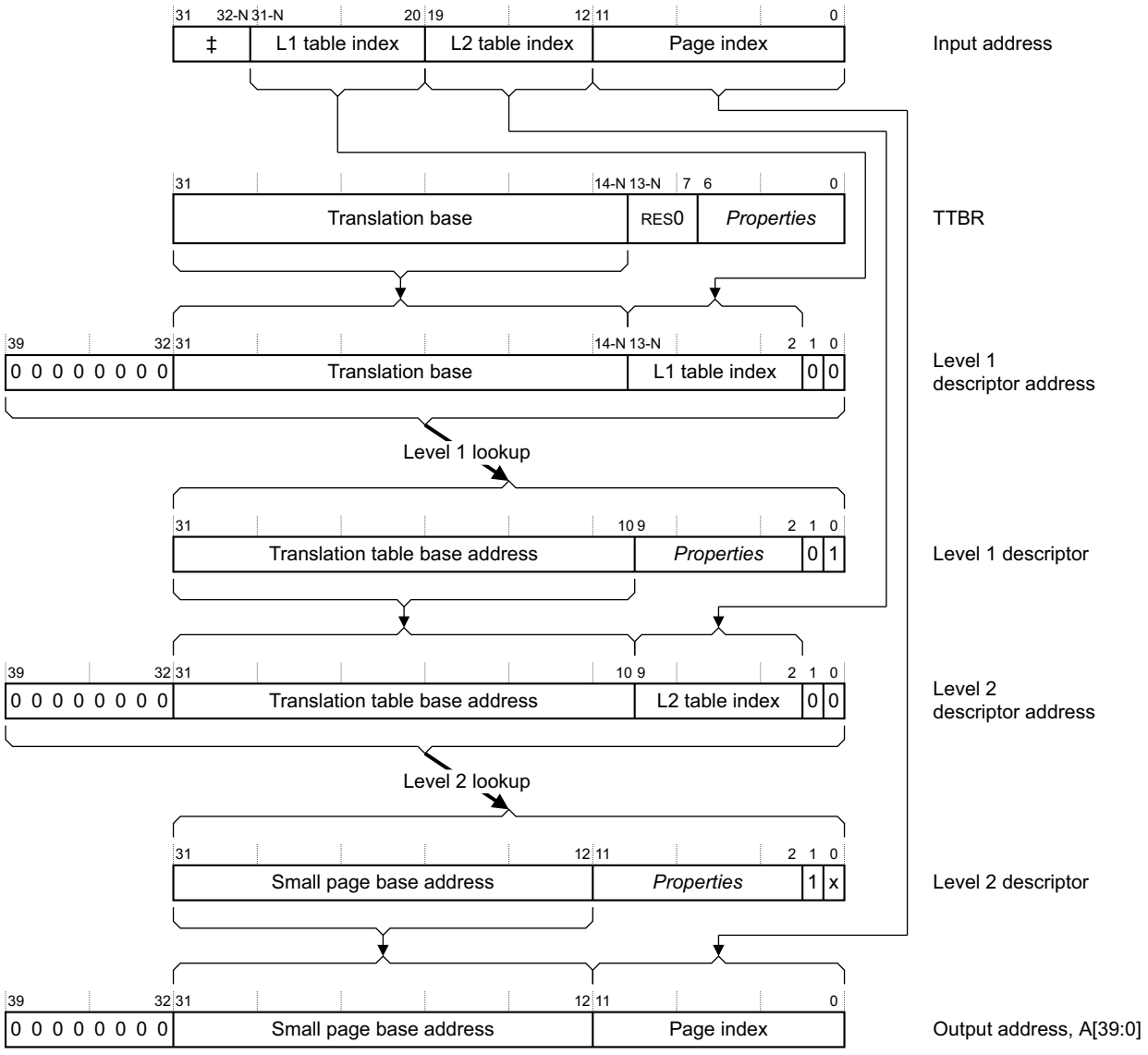
Note

Figure K7-16 shows how, when the input address, the VA, addresses a Large page, the top four bits of the *page index* bits of the address overlap the bottom four bits of the *level 1 table index* bits. For more information, see *Additional requirements for Short-descriptor format translation tables* on page G5-5943.

Translation flow for a Small page

Figure K7-17 shows the complete translation flow for a Small page. For more information about the fields shown in this figure see *The address and Properties fields shown in the translation flows*.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta



‡ This field is absent if N is 0
L1 = Level 1, L2 = Level 2
For a translation based on TTBR0, N is the value of TTBCR.N
For a translation based on TTBR1, N is 0
For details of *Properties* fields, see the register or descriptor description.

Figure K7-17 VMSAv8-32 Short-descriptor Small page address translation

The address and Properties fields shown in the translation flows

For the Non-secure PL1&0 stage 1 translation tables:

- Any descriptor address is the IPA of the required descriptor.
- The final output address is the IPA of the Section, Supersection, Large page, or Small page.

In these cases, a PL1&0 stage 2 translation is performed to translate the IPA to the required PA.

Otherwise, the address is the PA of the descriptor, Section, Supersection, Large page, or Small page.

Properties indicates register or translation table fields that return information, other than address information, about the translation or the targeted memory region. For more information see *Information returned by a translation table lookup* on page G5-5935, and the description of the register or translation table descriptor.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For translations using the Short-descriptor translation table format, *VMSAv8-32 Short-descriptor translation table format descriptors* on page G5-5940 describes the descriptors formats.

K7.2.2 Address translation examples using the VMSAv8-32 Long descriptor translation table format

As described in *Translation table walks, when using the VMSAv8-32 Long-descriptor translation table format* on page G5-5963, in a translation table walk, only the first lookup uses the translation table base address from the appropriate TTBR. Subsequent lookups use a combination of address information from:

- The table descriptor read in the previous lookup.
- The input address.

The following sections give examples of full VMSAv8-32 Long-descriptor format address translation flows, down to an entry for a 4KB page:

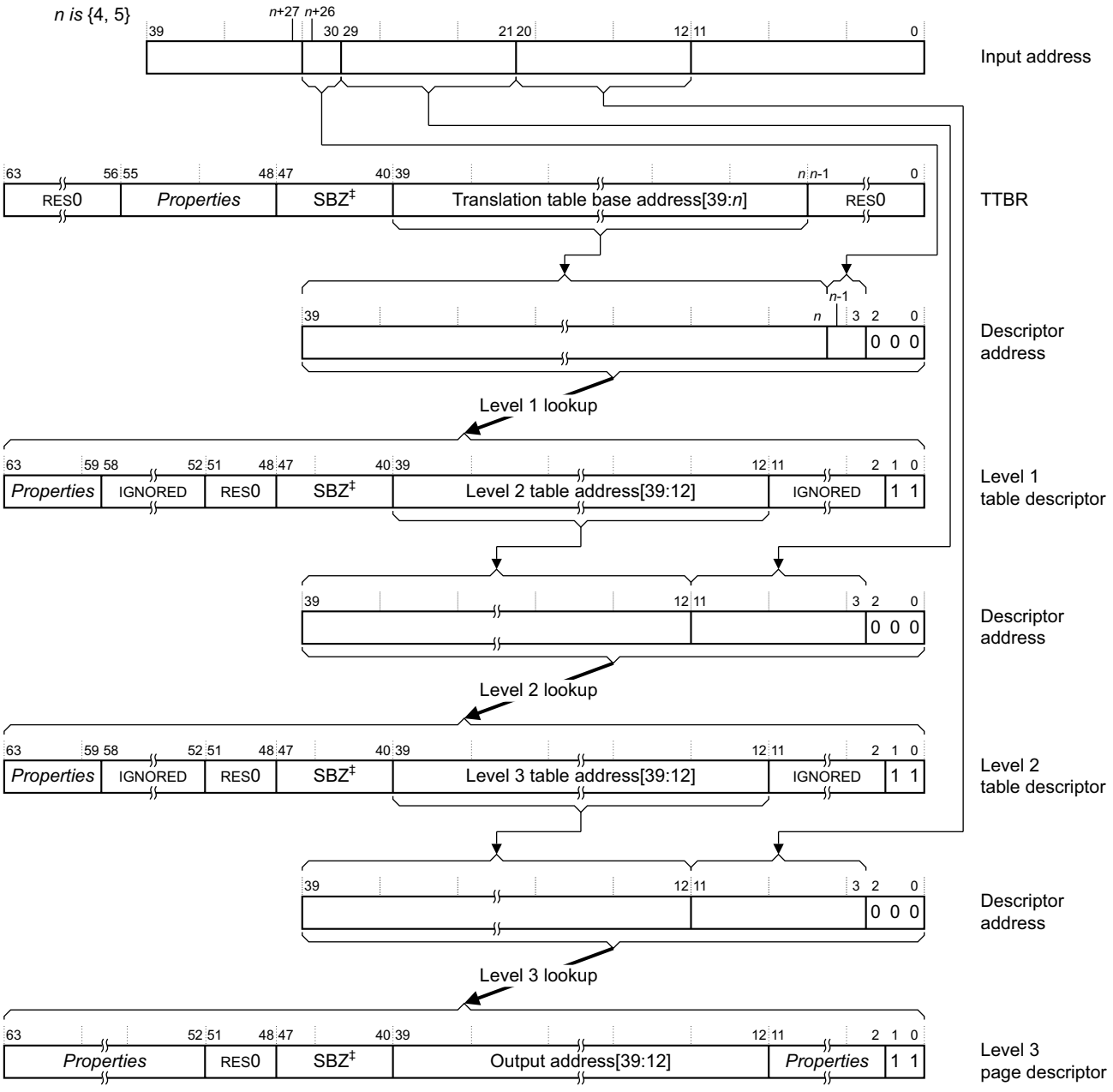
- *Full translation flow, starting at level 1 lookup*.
- *Full translation flow, starting at level 2 lookup* on page K7-7945.

The address and Properties fields shown in the translation flows on page K7-7942 summarizes the information returned by the lookup.

Full translation flow, starting at level 1 lookup

Figure K7-18 on page K7-7944 shows the complete translation flow for a VMSAv8-32 Long-descriptor stage 1 translation table walk that starts with a level 1 lookup. For more information about the fields shown in the figure see *The address and Properties fields shown in the translation flows* on page K7-7942.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta



For details of *Properties* fields, see the register or descriptor description.
‡ See the lookup description for more information about bits[40:47] of the TTBR and descriptors

Figure K7-18 Complete VMSAv8-32 Long-descriptor format stage 1 translation, starting at level 1

If the level 1 lookup or the level 2 lookup returns a block descriptor then the translation table walk completes at that level.

If bits[47:40] of the TTBR or the descriptor are not zero then the lookup will generate an Address size fault, see *Address size fault* on page G5-6014.

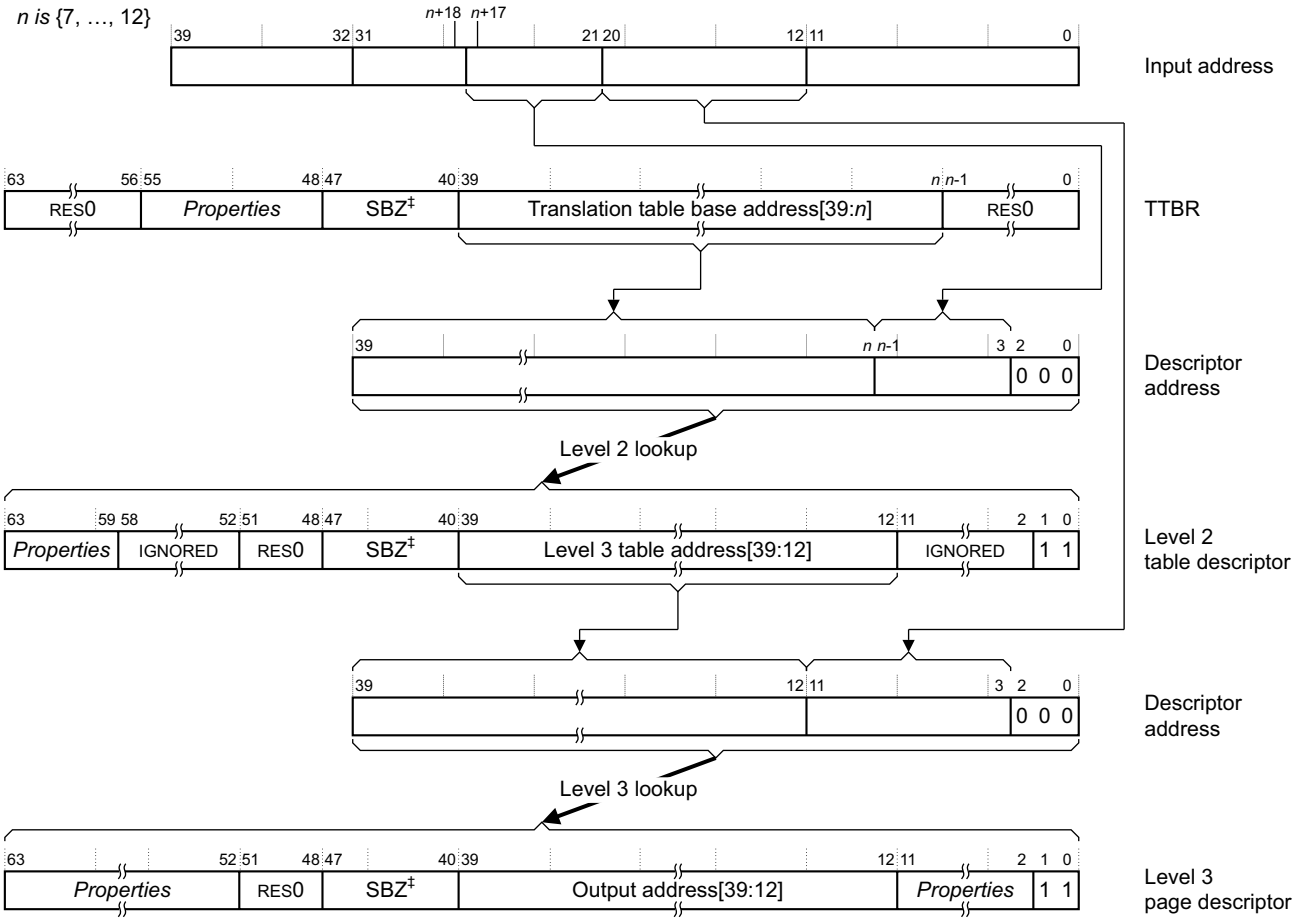
A stage 2 translation that starts at a level 1 lookup differs from the translation shown in Figure K7-18 only as follows:

- The possible values of *n* are 4-13, to support an input address of between 31 and 40 bits.
- A descriptor and output addresses are always PAs.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Full translation flow, starting at level 2 lookup

Figure K7-19 shows the complete translation flow for a stage 1 VMSAv8-32 Long-descriptor translation table walk that starts at a level 2 lookup. For more information about the fields shown in the figure see *The address and Properties fields shown in the translation flows* on page K7-7942.



For details of *Properties* fields, see the register or descriptor description.

‡ See the lookup description for more information about bits[40:47] of the TTBR and descriptors

Figure K7-19 Complete VMSAv8-32 Long-descriptor format stage 1 translation, starting at level 2

If the level 2 lookup returns a block descriptor then the translation table walk completes at that level.

If bits[47:40] of the TTBR or the descriptor are not zero then the lookup will generate an Address size fault, see *Address size fault* on page G5-6014.

A stage 2 translation that starts at a level 2 lookup differs from the translation shown in Figure K7-19 only as follows:

- The possible values of *n* are 7-16, to support an input address of up to 34 bits.
- The descriptor and output addresses are always PAs.

The address and Properties fields shown in the translation flows

For the Non-secure PL1&0 stage 1 translation:

- Any descriptor address is the IPA of the required descriptor.
- The final output address is the IPA of the block or page.

In these cases, a PL1&0 stage 2 translation is performed to translate the IPA to the required PA.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For all other translations, the final output address is the PA of the block or page, and any descriptor address is the PA of the descriptor.

Properties indicates register or translation table fields that return information, other than address information, about the translation or the targeted memory region. For more information see *Information returned by a translation table lookup* on page G5-5935, and the description of the register or translation table descriptor.

For translations using the Long-descriptor translation table format, *VMSAv8-32 Long-descriptor translation table format descriptors* on page G5-5949 describes the descriptors formats.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Appendix K8
Example OS Save and Restore Sequences

This appendix provides possible OS Save and Restore sequences for a v8A Debug implementation. It contains the following sections:

- *Save Debug registers* on page K8-7948.
- *Restore Debug registers* on page K8-7950.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K8.1 Save Debug registers

This section shows how to save the registers that are used by an external debugger.

```
; On entry, X0 points to a block to save the debug registers in.
; Returns the pointer beyond the block and corrupts X1-X3

SaveDebugRegisters
    ; (1) Set OS lock.
    MOV     X2,#1                       ; Set the OS lock. In AArch64 state, the OS lock
    MSR     OSLAR_EL1,X2                ; is writable via OSLAR.
    ISB                                 ; Context synchronization event

    ; (2) Walk through the registers, saving them
    MRS     X1,OSDTRRX_EL1              ; Read DTRRX
    MRS     X2,OSDTRTX_EL1              ; Read DTRTX
    STP     W1,W2,[X0],#8               ; Save { DTRRX, DTRTX }
    MRS     X1,OSECCR_EL1               ; Read ECCR
    MRS     X2,MDSCR_EL1                ; Read DSCR
    STP     W1,W2,[X0],#8               ; Save { ECCR, DSCR }
    [ AARCH32_SUPPORTED
    MRS     X1,DBGVCR32_EL2             ; Read DBGVCR
    MRS     X2,DBGCLAIMCLR_EL1          ; Read CLAIM - note, have to read via CLAIMCLR
    STP     W1,W2,[X0],#8               ; Save { VCR, CLAIM }
    ]

    ;; Macros for saving off a "register pair"
    ;; $WB      is W for watchpoint, B for breakpoint
    ;; $num     is the pair's number
    ;; X0 contains a pointer for the value words
    ;; X1 contains a pointer for the control words
    ;; W2 contains the max index
    MACRO
    SaveRP  $WB,$num, $exit
    MRS     X3,DBG$WB.VR$num._EL1       ; Read DBGxVRn
    STR     X3,[X0],#8                  ; Save { xVRn }
    MRS     X3,DBG$WB.CR$num._EL1       ; Read DBGxCRn
    STR     W3,[X0],#4                  ; Save { xCRn }.
    [ $num > 1 :LAND: $num < 15
    CMP     W1,#$num
    BEQ     $exit
    ]
    MEND

    ; (3) Breakpoints
    MRS     X1,ID_AA64DFR0_EL1
    UBFX    W1,W1,#12,#4                ; Extract BRPs field
    MACRO
    SaveBRP $num                        ; Save a Breakpoint Register Pair
    SaveRP  B,$num,SaveDebugRegisters_Watchpoints
    MEND
    SaveBRP 0
    SaveBRP 1
    SaveBRP 2
    ;; and so on to ...
    SaveBRP 15

SaveDebugRegisters_Watchpoints
    ; (4) Watchpoints
    MRS     X1,ID_AA64DFR0_EL1          ; Read DBGDIDR
    UBFX    W1,W1,#20,#4                ; Extract WRPs field
    MACRO
    SaveWRP $num                        ; Save a Watchpoint Register Pair
    SaveRP  W,$num,SaveDebugRegisters_Exit
    MEND
    SaveWRP 0
    SaveWRP 1
    SaveWRP 2
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    ;; and so on to ...
    SaveWRP 15

SaveDebugRegisters_Exit
    ; (5) Return the pointer to first word not read. This pointer is already in X0, so
    ; all that is needed is to return from this function. The OS double-lock (OSDLR_EL1.DLK) is
    ; locked later, just before the final entry to WFI state.
    RET
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K8.2 Restore Debug registers

This section shows how to restore the registers that are used by an external debugger.

```
; On entry, X0 points to a block of saved debug registers.
; Returns the pointer beyond the block and corrupts R1-R3,R12.

RestoreDebugRegisters
    ; (1) Lock OS lock. The lock will already be set, but this write is included to ensure it
    ; is locked.
    MOV     X2,#1                       ; Lock the OS lock. In AArch64 state, the OS lock
    MSR     OSLAR_EL1,X2                ; is writable via OSLAR.
    ISB                                 ; Context synchronization event

    MSR MDSCR_EL1, XZR                  ; Initialize MDSCR_EL1

    ; (2) Walk through the registers, restoring them
    LDP     W1,W2,[X0],#8               ; Read { DTRRX,DTRTX }
    MSR     OSDTRRX_EL1,X1              ; Restore DTRRX
    MSR     OSDTRTX_EL1,X2              ; Restore DTRTX
    LDP     W1,W3,[X0],#8               ; Read { DSCR, ECCR }
    MSR     OSECCR_EL1,X2               ; Restore ECCR
    [ AARCH32_SUPPORTED
    LDP     W1,W2,[X0],#8               ; Read { VCR,CLAIM }
    MSR     DBGVCR32_EL2,X1             ; Restore DBGVCR
    MSR     DBGCLAIMSET_EL1,X2          ; Restore CLAIM - note, writes CLAIMSET
    ]

    ;; Macro for restoring a "register pair"
    MACRO
    RestoreRP $WB,$num,$exit
    LDR     X3,[X0],#8                  ; Read { xVRn }
    MSR     DBG$WB.VR$num._EL1,X3       ; Restore DBGxVRn
    LDR     W3,[X0],#4                  ; Read { xCRn }
    MSR     DBG$WB.CR$num._EL1,X3       ; Restore DBGxCRn
    [ $num >= 1 :LAND: $num < 15
    CMP     W1,#$num
    BEQ     $exit
    ]
    MEND

    ; (3) Breakpoints
    MRS     X1,ID_AA64DFR0_EL1
    UBFX    W1,W1,#12,#4                ; Extract BRPs field
    MACRO
    RestoreBRP $num                     ; Restore a Breakpoint Register Pair
    RestoreRP  B,$num,RestoreDebugRegisters_Watchpoints
    MEND
    RestoreBRP 0
    RestoreBRP 1
    RestoreBRP 2
    ;; and so on until ...
    RestoreBRP 15

RestoreDebugRegisters_Watchpoints
    ; (4) Watchpoints
    MRS     X1,ID_AA64DFR0_EL1          ; Read DBGDIDR
    UBFX    W1,W1,#20,#4                ; Extract WRPs field
    MACRO
    RestoreWRP $num                     ; Restore a Watchpoint Register Pair
    RestoreRP  W,$num,RestoreDebugRegisters_Exit
    MEND
    RestoreWRP 0
    RestoreWRP 1
    RestoreWRP 2
    ;; and so on until ...
    RestoreWRP 15
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
RestoreDebugRegisters_Exit
    MSR MDSCR_EL1, X3                   ; Restore DSCR

    ; (5) Clear the OS lock.
    ISB
    MOV     X2,#0                       ; Clear the OS lock. In AArch64 state, the OS lock
    MSR     OSLAR_EL1,X2                ; is writable via OSLAR.

    ; (6) A final ISB guarantees the restored register values are visible to subsequent
    ; instructions.
    ISB

    ; (7) Return the pointer to first word not read. This pointer is already in X0, so
    ; all that is needed is to return from this function.
    RET
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Appendix K9
Recommended Upload and Download Processes for External Debug

This appendix contains the following section:

- *Using memory access mode in AArch64 state* on page K9-7954.

Note

This description is not part of the Arm architecture specification. It is included here as supplementary information, for the convenience of developers and users who might find this information useful.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K9.1 Using memory access mode in AArch64 state

Figure K9-1 and Figure K9-2 on page K9-7955 show the processes for using memory access mode to implement a download (external host to target) and an upload (target to external host).

To transfer *n* words of data:

- The download sequence needs *n*+6 accesses by the external debug interface.
- The upload sequence needs *n*+8 accesses by the external debug interface.

In both cases, in the innermost loop the debugger can make an external access to a DTR without polling EDSCR after each write as underrun and overrun detection prevent failure. Normally external accesses from the debugger are outpaced by the memory accesses of the PE, making underruns and overruns unlikely. If this is not the case, the EDSCR.ERR flag is set to 1. This is checked once at the end of the sequence, although a debugger can check it more often, for example once for each page. If the EDSCR.ERR flag is set to 1 because of overrun or underrun, the debugger can restart. The address to restart from is frozen in X0. EDSCR.ERR might also be set because of a Data abort.

If underruns and overruns are common, the debugger can pace itself accordingly.

Note

- The base address must be a multiple of 4.
- The order of the writes that set up the address does not matter in Debug state.

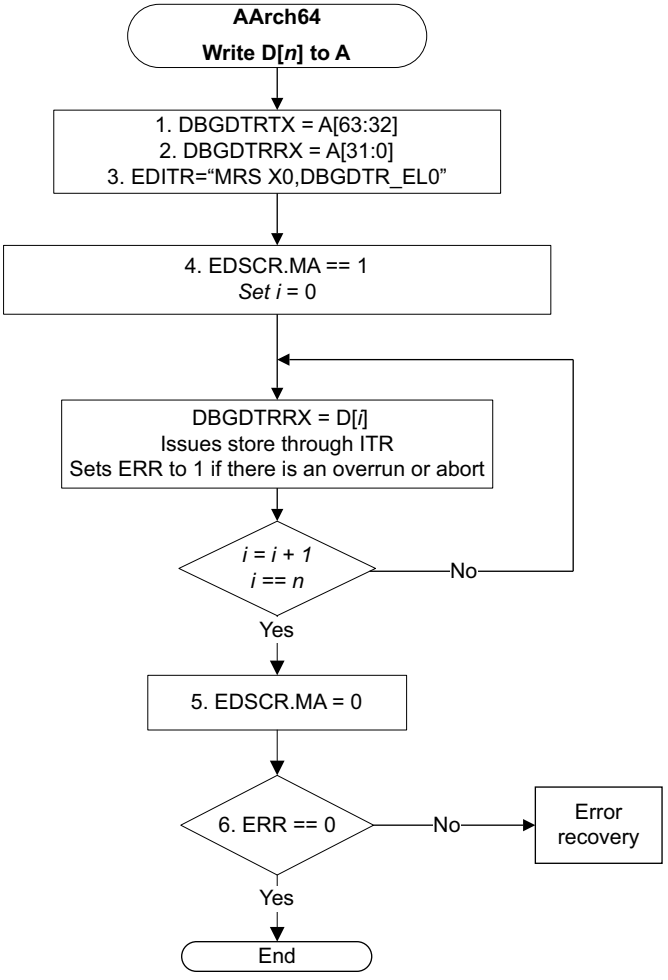


Figure K9-1 Fast code download in AArch64 state (external host to target)

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In Figure K9-1 on page K9-7954, the sequence for the fast code download is as follows:

1. Setup. From the external debug interface:
   a. Write address [31:0] to DBGDTRRX_EL0.
   b. Write address [63:32] to DBGDTRTX_EL0.
   c. Write `MRS X0, DBGDTR_EL0` to EDITR. The PE executes this instruction.
   d. Set EDSCR.MA to 1.
2. Loop *n* times. From the external debug interface:
   a. Write to DBGDTRRX_EL0. The PE reads the word from DTRRX and stores it to memory. It increments X0 by 4.
3. Epilogue. From the external debug interface:
   a. Clear EDSCR.MA to 0.
   b. Read EDSCR to check for overruns or Data Aborts during download.

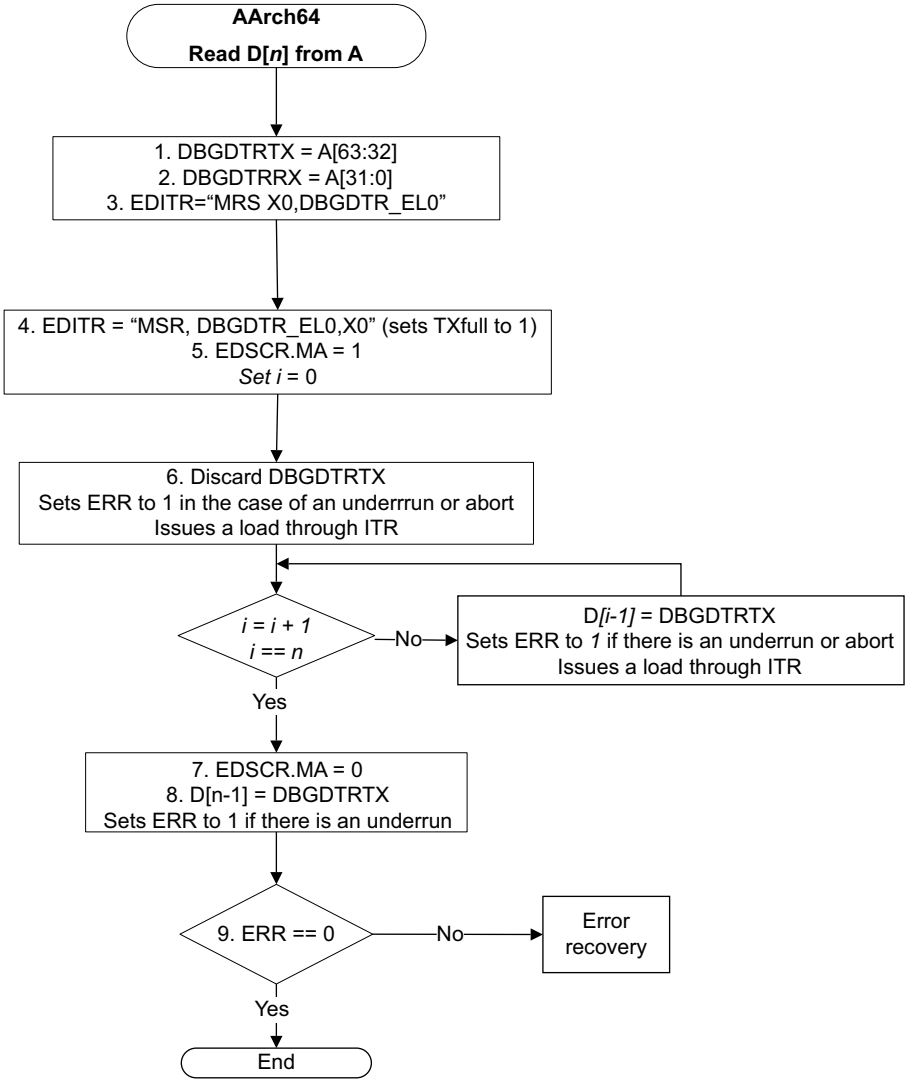


Figure K9-2 Fast data upload in AArch64 state (target to external host)

In Figure K9-2, the sequence for the fast code download is as follows:

1. Setup. From the external debug interface:
   a. Write address [31:0] to DBGDTRRX_EL0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

   b. Write address [63:32] to DBGDTRTX_EL0.
   c. Write `MRS X0, DBGDTR_EL0` to EDITR.
   d. Write `MSR DBGDTR_EL0, X0` to EDITR. This dummy operation ensures EDSCR.TXfull == 1.
   e. Set EDSCR.MA to 1.
   f. Read DBGDTRTX_EL0 and discard the value. The PE returns the previous DTR value, loads the first word, and writes it to DTR. It increments X0 by 4.

2. Loop *n*-1 times. From the external debug interface:
   a. Read DBGDTRTX_EL0. The PE returns the previous DTRTX value, loads a new word, and writes it to DTRTX. It increments X0 by 4.

3. Epilogue. From the external debug interface:
   a. Clear EDSCR.MA to 0.
   b. Read DBGDTRTX_EL0 for the nth value.
   c. Read EDSCR to check for underruns, overruns or Data Aborts during upload.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Appendix K10
Software Usage Examples

This appendix gives software usage examples, for cases where these are likely to contribute significantly to an understanding of the Arm architecture.

It contains the following sections:

- *Use of the Advanced SIMD complex number instructions* on page K10-7958.
- *Use of the Armv8.2 extensions to the Cryptographic Extension* on page K10-7960.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K10.1 Use of the Advanced SIMD complex number instructions

ARMv8.3-CompNum provides instructions to aid floating-point computations of complex numbers. This section illustrates the use of these instructions for complex arithmetic. It is not part of the Arm architecture definition.

This section uses the AArch64 instructions `FCADD` and `FCMLA` - usage of the AArch32 instructions `VCADD` and `VCMLA` is similar.

When using the instructions implemented by ARMv8.3-CompNum, a complex numbers is represented in a SIMD&FP register as a pair of adjacent elements, each holding a floating-point number, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number.

K10.1.1 Complex addition

Simple complex addition on a vector of complex numbers is already provided by the vector form of the `FADD` instruction.

The functionality that `FCADD` adds is to rotate each complex number in the second vector by 90 degrees or 270 degrees counterclockwise (considering the complex numbers on an Argand diagram) before performing the addition. Mathematically, this is equivalent to multiplying the second complex number by i or -i before addition.

This means, given a complex number z stored in a pair of elements in one vector, and a complex number w stored in the corresponding element pair in another vector:

- `FADD` calculates z + w.
- `FCADD` calculates z ± iw.

K10.1.2 Complex multiplication

The `FCMLA` instruction does not provide functionality for complex multiplication directly. However, a pair of `FCMLA` instructions can provide this function.

The `FCMLA` instruction operates on corresponding pairs of complex numbers stored in SIMD&FP vector registers, and adds the result to the corresponding complex number in the destination SIMD&FP vector register. This computation is as follows:

1. The second complex number is rotated by 0, 90, 180 or 270 degrees counterclockwise.
2. That complex number is multiplied by either the real or imaginary part of the first complex number:
   - When the rotation is 0 or 180 degrees, the real part is used.
   - When the rotation is 90 or 270 degrees, the imaginary part is used.
3. The resulting complex number is added to the corresponding complex number in the destination register.

Mathematically, considering the complex numbers on an Argand diagram:

- Rotation by 180 degrees is equivalent to negation.
- Rotation by 90 degrees is equivalent to multiplying by i.
- Rotation by 270 degrees is equivalent to multiplying by -i.

This means that, for a first complex number z, where z = a+bi, and a second complex number w, if initially the corresponding complex number in the destination register is zero:

- When the rotation is 0 degrees the result of the multiply-add is aw.
- When the rotation is 180 degrees, the result is -aw.
- When the rotation is 90 degrees, the result is biw.
- When the rotation is 270 degrees, the result is -biw.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This means that, if the destination register is zeroed and an `FCMLA` instruction is executed with a rotation parameter of 0, and then the same instruction is executed with a rotation parameter of 90:

- The first execution returns aw in the destination register.
- The second execution accumulates biw to this, meaning the result is aw+biw.
- This result is the product of (a+bi)w, which is the product zw.

So, this pair of instructions can be used to implement complex multiplication.

After zeroing V0, the syntax of a pair of instructions to perform this complex number multiplication might be:

```
FCMLA V0.4S, V1.4S, V2.4S, #0
FCMLA V0.4S, V1.4S, V2.4S, #90
```

Other simple pairs of `FCMLA` instructions perform useful computations. For example, considering a first complex number z and second complex number w, defined as before, and a destination register that has been zeroed before the first `FCMLA` instruction is executed:

1. The following pair of instructions calculates the complex conjugate of z multiplied by w.

   ```
   FCMLA V0.4S, V1.4S, V2.4S, #0
   FCMLA V0.4S, V1.4S, V2.4S, #270
   ```

2. The following pair of instructions calculates the negation of z multiplied by w.

   ```
   FCMLA V0.4S, V1.4S, V2.4S, #180
   FCMLA V0.4S, V1.4S, V2.4S, #270
   ```

3. The following pair of instructions calculates the negation of the complex conjugate of z multiplied by w.

   ```
   FCMLA V0.4S, V1.4S, V2.4S, #180
   FCMLA V0.4S, V1.4S, V2.4S, #90
   ```

Note

For these examples, the following caveats must be considered:

- `FCMLA` performs a fused multiply-add, meaning there is no intermediate rounding. This lack of intermediate rounding can give unexpected results in some cases. Arm expects that these instructions are only used in situations where the effect of the rounding of these results is not material to the calculation.
- When using the `FCMLA` instructions, the behavior of (∞+∞i) multiplied by (0+i) is (NaN+NaNi), rather than the result expected by ISO C, which is complex ∞.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K10.2 Use of the Armv8.2 extensions to the Cryptographic Extension

K10.2.1 Use of the SHA512 instructions

These instructions are implemented when ARMv8.2-SHA is implemented.

The following code sequence shows the use of the SHA512 instructions to calculate a SHA512 hash iteration of 80 rounds. This code is not fully optimized.

```
// X0 contains the pointer to the bottom of the (padded) 16*64 bytes of message to be
// hashed, with space above the that message to hold a further 64 * 64 bytes of working
// data
// X1 contains the pointer to the 0th element of 80 64-bit constants (in ascending addresses) defined in
the SHA2 specification
// X2 contains a loop variable
// V4,V5,V6, V7 hold VS0 to VS3 respectively
// V8 holds running hash V1
// V9 holds running hash V0
    MOV        X2, #0
loop1:
  LD1        {V0.2D}, [X0]                // Data
  LD1        {V1.2D}, [X1]                // K values
  ADD        X1, X1, #16
  ADD        X0, X0, #16
  ADD        X2, X2, #16
  ADD        V2.2D, V0.2D, V1.2D
  EXT        V2.16B, V2.16B, V2.16B, #8
  EXT        V8.16B, V6.16B, V7.16B, #8
  EXT        V9.16B, V5.16B, V6.16B, #8
  ADD        V7.2D, V7.2D, V2.2D
  SHA512H    Q7, Q8, V9.2D
  ADD        V10.2D, V5.2D, V7.2D
  SHA512H2   Q7, Q5, V4.2D
  MOV        V5.16B, V4.16B
  MOV        V4.16B, V7.16B
  MOV        V7.16B, V6.16B
  MOV        V6.16B, V10.16B
  CMP        X2, #128
  BLT        loop1

  // work out pointers to previous words in the data
  SUB       X3, X0, #128
  SUB       X4, X0, #112
  SUB       X5, X0, #16
  SUB       X6, X0, #56
loop2:
  LD1      {V11.2D}, [X3]
  LD1      {V12.2D}, [X4]
  LD1      {V13.2D}, [X5]
  LD1      {V14.2D}, [X6]
  SHA512SU0 V11.2D, V12.2D
  SHA512SU1 V11.2D, V13.2D, V14.2D
  ST1      {V11.2D}, [X0]
  LD1      {V1.2D}, [X1]                    // K values
  ADD      X0, X0, #16
  ADD      X1, X1, #16
  ADD      X3, X3, #16
  ADD      X4, X4, #16
  ADD      X5, X5, #16
  ADD      X6, X6, #16
  ADD      X2, X2, #16
  ADD      V2.2D, V11.2D, V1.2D
  EXT      V2.16B, V2.16B, V2.16B, #8
  EXT      V8.16B, V6.16B, V7.16B, #8
  EXT      V9.16B, V5.16B, V6.16B, #8
  ADD      V7.2D, V7.2D, V2.2D
  SHA512H  Q7, Q8, V9.2D
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
ADD      V10.2D, V5.2D, V7.2D
SHA512H2 Q7, Q5, V4.2D
MOV      V5.16B, V4.16B
MOV      V4.16B, V7.16B
MOV      V7.16B, V6.16B
MOV      V6.16B, V10.16B
CMP      X2, #320
BLT      loop2
```

K10.2.2 Use of the SHA3 instructions

These instructions are implemented when ARMv8.2-SHA is implemented.

The following code sequence shows the use of the SHA3 instructions to obtain the combined theta, phi, rho and chi operations of a SHA3 iteration. Arm expects the iota operation to be performed using a lookup table.

This code is not fully optimized for multiple iterations.

```
// Input State:
    //      x=0   x=1   x=2   x=3   x=4
    // y=0  v12   v13   v14   v10   v11
    // y=1   v7    v8    v9    v5    v6
    // y=2   v2    v3    v4    v0    v1
    // y=3  v22   v23   v24   v20   v21
    // y=4  v17  v18    v19   v15   v16

    //- Theta Calculations -//
    eor3 v25.16B, v12.16B, v7.16B,  v2.16B
    eor3 v25.16B, v25.16B, v22.16B, v17.16B
    eor3 v26.16B, v13.16B, v8.16B,  v3.16B
    eor3 v26.16B, v26.16B, v23.16B, v18.16B
    eor3 v27.16B, v14.16B, v9.16B,  v4.16B
    eor3 v27.16B, v27.16B, v24.16B, v19.16B
    eor3 v28.16B, v10.16B, v5.16B,  v0.16B
    eor3 v28.16B, v28.16B, v20.16B, v15.16B
    eor3 v29.16B, v11.16B, v6.16B,  v1.16B
    eor3 v29.16B, v29.16B, v21.16B, v16.16B

    rax1 v30.2D, v29.2D, v26.2D
    rax1 v31.2D, v27.2D, v29.2D
    rax1 v29.2D, v25.2D, v27.2D
    rax1 v27.2D, v28.2D, v25.2D
    rax1 v25.2D, v26.2D, v28.2D

    //- Phi\rho Stage -//
    eor v12.8B, v12.8B, v30.8B
    xar v26.2D, v21.2D, v27.2D, #56
    xar v21.2D, v15.2D, v31.2D, #8
    xar v15.2D, v22.2D, v30.2D, #23
    xar v22.2D, v11.2D, v27.2D, #37
    xar v11.2D, v16.2D, v27.2D, #50
    xar v16.2D, v18.2D, v29.2D, #62
    xar v18.2D, v5.2D,  v31.2D, #9
    xar v5.2D,  v23.2D, v29.2D, #19
    xar v23.2D, v7.2D,  v30.2D, #28
    xar v7.2D,  v10.2D, v31.2D, #36
    xar v10.2D, v20.2D, v31.2D, #43
    xar v20.2D, v24.2D, v25.2D, #49
    xar v24.2D, v3.2D,  v29.2D, #54
    xar v3.2D,  v9.2D,  v25.2D, #58
    xar v9.2D,  v2.2D,  v30.2D, #61
    xar v2.2D,  v13.2D, v29.2D, #63
    xar v13.2D, v8.2D,  v29.2D, #20
    xar v8.2D,  v6.2D,  v27.2D, #44
    xar v6.2D,  v19.2D, v25.2D, #3
    xar v19.2D, v1.2D,  v27.2D, #25
    xar v1.2D,  v17.2D, v30.2D, #46
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
xar v17.2D, v14.2D, v25.2D, #2
xar v14.2D, v4.2D,  v25.2D, #21
xar v4.2D,  v0.2D,  v31.2D, #39

// XAR Output:
//
// v12  v2 v17  v7 v22
// v23 v13  v3 v18  v8
//  v9 v24 v14  v4 v19
// v15  v5 v20 v10 v26
//  v1 v16  v6 v21 v11
//
// temp: v0, v25, v27, v28, v29, v30, v31

// Phi Output:
//
// v12 v13 v14 v10 v11
//  v7  v8  v9  v5  v6
//  v2  v3  v4 v26  v1
// v22 v23 v24 v20 v21
// v17 v18 v19 v15 v16

//- Chi transformations -//
bcax v31.16B, v26.16B, v2.16B,  v1.16B
bcax v27.16B, v1.16B,  v3.16B,  v2.16B
bcax v28.16B, v2.16B,  v4.16B,  v3.16B
bcax v29.16B, v3.16B,  v26.16B, v4.16B
bcax v30.16B, v4.16B,  v1.16B,  v26.16B
bcax v0.16B,  v5.16B,  v7.16B,  v6.16B
bcax v1.16B,  v6.16B,  v8.16B,  v7.16B
bcax v2.16B,  v7.16B,  v9.16B,  v8.16B
bcax v3.16B,  v8.16B,  v5.16B,  v9.16B
bcax v4.16B,  v9.16B,  v6.16B,  v5.16B
bcax v5.16B,  v10.16B, v12.16B, v11.16B
bcax v6.16B,  v11.16B, v13.16B, v12.16B
bcax v7.16B,  v12.16B, v14.16B, v13.16B
bcax v8.16B,  v13.16B, v10.16B, v14.16B
bcax v9.16B,  v14.16B, v11.16B, v10.16B
bcax v10.16B, v15.16B, v17.16B, v16.16B
bcax v11.16B, v16.16B, v18.16B, v17.16B
bcax v12.16B, v17.16B, v19.16B, v18.16B
bcax v13.16B, v18.16B, v15.16B, v19.16B
bcax v14.16B, v19.16B, v16.16B, v15.16B
bcax v15.16B, v20.16B, v22.16B, v21.16B
bcax v16.16B, v21.16B, v23.16B, v22.16B
bcax v17.16B, v22.16B, v24.16B, v23.16B
bcax v18.16B, v23.16B, v20.16B, v24.16B
bcax v19.16B, v24.16B, v21.16B, v20.16B

// Output State from Chi:
//
//      x=0   x=1   x=2   x=3   x=4
// y=0   v7    v8    v9    v5    v6
// y=1   v2    v3    v4    v0    v1
// y=2  v28   v29   v30   v31   v27
// y=3  v17   v18   v19   v15   v16
// y=4  v12   v13   v14   v10   v11
```

K10.2.3 Use of the SM3 instructions

These instructions are implemented when ARMv8.2-SM is implemented.

The following code sequence shows the use of the SM3 instructions to generate a SM3 hash.

```
.macro MessageExpand VA, VB, VC, VD, VOUT
 EXT \VOUT().16B, \VB().16B, \VC().16B, #12
 SM3PARTW1 \VOUT().4S, \VA().4S, \VD().4S
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
 EXT V17.16B, \VA().16B, \VB().16B, #12
 EXT V18.16B, \VC().16B, \VD().16B, #8
 SM3PARTW2 \VOUT().4S, V18.4S, V17.4S
 .endm

.macro HashPt1 VA, VB, Number SM3SS1 V23.4S, V20.4S, V22.4S, V19.4S
 EOR V21.16B, \VA().16B, \VB().16B
 SM3TT1a V20.4S, V23.4S, V21.S[\Number]
 SM3TT2a V19.4S, V23.4S, \VA().S[\Number]
 SHL V24.4S, V22.4S, #1
 SRI V24.4S, V22.4S, #31
 MOV V22.16B, V24.16B
 .endm

.macro HashPt2 VA, VB, Number SM3SS1 V23.4S, V20.4S, V25.4S, V19.4S
 EOR V21.16B, \VA().16B, \VB().16B
 SM3TT1b V20.4S, V23.4S, V21.S[\Number]
 SM3TT2b V19.4S, V23.4S, \VA().S[\Number]
 SHL V26.4S, V25.4S, #1
 SRI V26.4S, V25.4S, #31
 MOV V25.16B, V26.16B
 .endm


// V0-V3 holds the initial message
// V19 holds EFGH which is the lower half of the input hash
// V20 holds ABCD which is the upper half of the input hash
// V21 = current VPrime
// V22 holds T in bits[127:96] = 0x79cc4519
// V25 holds second value of T in bits[127:96] = 0x9d8a7a87<31:0>;

MessageExpand V0, V1, V2, V3, V4
MessageExpand V1, V2, V3, V4, V5
MessageExpand V2, V3, V4, V5, V6
MessageExpand V3, V4, V5, V6, V7
MessageExpand V4, V5, V6, V7, V8
MessageExpand V5, V6, V7, V8, V9
MessageExpand V6, V7, V8, V9, V10
MessageExpand V7, V8, V9, V10, V11
MessageExpand V8, V9, V10, V11, V12
MessageExpand V9, V10, V11, V12, V13
MessageExpand V10, V11, V12, V13, V14
MessageExpand V11, V12, V13, V14, V15
MessageExpand V12, V13, V14, V15, V16


MOV V29.16B, V19.16B
MOV V30.16B, V20.16B

HashPt1 V0,V1, 0
HashPt1 V0,V1, 1
HashPt1 V0,V1, 2
HashPt1 V0,V1, 3
HashPt1 V1,V2, 0
HashPt1 V1,V2, 1
HashPt1 V1,V2, 2
HashPt1 V1,V2, 3
HashPt1 V2,V3, 0
HashPt1 V2,V3, 1
HashPt1 V2,V3, 2
HashPt1 V2,V3, 3
HashPt1 V3,V4, 0
HashPt1 V3,V4, 1
HashPt1 V3,V4, 2
HashPt1 V3,V4, 3

HashPt2 V4,V5, 0
HashPt2 V4,V5, 1
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
HashPt2 V4,V5, 2
HashPt2 V4,V5, 3
HashPt2 V5,V6, 0
HashPt2 V5,V6, 1
HashPt2 V5,V6, 2
HashPt2 V5,V6, 3
HashPt2 V6,V7, 0
HashPt2 V6,V7, 1
HashPt2 V6,V7, 2
HashPt2 V6,V7, 3
HashPt2 V7,V8, 0
HashPt2 V7,V8, 1
HashPt2 V7,V8, 2
HashPt2 V7,V8, 3
HashPt2 V8,V9, 0
HashPt2 V8,V9, 1
HashPt2 V8,V9, 2
HashPt2 V8,V9, 3
HashPt2 V9,V10, 0
HashPt2 V9,V10, 1
HashPt2 V9,V10, 2
HashPt2 V9,V10, 3
HashPt2 V10,V11, 0
HashPt2 V10,V11, 1
HashPt2 V10,V11, 2
HashPt2 V10,V11, 3
HashPt2 V11,V12, 0
HashPt2 V11,V12, 1
HashPt2 V11,V12, 2
HashPt2 V11,V12, 3
HashPt2 V12,V13, 0
HashPt2 V12,V13, 1
HashPt2 V12,V13, 2
HashPt2 V12,V13, 3
HashPt2 V13,V14, 0
HashPt2 V13,V14, 1
HashPt2 V13,V14, 2
HashPt2 V13,V14, 3
HashPt2 V14,V15, 0
HashPt2 V14,V15, 1
HashPt2 V14,V15, 2
HashPt2 V14,V15, 3
HashPt2 V15,V16, 0
HashPt2 V15,V16, 1
HashPt2 V15,V16, 2
HashPt2 V15,V16, 3


EOR V19.16B, V29.16B, V19.16B
EOR V20.16B, V30.16B, V20.16B

// V19 holds EFGH which is the lower half of the output hash
// V20 holds ABCD which is the upper half of the output hash
```

K10.2.4 Use of the SM4 instructions

These instructions are implemented when ARMv8.2-SM is implemented.

The following code sequences show the use of the SM4 instructions to perform SM4 encryption and decryption:

Encryption

```
// Encryption
// V0 contains 0xb27022dc677d919756aa3350a3b1bac6<127:0>;
// V8 contains the Key
// V2 contains the data to be encrypted
// V16 contains: 0x545b6269383f464d1c232a3100070e15;
// V17 contains: 0xc4cbd2d9a8afb6bd8c939aa170777e85;
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
// V18 contains: 0x343b4249181f262dfc030a11e0e7eef5;
// V19 contains: 0xa4abb2b9888f969d6c737a8150575e65;
// V20 contains: 0x141b2229f8ff060ddce3eaf1c0c7ced5;
// V21 contains: 0x848b9299686f767d4c535a6130373e45;
// V22 contains: 0xf4fb0209d8dfe6edbcc3cad1a0a7aeb5;
// V23 contains: 0x646b7279484f565d2c333a4110171e25;

EOR V8.16b, V8.16b, V0.16b;
SM4EKEY V8.4S, V8.4S, V16.4S
SM4EKEY V9.4S, V8.4S, V17.4S
SM4EKEY V10.4S, V9.4S, V18.4S
SM4EKEY V11.4S, V10.4S, V19.4S
SM4EKEY V12.4S, V11.4S, V20.4S
SM4EKEY V13.4S, V12.4S, V21.4S
SM4EKEY V14.4S, V13.4S, V22.4S
SM4EKEY V15.4S, V14.4S, V23.4S

SM4E V2.4S, V8.4S
SM4E V2.4S, V9.4S
SM4E V2.4S, V10.4S
SM4E V2.4S, V11.4S
SM4E V2.4S, V12.4S
SM4E V2.4S, V13.4S
SM4E V2.4S, V14.4S
SM4E V2.4S, V15.4S

// need to reverse the order of the words at the end of the operation
REV64 v2.4S, v2.4S
EXT V2.16B, V2.16B, V2.16B, #8
```

Decryption

```
// Decryption
// V0 contains 0xb27022dc677d919756aa3350a3b1bac6<127:0>;
// V8 contains the Key
// V2 contains the data to be decrypted
// V16 contains: 0x545b6269383f464d1c232a3100070e15;
// V17 contains: 0xc4cbd2d9a8afb6bd8c939aa170777e85;
// V18 contains: 0x343b4249181f262dfc030a11e0e7eef5;
// V19 contains: 0xa4abb2b9888f969d6c737a8150575e65;
// V20 contains: 0x141b2229f8ff060ddce3eaf1c0c7ced5;
// V21 contains: 0x848b9299686f767d4c535a6130373e45;
// V22 contains: 0xf4fb0209d8dfe6edbcc3cad1a0a7aeb5;
// V23 contains: 0x646b7279484f565d2c333a4110171e25;

// need to reverse the order of the keys to do a decryption:

EOR V8.16b, V8.16b, V0.16b;
SM4EKEY V8.4S, V8.4S, V16.4S
SM4EKEY V9.4S, V8.4S, V17.4S
SM4EKEY V10.4S, V9.4S, V18.4S
SM4EKEY V11.4S, V10.4S, V19.4S
SM4EKEY V12.4S, V11.4S, V20.4S
SM4EKEY V13.4S, V12.4S, V21.4S
SM4EKEY V14.4S, V13.4S, V22.4S
SM4EKEY V15.4S, V14.4S, V23.4S

REV64 V8.4S, V8.4S
EXT V8.16B, V8.16B, V8.16B, #8
REV64 V9.4S, V9.4S
EXT V9.16B, V9.16B, V9.16B, #8
REV64 V10.4S, V10.4S
EXT V10.16B, V10.16B, V10.16B, #8
REV64 V11.4S, V11.4S
EXT V11.16B, V11.16B, V11.16B, #8
REV64 V12.4S, V12.4S
EXT V12.16B, V12.16B, V12.16B, #8
REV64 V13.4S, V13.4S
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
EXT V13.16B, V13.16B, V13.16B, #8
REV64 V14.4S, V14.4S
EXT V14.16B, V14.16B, V14.16B, #8
REV64 V15.4S, V15.4S
EXT V15.16B, V15.16B, V15.16B, #8

SM4E V2.4S, V15.4S
SM4E V2.4S, V14.4S
SM4E V2.4S, V13.4S
SM4E V2.4S, V12.4S
SM4E V2.4S, V11.4S
SM4E V2.4S, V10.4S
SM4E V2.4S, V9.4S
SM4E V2.4S, V8.4S

// final reversal of the order of the words in the result:
REV64 V2.4S, V2.4S
EXT V2.16B, V2.16B, V2.16B, #8
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Appendix K11
Barrier Litmus Tests

This appendix gives examples of the use of the barrier instructions provided by the Armv8 architecture. It contains the following sections:

- *Introduction* on page K11-7968.
- *Load-Acquire, Store-Release and barriers* on page K11-7971.
- *Load-Acquire Exclusive, Store-Release Exclusive and barriers* on page K11-7975.
- *Using a mailbox to send an interrupt* on page K11-7980.
- *Cache and TLB maintenance instructions and barriers* on page K11-7981.
- *Armv7 compatible approaches for ordering, using DMB and DSB barriers* on page K11-7993.

Note

This information is not part of the Arm architecture specification. It is included here as supplementary information, for the convenience of developers and users who might require this information.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K11.1 Introduction

The exact rules for the insertion of barriers into code sequences is a very complicated subject, and this appendix describes many of the corner cases and behaviors that are possible in an implementation of the Armv8 architecture.

This appendix is to help programmers, hardware design engineers, and validation engineers understand the need for the different kinds of barriers.

K11.1.1 Overview of memory consistency

Early generations of microprocessors were relatively simple processing engines that executed each instruction in program order. In such processors, the effective behavior was that each instruction was executed in its entirety before a subsequent instruction started to be executed. This behavior is sometimes referred to as the *Sequential Execution Model* (SEM), and in this Manual it is described as *Simple sequential execution* of the program.

In later processor generations, the needs to increase processor performance, both in terms of the frequency of operation and the number of instructions executed each cycle, mean that such a simple form of execution is abandoned. Many techniques, such as pipelining, write buffering, caching, speculation, and out-of-order execution, are introduced to provide improved performance.

For general purpose PEs, such as Arm, these microarchitectural innovations are largely hidden from the programmer by a number of microarchitectural techniques. These techniques ensure that, within an individual PE, the behavior of the PE largely remains the same as the SEM. There are some exceptions to this where explicit synchronization is required. In the Arm architecture, these are limited to cases such as:

- Synchronization of changes to the instruction stream.
- Synchronization of changes to System registers.

In both these cases, the `ISB` instruction provides the necessary synchronization.

While the effect of ordering is largely hidden from the programmer within a single PE, the microarchitectural innovations have a profound impact on the ordering of memory accesses. Write buffering, speculation, and cache coherency protocols, in particular, can all mean that the order in which memory accesses occur, as seen by an external observer, differs significantly from the order of accesses that would appear in the SEM. This is usually invisible in a uniprocessor environment, but the effect becomes much more significant when multiple PEs are trying to communicate with memory. In reality, these effects are often only significant at particular synchronization boundaries between the different threads of execution.

The problems that arise from memory ordering considerations are sometimes described as the problem of *memory consistency*. Processor architectures have adopted one or more *memory consistency models*, or *memory models*, that describe the permitted limits of the memory re-ordering that can be performed by an implementation of the architecture. The comparison and categorization of these has generated significant research and comment in academic circles, and Arm recommends the *Memory Consistency Models for Shared Memory-Multiprocessors* paper as an excellent detailed treatment of this subject.

This appendix does not reproduce such a work, but instead concentrates on some cases that demonstrate the features of the weakly-ordered memory model of the Arm architecture from Armv6. In particular, the examples show how the use of the `DMB` and `DSB` memory barrier instructions can provide the necessary safeguards to limit memory ordering effects at the required synchronization points.

K11.1.2 Barrier operation definitions

The following reference, or provide, definitions of terms used in this appendix:

DMB See *Data Memory Barrier (DMB)* on page B2-135.

DSB See *Data Synchronization Barrier (DSB)* on page B2-138.

ISB See *Instruction Synchronization Barrier (ISB)* on page B2-135.

Observer, Completion

See *Definition of the Armv8 memory model* on page B2-123.

See *Completion and endpoint ordering* on page B2-130.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Program order

The order of instructions as they appear in an assembly language program. This appendix does not attempt to describe or define the legal transformations from a program written in a higher level programming language, such as C or C++, into the machine language that can then be disassembled to give an equivalent assembly language program. Such transformations are a function of the semantics of the higher level language and the capabilities and options on the compiler.

K11.1.3 Conventions

Many of the examples are written in a stylized extension to Arm assembler, to avoid confusing the examples with unnecessary code sequences.

AArch32

The construct `WAIT([Rx]==1)` describes the following sequence:

```
loop
    LDR R12, [Rx]
    CMP R12, #1
    BNE loop
```

Also, the construct `WAIT_ACQ([Rx]==1)` describes the following sequence:

```
   loop
    LDA R12, [Rx]       ; load acquire ensures it is ordered before subsequent loads/stores
    CMP R12, #1
    BNE loop
```

R12 is chosen as an arbitrary temporary register that is not in use. It is named to permit the generation of a false dependency to ensure ordering.

AArch64

The construct `WAIT([Xx]==1)` describes the following sequence:

```
loop
    LDR W12, [Xx]
    CMP W12, #1
    B.NE loop
```

Also, the construct `WAIT_ACQ([Xx]==1)` and describes the following sequence:

```
   loop
    LDAR W12, [Xx]       ; load acquire ensures it is ordered before subsequent loads/stores
    CMP W12, #1
    B.NE loop
```

For each example, a code sequence is preceded by an identifier of the observer running it:

- P0, P1…P*x* refer to caching coherent PEs that implement the Armv8 architecture, and are in the same shareability domain.
- E0, E1…E*x* refer to non-caching observers, that do not participate in the coherency protocol, but execute Armv8 instructions and have a weakly-ordered memory model. This does not preclude these observers being different objects, such as DMA engines or other system masters.

These observers are unsynchronized other than as required by the documented code sequence.

Note

Throughout this appendix, *Armv8 instruction* and *instruction* refer to instructions from the A64, A32, or T32 instruction set, provided by Armv8 implementations.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Results are expressed in terms of `<agent>:<register>`, such as `P0:R5`. The results can be described as:

Permissible This does not imply that the results expressed are required or are the only possible results. In most cases they are results that would not be possible under a sequentially consistent running of the code sequences on the agents involved. In general terms, this means that these results might be unexpected to anyone unfamiliar with memory consistency issues.

Not permissible Results that the architecture expressly forbids.

Required Results that the architecture expressly requires.

The examples omit the required shareability domain arguments of `DMB` and `DSB` instructions. The arguments are assumed to be selected appropriately for the shareability domains of the observers.

In AArch32 state, where the barrier function in the litmus test can be achieved by a `DMB ST`, that is a barrier to stores only, this is shown by the use of `DMB [ST]`. This indicates that the `ST` qualifier can be omitted without affecting the result of the test. In some implementations `DMB ST` is faster than `DMB`.

For AArch64 code, the shareability domain of the `DMB` or `DSB` must be included. This is shown in this manual using the notation `DMB <domain>` and `DSB <domain>` respectively.

Except where otherwise stated, other conventions are:

- All memory initializes to 0.
- R0 and W0 contain the value 1.
- R1 - R4 and W1 - W4 contain arbitrary independent addresses that initialize to the same value on all PEs. The addresses held in these registers are shareable and:
  - The addresses held in R1 and R2 are in Write-Back Cacheable Normal memory.
  - The address held in R3 is in Write-Through Cacheable Normal memory.
  - The address held in R4 is in Non-cacheable Normal memory.
- R5 - R8 and W5 - W8 contain:
  - When used with an `STR` instruction, `0x55`, `0x66`, `0x77`, and `0x88` respectively.
  - When used with an `LDR` instruction, the value 0.
- R11 and W11 contain a new instruction or new translation table entry, as appropriate, and R10 contains the virtual address and the ASID, for use in this change of translation table entry.
- Memory locations are Normal memory locations unless otherwise stated.

The examples use mnemonics for the cache maintenance and TLB maintenance instructions. The following tables describe the mnemonics:

- *Cache maintenance system instructions* on page K14-8090.
- *TLB maintenance system instructions* on page K14-8091.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K11.2 Load-Acquire, Store-Release and barriers

The Load-Acquire and Store-Release instructions are described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-140.

The following sections show that most of the examples in sections *Simple ordering and barrier cases* on page K11-7993 and *Load-Exclusive, Store-Exclusive and barriers* on page K11-7997 can be achieved using the Load-Acquire and Store-Release instructions without the need for additional barriers.

K11.2.1 Message passing

The following sections describe:

- *Resolving weakly-ordered message passing by using Acquire and Release*.
- *Resolving message passing by the use of Store-Release and address dependency* on page K11-7972.

Resolving weakly-ordered message passing by using Acquire and Release

The message passing problem described in *Weakly-ordered message passing problem* on page K11-7993 can be solved by the use of Load-Acquire and Store-Release instructions when accessing the communications flag:

AArch32

P1

```
STR R5, [R1]          ; sets new data
STL R0, [R2]          ; sends flag indicating data ready, which is ordered after the STR
```

P2

```
WAIT_ACQ([R2]==1)     ; waits on flag
LDR R5, [R1]
```

AArch64

P1

```
STR W5, [X1]          ; sets new data
STLR W0, [X2]         ; sends flag indicating data ready, which is ordered after the STR
```

P2

```
WAIT_ACQ([X2]==1)     ; waits on flag
LDR W5, [X1]
```

This ensures the observed order of both the reads and the writes allows transfer of data such that the result P2:R5==`0x55` is guaranteed.

This approach also works with multiple observers, in a way that further observers use the same sequence as P2 uses:

AArch32

P3

```
WAIT_ACQ([R2]==1)     ; waits on flag
LDR R5, [R1]
```

AArch64

P3

```
WAIT_ACQ([X2]==1)     ; waits on flag
LDR W5, [X1]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Resolving message passing by the use of Store-Release and address dependency

The lack of ordering of stores discussed in *Message passing with multiple observers* on page K11-7994 can be resolved by the use of Store-Release for the store of the valid flag by P1, even when the observers are using an address dependency:

AArch32

P1

```
STR R5, [R1]          ; sets new data
STL R0, [R2]          ; sends flag indicating data ready using a Store-Release
```

P2

```
WAIT([R2]==1)
AND R12, R12, #0      ; R12 is the destination of LDR in the WAIT macro
LDR R5, [R1, R12]     ; the load has an address dependency on R12
                      : and so is ordered after the flag has been seen
```

AArch64

P1

```
STR W5, [X1]          ; sets new data
STLR W0, [X2]         ; sends flag indicating data ready using a Store-Release
```

P2

```
WAIT([X2]==1)
AND W12, W12, WZR     ; W12 is the destination of LDR in the WAIT macro
LDR W5, [X1, X12]     ; the load has an address dependency on W12
                      : and so is ordered after the flag has been seen
```

This ensures the observed order of the writes allows transfer of data such that P2:R5 and P3:R5 contain the same value of `0x55`.

This approach also works with multiple observers, in a way that further observers use the same sequence as P2 uses:

AArch32

P3

```
WAIT([R2]==1)
AND R12, R12, #0      ; R12 is the destination of LDR in the WAIT macro
LDR R5, [R1, R12]     ; the load has an address dependency on R12
                      : and so is ordered after the flag has been seen
```

AArch64

P3

```
WAIT([X2]==1)
AND W12, W12, WZR     ; R12 is the destination of LDR in the WAIT macro
LDR W5, [X1, X12]     ; the load has an address dependency on W12
                      : and so is ordered after the flag has been seen
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K11.2.2 Address dependency with object construction

When accessing an object-oriented data structure, the address dependency rule means that barriers are not required, even when initializing the object. A Store-Release can be used to ensure the order of the update of the base address:

AArch32

P1

```
STR R5, [R1, #offset] ; sets new data in a field
STL R1, [R2]          ; updates base address
```

P2

```
LDR R1, [R2]          ; reads base address
CMP R1, #0            ; checks if it is valid
BEQ null_trap
LDR R5, [R1, #offset] ; uses base address to read field
```

AArch64

P1

```
STR W5, [X1, #offset] ; sets new data in a field
STLR X1, [X2]         ; updates base address
```

P2

```
LDR X1, [X2]          ; reads base address
CMP X1, #0            ; check if it is valid
B.EQ null_trap
LDR W5, [X1, #offset] ; uses base address to read field
```

It is required that P2:R5==`0x55` if the null_trap is not taken. This avoids P2 observing a partially constructed object from P1. Significantly, P2 does not need a barrier to ensure this behavior.

The read of the base address in P2 could be a Load-Acquire, but it is not necessary in this case.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K11.2.3 WFE and WFI and barriers

The Wait For Event and Wait For Interrupt instructions permit the PE to suspend execution and enter a low-power state. An explicit `DSB` barrier instruction is required if it is necessary to ensure memory accesses made before the `WFI` or `WFE` are visible to other observers, unless some other mechanism has ensured this visibility. Examples of other mechanism that would guarantee the required visibility are the `DMB` described in *Posting a store before polling for acknowledgement* on page K11-7996, or a dependency on a load.

The following example requires the `DSB` to ensure that the store is visible:

AArch32

P1

```
    STR R0, [R2]
    DSB
Loop
    WFI
    B Loop
```

AArch64

P1

```
    STR W0, [X2]
    DSB <domain>
Loop
    WFI
    B Loop
```

This requirement is unchanged in Armv8 by the presence of Load-Acquire or Store-Release.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K11.3 Load-Acquire Exclusive, Store-Release Exclusive and barriers

The Armv8 architecture adds the acquire and release semantics to Load-Exclusive and Store-Exclusive instructions, which allows them to gain ordering acquire and/or release semantics.

The Load-Exclusive instruction can be specified to have acquire semantics, and the Store-Exclusive instruction can be specified to have release semantics. These can be arbitrarily combined to allow the atomic update created by a successful Load-Exclusive and Store-Exclusive pair to have any of:

- No Ordering semantics (using `LDREX` and `STREX`).
- Acquire only semantics (using `LDAEX` and `STREX`).
- Release only semantics (using `LDREX` and `STLEX`).
- Sequentially consistent semantics (using `LDAEX` and `STLEX`).

In addition, the Armv8 specification requires that the clearing of a global monitor will generate an event for the PE associated with the global monitor, which can simplify the use of `WFE`, by removing the need for a `DSB` barrier and `SEV` instruction.

K11.3.1 Acquiring a lock

A common use of Load-Exclusive and Store-Exclusive instructions is to claim a lock to permit entry into a critical region. This is typically performed by testing a lock variable that indicates 0 for a free lock and some other value, commonly 1 or an identifier of the process holding the lock, for a taken lock.

Note

The inclusion of AArch32 `PLDW` instructions or AArch64 `PFRM PST*` instructions in these examples is not a functional requirement, but will improve performance on many implementations. The performance benefit of adding these instructions will vary between different implementations of the architecture.

For a critical region, the requirement on taking a lock is usually for acquire semantics, while the clearing of a lock requires release semantics:

AArch32

Px

```
    PLDW[R1]              ; preload into cache in unique state
Loop
    LDAEX R5, [R1]        ; read lock with acquire
    CMP R5, #0            ; check if 0
    STREXEQ R5, R0, [R1]  ; attempt to store new value
    CMPEQ R5, #0          ; test if store suceeded
    BNE Loop              ; retry if not

    ; loads and stores in the critical region can now be performed
```

AArch64

Px

```
    PRFM PSTL1KEEP, [X1]  ; preload into cache in unique state
Loop
    LDAXR W5, [X1]        ; read lock with acquire
    CBNZ W5, Loop         ; check if 0
    STXR W5, W0, [X1]     ; attempt to store new value
    CBNZ W5, Loop         ; test if store succeeded and retry if not

    ; loads and stores in the critical region can now be performed
```

The acquire associated with the load is sufficient to ensure the required ordering in a lock situation. The Store-Exclusive will fail (and so be retried) if there is a store to the location being monitored between the Load-Exclusive and the Store-Exclusive.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K11.3.2 Releasing a lock

The converse operation of releasing a lock does not require the use of Load-Exclusive and Store-Exclusive instructions, because only a single observer is able to write to the lock. However, often it is necessary for any observer to observe any memory updates, or any values that are loaded into memory, before they observe the release of the lock. Therefore, the lock release needs release semantics:

AArch32

Px

```
    ; loads and stores in the critical region
    MOV R0, #0
    STL R0, [R1]          ; clear the lock with release semantics
```

AArch64

Px

```
    ; loads and stores in the critical region
    STLR WZR, [X1]        ; clear the lock with release semantics
```

K11.3.3 Ticket locks

When a lock is free, in order to avoid a rush to get the lock by many PEs, the use of ticket locks is common in more advanced systems. When the use is requested, the ticket locks determine the order of the users of the critical sections, in order to avoid starvation that can occur with a simple contention based spin lock.

A ticket lock allocates each thread a ticket number when it first requests the lock, and then compares that number with the current number for the lock. If they are the same, then the critical section can be entered. Otherwise the thread waits until the current number is equal to the ticket number for that thread.

The reading of the current number of the lock needs acquire semantics for the lock to be acquired.

Note

- The code in this section is little-endian code, as it views the combined current and next values as a single combined quantity. The addresses of the current and next ticket values need to be adjusted for a big-endian system.
- The inclusion of AArch32 `PLDW` instructions or AArch64 `PFRM PST*` instructions in these examples is not a functional requirement, but will improve performance on many implementations. The performance benefit of adding these instructions will vary between different implementations of the architecture.

This is shown in the implementation below:

AArch32

Px

```
    ; R1 holds two 16 bit quantities
    ; the lower halfword holds the current ticket number
    ; the higher halfword holds the next ticket number

    PLDW[R1]              ; preload into cache in unique state
Loop1
    LDAEX R5, [R1]        ; read current and next
    ADD R5, R5, #0x10000  ; increment the next number
    STREX R6, R5, [R1]    ; and update the value
    CMP R6, #0            ; did the exclusive pass
    BNE Loop1             ; retry if not
    CMP R5, R5, ROR #16   ; is the current ticket ours
    BEQ block_start
Loop2
    LDAH R6, [R1]         ; read current value
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    CMP R6, R5, LSR #16   ; compare it with our allocated ticket
    BNE Loop2             ; retry (spin) if it is not the same
block_start
```

AArch64

Px

```
    ; X1 holds 2 16 bit quantities
    ; the lower halfword holds the current ticket number
    ; the higher halfword holds the next ticket number

    PRFM PSTL1KEEP, [X1]  ; preload into cache in unique state
Loop1
    LDAXR W5, [X1]        ; read current and next
    ADD W5, W5, #0x10000  ; increment the next number
    STXR W6, W5, [X1]     ; and update the value
    CBNZ W6, Loop1        ; did the exclusive pass – retry if not

    AND W6, W5, #0xFFFF
    CMP W6, W5, LSR #16   ; is the current ticket ours
    B.EQ block_start
Loop2
    LDARH W6, [X1]        ; read current value
    CMP W6, W5, LSR #16   ; compare it with the our allocated ticket
    B.NE Loop2            ; retry (spin) if it isn't the same
block_start
```

Releasing the ticket lock simply involves incrementing the current ticket number, that is still assumed to be in R3, and doing a Store-Release:

AArch32

```
    ADD R6, R6, #1
    STLH R6, [R1]
```

AArch64

```
    ADD W6, W6, #1
    STLRH W6, [X1]
```

K11.3.4 Use of Wait For Event (WFE) and Send Event (SEV) with locks

The Armv8 architecture can use the Wait For Event mechanism to minimise the energy cost of polling variables by putting the PE into a low power state, suspending execution, until an asynchronous exception or an explicit event is seen by that PE. In Armv8, the event can be generated as a result of clearing the global monitor, so removing the need for a `DSB` barrier or an explicit send event message.

This can be used with simple locks or with ticket locks.

Note

The inclusion of AArch32 `PLDW` instructions or AArch64 `PFRM PST*` instructions in these examples is not a functional requirement, but will improve performance on many implementations. The performance benefit of adding these instructions will vary between different implementations of the architecture.

Simple lock

The following is an example of lock acquire code using `WFE`:

AArch32

Px

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    PLDW[R1]              ; preload into cache in unique state
Loop
    LDAEX R5, [R1]        ; read lock with acquire
    CMP R5, #0            ; check if 0
    WFENE                 ; sleep if the lock is held
    STREXEQ R5, R0, [R1]  ; attempt to store new value
    CMPEQ R5, #0          ; test if store succeeded
    BNE Loop              ; retry if not
```

AArch64

Px

```
    SEVL                  ; invalidates the WFE on the first loop iteration
    PRFM PSTL1KEEP, [X1]  ; allocate into cache in unique state
Loop
    WFE
    LDAXR W5, [X1]        ; read lock with acquire
    CBNZ W5, Loop         ; check if 0
    STXR W5, W0, [X1]     ; attempt to store new value
    CBNZ W5, Loop         ; test if store succeeded and retry if not

    ; loads and stores in the critical region can now be performed
```

And the following is an example of lock release code:

AArch32

Px

```
    ; loads and stores in the critical region
    MOV R0, #0
    STL R0, [R1]          ; clear the lock
```

AArch64

Px

```
    ; loads and stores in the critical region
    STLR WZR, [X1]        ; clear the lock
```

Ticket lock

In the Ticket lock case, the Load-Exclusive instruction can be used to move the monitor into the exclusive state for the express purpose of creating an event when the monitor changes state:

AArch32

Px

```
    ; R1 holds 2 16 bit quantities
    ; the lower halfword holds the current ticket number
    ; the higher halfword holds the next ticket number

    PLDW[R1]              ; preload into cache in unique state
Loop1
    LDAEX R5, [R1]        ; read current and next
    ADD R5, R5, #0x10000  ; increment the next number
    STREX R6, R5, [R1]    ; and update the value
    CMP R6, #0            ; did the exclusive pass
    BNE Loop              ; retry if not
    CMP R5, R5, ROR #16   ; is the current ticket ours
    BEQ block_start
    SEVL
Loop2
    WFE                   ; wait if there has not been a change to the count since last
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
                        ; read
    LDAEXH R6, [R1]     ; check the current count
    CMP R6, R5, LSR #16 ; check if it is equal
    BNE Loop2
block_start
```

AArch64

Px

```
    ; X1 holds 2 16 bit quantities
    ; the lower halfword holds the current ticket number
    ; the higher halfword holds the next ticket number

    PRFM PSTL1KEEP, [X1]  ; preload into cache in unique state
Loop1
    LDAXR W5, [X1]        ; read current and next
    ADD W5, W5, #0x10000  ; increment the next number
    STXR W6, W5, [X1]     ; and update the value
    CBNZ W6, Loop1        ; did the exclusive pass - retry if not

    AND W6, W5, 0xFFFF
    CMP W6, W5, LSR #16   ; is the current ticket ours
    B.EQ block_start
    SEVL
Loop2
    WFE
    LDAXRH W6, [X1]       ; read current value
    CMP W6, W5, LSR #16   ; compare it with our allocated ticket
    B.NE Loop2            ; retry (spin) if it is not the same
block_start
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K11.4 Using a mailbox to send an interrupt

In some message passing systems, it is common for one observer to update memory and then notify a second observer of the update by sending an interrupt, using a mailbox.

Although a memory access might be made to initiate the sending of the mailbox interrupt, a `DSB` instruction is required to ensure the completion of previous memory accesses.

Therefore, the following sequence is required to ensure that P2 observes the updated value:

AArch32

P1

```
STR R5, [R1]          ; message stored to shared memory location
DSB ST
STR R0, [R4]          ; R4 contains the address of a mailbox
```

P2

```
; interrupt service routine
LDR R5, [R1]
```

AArch64

P1

```
STR W5, [X1]          ; message stored to shared memory location
DSB ST
STR W0, [X4]          ; R4 contains the address of a mailbox
```

P2

```
; interrupt service routine
LDR W5, [X1]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K11.5 Cache and TLB maintenance instructions and barriers

The following sections describe the use of barriers with cache and TLB maintenance instructions:

- *Data cache maintenance instructions*.
- *Instruction cache maintenance instructions* on page K11-7985.
- *TLB maintenance instructions and barriers* on page K11-7988.

K11.5.1 Data cache maintenance instructions

The following sections describe the use of barriers with data cache maintenance instructions:

- *Message passing to non-caching observers*.
- *Multiprocessing message passing to non-caching observers*.
- *Invalidating DMA buffers, non-functional example* on page K11-7982.
- *Invalidating DMA buffers, functional example with single PE* on page K11-7983.
- *Invalidating DMA buffers, functional example with multiple coherent PEs* on page K11-7984.

Message passing to non-caching observers

The Armv8 architecture requires the use of `DMB` instructions to ensure the ordering of data cache maintenance instructions and their effects. The Load-Acquire and Store-Release instructions have no effect on cache maintenance instruction. This means the following message passing approaches can be used when communicating between caching observers and non-caching observers:

AArch32

P1

```
STR R5, [R1]          ; updates data (assumed to be in P1 cache)
DCCMVAC R1            ; cleans cache to point of coherency
DMB                   ; ensures effects of the clean will be observed before the
                      ; flag is set
STR R0, [R4]          ; sends flag to external agent (Non-cacheable location)
```

E1

```
WAIT_ACQ ([R4] == 1)  ; waits for the flag (with order)
LDR R5, [R1]          ; reads the data
```

AArch64

P1

```
STR W5, [X1]          ; updates data (assumed to be in P1 cache)
DC CVAC, X1           ; cleans cache to point of coherency
DMB ISH               ; ensures effects of the clean will be observed before the
                      ; flag is set
STR W0, [X4]          ; sends flag to external agent (Non-cacheable location)
```

E1

```
WAIT_ACQ ([X4] == 1)  ; waits for the flag (with order)
LDR W5, [X1]          ; reads the data
```

In this example, it is required that E1:R5==`0x55`.

Multiprocessing message passing to non-caching observers

The broadcast nature of the cache maintenance instructions combined with properties of barriers, means that the message passing principle for non-caching observers is:

AArch32

P1

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
STR R5, [R1]          ; updates data (assumed to be in P1 cache)
STL R0, [R2]          ; sends a flag for P2 (ordered by the store release)
```

P2

```
WAIT ([R2] == 1)      ; waits for P1 flag
DMB                   ; ensures cache clean is observed after P1 flag is observed
DCCMVAC R1            ; cleans cache to point of coherency - will clean P1 cache
DMB                   ; ensures effects of the clean will be observed before the
                      ; flag to E1 is set
STR R0, [R4]          ; sends flag to E1
```

E1

```
WAIT_ACQ ([R4] == 1)  ; waits for P2 flag (ordered)
LDR R5, [R1]          ; reads data
```

AArch64

P1

```
STR W5, [X1]          ; updates data (assumed to be in P1 cache)
STLR W0, [X2]         ; sends a flag for P2 (ordered)
```

P2

```
WAIT ([X2] == 1)      ; waits sfor P1 flag
DMB SY                ; ensure cache clean is observed after P1 flag is observed
DC CVAC, X1           ; cleans cache to point of coherency, will clean P1 cache
DMB SY                ; ensures effects of the clean will be observed before the
                      ; flag to E1 is set
STR W0, [X4]          ; sends flag to E1
```

E1

```
WAIT_ACQ ([X4] == 1)  ; waits for P2 flag
LDR W5, [X1]          ; reads data
```

In this example, it is required that E1:R5==`0x55`. The clean operation executed by P2 affects the data location in the P1 cache. The cast-out from the P1 cache is guaranteed to be observed before P2 updates [R4].

Note

The cache maintenance instructions are not ordered by the Load-Acquire and Store-Release instructions.

Invalidating DMA buffers, non-functional example

The basic scheme for communicating with an external observer that is a process that passes data in to a Cacheable memory region must take account of the architectural requirement that regions with a Normal Cacheable attribute can be allocated into a cache at any time, for example as a result of speculation. The following example shows this possibility:

AArch32

P1

```
DCIMVAC R1            ; ensures cache is not dirty. A clean operation could be used
                      ; but as the DMA will subsequently overwrite this region an
                      ; invalidate operation is sufficient and usually more efficient
DMB                   ; ensures cache invalidation is observed before the next store
                      ; is observed
STR R0, [R3]          ; sends flag to external agent
WAIT_ACQ ([R4]==1)    ; waits for a different flag from an external agent
LDR R5, [R1]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

E1

```
WAIT ([R3] == 1)      ; waits for flag
STR R5, [R1]          ; stores new data
STL R0, [R4]          ; sends a flag
```

AArch64

P1

```
DC IVAC, X1           ; ensure cache is not dirty. A clean operation could be used
                      ; but as the DMA will subsequently overwrite this region an
                      ; invalidate operation is sufficient and usually more efficient
DMB SY                ; ensures cache invalidation is observed before the next store
                      ; is observed
STR W0, [X3]          ; sends flag to external agent
WAIT_ACQ ([X4]==1)    ; waits for a different flag from an external agent
LDR W5, [X1]
```

E1

```
WAIT ([X3] == 1)      ; waits for flag
STR W5, [X1]          ; stores new data
STLR W0, [X4]         ; sends a flag
```

If a speculative access occurs, there is no guarantee that the cache line containing [R1] is not brought back into the cache after the cache invalidation, but before [R1] is written by E1. Therefore, the result P1:R5=0 is permissible.

Invalidating DMA buffers, functional example with single PE

AArch32

P1

```
DCIMVAC R1            ; ensures cache is not dirty. A clean operation could be used
                      ; but as the DMA will subsequently overwrite this region an
                      ; invalidate operation is sufficient and usually more efficient
DMB                   ; ensures cache invalidation is observed before the next store
                      ; is observed
STR R0, [R3]          ; sends flag to external agent
WAIT ([R4]==1)        ; waits for a different flag from an external agent
DMB                   ; ensures that cache invalidate is observed after the flag
                      ; from external agent is observed
DCIMVAC R1            ; ensures cache discards stale copies before use
LDR R5, [R1]
```

E1

```
WAIT ([R3] == 1)      ; waits for flag
STR R5, [R1]          ; stores new data
STL R0, [R4]          ; sends a flag
```

AArch64

P1

```
DC IVAC, X1           ; ensures cache is not dirty. A clean operation could be used
                      ; but as the DMA will subsequently overwrite this region an
                      ; invalidate operation is sufficient and usually more efficient
DMB SY                ; ensures cache invalidation is observed before the next store
                      ; is observed
STR W0, [X3]          ; sends flag to external agent
WAIT ([X4]==1)        ; waits for a different flag from an external agent
DMB SY                ; ensures that cache invalidate is observed after the flag
                      ; from external agent is observed
DC IVAC, X1           ; ensures cache discards stale copies before use
LDR W5, [X1]
```

E1

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
WAIT ([X3] == 1)      ; waits for flag
STR W5, [X1]          ; stores new data
STLR W0, [X4]         ; sends a flag
```

In this example, the result P1:R5 == `0x55` is required. Including a cache invalidation after the store by E1 to [R1] is observed ensures that the line is fetched from external memory after it has been updated.

Invalidating DMA buffers, functional example with multiple coherent PEs

The broadcasting of cache maintenance instructions, and the use of `DMB` instructions to ensure their observability, means that the previous example extends naturally to a multiprocessor system. Typically this requires a transfer of ownership of the region that the external observer is updating.

AArch32

P0

```
(Use data from [R1], potentially using [R1] as scratch space)
STL R0, [R2]          ; signals release of [R1]
WAIT_ACQ ([R2] == 0)  ; waits for new value from DMA
LDR R5, [R1]
```

P1

```
WAIT ([R2] == 1)      ; waits for release of [R1] by P0
DCIMVAC R1            ; ensures caches are not dirty, an invalidate is sufficient
DMB
STR R0, [R3]          ; requests new data for [R1]
WAIT ([R4] == 1)      ; waits for new data
DMB
DCIMVAC R1            ; ensures caches discard stale copies before use
DMB
MOV R0, #0
STR R0, [R2]          ; signals availability of new [R1]
```

E1

```
WAIT ([R3] == 1)      ; waits for new data request
STR R5, [R1]          ; sends new [R1]
DMB [ST]
STR R0, [R4]          ; indicates that new data is available to P1
```

AArch64

P0

```
(Use data from [X1], potentially using [X1] as scratch space)
STLR W0, [X2]         ; signals release of [X1]
WAIT_ACQ ([X2] == 0)  ; waits for new value from DMA
LDR W5, [X1]
```

P1

```
WAIT ([X2] == 1)      ; waits for release of [R1] by P0
DC IVAC, X1           ; ensures caches are not dirty, an invalidate is sufficient
DMB SY
STR W0, [X3]          ; requests new data for [R1]
WAIT ([X4] == 1)      ; waits for new data
DMB SY
DCIMVAC X1            ; ensures caches discard stale copies before use
DMB SY
STR WZR, [X2]         ; signals availability of new [R1]
```

E1

```
WAIT ([X3] == 1)      ; waits for new data request
STR W5, [X1]          ; sends new [R1]
STR W0, [X4]          ; indicates new data is available to P1
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In this example, the result P0:R5 == `0x55` is required. The `DMB` issued by P1 after the first data cache invalidation ensures that effect of the cache invalidation on P0 is seen by E1 before the store by E1 to [R1]. The `DMB` issued by P1 after the second data cache invalidation ensures that its effects are seen before the store of 0 to the semaphore location in [R2].

K11.5.2 Instruction cache maintenance instructions

The following sections describe the use of barriers with instruction cache maintenance instructions:

- *Ensuring the visibility of updates to instructions for a uniprocessor*.
- *Ensuring the visibility of updates to instructions for a multiprocessor*.

Ensuring the visibility of updates to instructions for a uniprocessor

On a single PE, the agent that causes instruction fetches, or instruction cache linefills, is a separate memory system observer from the agent that causes data accesses. Therefore, any operations to invalidate the instruction cache can rely only on seeing updates to memory that are complete. This must be ensured by the use of a `DSB` instruction.

Also, instruction cache maintenance instructions are only guaranteed to complete after the execution of a `DSB`, and an `ISB` is required to discard any instructions that might have been prefetched before the instruction cache invalidation completed. Therefore, on a uniprocessor, to ensure the visibility of an update to code and to branch to it, the following sequence is required:

AArch32

P1

```
STR R11, [R1]         ; R11 contains a new instruction to be stored in program memory
DCCMVAU R1            ; clean to PoU makes the new instruction visible to the instruction cache
DSB
ICIMVAU R1            ; ensures instruction cache/branch predictor discards stale data
BPIMVA R1
DSB                   ; ensures completion of the invalidation
ISB                   ; ensures instruction fetch path sees new instruction cache state
BX R1
```

In AArch64 state, the branch predictor maintenance is not required.

AArch64

P1

```
STR W11, [X1]         ; W11 contains a new instruction to be stored in program memory
DC CVAU, X1           ; clean to PoU makes the new instruction visible to instruction cache
DSB ISH
IC IVAU, X1           ; ensures instruction cache/branch predictor discards stale data
DSB ISH               ; ensures completion of the invalidation
ISB                   ; ensures instruction fetch path sees new instruction cache state
BR X1
```

Note

Where the changes to the instructions span multiple cache lines, then the data cache and instruction cache maintenance instructions can be duplicated to cover each of the lines to be cleaned and to be invalidated.

Ensuring the visibility of updates to instructions for a multiprocessor

The Armv8 architecture requires a PE that executes an instruction cache maintenance instruction to execute a `DSB` instruction to ensure completion of the maintenance operation. This ensures that the cache maintenance instruction is complete on all PEs in the Inner Shareable shareability domain.

An `ISB` is not broadcast, and so does not affect other PEs. This means that any other PE must perform its own `ISB` synchronization after it knows that the update is visible, if it is necessary to ensure its synchronization with the update. The following example shows how this might be done:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

AArch32

P1

```
STR R11, [R1]         ; R11 contains a new instruction to be stored in program memory
DCCMVAU R1            ; clean to PoU makes the new instruction visible to the instruction cache
DSB                   ; ensures completion of the clean on all PEs
ICIMVAU R1            ; ensures instruction cache discards stale data
BPIMVA R              ; ensures branch predictor discards stale data
DSB                   ; ensures completion of the instruction cache and branch predictor
                      ; invalidation on all PEs
STR R0, [R2]          ; sets flag to signal completion
ISB                   ; synchronizes context on this PE
BX R1                 ; branches to new code
```

P2-P*x*

```
WAIT ([R2] == 1)      ; waits for flag signalling completion
ISB                   ; synchronizes context on this PE
BX R1                 ; branches to new code
```

AArch64

P1

```
STR X11, [X1]         ; X11 contains a new instruction to be stored in program memory
DC CVAU, X1           ; clean to PoU makes the new instruction visible to the instruction cache
DSB ISH               ; ensures completion of the clean on all PEs
IC IVAU, X1           ; ensures instruction cache/branch predictor discards stale data
DSB ISH               ; ensures completion of the instruction cache/branch predictor
                      ; invalidation on all PEs
STR W0, [X2]          ; sets flag to signal completion
ISB                   ; synchronizes context on this PE
BR R1                 ; branches to new code
```

P2-P*x*

```
WAIT ([X2] == 1)      ; waits for flag signalling completion
ISB                   ; synchronizes context on this PE
BR X1                 ; branches to new code
```

Nonfunctional approach

The following sequence does not have the same effect, because a `DSB` is not required to complete the instruction cache maintenance instructions that other PEs issue:

AArch32

P1

```
STR R11, [R1]         ; R11 contains a new instruction to be stored in program memory
DCCMVAU R1            ; clean to PoU makes the new instruction visible to the instruction cache
DSB                   ; ensures completion of the clean on all PEs
ICIMVAU R1            ; ensures instruction cache discards stale data
BPIMVA R1             : ensures branch predictor discards stale data
DMB                   ; ensures ordering of the store after the invalidation
                      ; DOES NOT guarantee completion of instruction cache/branch
                      ; predictor on other PEs
STR R0, [R2]          ; sets flag to signal completion
DSB                   ; ensures completion of the invalidation on all PEs
ISB                   ; synchronizes context on this PE
BX R1                 ; branches to new code
```

P2-P*x*

```
WAIT ([R2] == 1)      ; waits for flag signalling completion
DSB                   ; this DSB does not guarantee completion of P1
                      ; ICIMVAU/BPIMVA
ISB
BX R1
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

AArch64

P1

```
STR W11, [X1]         ; W11 contains a new instruction to be stored in program memory
DC CVAU, X1           ; clean to PoU makes the new instruction visible to instruction cache
DSB ISH               ; ensures completion of the clean on all PEs
IC IVAU, X1           ; ensures instruction cache/branch predictor discards stale data
DMB ISH               ; ensures ordering of the store after the invalidation
                      ; DOES NOT guarantee completion of instruction cache/branch
                      ; predictor on other PEs
STR W0, [X2]          ; sets flag to signal completion
DSB ISH               ; ensures completion of the invalidation on all PEs
ISB                   ; synchronizes context on this PE
BR X1                 ; branches to new code
```

P2-P*x*

```
WAIT ([X2] == 1)      ; waits for flag signalling completion
DSB ISH               ; this DSB does not guarantee completion of P1
                      ; ICIMVAU/BPIMVA
ISB
BR X1
```

In this example, P2…P*x* might not see the updated region of code at R1.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K11.5.3 TLB maintenance instructions and barriers

The following sections describe the use of barriers with TLB maintenance instructions:

- *Ensuring the visibility of updates to translation tables for a uniprocessor*.
- *Ensuring the visibility of updates to translation tables for a multiprocessor*.
- *Paging memory in and out* on page K11-7989.
- *Using break-before-make when updating translation table entries* on page K11-7990.

Ensuring the visibility of updates to translation tables for a uniprocessor

On a single PE, the agent that causes translation table walks is a separate memory system observer from the agent that causes data accesses. Therefore, any operations to invalidate the TLB can only rely on seeing updates to memory that are complete. This must be ensured by the use of a `DSB` instruction.

The Armv8 architecture requires that translation table walks look in the data or unified caches at L1, so such systems do not require data cache cleaning.

After the translation tables update, any old copies of entries that might be held in the TLBs must be invalidated. This operation is only guaranteed to affect all instructions, including instruction fetches and data accesses, after the execution of a `DSB` and an `ISB`. Therefore, the code for updating a translation table entry is:

AArch32

P1

```
STR R11, [R1]         ; updates the translation table entry
DSB                   ; ensures visibility of the update to translation table walks
TLBIMVA R10
BPIALL
DSB                   ; ensures completion of the BP and TLB invalidation
ISB                   ; synchronises context on this PE
; new translation table entry can be relied upon at this point and all accesses
; generated by this observer using
; the old mapping have been completed
```

AArch64

P1

```
STR X11, [X1]         ; updates the translation table entry
DSB ISH               ; ensures visibility of the update to translation table walks
TLBI VAE1, X10        ; assumes we are in the EL1
DSB ISH               ; ensures completion of the TLB invalidation
ISB                   ; synchronise context on this PE
; new translation table entry can be relied upon at this point and all accesses
; generated by this observer using
; the old mapping have been completed
```

Importantly, by the end of this sequence, all accesses that used the old translation table mappings have been observed by all observers.

An example of this is where a translation table entry is marked as invalid. Such a system must provide a mechanism to ensure that any access to a region of memory being marked as invalid has completed before any action is taken as a result of the region being marked as invalid.

Ensuring the visibility of updates to translation tables for a multiprocessor

The same code sequence can be used in a multiprocessing system. The Armv8 architecture requires a PE that executes a TLB maintenance instruction to execute a `DSB` instruction to ensure completion of the maintenance operation. This ensures that the TLB maintenance instruction is complete on all PEs in the Inner Shareable shareability domain.

The completion of a `DSB` that completes a TLB maintenance instruction ensures that all accesses that used the old mapping have completed.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

AArch32

P1

```
STR R11, [R1]         ; updates the translation table entry
DSB                   ; ensures visibility of the update to translation table walks
TLBIMVAIS R10
BPIALLIS
DSB                   ; ensures completion of the BP and TLB invalidation
ISB                   ; Note ISB is not broadcast and must be executed locally
                      ; on other PEs
;  new translation table entry can be relied upon at this point and all accesses
;  generated by any observers affected by the broadcast TLBIMVAIS operation using
;  the old mapping have been completed
```

AArch64

P1

```
STR X11, [X1]         ; updates the translation table entry
DSB ISH               ; ensures visibility of the update to translation table walks
TLBI VAE1IS,  X10
DSB ISH               ; ensures completion of the TLB invalidation
ISB                   ; Note ISB is not broadcast and must be executed locally
                      ; on other PEs
;  new translation table entry can be relied upon at this point and all accesses
;  generated by any observers affected by the broadcast TLBIMVAIS operation using
;  the old mapping have been completed
```

The completion of the TLB maintenance instruction is guaranteed only by the execution of a `DSB` by the observer that performed the TLB maintenance instruction. The execution of a `DSB` by a different observer does not have this effect, even if the `DSB` is known to be executed after the TLB maintenance instruction is observed by that different observer.

Paging memory in and out

In a multiprocessor system there is a requirement to ensure the visibility of translation table updates when paging regions of memory into RAM from a backing store. This might, or might not, also involve paging existing locations in memory from RAM to a backing store. In such situations, the operating system selects one or more pages of memory that might be in use but are suitable to discard, with or without copying to a backing store, depending on whether or not the region of memory is writable. Disabling the translation table mappings for a page, and ensuring the visibility of that update to the translation tables, prevents agents accessing the page.

For this reason, it is important that the `DSB` that is performed after the TLB invalidation ensures that no other updates to memory using those mappings are possible.

An example sequence for the paging out of an updated region of memory, and the subsequent paging in of memory, is as follows:

AArch32

P1

```
STR R11, [R1]         ; updates the translation table for the region being paged out
DSB                   ; ensures visibility of the update to translation table walks
TLBIMVAIS R10         ; invalidates the old entry
DSB                   ; ensures completion of the invalidation on all PEs
ISB                   ; ensures visibility of the invalidation
BL SaveMemoryPageToBackingStore
BL LoadMemoryFromBackingStore
DSB                   ; ensures completion of the memory transfer (this could be part of
                      ; LoadMemoryFromBackingStore)
ICIALLUIS             ; also invalidates the branch predictor
DSB                   ; ensures completion of the instruction cache
                      ; and branch predictor invalidation
STR R9, [R1]          ; creates a new translation table entry with a new mapping
DSB                   ; ensures visibility of the new translation table mapping
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
ISB                  ; ensures synchronisation of this instruction stream
```

AArch64

P1

```
STR X11, [X1]        ; updates the translation table for the region being paged out
DSB ISH              ; ensures visibility of the update to translation table walks
TLBI VAE1IS, X10     ; invalidates the old entry
DSB ISH              ; ensures completion of the invalidation on all PEs
ISB                  ; ensures visibility of the invalidation
BL SaveMemoryPageToBackingStore
BL LoadMemoryFromBackingStore
DSB ISH              ; ensures completion of the memory transfer (this could be part of
                     ; LoadMemoryFromBackingStore)
IC IALLUIS           ; also invalidates the branch predictor
DSB ISH              ; ensures completion of the instruction cache
                     ; and branch predictor invalidation
STR X9, [X1]         ; creates a new translation table entry with a new mapping
DSB ISH              ; ensures visibility of the new translation table mapping
ISB                  ; ensures synchronisation of this instruction stream
```

This example assumes the memory copies are performed by an observer that is coherent with the caches of PE P1. This observer might be P1 itself, using a specific paging mapping. For clarity, the example omits the functional descriptions of `SaveMemoryPageToBackingStore` and `LoadMemoryFromBackingStore`. `LoadMemoryFromBackingStore` is required to ensure that the memory updates that it makes are visible to instruction fetches.

In this example, the use of ICIALLUIS in AArch32 state and IC IALLUIS in AArch64 state to invalidate the entire instruction cache is a simplification, that might not be optimal for performance. An alternative approach involves invalidating all of the lines in the caches using ICIMVAU in AArch32 state and IC IVAU operations in AArch64 state. This invalidation must be done when the mapping used for the ICIMVAU and IC IVAU operations is valid but not executable.

Using break-before-make when updating translation table entries

The Arm Architecture requires that reads to the same location are observed in order, and since application level software relies on this behavior, the operating system needs to maintain this illusion when it is changing a virtual to physical address mapping for a location, as is the case with copy on write or other memory management techniques. This illusion can be maintained provided that the software uses a break-before-make sequence when updating translation table entries whenever multiple threads of execution can use the same translation tables and the change to the translation entries involves any of:

- Changing the memory type.
- Changing the cacheability attributes
- Changing the output address (OA), if the OA of at least one of the old translation table entry and the new translation table entry is writable.

The architecture requires use of a break-before make sequence in these situations, see *Using break-before-make when updating translation table entries* on page D5-2669 for more information. However, if software did not use a break-before-make approach, an implementation might give a result that would occur if the two reads to the same virtual address did not occur in program order. An example of such an occurrence would be an implementation of copy-on-write, where one PE is performing two reads to the same virtual address at the same time as a second PE, running code associated with the operating system, is copying the data from one physical location that is mapped to by that virtual address, where the page was mapped as read-only, to a different physical location which will be mapped as read-write.

If the operating system changed the address mapping without going through an invalid entry, then it would be possible for a third PE to perform a write to the location that would be seen by the first load by the first PE, and not seen by the second load by the same PE.

The required break-before-make code sequence in this case is:

AArch32

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

P1

```
; R1, R2 contain an invalid translation table entry (that is, one with bit[0] == 0)
; R3 contains the address of the translation table entry
; R4 contains the Virtual Address and ASID of the VA being remapped
; R5, R6 contain the new valid translation table entry
STRD R1, R2, [R3]          ; stores invalid entry
DSB ISH                    ; ensures visibility of the update to translation table walks
TLBIMVAIS R4               ; invalidates the old entry
DSB ISH                    ; ensures completion of the invalidation on all PEs
ICIALLUIS                  ; also invalidates the branch predictor
STRD R5, R6, [R3]          ; store new mapping
DSB ISH                    ; ensures visibility of the update to translation table walks
ISB                        ; ensures synchronisation of this instruction stream
```

Note

This example shows an update to an entry in a translation table that is using the long-descriptor format.

AArch64

P1

```
; X1 contains an invalid translation table entry (that is, one with bit[0] == 0)
; X2 contains the address of the translation table entry
; X3 contains the Virtual Address and ASID of the VA being remapped
; X4 contains the new valid translation table entry
STR X1, [X2]               ; stores invalid entry
DSB ISH                    ; ensures visibility of the update to translation table walks
TLBI VAE1IS, X3            ; invalidates the old entry
DSB ISH                    ; ensures completion of the invalidation on all PEs
IC IALLUIS                 ; also invalidates the branch predictor
STR X4, [X2]               ; store new mapping
DSB ISH                    ; ensures visibility of the update to translation table walks
ISB                        ; ensures synchronisation of this instruction stream
```

If this sequence is correctly followed, then the architecture guarantees that the loads to a virtual address being remapped will be seen in the correct order.

The instruction cache maintenance is only required if the mapping from input address to output address has been changed as part of the change of the translation table entries, and the memory being moved is executable. In this example, the use of `ICIALLUIS` in AArch32 state and `IC IALLUIS` in AArch64 state to invalidate the entire instruction cache is a simplification, that might not be optimal for performance. An alternative approach involves invalidating all of the lines in the caches using `ICIMVAU` in AArch32 state, and `IC IVAU` in AArch64 state. This invalidation must be done when the mapping used for the `ICIMVAU` and `IC IVAU` operations is valid but not executable.

K11.5.4 Ordering of Memory-mapped device control with payloads

With a Memory-mapped peripheral, such as a DMA, which can also access memory for its own use, it is common to have control or status registers which are Memory-mapped. These registers need to be accessed in an ordered manner with respect to the data that the Memory-mapped peripheral is handling.

Two simple examples of this are:

- When a processing element is writing a buffer of data, and then writing to a control register in the DMA peripheral to start that peripheral to access the buffer of data.
- When a DMA peripheral has written to a buffer of data in memory, and the processing element is reading a status register to determine that the DMA transfer has completed, and then is reading the data.

For the case of the processing element writing a buffer of data, before starting the DMA peripheral, the ordering requirements between the stores to the data buffer and the stores to the Memory-mapped a to the DMA peripheral can be met by the insertion of a `DSB <domain>` instruction between these sets of accesses as this ensures the global observation of the stores before the DMA is started. this is shown by the following code:

AArch32

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

P1

```
STR R5, [R2]          ; data written to the data buffer
DSB
STR R0, [R4]          ; R4 contains the address of the DMA control register
```

AArch64

P1

```
STR W5, [X2]          ; data written to the data buffer
DSB <domain>
STR W0, [X4]          ; X4 contains the address of the DMA control register
```

For the case of DMA peripheral writing the data buffer and then setting a status register when those stores are complete (and so globally observed) and then having this status register polled by the processing element before the processing element reads the data buffer, the processing element must insert a `DSB <domain>` between the load that reads the status register, and the read of the buffer. A DMB, or load-acquire, is not sufficient as this problem is not solely concerned with observation order, since the polling read is actually a read of a status register at a slave, not the polling a data value that has been written by an observer.

For this case, the code is therefore:

AArch32

P1

```
WAIT ([R4] == 1)      ; R4 contains the address of the status register,
                      ; and the value '1' indicates completion of the DMA transfer
DSB
LDR R5, [R2]          ; reads data from the data buffer
```

AArch64

P1

```
WAIT ([X4] == 1)      ; X4 contains the address of the status register,
                      ; and the value '1' indicates completion of the DMA transfer
DSB <domain>
LDR W5, [X2]          ; reads data from the data buffer
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K11.6 Armv7 compatible approaches for ordering, using DMB and DSB barriers

The following sections describe the Armv7 compatible approaches for ordering, using DMB and DSB barriers:

- *Simple ordering and barrier cases*.
- *Load-Exclusive, Store-Exclusive and barriers* on page K11-7997.
- *Using a mailbox to send an interrupt* on page K11-7999.
- *Cache and TLB maintenance instructions and barriers* on page K11-7999.

K11.6.1 Simple ordering and barrier cases

Arm implements a weakly consistent memory model for Normal memory. In general terms, this means that the order of memory accesses observed by other observers might not be the order that appears in the program, for either loads or stores.

This section includes examples of this.

Simple weakly consistent ordering example

P1

```
STR R5, [R1]
LDR R6, [R2]
```

P2

```
STR R6, [R2]
LDR R5, [R1]
```

In the absence of barriers, the result of P1: R6=0, P2: R5=0 is permissible.

Message passing

The following sections describe:

- *Weakly-ordered message passing problem*.
- *Message passing with multiple observers* on page K11-7994.

Weakly-ordered message passing problem

P1

```
STR R5, [R1]            ; sets new data
STR R0, [R2]            ; sends flag indicating data ready
```

P2

```
WAIT([R2]==1)           ; waits on flag
LDR R5, [R1]            ; reads new data
```

In the absence of barriers, an end result of P2: R5=0 is permissible.

Resolving by the addition of barriers

The addition of barriers, to ensure the observed order of the reads and the writes, ensures that data is transferred so that the result P2:R5==`0x55` is guaranteed, as follows:

P1

```
STR R5, [R1]            ; sets new data
DMB [ST]                ; ensures all observers observe data before the flag
STR R0, [R2]            ; sends flag indicating data ready
```

P2

```
WAIT([R2]==1)           ; waits on flag
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
DMB                     ; ensures that the load of data is after the flag has been observed
LDR R5, [R1]
```

Resolving by the use of barriers and address dependency

There is a rule within the Arm architecture that:

- Where the value returned by a read is used for computation of the virtual address of a subsequent read or write, then these two memory accesses are observed in program order.

  Where the value returned by a read is used for computation of the virtual address of a subsequent read or write, this is called an *address dependency*. An address dependency exists even if the value returned by the first read has no effect on the virtual address. This might occur if the value returned is masked off before it is used, or if it confirms a predicted address value that it might have changed.

  This restriction applies only when the data value returned by a read is used as a data value to calculate the address of a subsequent read or write. It does not apply if the data value returned by a read determines the condition flags values, and the values of the flags are used for condition code evaluation to determine the address of a subsequent read, either through conditional execution or the evaluation of a branch. This is called a *control dependency*.

  Where both a control and address dependency exist, the ordering behavior is consistent with the address dependency.

Table K11-1 shows examples of address dependencies, control dependencies, and an address and control dependency.

Table K11-1 Dependency examples

Address dependency		Control dependency		Address and control dependency[a]
(a)	**(b)**	**(c)**	**(d)**	**(e)**
`LDR r1, [r0]`	`LDR r1, [r0]`	`LDR r1, [r0]`	`LDR r1, [r0]`	`LDR r1, [r0]`
`LDR r2, [r1]`	`AND r1, r1, #0`	`CMP r1, #55`	`CMP r1, #55`	`CMP r1, #0`
	`LDR r2, [r3, r1]`	`LDRNE r2, [r3]`	`MOVNE r4, #22`	`LDRNE r2, [r1]`
			`LDR r2, [r3, r4]`	

a. The address dependency takes priority.

This means that the data transfer example of *Weakly-ordered message passing problem* on page K11-7993 can also be satisfied as shown in the following example:

P1

```
STR R5, [R1]            ; sets new data
DMB [ST]                ; ensures all observers observe data before the flag
STR R0, [R2]            ; sends flag indicating data ready
```

P2

```
WAIT([R2]==1)
AND R12, R12, #0        ; R12 is destination of LDR in WAIT macro
LDR R5, [R1, R12]       ; the load has an address dependency on R12
                        ; and so is ordered after the flag has been seen
```

The load of R5 by P2 is ordered with respect to the load from [R2] because there is an address dependency using R12. P1 uses a `DMB` to ensure that P2 does not observe the write of [R2] before the write of [R1].

Message passing with multiple observers

Where the ordering of Normal memory accesses is not resolved by the use of barriers or dependencies, then different observers might observe the accesses in a different order, as shown in the following example:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

P1

```
STR R5, [R1]            ; sets new data
STR R0, [R2]            ; sends flag indicating data ready
```

P2

```
WAIT([R2]==1)
AND R12, R12, #0        ; R12 is destination of LDR in WAIT macro
LDR R5, [R1, R12]       ; the load has an address dependency on R12
                        ; and so is ordered after the flag has been seen
```

P3

```
WAIT([R2]==1)
AND R12, R12, #0        ; R12 is destination of LDR in WAIT macro
LDR R5, [R1, R12]       ; the load is address depndent on R12
                        ; and so is ordered after the flag has been seen
```

In this case, it is permissible for P2:R5 and P3:R5 to contain different values, because there is no order guaranteed between the two stores performed by P1.

Resolving by the addition of barriers

The addition of a barrier by P1, as shown in the following example, ensures the observed order of the writes, transferring data so that P2:R5 and P3:R5 both contain the value `0x55`:

P1

```
STR R5, [R1]            ; sets new data
DMB [ST]                ; ensures all observers observe data before the flag
STR R0, [R2]            ; sends flag indicating data ready
```

P2

```
WAIT([R2]==1)
AND R12, R12, #0        ; R12 is the destination of LDR in WAIT macro
LDR R5, [R1, R12]       ; the load has an address dependency on R12
                        ; and so is ordered after the flag has been seen
```

P3

```
WAIT([R2]==1)
AND R12, R12, #0        ; R12 is the destination of LDR in WAIT macro
LDR R5, [R1, R12]       ; the load has an address dependency on R12
                        ; and so is ordered after the flag has been seen
```

Address dependency with object construction

When accessing an object-oriented data structure, the address dependency rule means that barriers are not required, even when initializing the object:

P1

```
STR R5, [R1, #offset]   ; sets new data in a field
DMB [ST]                ; ensures all observers observe data before base address is updated
STR R1, [R2]            ; updates base address
```

P2

```
LDR R1, [R2]            ; reads for base address
CMP R1, #0              ; checks if it is valid
BEQ null_trap
LDR R5, [R1, #offset]   ; uses base address to read field
```

If the null_trap is not taken, it is required that P2:R5==`0x55`. This avoids P2 observing a partially constructed object from P1. Significantly, P2 does not require a barrier to ensure this behavior.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

P1 requires a barrier to ensure the observed order of the writes by P1. In general, the impact of requiring a barrier during the construction phase is much less than the impact of requiring a barrier for every read access.

Posting a store before polling for acknowledgement

In the case where an observer stores to a location, and then polls for an acknowledge from a different observer, the weak ordering of the memory model can lead to a deadlock, as the following example shows:

P1

```
STR R0, [R2]
WAIT ([R3]==1)
```

P2

```
WAIT ([R2]==1)
STR R0, [R3]
```

In Armv7 implementations that do not include the Multiprocessing Extensions, then this can deadlock because P2 might not observe the store by P1 in finite time. For Armv7 implementations with the Multiprocessing Extensions and for Armv8, this is not an issue as all stores must be observed by all observers within their shareability domain in finite time.

The addition of a `DMB` instruction prevents this deadlock in Armv7 implementations that do not include the Multiprocessing Extensions:

P1

```
STR R0, [R2]
DMB
WAIT ([R3]==1)
```

P2

```
WAIT ([R2]==1)
STR R0, [R3]
```

The `DMB` executed by P1 ensures that P2 observes the store by P1 before it observes the load by P1. This ensures a timely completion.

The following example is a variant of the previous example, where the two observers poll the same memory location:

P1

```
STR R0, [R2]
WAIT ([R2]==2)
```

P2

```
WAIT ([R2]==1)
LDR R0, [R2]
ADD R0, R0, #1
STR R0, [R2]
```

In this example, the same deadlock can occur in Armv7 implementations that do not include the Multiprocessing Extensions, because the architecture permits P1 to read the result of its own store to [R2] early, and continue doing so for an indefinite amount of time. The addition of a `DMB` instruction prevents this deadlock:

P1

```
STR R0, [R2]
DMB
WAIT ([R2]==2)
```

P2

```
WAIT ([R2]==1)
LDR R0, [R2]
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    ADD R0, R0, #1
    STR R0, [R2]
```

WFE and WFI and barriers

The Wait For Event and Wait For Interrupt instructions permit the PE to suspend execution and enter a low-power state. A `DSB` barrier instruction is required if it is necessary to ensure that memory accesses made before the `WFI` or `WFE` are visible to other observers, unless some other mechanism has ensured this visibility. Examples of other mechanism that would guarantee the required visibility are the `DMB` described in *Posting a store before polling for acknowledgement* on page K11-7996, or a dependency on a load.

The following example requires the `DSB` to ensure that the store is visible:

P1

```
    STR R0, [R2]
    DSB
Loop
    WFI
    B Loop
```

However, if the example in *Posting a store before polling for acknowledgement* on page K11-7996 is extended to include a WFE, there is no risk of a deadlock. The extended example is:

P1

```
    STR R0, [R2]
    DMB
Loop
    LDR R12, [R3]
    CMP R12, #1
    WFENE
    BNE Loop
```

P2

```
    WAIT ([R2]==1)
    STR R0, [R3]
    DSB
    SEV
```

In this example:

- The `DMB` by P1 ensures that P2 observes the store by P1 before it observes the load by P1.
- The dependency of the `WFE` on the result of the load by P1 means that this load must complete before P1 executes the `WFE`.

For more information about `SEV`, see *Use of Wait For Event (WFE) and Send Event (SEV) with locks* on page K11-7998.

K11.6.2 Load-Exclusive, Store-Exclusive and barriers

The Load-Exclusive and Store-Exclusive instructions, described in *Synchronization and semaphores* on page B2-167, are predictable only with Normal memory. These instructions do not have any implicit barrier functionality. Therefore, any use of these instructions to implement locks of any type requires the addition of explicit barriers.

Acquiring a lock

A common use of Load-Exclusive and Store-Exclusive instructions is to claim a lock to permit entry into a critical region. This is typically performed by testing a lock variable that indicates 0 for a free lock and some other value, commonly 1 or an identifier of the process holding the lock, for a taken lock.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The lack of implicit barriers in the Load-Exclusive and Store-Exclusive instructions means that the mechanism requires a `DMB` instruction between acquiring a lock and making the first access to the critical region, to ensure that all observers observe the successful claim of the lock before they observe any subsequent loads or stores to the region. This example shows Px acquiring a lock:

P*x*

```
Loop
    LDREX R5, [R1]           ; reads lock
    CMP R5, #0               ; checks if 0
    STREXEQ R5, R0, [R1]     ; attempts to store new value
    CMPEQ R5, #0             ; tests if store succeeded
    BNE Loop                 ; retries if not
    DMB                      ; ensures that all subsequent accesses are observed after the
                             ; gaining of the lock is observed
    ; loads and stores in the critical region can now be performed
```

Releasing a lock

The converse operation of releasing a lock does not require the use of Load-Exclusive and Store-Exclusive instructions, because only a single observer is able to write to the lock. However, often it is necessary for any observer to observe any memory updates, or any values that are loaded into memory, before they observe the release of the lock. Therefore, a `DMB` usually precedes the lock release, as the following example shows.

P*x*

```
    ; loads and stores in the critical region
    MOV R0, #0
    DMB                      ; ensures all previous accesses are observed before the lock is cleared
    STR R0, [R1]             ; clears the lock
```

Use of Wait For Event (WFE) and Send Event (SEV) with locks

The Armv8 architecture includes Wait For Event and Send Event instructions, that can be executed to reduce the required number of iterations of a lock-acquire loop, or *spinlock*, to reduce power. The basic mechanism involves an observer that is in a spinlock executing a `WFE` instruction that suspends execution on that observer until an asynchronous exception or an explicit event, sent by some other observer using the `SEV` instruction, is seen by the suspended observer. An observer that holds the lock executes an `SEV` instruction to send an event after it has released the lock.

The Event signal is a non-memory communication, and therefore the memory update that releases the lock must be observable by all observers before the `SEV` instruction is executed and the event is sent. This requires the use of `DSB` instruction, rather than `DMB`.

Therefore, the following is an example of lock acquire code using `WFE`:

P*x*

```
   Loop
    LDREX R5, [R1]           ; reads lock
    CMP R5, #0               ; checks if 0
    WFENE                    ; sleeps if the lock is held
    STREXEQ R5, R0, [R1]     ; attempts to store new value
    CMPEQ R5, #0             ; tests if store succeeded
    BNE Loop                 ; retries if not
    DMB                      ; ensures that all subsequent accesses are observed after the
                             ; gaining of the lock is observed
    ; loads and stores in the critical region can now be performed
```

And the following is an example of lock release code using `SEV`:

P*x*

```
    ; loads and stores in the critical region
    MOV R0, #0
    DMB                      ; ensures all previous accesses are observed before the lock is cleared
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
STR R0, [R1]            ; clears the lock
DSB                     ; ensures completion of the store that cleared the lock before
                        ; sending the event
SEV
```

K11.6.3 Using a mailbox to send an interrupt

In some message passing systems, it is common for one observer to update memory and then notify a second observer of the update by sending an interrupt, using a mailbox.

Although a memory access might be made to initiate the sending of the mailbox interrupt, a `DSB` instruction is required to ensure the completion of previous memory accesses.

Therefore, the following sequence is required to ensure that P2 observes the updated value:

P1

```
STR R5, [R1]            ; message stored to shared memory location
DSB [ST]
STR R1, [R4]            ; R4 contains the address of a mailbox
```

P2

```
; interrupt service routine
LDR R5, [R1]
```

Note

The `DSB` executed by P1 ensures global observation of the store to [R1].The interrupt timing ensures that the code executed by P2 is executed after the global observation of the update to [R1], and therefore must see this update. In some implementations, this might be implemented by requiring that interrupts flush non-coherent buffers that hold speculatively loaded data.

K11.6.4 Cache and TLB maintenance instructions and barriers

The following sections describe the use of barriers with cache and TLB maintenance instructions:

- *Data cache maintenance instructions*.
- *Instruction cache maintenance instructions* on page K11-8002.
- *TLB maintenance instructions and barriers* on page K11-8003.

Data cache maintenance instructions

The following sections describe the use of barriers with data cache maintenance instructions:

- *Message passing to non-caching observers*.
- *Multiprocessing message passing to non-caching observers* on page K11-8000.
- *Invalidating DMA buffers, non-functional example* on page K11-8000.
- *Invalidating DMA buffers, functional example with single PE* on page K11-8001.
- *Invalidating DMA buffers, functional example with multiple coherent PEs* on page K11-8001.

Message passing to non-caching observers

The Armv8 architecture requires the use of `DMB` instructions to ensure the ordering of data cache maintenance instructions and their effects. This means the following message passing approaches can be used when communicating between caching observers and non-caching observers:

P1

```
STR R5, [R1]            ; updates data (assumed to be in P1's cache)
DCCMVAC R1              ; cleans cache to point of coherency
DMB                     ; ensures effects of the clean will be observed before the flag is set
STR R0, [R4]            ; sends flag to external agent (Non-cacheable location)
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

E1

```
WAIT ([R4] == 1)        ; waits for the flag
DMB                     ; ensures that flag has been seen before reading data
LDR R5, [R1]            ; reads the data
```

In this example, it is required that E1:R5==`0x55`.

Multiprocessing message passing to non-caching observers

The broadcast nature of the cache maintenance instructions in Armv8, and in Armv7 implementations that include the Multiprocessing Extensions, combined with properties of barriers, means that the message passing principle for non-caching observers is:

P1

```
STR R5, [R1]            ; updates data (assumed to be in P1's cache)
DMB [ST]                ; ensures new data is observed before the flag to P2 is set
STR R0, [R2]            ; sends flag to P2
```

P2

```
WAIT ([R2] == 1)        ; waits for flag from P1
DMB                     ; ensures cache clean is observed after P1 flag is observed
DCCMVAC R1              ; cleans cache to point of coherency - this cleans the cache of P1
DMB                     ; ensures effects of the clean are observed before the flag to E1 is set
STR R0, [R4]            ; sends flag to E1
```

E1

```
WAIT ([R4] == 1)        ; waits for flag from P2
DMB                     ; ensures that flag has been observed before reading the data
LDR R5, [R1]            ; reads the data
```

In this example, it is required that E1:R5==`0x55`. The clean operation executed by P2 affects the data location in the P1 cache. The cast-out from the P1 cache is guaranteed to be observed before P2 updates [R4].

Invalidating DMA buffers, non-functional example

The basic scheme for communicating with an external observer that is a process that passes data in to a Cacheable memory region must take account of the architectural requirement that regions with a Normal Cacheable attribute can be allocated into a cache at any time, for example as a result of speculation. The following example shows this possibility:

P1

```
DCIMVAC R1              ; ensures caches are not dirty. A clean operation could be
                        ; used but the DMA overwrites this region so an invalidate operation
                        ; is sufficient and usually more efficient
DMB                     ; ensures cache invalidation is observed before the next store is observed
STR R0, [R3]            ; sends flag to external agent
WAIT ([R4]==1)          ; waits for a different flag from an external agent
DMB                     ; observes flag from external agent before reading new data. However [R1]
                        ; could have been brought into cache earlier
LDR R5, [R1]
```

E1

```
WAIT ([R3] == 1)        ; waits for flag
STR R5, [R1]            ; stores new data
DMB
STR R0, [R4]            ; sends a flag
```

If a speculative access occurs, there is no guarantee that the cache line containing [R1] is not brought back into the cache after the cache invalidation, but before [R1] is written by E1. Therefore, the result P1:R5=0 is permissible.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Invalidating DMA buffers, functional example with single PE

P1

```
DCIMVAC R1              ; ensures cache is not dirty. A clean operation could be
                        ; used but the DMA overwrites this region so an invalidate operation
                        ; is sufficient and usually more efficient
DMB                     ; ensures cache invalidation is observed before the next store is observed
STR R0, [R3]            ; sends flag to external agent
WAIT ([R4]==1)          ; waits for a different flag from an external agent
DMB                     ; ensures that cache invalidate is observed after the flag
                        ; from external agent is observed
DCIMVAC R1              ; ensures cache discards stale copies before use
LDR R5, [R1]
```

E1

```
WAIT ([R3] == 1)        ; waits for flag
STR R5, [R1]            ; stores new data
DMB [ST]
STR R0, [R4]            ; sends a flag
```

In this example, the result P1:R5 == `0x55` is required. Including a cache invalidation after the store by E1 to [R1] is observed ensures that the line is fetched from external memory after it has been updated.

Invalidating DMA buffers, functional example with multiple coherent PEs

The broadcasting of cache maintenance instructions, and the use of `DMB` instructions to ensure their observability, means that the previous example extends naturally to a multiprocessor system. Typically this requires a transfer of ownership of the region that the external observer is updating.

P0

```
(Use data from [R1], potentially using [R1] as scratch space)
DMB
STR R0, [R2]            ; signals release of [R1]
WAIT ([R2] == 0)        ; waits for new value from DMA
DMB
LDR R5, [R1]
```

P1

```
WAIT ([R2] == 1)        ; waits for release of [R1] by P0
DCIMVAC R1              ; ensures caches are not dirty, invalidate is sufficient
DMB
STR R0, [R3]            ; requests new data for [R1]
WAIT ([R4] == 1)        ; waits for new data
DMB
DCIMVAC R1              ; ensures caches discard stale copies before use
DMB
MOV R0, #0
STR R0, [R2]            ; signals availability of new [R1]
```

E1

```
WAIT ([R3] == 1)        ; waits for new data request
STR R5, [R1]            ; sends new [R1]
DMB [ST]
STR R0, [R4]            ; indicates new data available to P1
```

In this example, the result P0:R5 == `0x55` is required. The `DMB` issued by P1 after the first data cache invalidation ensures that effect of the cache invalidation on P0 is seen by E1 before the store by E1 to [R1]. The `DMB` issued by P1 after the second data cache invalidation ensures that its effects are seen before the store of 0 to the semaphore location in [R2].

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Instruction cache maintenance instructions

The following sections describe the use of barriers with instruction cache maintenance instructions:

- *Ensuring the visibility of updates to instructions for a uniprocessor*.
- *Ensuring the visibility of updates to instructions for a multiprocessor*.

Ensuring the visibility of updates to instructions for a uniprocessor

On a single PE, the agent that causes instruction fetches, or instruction cache linefills, is a separate memory system observer from the agent that causes data accesses. Therefore, any operations to invalidate the instruction cache can rely only on seeing updates to memory that are complete. This must be ensured by the use of a `DSB` instruction.

Also, instruction cache maintenance instructions are only guaranteed to complete after the execution of a `DSB`, and an `ISB` is required to discard any instructions that might have been prefetched before the instruction cache invalidation completed. Therefore, on a uniprocessor, to ensure the visibility of an update to code and to branch to it, the following sequence is required:

P1

```
STR R11, [R1]           ; R11 contains a new instruction to store in program memory
DCCMVAU R1              ; clean to PoU makes new instructions visible to instruction cache
DSB
ICIMVAU R1              ; ensures instruction cache and branch predictor discard stale data
BPIMVA R1
DSB                     ; ensures completion of the invalidation
ISB                     ; ensures instruction fetch path observes new instruction cache state
BX R1
```

Ensuring the visibility of updates to instructions for a multiprocessor

Armv8, and an Armv7 implementation that includes the Multiprocessing Extensions, requires a PE that executes an instruction cache maintenance instruction to execute a `DSB` instruction to ensure completion of the maintenance operation. This ensures that the cache maintenance instruction is complete on all PEs in the Inner Shareable shareability domain.

An `ISB` is not broadcast, and so does not affect other PEs. This means that any other PE must perform its own `ISB` synchronization after it knows that the update is visible, if it is necessary to ensure its synchronization with the update. The following example shows how this might be done:

P1

```
STR R11, [R1]           ; R11 contains a new instruction to store in program memory
DCCMVAU R1              ; clean to PoU makes new instructions visible to instruction cache
DSB                     ; ensures completion of the clean on all processors
ICIMVAU R1              ; ensures instruction cache/branch predictor discards stale data
BPIMVA R1
DSB                     ; ensures completion of the instruction cache and branch predictor
                        ; invalidation on all PEs
STR R0, [R2]            ; sets flag to signal completion
ISB                     ; synchronizes context on this PE
BX R1                   ; branches to new code
```

P2-P*x*

```
WAIT ([R2] == 1)        ; waits for flag signaling completion
ISB                     ; synchronizes context on this processor
BX R1                   ; branches to new code
```

Nonfunctional approach

The following sequence does not have the same effect, because a `DSB` is not required to complete the instruction cache maintenance instructions that other PEs issue:

P1

```
STR R11, [R1]           ; R11 contains a new instruction to store in program memory
DCCMVAU R1              ; clean to PoU makes new instructions visible to instruction cache
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
DSB                     ; ensure completion of the clean on all PEs
ICIMVAU R1              ; ensure instruction cache/branch predictor discards stale data
BPIMVA R1
DMB                     ; ensure ordering of the store after the invalidation
                        ; DOES NOT guarantee completion of instruction cache/branch
                        ; predictor on other PEs
STR R0, [R2]            ; sets flag to signal completion
DSB                     ; ensures completion of the invalidation on all PEs
ISB                     ; synchronizes context on this PE
BX R1                   ; branches to new code
```

P2-P*x*

```
WAIT ([R2] == 1)        ; waits for flag signaling completion
DSB                     ; this DSB does not guarantee completion of P1's ICIMVAU/BPIMVA
ISB
BX R1
```

In this example, P2…P*x* might not see the updated region of code at R1.

TLB maintenance instructions and barriers

The following sections describe the use of barriers with TLB maintenance instructions:

- *Ensuring the visibility of updates to translation tables for a uniprocessor*.
- *Ensuring the visibility of updates to translation tables for a multiprocessor* on page K11-8004.
- *Paging memory in and out* on page K11-8004.

Ensuring the visibility of updates to translation tables for a uniprocessor

On a single PE, the agent that causes translation table walks is a separate memory system observer from the agent that causes data accesses. Therefore, any operations to invalidate the TLB can only rely on seeing updates to memory that are complete. This must be ensured by the use of a `DSB` instruction.

In the Armv8 architecture, and in an Armv7 implementation that includes the Multiprocessing Extensions, translation table walks must look in the data or unified caches at L1, so such systems do not require data cache cleaning.

After the translation tables update, any old copies of entries that might be held in the TLBs must be invalidated. This operation is only guaranteed to affect all instructions, including instruction fetches and data accesses, after the execution of a `DSB` and an `ISB`. Therefore, the code for updating a translation table entry is:

P1

```
STR R11, [R1]           ; updates the translation table entry
DSB                     ; ensures visibility of the update to translation table walks
TLBIMVA R10
BPIALL
DSB                     ; ensures completion of the BP and TLB invalidation
ISB                     ; synchronizes context on this PE
;
; new translation table entry can be relied upon at this point and all accesses
; generated by this observer using the old mapping have been completed
```

Importantly, by the end of this sequence, all accesses that used the old translation table mappings have been observed by all observers.

An example of this is where a translation table entry is marked as invalid. Such a system must provide a mechanism to ensure that any access to a region of memory being marked as invalid has completed before any action is taken as a result of the region being marked as invalid.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Ensuring the visibility of updates to translation tables for a multiprocessor

The same code sequence can be used in a multiprocessing system. In the Armv8 architecture, and in an Armv7 implementation that includes the Multiprocessing Extensions, a PE that executes a TLB maintenance instruction must execute a `DSB` instruction to ensure completion of the maintenance operation. This ensures that the TLB maintenance instruction is complete on all PEs in the Inner Shareable shareability domain.

The completion of a `DSB` that completes a TLB maintenance instruction ensures that all accesses that used the old mapping have completed.

P1

```
STR R11, [R1]            ; updates the translation table entry
DSB                      ; ensures visibility of the update to translation table walks
TLBIMVAIS R10
BPIALLIS
DSB                      ; ensures completion of the BP and TLB invalidation
ISB                      ; Note ISB is not broadcast and must be executed locally on other PEs
;
; new translation table entry can be relied upon at this point and all accesses generated by any
; observers affected by the broadcast TLBIMVAIS operation using the old mapping have completed
```

The completion of the TLB maintenance instruction is guaranteed only by the execution of a `DSB` by the observer that performed the TLB maintenance instruction. The execution of a `DSB` by a different observer does not have this effect, even if the `DSB` is known to be executed after the TLB maintenance instruction is observed by that different observer.

Paging memory in and out

In a multiprocessor system there is a requirement to ensure the visibility of translation table updates when paging regions of memory into RAM from a backing store. This might, or might not, also involve paging existing locations in memory from RAM to a backing store. In such situations, the operating system selects one or more pages of memory that might be in use but are suitable to discard, with or without copying to a backing store, depending on whether or not the region of memory is writable. Disabling the translation table mappings for a page, and ensuring the visibility of that update to the translation tables, prevents agents accessing the page.

For this reason, it is important that the `DSB` that is performed after the TLB invalidation ensures that no other updates to memory using those mappings are possible.

An example sequence for the paging out of an updated region of memory, and the subsequent paging in of memory, is as follows:

P1

```
STR R11, [R1]            ; updates the translation table for the region being paged out
DSB                      ; ensures visibility of the update to translation table walks
TLBIMVAIS R10            ; invalidates the old entry
DSB                      ; ensures completion of the invalidation on all processors
ISB                      ; ensures visibility of the invalidation
BL SaveMemoryPageToBackingStore
BL LoadMemoryFromBackingStore
DSB                      ; ensures completion of the memory transfer (this could be part of
                         ; LoadMemoryFromBackingStore
ICIALLUIS                ; also invalidates the branch predictor
DSB                      ; ensures completion of the instruction cache
                         ; and branch predictor invalidation
STR R9, [R1]             ; creates a new translation table entry with a new mapping
DSB                      ; ensures visibility of the new translation table mapping
ISB                      ; ensures synchronization of this instruction stream
```

This example assumes the memory copies are performed by an observer that is coherent with the caches of PE P1. This observer might be P1 itself, using a specific paging mapping. For clarity, the example omits the functional descriptions of `SaveMemoryPageToBackingStore` and `LoadMemoryFromBackingStore`. `LoadMemoryFromBackingStore` is required to ensure that the memory updates that it makes are visible to instruction fetches.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In this example, the use of ICIALLUIS to invalidate the entire instruction cache is a simplification, that might not be optimal for performance. An alternative approach involves invalidating all of the lines in the caches using ICIMVAU operations. This invalidation must be done when the mapping used for the ICIMVAU operations is valid but not executable.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Appendix K12
Random Number Generation

This appendix provides further information on the generation of random numbers using ARMv8.5-RNG. It contains the following sections:

- *Properties of the generated random number* on page K12-8008.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K12.1 Properties of the generated random number

When ARMv8.5-RNG is implemented, reads to the RNDR and RNDRRS registers return a 64-bit random number. The random numbers must meet the properties and conform to the standards that are detailed in this section.

The output random number is from a Deterministic Random Bit Generator (DRBG), which is seeded from a True Random Number Generator (TRNG).

The TRNG provides entropy in the form of random numbers, from the sampled output of an unpredictable physical process.

The TRNG should conform to:

- The NIST SP800-90B standard.
- The NIST SP800-22 standard.
- The FIPS 140-2 standard.
- The BSI AIS-31 standard.

The DRBG produces random numbers from a cryptographically secure algorithm.

The DRBG is seeded from the TRNG.

The DRBG algorithm should conform to the NIST SP800-90A Rev 1 standard.

The DRBG is reseeded after an IMPLEMENTATION DEFINED number of random numbers has been generated and read using the RNDR register.

The DRBG is reseeded immediately before the random number is generated and read using the RNDRRS register.

The entire random number generation should conform to the NIST SP800-90C standard.

Note

Since a TRNG can only generate random bits at a limited rate, the random number bits are commonly collected in an “entropy pool” until needed. An implementation should ensure that lower privileged software cannot impact the performance of higher privileged software by entirely draining this “entropy pool”. The refill time cost of the “entropy pool” should be paid for by the persistent caller.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Appendix K13
Arm Pseudocode Definition

This appendix provides a definition of the pseudocode that is used in this manual, and defines some *helper* procedures and functions that are used by pseudocode. It contains the following sections:

- *About the Arm pseudocode* on page K13-8010.
- *Pseudocode for instruction descriptions* on page K13-8011.
- *Data types* on page K13-8013.
- *Operators* on page K13-8018.
- *Statements and control structures* on page K13-8024.
- *Built-in functions* on page K13-8029.
- *Miscellaneous helper procedures and functions* on page K13-8032.
- *Arm pseudocode definition index* on page K13-8034.

Note

This appendix is not a formal language definition for the pseudocode. It is a guide to help understand the use of Arm pseudocode. This appendix is not complete. Changes are planned for future releases.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K13.1 About the Arm pseudocode

The Arm pseudocode provides precise descriptions of some areas of the Arm architecture. This includes description of the decoding and operation of all valid instructions. *Pseudocode for instruction descriptions* on page K13-8011 gives general information about this instruction pseudocode, including its limitations.

The following sections describe the Arm pseudocode in detail:

- *Data types* on page K13-8013.
- *Operators* on page K13-8018.
- *Statements and control structures* on page K13-8024.

Built-in functions on page K13-8029 and *Miscellaneous helper procedures and functions* on page K13-8032 describe some built-in functions and pseudocode helper functions that are used by the pseudocode functions that are described elsewhere in this manual. *Arm pseudocode definition index* on page K13-8034 contains the indexes to the pseudocode.

K13.1.1 General limitations of Arm pseudocode

The pseudocode statements `IMPLEMENTATION_DEFINED`, `SEE`, `UNDEFINED`, and `UNPREDICTABLE` indicate behavior that differs from that indicated by the pseudocode being executed. If one of them is encountered:

- Earlier behavior indicated by the pseudocode is only specified as occurring to the extent required to determine that the statement is executed.
- No subsequent behavior indicated by the pseudocode occurs.

For more information, see *Special statements* on page K13-8028.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K13.2 Pseudocode for instruction descriptions

Each instruction description includes pseudocode that provides a precise description of what the instruction does, subject to the limitations described in *General limitations of Arm pseudocode* on page K13-8010 and *Limitations of the instruction pseudocode* on page K13-8012.

In the instruction pseudocode, instruction fields are referred to by the names shown in the encoding diagram for the instruction. *Instruction encoding diagrams and instruction pseudocode* gives more information about the pseudocode provided for each instruction.

K13.2.1 Instruction encoding diagrams and instruction pseudocode

Instruction descriptions in this manual contain:

- An Encoding section, containing one or more encoding diagrams, each followed by some encoding-specific pseudocode that translates the fields of the encoding into inputs for the common pseudocode of the instruction, and picks out any encoding-specific special cases.
- An Operation section, containing common pseudocode that applies to all of the encodings being described. The Operation section pseudocode contains a call to the `EncodingSpecificOperations()` function, either at its start or only after a condition code check performed by `if ConditionPassed() then`.

An encoding diagram specifies each bit of the instruction as one of the following:

- An obligatory 0 or 1, represented in the diagram as 0 or 1. If this bit does not have this value, the encoding corresponds to a different instruction.
- A *should be* 0 or 1, represented in the diagram as (0) or (1). If this bit does not have this value, the instruction is CONSTRAINED UNPREDICTABLE. For more information, see *SBZ or SBO fields T32 and A32 in instructions* on page K1-7834.
- A named single bit or a bit in a named multi-bit field. The `cond` field in bits[31:28] of many A32/T32 instructions has some special rules associated with it.

An encoding diagram matches an instruction if all obligatory bits are identical in the encoding diagram and the instruction, and one of the following is true:

- The encoding diagram is not for an A32/T32 instruction.
- The encoding diagram is for an A32/T32 instruction that does not have a `cond` field in bits[31:28].
- The encoding diagram is for an A32/T32 instruction that has a `cond` field in bits[31:28], and bits[31:28] of the instruction are not `0b1111`.

In the context of the instruction pseudocode, the execution model for an instruction is:

1. Find all encoding diagrams that match the instruction. It is possible that no encoding diagram matches. In that case, abandon this execution model and consult the relevant instruction set chapter instead to find out how the instruction is to be treated. The bit pattern of such an instruction is usually reserved and UNDEFINED, though there are some other possibilities. For example, unallocated hint instructions are documented as being reserved and executed as NOPs.
2. If the operation pseudocode for the matching encoding diagrams starts with a condition code check, perform that check. If the condition code check fails, abandon this execution model and treat the instruction as a NOP. If there are multiple matching encoding diagrams, either all or none of their corresponding pieces of common pseudocode start with a condition code check.
3. Perform the encoding-specific pseudocode for each of the matching encoding diagrams independently and in parallel. Each such piece of encoding-specific pseudocode starts with a bitstring variable for each named bit or multi-bit field in its corresponding encoding diagram, named the same as the bit or multi-bit field and initialized with the values of the corresponding bit or bits from the bit pattern of the instruction.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In a few cases, the encoding diagram contains more than one bit or field with same name. In these cases, the values of the different instances of those bits or fields must be identical. The encoding-specific pseudocode contains a special case using the `Consistent()` function to specify what happens if they are not identical. `Consistent()` returns `TRUE` if all instruction bits or fields with the same name as its argument have the same value, and `FALSE` otherwise.

If there are multiple matching encoding diagrams, all but one of the corresponding pieces of pseudocode must contain a special case that indicates that it does not apply. Discard the results of all such pieces of pseudocode and their corresponding encoding diagrams.

There is now one remaining piece of pseudocode and its corresponding encoding diagram left to consider. This pseudocode might also contain a special case, most commonly one indicating that it is CONSTRAINED UNPREDICTABLE. If so, abandon this execution model and treat the instruction according to the special case.

4. Check the *should be* bits of the encoding diagram against the corresponding bits of the bit pattern of the instruction. If any of them do not match, abandon this execution model and treat the instruction as CONSTRAINED UNPREDICTABLE, see *SBZ or SBO fields T32 and A32 in instructions* on page K1-7834.

5. Perform the rest of the operation pseudocode for the instruction description that contains the encoding diagram. That pseudocode starts with all variables set to the values they were left with by the encoding-specific pseudocode.

The `ConditionPassed()` call in the common pseudocode, if present, performs step 2, and the `EncodingSpecificOperations()` call performs steps 3 and 4.

K13.2.2 Limitations of the instruction pseudocode

The pseudocode descriptions of instruction functionality have a number of limitations. These are mainly due to the fact that, for clarity and brevity, the pseudocode is a sequential and mostly deterministic language.

These limitations include:

- Pseudocode does not describe the ordering requirements when an instruction generates multiple memory accesses. For a description of the ordering requirements on memory accesses see *Ordering constraints* on page E2-3984.

- Pseudocode does not describe the exact rules when an UNDEFINED instruction fails its condition code check. In such cases, the `UNDEFINED` pseudocode statement lies inside the `if ConditionPassed() then …` structure, either directly or in the `EncodingSpecificOperations()` function call, and so the pseudocode indicates that the instruction executes as a NOP. *Conditional execution of undefined instructions* on page G1-5740 describes the exact rules.

- Pseudocode does not describe the exact ordering requirements when a single floating-point instruction generates more than one floating-point exception and one or more of those floating-point exceptions is trapped. *Combinations of floating-point exceptions* on page E1-3965 describes the exact rules.

  Note

  There is no limitation in the case where all the floating-point exceptions are untrapped, because the pseudocode specifies the same behavior as the cross-referenced section.

- An exception can be taken during execution of the pseudocode for an instruction, either explicitly as a result of the execution of a pseudocode function such as `Abort()`, or implicitly, for example if an interrupt is taken during execution of an `LDM` instruction. If this happens, the pseudocode does not describe the extent to which the normal behavior of the instruction occurs. To determine that, see the descriptions of the exceptions in *Handling exceptions that are taken to an Exception level using AArch32* on page G1-5703.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K13.3 Data types

This section describes:

- *General data type rules*.
- *Bitstrings*.
- *Integers* on page K13-8014.
- *Reals* on page K13-8014.
- *Booleans* on page K13-8014.
- *Enumerations* on page K13-8015.
- *Structures* on page K13-8015.
- *Tuples* on page K13-8016.
- *Arrays* on page K13-8017.

K13.3.1 General data type rules

Arm architecture pseudocode is a strongly typed language. Every literal and variable is of one of the following types:

- Bitstring.
- Integer.
- Boolean.
- Real.
- Enumeration.
- Tuple.
- Struct.
- Array.

The type of a literal is determined by its syntax. A variable can be assigned to without an explicit declaration. The variable implicitly has the type of the assigned value. For example, the following assignments implicitly declare the variables `x`, `y` and `z` to have types integer, bitstring of length 1, and Boolean, respectively.

```
x = 1;
y = '1';
z = TRUE;
```

Variables can also have their types declared explicitly by preceding the variable name with the name of the type. The following example declares explicitly that a variable named `count` is an integer.

```
integer count;
```

This is most often done in function definitions for the arguments and the result of the function.

The remaining subsections describe each data type in more detail.

K13.3.2 Bitstrings

This section describes the bitstring data type.

Syntax

`bits(N)`	The type name of a bitstring of length `N`.
`bit`	A synonym of `bits(1)`.

Description

A bitstring is a finite-length string of 0s and 1s. Each length of bitstring is a different type. The minimum permitted length of a bitstring is 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bitstring constants literals are written as a single quotation mark, followed by the string of 0s and 1s, followed by another single quotation mark. For example, the two constants literals of type `bit` are `'0'` and `'1'`. Spaces can be included in bitstrings for clarity.

The bits in a bitstring are numbered from left to right *N*-1 to 0. This numbering is used when accessing the bitstring using bitslices. In conversions to and from integers, bit *N*-1 is the MSByte and bit 0 is the LSByte. This order matches the order in which bitstrings derived from encoding diagrams are printed.

Every bitstring value has a left-to-right order, with the bits being numbered in standard *little-endian* order. That is, the leftmost bit of a bitstring of length `N` is bit (*N*–1) and its right-most bit is bit 0. This order is used as the most-significant-to-least-significant bit order in conversions to and from integers. For bitstring constants and bitstrings that are derived from encoding diagrams, this order matches the way that they are printed.

Bitstrings are the only concrete data type in pseudocode, corresponding directly to the contents values that are manipulated in registers, memory locations, and instructions. All other data types are abstract.

K13.3.3 Integers

This section describes the data type for integer numbers.

Syntax

`integer` The type name for the integer data type.

Description

Pseudocode integers are unbounded in size and can be either positive or negative. That is, they are mathematical integers rather than what computer languages and architectures commonly call integers. Computer integers are represented in pseudocode as bitstrings of the appropriate length, and the pseudocode provides functions to interpret those bitstrings as integers.

Integer literals are normally written in decimal form, such as `0`, `15`, `-1234`. They can also be written in C-style hexadecimal form, such as `0x55` or `0x80000000`. Hexadecimal integer literals are treated as positive unless they have a preceding minus sign. For example, `0x80000000` is the integer $+2^{31}$. If -2^{31} needs to be written in hexadecimal, it must be written as `-0x80000000`.

K13.3.4 Reals

This section describes the data type for real numbers.

Syntax

`real` The type name for the real data type.

Description

Pseudocode reals are unbounded in size and precision. That is, they are mathematical real numbers, not computer floating-point numbers. Computer floating-point numbers are represented in pseudocode as bitstrings of the appropriate length, and the pseudocode provides functions to interpret those bitstrings as reals.

Real constant literals are written in decimal form with a decimal point. This means `0` is an integer constant literal, but `0.0` is a real constant literal.

K13.3.5 Booleans

This section describes the Boolean data type.

Syntax

`boolean` The type name for the Boolean data type.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`TRUE` The two values a Boolean variable can take.

Description

A Boolean is a logical `TRUE` or `FALSE` value.

Note

This is not the same type as `bit`, which is a bitstring of length 1. A Boolean can only take on one of two values: `TRUE` or `FALSE`.

K13.3.6 Enumerations

This section describes the enumeration data type.

Syntax and examples

`enumeration` Keyword to defined a new enumeration type.

```
enumeration Example {Example_One, Example_Two, Example_Three};
```

A definition of a new enumeration called `Example`, which can take on the values `Example_One`, `Example_Two`, `Example_Three`.

Description

An enumeration is a defined set of named values.

An enumeration must contain at least one named value. A named value must not be shared between enumerations.

Enumerations must be defined explicitly, although a variable of an enumeration type can be declared implicitly by assigning one of the named values to it. By convention, each named value starts with the name of the enumeration followed by an underscore. The name of the enumeration is its *type name*, or *type*, and the named values are its possible *values*.

K13.3.7 Structures

This section describes the structure data type.

Syntax and examples

`type` The keyword used to declare the structure data type.

```
type ShiftSpec is (bits(2) shift, integer amount)
```

An example definition for a new structure called `ShiftSpec` that contains an bitstring member called `shift` and a integer member named `amount`. Structure definitions must not be terminated with a semicolon.

```
ShiftSpec abc;
```

A declaration of a variable named `abc` of type `ShiftSpec`.

```
abc.shift
```

Syntax to refer to the individual members within the structure variable.

Description

A structure is a compound data type composed of one or more data items. The data items can be of different data types. This can include compound data types. The data items of a structure are called its members and are named.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In the syntax section, the example defines a structure called `ShiftSpec` with two members. The first is a bitstring of length 2 named `shift` and the second is an integer named `amount`. After declaring a variable of that type named `abc`, the members of this structure are referred to as `abc.shift` and `abc.amount`.

Every definition of a structure creates a different type, even if the number and type of their members are identical. For example:

```
type ShiftSpec1 is (bits(2) shift, integer amount)
type ShiftSpec2 is (bits(2) shift, integer amount)
```

`ShiftSpec1` and `ShiftSpec2` are two different types despite having identical definitions. This means that the value in a variable of type `ShiftSpec1` cannot be assigned to variable of type `ShiftSpec2`.

K13.3.8 Tuples

This section describes the tuple data type.

Examples

`(bits(32) shifter_result, bit shifter_carry_out)`

An example of the tuple syntax.

`(shift_t, shift_n) = ('00', 0);`

An example of assigning values to a tuple.

Description

A tuple is an ordered set of data items, separated by commas and enclosed in parentheses. The items can be of different types and a tuple must contain at least one data item.

Tuples are often used as the return type for functions that return multiple results. For example, in the syntax section, the example tuple is the return type of the function `Shift_C()`, which performs a standard A32/T32 shift or rotation. Its return type is a tuple containing two data items, with the first of type `bits(32)` and the second of type `bit`.

Each tuple is a separate compound data type. The compound data type is represented as a comma-separated list of ordered data types between parentheses. This means that the example tuple at the start of this section is of type `(bits(32), bit)`. The general principle that types can be implied by an assignment extends to implying the type of the elements in the tuple. For example, in the syntax section, the example assignment implicitly declares:

- `shift_t` to be of type `bits(2)`.
- `shift_n` to be of type `integer`.
- `(shift_t, shift_n)` to be a tuple of type `(bits(2), integer)`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K13.3.9 Arrays

This section describes the array data type.

Syntax

`array` The type name for the array data type.

`array data_type array_name[A..B];`

Declaration of an array of type `data_type`, which might be compound data type. It is named `array_name` and is indexed with an integer range from `A` to `B`.

Description

An array is an ordered set of fixed size containing items of a single data type. This can include compound data types. Pseudocode arrays are indexed by either enumerations or integer ranges. An integer range is represented by the lower inclusive end of the range, then `..`, then the upper inclusive end of the range.

For example:

The following example declares an array of 31 bitstrings of length 64, indexed from 0 to 30.

```
array bits(64) _R[0..30];
```

Arrays are always explicitly declared, and there is no notation for a constant literal array. Arrays always contain at least one element data item, because:

- Enumerations always contain at least one symbolic constant named value.
- Integer ranges always contain at least one integer.

An array declared with an enumeration type as the index must be accessed using enumeration values of that enumeration type. An array declared with an integer range type as the index must be accessed using integer values from that inclusive range. Accessing such an array with an integer value outside of the range is a coding error.

Arrays do not usually appear directly in pseudocode. The items that syntactically look like arrays in pseudocode are usually array-like functions such as `R[i]`, `MemU[address, size]` or `Elem[vector, i, size]`. These functions package up and abstract additional operations normally performed on accesses to the underlying arrays, such as register banking, memory protection, endian-dependent byte ordering, exclusive-access housekeeping and Advanced SIMD element processing. See *Function and procedure calls* on page K13-8024.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K13.4 Operators

This section describes:

- *Relational operators*.
- *Boolean operators*.
- *Bitstring operators* on page K13-8019.
- *Arithmetic operators* on page K13-8019.
- *The assignment operator* on page K13-8020.
- *Precedence rules* on page K13-8022.
- *Conditional expressions* on page K13-8022.
- *Operator polymorphism* on page K13-8022.

K13.4.1 Relational operators

The following operations yield results of type `boolean`.

Equality and non-equality

If two variables `x` and `y` are of the same type, their values can be tested for equality by using the expression `x == y` and for non-equality by using the expression `x != y`. In both cases, the result is of type `boolean`.

Both `x` and `y` must be of type `bits(N)`, `real`, `enumeration`, `boolean`, or `integer`. Named values from an `enumeration` can only be compared if they are both from the same `enumeration`. An exception is that a bitstring can be tested for equality with an integer to allow a `d==15` test.

A special form of comparison is defined with a bitstring literal that can contain bit values `'0'`, `'1'`, and `'x'`. Any bit with value `'x'` is ignored in determining the result of the comparison. For example, if `opcode` is a 4-bit bitstring, the expression `opcode == '1x0x'` matches the values `'1000'`, `'1100'`, `'1001'`, and `'1101'`. This is known as a bitmask.

Note

This special form is permitted in the implied equality comparisons in the `when` parts of `case … of …` structures.

Comparisons

If `x` and `y` are integers or reals, then `x < y`, `x <= y`, `x > y`, and `x >= y` are less than, less than or equal, greater than, and greater than or equal comparisons between them, producing Boolean results.

Set membership with `IN`

`<expression> IN {<set>}` produces `TRUE` if `<expression>` is a member of `<set>`. Otherwise, it is `FALSE`. `<set>` must be a list of expressions separated by commas.

K13.4.2 Boolean operators

If `x` is a Boolean expression, then `!x` is its logical inverse.

If `x` and `y` are Boolean expressions, then `x && y` is the result of ANDing them together. As in the C language, if `x` is `FALSE`, the result is determined to be `FALSE` without evaluating `y`.

Note

This is known as short circuit evaluation.

If `x` and `y` are `booleans`, then `x || y` is the result of ORing them together. As in the C language, if `x` is `TRUE`, the result is determined to be `TRUE` without evaluating `y`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

If x and y are booleans or Boolean expressions, then the result of x != y is the same as the result of exclusive-ORing x and y together. The operator EOR only accepts bitstring arguments.

K13.4.3 Bitstring operators

The following operations can be applied only to bitstrings.

Logical operations on bitstrings

If x is a bitstring, NOT(x) is the bitstring of the same length obtained by logically inverting every bit of x.

If x and y are bitstrings of the same length, x AND y, x OR y, and x EOR y are the bitstrings of that same length obtained by logically ANDing, logically ORing, and exclusive-ORing corresponding bits of x and y together.

Bitstring concatenation and slicing

If x and y are bitstrings of lengths N and M respectively, then x:y is the bitstring of length N+M constructed by concatenating x and y in left-to-right order.

The bitstring slicing operator addresses specific bits in a bitstring. This can be used to create a new bitstring from extracted bits or to set the value of specific bits. Its syntax is x<integer_list>, where x is the integer or bitstring being sliced, and <integer_list> is a comma-separated list of integers enclosed in angle brackets. The length of the resulting bitstring is equal to the number of integers in <integer_list>. In x<integer_list>, each of the integers in <integer_list> must be:

- >= 0.
- < Len(x) if x is a bitstring.

The definition of x<integer_list> depends on whether integer_list contains more than one integer:

- If integer_list contains more than one integer, x<i, j, k,…, n> is defined to be the concatenation:

  x<i> : x<j> : x<k> : … : x<n>.

- If integer_list consists of just one integer i, x<i> is defined to be:
  - If x is a bitstring, '0' if bit i of x is a zero and '1' if bit i of x is a one.
  - If x is an integer, and y is the unique integer in the range 0 to 2^(i+1)-1 that is congruent to x modulo 2^(i+1). Then x<i> is '0' if y < 2^i and '1' if y >= 2^i.

    Loosely, this definition treats an integer as equivalent to a sufficiently long two's complement representation of it as a bitstring.

The notation for a range expression is i:j with i >= j is shorthand for the integers in order from i down to j, with both end values included. For example, instr<31:28> represents instr<31, 30, 29, 28>.

x<integer_list> is assignable provided x is an assignable bitstring and no integer appears more than once in <integer_list>. In particular, x<i> is assignable if x is an assignable bitstring and 0 <= i < Len(x).

Encoding diagrams for registers frequently show named bits or multi-bit fields. For example, the encoding diagram for the APSR shows its bit<31> as N. In such cases, the syntax APSR.N is used as a more readable synonym for APSR<31> as named bits can be referred to with the same syntax as referring to members of a struct. A comma-separated list of named bits enclosed in angle brackets following the register name allows multiple bits to be addressed simultaneously. For example, APSR.<N, C, Q> is synonymous with APSR <31, 29, 27>.

K13.4.4 Arithmetic operators

Most pseudocode arithmetic is performed on integer or real values, with operands obtained by conversions from bitstrings and results converted back to bitstrings. As these data types are the unbounded mathematical types, no issues arise about overflow or similar errors.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Unary plus and minus

If `x` is an integer or real, then `+x` is `x` unchanged, `-x` is `x` with its sign reversed. Both are of the same type as `x`.

Addition and subtraction

If `x` and `y` are integers or reals, `x+y` and `x-y` are their sum and difference. Both are of type `integer` if `x` and `y` are both of type `integer`, and `real` otherwise.

There are two cases where the types of `x` and `y` can be different. A bitstring and an integer can be added together to allow the operation `PC + 4`. An integer can be subtracted from a bitstring to allow the operation `PC - 2`.

If `x` and `y` are bitstrings of the same length `N`, so that `N = Len(x) = Len(y)`, then `x+y` and `x-y` are the least significant *N* bits of the results of converting `x` and `y` to integers and adding or subtracting them. Signed and unsigned conversions produce the same result:

```
x+y = (SInt(x) + SInt(y))<N-1:0>
    = (UInt(x) + UInt(y))<N-1:0>
x-y = (SInt(x) - SInt(y))<N-1:0>
    = (UInt(x) - UInt(y))<N-1:0>
```

If `x` is a bitstring of length `N` and `y` is an integer, `x+y` and `x-y` are the bitstrings of length `N` defined by `x+y = x + y<N-1:0>` and `x-y = x - y<N-1:0>`. Similarly, if `x` is an integer and `y` is a bitstring of length `M`, `x+y` and `x-y` are the bitstrings of length `M` defined by `x+y = x<M-1:0> + y` and `x-y = x<M-1:0> - y`.

Multiplication

If `x` and `y` are integers or reals, then `x * y` is the product of `x` and `y`. It is of type `integer` if `x` and `y` are both of type `integer`, and `real` otherwise.

Division and modulo

If `x` and `y` are reals, then `x/y` is the result of dividing `x` by `y`, and is always of type `real`.

If `x` and `y` are integers, then `x DIV y` and `x MOD y` are defined by:

```
x DIV y = RoundDown(x/y)
x MOD y = x - y * (x DIV y)
```

It is a pseudocode error to use any of `x/y`, `x MOD y`, or `x DIV y` in any context where `y` can be zero.

Scaling

If `x` and `n` are of type `integer`, then:

- `x << n = RoundDown(x * 2^n)`.
- `x >> n = RoundDown(x * 2^(-n))`.

Raising to a power

If `x` is an integer or a real and `n` is an integer then `x^n` is the result of raising `x` to the power of `n`, and:

- If `x` is of type `integer` then `x^n` is of type `integer`.
- If `x` is of type `real` then `x^n` is of type `real`.

K13.4.5 The assignment operator

The assignment operator is the `=` character, which assigns the value of the right-hand side to the left-hand side. An assignment statement takes the form:

```
<assignable_expression> = <expression>;
```

This following subsection defines valid expression syntax.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

General expression syntax

An expression is one of the following:

- A literal.
- A variable, optionally preceded by a data type name to declare its type.
- The word `UNKNOWN` preceded by a data type name to declare its type.
- The result of applying a language-defined operator to other expressions.
- The result of applying a function to other expressions.

Variable names normally consist of alphanumeric and underscore characters, starting with an alphabetic or underscore character.

Each register defined in an Arm architecture specification defines a correspondingly named pseudocode bitstring variable, and that variable has the stated behavior of the register. For example, if a bit of a register is defined as RAZ/WI, then the corresponding bit of its variable reads as '0' and ignore writes.

An expression like `bits(32) UNKNOWN` indicates that the result of the expression is a value of the given type, but the architecture does not specify what value it is and software must not rely on such values. The value produced must not:

- Return information that cannot be accessed at the current or a lower level of privilege using instructions that are not UNPREDICTABLE or `CONSTRAINED UNPREDICTABLE` and do not return UNKNOWN values,
- Be promoted as providing any useful information to software.

Note

UNKNOWN values are similar to the definition of UNPREDICTABLE, but do not indicate that the entire architectural state becomes unspecified.

Only the following expressions are assignable. This means that these are the only expressions that can be placed on the left-hand side of an assignment.

- Variables.
- The results of applying some operators to other expressions.

  The description of each language-defined operator that can generate an assignable expression specifies the circumstances under which it does so. For example, those circumstances might require that one or more of the expressions the operator operates on is an assignable expression.
- The results of applying array-like functions to other expressions. The description of an array-like function specifies the circumstances under which it can generate an assignable expression.

Note

If the right-hand side in an assignment is a function returning a tuple, an item in the assignment destination can be written as `-` to indicate that the corresponding item of the assigned tuple value is discarded. For example:

```
(shifted, -) = LSL_C(operand, amount);
```

The expression on the right-hand side itself can be a tuple. For example:

```
(x, y) = (function_1(), function_2());
```

Every expression has a data type.

- For a literal, this data type is determined by the syntax of the literal.
- For a variable, there are the following possible sources for the data type
  - An optional preceding data type name.
  - A data type the variable was given earlier in the pseudocode by recursive application of this rule.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— A data type the variable is being given by assignment, either by direct assignment to the variable, or by assignment to a list of which the variable is a member.

It is a pseudocode error if none of these data type sources exists for a variable, or if more than one of them exists and they do not agree about the type.

- For a language-defined operator, the definition of the operator determines the data type.
- For a function, the definition of the function determines the data type.

K13.4.6 Precedence rules

The precedence rules for expressions are:

1. Literals, variables and function invocations are evaluated with higher priority than any operators using their results, but see *Boolean operators* on page K13-8018.
2. Operators on integers follow the normal operator precedence rules of *exponentiation before multiply/divide before add/subtract*, with sequences of multiply/divides or add/subtracts evaluated left-to-right.
3. Other expressions must be parenthesized to indicate operator precedence if ambiguity is possible, but need not be if all permitted precedence orders under the type rules necessarily lead to the same result. For example, if `i`, `j` and `k` are integer variables, `i > 0 && j > 0 && k > 0` is acceptable, but `i > 0 && j > 0 || k > 0` is not.

K13.4.7 Conditional expressions

If `x` and `y` are two values of the same type and `t` is a value of type `boolean`, then `if t then x else y` is an expression of the same type as `x` and `y` that produces `x` if `t` is `TRUE` and `y` if `t` is `FALSE`.

K13.4.8 Operator polymorphism

Operators in pseudocode can be polymorphic, with different functionality when applied to different data types. Each resulting form of an operator has a different prototype definition. For example, the operator `+` has forms that act on various combinations of integers, reals and bitstrings.

Table K13-1 summarizes the operand types valid for each unary operator and the result type. Table K13-2 on page K13-8023 summarizes the operand types valid for each binary operator and the result type.

Table K13-1 Result and operand types permitted for unary operators

Operator	Operand Type	Result Type
`-`	`integer`	`integer`
	`real`	`real`
`NOT`	`bits(N)`	`bits(N)`
`!`	`boolean`	`boolean`

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K13-2 Result and operand types permitted for binary operators

Operator	First operand type	Second operand type	Result type
==	bits(N)	integer	boolean
		bits(N)	
	integer	integer	
	real	real	
	enumeration	enumeration	
	boolean	boolean	
!=	bits(N)	bits(N)	boolean
	integer	integer	
	real	real	
<, > <= , >=	integer	integer	boolean
	real	real	
+, -	integer	integer	integer
	real	real	real
	bits(N)	bits(N)	bits(N)
		integer	
<<, >>	integer	integer	integer
*	integer	integer	integer
	real	real	real
	bits(N)	bits(N)	bits(N)
/	real	real	real
DIV	integer	integer	integer
MOD	integer	integer	integer
	bits(N)	integer	
&&, ||	boolean	boolean	boolean
AND, OR, EOR	bits(N)	bits(N)	bits(N)
^	integer	integer	integer
	real	integer	real

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K13.5 Statements and control structures

This section describes the statements and program structures available in the pseudocode:

- *Statements and Indentation*.
- *Function and procedure calls*.
- *Conditional control structures* on page K13-8026.
- *Loop control structures* on page K13-8027.
- *Special statements* on page K13-8028.
- *Comments* on page K13-8028.

K13.5.1 Statements and Indentation

A simple statement is either an assignment, a function call, or a procedure call. Each statement must be terminated with a semicolon.

Indentation normally indicates the structure in compound statements. The statements contained in structures such as `if … then … else …` or procedure and function definitions are indented more deeply than the statement structure itself. The end of a compound statement structure and their end is indicated by returning to the original indentation level or less.

Indentation is normally done by four spaces for each level. Standard indentation uses four spaces for each level of indent.

K13.5.2 Function and procedure calls

This section describes how functions and procedures are defined and called in the pseudocode.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Procedure and function definitions

A procedure definition has the form:

```
<procedure name>(<argument prototypes>)
    <statement 1>;
    <statement 2>;
    …
    <statement n>;
```

where `<argument prototypes>` consists of zero or more argument definitions, separated by commas. Each argument definition consists of a type name followed by the name of the argument.

Note

This first definition line is not terminated by a semicolon. This distinguishes it from a procedure call.

A function definition is similar, but also declares the return type of the function:

```
<return type> <function name>(<argument prototypes>)
    <statement 1>;
    <statement 2>;
    …
    <statement n>;
```

Note

A function or procedure name can include a "`.`". This is a convention used for functions that have similar but different behaviors in AArch32 and AArch64 states.

Array-like functions are similar, but are written with square brackets and have two forms. These two forms exist because reading from and writing to an array element require different functions. They are frequently used in memory operations. An array-like function definition with a return type is equivalent to reading from an array. For example:

```
<return type> <function name>[<argument prototypes>]
    <statement 1>;
    <statement 2>;
    …
    <statement n>;
```

Its related function definition with no return type is equivalent to writing to an array. For example:

```
<function name>[<argument prototypes>] = <value prototype>
    <statement 1>;
    <statement 2>;
    …
    <statement n>;
```

The value prototype determines what data type can be written to the array. The two related functions must share the same name, but the value prototype and return type can be different.

Procedure calls

A procedure call has the form:

```
<procedure_name>(<arguments>);
```

Return statements

A procedure return has the form:

```
return;
```

A function return has the form:

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
return <expression>;
```

where `<expression>` is of the type declared in the function prototype line.

K13.5.3 Conditional control structures

This section describes how conditional control structures are used in the pseudocode.

if … then … else …

In addition to being a ternary operator, a multi-line `if … then … else …` structure can act as a control structure and has the form:

```
if <boolean_expression> then
    <statement 1>;
    <statement 2>;
    …
    <statement n>;

elsif <boolean_expression> then
    <statement a>;
    <statement b>;
    …
    <statement z>;
else
    <statement A>;
    <statement B>;
    …
    <statement Z>;
```

The block of lines consisting of `elsif` and its indented statements is optional, and multiple `elsif` blocks can be used.

The block of lines consisting of `else` and its indented statements is optional.

Abbreviated one-line forms can be used when the `then` part, and in the `else` part if it is present, contain only simple statements such as:

```
if <boolean_expression> then <statement 1>;
if <boolean_expression> then <statement 1>; else <statement A>;
if <boolean_expression> then <statement 1>; <statement 2>; else <statement A>;
```

Note

In these forms, `<statement 1>`, `<statement 2>` and `<statement A>` must be terminated by semicolons. This and the fact that the `else` part is optional distinguish its use as a control structure from its use as a ternary operator.

case … of …

A `case … of …` structure has the form:

```
case <expression> of
    when <literal values1>
        <statement 1>;
        <statement 2>;
        …
        <statement n>;

    when <literal values2>
        <statement 1>;
        <statement 2>;
        …
        <statement n>;

    … more "when" groups if required …
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    otherwise
        <statement A>;
        <statement B>;
        …
        <statement Z>;
```

In this structure, `<literal values1>` and `<literal values2>` consist of literal values of the same type as `<expression>`, separated by commas. There can be additional `when` groups in the structure. Abbreviated one line forms of `when` and `otherwise` parts can be used when they contain only simple statements.

If `<expression>` has a bitstring type, the literal values can also include bitstring literals containing `'x'` bits, known as bitmasks. For details see *Equality and non-equality* on page K13-8018.

K13.5.4 Loop control structures

This section describes the three loop control structures used in the pseudocode.

repeat … until …

A `repeat … until …` structure has the form:

```
repeat
    <statement 1>;
    <statement 2>;
    …
    <statement n>;
until <boolean_expression>;
```

It executes the statement block at least once, and the loop repeats until `<boolean expression>` evaluates to `TRUE`. Variables explicitly declared inside the loop body have scope local to that loop and might not be accessed outside the loop body.

while … do

A `while … do` structure has the form:

```
while <boolean_expression> do
    <statement 1>;
    <statement 2>;
    …
    <statement n>;
```

It begins executing the statement block only if the Boolean expression is true. The loop then runs until the expression is false.

for …

A `for …` structure has the form:

```
for <assignable_expression> = <integer_expr1> to <integer_expr2>
    <statement 1>;
    <statement 2>;
    …
    <statement n>;
```

The `<assignable_expression>` is initialized to `<integer_expr1>` and compared to `<integer_expr2>`. If `<integer_expr1>` is less than `<integer_expr2>`, the loop body is executed and the `<assignable_expression>` incremented by one. This repeats until `<assignable expression>` is more than or equal to `<integer_expr2>`.

There is an alternate form:

```
for <assignable_expression> = <integer_expr1> downto <integer_expr2>
```

where `<integer_expr1>` is decremented after the loop body executes and continues until `<assignable expression>` is less than or equal than `<integer_expr2>`.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K13.5.5 Special statements

This section describes statements with particular architecturally-defined behaviors.

UNDEFINED

This subsection describes the statement:

```
UNDEFINED;
```

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is that the Undefined Instruction exception is taken.

UNPREDICTABLE

This subsection describes the statement:

```
UNPREDICTABLE;
```

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is UNPREDICTABLE.

SEE…

This subsection describes the statement:

```
SEE <reference>;
```

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is that nothing occurs as a result of the current pseudocode because some other piece of pseudocode defines the required behavior. The `<reference>` indicates where that other pseudocode can be found.

It usually refers to another instruction, but can also refer to another encoding or note of the same instruction.

IMPLEMENTATION_DEFINED

This subsection describes the statement:

```
IMPLEMENTATION_DEFINED {"<text>"};
```

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is IMPLEMENTATION DEFINED. An optional `<text>` field can give more information.

K13.5.6 Comments

The pseudocode supports two styles of comments:

- `//` starts a comment that is terminated by the end of the line.
- `/*` starts a comment that is terminated by `*/`.

`/**/` statements might not be nested, and the first `*/` ends the comment.

Note

Comment lines do not require a terminating semicolon.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K13.6 Built-in functions

This section describes:

- *Bitstring manipulation functions*.
- *Arithmetic functions* on page K13-8030.

K13.6.1 Bitstring manipulation functions

The following bitstring manipulation functions are defined:

Bitstring length and most significant bit

If `x` is a bitstring:

- The bitstring length function `Len(x)` returns the length of `x` as an integer.

Bitstring concatenation and replication

If `x` is a bitstring and `n` is an integer with `n >= 0`:

- `Replicate(x, n)` is the bitstring of length `n*Len(x)` consisting of `n` copies of `x` concatenated together.
- `Zeros(n)` = `Replicate('0', n)`.
- `Ones(n)` = `Replicate('1', n)`.

Bitstring count

If `x` is a bitstring, `BitCount(x)` is an integer result equal to the number of bits of `x` that are ones.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Testing a bitstring for being all zero or all ones

If x is a bitstring:

- IsZero(x) produces TRUE if all of the bits of x are zeros and FALSE if any of them are ones
- IsZeroBit(x) produces '1' if all of the bits of x are zeros and '0' if any of them are ones.

IsOnes(x) and IsOnesBit(x) work in the corresponding ways. This means:

```
IsZero(x)    = (BitCount(x) == 0)
IsOnes(x)    = (BitCount(x) == Len(x))
IsZeroBit(x) = if IsZero(x) then '1' else '0'
IsOnesBit(x) = if IsOnes(x) then '1' else '0'
```

Lowest and highest set bits of a bitstring

If x is a bitstring, and N = Len(x):

- LowestSetBit(x) is the minimum bit number of any of the bits of x that are ones. If all of its bits are zeros, LowestSetBit(x) = N.
- HighestSetBit(x) is the maximum bit number of any of the bits of x that are ones. If all of its bits are zeros, HighestSetBit(x) = -1.
- CountLeadingZeroBits(x) is the number of zero bits at the left end of x, in the range 0 to *N*. This means:

  CountLeadingZeroBits(x) = N - 1 - HighestSetBit(x).
- CountLeadingSignBits(x) is the number of copies of the sign bit of x at the left end of x, excluding the sign bit itself, and is in the range 0 to *N*-1. This means:

  CountLeadingSignBits(x) = CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>).

Zero-extension and sign-extension of bitstrings

If x is a bitstring and i is an integer, then ZeroExtend(x, i) is x extended to a length of i bits, by adding sufficient zero bits to its left. That is, if i == Len(x), then ZeroExtend(x, i) = x, and if i > Len(x), then:

```
ZeroExtend(x, i) = Replicate('0', i-Len(x)) : x
```

If x is a bitstring and i is an integer, then SignExtend(x, i) is x extended to a length of i bits, by adding sufficient copies of its leftmost bit to its left. That is, if i == Len(x), then SignExtend(x, i) = x, and if i > Len(x), then:

```
SignExtend(x, i) = Replicate(TopBit(x), i-Len(x)) : x
```

It is a pseudocode error to use either ZeroExtend(x, i) or SignExtend(x, i) in a context where it is possible that i < Len(x).

Converting bitstrings to integers

If x is a bitstring, SInt() is the integer whose two's complement representation is x.

UInt() is the integer whose unsigned representation is x.

Int(x, unsigned) returns either SInt(x) or UInt(x) depending on the value of its second argument.

K13.6.2 Arithmetic functions

This section defines built-in arithmetic functions.

Absolute value

If x is either of type real or integer, Abs(x) returns the absolute value of x. The result is the same type as x.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Rounding and aligning

If `x` is a real:

- `RoundDown(x)` produces the largest integer `n` such that `n <= x`.
- `RoundUp(x)` produces the smallest integer `n` such that `n >= x`.
- `RoundTowardsZero(x)` produces:
  - `RoundDown(x)` if `x > 0.0`.
  - `0` if `x == 0.0`.
  - `RoundUp(x)` if `x < 0.0`.

If `x` and `y` are both of type `integer`, `Align(x, y) = y * (x DIV y)`, and is of type `integer`.

If `x` is of type `bitstring` and `y` is of type `integer`, `Align(x, y) = (Align(UInt(x), y))<Len(x)-1:0>`, and is a `bitstring` of the same length as `x`.

It is a pseudocode error to use either form of `Align(x, y)` in any context where `y` can be 0. In practice, `Align(x, y)` is only used with `y` a constant power of two, and the bitstring form used with `y = 2^n` has the effect of producing its argument with its n low-order bits forced to zero.

Maximum and minimum

If `x` and `y` are integers or reals, then `Max(x, y)` and `Min(x, y)` are their maximum and minimum respectively. x and y must both be of type integer or of type real. The function returns a value of the same type as its operands.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K13.7 Miscellaneous helper procedures and functions

This section lists the prototypes of miscellaneous *helper* procedures and functions used by the pseudocode, together with a brief description of the effect of the procedure or function. The pseudocode does not define the operation of these helper procedures and functions.

Note

Chapter J1 *Armv8 Pseudocode* also has an entry for each of these functions, but currently these entries do not say anything about the effect of the function. When this information is added in Chapter J1 this section will be removed from the manual.

K13.7.1 EndOfInstruction()

This procedure terminates processing of the current instruction.

```
EndOfInstruction();
```

K13.7.2 Hint_Debug()

This procedure supplies a hint to the debug system.

```
Hint_Debug(bits(4) option);
```

K13.7.3 Hint_PreloadData()

This procedure performs a *preload data* hint.

```
Hint_PreloadData(bits(32) address);
```

K13.7.4 Hint_PreloadDataForWrite()

This procedure performs a *preload data* hint with a probability that the use will be for a write.

```
Hint_PreloadDataForWrite(bits(32) address);
```

K13.7.5 Hint_PreloadInstr()

This procedure performs a *preload instructions* hint.

```
Hint_PreloadInstr(bits(32) address);
```

K13.7.6 Hint_Yield()

This procedure performs a *Yield* hint.

```
Hint_Yield();
```

K13.7.7 IsExternalAbort()

This function returns `TRUE` if the abort currently being processed is an External abort and FALSE otherwise. It is used only in exception entry pseudocode.

```
boolean IsExternalAbort(Fault type)
    assert type != Fault_None;

boolean IsExternalAbort(FaultRecord fault);
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K13.7.8 IsAsyncAbort()

This function returns TRUE if the abort currently being processed is an asynchronous abort, and FALSE otherwise. It is used only in exception entry pseudocode.

```
boolean IsAsyncAbort(Fault type)
    assert type != Fault_None;

boolean IsAsyncAbort(FaultRecord fault);
```

K13.7.9 LSInstructionSyndrome()

This function returns the extended syndrome information for a fault reported in the HSR.

```
bits(11) LSInstructionSyndrome();
```

K13.7.10 ProcessorID()

This function returns an integer that uniquely identifies the executing PE in the system.

```
integer ProcessorID();
```

K13.7.11 RemapRegsHaveResetValues()

This function returns TRUE if the remap registers PRRR and NMRR have their IMPLEMENTATION DEFINED reset values, and FALSE otherwise.

```
boolean RemapRegsHaveResetValues();
```

K13.7.12 ResetControlRegisters()

This function resets the System registers and memory-mapped control registers that have architecturally-defined reset values to those values. For more information about the affected registers see:

- *PE state on reset to AArch64 state* on page D1-2336.
- *PE state on reset into AArch32 state* on page G1-5761.

```
AArch64.ResetControlRegisters(boolean ResetIsCold)
AArch32.ResetControlRegisters(boolean ResetIsCold)
```

K13.7.13 ThisInstr()

This function returns the bitstring encoding of the currently-executing instruction.

```
bits(32) ThisInstr();
```

Note

Currently, this function is used only on 32-bit instruction encodings.

K13.7.14 ThisInstrLength()

This function returns the length, in bits, of the current instruction. This means it returns 32 or 16:

```
integer ThisInstrLength();
```


Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K13.8 Arm pseudocode definition index

This section contains the following tables:

- Table K13-3 which contains the pseudocode data types.
- Table K13-4 which contains the pseudocode operators.
- Table K13-5 on page K13-8035 which contains the pseudocode keywords and control structures.
- Table K13-6 on page K13-8036 which contains the statements with special behaviors.

Table K13-3 Index of pseudocode data types

Keyword	Meaning
`array`	Type name for the array type
`bit`	Keyword equivalent to `bits(1)`
`bits(N)`	Type name for the bitstring of length `N` data type
`boolean`	Type name for the Boolean data type
`enumeration`	Keyword to define a new enumeration type
`integer`	Type name for the integer data type
`real`	Type name for the real data type
`type`	Keyword to define a new structure

Table K13-4 Index of pseudocode operators

Operator	Meaning
`-`	Unary minus on integers or reals
	Subtraction of integers, reals and bitstrings
	Used in the left-hand side of an assignment or a tuple to discard the result
`+`	Unary plus on integers or reals
	Addition of integers, reals and bitstrings
`.`	Extract named member from a list
	Extract named bit or field from a register
`:`	Bitstring concatenation
	Integer range in bitstring extraction operator
`!`	Boolean NOT
`!=`	Comparison for inequality
`(…)`	Around arguments of procedure or function
`[…]`	Around array index
	Around arguments of array-like function
`*`	Multiplication of integers, reals, and bitstrings
`/`	Division of reals

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K13-4 Index of pseudocode operators (continued)

Operator	Meaning
`&&`	Boolean AND
`<`	*Less than* comparison of integers and reals
`<…>`	Slicing of specified bits of bitstring or integer
`<<`	Multiply integer by power of 2
`<=`	*Less than or equal* comparison of integers and reals
`=`	Assignment operator
`==`	Comparison for equality
`>`	*Greater than* comparison of integers and reals
`>=`	*Greater than or equal* comparison of integers and reals
`>>`	Divide integer by power of 2
`||`	Boolean OR
`^`	Exponential operator
`AND`	Bitwise AND of bitstrings
`DIV`	Quotient from integer division
`EOR`	Bitwise EOR of bitstrings
`IN`	Tests membership of a certain expression in a set of values
`MOD`	Remainder from integer division
`NOT`	Bitwise inversion of bitstrings
`OR`	Bitwise OR of bitstrings
`case … of …`	Control structure for the
`if … then … else …`	Condition expression selecting between two values

Table K13-5 Index of pseudocode keywords and control structures

Operator	Meaning
`/*…*/`	Comment delimiters
`//`	Introduces comment terminated by end of line
`FALSE`	One of two values a Boolean can take (other than `TRUE`)
`for … = …to …`	Loop control structure, counting up from the initial value to the upper limit
`for … = … downto …`	Loop control structure, counting down from the initial value to the lower limit
`if … then … else …`	Conditional control structure
`otherwise`	Introduces default case in `case … of …` control structure

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K13-5 Index of pseudocode keywords and control structures (continued)

Operator	Meaning
`repeat … until …`	Loop control structure that runs at least once until the termination condition is satisfied
`return`	Procedure or function return
`TRUE`	One of two values a Boolean can take (other than `FALSE`)
`when`	Introduces specific case in `case … of …` control structure
`while … do …`	Loop control structure that runs until the termination condition is satisfied

Table K13-6 Index of special statements

Keyword	Meaning
`IMPLEMENTATION_DEFINED`	Describes IMPLEMENTATION DEFINED behavior
`SEE`	Points to other pseudocode to use instead
`UNDEFINED`	Cause Undefined Instruction exception
`UNKNOWN`	Unspecified value
`UNPREDICTABLE`	Unspecified behavior

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Appendix K14
Registers Index

This appendix provides indexes to the register descriptions in this manual. It contains the following sections:

- *Introduction and register disambiguation* on page K14-8038.
- *Alphabetical index of AArch64 registers and System instructions* on page K14-8043.
- *Functional index of AArch64 registers and System instructions* on page K14-8058.
- *Alphabetical index of AArch32 registers and System instructions* on page K14-8074.
- *Functional index of AArch32 registers and System instructions* on page K14-8084.
- *Alphabetical index of memory-mapped registers* on page K14-8096.
- *Functional index of memory-mapped registers* on page K14-8103.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K14.1 Introduction and register disambiguation

In some sections of this manual, registers are referred to by a *general name*, where the description applies to more than one context. Generally, this is one of the following:

- The description applies to both AArch32 state and AArch64 state, and therefore the register names could apply to either AArch32 System registers or AArch64 System registers.
- The description applies to multiple Exception levels, and therefore at a particular Exception level the register names need to take the appropriate Exception. level suffix, _EL0, _EL1, _EL2, or _EL3.

The following sections disambiguate the general register names:

- *Register name disambiguation by Execution state*.
- *Register name disambiguation by Exception level* on page K14-8042.

K14.1.1 Register name disambiguation by Execution state

Table K14-1 disambiguates the general names of the registers by Execution state.

Table K14-1 Disambiguation of general names of registers by Execution state

General name	Short description	AArch64 register	AArch32 register
CONTEXTIDR	Context ID	CONTEXTIDR_EL1	CONTEXTIDR
DBGBCR	Debug Breakpoint Control Registers	DBGBCR<n>_EL1	DBGBCR<n>
DBGBVR	Debug Breakpoint Value Registers	DBGBVR<n>_EL1	DBGBVR<n> DBGBXVR<n>
DBGCLAIMCLR	Debug CLAIM Tag Clear register	DBGCLAIMCLR_EL1	DBGCLAIMCLR
DBGCLAIMSET	Debug CLAIM Tag Set register	DBGCLAIMSET_EL1	DBGCLAIMSET
DBGDTRRX	Debug Data Transfer Register, Receive	DBGDTRRX_EL0	DBGDTRRXint
DBGDTRTX	Debug Data Transfer Register, Transmit	DBGDTRTX_EL0	DBGDTRTXint
DBGPRCR	Debug Power Control Register	DBGPRCR_EL1	DBGPRCR
DBGVCR	Debug Vector Catch Register	DBGVCR32_EL2	DBGVCR
DBGWCR	Debug Watchpoint Control Registers	DBGWCR<n>_EL1	DBGWCR<n>
DBGWVR	Debug Watchpoint Value Registers	DBGWVR<n>_EL1	DBGWVR<n>
DCCINT	Debug Comms Channel Interrupt Enable Register	MDCCINT_EL1	DBGDCCINT
DCCSR	Debug Comms Channel Status Register	MDCCSR_EL0	DBGDSCRint
DBGAUTHSTATUS	Debug Authentication Status	DBGAUTHSTATUS_EL1	DBGAUTHSTATUS
DLR	Debug Link Register	DLR_EL0[31:0]	DLR
DSCR	Debug System Control Register	MDSCR_EL1	DBGDSCRext
DSPSR	Debug Saved PE State Register	DSPSR_EL0	DSPSR
FAR	Fault Address Register	FAR_EL1 FAR_EL2 FAR_EL3 HPFAR_EL2	DFAR, IFAR HDFAR, HIFAR FAR_EL3 HPFAR

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-1 Disambiguation of general names of registers by Execution state (continued)

General name	Short description	AArch64 register	AArch32 register
HCR	Hypervisor Configuration Register	HCR_EL2	HCR HCR2
HDCR	Hyp or EL2 Debug Control Register	MDCR_EL2	HDCR
HSCTLR	Hypervisor System Control Register	SCTLR_EL2	HSCTLR
HTTBR	EL2 Translation Table Base Register	TTBR0_EL2	HTTBR
ISR	Interrupt Status Register	ISR_EL1	ISR
MPIDR	Multiprocessor Affinity Register	MPIDR_EL1	MPIDR
OSDLR	OS Double-Lock Register	OSDLR_EL1	DBGOSDLR
OSDTRRX	OS Lock Data Transfer Register, Receive	OSDTRRX_EL1	DBGDTRRXext
OSDTRTX	OS Lock Data Transfer Register, Transmit	OSDTRTX_EL1	DBGDTRTXext
OSECCR	OS Lock Exception Catch Control Register	OSECCR_EL1	DBGOSECCR
OSLAR	OS Lock Access Register	OSLAR_EL1	DBGOSLAR
OSLSR	OS Lock Status Register	OSLSR_EL1	DBGOSLSR
PMMIR	Performance Monitors Machine Identification Register	PMMIR_EL1	PMMIR
SCR	Secure Configuration Register	SCR_EL3	SCR
SCTLR	System Control Register	SCTLR_EL1 SCTLR_EL2 SCTLR_EL3	SCTLR (NS) HSCTLR SCTLR (S)
SDCR	Secure or EL3 Debug Configuration Register	MDCR_EL3	SDCR
SDER	Secure Debug Enable Register	SDER32_EL3	SDER
SPSR	Saved Program Status Register	SPSR_EL1 SPSR_EL2 SPSR_EL3	SPSR (general description) SPSR_abt SPSR_fiq SPSR_hyp SPSR_irq SPSR_mon SPSR_svc SPSR_und
TCR	Translation Control Register	TCR_EL1 TCR_EL2 TCR_EL3 VTCR_EL2	TTBCR(NS) HTCR TTBCR(S) VTCR
TTBR	Translation Table Base Register	TTBR0_EL1 TTBR0_EL2 TTBR0_EL3 TTBR1_EL1 VTTBR_EL2	TTBR0 TTBR1 HTTBR VTTBR

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-1 Disambiguation of general names of registers by Execution state (continued)

General name	Short description	AArch64 register	AArch32 register
VCR	PL1&0 stage 2 Translation Control Register	VTCR_EL2	VTCR
VBAR	Vector Base Address Register	VBAR_EL1 VBAR_EL2 VBAR_EL3	VBAR HVBAR MVBAR
VTTBR	PL1&0 stage 2 Translation Table Base Register	VTTBR_EL2	VTTBR

Table K14-2 disambiguates the general names of the System registers that provide access to the Performance Monitors by Execution state.

Table K14-2 Disambiguation of general names of the Performance Monitors System registers by Execution state

General name	Short description	AArch64 register	AArch32 register
PMCCFILTR	Cycle Count Filter Register	PMCCFILTR_EL0	PMCCFILTR
PMCCNTR	Cycle Count Register	PMCCNTR_EL0	PMCCNTR
PMCEID0	Performance Monitors Cycle Count Filter Register 0	PMCEID0_EL0	PMCEID0
PMCEID1	Performance Monitors Cycle Count Filter Register 1	PMCEID1_EL0	PMCEID1
PMCNTENCLR	Performance Monitors Count Enable Clear register	PMCNTENCLR_EL0	PMINTENCLR
PMCNTENSET	Performance Monitors Count Enable Set register	PMCNTENSET_EL0	PMCNTENSET
PMCR	Performance Monitors Control Register	PMCR_EL0	PMCR
PMEVCNTR<n>	Performance Monitors Event Count Registers, n = 0-30	PMEVCNTR<n>_EL0	PMEVCNTR<n>
PMEVTYPER<n>	Performance Monitors Event Type Registers, n = 0-30	PMEVTYPER<n>_EL0	PMEVTYPER<n>
PMINTENCLR	Performance Monitors Interrupt Enable Clear register	PMINTENCLR_EL1	PMINTENCLR
PMINTENSET	Performance Monitors Interrupt Enable Set register	PMINTENSET_EL1	PMINTENSET
PMMIR	Performance Monitors Machine Identification Register	PMMIR_EL1	PMMIR
PMOVSCLR	Performance Monitors Overflow Flag Status Register	PMOVSCLR_EL0	PMOVSR
PMOVSSET	Performance Monitors Overflow Flag Status Set register	PMOVSSET_EL0	PMOVSSET
PMSELR	Performance Monitors Event Counter Selection Register	PMSELR_EL0	PMSELR
PMSWINC	Performance Monitors Software Increment register	PMSWINC_EL0	PMSWINC
PMUSERENR	Performance Monitors User Enable Register	PMUSERENR_EL0	PMUSERENR
PMXEVCNTR	Performance Monitors Selected Event Count Register	PMXEVCNTR_EL0	PMXEVCNTR
PMXEVTYPER	Performance Monitors Selected Event Type Register	PMXEVTYPER_EL0	PMXEVTYPER

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-3 disambiguates the general names of the System registers that provide access to the Activity Monitors by Execution state.

Table K14-3 Disambiguation of general names of the Activity Monitors System registers by Execution state

General name	Short description	AArch64 register	AArch32 register
AMCR	Activity Monitors Control Register	AMCR_EL0	AMCR
AMCFGR	Activity Monitors Configuration Register	AMCFGR_EL0	AMCFGR
AMCGCR	Activity Monitors Counter Group Configuration Register	AMCGCR_EL0	AMCGCR
AMUSERENR	Activity Monitors User Enable Register	AMUSERENR_EL0	AMUSERENR
AMCNTENCLR0	Activity Monitors Count Enable Clear Register 0	AMCNTENCLR0_EL0	AMCNTENCLR0
AMCNTENSET0	Activity Monitors Count Enable Set Register 0	AMCNTENSET0_EL0	AMCNTENSET0
AMEVCNTR0<n>	Activity Monitors Event Counter Registers 0, n = 0-15	AMEVCNTR0<n>_EL0	AMEVCNTR0<n>
AMEVTYPER0<n>	Activity Monitors Event Type Registers 0, n = 0-15	AMEVTYPER0<n>_EL0	AMEVTYPER0<n>
AMCNTENCLR1	Activity Monitors Count Enable Clear Register 1	AMCNTENCLR1_EL0	AMCNTENCLR1
AMCNTENSET1	Activity Monitors Count Enable Set Register 1	AMCNTENSET1_EL0	AMCNTENSET1
AMEVCNTR1<n>	Activity Monitors Event Counter Registers 1, n = 0-15	AMEVCNTR1<n>_EL0	AMEVCNTR1<n>
AMEVTYPER1<n>	Activity Monitors Event Type Registers 1, n = 0-15	AMEVTYPER1<n>_EL0	AMEVTYPER1<n>

Table K14-4 disambiguates the general names of the System registers that provide access to the Performance Monitors by Execution state.

Table K14-4 Disambiguation of general names of the Generic Timer System registers by Execution state

General name	Short description	AArch64 register	AArch32 register
CNTFRQ	Counter-timer Frequency register	CNTFRQ_EL0	CNTFRQ
CNTHCTL	Counter-timer Hypervisor Control register	CNTHCTL_EL2	CNTHCTL
CNTHP_CTL	Counter-timer Hypervisor Physical Timer Control register	CNTHP_CTL_EL2	CNTHP_CTL
CNTHP_CVAL	Counter-timer Hypervisor Physical Timer CompareValue register	CNTHP_CVAL_EL2	CNTHP_CVAL
CNTHP_TVAL	Counter-timer Hypervisor Physical Timer TimerValue register	CNTHP_TVAL_EL2	CNTHP_TVAL
CNTKCTL	Counter-timer Kernel Control register	CNTKCTL_EL1	CNTKCTL
CNTP_CTL	Counter-timer Physical Timer Control register	CNTP_CTL_EL0	CNTP_CTL
CNTP_CVAL	Counter-timer Physical Timer CompareValue register	CNTP_CVAL_EL0	CNTP_CVAL
CNTP_TVAL	Counter-timer Physical Timer TimerValue register	CNTP_TVAL_EL0	CNTP_TVAL
CNTPCT	Counter-timer Physical Count register	CNTPCT_EL0	CNTPCT
CNTPS_CTL	Counter-timer Physical Secure Timer Control register	CNTPS_CTL_EL1	-
CNTPS_CVAL	Counter-timer Physical Secure Timer CompareValue register	CNTPS_CVAL_EL1	-
CNTPS_TVAL	Counter-timer Physical Secure Timer TimerValue register	CNTPS_TVAL_EL1	-

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-4 Disambiguation of general names of the Generic Timer System registers by Execution state (continued)

General name	Short description	AArch64 register	AArch32 register
CNTV_CTL	Counter-timer Virtual Timer Control register	CNTV_CTL_EL0	CNTV_CTL
CNTV_CVAL	Counter-timer Virtual Timer CompareValue register	CNTV_CVAL_EL0	CNTV_CVAL
CNTV_TVAL	Counter-timer Virtual Timer TimerValue register	CNTV_TVAL_EL0	CNTV_TVAL
CNTVCT	Counter-timer Virtual Count register	CNTVCT_EL0	CNTVCT
CNTVOFF	Counter-timer Virtual Offset register	CNTVOFF_EL2	CNTVOFF

K14.1.2 Register name disambiguation by Exception level

Table K14-5 disambiguates the general names of the AArch64 System registers by Exception level.

Table K14-5 Disambiguation of AArch64 System registers by Exception level

General form	EL0	EL1	EL2	EL3
AFSR0_ELx	-	AFSR0_EL1	AFSR0_EL2	AFSR0_EL3
AFSR1_ELx	-	AFSR1_EL1	AFSR1_EL2	AFSR1_EL3
CONTEXTIDR_ELx	-	CONTEXTIDR_EL1	CONTEXTIDR_EL2	-
CPTR_ELx	-	-	CPTR_EL2	CPTR_EL3
ELR_ELx	-	ELR_EL1	ELR_EL2	ELR_EL3
ESR_ELx	-	ESR_EL1	ESR_EL2	ESR_EL3
FAR_ELx	-	FAR_EL1	FAR_EL2	FAR_EL3
MAIR_ELx	-	MAIR_EL1	MAIR_EL2	MAIR_EL3
RMR_ELx	-	RMR_EL1	RMR_EL2	RMR_EL3
RVBAR_ELx	-	RVBAR_EL1	RVBAR_EL2	RVBAR_EL3
SCTLR_ELx	-	SCTLR_EL1	SCTLR_EL2	SCTLR_EL3
SCXTNUM_ELx	SCXTNUM_EL0	SCXTNUM_EL1	SCXTNUM_EL2	SCXTNUM_EL3
SP_ELx	SP_EL0	SP_EL1	SP_EL2	SP_EL3
SPSR_ELx	-	SPSR_EL1	SPSR_EL2	SPSR_EL3
TCR_ELx	-	TCR_EL1	TCR_EL2	TCR_EL3
TFSR_ELx	TFSRE0_EL1	TFSR_EL1	TFSR_EL2	TFSR_EL3
TTBR0_ELx	-	TTBR0_EL1	TTBR0_EL2	TTBR0_EL3
TTBR1_ELx	-	TTBR1_EL1	-	-
VBAR_ELx	-	VBAR_EL1	VBAR_EL2	VBAR_EL3

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K14.2 Alphabetical index of AArch64 registers and System instructions

This section is an index of AArch64 registers and System instructions in alphabetical order.

Table K14-6 Alphabetical index of AArch64 Registers

Register	Description, see
ACTLR_EL1	*ACTLR_EL1, Auxiliary Control Register (EL1)* on page D13-2877
ACTLR_EL2	*ACTLR_EL2, Auxiliary Control Register (EL2)* on page D13-2879
ACTLR_EL3	*ACTLR_EL3, Auxiliary Control Register (EL3)* on page D13-2881
AFSR0_EL1	*AFSR0_EL1, Auxiliary Fault Status Register 0 (EL1)* on page D13-2883
AFSR0_EL2	*AFSR0_EL2, Auxiliary Fault Status Register 0 (EL2)* on page D13-2886
AFSR0_EL3	*AFSR0_EL3, Auxiliary Fault Status Register 0 (EL3)* on page D13-2889
AFSR1_EL1	*AFSR1_EL1, Auxiliary Fault Status Register 1 (EL1)* on page D13-2891
AFSR1_EL2	*AFSR1_EL2, Auxiliary Fault Status Register 1 (EL2)* on page D13-2894
AFSR1_EL3	*AFSR1_EL3, Auxiliary Fault Status Register 1 (EL3)* on page D13-2897
AIDR_EL1	*AIDR_EL1, Auxiliary ID Register* on page D13-2899
AMAIR_EL1	*AMAIR_EL1, Auxiliary Memory Attribute Indirection Register (EL1)* on page D13-2900
AMAIR_EL2	*AMAIR_EL2, Auxiliary Memory Attribute Indirection Register (EL2)* on page D13-2903
AMAIR_EL3	*AMAIR_EL3, Auxiliary Memory Attribute Indirection Register (EL3)* on page D13-2906
AMCFGR_EL0	*AMCFGR_EL0, Activity Monitors Configuration Register* on page D13-3725
AMCG1IDR_EL0	*AMCG1IDR_EL0, Activity Monitors Counter Group 1 Identification Register* on page D13-3727
AMCGCR_EL0	*AMCGCR_EL0, Activity Monitors Counter Group Configuration Register* on page D13-3729
AMCNTENCLR0_EL0	*AMCNTENCLR0_EL0, Activity Monitors Count Enable Clear Register 0* on page D13-3731
AMCNTENCLR1_EL0	*AMCNTENCLR1_EL0, Activity Monitors Count Enable Clear Register 1* on page D13-3733
AMCNTENSET0_EL0	*AMCNTENSET0_EL0, Activity Monitors Count Enable Set Register 0* on page D13-3735
AMCNTENSET1_EL0	*AMCNTENSET1_EL0, Activity Monitors Count Enable Set Register 1* on page D13-3737
AMCR_EL0	*AMCR_EL0, Activity Monitors Control Register* on page D13-3739
AMEVCNTR0<n>_EL0	*AMEVCNTR0<n>_EL0, Activity Monitors Event Counter Registers 0, n = 0 - 15* on page D13-3741
AMEVCNTR1<n>_EL0	*AMEVCNTR1<n>_EL0, Activity Monitors Event Counter Registers 1, n = 0 - 15* on page D13-3743
AMEVCNTVOFF0<n>_EL2	*AMEVCNTVOFF0<n>_EL2, Activity Monitors Event Counter Virtual Offset Registers 0, n = 0 - 15* on page D13-3745
AMEVCNTVOFF1<n>_EL2	*AMEVCNTVOFF1<n>_EL2, Activity Monitors Event Counter Virtual Offset Registers 1, n = 0 - 15* on page D13-3747
AMEVTYPER0<n>_EL0	*AMEVTYPER0<n>_EL0, Activity Monitors Event Type Registers 0, n = 0 - 15* on page D13-3749
AMEVTYPER1<n>_EL0	*AMEVTYPER1<n>_EL0, Activity Monitors Event Type Registers 1, n = 0 - 15* on page D13-3751
AMUSERENR_EL0	*AMUSERENR_EL0, Activity Monitors User Enable Register* on page D13-3753

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-6 Alphabetical index of AArch64 Registers (continued)

Register	Description, see
APDAKeyHi_EL1	*APDAKeyHi_EL1, Pointer Authentication Key A for Data (bits[127:64])* on page D13-2908
APDAKeyLo_EL1	*APDAKeyLo_EL1, Pointer Authentication Key A for Data (bits[63:0])* on page D13-2910
APDBKeyHi_EL1	*APDBKeyHi_EL1, Pointer Authentication Key B for Data (bits[127:64])* on page D13-2912
APDBKeyLo_EL1	*APDBKeyLo_EL1, Pointer Authentication Key B for Data (bits[63:0])* on page D13-2914
APGAKeyHi_EL1	*APGAKeyHi_EL1, Pointer Authentication Key A for Code (bits[127:64])* on page D13-2916
APGAKeyLo_EL1	*APGAKeyLo_EL1, Pointer Authentication Key A for Code (bits[63:0])* on page D13-2918
APIAKeyHi_EL1	*APIAKeyHi_EL1, Pointer Authentication Key A for Instruction (bits[127:64])* on page D13-2920
APIAKeyLo_EL1	*APIAKeyLo_EL1, Pointer Authentication Key A for Instruction (bits[63:0])* on page D13-2922
APIBKeyHi_EL1	*APIBKeyHi_EL1, Pointer Authentication Key B for Instruction (bits[127:64])* on page D13-2924
APIBKeyLo_EL1	*APIBKeyLo_EL1, Pointer Authentication Key B for Instruction (bits[63:0])* on page D13-2926
AT S12E0R	*AT S12E0R, Address Translate Stages 1 and 2 EL0 Read* on page C5-537
AT S12E0W	*AT S12E0W, Address Translate Stages 1 and 2 EL0 Write* on page C5-539
AT S12E1R	*AT S12E1R, Address Translate Stages 1 and 2 EL1 Read* on page C5-541
AT S12E1W	*AT S12E1W, Address Translate Stages 1 and 2 EL1 Write* on page C5-543
AT S1E0R	*AT S1E0R, Address Translate Stage 1 EL0 Read* on page C5-545
AT S1E0W	*AT S1E0W, Address Translate Stage 1 EL0 Write* on page C5-547
AT S1E1R	*AT S1E1R, Address Translate Stage 1 EL1 Read* on page C5-549
AT S1E1RP	*AT S1E1RP, Address Translate Stage 1 EL1 Read PAN* on page C5-551
AT S1E1W	*AT S1E1W, Address Translate Stage 1 EL1 Write* on page C5-553
AT S1E1WP	*AT S1E1WP, Address Translate Stage 1 EL1 Write PAN* on page C5-555
AT S1E2R	*AT S1E2R, Address Translate Stage 1 EL2 Read* on page C5-557
AT S1E2W	*AT S1E2W, Address Translate Stage 1 EL2 Write* on page C5-558
AT S1E3R	*AT S1E3R, Address Translate Stage 1 EL3 Read* on page C5-559
AT S1E3W	*AT S1E3W, Address Translate Stage 1 EL3 Write* on page C5-560
CCSIDR2_EL1	*CCSIDR2_EL1, Current Cache Size ID Register 2* on page D13-2928
CCSIDR_EL1	*CCSIDR_EL1, Current Cache Size ID Register* on page D13-2930
CFP RCTX	*CFP RCTX, Control Flow Prediction Restriction by Context* on page C5-761
CLIDR_EL1	*CLIDR_EL1, Cache Level ID Register* on page D13-2933
CNTFRQ_EL0	*CNTFRQ_EL0, Counter-timer Frequency register* on page D13-3840
CNTHCTL_EL2	*CNTHCTL_EL2, Counter-timer Hypervisor Control register* on page D13-3842
CNTHP_CTL_EL2	*CNTHP_CTL_EL2, Counter-timer Hypervisor Physical Timer Control register* on page D13-3852

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-6 Alphabetical index of AArch64 Registers (continued)

Register	Description, see
CNTHP_CVAL_EL2	*CNTHP_CVAL_EL2, Counter-timer Physical Timer CompareValue register (EL2)* on page D13-3856
CNTHP_TVAL_EL2	*CNTHP_TVAL_EL2, Counter-timer Physical Timer TimerValue register (EL2)* on page D13-3860
CNTHPS_CTL_EL2	*CNTHPS_CTL_EL2, Counter-timer Secure Physical Timer Control register (EL2)* on page D13-3864
CNTHPS_CVAL_EL2	*CNTHPS_CVAL_EL2, Counter-timer Secure Physical Timer CompareValue register (EL2)* on page D13-3868
CNTHPS_TVAL_EL2	*CNTHPS_TVAL_EL2, Counter-timer Secure Physical Timer TimerValue register (EL2)* on page D13-3872
CNTHV_CTL_EL2	*CNTHV_CTL_EL2, Counter-timer Virtual Timer Control register (EL2)* on page D13-3876
CNTHV_CVAL_EL2	*CNTHV_CVAL_EL2, Counter-timer Virtual Timer CompareValue register (EL2)* on page D13-3880
CNTHV_TVAL_EL2	*CNTHV_TVAL_EL2, Counter-timer Virtual Timer TimerValue Register (EL2)* on page D13-3883
CNTHVS_CTL_EL2	*CNTHVS_CTL_EL2, Counter-timer Secure Virtual Timer Control register (EL2)* on page D13-3886
CNTHVS_CVAL_EL2	*CNTHVS_CVAL_EL2, Counter-timer Secure Virtual Timer CompareValue register (EL2)* on page D13-3890
CNTHVS_TVAL_EL2	*CNTHVS_TVAL_EL2, Counter-timer Secure Virtual Timer TimerValue register (EL2)* on page D13-3894
CNTKCTL_EL1	*CNTKCTL_EL1, Counter-timer Kernel Control register* on page D13-3898
CNTP_CTL_EL0	*CNTP_CTL_EL0, Counter-timer Physical Timer Control register* on page D13-3903
CNTP_CVAL_EL0	*CNTP_CVAL_EL0, Counter-timer Physical Timer CompareValue register* on page D13-3907
CNTP_TVAL_EL0	*CNTP_TVAL_EL0, Counter-timer Physical Timer TimerValue register* on page D13-3911
CNTPCT_EL0	*CNTPCT_EL0, Counter-timer Physical Count register* on page D13-3917
CNTPCTSS_EL0	*CNTPCTSS_EL0, Counter-timer Self-Synchronized Physical Count register* on page D13-3915
CNTPOFF_EL2	*CNTPOFF_EL2, Counter-timer Physical Offset register* on page D13-3921
CNTPS_CTL_EL1	*CNTPS_CTL_EL1, Counter-timer Physical Secure Timer Control register* on page D13-3919
CNTPS_CVAL_EL1	*CNTPS_CVAL_EL1, Counter-timer Physical Secure Timer CompareValue register* on page D13-3923
CNTPS_TVAL_EL1	*CNTPS_TVAL_EL1, Counter-timer Physical Secure Timer TimerValue register* on page D13-3925
CNTV_CTL_EL0	*CNTV_CTL_EL0, Counter-timer Virtual Timer Control register* on page D13-3927
CNTV_CVAL_EL0	*CNTV_CVAL_EL0, Counter-timer Virtual Timer CompareValue register* on page D13-3931
CNTV_TVAL_EL0	*CNTV_TVAL_EL0, Counter-timer Virtual Timer TimerValue register* on page D13-3935
CNTVCT_EL0	*CNTVCT_EL0, Counter-timer Virtual Count register* on page D13-3941
CNTVCTSS_EL0	*CNTVCTSS_EL0, Counter-timer Self-Synchronized Virtual Count register* on page D13-3939
CNTVOFF_EL2	*CNTVOFF_EL2, Counter-timer Virtual Offset register* on page D13-3943

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-6 Alphabetical index of AArch64 Registers (continued)

Register	Description, see
CONTEXTIDR_EL1	*CONTEXTIDR_EL1, Context ID Register (EL1)* on page D13-2936
CONTEXTIDR_EL2	*CONTEXTIDR_EL2, Context ID Register (EL2)* on page D13-2939
CPACR_EL1	*CPACR_EL1, Architectural Feature Access Control Register* on page D13-2942
CPP RCTX	*CPP RCTX, Cache Prefetch Prediction Restriction by Context* on page C5-764
CPTR_EL2	*CPTR_EL2, Architectural Feature Trap Register (EL2)* on page D13-2947
CPTR_EL3	*CPTR_EL3, Architectural Feature Trap Register (EL3)* on page D13-2955
CSSELR_EL1	*CSSELR_EL1, Cache Size Selection Register* on page D13-2959
CTR_EL0	*CTR_EL0, Cache Type Register* on page D13-2961
CurrentEL	*CurrentEL, Current Exception Level* on page C5-390
DACR32_EL2	*DACR32_EL2, Domain Access Control Register* on page D13-2964
DAIF	*DAIF, Interrupt Mask Bits* on page C5-392
DBGAUTHSTATUS_EL1	*DBGAUTHSTATUS_EL1, Debug Authentication Status register* on page D13-3561
DBGBCR<n>_EL1	*DBGBCR<n>_EL1, Debug Breakpoint Control Registers, n = 0 - 15* on page D13-3563
DBGBVR<n>_EL1	*DBGBVR<n>_EL1, Debug Breakpoint Value Registers, n = 0 - 15* on page D13-3568
DBGCLAIMCLR_EL1	*DBGCLAIMCLR_EL1, Debug CLAIM Tag Clear register* on page D13-3574
DBGCLAIMSET_EL1	*DBGCLAIMSET_EL1, Debug CLAIM Tag Set register* on page D13-3576
DBGDTR_EL0	*DBGDTR_EL0, Debug Data Transfer Register, half-duplex* on page D13-3578
DBGDTRRX_EL0	*DBGDTRRX_EL0, Debug Data Transfer Register, Receive* on page D13-3581
DBGDTRTX_EL0	*DBGDTRTX_EL0, Debug Data Transfer Register, Transmit* on page D13-3583
DBGPRCR_EL1	*DBGPRCR_EL1, Debug Power Control Register* on page D13-3585
DBGVCR32_EL2	*DBGVCR32_EL2, Debug Vector Catch Register* on page D13-3588
DBGWCR<n>_EL1	*DBGWCR<n>_EL1, Debug Watchpoint Control Registers, n = 0 - 15* on page D13-3592
DBGWVR<n>_EL1	*DBGWVR<n>_EL1, Debug Watchpoint Value Registers, n = 0 - 15* on page D13-3597
DC CGDSW	*DC CGDSW, Data, Allocation Tag or unified Cache line Clean of Data and Allocation Tags by Set/Way* on page C5-476
DC CGDVAC	*DC CGDVAC, Data, Allocation Tag or unified Cache line Clean of Allocation Tags by VA to PoC* on page C5-478
DC CGDVADP	*DC CGDVADP, Data, Allocation Tag or unified Cache line Clean of Allocation Tags by VA to PoDP* on page C5-480
DC CGDVAP	*DC CGDVAP, Data, Allocation Tag or unified Cache line Clean of Data and Allocation Tags by VA to PoP* on page C5-482
DC CGSW	*DC CGSW, Data, Allocation Tag or unified Cache line Clean of Allocation Tags by Set/Way* on page C5-484
DC CGVAC	*DC CGVAC, Data, Allocation Tag or unified Cache line Clean of Allocation Tags by VA to PoC* on page C5-486

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-6 Alphabetical index of AArch64 Registers (continued)

Register	Description, see
DC CGVADP	*DC CGVADP, Data, Allocation Tag or unified Cache line Clean of Data and Allocation Tags by VA to PoDP* on page C5-488
DC CGVAP	*DC CGVAP, Data, Allocation Tag or unified Cache line Clean of Allocation Tags by VA to PoP* on page C5-490
DC CIGDSW	*DC CIGDSW, Data, Allocation Tag or unified Cache line Clean and Invalidate of Data and Allocation Tags by Set/Way* on page C5-492
DC CIGDVAC	*DC CIGDVAC, Data, Allocation Tag or unified Cache line Clean and Invalidate of Data and Allocation Tags by VA to PoC* on page C5-494
DC CIGSW	*DC CIGSW, Data, Allocation Tag or unified Cache line Clean and Invalidate of Allocation Tags by Set/Way* on page C5-496
DC CIGVAC	*DC CIGVAC, Data, Allocation Tag or unified Cache line Clean and Invalidate of Allocation Tags by VA to PoC* on page C5-498
DC CISW	*DC CISW, Data or unified Cache line Clean and Invalidate by Set/Way* on page C5-500
DC CIVAC	*DC CIVAC, Data or unified Cache line Clean and Invalidate by VA to PoC* on page C5-502
DC CSW	*DC CSW, Data or unified Cache line Clean by Set/Way* on page C5-504
DC CVAC	*DC CVAC, Data or unified Cache line Clean by VA to PoC* on page C5-506
DC CVADP	*DC CVADP, Data or unified Cache line Clean by VA to PoDP* on page C5-508
DC CVAP	*DC CVAP, Data or unified Cache line Clean by VA to PoP* on page C5-510
DC CVAU	*DC CVAU, Data or unified Cache line Clean by VA to PoU* on page C5-512
DC GVA	*DC GVA, Data Cache set Allocation Tag by VA* on page C5-514
DC GZVA	*DC GZVA, Data Cache set Allocation Tags and Zero by VA* on page C5-516
DC IGDSW	*DC IGDSW, Data, Allocation Tag or unified Cache line Invalidate of Data and Allocation Tags by Set/Way* on page C5-518
DC IGDVAC	*DC IGDVAC, Data, Allocation Tag or unified Cache line Invalidate of Allocation Tags by VA to PoC* on page C5-520
DC IGSW	*DC IGSW, Data, Allocation Tag or unified Cache line Invalidate of Allocation Tags by Set/Way* on page C5-522
DC IGVAC	*DC IGVAC, Data, Allocation Tag or unified Cache line Invalidate of Allocation Tags by VA to PoC* on page C5-524
DC ISW	*DC ISW, Data or unified Cache line Invalidate by Set/Way* on page C5-526
DC IVAC	*DC IVAC, Data or unified Cache line Invalidate by VA to PoC* on page C5-528
DC ZVA	*DC ZVA, Data Cache Zero by VA* on page C5-530
DCZID_EL0	*DCZID_EL0, Data Cache Zero ID register* on page D13-2966
DISR_EL1	*DISR_EL1, Deferred Interrupt Status Register* on page D13-3802
DIT	*DIT, Data Independent Timing* on page C5-395
DLR_EL0	*DLR_EL0, Debug Link Register* on page D13-3600

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-6 Alphabetical index of AArch64 Registers (continued)

Register	Description, see
DSPSR_EL0	*DSPSR_EL0, Debug Saved Program Status Register* on page D13-3601
DVP RCTX	*DVP RCTX, Data Value Prediction Restriction by Context* on page C5-767
ELR_EL1	*ELR_EL1, Exception Link Register (EL1)* on page C5-398
ELR_EL2	*ELR_EL2, Exception Link Register (EL2)* on page C5-401
ELR_EL3	*ELR_EL3, Exception Link Register (EL3)* on page C5-404
ERRIDR_EL1	*ERRIDR_EL1, Error Record ID Register* on page D13-3805
ERRSELR_EL1	*ERRSELR_EL1, Error Record Select Register* on page D13-3807
ERXADDR_EL1	*ERXADDR_EL1, Selected Error Record Address Register* on page D13-3809
ERXCTLR_EL1	*ERXCTLR_EL1, Selected Error Record Control Register* on page D13-3811
ERXFR_EL1	*ERXFR_EL1, Selected Error Record Feature Register* on page D13-3813
ERXMISC0_EL1	*ERXMISC0_EL1, Selected Error Record Miscellaneous Register 0* on page D13-3815
ERXMISC1_EL1	*ERXMISC1_EL1, Selected Error Record Miscellaneous Register 1* on page D13-3817
ERXMISC2_EL1	*ERXMISC2_EL1, Selected Error Record Miscellaneous Register 2* on page D13-3819
ERXMISC3_EL1	*ERXMISC3_EL1, Selected Error Record Miscellaneous Register 3* on page D13-3821
ERXPFGCDN_EL1	*ERXPFGCDN_EL1, Selected Pseudo-fault Generation Countdown register* on page D13-3823
ERXPFGCTL_EL1	*ERXPFGCTL_EL1, Selected Pseudo-fault Generation Control register* on page D13-3825
ERXPFGF_EL1	*ERXPFGF_EL1, Selected Pseudo-fault Generation Feature register* on page D13-3827
ERXSTATUS_EL1	*ERXSTATUS_EL1, Selected Error Record Primary Status Register* on page D13-3829
ESR_EL1	*ESR_EL1, Exception Syndrome Register (EL1)* on page D13-2968
ESR_EL2	*ESR_EL2, Exception Syndrome Register (EL2)* on page D13-3008
ESR_EL3	*ESR_EL3, Exception Syndrome Register (EL3)* on page D13-3048
FAR_EL1	*FAR_EL1, Fault Address Register (EL1)* on page D13-3086
FAR_EL2	*FAR_EL2, Fault Address Register (EL2)* on page D13-3090
FAR_EL3	*FAR_EL3, Fault Address Register (EL3)* on page D13-3093
FPCR	*FPCR, Floating-point Control Register* on page C5-406
FPEXC32_EL2	*FPEXC32_EL2, Floating-Point Exception Control register* on page D13-3095
FPSR	*FPSR, Floating-point Status Register* on page C5-411
GCR_EL1	*GCR_EL1, Tag Control Register.* on page D13-3101
GMID_EL1	*GMID_EL1, Multiple tag transfer ID register* on page D13-3103
HACR_EL2	*HACR_EL2, Hypervisor Auxiliary Control Register* on page D13-3104
HAFGRTR_EL2	*HAFGRTR_EL2, Hypervisor Activity Monitors Fine-Grained Read Trap Register* on page D13-3106
HCR_EL2	*HCR_EL2, Hypervisor Configuration Register* on page D13-3118

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-6 Alphabetical index of AArch64 Registers (continued)

Register	Description, see
HDFGRTR_EL2	*HDFGRTR_EL2, Hypervisor Debug Fine-Grained Read Trap Register* on page D13-3147
HDFGWTR_EL2	*HDFGWTR_EL2, Hypervisor Debug Fine-Grained Write Trap Register* on page D13-3163
HFGITR_EL2	*HFGITR_EL2, Hypervisor Fine-Grained Instruction Trap Register* on page D13-3178
HFGRTR_EL2	*HFGRTR_EL2, Hypervisor Fine-Grained Read Trap Register* on page D13-3193
HFGWTR_EL2	*HFGWTR_EL2, Hypervisor Fine-Grained Write Trap Register* on page D13-3206
HPFAR_EL2	*HPFAR_EL2, Hypervisor IPA Fault Address Register* on page D13-3217
HSTR_EL2	*HSTR_EL2, Hypervisor System Trap Register* on page D13-3220
IC IALLU	*IC IALLU, Instruction Cache Invalidate All to PoU* on page C5-532
IC IALLUIS	*IC IALLUIS, Instruction Cache Invalidate All to PoU, Inner Shareable* on page C5-533
IC IVAU	*IC IVAU, Instruction Cache line Invalidate by VA to PoU* on page C5-534
ID_AA64AFR0_EL1	*ID_AA64AFR0_EL1, AArch64 Auxiliary Feature Register 0* on page D13-3223
ID_AA64AFR1_EL1	*ID_AA64AFR1_EL1, AArch64 Auxiliary Feature Register 1* on page D13-3225
ID_AA64DFR0_EL1	*ID_AA64DFR0_EL1, AArch64 Debug Feature Register 0* on page D13-3226
ID_AA64DFR1_EL1	*ID_AA64DFR1_EL1, AArch64 Debug Feature Register 1* on page D13-3230
ID_AA64ISAR0_EL1	*ID_AA64ISAR0_EL1, AArch64 Instruction Set Attribute Register 0* on page D13-3231
ID_AA64ISAR1_EL1	*ID_AA64ISAR1_EL1, AArch64 Instruction Set Attribute Register 1* on page D13-3235
ID_AA64MMFR0_EL1	*ID_AA64MMFR0_EL1, AArch64 Memory Model Feature Register 0* on page D13-3240
ID_AA64MMFR1_EL1	*ID_AA64MMFR1_EL1, AArch64 Memory Model Feature Register 1* on page D13-3244
ID_AA64MMFR2_EL1	*ID_AA64MMFR2_EL1, AArch64 Memory Model Feature Register 2* on page D13-3247
ID_AA64PFR0_EL1	*ID_AA64PFR0_EL1, AArch64 Processor Feature Register 0* on page D13-3252
ID_AA64PFR1_EL1	*ID_AA64PFR1_EL1, AArch64 Processor Feature Register 1* on page D13-3257
ID_AFR0_EL1	*ID_AFR0_EL1, AArch32 Auxiliary Feature Register 0* on page D13-3260
ID_DFR0_EL1	*ID_DFR0_EL1, AArch32 Debug Feature Register 0* on page D13-3262
ID_DFR1_EL1	*ID_DFR1_EL1, Debug Feature Register 1* on page D13-3265
ID_ISAR0_EL1	*ID_ISAR0_EL1, AArch32 Instruction Set Attribute Register 0* on page D13-3267
ID_ISAR1_EL1	*ID_ISAR1_EL1, AArch32 Instruction Set Attribute Register 1* on page D13-3270
ID_ISAR2_EL1	*ID_ISAR2_EL1, AArch32 Instruction Set Attribute Register 2* on page D13-3273
ID_ISAR3_EL1	*ID_ISAR3_EL1, AArch32 Instruction Set Attribute Register 3* on page D13-3276
ID_ISAR4_EL1	*ID_ISAR4_EL1, AArch32 Instruction Set Attribute Register 4* on page D13-3279
ID_ISAR5_EL1	*ID_ISAR5_EL1, AArch32 Instruction Set Attribute Register 5* on page D13-3282
ID_ISAR6_EL1	*ID_ISAR6_EL1, AArch32 Instruction Set Attribute Register 6* on page D13-3285
ID_MMFR0_EL1	*ID_MMFR0_EL1, AArch32 Memory Model Feature Register 0* on page D13-3288

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-6 Alphabetical index of AArch64 Registers (continued)

Register	Description, see
ID_MMFR1_EL1	*ID_MMFR1_EL1, AArch32 Memory Model Feature Register 1* on page D13-3291
ID_MMFR2_EL1	*ID_MMFR2_EL1, AArch32 Memory Model Feature Register 2* on page D13-3295
ID_MMFR3_EL1	*ID_MMFR3_EL1, AArch32 Memory Model Feature Register 3* on page D13-3298
ID_MMFR4_EL1	*ID_MMFR4_EL1, AArch32 Memory Model Feature Register 4* on page D13-3301
ID_MMFR5_EL1	*ID_MMFR5_EL1, AArch32 Memory Model Feature Register 5* on page D13-3304
ID_PFR0_EL1	*ID_PFR0_EL1, AArch32 Processor Feature Register 0* on page D13-3306
ID_PFR1_EL1	*ID_PFR1_EL1, AArch32 Processor Feature Register 1* on page D13-3309
ID_PFR2_EL1	*ID_PFR2_EL1, AArch32 Processor Feature Register 2* on page D13-3313
IFSR32_EL2	*IFSR32_EL2, Instruction Fault Status Register (EL2)* on page D13-3315
ISR_EL1	*ISR_EL1, Interrupt Status Register* on page D13-3320
LORC_EL1	*LORC_EL1, LORegion Control (EL1)* on page D13-3322
LOREA_EL1	*LOREA_EL1, LORegion End Address (EL1)* on page D13-3325
LORID_EL1	*LORID_EL1, LORegionID (EL1)* on page D13-3328
LORN_EL1	*LORN_EL1, LORegion Number (EL1)* on page D13-3330
LORSA_EL1	*LORSA_EL1, LORegion Start Address (EL1)* on page D13-3332
MAIR_EL1	*MAIR_EL1, Memory Attribute Indirection Register (EL1)* on page D13-3335
MAIR_EL2	*MAIR_EL2, Memory Attribute Indirection Register (EL2)* on page D13-3339
MAIR_EL3	*MAIR_EL3, Memory Attribute Indirection Register (EL3)* on page D13-3343
MDCCINT_EL1	*MDCCINT_EL1, Monitor DCC Interrupt Enable Register* on page D13-3608
MDCCSR_EL0	*MDCCSR_EL0, Monitor DCC Status Register* on page D13-3610
MDCR_EL2	*MDCR_EL2, Monitor Debug Configuration Register (EL2)* on page D13-3612
MDCR_EL3	*MDCR_EL3, Monitor Debug Configuration Register (EL3)* on page D13-3622
MDRAR_EL1	*MDRAR_EL1, Monitor Debug ROM Address Register* on page D13-3631
MDSCR_EL1	*MDSCR_EL1, Monitor Debug System Control Register* on page D13-3633
MIDR_EL1	*MIDR_EL1, Main ID Register* on page D13-3346
MPIDR_EL1	*MPIDR_EL1, Multiprocessor Affinity Register* on page D13-3349
MVFR0_EL1	*MVFR0_EL1, AArch32 Media and VFP Feature Register 0* on page D13-3351
MVFR1_EL1	*MVFR1_EL1, AArch32 Media and VFP Feature Register 1* on page D13-3355
MVFR2_EL1	*MVFR2_EL1, AArch32 Media and VFP Feature Register 2* on page D13-3359
NZCV	*NZCV, Condition Flags* on page C5-416
OSDLR_EL1	*OSDLR_EL1, OS Double Lock Register* on page D13-3639
OSDTRRX_EL1	*OSDTRRX_EL1, OS Lock Data Transfer Register, Receive* on page D13-3641

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-6 Alphabetical index of AArch64 Registers (continued)

Register	Description, see
OSDTRTX_EL1	*OSDTRTX_EL1, OS Lock Data Transfer Register, Transmit* on page D13-3643
OSECCR_EL1	*OSECCR_EL1, OS Lock Exception Catch Control Register* on page D13-3645
OSLAR_EL1	*OSLAR_EL1, OS Lock Access Register* on page D13-3647
OSLSR_EL1	*OSLSR_EL1, OS Lock Status Register* on page D13-3649
PAN	*PAN, Privileged Access Never* on page C5-418
PAR_EL1	*PAR_EL1, Physical Address Register* on page D13-3361
PMBIDR_EL1	*PMBIDR_EL1, Profiling Buffer ID Register* on page D13-3756
PMBLIMITR_EL1	*PMBLIMITR_EL1, Profiling Buffer Limit Address Register* on page D13-3758
PMBPTR_EL1	*PMBPTR_EL1, Profiling Buffer Write Pointer Register* on page D13-3761
PMBSR_EL1	*PMBSR_EL1, Profiling Buffer Status/syndrome Register* on page D13-3763
PMCCFILTR_EL0	*PMCCFILTR_EL0, Performance Monitors Cycle Count Filter Register* on page D13-3664
PMCCNTR_EL0	*PMCCNTR_EL0, Performance Monitors Cycle Count Register* on page D13-3668
PMCEID0_EL0	*PMCEID0_EL0, Performance Monitors Common Event Identification register 0* on page D13-3671
PMCEID1_EL0	*PMCEID1_EL0, Performance Monitors Common Event Identification register 1* on page D13-3674
PMCNTENCLR_EL0	*PMCNTENCLR_EL0, Performance Monitors Count Enable Clear register* on page D13-3677
PMCNTENSET_EL0	*PMCNTENSET_EL0, Performance Monitors Count Enable Set register* on page D13-3680
PMCR_EL0	*PMCR_EL0, Performance Monitors Control Register* on page D13-3683
PMEVCNTR<n>_EL0	*PMEVCNTR<n>_EL0, Performance Monitors Event Count Registers, n = 0 - 30* on page D13-3688
PMEVTYPER<n>_EL0	*PMEVTYPER<n>_EL0, Performance Monitors Event Type Registers, n = 0 - 30* on page D13-3691
PMINTENCLR_EL1	*PMINTENCLR_EL1, Performance Monitors Interrupt Enable Clear register* on page D13-3697
PMINTENSET_EL1	*PMINTENSET_EL1, Performance Monitors Interrupt Enable Set register* on page D13-3699
PMMIR_EL1	*PMMIR_EL1, Performance Monitors Machine Identification Register* on page D13-3701
PMOVSCLR_EL0	*PMOVSCLR_EL0, Performance Monitors Overflow Flag Status Clear Register* on page D13-3703
PMOVSSET_EL0	*PMOVSSET_EL0, Performance Monitors Overflow Flag Status Set register* on page D13-3706
PMSCR_EL1	*PMSCR_EL1, Statistical Profiling Control Register (EL1)* on page D13-3768
PMSCR_EL2	*PMSCR_EL2, Statistical Profiling Control Register (EL2)* on page D13-3773
PMSELR_EL0	*PMSELR_EL0, Performance Monitors Event Counter Selection Register* on page D13-3709
PMSEVFR_EL1	*PMSEVFR_EL1, Sampling Event Filter Register* on page D13-3777
PMSFCR_EL1	*PMSFCR_EL1, Sampling Filter Control Register* on page D13-3787

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-6 Alphabetical index of AArch64 Registers (continued)

Register	Description, see
PMSICR_EL1	*PMSICR_EL1, Sampling Interval Counter Register* on page D13-3790
PMSIDR_EL1	*PMSIDR_EL1, Sampling Profiling ID Register* on page D13-3793
PMSIRR_EL1	*PMSIRR_EL1, Sampling Interval Reload Register* on page D13-3796
PMSLATFR_EL1	*PMSLATFR_EL1, Sampling Latency Filter Register* on page D13-3799
PMSWINC_EL0	*PMSWINC_EL0, Performance Monitors Software Increment register* on page D13-3712
PMUSERENR_EL0	*PMUSERENR_EL0, Performance Monitors User Enable Register* on page D13-3714
PMXEVCNTR_EL0	*PMXEVCNTR_EL0, Performance Monitors Selected Event Count Register* on page D13-3718
PMXEVTYPER_EL0	*PMXEVTYPER_EL0, Performance Monitors Selected Event Type Register* on page D13-3721
REVIDR_EL1	*REVIDR_EL1, Revision ID Register* on page D13-3366
RGSR_EL1	*RGSR_EL1, Random Allocation Tag Seed Register.* on page D13-3367
RMR_EL1	*RMR_EL1, Reset Management Register (EL1)* on page D13-3369
RMR_EL2	*RMR_EL2, Reset Management Register (EL2)* on page D13-3371
RMR_EL3	*RMR_EL3, Reset Management Register (EL3)* on page D13-3373
RNDR	*RNDR, Random Number* on page D13-3375
RNDRRS	*RNDRRS, Reseeded Random Number* on page D13-3376
RVBAR_EL1	*RVBAR_EL1, Reset Vector Base Address Register (if EL2 and EL3 not implemented)* on page D13-3377
RVBAR_EL2	*RVBAR_EL2, Reset Vector Base Address Register (if EL3 not implemented)* on page D13-3378
RVBAR_EL3	*RVBAR_EL3, Reset Vector Base Address Register (if EL3 implemented)* on page D13-3379
S3_<op1>_<Cn>_<Cm>_<op2>	*S3_<op1>_<Cn>_<Cm>_<op2>, IMPLEMENTATION DEFINED registers* on page D13-3380
SCR_EL3	*SCR_EL3, Secure Configuration Register* on page D13-3382
SCTLR_EL1	*SCTLR_EL1, System Control Register (EL1)* on page D13-3393
SCTLR_EL2	*SCTLR_EL2, System Control Register (EL2)* on page D13-3409
SCTLR_EL3	*SCTLR_EL3, System Control Register (EL3)* on page D13-3430
SCXTNUM_EL0	*SCXTNUM_EL0, EL0 Read/Write Software Context Number* on page D13-3438
SCXTNUM_EL1	*SCXTNUM_EL1, EL1 Read/Write Software Context Number* on page D13-3440
SCXTNUM_EL2	*SCXTNUM_EL2, EL2 Read/Write Software Context Number* on page D13-3443
SCXTNUM_EL3	*SCXTNUM_EL3, EL3 Read/Write Software Context Number* on page D13-3446
SDER32_EL2	*SDER32_EL2, AArch32 Secure Debug Enable Register* on page D13-3651
SDER32_EL3	*SDER32_EL3, AArch32 Secure Debug Enable Register* on page D13-3653
SP_EL0	*SP_EL0, Stack Pointer (EL0)* on page C5-420
SP_EL1	*SP_EL1, Stack Pointer (EL1)* on page C5-422
SP_EL2	*SP_EL2, Stack Pointer (EL2)* on page C5-424

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-6 Alphabetical index of AArch64 Registers (continued)

Register	Description, see
SP_EL3	*SP_EL3, Stack Pointer (EL3)* on page C5-426
SPSel	*SPSel, Stack Pointer Select* on page C5-427
SPSR_abt	*SPSR_abt, Saved Program Status Register (Abort mode)* on page C5-429
SPSR_EL1	*SPSR_EL1, Saved Program Status Register (EL1)* on page C5-433
SPSR_EL2	*SPSR_EL2, Saved Program Status Register (EL2)* on page C5-442
SPSR_EL3	*SPSR_EL3, Saved Program Status Register (EL3)* on page C5-450
SPSR_fiq	*SPSR_fiq, Saved Program Status Register (FIQ mode)* on page C5-457
SPSR_irq	*SPSR_irq, Saved Program Status Register (IRQ mode)* on page C5-461
SPSR_und	*SPSR_und, Saved Program Status Register (Undefined mode)* on page C5-465
SSBS	*SSBS, Speculative Store Bypass Safe* on page C5-469
TCO	*TCO, Tag Check Override* on page C5-471
TCR_EL1	*TCR_EL1, Translation Control Register (EL1)* on page D13-3448
TCR_EL2	*TCR_EL2, Translation Control Register (EL2)* on page D13-3461
TCR_EL3	*TCR_EL3, Translation Control Register (EL3)* on page D13-3478
TFSR_EL1	*TFSR_EL1, Tag Fault Status Register (EL1)* on page D13-3486
TFSR_EL2	*TFSR_EL2, Tag Fault Status Register (EL2)* on page D13-3490
TFSR_EL3	*TFSR_EL3, Tag Fault Status Register (EL3)* on page D13-3493
TFSRE0_EL1	*TFSRE0_EL1, Tag Fault Status Register (EL0).* on page D13-3484
TLBI ALLE1	*TLBI ALLE1, TLB Invalidate All, EL1* on page C5-562
TLBI ALLE1IS	*TLBI ALLE1IS, TLB Invalidate All, EL1, Inner Shareable* on page C5-564
TLBI ALLE1OS	*TLBI ALLE1OS, TLB Invalidate All, EL1, Outer Shareable* on page C5-566
TLBI ALLE2	*TLBI ALLE2, TLB Invalidate All, EL2* on page C5-568
TLBI ALLE2IS	*TLBI ALLE2IS, TLB Invalidate All, EL2, Inner Shareable* on page C5-569
TLBI ALLE2OS	*TLBI ALLE2OS, TLB Invalidate All, EL2, Outer Shareable* on page C5-570
TLBI ALLE3	*TLBI ALLE3, TLB Invalidate All, EL3* on page C5-571
TLBI ALLE3IS	*TLBI ALLE3IS, TLB Invalidate All, EL3, Inner Shareable* on page C5-572
TLBI ALLE3OS	*TLBI ALLE3OS, TLB Invalidate All, EL3, Outer Shareable* on page C5-573
TLBI ASIDE1	*TLBI ASIDE1, TLB Invalidate by ASID, EL1* on page C5-574
TLBI ASIDE1IS	*TLBI ASIDE1IS, TLB Invalidate by ASID, EL1, Inner Shareable* on page C5-576
TLBI ASIDE1OS	*TLBI ASIDE1OS, TLB Invalidate by ASID, EL1, Outer Shareable* on page C5-578
TLBI IPAS2E1	*TLBI IPAS2E1, TLB Invalidate by Intermediate Physical Address, Stage 2, EL1* on page C5-580
TLBI IPAS2E1IS	*TLBI IPAS2E1IS, TLB Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable* on page C5-583

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-6 Alphabetical index of AArch64 Registers (continued)

Register	Description, see
TLBI IPAS2E1OS	*TLBI IPAS2E1OS, TLB Invalidate by Intermediate Physical Address, Stage 2, EL1, Outer Shareable* on page C5-586
TLBI IPAS2LE1	*TLBI IPAS2LE1, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1* on page C5-589
TLBI IPAS2LE1IS	*TLBI IPAS2LE1IS, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable* on page C5-592
TLBI IPAS2LE1OS	*TLBI IPAS2LE1OS, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable* on page C5-595
TLBI RIPAS2E1	*TLBI RIPAS2E1, TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1* on page C5-598
TLBI RIPAS2E1IS	*TLBI RIPAS2E1IS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable* on page C5-601
TLBI RIPAS2E1OS	*TLBI RIPAS2E1OS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Outer Shareable* on page C5-604
TLBI RIPAS2LE1	*TLBI RIPAS2LE1, TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1* on page C5-607
TLBI RIPAS2LE1IS	*TLBI RIPAS2LE1IS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable* on page C5-610
TLBI RIPAS2LE1OS	*TLBI RIPAS2LE1OS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable* on page C5-613
TLBI RVAAE1	*TLBI RVAAE1, TLB Range Invalidate by VA, All ASID, EL1* on page C5-616
TLBI RVAAE1IS	*TLBI RVAAE1IS, TLB Range Invalidate by VA, All ASID, EL1, Inner Shareable* on page C5-619
TLBI RVAAE1OS	*TLBI RVAAE1OS, TLB Range Invalidate by VA, All ASID, EL1, Outer Shareable* on page C5-622
TLBI RVAALE1	*TLBI RVAALE1, TLB Range Invalidate by VA, All ASID, Last level, EL1* on page C5-625
TLBI RVAALE1IS	*TLBI RVAALE1IS, TLB Range Invalidate by VA, All ASID, Last Level, EL1, Inner Shareable* on page C5-628
TLBI RVAALE1OS	*TLBI RVAALE1OS, TLB Range Invalidate by VA, All ASID, Last Level, EL1, Outer Shareable* on page C5-631
TLBI RVAE1	*TLBI RVAE1, TLB Range Invalidate by VA, EL1* on page C5-634
TLBI RVAE1IS	*TLBI RVAE1IS, TLB Range Invalidate by VA, EL1, Inner Shareable* on page C5-637
TLBI RVAE1OS	*TLBI RVAE1OS, TLB Range Invalidate by VA, EL1, Outer Shareable* on page C5-640
TLBI RVAE2	*TLBI RVAE2, TLB Range Invalidate by VA, EL2* on page C5-643
TLBI RVAE2IS	*TLBI RVAE2IS, TLB Range Invalidate by VA, EL2, Inner Shareable* on page C5-646
TLBI RVAE2OS	*TLBI RVAE2OS, TLB Range Invalidate by VA, EL2, Outer Shareable* on page C5-649
TLBI RVAE3	*TLBI RVAE3, TLB Range Invalidate by VA, EL3* on page C5-652
TLBI RVAE3IS	*TLBI RVAE3IS, TLB Range Invalidate by VA, EL3, Inner Shareable* on page C5-654
TLBI RVAE3OS	*TLBI RVAE3OS, TLB Range Invalidate by VA, EL3, Outer Shareable* on page C5-656

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-6 Alphabetical index of AArch64 Registers (continued)

Register	Description, see
TLBI RVALE1	*TLBI RVALE1, TLB Range Invalidate by VA, Last level, EL1* on page C5-658
TLBI RVALE1IS	*TLBI RVALE1IS, TLB Range Invalidate by VA, Last level, EL1, Inner Shareable* on page C5-661
TLBI RVALE1OS	*TLBI RVALE1OS, TLB Range Invalidate by VA, Last level, EL1, Outer Shareable* on page C5-664
TLBI RVALE2	*TLBI RVALE2, TLB Range Invalidate by VA, Last level, EL2* on page C5-667
TLBI RVALE2IS	*TLBI RVALE2IS, TLB Range Invalidate by VA, Last level, EL2, Inner Shareable* on page C5-670
TLBI RVALE2OS	*TLBI RVALE2OS, TLB Range Invalidate by VA, Last level, EL2, Outer Shareable* on page C5-673
TLBI RVALE3	*TLBI RVALE3, TLB Range Invalidate by VA, Last level, EL3* on page C5-676
TLBI RVALE3IS	*TLBI RVALE3IS, TLB Range Invalidate by VA, Last level, EL3, Inner Shareable* on page C5-678
TLBI RVALE3OS	*TLBI RVALE3OS, TLB Range Invalidate by VA, Last level, EL3, Outer Shareable* on page C5-680
TLBI VAAE1	*TLBI VAAE1, TLB Invalidate by VA, All ASID, EL1* on page C5-682
TLBI VAAE1IS	*TLBI VAAE1IS, TLB Invalidate by VA, All ASID, EL1, Inner Shareable* on page C5-685
TLBI VAAE1OS	*TLBI VAAE1OS, TLB Invalidate by VA, All ASID, EL1, Outer Shareable* on page C5-688
TLBI VAALE1	*TLBI VAALE1, TLB Invalidate by VA, All ASID, Last level, EL1* on page C5-691
TLBI VAALE1IS	*TLBI VAALE1IS, TLB Invalidate by VA, All ASID, Last Level, EL1, Inner Shareable* on page C5-694
TLBI VAALE1OS	*TLBI VAALE1OS, TLB Invalidate by VA, All ASID, Last Level, EL1, Outer Shareable* on page C5-697
TLBI VAE1	*TLBI VAE1, TLB Invalidate by VA, EL1* on page C5-700
TLBI VAE1IS	*TLBI VAE1IS, TLB Invalidate by VA, EL1, Inner Shareable* on page C5-703
TLBI VAE1OS	*TLBI VAE1OS, TLB Invalidate by VA, EL1, Outer Shareable* on page C5-706
TLBI VAE2	*TLBI VAE2, TLB Invalidate by VA, EL2* on page C5-709
TLBI VAE2IS	*TLBI VAE2IS, TLB Invalidate by VA, EL2, Inner Shareable* on page C5-712
TLBI VAE2OS	*TLBI VAE2OS, TLB Invalidate by VA, EL2, Outer Shareable* on page C5-715
TLBI VAE3	*TLBI VAE3, TLB Invalidate by VA, EL3* on page C5-718
TLBI VAE3IS	*TLBI VAE3IS, TLB Invalidate by VA, EL3, Inner Shareable* on page C5-720
TLBI VAE3OS	*TLBI VAE3OS, TLB Invalidate by VA, EL3, Outer Shareable* on page C5-722
TLBI VALE1	*TLBI VALE1, TLB Invalidate by VA, Last level, EL1* on page C5-724
TLBI VALE1IS	*TLBI VALE1IS, TLB Invalidate by VA, Last level, EL1, Inner Shareable* on page C5-727
TLBI VALE1OS	*TLBI VALE1OS, TLB Invalidate by VA, Last level, EL1, Outer Shareable* on page C5-730
TLBI VALE2	*TLBI VALE2, TLB Invalidate by VA, Last level, EL2* on page C5-733
TLBI VALE2IS	*TLBI VALE2IS, TLB Invalidate by VA, Last level, EL2, Inner Shareable* on page C5-736
TLBI VALE2OS	*TLBI VALE2OS, TLB Invalidate by VA, Last level, EL2, Outer Shareable* on page C5-739
TLBI VALE3	*TLBI VALE3, TLB Invalidate by VA, Last level, EL3* on page C5-742

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-6 Alphabetical index of AArch64 Registers (continued)

Register	Description, see
TLBI VALE3IS	*TLBI VALE3IS, TLB Invalidate by VA, Last level, EL3, Inner Shareable* on page C5-744
TLBI VALE3OS	*TLBI VALE3OS, TLB Invalidate by VA, Last level, EL3, Outer Shareable* on page C5-746
TLBI VMALLE1	*TLBI VMALLE1, TLB Invalidate by VMID, All at stage 1, EL1* on page C5-748
TLBI VMALLE1IS	*TLBI VMALLE1IS, TLB Invalidate by VMID, All at stage 1, EL1, Inner Shareable* on page C5-750
TLBI VMALLE1OS	*TLBI VMALLE1OS, TLB Invalidate by VMID, All at stage 1, EL1, Outer Shareable* on page C5-752
TLBI VMALLS12E1	*TLBI VMALLS12E1, TLB Invalidate by VMID, All at Stage 1 and 2, EL1* on page C5-754
TLBI VMALLS12E1IS	*TLBI VMALLS12E1IS, TLB Invalidate by VMID, All at Stage 1 and 2, EL1, Inner Shareable* on page C5-756
TLBI VMALLS12E1OS	*TLBI VMALLS12E1OS, TLB Invalidate by VMID, All at Stage 1 and 2, EL1, Outer Shareable* on page C5-758
TPIDR_EL0	*TPIDR_EL0, EL0 Read/Write Software Thread ID Register* on page D13-3495
TPIDR_EL1	*TPIDR_EL1, EL1 Software Thread ID Register* on page D13-3497
TPIDR_EL2	*TPIDR_EL2, EL2 Software Thread ID Register* on page D13-3499
TPIDR_EL3	*TPIDR_EL3, EL3 Software Thread ID Register* on page D13-3501
TPIDRRO_EL0	*TPIDRRO_EL0, EL0 Read-Only Software Thread ID Register* on page D13-3503
TRFCR_EL1	*TRFCR_EL1, Trace Filter Control Register (EL1)* on page D13-3655
TRFCR_EL2	*TRFCR_EL2, Trace Filter Control Register (EL2)* on page D13-3659
TTBR0_EL1	*TTBR0_EL1, Translation Table Base Register 0 (EL1)* on page D13-3505
TTBR0_EL2	*TTBR0_EL2, Translation Table Base Register 0 (EL2)* on page D13-3510
TTBR0_EL3	*TTBR0_EL3, Translation Table Base Register 0 (EL3)* on page D13-3515
TTBR1_EL1	*TTBR1_EL1, Translation Table Base Register 1 (EL1)* on page D13-3518
TTBR1_EL2	*TTBR1_EL2, Translation Table Base Register 1 (EL2)* on page D13-3523
UAO	*UAO, User Access Override* on page C5-473
VBAR_EL1	*VBAR_EL1, Vector Base Address Register (EL1)* on page D13-3527
VBAR_EL2	*VBAR_EL2, Vector Base Address Register (EL2)* on page D13-3530
VBAR_EL3	*VBAR_EL3, Vector Base Address Register (EL3)* on page D13-3533
VDISR_EL2	*VDISR_EL2, Virtual Deferred Interrupt Status Register* on page D13-3831
VMPIDR_EL2	*VMPIDR_EL2, Virtualization Multiprocessor ID Register* on page D13-3535
VNCR_EL2	*VNCR_EL2, Virtual Nested Control Register* on page D13-3538
VPIDR_EL2	*VPIDR_EL2, Virtualization Processor ID Register* on page D13-3540
VSESR_EL2	*VSESR_EL2, Virtual SError Exception Syndrome Register* on page D13-3836
VSTCR_EL2	*VSTCR_EL2, Virtualization Secure Translation Control Register* on page D13-3543

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-6 Alphabetical index of AArch64 Registers (continued)

Register	Description, see
VSTTBR_EL2	*VSTTBR_EL2, Virtualization Secure Translation Table Base Register* on page D13-3547
VTCR_EL2	*VTCR_EL2, Virtualization Translation Control Register* on page D13-3550
VTTBR_EL2	*VTTBR_EL2, Virtualization Translation Table Base Register* on page D13-3556

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K14.3 Functional index of AArch64 registers and System instructions

This section is an index of the AArch64 registers and System instructions, divided by functional group. Each of the following sections lists the registers for a functional group:

- *Special-purpose registers*.
- *VMSA-specific registers* on page K14-8059.
- *ID registers* on page K14-8060.
- *Performance monitors registers* on page K14-8061.
- *Debug registers* on page K14-8062.
- *RAS registers* on page K14-8063.
- *Generic timer registers* on page K14-8063.
- *Cache maintenance system instructions* on page K14-8064.
- *Address translation system instructions* on page K14-8065.
- *TLB maintenance system instructions* on page K14-8066.
- *Prediction restriction System instructions* on page K14-8069.
- *Base system registers* on page K14-8069.

K14.3.1 Special-purpose registers

This section is an index to the registers in the Special-purpose registers functional group.

Table K14-7 Special-purpose registers

Register	Description, see
ELR_EL1	ELR_EL1
ELR_EL2	ELR_EL2
ELR_EL3	ELR_EL3
SP_EL0	SP_EL0
SP_EL1	SP_EL1
SP_EL2	SP_EL2
SP_EL3	SP_EL3
SPSR_abt	SPSR_abt
SPSR_EL1	SPSR_EL1
SPSR_EL2	SPSR_EL2
SPSR_EL3	SPSR_EL3
SPSR_fiq	SPSR_fiq
SPSR_irq	SPSR_irq
SPSR_und	SPSR_und

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K14.3.2 VMSA-specific registers

This section is an index to the registers in the Virtual memory control registers functional group.

Table K14-8 VMSA-specific registers

Register	Description, see
AMAIR_EL1	AMAIR_EL1
AMAIR_EL2	AMAIR_EL2
AMAIR_EL3	AMAIR_EL3
CONTEXTIDR_EL1	CONTEXTIDR_EL1
CONTEXTIDR_EL2	CONTEXTIDR_EL2
DACR32_EL2	DACR32_EL2
LORC_EL1	LORC_EL1
LOREA_EL1	LOREA_EL1
LORID_EL1	LORID_EL1
LORN_EL1	LORN_EL1
LORSA_EL1	LORSA_EL1
MAIR_EL1	MAIR_EL1
MAIR_EL2	MAIR_EL2
MAIR_EL3	MAIR_EL3
TCR_EL1	TCR_EL1
TCR_EL2	TCR_EL2
TCR_EL3	TCR_EL3
TTBR0_EL1	TTBR0_EL1
TTBR0_EL2	TTBR0_EL2
TTBR0_EL3	TTBR0_EL3
TTBR1_EL1	TTBR1_EL1
TTBR1_EL2	TTBR1_EL2
VTCR_EL2	VTCR_EL2
VTTBR_EL2	VTTBR_EL2

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K14.3.3 ID registers

This section is an index to the registers in the Identification registers functional group.

Table K14-9 ID registers

Register	Description, see
CCSIDR2_EL1	CCSIDR2_EL1
CCSIDR_EL1	CCSIDR_EL1
CLIDR_EL1	CLIDR_EL1
CSSELR_EL1	CSSELR_EL1
CTR_EL0	CTR_EL0
DCZID_EL0	DCZID_EL0
GMID_EL1	GMID_EL1
ID_AA64AFR0_EL1	ID_AA64AFR0_EL1
ID_AA64AFR1_EL1	ID_AA64AFR1_EL1
ID_AA64DFR0_EL1	ID_AA64DFR0_EL1
ID_AA64DFR1_EL1	ID_AA64DFR1_EL1
ID_AA64ISAR0_EL1	ID_AA64ISAR0_EL1
ID_AA64ISAR1_EL1	ID_AA64ISAR1_EL1
ID_AA64MMFR0_EL1	ID_AA64MMFR0_EL1
ID_AA64MMFR1_EL1	ID_AA64MMFR1_EL1
ID_AA64MMFR2_EL1	ID_AA64MMFR2_EL1
ID_AA64PFR0_EL1	ID_AA64PFR0_EL1
ID_AA64PFR1_EL1	ID_AA64PFR1_EL1
ID_AFR0_EL1	ID_AFR0_EL1
ID_DFR0_EL1	ID_DFR0_EL1
ID_DFR1_EL1	ID_DFR1_EL1
ID_ISAR0_EL1	ID_ISAR0_EL1
ID_ISAR1_EL1	ID_ISAR1_EL1
ID_ISAR2_EL1	ID_ISAR2_EL1
ID_ISAR3_EL1	ID_ISAR3_EL1
ID_ISAR4_EL1	ID_ISAR4_EL1
ID_ISAR5_EL1	ID_ISAR5_EL1
ID_ISAR6_EL1	ID_ISAR6_EL1
ID_MMFR0_EL1	ID_MMFR0_EL1
ID_MMFR1_EL1	ID_MMFR1_EL1

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-9 ID registers (continued)

Register	Description, see
ID_MMFR2_EL1	ID_MMFR2_EL1
ID_MMFR3_EL1	ID_MMFR3_EL1
ID_MMFR4_EL1	ID_MMFR4_EL1
ID_MMFR5_EL1	ID_MMFR5_EL1
ID_PFR0_EL1	ID_PFR0_EL1
ID_PFR1_EL1	ID_PFR1_EL1
ID_PFR2_EL1	ID_PFR2_EL1
MIDR_EL1	MIDR_EL1
MPIDR_EL1	MPIDR_EL1
REVIDR_EL1	REVIDR_EL1

K14.3.4 Performance monitors registers

This section is an index to the registers in the Performance Monitors registers functional group.

Table K14-10 Performance monitors registers

Register	Description, see
PMCCFILTR_EL0	PMCCFILTR_EL0
PMCCNTR_EL0	PMCCNTR_EL0
PMCEID0_EL0	PMCEID0_EL0
PMCEID1_EL0	PMCEID1_EL0
PMCNTENCLR_EL0	PMCNTENCLR_EL0
PMCNTENSET_EL0	PMCNTENSET_EL0
PMCR_EL0	PMCR_EL0
PMEVCNTR<n>_EL0	PMEVCNTR<n>_EL0
PMEVTYPER<n>_EL0	PMEVTYPER<n>_EL0
PMINTENCLR_EL1	PMINTENCLR_EL1
PMINTENSET_EL1	PMINTENSET_EL1
PMMIR_EL1	PMMIR_EL1
PMOVSCLR_EL0	PMOVSCLR_EL0
PMOVSSET_EL0	PMOVSSET_EL0
PMSELR_EL0	PMSELR_EL0
PMSWINC_EL0	PMSWINC_EL0

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-10 Performance monitors registers (continued)

Register	Description, see
PMUSERENR_EL0	PMUSERENR_EL0
PMXEVCNTR_EL0	PMXEVCNTR_EL0
PMXEVTYPER_EL0	PMXEVTYPER_EL0

K14.3.5 Debug registers

This section is an index to the registers in the Debug registers functional group.

Table K14-11 Debug registers

Register	Description, see
DBGAUTHSTATUS_EL1	DBGAUTHSTATUS_EL1
DBGBCR<n>_EL1	DBGBCR<n>_EL1
DBGBVR<n>_EL1	DBGBVR<n>_EL1
DBGCLAIMCLR_EL1	DBGCLAIMCLR_EL1
DBGCLAIMSET_EL1	DBGCLAIMSET_EL1
DBGDTR_EL0	DBGDTR_EL0
DBGDTRRX_EL0	DBGDTRRX_EL0
DBGDTRTX_EL0	DBGDTRTX_EL0
DBGPRCR_EL1	DBGPRCR_EL1
DBGVCR32_EL2	DBGVCR32_EL2
DBGWCR<n>_EL1	DBGWCR<n>_EL1
DBGWVR<n>_EL1	DBGWVR<n>_EL1
DLR_EL0	DLR_EL0
DSPSR_EL0	DSPSR_EL0
MDCCINT_EL1	MDCCINT_EL1
MDCCSR_EL0	MDCCSR_EL0
MDRAR_EL1	MDRAR_EL1
MDSCR_EL1	MDSCR_EL1
OSDLR_EL1	OSDLR_EL1
OSDTRRX_EL1	OSDTRRX_EL1
OSDTRTX_EL1	OSDTRTX_EL1
OSECCR_EL1	OSECCR_EL1
OSLAR_EL1	OSLAR_EL1

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-11 Debug registers (continued)

Register	Description, see
OSLSR_EL1	OSLSR_EL1
TRFCR_EL1	TRFCR_EL1
TRFCR_EL2	TRFCR_EL2

K14.3.6 RAS registers

This section is an index to the registers in the RAS registers functional group.

Table K14-12 RAS registers

Register	Description, see
DISR_EL1	DISR_EL1
ERRIDR_EL1	ERRIDR_EL1
ERRSELR_EL1	ERRSELR_EL1
ERXADDR_EL1	ERXADDR_EL1
ERXCTLR_EL1	ERXCTLR_EL1
ERXFR_EL1	ERXFR_EL1
ERXMISC0_EL1	ERXMISC0_EL1
ERXMISC1_EL1	ERXMISC1_EL1
ERXMISC2_EL1	ERXMISC2_EL1
ERXMISC3_EL1	ERXMISC3_EL1
ERXPFGCDN_EL1	ERXPFGCDN_EL1
ERXPFGCTL_EL1	ERXPFGCTL_EL1
ERXPFGF_EL1	ERXPFGF_EL1
ERXSTATUS_EL1	ERXSTATUS_EL1
VDISR_EL2	VDISR_EL2
VSESR_EL2	VSESR_EL2

K14.3.7 Generic timer registers

This section is an index to the registers in the Generic Timer registers functional group.

Table K14-13 Generic timer registers

Register	Description, see
CNTFRQ_EL0	CNTFRQ_EL0
CNTHV_CTL_EL2	CNTHV_CTL_EL2
CNTHV_CVAL_EL2	CNTHV_CVAL_EL2

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-13 Generic timer registers (continued)

Register	Description, see
CNTHV_TVAL_EL2	CNTHV_TVAL_EL2
CNTHVS_CTL_EL2	CNTHVS_CTL_EL2
CNTHVS_CVAL_EL2	CNTHVS_CVAL_EL2
CNTHVS_TVAL_EL2	CNTHVS_TVAL_EL2
CNTKCTL_EL1	CNTKCTL_EL1
CNTP_CTL_EL0	CNTP_CTL_EL0
CNTP_CVAL_EL0	CNTP_CVAL_EL0
CNTP_TVAL_EL0	CNTP_TVAL_EL0
CNTPCT_EL0	CNTPCT_EL0
CNTPCTSS_EL0	CNTPCTSS_EL0
CNTPOFF_EL2	CNTPOFF_EL2
CNTPS_CTL_EL1	CNTPS_CTL_EL1
CNTPS_CVAL_EL1	CNTPS_CVAL_EL1
CNTPS_TVAL_EL1	CNTPS_TVAL_EL1
CNTV_CTL_EL0	CNTV_CTL_EL0
CNTV_CVAL_EL0	CNTV_CVAL_EL0
CNTV_TVAL_EL0	CNTV_TVAL_EL0
CNTVCT_EL0	CNTVCT_EL0
CNTVCTSS_EL0	CNTVCTSS_EL0

K14.3.8 Cache maintenance system instructions

This section is an index to the registers in the Cache maintenance instructions functional group.

Table K14-14 Cache maintenance system instructions

Register	Description, see
DC CGDSW	DC CGDSW
DC CGDVAC	DC CGDVAC
DC CGDVADP	DC CGDVADP
DC CGDVAP	DC CGDVAP
DC CGSW	DC CGSW
DC CGVAC	DC CGVAC
DC CGVADP	DC CGVADP
DC CGVAP	DC CGVAP

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-14 Cache maintenance system instructions (continued)

Register	Description, see
DC CIGDSW	DC CIGDSW
DC CIGDVAC	DC CIGDVAC
DC CIGSW	DC CIGSW
DC CIGVAC	DC CIGVAC
DC CISW	DC CISW
DC CIVAC	DC CIVAC
DC CSW	DC CSW
DC CVAC	DC CVAC
DC CVADP	DC CVADP
DC CVAP	DC CVAP
DC CVAU	DC CVAU
DC GVA	DC GVA
DC GZVA	DC GZVA
DC IGDSW	DC IGDSW
DC IGDVAC	DC IGDVAC
DC IGSW	DC IGSW
DC IGVAC	DC IGVAC
DC ISW	DC ISW
DC IVAC	DC IVAC
DC ZVA	DC ZVA
IC IALLU	IC IALLU
IC IALLUIS	IC IALLUIS
IC IVAU	IC IVAU

K14.3.9 Address translation system instructions

This section is an index to the registers in the Address translation instructions functional group.

Table K14-15 Address translation system instructions

Register	Description, see
AT S12E0R	AT S12E0R
AT S12E0W	AT S12E0W
AT S12E1R	AT S12E1R
AT S12E1W	AT S12E1W

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-15 Address translation system instructions (continued)

Register	Description, see
AT S1E0R	AT S1E0R
AT S1E0W	AT S1E0W
AT S1E1R	AT S1E1R
AT S1E1RP	AT S1E1RP
AT S1E1W	AT S1E1W
AT S1E1WP	AT S1E1WP
AT S1E2R	AT S1E2R
AT S1E2W	AT S1E2W
AT S1E3R	AT S1E3R
AT S1E3W	AT S1E3W

K14.3.10 TLB maintenance system instructions

This section is an index to the registers in the TLB maintenance instructions functional group.

Table K14-16 TLB maintenance system instructions

Register	Description, see
TLBI ALLE1	TLBI ALLE1
TLBI ALLE1IS	TLBI ALLE1IS
TLBI ALLE1OS	TLBI ALLE1OS
TLBI ALLE2	TLBI ALLE2
TLBI ALLE2IS	TLBI ALLE2IS
TLBI ALLE2OS	TLBI ALLE2OS
TLBI ALLE3	TLBI ALLE3
TLBI ALLE3IS	TLBI ALLE3IS
TLBI ALLE3OS	TLBI ALLE3OS
TLBI ASIDE1	TLBI ASIDE1
TLBI ASIDE1IS	TLBI ASIDE1IS
TLBI ASIDE1OS	TLBI ASIDE1OS
TLBI IPAS2E1	TLBI IPAS2E1
TLBI IPAS2E1IS	TLBI IPAS2E1IS
TLBI IPAS2E1OS	TLBI IPAS2E1OS
TLBI IPAS2LE1	TLBI IPAS2LE1
TLBI IPAS2LE1IS	TLBI IPAS2LE1IS

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-16 TLB maintenance system instructions (continued)

Register	Description, see
TLBI IPAS2LE1OS	TLBI IPAS2LE1OS
TLBI RIPAS2E1	TLBI RIPAS2E1
TLBI RIPAS2E1IS	TLBI RIPAS2E1IS
TLBI RIPAS2E1OS	TLBI RIPAS2E1OS
TLBI RIPAS2LE1	TLBI RIPAS2LE1
TLBI RIPAS2LE1IS	TLBI RIPAS2LE1IS
TLBI RIPAS2LE1OS	TLBI RIPAS2LE1OS
TLBI RVAAE1	TLBI RVAAE1
TLBI RVAAE1IS	TLBI RVAAE1IS
TLBI RVAAE1OS	TLBI RVAAE1OS
TLBI RVAALE1	TLBI RVAALE1
TLBI RVAALE1IS	TLBI RVAALE1IS
TLBI RVAALE1OS	TLBI RVAALE1OS
TLBI RVAE1	TLBI RVAE1
TLBI RVAE1IS	TLBI RVAE1IS
TLBI RVAE1OS	TLBI RVAE1OS
TLBI RVAE2	TLBI RVAE2
TLBI RVAE2IS	TLBI RVAE2IS
TLBI RVAE2OS	TLBI RVAE2OS
TLBI RVAE3	TLBI RVAE3
TLBI RVAE3IS	TLBI RVAE3IS
TLBI RVAE3OS	TLBI RVAE3OS
TLBI RVALE1	TLBI RVALE1
TLBI RVALE1IS	TLBI RVALE1IS
TLBI RVALE1OS	TLBI RVALE1OS
TLBI RVALE2	TLBI RVALE2
TLBI RVALE2IS	TLBI RVALE2IS
TLBI RVALE2OS	TLBI RVALE2OS
TLBI RVALE3	TLBI RVALE3
TLBI RVALE3IS	TLBI RVALE3IS
TLBI RVALE3OS	TLBI RVALE3OS
TLBI VAAE1	TLBI VAAE1

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-16 TLB maintenance system instructions (continued)

Register	Description, see
TLBI VAAE1IS	TLBI VAAE1IS
TLBI VAAE1OS	TLBI VAAE1OS
TLBI VAALE1	TLBI VAALE1
TLBI VAALE1IS	TLBI VAALE1IS
TLBI VAALE1OS	TLBI VAALE1OS
TLBI VAE1	TLBI VAE1
TLBI VAE1IS	TLBI VAE1IS
TLBI VAE1OS	TLBI VAE1OS
TLBI VAE2	TLBI VAE2
TLBI VAE2IS	TLBI VAE2IS
TLBI VAE2OS	TLBI VAE2OS
TLBI VAE3	TLBI VAE3
TLBI VAE3IS	TLBI VAE3IS
TLBI VAE3OS	TLBI VAE3OS
TLBI VALE1	TLBI VALE1
TLBI VALE1IS	TLBI VALE1IS
TLBI VALE1OS	TLBI VALE1OS
TLBI VALE2	TLBI VALE2
TLBI VALE2IS	TLBI VALE2IS
TLBI VALE2OS	TLBI VALE2OS
TLBI VALE3	TLBI VALE3
TLBI VALE3IS	TLBI VALE3IS
TLBI VALE3OS	TLBI VALE3OS
TLBI VMALLE1	TLBI VMALLE1
TLBI VMALLE1IS	TLBI VMALLE1IS
TLBI VMALLE1OS	TLBI VMALLE1OS
TLBI VMALLS12E1	TLBI VMALLS12E1
TLBI VMALLS12E1IS	TLBI VMALLS12E1IS
TLBI VMALLS12E1OS	TLBI VMALLS12E1OS

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K14.3.11 Prediction restriction System instructions

This section is an index to the registers in the prediction restriction instructions functional group.

Table K14-17 Prediction restriction System instructions

Register	Description, see
CFP RCTX	CFP RCTX
CPP RCTX	CPP RCTX
DVP RCTX	DVP RCTX

K14.3.12 Base system registers

This section is an index to the registers in the functional group.

Table K14-18 Base system registers

Register	Description, see
ACTLR_EL1	ACTLR_EL1
ACTLR_EL2	ACTLR_EL2
ACTLR_EL3	ACTLR_EL3
AFSR0_EL1	AFSR0_EL1
AFSR0_EL2	AFSR0_EL2
AFSR0_EL3	AFSR0_EL3
AFSR1_EL1	AFSR1_EL1
AFSR1_EL2	AFSR1_EL2
AFSR1_EL3	AFSR1_EL3
AIDR_EL1	AIDR_EL1
AMCFGR_EL0	AMCFGR_EL0
AMCG1IDR_EL0	AMCG1IDR_EL0
AMCGCR_EL0	AMCGCR_EL0
AMCNTENCLR0_EL0	AMCNTENCLR0_EL0
AMCNTENCLR1_EL0	AMCNTENCLR1_EL0
AMCNTENSET0_EL0	AMCNTENSET0_EL0
AMCNTENSET1_EL0	AMCNTENSET1_EL0
AMCR_EL0	AMCR_EL0
AMEVCNTR0<n>_EL0	AMEVCNTR0<n>_EL0
AMEVCNTR1<n>_EL0	AMEVCNTR1<n>_EL0
AMEVCNTVOFF0<n>_EL2	AMEVCNTVOFF0<n>_EL2
AMEVCNTVOFF1<n>_EL2	AMEVCNTVOFF1<n>_EL2

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-18 Base system registers (continued)

Register	Description, see
AMEVTYPER0<n>_EL0	AMEVTYPER0<n>_EL0
AMEVTYPER1<n>_EL0	AMEVTYPER1<n>_EL0
AMUSERENR_EL0	AMUSERENR_EL0
APDAKeyHi_EL1	APDAKeyHi_EL1
APDAKeyLo_EL1	APDAKeyLo_EL1
APDBKeyHi_EL1	APDBKeyHi_EL1
APDBKeyLo_EL1	APDBKeyLo_EL1
APGAKeyHi_EL1	APGAKeyHi_EL1
APGAKeyLo_EL1	APGAKeyLo_EL1
APIAKeyHi_EL1	APIAKeyHi_EL1
APIAKeyLo_EL1	APIAKeyLo_EL1
APIBKeyHi_EL1	APIBKeyHi_EL1
APIBKeyLo_EL1	APIBKeyLo_EL1
CNTHCTL_EL2	CNTHCTL_EL2
CNTHP_CTL_EL2	CNTHP_CTL_EL2
CNTHP_CVAL_EL2	CNTHP_CVAL_EL2
CNTHP_TVAL_EL2	CNTHP_TVAL_EL2
CNTHPS_CTL_EL2	CNTHPS_CTL_EL2
CNTHPS_CVAL_EL2	CNTHPS_CVAL_EL2
CNTHPS_TVAL_EL2	CNTHPS_TVAL_EL2
CNTVOFF_EL2	CNTVOFF_EL2
CPACR_EL1	CPACR_EL1
CPTR_EL2	CPTR_EL2
CPTR_EL3	CPTR_EL3
CurrentEL	CurrentEL
DAIF	DAIF
DIT	DIT
ESR_EL1	ESR_EL1
ESR_EL2	ESR_EL2
ESR_EL3	ESR_EL3
FAR_EL1	FAR_EL1
FAR_EL2	FAR_EL2

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-18 Base system registers (continued)

Register	Description, see
FAR_EL3	FAR_EL3
FPCR	FPCR
FPEXC32_EL2	FPEXC32_EL2
FPSR	FPSR
GCR_EL1	GCR_EL1
HACR_EL2	HACR_EL2
HAFGRTR_EL2	HAFGRTR_EL2
HCR_EL2	HCR_EL2
HDFGRTR_EL2	HDFGRTR_EL2
HDFGWTR_EL2	HDFGWTR_EL2
HFGITR_EL2	HFGITR_EL2
HFGRTR_EL2	HFGRTR_EL2
HFGWTR_EL2	HFGWTR_EL2
HPFAR_EL2	HPFAR_EL2
HSTR_EL2	HSTR_EL2
IFSR32_EL2	IFSR32_EL2
ISR_EL1	ISR_EL1
MDCR_EL2	MDCR_EL2
MDCR_EL3	MDCR_EL3
MVFR0_EL1	MVFR0_EL1
MVFR1_EL1	MVFR1_EL1
MVFR2_EL1	MVFR2_EL1
NZCV	NZCV
PAN	PAN
PAR_EL1	PAR_EL1
PMBIDR_EL1	PMBIDR_EL1
PMBLIMITR_EL1	PMBLIMITR_EL1
PMBPTR_EL1	PMBPTR_EL1
PMBSR_EL1	PMBSR_EL1
PMSCR_EL1	PMSCR_EL1
PMSCR_EL2	PMSCR_EL2
PMSEVFR_EL1	PMSEVFR_EL1

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-18 Base system registers (continued)

Register	Description, see
PMSFCR_EL1	PMSFCR_EL1
PMSICR_EL1	PMSICR_EL1
PMSIDR_EL1	PMSIDR_EL1
PMSIRR_EL1	PMSIRR_EL1
PMSLATFR_EL1	PMSLATFR_EL1
RGSR_EL1	RGSR_EL1
RMR_EL1	RMR_EL1
RMR_EL2	RMR_EL2
RMR_EL3	RMR_EL3
RNDR	RNDR
RNDRRS	RNDRRS
RVBAR_EL1	RVBAR_EL1
RVBAR_EL2	RVBAR_EL2
RVBAR_EL3	RVBAR_EL3
S3_<op1>_<Cn>_<Cm>_<op2>	S3_<op1>_<Cn>_<Cm>_<op2>
SCR_EL3	SCR_EL3
SCTLR_EL1	SCTLR_EL1
SCTLR_EL2	SCTLR_EL2
SCTLR_EL3	SCTLR_EL3
SCXTNUM_EL0	SCXTNUM_EL0
SCXTNUM_EL1	SCXTNUM_EL1
SCXTNUM_EL2	SCXTNUM_EL2
SCXTNUM_EL3	SCXTNUM_EL3
SDER32_EL2	SDER32_EL2
SDER32_EL3	SDER32_EL3
SPSel	SPSel
SSBS	SSBS
TCO	TCO
TFSR_EL1	TFSR_EL1
TFSR_EL2	TFSR_EL2
TFSR_EL3	TFSR_EL3
TFSRE0_EL1	TFSRE0_EL1

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-18 Base system registers (continued)

Register	Description, see
TPIDR_EL0	TPIDR_EL0
TPIDR_EL1	TPIDR_EL1
TPIDR_EL2	TPIDR_EL2
TPIDR_EL3	TPIDR_EL3
TPIDRRO_EL0	TPIDRRO_EL0
UAO	UAO
VBAR_EL1	VBAR_EL1
VBAR_EL2	VBAR_EL2
VBAR_EL3	VBAR_EL3
VMPIDR_EL2	VMPIDR_EL2
VNCR_EL2	VNCR_EL2
VPIDR_EL2	VPIDR_EL2
VSTCR_EL2	VSTCR_EL2
VSTTBR_EL2	VSTTBR_EL2

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K14.4 Alphabetical index of AArch32 registers and System instructions

This section is an index of AArch32 registers and System instructions in alphabetical order.

Table K14-19 Alphabetical index of AArch32 Registers

Register	Description, see
ACTLR	*ACTLR, Auxiliary Control Register* on page G8-6112
ACTLR2	*ACTLR2, Auxiliary Control Register 2* on page G8-6114
ADFSR	*ADFSR, Auxiliary Data Fault Status Register* on page G8-6116
AIDR	*AIDR, Auxiliary ID Register* on page G8-6118
AIFSR	*AIFSR, Auxiliary Instruction Fault Status Register* on page G8-6119
AMAIR0	*AMAIR0, Auxiliary Memory Attribute Indirection Register 0* on page G8-6121
AMAIR1	*AMAIR1, Auxiliary Memory Attribute Indirection Register 1* on page G8-6124
AMCFGR	*AMCFGR, Activity Monitors Configuration Register* on page G8-6760
AMCGCR	*AMCGCR, Activity Monitors Counter Group Configuration Register* on page G8-6763
AMCNTENCLR0	*AMCNTENCLR0, Activity Monitors Count Enable Clear Register 0* on page G8-6765
AMCNTENCLR1	*AMCNTENCLR1, Activity Monitors Count Enable Clear Register 1* on page G8-6767
AMCNTENSET0	*AMCNTENSET0, Activity Monitors Count Enable Set Register 0* on page G8-6770
AMCNTENSET1	*AMCNTENSET1, Activity Monitors Count Enable Set Register 1* on page G8-6772
AMCR	*AMCR, Activity Monitors Control Register* on page G8-6775
AMEVCNTR0<n>	*AMEVCNTR0<n>, Activity Monitors Event Counter Registers 0, n = 0 - 15* on page G8-6778
AMEVCNTR1<n>	*AMEVCNTR1<n>, Activity Monitors Event Counter Registers 1, n = 0 - 15* on page G8-6781
AMEVTYPER0<n>	*AMEVTYPER0<n>, Activity Monitors Event Type Registers 0, n = 0 - 15* on page G8-6785
AMEVTYPER1<n>	*AMEVTYPER1<n>, Activity Monitors Event Type Registers 1, n = 0 - 15* on page G8-6787
AMUSERENR	*AMUSERENR, Activity Monitors User Enable Register* on page G8-6790
APSR	*APSR, Application Program Status Register* on page G8-6127
ATS12NSOPR	*ATS12NSOPR, Address Translate Stages 1 and 2 Non-secure Only PL1 Read* on page G8-6129
ATS12NSOPW	*ATS12NSOPW, Address Translate Stages 1 and 2 Non-secure Only PL1 Write* on page G8-6130
ATS12NSOUR	*ATS12NSOUR, Address Translate Stages 1 and 2 Non-secure Only Unprivileged Read* on page G8-6131
ATS12NSOUW	*ATS12NSOUW, Address Translate Stages 1 and 2 Non-secure Only Unprivileged Write* on page G8-6132
ATS1CPR	*ATS1CPR, Address Translate Stage 1 Current state PL1 Read* on page G8-6133
ATS1CPRP	*ATS1CPRP, Address Translate Stage 1 Current state PL1 Read PAN* on page G8-6134
ATS1CPW	*ATS1CPW, Address Translate Stage 1 Current state PL1 Write* on page G8-6135
ATS1CPWP	*ATS1CPWP, Address Translate Stage 1 Current state PL1 Write PAN* on page G8-6136
ATS1CUR	*ATS1CUR, Address Translate Stage 1 Current state Unprivileged Read* on page G8-6137

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-19 Alphabetical index of AArch32 Registers (continued)

Register	Description, see
ATS1CUW	*ATS1CUW, Address Translate Stage 1 Current state Unprivileged Write* on page G8-6138
ATS1HR	*ATS1HR, Address Translate Stage 1 Hyp mode Read* on page G8-6139
ATS1HW	*ATS1HW, Address Translate Stage 1 Hyp mode Write* on page G8-6141
BPIALL	*BPIALL, Branch Predictor Invalidate All* on page G8-6143
BPIALLIS	*BPIALLIS, Branch Predictor Invalidate All, Inner Shareable* on page G8-6144
BPIMVA	*BPIMVA, Branch Predictor Invalidate by VA* on page G8-6145
CCSIDR	*CCSIDR, Current Cache Size ID Register* on page G8-6146
CCSIDR2	*CCSIDR2, Current Cache Size ID Register 2* on page G8-6149
CFPRCTX	*CFPRCTX, Control Flow Prediction Restriction by Context* on page G8-6151
CLIDR	*CLIDR, Cache Level ID Register* on page G8-6154
CNTFRQ	*CNTFRQ, Counter-timer Frequency register* on page G8-6841
CNTHCTL	*CNTHCTL, Counter-timer Hyp Control register* on page G8-6843
CNTHP_CTL	*CNTHP_CTL, Counter-timer Hyp Physical Timer Control register* on page G8-6846
CNTHP_CVAL	*CNTHP_CVAL, Counter-timer Hyp Physical CompareValue register* on page G8-6850
CNTHP_TVAL	*CNTHP_TVAL, Counter-timer Hyp Physical Timer TimerValue register* on page G8-6854
CNTHPS_CTL	*CNTHPS_CTL, Counter-timer Secure Physical Timer Control Register (EL2)* on page G8-6858
CNTHPS_CVAL	*CNTHPS_CVAL, Counter-timer Secure Physical Timer CompareValue Register (EL2)* on page G8-6862
CNTHPS_TVAL	*CNTHPS_TVAL, Counter-timer Secure Physical Timer TimerValue Register (EL2)* on page G8-6865
CNTHV_CTL	*CNTHV_CTL, Counter-timer Virtual Timer Control register (EL2)* on page G8-6868
CNTHV_CVAL	*CNTHV_CVAL, Counter-timer Virtual Timer CompareValue register (EL2)* on page G8-6871
CNTHV_TVAL	*CNTHV_TVAL, Counter-timer Virtual Timer TimerValue register (EL2)* on page G8-6874
CNTHVS_CTL	*CNTHVS_CTL, Counter-timer Secure Virtual Timer Control Register (EL2)* on page G8-6877
CNTHVS_CVAL	*CNTHVS_CVAL, Counter-timer Secure Virtual Timer CompareValue Register (EL2)* on page G8-6880
CNTHVS_TVAL	*CNTHVS_TVAL, Counter-timer Secure Virtual Timer TimerValue Register (EL2)* on page G8-6883
CNTKCTL	*CNTKCTL, Counter-timer Kernel Control register* on page G8-6886
CNTP_CTL	*CNTP_CTL, Counter-timer Physical Timer Control register* on page G8-6889
CNTP_CVAL	*CNTP_CVAL, Counter-timer Physical Timer CompareValue register* on page G8-6893
CNTP_TVAL	*CNTP_TVAL, Counter-timer Physical Timer TimerValue register* on page G8-6896
CNTPCT	*CNTPCT, Counter-timer Physical Count register* on page G8-6899
CNTPCTSS	*CNTPCTSS, Counter-timer Self-Synchronized Physical Count register* on page G8-6901

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-19 Alphabetical index of AArch32 Registers (continued)

Register	Description, see
CNTV_CTL	*CNTV_CTL, Counter-timer Virtual Timer Control register* on page G8-6903
CNTV_CVAL	*CNTV_CVAL, Counter-timer Virtual Timer CompareValue register* on page G8-6906
CNTV_TVAL	*CNTV_TVAL, Counter-timer Virtual Timer TimerValue register* on page G8-6909
CNTVCT	*CNTVCT, Counter-timer Virtual Count register* on page G8-6912
CNTVCTSS	*CNTVCTSS, Counter-timer Self-Synchronized Virtual Count register* on page G8-6914
CNTVOFF	*CNTVOFF, Counter-timer Virtual Offset register* on page G8-6916
CONTEXTIDR	*CONTEXTIDR, Context ID Register* on page G8-6156
CP15DMB	*CP15DMB, Data Memory Barrier System instruction* on page G8-6159
CP15DSB	*CP15DSB, Data Synchronization Barrier System instruction* on page G8-6161
CP15ISB	*CP15ISB, Instruction Synchronization Barrier System instruction* on page G8-6163
CPACR	*CPACR, Architectural Feature Access Control Register* on page G8-6165
CPPRCTX	*CPPRCTX, Cache Prefetch Prediction Restriction by Context* on page G8-6174
CPSR	*CPSR, Current Program Status Register* on page G8-6169
CSSELR	*CSSELR, Cache Size Selection Register* on page G8-6177
CTR	*CTR, Cache Type Register* on page G8-6180
DACR	*DACR, Domain Access Control Register* on page G8-6183
DBGAUTHSTATUS	*DBGAUTHSTATUS, Debug Authentication Status register* on page G8-6587
DBGBCR<n>	*DBGBCR<n>, Debug Breakpoint Control Registers, n = 0 - 15* on page G8-6589
DBGBVR<n>	*DBGBVR<n>, Debug Breakpoint Value Registers, n = 0 - 15* on page G8-6594
DBGBXVR<n>	*DBGBXVR<n>, Debug Breakpoint Extended Value Registers, n = 0 - 15* on page G8-6598
DBGCLAIMCLR	*DBGCLAIMCLR, Debug CLAIM Tag Clear register* on page G8-6601
DBGCLAIMSET	*DBGCLAIMSET, Debug CLAIM Tag Set register* on page G8-6603
DBGDCCINT	*DBGDCCINT, DCC Interrupt Enable Register* on page G8-6605
DBGDEVID	*DBGDEVID, Debug Device ID register 0* on page G8-6608
DBGDEVID1	*DBGDEVID1, Debug Device ID register 1* on page G8-6611
DBGDEVID2	*DBGDEVID2, Debug Device ID register 2* on page G8-6613
DBGDIDR	*DBGDIDR, Debug ID Register* on page G8-6614
DBGDRAR	*DBGDRAR, Debug ROM Address Register* on page G8-6617
DBGDSAR	*DBGDSAR, Debug Self Address Register* on page G8-6620
DBGDSCRext	*DBGDSCRext, Debug Status and Control Register, External View* on page G8-6622
DBGDSCRint	*DBGDSCRint, Debug Status and Control Register, Internal View* on page G8-6628
DBGDTRRXext	*DBGDTRRXext, Debug OS Lock Data Transfer Register, Receive, External View* on page G8-6631

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-19 Alphabetical index of AArch32 Registers (continued)

Register	Description, see
DBGDTRRXint	*DBGDTRRXint, Debug Data Transfer Register, Receive* on page G8-6633
DBGDTRTXext	*DBGDTRTXext, Debug OS Lock Data Transfer Register, Transmit* on page G8-6635
DBGDTRTXint	*DBGDTRTXint, Debug Data Transfer Register, Transmit* on page G8-6637
DBGOSDLR	*DBGOSDLR, Debug OS Double Lock Register* on page G8-6639
DBGOSECCR	*DBGOSECCR, Debug OS Lock Exception Catch Control Register* on page G8-6641
DBGOSLAR	*DBGOSLAR, Debug OS Lock Access Register* on page G8-6643
DBGOSLSR	*DBGOSLSR, Debug OS Lock Status Register* on page G8-6645
DBGPRCR	*DBGPRCR, Debug Power Control Register* on page G8-6647
DBGVCR	*DBGVCR, Debug Vector Catch Register* on page G8-6650
DBGWCR<n>	*DBGWCR<n>, Debug Watchpoint Control Registers, n = 0 - 15* on page G8-6657
DBGWFAR	*DBGWFAR, Debug Watchpoint Fault Address Register* on page G8-6662
DBGWVR<n>	*DBGWVR<n>, Debug Watchpoint Value Registers, n = 0 - 15* on page G8-6664
DCCIMVAC	*DCCIMVAC, Data Cache line Clean and Invalidate by VA to PoC* on page G8-6185
DCCISW	*DCCISW, Data Cache line Clean and Invalidate by Set/Way* on page G8-6186
DCCMVAC	*DCCMVAC, Data Cache line Clean by VA to PoC* on page G8-6188
DCCMVAU	*DCCMVAU, Data Cache line Clean by VA to PoU* on page G8-6189
DCCSW	*DCCSW, Data Cache line Clean by Set/Way* on page G8-6191
DCIMVAC	*DCIMVAC, Data Cache line Invalidate by VA to PoC* on page G8-6193
DCISW	*DCISW, Data Cache line Invalidate by Set/Way* on page G8-6195
DFAR	*DFAR, Data Fault Address Register* on page G8-6197
DFSR	*DFSR, Data Fault Status Register* on page G8-6199
DISR	*DISR, Deferred Interrupt Status Register* on page G8-6793
DLR	*DLR, Debug Link Register* on page G8-6666
DSPSR	*DSPSR, Debug Saved Program Status Register* on page G8-6667
DTLBIALL	*DTLBIALL, Data TLB Invalidate All* on page G8-6206
DTLBIASID	*DTLBIASID, Data TLB Invalidate by ASID match* on page G8-6208
DTLBIMVA	*DTLBIMVA, Data TLB Invalidate by VA* on page G8-6210
DVPRCTX	*DVPRCTX, Data Value Prediction Restriction by Context* on page G8-6212
ELR_hyp	*ELR_hyp, Exception Link Register (Hyp mode)* on page G8-6215
ERRIDR	*ERRIDR, Error Record ID Register* on page G8-6798
ERRSELR	*ERRSELR, Error Record Select Register* on page G8-6800
ERXADDR	*ERXADDR, Selected Error Record Address Register* on page G8-6803

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-19 Alphabetical index of AArch32 Registers (continued)

Register	Description, see
ERXADDR2	*ERXADDR2, Selected Error Record Address Register 2* on page G8-6805
ERXCTLR	*ERXCTLR, Selected Error Record Control Register* on page G8-6807
ERXCTLR2	*ERXCTLR2, Selected Error Record Control Register 2* on page G8-6809
ERXFR	*ERXFR, Selected Error Record Feature Register* on page G8-6811
ERXFR2	*ERXFR2, Selected Error Record Feature Register 2* on page G8-6813
ERXMISC0	*ERXMISC0, Selected Error Record Miscellaneous Register 0* on page G8-6815
ERXMISC1	*ERXMISC1, Selected Error Record Miscellaneous Register 1* on page G8-6817
ERXMISC2	*ERXMISC2, Selected Error Record Miscellaneous Register 2* on page G8-6819
ERXMISC3	*ERXMISC3, Selected Error Record Miscellaneous Register 3* on page G8-6821
ERXMISC4	*ERXMISC4, Selected Error Record Miscellaneous Register 4* on page G8-6823
ERXMISC5	*ERXMISC5, Selected Error Record Miscellaneous Register 5* on page G8-6825
ERXMISC6	*ERXMISC6, Selected Error Record Miscellaneous Register 6* on page G8-6827
ERXMISC7	*ERXMISC7, Selected Error Record Miscellaneous Register 7* on page G8-6829
ERXSTATUS	*ERXSTATUS, Selected Error Record Primary Status Register* on page G8-6831
FCSEIDR	*FCSEIDR, FCSE Process ID register* on page G8-6216
FPEXC	*FPEXC, Floating-Point Exception Control register* on page G8-6218
FPSCR	*FPSCR, Floating-Point Status and Control Register* on page G8-6224
FPSID	*FPSID, Floating-Point System ID register* on page G8-6231
HACR	*HACR, Hyp Auxiliary Configuration Register* on page G8-6234
HACTLR	*HACTLR, Hyp Auxiliary Control Register* on page G8-6236
HACTLR2	*HACTLR2, Hyp Auxiliary Control Register 2* on page G8-6238
HADFSR	*HADFSR, Hyp Auxiliary Data Fault Status Register* on page G8-6240
HAIFSR	*HAIFSR, Hyp Auxiliary Instruction Fault Status Register* on page G8-6242
HAMAIR0	*HAMAIR0, Hyp Auxiliary Memory Attribute Indirection Register 0* on page G8-6244
HAMAIR1	*HAMAIR1, Hyp Auxiliary Memory Attribute Indirection Register 1* on page G8-6246
HCPTR	*HCPTR, Hyp Architectural Feature Trap Register* on page G8-6248
HCR	*HCR, Hyp Configuration Register* on page G8-6253
HCR2	*HCR2, Hyp Configuration Register 2* on page G8-6263
HDCR	*HDCR, Hyp Debug Control Register* on page G8-6671
HDFAR	*HDFAR, Hyp Data Fault Address Register* on page G8-6268
HIFAR	*HIFAR, Hyp Instruction Fault Address Register* on page G8-6270
HMAIR0	*HMAIR0, Hyp Memory Attribute Indirection Register 0* on page G8-6272

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-19 Alphabetical index of AArch32 Registers (continued)

Register	Description, see
HMAIR1	*HMAIR1, Hyp Memory Attribute Indirection Register 1* on page G8-6275
HPFAR	*HPFAR, Hyp IPA Fault Address Register* on page G8-6278
HRMR	*HRMR, Hyp Reset Management Register* on page G8-6280
HSCTLR	*HSCTLR, Hyp System Control Register* on page G8-6282
HSR	*HSR, Hyp Syndrome Register* on page G8-6288
HSTR	*HSTR, Hyp System Trap Register* on page G8-6309
HTCR	*HTCR, Hyp Translation Control Register* on page G8-6311
HTPIDR	*HTPIDR, Hyp Software Thread ID Register* on page G8-6315
HTRFCR	*HTRFCR, Hyp Trace Filter Control Register* on page G8-6678
HTTBR	*HTTBR, Hyp Translation Table Base Register* on page G8-6317
HVBAR	*HVBAR, Hyp Vector Base Address Register* on page G8-6320
ICIALLU	*ICIALLU, Instruction Cache Invalidate All to PoU* on page G8-6322
ICIALLUIS	*ICIALLUIS, Instruction Cache Invalidate All to PoU, Inner Shareable* on page G8-6323
ICIMVAU	*ICIMVAU, Instruction Cache line Invalidate by VA to PoU* on page G8-6324
ID_AFR0	*ID_AFR0, Auxiliary Feature Register 0* on page G8-6326
ID_DFR0	*ID_DFR0, Debug Feature Register 0* on page G8-6328
ID_DFR1	*ID_DFR1, Debug Feature Register 1* on page G8-6331
ID_ISAR0	*ID_ISAR0, Instruction Set Attribute Register 0* on page G8-6333
ID_ISAR1	*ID_ISAR1, Instruction Set Attribute Register 1* on page G8-6336
ID_ISAR2	*ID_ISAR2, Instruction Set Attribute Register 2* on page G8-6339
ID_ISAR3	*ID_ISAR3, Instruction Set Attribute Register 3* on page G8-6342
ID_ISAR4	*ID_ISAR4, Instruction Set Attribute Register 4* on page G8-6345
ID_ISAR5	*ID_ISAR5, Instruction Set Attribute Register 5* on page G8-6348
ID_ISAR6	*ID_ISAR6, Instruction Set Attribute Register 6* on page G8-6351
ID_MMFR0	*ID_MMFR0, Memory Model Feature Register 0* on page G8-6354
ID_MMFR1	*ID_MMFR1, Memory Model Feature Register 1* on page G8-6357
ID_MMFR2	*ID_MMFR2, Memory Model Feature Register 2* on page G8-6361
ID_MMFR3	*ID_MMFR3, Memory Model Feature Register 3* on page G8-6364
ID_MMFR4	*ID_MMFR4, Memory Model Feature Register 4* on page G8-6367
ID_MMFR5	*ID_MMFR5, Memory Model Feature Register 5* on page G8-6370
ID_PFR0	*ID_PFR0, Processor Feature Register 0* on page G8-6372
ID_PFR1	*ID_PFR1, Processor Feature Register 1* on page G8-6375

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-19 Alphabetical index of AArch32 Registers (continued)

Register	Description, see
ID_PFR2	*ID_PFR2, Processor Feature Register 2* on page G8-6379
IFAR	*IFAR, Instruction Fault Address Register* on page G8-6381
IFSR	*IFSR, Instruction Fault Status Register* on page G8-6383
ISR	*ISR, Interrupt Status Register* on page G8-6388
ITLBIALL	*ITLBIALL, Instruction TLB Invalidate All* on page G8-6390
ITLBIASID	*ITLBIASID, Instruction TLB Invalidate by ASID match* on page G8-6392
ITLBIMVA	*ITLBIMVA, Instruction TLB Invalidate by VA* on page G8-6394
JIDR	*JIDR, Jazelle ID Register* on page G8-6396
JMCR	*JMCR, Jazelle Main Configuration Register* on page G8-6397
JOSCR	*JOSCR, Jazelle OS Control Register* on page G8-6399
MAIR0	*MAIR0, Memory Attribute Indirection Register 0* on page G8-6401
MAIR1	*MAIR1, Memory Attribute Indirection Register 1* on page G8-6404
MIDR	*MIDR, Main ID Register* on page G8-6407
MPIDR	*MPIDR, Multiprocessor Affinity Register* on page G8-6410
MVBAR	*MVBAR, Monitor Vector Base Address Register* on page G8-6412
MVFR0	*MVFR0, Media and VFP Feature Register 0* on page G8-6414
MVFR1	*MVFR1, Media and VFP Feature Register 1* on page G8-6418
MVFR2	*MVFR2, Media and VFP Feature Register 2* on page G8-6422
NMRR	*NMRR, Normal Memory Remap Register* on page G8-6424
NSACR	*NSACR, Non-Secure Access Control Register* on page G8-6427
PAR	*PAR, Physical Address Register* on page G8-6431
PMCCFILTR	*PMCCFILTR, Performance Monitors Cycle Count Filter Register* on page G8-6695
PMCCNTR	*PMCCNTR, Performance Monitors Cycle Count Register* on page G8-6699
PMCEID0	*PMCEID0, Performance Monitors Common Event Identification register 0* on page G8-6704
PMCEID1	*PMCEID1, Performance Monitors Common Event Identification register 1* on page G8-6707
PMCEID2	*PMCEID2, Performance Monitors Common Event Identification register 2* on page G8-6710
PMCEID3	*PMCEID3, Performance Monitors Common Event Identification register 3* on page G8-6712
PMCNTENCLR	*PMCNTENCLR, Performance Monitors Count Enable Clear register* on page G8-6714
PMCNTENSET	*PMCNTENSET, Performance Monitors Count Enable Set register* on page G8-6717
PMCR	*PMCR, Performance Monitors Control Register* on page G8-6720
PMEVCNTR<n>	*PMEVCNTR<n>, Performance Monitors Event Count Registers, n = 0 - 30* on page G8-6726
PMEVTYPER<n>	*PMEVTYPER<n>, Performance Monitors Event Type Registers, n = 0 - 30* on page G8-6729

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-19 Alphabetical index of AArch32 Registers (continued)

Register	Description, see
PMINTENCLR	*PMINTENCLR, Performance Monitors Interrupt Enable Clear register* on page G8-6734
PMINTENSET	*PMINTENSET, Performance Monitors Interrupt Enable Set register* on page G8-6736
PMMIR	*PMMIR, Performance Monitors Machine Identification Register* on page G8-6681
PMOVSR	*PMOVSR, Performance Monitors Overflow Flag Status Register* on page G8-6738
PMOVSSET	*PMOVSSET, Performance Monitors Overflow Flag Status Set register* on page G8-6741
PMSELR	*PMSELR, Performance Monitors Event Counter Selection Register* on page G8-6744
PMSWINC	*PMSWINC, Performance Monitors Software Increment register* on page G8-6747
PMUSERENR	*PMUSERENR, Performance Monitors User Enable Register* on page G8-6749
PMXEVCNTR	*PMXEVCNTR, Performance Monitors Selected Event Count Register* on page G8-6752
PMXEVTYPER	*PMXEVTYPER, Performance Monitors Selected Event Type Register* on page G8-6756
PRRR	*PRRR, Primary Region Remap Register* on page G8-6441
REVIDR	*REVIDR, Revision ID Register* on page G8-6445
RMR	*RMR, Reset Management Register* on page G8-6446
RVBAR	*RVBAR, Reset Vector Base Address Register* on page G8-6448
SCR	*SCR, Secure Configuration Register* on page G8-6450
SCTLR	*SCTLR, System Control Register* on page G8-6455
SDCR	*SDCR, Secure Debug Control Register* on page G8-6683
SDER	*SDER, Secure Debug Enable Register* on page G8-6689
SPSR	*SPSR, Saved Program Status Register* on page G8-6464
SPSR_abt	*SPSR_abt, Saved Program Status Register (Abort mode)* on page G8-6467
SPSR_fiq	*SPSR_fiq, Saved Program Status Register (FIQ mode)* on page G8-6471
SPSR_hyp	*SPSR_hyp, Saved Program Status Register (Hyp mode)* on page G8-6475
SPSR_irq	*SPSR_irq, Saved Program Status Register (IRQ mode)* on page G8-6479
SPSR_mon	*SPSR_mon, Saved Program Status Register (Monitor mode)* on page G8-6483
SPSR_svc	*SPSR_svc, Saved Program Status Register (Supervisor mode)* on page G8-6487
SPSR_und	*SPSR_und, Saved Program Status Register (Undefined mode)* on page G8-6491
TCMTR	*TCMTR, TCM Type Register* on page G8-6495
TLBIALL	*TLBIALL, TLB Invalidate All* on page G8-6496
TLBIALLH	*TLBIALLH, TLB Invalidate All, Hyp mode* on page G8-6498
TLBIALLHIS	*TLBIALLHIS, TLB Invalidate All, Hyp mode, Inner Shareable* on page G8-6499
TLBIALLIS	*TLBIALLIS, TLB Invalidate All, Inner Shareable* on page G8-6500
TLBIALLNSNH	*TLBIALLNSNH, TLB Invalidate All, Non-Secure Non-Hyp* on page G8-6502

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-19 Alphabetical index of AArch32 Registers (continued)

Register	Description, see
TLBIALLNSNHIS	*TLBIALLNSNHIS, TLB Invalidate All, Non-Secure Non-Hyp, Inner Shareable* on page G8-6503
TLBIASID	*TLBIASID, TLB Invalidate by ASID match* on page G8-6504
TLBIASIDIS	*TLBIASIDIS, TLB Invalidate by ASID match, Inner Shareable* on page G8-6506
TLBIIPAS2	*TLBIIPAS2, TLB Invalidate by Intermediate Physical Address, Stage 2* on page G8-6508
TLBIIPAS2IS	*TLBIIPAS2IS, TLB Invalidate by Intermediate Physical Address, Stage 2, Inner Shareable* on page G8-6510
TLBIIPAS2L	*TLBIIPAS2L, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level* on page G8-6512
TLBIIPAS2LIS	*TLBIIPAS2LIS, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, Inner Shareable* on page G8-6514
TLBIMVA	*TLBIMVA, TLB Invalidate by VA* on page G8-6516
TLBIMVAA	*TLBIMVAA, TLB Invalidate by VA, All ASID* on page G8-6518
TLBIMVAAIS	*TLBIMVAAIS, TLB Invalidate by VA, All ASID, Inner Shareable* on page G8-6520
TLBIMVAAL	*TLBIMVAAL, TLB Invalidate by VA, All ASID, Last level* on page G8-6522
TLBIMVAALIS	*TLBIMVAALIS, TLB Invalidate by VA, All ASID, Last level, Inner Shareable* on page G8-6524
TLBIMVAH	*TLBIMVAH, TLB Invalidate by VA, Hyp mode* on page G8-6526
TLBIMVAHIS	*TLBIMVAHIS, TLB Invalidate by VA, Hyp mode, Inner Shareable* on page G8-6528
TLBIMVAIS	*TLBIMVAIS, TLB Invalidate by VA, Inner Shareable* on page G8-6530
TLBIMVAL	*TLBIMVAL, TLB Invalidate by VA, Last level* on page G8-6532
TLBIMVALH	*TLBIMVALH, TLB Invalidate by VA, Last level, Hyp mode* on page G8-6534
TLBIMVALHIS	*TLBIMVALHIS, TLB Invalidate by VA, Last level, Hyp mode, Inner Shareable* on page G8-6536
TLBIMVALIS	*TLBIMVALIS, TLB Invalidate by VA, Last level, Inner Shareable* on page G8-6538
TLBTR	*TLBTR, TLB Type Register* on page G8-6540
TPIDRPRW	*TPIDRPRW, PL1 Software Thread ID Register* on page G8-6542
TPIDRURO	*TPIDRURO, PL0 Read-Only Software Thread ID Register* on page G8-6544
TPIDRURW	*TPIDRURW, PL0 Read/Write Software Thread ID Register* on page G8-6546
TRFCR	*TRFCR, Trace Filter Control Register* on page G8-6691
TTBCR	*TTBCR, Translation Table Base Control Register* on page G8-6548
TTBCR2	*TTBCR2, Translation Table Base Control Register 2* on page G8-6554
TTBR0	*TTBR0, Translation Table Base Register 0* on page G8-6559
TTBR1	*TTBR1, Translation Table Base Register 1* on page G8-6565
VBAR	*VBAR, Vector Base Address Register* on page G8-6571
VDFSR	*VDFSR, Virtual SError Exception Syndrome Register* on page G8-6833

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-19 Alphabetical index of AArch32 Registers (continued)

Register	Description, see
VDISR	*VDISR, Virtual Deferred Interrupt Status Register* on page G8-6835
VMPIDR	*VMPIDR, Virtualization Multiprocessor ID Register* on page G8-6573
VPIDR	*VPIDR, Virtualization Processor ID Register* on page G8-6576
VTCR	*VTCR, Virtualization Translation Control Register* on page G8-6579
VTTBR	*VTTBR, Virtualization Translation Table Base Register* on page G8-6583

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K14.5 Functional index of AArch32 registers and System instructions

This section is an index of the AArch32 registers and System instructions, divided by functional group. Each of the following sections lists the registers for a functional group:

- *Special-purpose registers*.
- *VMSA-specific registers*.
- *ID registers* on page K14-8085.
- *Performance monitors registers* on page K14-8086.
- *Debug registers* on page K14-8087.
- *RAS registers* on page K14-8088.
- *Generic timer registers* on page K14-8089.
- *Cache maintenance system instructions* on page K14-8090.
- *Address translation system instructions* on page K14-8090.
- *TLB maintenance system instructions* on page K14-8091.
- *Legacy feature registers and system instructions* on page K14-8092.
- *Base system registers* on page K14-8093.

K14.5.1 Special-purpose registers

This section is an index to the registers in the Processor state registers functional group.

Table K14-20 Special-purpose registers

Register	Description, see
DLR	DLR
DSPSR	DSPSR
ELR_hyp	ELR_hyp
SPSR	SPSR
SPSR_abt	SPSR_abt
SPSR_fiq	SPSR_fiq
SPSR_hyp	SPSR_hyp
SPSR_irq	SPSR_irq
SPSR_mon	SPSR_mon
SPSR_svc	SPSR_svc
SPSR_und	SPSR_und

K14.5.2 VMSA-specific registers

This section is an index to the registers in the Virtual memory control registers functional group.

Table K14-21 VMSA-specific registers

Register	Description, see
AMAIR0	AMAIR0
AMAIR1	AMAIR1
CONTEXTIDR	CONTEXTIDR

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-21 VMSA-specific registers (continued)

Register	Description, see
DACR	DACR
HAMAIR0	HAMAIR0
HAMAIR1	HAMAIR1
HMAIR0	HMAIR0
HMAIR1	HMAIR1
HTCR	HTCR
HTTBR	HTTBR
MAIR0	MAIR0
MAIR1	MAIR1
NMRR	NMRR
PRRR	PRRR
TTBCR	TTBCR
TTBCR2	TTBCR2
TTBR0	TTBR0
TTBR1	TTBR1
VTCR	VTCR
VTTBR	VTTBR

K14.5.3 ID registers

This section is an index to the registers in the Identification registers functional group.

Table K14-22 ID registers

Register	Description, see
CCSIDR	CCSIDR
CCSIDR2	CCSIDR2
CLIDR	CLIDR
CSSELR	CSSELR
CTR	CTR
ID_AFR0	ID_AFR0
ID_DFR0	ID_DFR0
ID_DFR1	ID_DFR1
ID_ISAR0	ID_ISAR0
ID_ISAR1	ID_ISAR1

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-22 ID registers (continued)

Register	Description, see
ID_ISAR2	ID_ISAR2
ID_ISAR3	ID_ISAR3
ID_ISAR4	ID_ISAR4
ID_ISAR5	ID_ISAR5
ID_ISAR6	ID_ISAR6
ID_MMFR0	ID_MMFR0
ID_MMFR1	ID_MMFR1
ID_MMFR2	ID_MMFR2
ID_MMFR3	ID_MMFR3
ID_MMFR4	ID_MMFR4
ID_MMFR5	ID_MMFR5
ID_PFR0	ID_PFR0
ID_PFR1	ID_PFR1
ID_PFR2	ID_PFR2
MIDR	MIDR
MPIDR	MPIDR
REVIDR	REVIDR
TCMTR	TCMTR
TLBTR	TLBTR

K14.5.4 Performance monitors registers

This section is an index to the registers in the Performance Monitors registers functional group.

Table K14-23 Performance monitors registers

Register	Description, see
PMCCFILTR	PMCCFILTR
PMCCNTR	PMCCNTR
PMCEID0	PMCEID0
PMCEID1	PMCEID1
PMCEID2	PMCEID2
PMCEID3	PMCEID3
PMCNTENCLR	PMCNTENCLR
PMCNTENSET	PMCNTENSET

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-23 Performance monitors registers (continued)

Register	Description, see
PMCR	PMCR
PMEVCNTR<n>	PMEVCNTR<n>
PMEVTYPER<n>	PMEVTYPER<n>
PMINTENCLR	PMINTENCLR
PMINTENSET	PMINTENSET
PMMIR	PMMIR
PMOVSR	PMOVSR
PMOVSSET	PMOVSSET
PMSELR	PMSELR
PMSWINC	PMSWINC
PMUSERENR	PMUSERENR
PMXEVCNTR	PMXEVCNTR
PMXEVTYPER	PMXEVTYPER

K14.5.5 Debug registers

This section is an index to the registers in the Debug registers functional group.

Table K14-24 Debug registers

Register	Description, see
DBGAUTHSTATUS	DBGAUTHSTATUS
DBGBCR<n>	DBGBCR<n>
DBGBVR<n>	DBGBVR<n>
DBGBXVR<n>	DBGBXVR<n>
DBGCLAIMCLR	DBGCLAIMCLR
DBGCLAIMSET	DBGCLAIMSET
DBGDCCINT	DBGDCCINT
DBGDEVID	DBGDEVID
DBGDEVID1	DBGDEVID1
DBGDEVID2	DBGDEVID2
DBGDIDR	DBGDIDR
DBGDRAR	DBGDRAR
DBGDSAR	DBGDSAR
DBGDSCRext	DBGDSCRext

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-24 Debug registers (continued)

Register	Description, see
DBGDSCRint	DBGDSCRint
DBGDTRRXext	DBGDTRRXext
DBGDTRRXint	DBGDTRRXint
DBGDTRTXext	DBGDTRTXext
DBGDTRTXint	DBGDTRTXint
DBGOSDLR	DBGOSDLR
DBGOSECCR	DBGOSECCR
DBGOSLAR	DBGOSLAR
DBGOSLSR	DBGOSLSR
DBGPRCR	DBGPRCR
DBGVCR	DBGVCR
DBGWCR<n>	DBGWCR<n>
DBGWFAR	DBGWFAR
DBGWVR<n>	DBGWVR<n>
TRFCR	TRFCR

K14.5.6 RAS registers

This section is an index to the registers in the RAS registers functional group.

Table K14-25 RAS registers

Register	Description, see
DISR	DISR
ERRIDR	ERRIDR
ERRSELR	ERRSELR
ERXADDR	ERXADDR
ERXADDR2	ERXADDR2
ERXCTLR	ERXCTLR
ERXCTLR2	ERXCTLR2
ERXFR	ERXFR
ERXFR2	ERXFR2
ERXMISC0	ERXMISC0
ERXMISC1	ERXMISC1
ERXMISC2	ERXMISC2

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-25 RAS registers (continued)

Register	Description, see
ERXMISC3	ERXMISC3
ERXMISC4	ERXMISC4
ERXMISC5	ERXMISC5
ERXMISC6	ERXMISC6
ERXMISC7	ERXMISC7
ERXSTATUS	ERXSTATUS
VDFSR	VDFSR
VDISR	VDISR

K14.5.7 Generic timer registers

This section is an index to the registers in the Generic Timer registers functional group.

Table K14-26 Generic timer registers

Register	Description, see
CNTFRQ	CNTFRQ
CNTHP_CTL	CNTHP_CTL
CNTHPS_CTL	CNTHPS_CTL
CNTHPS_CVAL	CNTHPS_CVAL
CNTHPS_TVAL	CNTHPS_TVAL
CNTHV_CTL	CNTHV_CTL
CNTHV_CVAL	CNTHV_CVAL
CNTHV_TVAL	CNTHV_TVAL
CNTHVS_CTL	CNTHVS_CTL
CNTHVS_CVAL	CNTHVS_CVAL
CNTHVS_TVAL	CNTHVS_TVAL
CNTKCTL	CNTKCTL
CNTP_CTL	CNTP_CTL
CNTP_CVAL	CNTP_CVAL
CNTP_TVAL	CNTP_TVAL
CNTPCT	CNTPCT
CNTPCTSS	CNTPCTSS
CNTV_CTL	CNTV_CTL
CNTV_CVAL	CNTV_CVAL

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-26 Generic timer registers (continued)

Register	Description, see
CNTV_TVAL	CNTV_TVAL
CNTVCT	CNTVCT
CNTVCTSS	CNTVCTSS

K14.5.8 Cache maintenance system instructions

This section is an index to the registers in the Cache maintenance instructions functional group.

Table K14-27 Cache maintenance system instructions

Register	Description, see
BPIALL	BPIALL
BPIALLIS	BPIALLIS
BPIMVA	BPIMVA
DCCIMVAC	DCCIMVAC
DCCISW	DCCISW
DCCMVAC	DCCMVAC
DCCMVAU	DCCMVAU
DCCSW	DCCSW
DCIMVAC	DCIMVAC
DCISW	DCISW
ICIALLU	ICIALLU
ICIALLUIS	ICIALLUIS
ICIMVAU	ICIMVAU

K14.5.9 Address translation system instructions

This section is an index to the registers in the Address translation instructions functional group.

Table K14-28 Address translation system instructions

Register	Description, see
ATS12NSOPR	ATS12NSOPR
ATS12NSOPW	ATS12NSOPW
ATS12NSOUR	ATS12NSOUR
ATS12NSOUW	ATS12NSOUW
ATS1CPR	ATS1CPR
ATS1CPRP	ATS1CPRP

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-28 Address translation system instructions (continued)

Register	Description, see
ATS1CPW	ATS1CPW
ATS1CPWP	ATS1CPWP
ATS1CUR	ATS1CUR
ATS1CUW	ATS1CUW
ATS1HR	ATS1HR
ATS1HW	ATS1HW

K14.5.10 TLB maintenance system instructions

This section is an index to the registers in the TLB maintenance instructions functional group.

Table K14-29 TLB maintenance system instructions

Register	Description, see
CFPRCTX	CFPRCTX
CPPRCTX	CPPRCTX
DTLBIALL	DTLBIALL
DTLBIASID	DTLBIASID
DTLBIMVA	DTLBIMVA
DVPRCTX	DVPRCTX
ITLBIALL	ITLBIALL
ITLBIASID	ITLBIASID
ITLBIMVA	ITLBIMVA
TLBIALL	TLBIALL
TLBIALLH	TLBIALLH
TLBIALLHIS	TLBIALLHIS
TLBIALLIS	TLBIALLIS
TLBIALLNSNH	TLBIALLNSNH
TLBIALLNSNHIS	TLBIALLNSNHIS
TLBIASID	TLBIASID
TLBIASIDIS	TLBIASIDIS
TLBIIPAS2	TLBIIPAS2
TLBIIPAS2IS	TLBIIPAS2IS
TLBIIPAS2L	TLBIIPAS2L
TLBIIPAS2LIS	TLBIIPAS2LIS

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-29 TLB maintenance system instructions (continued)

Register	Description, see
TLBIMVA	TLBIMVA
TLBIMVAA	TLBIMVAA
TLBIMVAAIS	TLBIMVAAIS
TLBIMVAAL	TLBIMVAAL
TLBIMVAALIS	TLBIMVAALIS
TLBIMVAH	TLBIMVAH
TLBIMVAHIS	TLBIMVAHIS
TLBIMVAIS	TLBIMVAIS
TLBIMVAL	TLBIMVAL
TLBIMVALH	TLBIMVALH
TLBIMVALHIS	TLBIMVALHIS
TLBIMVALIS	TLBIMVALIS

K14.5.11 Prediction restriction instructions

This section is an index to the registers in the Prediction restriction instructions functional group.

Table K14-30 Prediction restriction System instructions

Register	Description, see
CFPRCTX	CFPRCTX
CPPRCTX	CPPRCTX
DVPRCTX	DVPRCTX

K14.5.12 Legacy feature registers and system instructions

This section is an index to the registers in the Legacy feature registers functional group.

Table K14-31 Legacy feature registers and system instructions

Register	Description, see
CP15DMB	CP15DMB
CP15DSB	CP15DSB
CP15ISB	CP15ISB
FCSEIDR	FCSEIDR
JIDR	JIDR
JMCR	JMCR
JOSCR	JOSCR

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K14.5.13 Base system registers

This section is an index to the registers in the functional group.

Table K14-32 Base system registers

Register	Description, see
ACTLR	ACTLR
ACTLR2	ACTLR2
ADFSR	ADFSR
AIDR	AIDR
AIFSR	AIFSR
AMCFGR	AMCFGR
AMCGCR	AMCGCR
AMCNTENCLR0	AMCNTENCLR0
AMCNTENCLR1	AMCNTENCLR1
AMCNTENSET0	AMCNTENSET0
AMCNTENSET1	AMCNTENSET1
AMCR	AMCR
AMEVCNTR0<n>	AMEVCNTR0<n>
AMEVCNTR1<n>	AMEVCNTR1<n>
AMEVTYPER0<n>	AMEVTYPER0<n>
AMEVTYPER1<n>	AMEVTYPER1<n>
AMUSERENR	AMUSERENR
APSR	APSR
CNTHCTL	CNTHCTL
CNTHP_CVAL	CNTHP_CVAL
CNTHP_TVAL	CNTHP_TVAL
CNTVOFF	CNTVOFF
CPACR	CPACR
CPSR	CPSR
DFAR	DFAR
DFSR	DFSR
FPEXC	FPEXC
FPSCR	FPSCR
FPSID	FPSID
HACR	HACR

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-32 Base system registers (continued)

Register	Description, see
HACTLR	HACTLR
HACTLR2	HACTLR2
HADFSR	HADFSR
HAIFSR	HAIFSR
HCPTR	HCPTR
HCR	HCR
HCR2	HCR2
HDCR	HDCR
HDFAR	HDFAR
HIFAR	HIFAR
HPFAR	HPFAR
HRMR	HRMR
HSCTLR	HSCTLR
HSR	HSR
HSTR	HSTR
HTPIDR	HTPIDR
HTRFCR	HTRFCR
HVBAR	HVBAR
IFAR	IFAR
IFSR	IFSR
ISR	ISR
MVBAR	MVBAR
MVFR0	MVFR0
MVFR1	MVFR1
MVFR2	MVFR2
NSACR	NSACR
PAR	PAR
RMR	RMR
RVBAR	RVBAR
SCR	SCR
SCTLR	SCTLR
SDCR	SDCR

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-32 Base system registers (continued)

Register	Description, see
SDER	SDER
TPIDRPRW	TPIDRPRW
TPIDRURO	TPIDRURO
TPIDRURW	TPIDRURW
VBAR	VBAR
VMPIDR	VMPIDR
VPIDR	VPIDR

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K14.6 Alphabetical index of memory-mapped registers

This section is an index of memory-mapped registers in alphabetical order.

Table K14-33 Alphabetical index of Memory-Mapped Registers

Register	Description, see
AMCFGR	*AMCFGR, Activity Monitors Configuration Register* on page I5-7331
AMCGCR	*AMCGCR, Activity Monitors Counter Group Configuration Register* on page I5-7333
AMCIDR0	*AMCIDR0, Activity Monitors Component Identification Register 0* on page I5-7334
AMCIDR1	*AMCIDR1, Activity Monitors Component Identification Register 1* on page I5-7335
AMCIDR2	*AMCIDR2, Activity Monitors Component Identification Register 2* on page I5-7336
AMCIDR3	*AMCIDR3, Activity Monitors Component Identification Register 3* on page I5-7337
AMCNTENCLR0	*AMCNTENCLR0, Activity Monitors Count Enable Clear Register 0* on page I5-7338
AMCNTENCLR1	*AMCNTENCLR1, Activity Monitors Count Enable Clear Register 1* on page I5-7339
AMCNTENSET0	*AMCNTENSET0, Activity Monitors Count Enable Set Register 0* on page I5-7341
AMCNTENSET1	*AMCNTENSET1, Activity Monitors Count Enable Set Register 1* on page I5-7342
AMCR	*AMCR, Activity Monitors Control Register* on page I5-7344
AMDEVAFF0	*AMDEVAFF0, Activity Monitors Device Affinity Register 0* on page I5-7345
AMDEVAFF1	*AMDEVAFF1, Activity Monitors Device Affinity Register 1* on page I5-7346
AMDEVARCH	*AMDEVARCH, Activity Monitors Device Architecture Register* on page I5-7347
AMDEVTYPE	*AMDEVTYPE, Activity Monitors Device Type Register* on page I5-7349
AMEVCNTR0<n>	*AMEVCNTR0<n>, Activity Monitors Event Counter Registers 0, n = 0 - 15* on page I5-7350
AMEVCNTR1<n>	*AMEVCNTR1<n>, Activity Monitors Event Counter Registers 1, n = 0 - 15* on page I5-7352
AMEVTYPER0<n>	*AMEVTYPER0<n>, Activity Monitors Event Type Registers 0, n = 0 - 15* on page I5-7354
AMEVTYPER1<n>	*AMEVTYPER1<n>, Activity Monitors Event Type Registers 1, n = 0 - 15* on page I5-7356
AMIIDR	*AMIIDR, Activity Monitors Implementation Identification Register* on page I5-7358
AMPIDR0	*AMPIDR0, Activity Monitors Peripheral Identification Register 0* on page I5-7360
AMPIDR1	*AMPIDR1, Activity Monitors Peripheral Identification Register 1* on page I5-7361
AMPIDR2	*AMPIDR2, Activity Monitors Peripheral Identification Register 2* on page I5-7362
AMPIDR3	*AMPIDR3, Activity Monitors Peripheral Identification Register 3* on page I5-7363
AMPIDR4	*AMPIDR4, Activity Monitors Peripheral Identification Register 4* on page I5-7364
ASICCTL	*ASICCTL, CTI External Multiplexer Control register* on page H9-7178
CNTACR<n>	*CNTACR<n>, Counter-timer Access Control Registers, n = 0 - 7* on page I5-7367
CNTCR	*CNTCR, Counter Control Register* on page I5-7369
CNTCV	*CNTCV, Counter Count Value register* on page I5-7371
CNTEL0ACR	*CNTEL0ACR, Counter-timer EL0 Access Control Register* on page I5-7373

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-33 Alphabetical index of Memory-Mapped Registers (continued)

Register	Description, see
CNTFID0	*CNTFID0, Counter Frequency ID* on page I5-7375
CNTFID<n>	*CNTFID<n>, Counter Frequency IDs, n > 0, n = 1 - 1003* on page I5-7376
CNTFRQ	*CNTFRQ, Counter-timer Frequency* on page I5-7378
CNTID	*CNTID, Counter Identification Register* on page I5-7380
CNTNSAR	*CNTNSAR, Counter-timer Non-secure Access Register* on page I5-7381
CNTP_CTL	*CNTP_CTL, Counter-timer Physical Timer Control* on page I5-7383
CNTP_CVAL	*CNTP_CVAL, Counter-timer Physical Timer CompareValue* on page I5-7385
CNTP_TVAL	*CNTP_TVAL, Counter-timer Physical Timer TimerValue* on page I5-7387
CNTPCT	*CNTPCT, Counter-timer Physical Count* on page I5-7389
CNTSCR	*CNTSCR, Counter Scale Register* on page I5-7391
CNTSR	*CNTSR, Counter Status Register* on page I5-7392
CNTTIDR	*CNTTIDR, Counter-timer Timer ID Register* on page I5-7394
CNTV_CTL	*CNTV_CTL, Counter-timer Virtual Timer Control* on page I5-7396
CNTV_CVAL	*CNTV_CVAL, Counter-timer Virtual Timer CompareValue* on page I5-7398
CNTV_TVAL	*CNTV_TVAL, Counter-timer Virtual Timer TimerValue* on page I5-7400
CNTVCT	*CNTVCT, Counter-timer Virtual Count* on page I5-7402
CNTVOFF	*CNTVOFF, Counter-timer Virtual Offset* on page I5-7404
CNTVOFF<n>	*CNTVOFF<n>, Counter-timer Virtual Offsets, n = 0 - 7* on page I5-7406
CounterID<n>	*CounterID<n>, Counter ID registers, n = 0 - 11* on page I5-7408
CTIAPPCLEAR	*CTIAPPCLEAR, CTI Application Trigger Clear register* on page H9-7179
CTIAPPPULSE	*CTIAPPPULSE, CTI Application Pulse register* on page H9-7180
CTIAPPSET	*CTIAPPSET, CTI Application Trigger Set register* on page H9-7181
CTIAUTHSTATUS	*CTIAUTHSTATUS, CTI Authentication Status register* on page H9-7182
CTICHINSTATUS	*CTICHINSTATUS, CTI Channel In Status register* on page H9-7183
CTICHOUTSTATUS	*CTICHOUTSTATUS, CTI Channel Out Status register* on page H9-7184
CTICIDR0	*CTICIDR0, CTI Component Identification Register 0* on page H9-7185
CTICIDR1	*CTICIDR1, CTI Component Identification Register 1* on page H9-7186
CTICIDR2	*CTICIDR2, CTI Component Identification Register 2* on page H9-7187
CTICIDR3	*CTICIDR3, CTI Component Identification Register 3* on page H9-7188
CTICLAIMCLR	*CTICLAIMCLR, CTI CLAIM Tag Clear register* on page H9-7189
CTICLAIMSET	*CTICLAIMSET, CTI CLAIM Tag Set register* on page H9-7190
CTICONTROL	*CTICONTROL, CTI Control register* on page H9-7191

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-33 Alphabetical index of Memory-Mapped Registers (continued)

Register	Description, see
CTIDEVAFF0	*CTIDEVAFF0, CTI Device Affinity register 0* on page H9-7192
CTIDEVAFF1	*CTIDEVAFF1, CTI Device Affinity register 1* on page H9-7193
CTIDEVARCH	*CTIDEVARCH, CTI Device Architecture register* on page H9-7194
CTIDEVCTL	*CTIDEVCTL, CTI Device Control register* on page H9-7196
CTIDEVID	*CTIDEVID, CTI Device ID register 0* on page H9-7198
CTIDEVID1	*CTIDEVID1, CTI Device ID register 1* on page H9-7200
CTIDEVID2	*CTIDEVID2, CTI Device ID register 2* on page H9-7201
CTIDEVTYPE	*CTIDEVTYPE, CTI Device Type register* on page H9-7202
CTIGATE	*CTIGATE, CTI Channel Gate Enable register* on page H9-7203
CTIINEN<n>	*CTIINEN<n>, CTI Input Trigger to Output Channel Enable registers, n = 0 - 31* on page H9-7204
CTIINTACK	*CTIINTACK, CTI Output Trigger Acknowledge register* on page H9-7205
CTIITCTRL	*CTIITCTRL, CTI Integration mode Control register* on page H9-7207
CTILAR	*CTILAR, CTI Lock Access Register* on page H9-7209
CTILSR	*CTILSR, CTI Lock Status Register* on page H9-7211
CTIOUTEN<n>	*CTIOUTEN<n>, CTI Input Channel to Output Trigger Enable registers, n = 0 - 31* on page H9-7213
CTIPIDR0	*CTIPIDR0, CTI Peripheral Identification Register 0* on page H9-7214
CTIPIDR1	*CTIPIDR1, CTI Peripheral Identification Register 1* on page H9-7215
CTIPIDR2	*CTIPIDR2, CTI Peripheral Identification Register 2* on page H9-7216
CTIPIDR3	*CTIPIDR3, CTI Peripheral Identification Register 3* on page H9-7217
CTIPIDR4	*CTIPIDR4, CTI Peripheral Identification Register 4* on page H9-7218
CTITRIGINSTATUS	*CTITRIGINSTATUS, CTI Trigger In Status register* on page H9-7219
CTITRIGOUTSTATUS	*CTITRIGOUTSTATUS, CTI Trigger Out Status register* on page H9-7220
DBGAUTHSTATUS_EL1	*DBGAUTHSTATUS_EL1, Debug Authentication Status register* on page H9-7076
DBGBCR<n>_EL1	*DBGBCR<n>_EL1, Debug Breakpoint Control Registers, n = 0 - 15* on page H9-7078
DBGBVR<n>_EL1	*DBGBVR<n>_EL1, Debug Breakpoint Value Registers, n = 0 - 15* on page H9-7082
DBGCLAIMCLR_EL1	*DBGCLAIMCLR_EL1, Debug CLAIM Tag Clear register* on page H9-7087
DBGCLAIMSET_EL1	*DBGCLAIMSET_EL1, Debug CLAIM Tag Set register* on page H9-7089
DBGDTRRX_EL0	*DBGDTRRX_EL0, Debug Data Transfer Register, Receive* on page H9-7091
DBGDTRTX_EL0	*DBGDTRTX_EL0, Debug Data Transfer Register, Transmit* on page H9-7093
DBGWCR<n>_EL1	*DBGWCR<n>_EL1, Debug Watchpoint Control Registers, n = 0 - 15* on page H9-7095
DBGWVR<n>_EL1	*DBGWVR<n>_EL1, Debug Watchpoint Value Registers, n = 0 - 15* on page H9-7098
EDAA32PFR	*EDAA32PFR, External Debug AArch32 Processor Feature Register* on page H9-7100

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-33 Alphabetical index of Memory-Mapped Registers (continued)

Register	Description, see
EDACR	*EDACR, External Debug Auxiliary Control Register* on page H9-7102
EDCIDR0	*EDCIDR0, External Debug Component Identification Register 0* on page H9-7103
EDCIDR1	*EDCIDR1, External Debug Component Identification Register 1* on page H9-7104
EDCIDR2	*EDCIDR2, External Debug Component Identification Register 2* on page H9-7105
EDCIDR3	*EDCIDR3, External Debug Component Identification Register 3* on page H9-7106
EDCIDSR	*EDCIDSR, External Debug Context ID Sample Register* on page H9-7107
EDDEVAFF0	*EDDEVAFF0, External Debug Device Affinity register 0* on page H9-7109
EDDEVAFF1	*EDDEVAFF1, External Debug Device Affinity register 1* on page H9-7110
EDDEVARCH	*EDDEVARCH, External Debug Device Architecture register* on page H9-7111
EDDEVID	*EDDEVID, External Debug Device ID register 0* on page H9-7113
EDDEVID1	*EDDEVID1, External Debug Device ID register 1* on page H9-7115
EDDEVID2	*EDDEVID2, External Debug Device ID register 2* on page H9-7116
EDDEVTYPE	*EDDEVTYPE, External Debug Device Type register* on page H9-7117
EDDFR	*EDDFR, External Debug Feature Register* on page H9-7118
EDECCR	*EDECCR, External Debug Exception Catch Control Register* on page H9-7121
EDECR	*EDECR, External Debug Execution Control Register* on page H9-7125
EDESR	*EDESR, External Debug Event Status Register* on page H9-7127
EDITCTRL	*EDITCTRL, External Debug Integration mode Control register* on page H9-7129
EDITR	*EDITR, External Debug Instruction Transfer Register* on page H9-7130
EDLAR	*EDLAR, External Debug Lock Access Register* on page H9-7132
EDLSR	*EDLSR, External Debug Lock Status Register* on page H9-7134
EDPCSR	*EDPCSR, External Debug Program Counter Sample Register* on page H9-7136
EDPFR	*EDPFR, External Debug Processor Feature Register* on page H9-7139
EDPIDR0	*EDPIDR0, External Debug Peripheral Identification Register 0* on page H9-7143
EDPIDR1	*EDPIDR1, External Debug Peripheral Identification Register 1* on page H9-7144
EDPIDR2	*EDPIDR2, External Debug Peripheral Identification Register 2* on page H9-7145
EDPIDR3	*EDPIDR3, External Debug Peripheral Identification Register 3* on page H9-7146
EDPIDR4	*EDPIDR4, External Debug Peripheral Identification Register 4* on page H9-7147
EDPRCR	*EDPRCR, External Debug Power/Reset Control Register* on page H9-7148
EDPRSR	*EDPRSR, External Debug Processor Status Register* on page H9-7152
EDRCR	*EDRCR, External Debug Reserve Control Register* on page H9-7160
EDSCR	*EDSCR, External Debug Status and Control Register* on page H9-7162

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-33 Alphabetical index of Memory-Mapped Registers (continued)

Register	Description, see
EDVIDSR	*EDVIDSR, External Debug Virtual Context Sample Register* on page H9-7168
EDWAR	*EDWAR, External Debug Watchpoint Address Register* on page H9-7171
ERRCIDR0	*ERRCIDR0, Component Identification Register 0* on page I5-7411
ERRCIDR1	*ERRCIDR1, Component Identification Register 1* on page I5-7412
ERRCIDR2	*ERRCIDR2, Component Identification Register 2* on page I5-7413
ERRCIDR3	*ERRCIDR3, Component Identification Register 3* on page I5-7414
ERRCRICR0	*ERRCRICR0, Critical Error Interrupt Configuration Register 0* on page I5-7415
ERRCRICR1	*ERRCRICR1, Critical Error Interrupt Configuration Register 1* on page I5-7416
ERRCRICR2	*ERRCRICR2, Critical Error Interrupt Configuration Register 2* on page I5-7417
ERRDEVAFF	*ERRDEVAFF, Device Affinity Register* on page I5-7420
ERRDEVARCH	*ERRDEVARCH, Device Architecture Register* on page I5-7424
ERRDEVID	*ERRDEVID, Device Configuration Register* on page I5-7426
ERRERICR0	*ERRERICR0, Error Recovery Interrupt Configuration Register 0* on page I5-7427
ERRERICR1	*ERRERICR1, Error Recovery Interrupt Configuration Register 1* on page I5-7428
ERRERICR2	*ERRERICR2, Error Recovery Interrupt Configuration Register 2* on page I5-7429
ERRFHICR0	*ERRFHICR0, Fault-Handling Interrupt Configuration Register 0* on page I5-7432
ERRFHICR1	*ERRFHICR1, Fault-Handling Interrupt Configuration Register 1* on page I5-7433
ERRFHICR2	*ERRFHICR2, Fault-Handling Interrupt Configuration Register 2* on page I5-7434
ERRGSR	*ERRGSR, Error Group Status Register* on page I5-7437
ERRIIDR	*ERRIIDR, Implementation Identification Register* on page I5-7438
ERRIRQCR<n>	*ERRIRQCR<n>, Generic Error Interrupt Configuration Register, n = 0 - 15* on page I5-7440
ERRIRQSR	*ERRIRQSR, Error Interrupt Status Register* on page I5-7441
ERR<n>ADDR	*ERR<n>ADDR, Error Record Address Register, n = 0 - 65534* on page I5-7444
ERR<n>CTLR	*ERR<n>CTLR, Error Record Control Register, n = 0 - 65534* on page I5-7446
ERR<n>FR	*ERR<n>FR, Error Record Feature Register, n = 0 - 65534* on page I5-7453
ERR<n>MISC0	*ERR<n>MISC0, Error Record Miscellaneous Register 0, n = 0 - 65534* on page I5-7459
ERR<n>MISC1	*ERR<n>MISC1, Error Record Miscellaneous Register 1, n = 0 - 65534* on page I5-7465
ERR<n>MISC2	*ERR<n>MISC2, Error Record Miscellaneous Register 2, n = 0 - 65534* on page I5-7467
ERR<n>MISC3	*ERR<n>MISC3, Error Record Miscellaneous Register 3, n = 0 - 65534* on page I5-7469
ERR<n>PFGCDN	*ERR<n>PFGCDN, Pseudo-fault Generation Countdown Register, n = 0 - 65534* on page I5-7471
ERR<n>PFGCTL	*ERR<n>PFGCTL, Pseudo-fault Generation Control Register, n = 0 - 65534* on page I5-7473
ERR<n>PFGF	*ERR<n>PFGF, Pseudo-fault Generation Feature Register, n = 0 - 65534* on page I5-7478

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-33 Alphabetical index of Memory-Mapped Registers (continued)

Register	Description, see
ERR<n>STATUS	*ERR<n>STATUS, Error Record Primary Status Register, n = 0 - 65534* on page I5-7482
ERRPIDR0	*ERRPIDR0, Peripheral Identification Register 0* on page I5-7496
ERRPIDR1	*ERRPIDR1, Peripheral Identification Register 1* on page I5-7497
ERRPIDR2	*ERRPIDR2, Peripheral Identification Register 2* on page I5-7499
ERRPIDR3	*ERRPIDR3, Peripheral Identification Register 3* on page I5-7501
ERRPIDR4	*ERRPIDR4, Peripheral Identification Register 4* on page I5-7503
MIDR_EL1	*MIDR_EL1, Main ID Register* on page H9-7173
OSLAR_EL1	*OSLAR_EL1, OS Lock Access Register* on page H9-7175
PMAUTHSTATUS	*PMAUTHSTATUS, Performance Monitors Authentication Status register* on page I5-7256
PMCCFILTR_EL0	*PMCCFILTR_EL0, Performance Monitors Cycle Counter Filter Register* on page I5-7258
PMCCNTR_EL0	*PMCCNTR_EL0, Performance Monitors Cycle Counter* on page I5-7261
PMCEID0	*PMCEID0, Performance Monitors Common Event Identification register 0* on page I5-7263
PMCEID1	*PMCEID1, Performance Monitors Common Event Identification register 1* on page I5-7265
PMCEID2	*PMCEID2, Performance Monitors Common Event Identification register 2* on page I5-7267
PMCEID3	*PMCEID3, Performance Monitors Common Event Identification register 3* on page I5-7269
PMCFGR	*PMCFGR, Performance Monitors Configuration Register* on page I5-7271
PMCID1SR	*PMCID1SR, CONTEXTIDR_EL1 Sample Register* on page I5-7277
PMCID2SR	*PMCID2SR, CONTEXTIDR_EL2 Sample Register* on page I5-7279
PMCIDR0	*PMCIDR0, Performance Monitors Component Identification Register 0* on page I5-7273
PMCIDR1	*PMCIDR1, Performance Monitors Component Identification Register 1* on page I5-7274
PMCIDR2	*PMCIDR2, Performance Monitors Component Identification Register 2* on page I5-7275
PMCIDR3	*PMCIDR3, Performance Monitors Component Identification Register 3* on page I5-7276
PMCNTENCLR_EL0	*PMCNTENCLR_EL0, Performance Monitors Count Enable Clear register* on page I5-7281
PMCNTENSET_EL0	*PMCNTENSET_EL0, Performance Monitors Count Enable Set register* on page I5-7283
PMCR_EL0	*PMCR_EL0, Performance Monitors Control Register* on page I5-7285
PMDEVAFF0	*PMDEVAFF0, Performance Monitors Device Affinity register 0* on page I5-7289
PMDEVAFF1	*PMDEVAFF1, Performance Monitors Device Affinity register 1* on page I5-7290
PMDEVARCH	*PMDEVARCH, Performance Monitors Device Architecture register* on page I5-7291
PMDEVID	*PMDEVID, Performance Monitors Device ID register* on page I5-7293
PMDEVTYPE	*PMDEVTYPE, Performance Monitors Device Type register* on page I5-7295
PMEVCNTR<n>_EL0	*PMEVCNTR<n>_EL0, Performance Monitors Event Count Registers, n = 0 - 30* on page I5-7296
PMEVTYPER<n>_EL0	*PMEVTYPER<n>_EL0, Performance Monitors Event Type Registers, n = 0 - 30* on page I5-7298

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-33 Alphabetical index of Memory-Mapped Registers (continued)

Register	Description, see
PMINTENCLR_EL1	*PMINTENCLR_EL1, Performance Monitors Interrupt Enable Clear register* on page I5-7302
PMINTENSET_EL1	*PMINTENSET_EL1, Performance Monitors Interrupt Enable Set register* on page I5-7304
PMITCTRL	*PMITCTRL, Performance Monitors Integration mode Control register* on page I5-7306
PMLAR	*PMLAR, Performance Monitors Lock Access Register* on page I5-7307
PMLSR	*PMLSR, Performance Monitors Lock Status Register* on page I5-7309
PMMIR	*PMMIR, Performance Monitors Machine Identification Register* on page I5-7311
PMOVSCLR_EL0	*PMOVSCLR_EL0, Performance Monitors Overflow Flag Status Clear register* on page I5-7312
PMOVSSET_EL0	*PMOVSSET_EL0, Performance Monitors Overflow Flag Status Set register* on page I5-7314
PMPCSR	*PMPCSR, Program Counter Sample Register* on page I5-7316
PMPIDR0	*PMPIDR0, Performance Monitors Peripheral Identification Register 0* on page I5-7319
PMPIDR1	*PMPIDR1, Performance Monitors Peripheral Identification Register 1* on page I5-7320
PMPIDR2	*PMPIDR2, Performance Monitors Peripheral Identification Register 2* on page I5-7321
PMPIDR3	*PMPIDR3, Performance Monitors Peripheral Identification Register 3* on page I5-7322
PMPIDR4	*PMPIDR4, Performance Monitors Peripheral Identification Register 4* on page I5-7323
PMSWINC_EL0	*PMSWINC_EL0, Performance Monitors Software Increment register* on page I5-7324
PMVIDSR	*PMVIDSR, VMID Sample Register* on page I5-7326

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K14.7 Functional index of memory-mapped registers

This section is an index of the memory-mapped registers, divided by functional group. Each of the following sections lists the registers for a functional group:

- *ID registers*.
- *Performance monitors registers*.
- *Debug registers* on page K14-8105.
- *RAS registers* on page K14-8106.
- *Cross-trigger interface registers* on page K14-8107.

K14.7.1 ID registers

This section is an index to the registers in the Identification registers functional group.

Table K14-34 ID registers

Register	Description, see
EDAA32PFR	EDAA32PFR
EDDFR	EDDFR
EDPFR	EDPFR
MIDR_EL1	MIDR_EL1

K14.7.2 Performance monitors registers

This section is an index to the registers in the Performance Monitors registers functional group.

Table K14-35 Performance monitors registers

Register	Description, see
PMAUTHSTATUS	PMAUTHSTATUS
PMCCFILTR_EL0	PMCCFILTR_EL0
PMCCNTR_EL0	PMCCNTR_EL0
PMCEID0	PMCEID0
PMCEID1	PMCEID1
PMCEID2	PMCEID2
PMCEID3	PMCEID3
PMCFGR	PMCFGR
PMCID1SR	PMCID1SR
PMCID2SR	PMCID2SR
PMCIDR0	PMCIDR0
PMCIDR1	PMCIDR1
PMCIDR2	PMCIDR2
PMCIDR3	PMCIDR3

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-35 Performance monitors registers (continued)

Register	Description, see
PMCNTENCLR_EL0	PMCNTENCLR_EL0
PMCNTENSET_EL0	PMCNTENSET_EL0
PMCR_EL0	PMCR_EL0
PMDEVAFF0	PMDEVAFF0
PMDEVAFF1	PMDEVAFF1
PMDEVARCH	PMDEVARCH
PMDEVID	PMDEVID
PMDEVTYPE	PMDEVTYPE
PMEVCNTR<n>_EL0	PMEVCNTR<n>_EL0
PMEVTYPER<n>_EL0	PMEVTYPER<n>_EL0
PMINTENCLR_EL1	PMINTENCLR_EL1
PMINTENSET_EL1	PMINTENSET_EL1
PMITCTRL	PMITCTRL
PMLAR	PMLAR
PMLSR	PMLSR
PMMIR	PMMIR
PMOVSCLR_EL0	PMOVSCLR_EL0
PMOVSSET_EL0	PMOVSSET_EL0
PMPCSR	PMPCSR
PMPIDR0	PMPIDR0
PMPIDR1	PMPIDR1
PMPIDR2	PMPIDR2
PMPIDR3	PMPIDR3
PMPIDR4	PMPIDR4
PMSWINC_EL0	PMSWINC_EL0
PMVIDSR	PMVIDSR

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

K14.7.3 Debug registers

This section is an index to the registers in the Debug registers functional group.

Table K14-36 Debug registers

Register	Description, see
DBGAUTHSTATUS_EL1	DBGAUTHSTATUS_EL1
DBGBCR<n>_EL1	DBGBCR<n>_EL1
DBGBVR<n>_EL1	DBGBVR<n>_EL1
DBGCLAIMCLR_EL1	DBGCLAIMCLR_EL1
DBGCLAIMSET_EL1	DBGCLAIMSET_EL1
DBGDTRRX_EL0	DBGDTRRX_EL0
DBGDTRTX_EL0	DBGDTRTX_EL0
DBGWCR<n>_EL1	DBGWCR<n>_EL1
DBGWVR<n>_EL1	DBGWVR<n>_EL1
EDACR	EDACR
EDCIDR0	EDCIDR0
EDCIDR1	EDCIDR1
EDCIDR2	EDCIDR2
EDCIDR3	EDCIDR3
EDCIDSR	EDCIDSR
EDDEVAFF0	EDDEVAFF0
EDDEVAFF1	EDDEVAFF1
EDDEVARCH	EDDEVARCH
EDDEVID	EDDEVID
EDDEVID1	EDDEVID1
EDDEVID2	EDDEVID2
EDDEVTYPE	EDDEVTYPE
EDECCR	EDECCR
EDECR	EDECR
EDESR	EDESR
EDITCTRL	EDITCTRL
EDITR	EDITR
EDLAR	EDLAR
EDLSR	EDLSR
EDPCSR	EDPCSR

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-36 Debug registers (continued)

Register	Description, see
EDPIDR0	EDPIDR0
EDPIDR1	EDPIDR1
EDPIDR2	EDPIDR2
EDPIDR3	EDPIDR3
EDPIDR4	EDPIDR4
EDPRCR	EDPRCR
EDPRSR	EDPRSR
EDRCR	EDRCR
EDSCR	EDSCR
EDVIDSR	EDVIDSR
EDWAR	EDWAR
OSLAR_EL1	OSLAR_EL1

K14.7.4 RAS registers

This section is an index to the registers in the RAS registers functional group.

Table K14-37 RAS registers

Register	Description, see
ERRCIDR0	ERRCIDR0
ERRCIDR1	ERRCIDR1
ERRCIDR2	ERRCIDR2
ERRCIDR3	ERRCIDR3
ERRCRICR0	ERRCRICR0
ERRCRICR1	ERRCRICR1
ERRCRICR2	ERRCRICR2
ERRDEVAFF	ERRDEVAFF
ERRDEVARCH	ERRDEVARCH
ERRDEVID	ERRDEVID
ERRERICR0	ERRERICR0
ERRERICR1	ERRERICR1
ERRERICR2	ERRERICR2
ERRFHICR0	ERRFHICR0
ERRFHICR1	ERRFHICR1

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-37 RAS registers (continued)

Register	Description, see
ERRFHICR2	ERRFHICR2
ERRGSR	ERRGSR
ERRIIDR	ERRIIDR
ERRIRQCR<n>	ERRIRQCR<n>
ERRIRQSR	ERRIRQSR
ERR<n>ADDR	ERR<n>ADDR
ERR<n>CTLR	ERR<n>CTLR
ERR<n>FR	ERR<n>FR
ERR<n>MISC0	ERR<n>MISC0
ERR<n>MISC1	ERR<n>MISC1
ERR<n>MISC2	ERR<n>MISC2
ERR<n>MISC3	ERR<n>MISC3
ERR<n>PFGCDN	ERR<n>PFGCDN
ERR<n>PFGCTL	ERR<n>PFGCTL
ERR<n>PFGF	ERR<n>PFGF
ERR<n>STATUS	ERR<n>STATUS
ERRPIDR0	ERRPIDR0
ERRPIDR1	ERRPIDR1
ERRPIDR2	ERRPIDR2
ERRPIDR3	ERRPIDR3
ERRPIDR4	ERRPIDR4

K14.7.5 Cross-trigger interface registers

This section is an index to the registers in the Cross-Trigger Interface registers functional group.

Table K14-38 Cross-trigger interface registers

Register	Description, see
ASICCTL	ASICCTL
CTIAPPCLEAR	CTIAPPCLEAR
CTIAPPPULSE	CTIAPPPULSE
CTIAPPSET	CTIAPPSET
CTIAUTHSTATUS	CTIAUTHSTATUS
CTICHINSTATUS	CTICHINSTATUS

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K14-38 Cross-trigger interface registers (continued)

Register	Description, see
CTICHOUTSTATUS	CTICHOUTSTATUS
CTICIDR0	CTICIDR0
CTICIDR1	CTICIDR1
CTICIDR2	CTICIDR2
CTICIDR3	CTICIDR3
CTICLAIMCLR	CTICLAIMCLR
CTICLAIMSET	CTICLAIMSET
CTICONTROL	CTICONTROL
CTIDEVAFF0	CTIDEVAFF0
CTIDEVAFF1	CTIDEVAFF1
CTIDEVARCH	CTIDEVARCH
CTIDEVCTL	CTIDEVCTL
CTIDEVID	CTIDEVID
CTIDEVID1	CTIDEVID1
CTIDEVID2	CTIDEVID2
CTIDEVTYPE	CTIDEVTYPE
CTIGATE	CTIGATE
CTIINEN<n>	CTIINEN<n>
CTIINTACK	CTIINTACK
CTIITCTRL	CTIITCTRL
CTILAR	CTILAR
CTILSR	CTILSR
CTIOUTEN<n>	CTIOUTEN<n>
CTIPIDR0	CTIPIDR0
CTIPIDR1	CTIPIDR1
CTIPIDR2	CTIPIDR2
CTIPIDR3	CTIPIDR3
CTIPIDR4	CTIPIDR4
CTITRIGINSTATUS	CTITRIGINSTATUS
CTITRIGOUTSTATUS	CTITRIGOUTSTATUS

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Glossary

A32 instruction

A word that specifies an operation to be performed by a PE that is executing in an Exception level that is using AArch32 and is in A32 state. A32 instructions must be word-aligned.

A32 instructions were previously called ARM instructions.

See also A32 state, A64 instruction, T32 instruction.

A32 state

The AArch32 Instruction set state in which the PE executes A32 instructions.

A32 state was previously called ARM state.

See also T32 instruction, T32 state.

A64 instruction

A word that specifies an operation to be performed by a PE that is executing in an Exception level that is using AArch64. A64 instructions must be word-aligned.

See also A32 instruction, T32 instruction.

AArch32

The 32-bit Execution state. In AArch32 state, addresses are held in 32-bit registers, and instructions in the base instruction sets use 32-bit registers for their processing. AArch32 state supports the T32 and A32 instruction sets

See also AArch64, A32 instruction, T32 instruction.

AArch64

The 64-bit Execution state. In AArch64 state, addresses are held in 64-bit registers, and instructions in the base instruction set can use 64-bit registers for their processing. AArch64 state supports the A64 instruction set.

See also AArch32, A64 instruction.

Abort

An exception caused by an illegal memory access. Aborts can be caused by the external memory system or the MMU.

Addressing mode

Means a method for generating the memory address used by a load/store instruction.

Advanced SIMD

A feature of the Arm architecture that provides SIMD operations on a register file of SIMD and floating-point registers. Where an implementation supports both Advanced SIMD and floating-point instructions, these instructions operate on the same register file.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Aligned

A data item stored at an address that is exactly divisible by the highest power of 2 that divides exactly into its size in bytes. Aligned halfwords, words and doublewords therefore have addresses that are divisible by 2, 4 and 8 respectively.

An aligned access is one where the address of the access is aligned to the size of each element of the access.

Architecturally executed

An instruction is architecturally executed only if it would be executed in a simple sequential execution of the program. When such an instruction has been executed and retired it has been *architecturally executed*. Any instruction that, in a simple sequential execution of a program, is treated as a `NOP` because it fails its condition code check, is an architecturally executed instruction.

In a PE that performs speculative execution, an instruction is not architecturally executed if the PE discards the results of a speculative execution.

See also Condition code check, Simple sequential execution.

Architecturally mapped

Where this manual describes a register as being *architecturally mapped* to another register, this indicates that, in an implementation that supports both of the registers, the two registers access the same state.

Architecturally UNKNOWN

An architecturally UNKNOWN value is a value that is not defined by the architecture but must meet the requirements of the definition of UNKNOWN. Implementations can define the value of the field, but are not required to do so.

See also IMPLEMENTATION DEFINED.

ARM core registers

Some older documentation uses *ARM core registers* to refer to the following set of registers for execution in AArch32 state:

- The 13 general-purpose registers, R0-R12, that software can use for processing.
- SP, the *stack pointer*, that can also be referred to as R13.
- LR, the *link register*, that can also be referred to as R14.
- PC, the *program counter*, that can also be referred to as R15.

See also General-purpose registers.

ARM instruction

See A32 instruction.

Associativity

See Cache associativity.

Atomicity

Describes either single-copy atomicity or multi-copy atomicity. *Atomicity in the Arm architecture* on page B2-118 defines these forms of atomicity for the Arm architecture.

See also Multi-copy atomicity, Single-copy atomicity.

Banked register

A register that has multiple instances, with the instance that is in use depending on the PE mode, Security state, or other PE state.

Base register

A register specified by a load/store instruction that is used as the base value for the address calculation for the instruction. Depending on the instruction and its addressing mode, an offset can be added to or subtracted from the base register value to form the virtual address that is sent to memory.

Base register writeback

Describes writing back a modified value to the base register used in an address calculation.

Behaves as if

Where this manual indicates that a PE *behaves as if* a certain condition applies, all descriptions of the operation of the PE must be re-evaluated taking account of that condition, together with any other conditions that affect operation.

Big-endian memory

Means that, for example:

- A byte or halfword at a word-aligned address is the most significant byte or halfword in the word at that address.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- A byte at a halfword-aligned address is the most significant byte in the halfword at that address.

See also Endianness, Little-endian memory.

Blocking

Describes an operation that does not permit following instructions to be executed before the operation completes.

A non-blocking operation can permit following instructions to be executed before the operation completes, and in the event of encountering an exception does not signal an exception to the PE. This enables implementations to retire following instructions while the non-blocking operation is executing, without the need to retain precise PE state.

Branch prediction

Is where a PE selects a future execution path to fetch along. For example, after a branch instruction, the PE can choose to speculatively fetch either the instruction following the branch or the instruction at the branch target.

See also Prefetching.

Breakpoint

A debug event triggered by the execution of a particular instruction, specified by one or both of the address of the instruction and the state of the PE when the instruction is executed.

Byte

An 8-bit data item.

Cache associativity

The number of locations in a cache set to which an address can be assigned. Each location is identified by its *way* value.

Cache level

The position of a cache in the cache hierarchy. In the Arm architecture, the lower numbered levels are those closest to the PE. For more information, see *Terms used in describing the cache maintenance instructions* on page D4-2506.

Cache line

The basic unit of storage in a cache. Its size in words is always a power of two, usually 4 or 8 words. A cache line must be aligned to a suitable memory boundary. A *memory cache line* is a block of memory locations with the same size and alignment as a cache line. Memory cache lines are sometimes loosely called cache lines.

Cache lockdown

Enables critical software and data to be loaded into the cache so that the cache lines containing them are not subsequently reallocated. It alleviates the delays caused by accessing a cache in a worst-case situation. This ensures that all subsequent accesses to the software and data concerned are cache hits and so complete quickly.

Cache miss

A memory access that cannot be processed at high speed because the data it addresses is not in the cache.

Cache sets

Areas of a cache, divided up to simplify and speed up the process of determining whether a cache hit occurs. The number of cache sets is always a power of two.

Cache way

A cache way consists of one cache line from each cache set. The cache ways are indexed from 0 to (Associativity-1). Each cache line in a cache way is chosen to have the same index as the cache way. For example, cache way *n* consists of the cache line with index *n* from each cache set.

Coherence order

See Coherent.

Coherent

Data accesses from a set of observers to a byte in memory are coherent if accesses to that byte in memory by the members of that set of observers are consistent with there being a single total order of all writes to that byte in memory by all members of the set of observers. This single total order of all to writes to that memory location is the *coherence order* for that byte in memory.

Condition code check

The process of determining whether a conditional instruction executes normally or is treated as a `NOP`. For an instruction that includes a condition code field, that field is compared with the condition flags to determine whether the instruction is executed normally. For a T32 instruction in an IT block, the value of PSTATE.IT determines whether the instruction is executed normally.

See also Condition code field, Condition flags, Conditional execution.

Condition code field

A 4-bit field in an instruction that specifies the condition under which the instruction executes.

See also Condition code check.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Condition flags The N, Z, C, and V bits of PSTATE, an SPSR, or FPSCR. See the register descriptions for more information.

See also Condition code check, PSTATE.

Conditional execution

When a conditional instruction starts executing, if the condition code check returns TRUE, the instruction executes normally. Otherwise, it is treated as a `NOP`.

See also Condition code check.

CONSTRAINED UNPREDICTABLE

Where an instruction can result in UNPREDICTABLE behavior, the Armv8 architecture specifies a narrow range of permitted behaviors. This range is the range of CONSTRAINED UNPREDICTABLE behavior. All implementations that are compliant with the architecture must follow the CONSTRAINED UNPREDICTABLE behavior.

Execution at Non-secure EL1 or EL0 of an instruction that is CONSTRAINED UNPREDICTABLE can be implemented as generating a trap exception that is taken to EL2, provided that at least one instruction that is not UNPREDICTABLE and is not CONSTRAINED UNPREDICTABLE causes a trap exception that is taken to EL2.

In body text, the term CONSTRAINED UNPREDICTABLE is shown in SMALL CAPITALS.

See also UNPREDICTABLE.

Context switch The saving and restoring of computational state when switching between different threads or processes. In this manual, the term context switch describes any situation where the context is switched by an operating system and might or might not include changes to the address space.

Context synchronization event

One of:

- Performing an ISB operation. An ISB operation is performed when an `ISB` instruction is executed and does not fail its condition code check.
- Exception entry, if ARMv8.5-CSEH is not implemented or the exception is taken to AArch32 or if ARMv8.5-CSEH is implemented and the appropriate SCTLR_ELx.EIS bit is set.
- Return from an exception, if ARMv8.5-CSEH is not implemented, or the exception is returning from AArch32 or if ARMv8.5-CSEH is implemented and the appropriate SCTLR_ELx.EOS bit is set.
- Exit from Debug state.
- Executing a `DCPS` instruction.
- Executing a `DRPS` instruction.

The effects of a Context synchronization event are:

- All unmasked interrupts that are pending at the time of the Context synchronization event are taken before the first instruction after the Context synchronization event.
- If halting is allowed, all Halting debug events that are pending at the time of the Context synchronization event are taken before the first instruction after the Context synchronization event.
- No instructions appearing in program order after an instruction that causes a Context synchronization event will have performed any part of their functionality until the Context synchronization event has occurred.
- All direct and indirect writes to System registers that are made before the Context synchronization event affect any instruction, including a direct read, that appears in program order after the instruction causing the Context synchronization event.
- All completed changes to the translation tables for entries that, before the change, were not permitted to be cached in a TLB, affect all instruction fetches that appear in program order after the instruction causing the Context synchronization event.
- All invalidations of TLBs, instruction caches, and, in AArch32 state, branch predictors, that are completed before the Context synchronization event affect all instructions that appear in program order after an instruction causing a Context synchronization event.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- In AArch32 state, all Non-cacheable writes that are completed before the Context synchronization event affect all instructions that appear in program order after an instruction causing a Context synchronization event.
- Changes to the Debug external authentication interfaces that are made before the Context synchronization event affect any instruction that appears in program order after the instruction causing the Context synchronization event.

Note

- The architecture requires that instructions that generate Context synchronization events do not appear to be executed speculatively, except that the performance monitor counters are permitted to reveal such speculation.
- *Context synchronization events* were previously described as *context synchronization operations*.

Data independent timing (DIT)
The time that it takes to execute a piece of code where the time is not a function of the data being operated on. For more information, see *About PSTATE.DIT* on page B1-113 and *About the DIT bit* on page E1-3959.

Debugger
In most of this manual, *debugger* refers to any agent that is performing debug. However, some chapters or parts of this manual require a more rigorous definition, and define debugger locally. See:

- *Definition of a debugger in the context of self-hosted debug* on page D2-2424.
- *Definition of a debugger in the context of self-hosted debug* on page G2-5814.
- *Definition and constraints of a debugger in the context of external debug* on page H1-6922.

Deprecated
Something that is present in the Arm architecture for backwards compatibility. Whenever possible software must avoid using deprecated features. Features that are deprecated but are not optional are present in current implementations of the Arm architecture, but might not be present, or might be deprecated and OPTIONAL, in future versions of the Arm architecture.

See also OPTIONAL.

Digital signal processing (DSP)
Algorithms for processing signals that have been sampled and converted to digital form. DSP algorithms often use saturated arithmetic.

Direct Memory Access (DMA)
An operation that accesses main memory directly, without the PE performing any accesses to the data concerned.

Direct read
A direct read of a System register is a read performed by a System register access instruction.

For more information, see *Direct read* on page D13-2871.

See also Direct write, Indirect read, Indirect write.

Direct write
A direct write of a System register is a write performed by a System register access instruction.

For more information, see *Direct write* on page D13-2871.

See also Direct read, Indirect read, Indirect write.

DMA
See Direct Memory Access (DMA).

DNM
See Do-Not-Modify (DNM).

Domain
In the Arm architecture, *domain* is used in the following contexts.

Shareability domain Defines a set of observers for which the shareability attributes make the data or unified caches transparent for data accesses.

Power domain Defines a block of logic with a single, common, power supply.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Memory regions domain

When using the Short-descriptor translation table format, defines a collection of Sections, Large pages and Small pages of memory, that can have their access permissions switched rapidly by writing to the *Domain Access Control Register* (DACR). Arm deprecates any use of memory regions domains.

Do-Not-Modify (DNM)

Means the value must not be altered by software. DNM fields read as UNKNOWN values, and must only be written with the value read from the same field on the same PE.

Double-precision value

Consists of two consecutive 32-bit words that are interpreted as a basic double-precision floating-point number according to the *IEEE Standard for Floating-point Arithmetic*.

Doubleword

A 64-bit data item. Doublewords are normally at least word-aligned in Arm systems.

Doubleword-aligned

Means that the address is divisible by 8.

DSP

See Digital signal processing (DSP).

Effective value

A register control field, meaning a field in a register that controls some aspect of the behavior, can be described as having an *Effective value*:

- In some cases, the description of a control *a* specifies that when control *a* is active it causes a register control field *b* to be treated as having a fixed value for all purposes other than direct reads, or direct reads and direct writes, of the register containing control field *b*. When control *a* is active that fixed value is described as the *Effective value* of register control field *b*. For example, when the value of HCR.DC is 1, the *Effective value* of HCR.VM is 1, regardless of its actual value.

  In other cases, in some contexts a register control field *b* is not implemented or is not accessible, but behavior of the PE is as if control field *b* was implemented and accessible, and had a particular value. In this case, that value is the *Effective value* of register control field *b*.

  Note

  Where a register control field is introduced in a particular version of the architecture, and is not implemented in an earlier version of the architecture, typically it will have an *Effective value* in that earlier version of the architecture.

- Otherwise, the *Effective value* of a register control field is the value of that field.

Endianness

An aspect of the system memory mapping.

See also Big-endian memory and Little-endian memory.

Exception

Handles an event. For example, an exception could handle an external interrupt or an undefined instruction.

Exception vector

A fixed address that contains the address of the first instruction of the corresponding exception handler.

Execution stream

The stream of instructions that would have been executed by sequential execution of the program.

Explicit access

A read from memory, or a write to memory, generated by a load or store instruction executed by the PE. Reads and writes generated by hardware translation table accesses are not explicit accesses.

External abort

An abort that is generated by the external memory system.

Fast Context Switch Extension (FCSE)

Modifies the behavior of an Arm memory system to enable multiple programs running on the Arm PE to use identical address ranges, while ensuring that the addresses they present to the rest of the memory system differ. From Armv6, Arm deprecates any use of the FCSE. The FCSE is:

- Optional in an Armv7 implementation that does not include the Multiprocessing Extensions.
- Obsolete from the introduction of the Multiprocessing Extensions.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

FCSE
See Fast Context Switch Extension (FCSE).

Flat address mapping
Is where the physical address for every access is equal to its virtual address.

Flush-to-zero mode
A processing mode that optimizes the performance of some floating-point algorithms by replacing the denormalized operands and intermediate results with zeros, without significantly affecting the accuracy of their final results.

General-purpose registers
The registers that the base instructions use for processing:

- In AArch32 state the general-purpose registers are R0-R14, that can also be described as R0-R12, SP, LR.

  Note

  Older documentation defines the AArch32 general-purpose registers as R0-R12, and the Arm core registers as R0-R12, SP, LR, and PC.

- In AArch64 state the general-purpose registers are:
  - W0-W30 when accessed as 32-bit registers.
  - X0-X30 when accessed as 64-bit registers.

See also High registers, Low registers.

Generated by
The memory model is written in terms of reads from memory and writes to memory. These reads and writes:

- Are generated by instructions such as loads, stores, and atomic memory accesses.
- Correspond to the memory accesses, other than translation table walks, that are defined in the instruction pseudocode.

Some instructions generate more than one read or write.

Halfword
A 16-bit data item. Halfwords are normally halfword-aligned in Arm systems.

Halfword-aligned
Means that the address is divisible by 2.

High registers
In AArch32 state, the general-purpose registers R8-R14. Most 16-bit T32 instructions cannot access the high registers.

Note

In some contexts, *high registers* refers to R8-R15, meaning R8-R14 and the PC.

See also General-purpose registers, Low registers.

High vectors
An alternative location for the exception vectors. The high vector address range is near the top of the address space, rather than at the bottom.

IGNORED
Indicates that the architecture guarantees that the bit or field is not interpreted or modified by hardware.

In body text, the term IGNORED is shown in SMALL CAPITALS.

Immediate and offset fields
Are unsigned unless otherwise stated.

Immediate value
A value that is encoded directly in the instruction and used as numeric data when the instruction is executed. Many A64, A32, and T32 instructions can be used with an immediate argument.

IMP
An abbreviation used in diagrams to indicate that one or more bits have IMPLEMENTATION DEFINED behavior.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

IMPLEMENTATION DEFINED

Means that the behavior is not architecturally defined, but must be defined and documented by individual implementations.

In body text, the term IMPLEMENTATION DEFINED is shown in SMALL CAPITALS.

Index register

A register specified in some load and store instructions. The value of this register is used as an offset to be added to or subtracted from the base register value to form the virtual address that is sent to memory. Some instruction forms permit the index register value to be shifted before the addition or subtraction.

Indirect read

When an instruction uses a System register value to establish operating conditions, that use of the System register is an indirect read of the System register.

For more information, including additional examples of indirect reads, see *Indirect read* on page D13-2871.

See also Direct read, Direct write, Indirect write.

Indirect write

An indirect write of a System register occurs when the contents of a register are updated by some mechanism other than a *Direct write* to that register. For example, an indirect write to a register might occur as a side-effect of executing an instruction that does not perform a direct write to the register, or because of some operation performed by an external agent.

For more information, see *Indirect write* on page D13-2871.

See also Direct read, Direct write, Indirect read.

Inline literals

These are constant addresses and other data items held in the same area as the software itself. They are automatically generated by compilers, and can also appear in assembler code.

Intermediate physical address (IPA)

An implementation of virtualization, the address to which a Guest OS maps a VA. A hypervisor might then map the IPA to a PA. Typically, the Guest OS is unaware of the translation from IPA to PA.

See also Physical address (PA), Virtual address (VA).

Interworking

A method of working that permits branches between software using the A32 and T32 instruction sets.

IPA

See Intermediate physical address (IPA).

Level

See Cache level.

Level of Coherence (LoC)

The last level of cache that must be cleaned or invalidated when cleaning or invalidating to the point of coherency. For more information, see *Terms used in describing the cache maintenance instructions* on page D4-2506.

See also Cache level, Point of coherency (PoC).

Level of Unification, Inner Shareable (LoUIS)

The last level of cache that must be cleaned or invalidated when cleaning or invalidating to the point of unification for the Inner Shareable shareability domain. For more information, see *Terms used in describing the cache maintenance instructions* on page D4-2506.

See also Cache level, Point of unification (PoU).

Level of Unification, uniprocessor (LoUU)

For a PE, the last level of cache that must be cleaned or invalidated when cleaning or invalidating to the point of unification for that PE. For more information, see *Terms used in describing the cache maintenance instructions* on page D4-2506.

See also Cache level, Point of unification (PoU).

Line

See Cache line.

Little-endian memory

Means that, for example:

- A byte or halfword at a word-aligned address is the least significant byte or halfword in the word at that address.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- A byte at a halfword-aligned address is the least significant byte in the halfword at that address.

See also Big-endian memory, Endianness.

Load/Store architecture
An architecture where data-processing operations only operate on register contents, not directly on memory contents.

LoC
See Level of Coherence (LoC).

LoUIS
See Level of Unification, Inner Shareable (LoUIS).

LoUU
See Level of Unification, uniprocessor (LoUU).

Lockdown
See Cache lockdown.

Low registers
In AArch32 state, general-purpose registers R0-R7. Unlike the high registers, all T32 instructions can access the Low registers.

See also General-purpose registers, High registers.

Memory barrier
See *Memory barriers* on page B2-135.

Memory coherency
The problem of ensuring that when a memory location is read, either by a data read or an instruction fetch, the value actually obtained is always the value that was most recently written to the location. This can be difficult when there are multiple possible physical locations, such as main memory and at least one of a write buffer and one or more levels of cache.

Memory Management Unit (MMU)
Provides detailed control of the part of a memory system that provides a single stage of address translation. Most of the control is provided using translation tables that are held in memory, and define the attributes of different regions of the physical memory map.

Memory Protection Unit (MPU)
A hardware unit whose registers provide simple control of a limited number of protection regions in memory.

Miss
See Cache miss.

MMU
See Memory Management Unit (MMU).

MPU
See Memory Protection Unit (MPU).

Multi-copy atomicity
The form of atomicity described in *Requirements for multi-copy atomicity* on page B2-120.

See also Atomicity, Single-copy atomicity.

NaN
Not a Number. A floating-point value that can be used when neither a numeric value nor an infinity is appropriate. A NaN can be a *quiet* NaN, that propagate through most floating-point operations, or a *signaling* NaN, that causes an Invalid Operation floating-point exception when used. For more information, see the *IEEE Standard for Floating-point Arithmetic*.

See also Quiet NaN, Signaling NaN.

Natural eviction
A natural eviction is an eviction that occurs in the course of the normal operation of the memory system, rather than because of an operation that explicitly causes an eviction from the cache, such as the execution of a cache maintenance instruction. Typically, a natural eviction occurs when the caching algorithm requires data to be cached but the cache does not have room for that data.

Observer
A PE or mechanism in the system, such as a peripheral device, that can generate reads from or writes to memory.

Obsolete
Obsolete indicates something that is no longer supported by Arm. When an architectural feature is described as obsolete, this indicates that the architecture has no support for that feature, although an earlier version of the architecture did support it.

Offset addressing
Means that the memory address is formed by adding or subtracting an offset to or from the base register value.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

OPTIONAL

When applied to a feature of the architecture, OPTIONAL indicates a feature that is not required in an implementation of the Arm architecture:

- If a feature is OPTIONAL and deprecated, this indicates that the feature is being phased out of the architecture. Arm expects such a features to be included in a new implementation only if there is a known backwards-compatibility reason for the inclusion of the feature.

  A feature that is OPTIONAL and deprecated might not be present in future versions of the architecture.

- A feature that is OPTIONAL but not deprecated is, typically, a feature added to a version of the Arm architecture after the initial release of that version of the architecture. Arm recommends that such features are included in all new implementations of the architecture.

In body text, these meanings of the term OPTIONAL are shown in SMALL CAPITALS.

Note

Do not confuse these Arm-specific uses of OPTIONAL with other uses of *optional*, where it has its usual meaning. These include:

- Optional arguments in the syntax of many instructions.
- Behavior determined by an implementation choice, for example the optional byte order reversal in an Armv7-R implementation, where the SCTLR.IE bit indicates the implemented option.

See also Deprecated.

PA

See Physical address (PA).

PE

See Processing element (PE).

Physical address (PA)

An address that identifies a location in the physical memory map.

See also Intermediate physical address (IPA), Virtual address (VA).

PoC

See Point of coherency (PoC).

PoP

See Point of persistence (PoP).

PoU

See Point of unification (PoU).

Point of coherency (PoC)

For a particular VA, the point at which all agents that can access memory are guaranteed to see the same copy of a memory location. For more information, see *Terms used in describing the cache maintenance instructions* on page D4-2506.

Point of persistence (PoP)

The point in a memory system where there is a system guarantee that there is sufficient energy within the system to ensure that a write to memory will be persistent if system power is removed. For more information, see *Terms used in describing the cache maintenance instructions* on page D4-2506.

Point of unification (PoU)

For a particular PE, the point by which the instruction and data caches and the translation table walks of that PE are guaranteed to see the same copy of a memory location. For more information, see *Terms used in describing the cache maintenance instructions* on page D4-2506.

Post-indexed addressing

Means that the memory address is the base register value, but an offset is added to or subtracted from the base register value and the result is written back to the base register.

Prefetching

Prefetching refers to speculatively fetching instructions or data from the memory system. In particular, instruction prefetching is the process of fetching instructions from memory before the instructions that precede them, in simple sequential execution of the program, have finished executing. Prefetching an instruction does not mean that the instruction has to be executed.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In this manual, references to instruction or data fetching apply also to prefetching, unless the context explicitly indicates otherwise.

Note

The Prefetch Abort exception can be generated on any instruction fetch, and is not limited to speculative instruction fetches.

See also Simple sequential execution.

Pre-indexed addressing
Means that the memory address is formed in the same way as for offset addressing, but the memory address is also written back to the base register.

Processing element (PE)
The abstract machine defined in the Arm architecture, as documented in an Arm Architecture Reference Manual. A PE implementation compliant with the Arm architecture must conform with the behaviors described in the corresponding Arm Architecture Reference Manual.

Protection region
A memory region whose position, size, and other properties are defined by Memory Protection Unit registers.

Protection Unit *See* Memory Protection Unit (MPU).

Pseudo-instruction
UAL assembler syntax that assembles to an instruction encoding that is expected to disassemble to a different assembler syntax, and is described in this manual under that other syntax. For example, `MOV <Rd>, <Rm>, LSL #<n>` is a pseudo-instruction that is expected to disassemble as `LSL <Rd>, <Rm>, #<n>`.

PSTATE An abstraction of process state information. All of the instruction sets provide instructions that operate on elements of PSTATE.

See also Condition flags.

Quadword A 128-bit data item. Quadwords are normally at least word-aligned in Arm systems.

Quadword-aligned
Means that the address is divisible by 16.

Quiet NaN A NaN that propagates unchanged through most floating-point operations.

See also NaN, Signaling NaN.

RAO *See* Read-As-One (RAO).

RAZ *See* Read-As-Zero (RAZ).

RAO/SBOP In versions of the Arm architecture before Armv8, Read-As-One, Should-Be-One-or-Preserved on writes.

In Armv8, RES1 replaces this description.

See also UNK/SBOP, Read-As-One (RAO), RES1, Should-Be-One-or-Preserved (SBOP).

RAO/WI Read-As-One, Writes Ignored.

Hardware must implement the field as Read-as-One, and must ignore writes to the field.

Software can rely on the field reading as all 1s, and on writes being ignored.

This description can apply to a single bit that reads as 1, or to a field that reads as all 1s.

See also Read-As-One (RAO).

RAZ/SBZP In versions of the Arm architecture before Armv8, Read-As-Zero, Should-Be-Zero-or-Preserved on writes.

In Armv8, RES0 replaces this description.

See also UNK/SBZP, Read-As-Zero (RAZ), RES0, Should-Be-Zero-or-Preserved (SBZP).

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

RAZ/WI

Read-As-Zero, Writes Ignored.

Hardware must implement the field as Read-as-Zero, and must ignore writes to the field.

Software can rely on the field reading as all 0s, and on writes being ignored.

This description can apply to a single bit that reads as 0, or to a field that reads as all 0s.

See also Read-As-Zero (RAZ).

Read-allocate cache

A cache in which a cache miss on reading data causes a cache line to be allocated into the cache.

Read-As-One (RAO)

Hardware must implement the field as reading as all 1s.

Software:

- Can rely on the field reading as all 1s.
- Must use a SBOP policy to write to the field.

This description can apply to a single bit that reads as 1, or to a field that reads as all 1s.

See also RAO/SBOP, RAO/WI, RES1.

Read-As-Zero (RAZ)

Hardware must implement the field as reading as all 0s.

Software:

- Can rely on the field reading as all 0s
- Must use a SBZP policy to write to the field.

This description can apply to a single bit that reads as 0, or to a field that reads as all 0s.

See also RAZ/SBZP, RAZ/WI, RES0.

Read, modify, write

In a read, modify, write instruction sequence, a value is read to a general-purpose register, the relevant fields updated in that register, and the new value written back.

RES0

A reserved bit. Used for fields in register descriptions, and for fields in architecturally-defined data structures that are held in memory, for example in translation table descriptors.

Within the architecture, there are some cases where a register bit or field:

- Is RES0 in some defined architectural context.
- Has different defined behavior in a different architectural context.

Note

- RES0 is not used in descriptions of instruction encodings.
- Where an AArch32 System register is *Architecturally mapped* to an AArch64 System register, and a bit or field in that register is RES0 in one Execution state and has defined behavior in the other Execution state, this is an example of a bit or field with behavior that depends on the architectural context.

This means the definition of RES0 for fields in read/write registers is:

If a bit is RES0 in all contexts

For a bit in a read/write register, it is IMPLEMENTATION DEFINED whether:

1. The bit is hardwired to 0. In this case:
   - Reads of the bit always return 0.
   - Writes to the bit are ignored.
2. The bit can be written. In this case:
   - An indirect write to the register sets the bit to 0.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- A read of the bit returns the last value successfully written, by either a direct or an indirect write, to the bit.

  If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
- A direct write to the bit must update a storage location associated with the bit.
- The value of the bit must have no effect on the operation of the PE, other than determining the value read back from the bit, unless this Manual explicitly defines additional properties for the bit.

Whether RES0 bits or fields follow behavior 1 or behavior 2 is IMPLEMENTATION DEFINED on a field-by-field basis.

If a bit is RES0 only in some contexts

For a bit in a read/write register, when the bit is described as RES0:

- An indirect write to the register sets the bit to 0.
- A read of the bit must return the value last successfully written to the bit, by either a direct or an indirect write, regardless of the use of the register when the bit was written.

  If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
- A direct write to the bit must update a storage location associated with the bit.
- While the use of the register is such that the bit is described as RES0, the value of the bit must have no effect on the operation of the PE, other than determining the value read back from that bit, unless this Manual explicitly defines additional properties for the bit.

Considering only contexts that apply to a particular implementation, if there is a context in which a bit is defined as RES0, another context in which the same bit is defined as RES1, and no context in which the bit is defined as a functional bit, then it is IMPLEMENTATION DEFINED whether:

- Writes to the bit are ignored, and reads of the bit return an UNKNOWN value.
- The value of the bit can be written, and a read returns the last value written to the bit.

The RES0 description can apply to bits or fields that are read-only, or are write-only:

- For a read-only bit, RES0 indicates that the bit reads as 0, but software must treat the bit as UNKNOWN.
- For a write-only bit, RES0 indicates that software must treat the bit as SBZ.

A bit that is RES0 in a context is reserved for possible future use in that context. To preserve forward compatibility, software:

- Must not rely on the bit reading as 0.
- Must use an SBZP policy to write to the bit.

This RES0 description can apply to a single bit, or to a field for which each bit of the field must be treated as RES0.

In body text, the term RES0 is shown in SMALL CAPITALS.

See also Read-As-Zero (RAZ), RES1, Should-Be-Zero-or-Preserved (SBZP), UNKNOWN.

RES1

A reserved bit. Used for fields in register descriptions, and for fields in architecturally-defined data structures that are held in memory, for example in translation table descriptors.

Within the architecture, there are some cases where a register bit or field:

- Is RES1 in some defined architectural context.
- Has different defined behavior in a different architectural context.

Note

- RES1 is not used in descriptions of instruction encodings.
- Where an AArch32 System register is *Architecturally mapped* to an AArch64 System register, and a bit or field in that register is RES1 in one Execution state and has defined behavior in the other Execution state, this is an example of a bit or field with behavior that depends on the architectural context.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This means the definition of RES1 for fields in read/write registers is:

If a bit is RES1 in all contexts

For a bit in a read/write register, it is IMPLEMENTATION DEFINED whether:

1. The bit is hardwired to 1. In this case:
   - Reads of the bit always return 1.
   - Writes to the bit are ignored.
2. The bit can be written. In this case:
   - An indirect write to the register sets the bit to 1.
   - A read of the bit returns the last value successfully written, by either a direct or an indirect write, to the bit.

     If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
   - A direct write to the bit must update a storage location associated with the bit.
   - The value of the bit must have no effect on the operation of the PE, other than determining the value read back from the bit, unless this Manual explicitly defines additional properties for the bit.

Whether RES1 bits or fields follow behavior 1 or behavior 2 is IMPLEMENTATION DEFINED on a field-by-field basis.

If a bit is RES1 only in some contexts

For a bit in a read/write register, when the bit is described as RES1:

- An indirect write to the register sets the bit to 1.
- A read of the bit must return the value last successfully written to the bit, regardless of the use of the register when the bit was written.

  Note

  As indicated in this list, this value might be written by an indirect write to the register.

  If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
- A direct write to the bit must update a storage location associated with the bit.
- While the use of the register is such that the bit is described as RES1, the value of the bit must have no effect on the operation of the PE, other than determining the value read back from that bit, unless this Manual explicitly defines additional properties for the bit.

Considering only contexts that apply to a particular implementation, if there is a context in which a bit is defined as RES0, another context in which the same bit is defined as RES1, and no context in which the bit is defined as a functional bit, then it is IMPLEMENTATION DEFINED whether:

- Writes to the bit are ignored, and reads of the bit return an UNKNOWN value.
- The value of the bit can be written, and a read returns the last value written to the bit.

The RES1 description can apply to bits or fields that are read-only, or are write-only:

- For a read-only bit, RES1 indicates that the bit reads as 1, but software must treat the bit as UNKNOWN.
- For a write-only bit, RES1 indicates that software must treat the bit as SBO.

A bit that is RES1 in a context is reserved for possible future use in that context. To preserve forward compatibility, software:

- Must not rely on the bit reading as 1.
- Must use an SBOP policy to write to the bit.

This RES1 description can apply to a single bit, or to a field for which each bit of the field must be treated as RES1.

In body text, the term RES1 is shown in SMALL CAPITALS.

See also Read-As-One (RAO), RES0, Should-Be-One-or-Preserved (SBOP), UNKNOWN.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Reserved

Unless otherwise stated:

- Instructions that are reserved or that access reserved registers have UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior.
- Bit positions described as reserved are:
  - In an RW or WO register, RES0.
  - In an RO register, UNK.

See also CONSTRAINED UNPREDICTABLE, RES0, RES1, UNDEFINED, UNK, UNPREDICTABLE.

RISC

Reduced Instruction Set Computer.

Rounding error

The value of the rounded result of an arithmetic operation minus the exact result of the operation.

Rounding mode

Specifies how the exact result of a floating-point operation is rounded to a value that is representable in the destination format. The rounding modes are defined by the *IEEE Standard for Floating-point Arithmetic*, see *Floating-point standards, and terminology* on page A1-54.

Saturated arithmetic

Integer arithmetic in which a result that would be greater than the largest representable number is set to the largest representable number, and a result that would be less than the smallest representable number is set to the smallest representable number. Signed saturated arithmetic is often used in DSP algorithms. It contrasts with the normal signed integer arithmetic used in Arm processors, in which overflowing results wrap around from $+2^{31}-1$ to -2^{31} or vice versa.

SBO

See Should-Be-One (SBO).

SBOP

See Should-Be-One-or-Preserved (SBOP).

SBZ

See Should-Be-Zero (SBZ).

SBZP

See Should-Be-Zero-or-Preserved (SBZP).

Security hole

A mechanism by which execution at the current level of privilege can achieve an outcome that cannot be achieved at the current or a lower level of privilege using instructions that are not UNPREDICTABLE and are not CONSTRAINED UNPREDICTABLE. The Arm architecture forbids security holes.

See also CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

Self-modifying code

Code that writes one or more instructions to memory and then executes them. When using self-modifying code, you must use cache maintenance and barrier instructions to ensure synchronization. For more information, see *Caches and memory hierarchy* on page B2-143.

Set

See Cache sets.

Should-Be-One (SBO)

Hardware must ignore writes to the field.

Arm strongly recommends that software writes the field as all 1s. If software writes a value that is not all 1s, it must expect an UNPREDICTABLE or CONSTRAINED UNPREDICTABLE result.

This description can apply to a single bit that should be written as 1, or to a field that should be written as all 1s.

See also CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

Should-Be-One-or-Preserved (SBOP)

From the introduction of the Armv8 architecture, the description *Should-Be-One-or-Preserved (SBOP)* is superseded by *RES1*.

Note

The Armv7 Large Physical Address Extension modified the definition of SBOP for register bits that are SBOP in some but not all contexts. The behavior of these bits is covered by the *RES1* definition, but not by the generic definition of SBOP given here.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Hardware must ignore writes to the field.

When writing this field, software must either write all 1s to this field or, if the register is being restored from a previously read state, write the previously read value to this field. If this is not done, then the result is unpredictable.

This description can apply to a single bit that should be written as its preserved value or as 1, or to a field that should be written as its preserved value or as all 1s.

See also CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

Should-Be-Zero (SBZ)

Hardware must ignore writes to the field.

Arm strongly recommends that software writes the field as all 0s. If software writes a value that is not all 0s, it must expect an UNPREDICTABLE or CONSTRAINED UNPREDICTABLE result.

This description can apply to a single bit that should be written as 0, or to a field that should be written as all 0s.

See also CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

Should-Be-Zero-or-Preserved (SBZP)

From the introduction of the Armv8 architecture, the description *Should-Be-Zero-or-Preserved (SBZP)* is superseded by *RES0*.

Note

The Armv7 Large Physical Address Extension modified the definition of SBZP for register bits that are SBZP in some but not all contexts. The behavior of these bits is covered by the *RES0* definition, but not by the generic definition of SBZP given here.

Hardware must ignore writes to the field.

When writing this field, software must either write all 0s to this field or, if the register is being restored from a previously read state, write the previously read value to this field. If this is not done, then the result is unpredictable.

This description can apply to a single bit that should be written as its preserved value or as 0, or to a field that should be written as its preserved value or as all 0s.

See also CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

Signaling NaN

An Invalid Operation floating-point exception occurs whenever any floating-point operation receives a signaling NaN as an operand. Signaling NaNs can be used in debugging, to track down some uses of uninitialized variables.

See also NaN, Quiet NaN.

Signed immediate and offset fields

Are encoded in two's complement notation unless otherwise stated.

SIMD

Single-Instruction, Multiple-Data.

The SIMD instructions in AArch32 state are:

- The instructions summarized in *Parallel addition and subtraction instructions* on page F1-4046.
- The Advanced SIMD instructions summarized in *Advanced SIMD and floating-point instructions* on page E1-3960, when operating on vectors.

  Note

  In Armv7, some VFP instructions can operate on vectors. However, Arm deprecates those instruction uses, and strongly recommends that Advanced SIMD instructions are always used for vector operations.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Simple sequential execution

The behavior of an implementation that fetches, decodes and completely executes each instruction before proceeding to the next instruction. Such an implementation performs no speculative accesses to memory, including to instruction memory. The implementation does not pipeline any phase of execution. In practice, this is the theoretical execution model that the architecture is based on, and Arm does not expect this model to correspond to a realistic implementation of the architecture.

Single-copy atomicity

The form of atomicity described in *Properties of single-copy atomic accesses* on page B2-120.

See also Atomicity, Multi-copy atomicity.

Single-precision value

A 32-bit word that is interpreted as a basic single-precision floating-point number according to the *IEEE Standard for Floating-point Arithmetic*.

Spatial locality

The observed effect that after a program has accessed a memory location, it is likely to also access nearby memory locations in the near future. Caches with multi-word cache lines exploit this effect to improve performance.

Speculative

Speculative operations are:

- Operations that are generated by instructions that appear in the Execution stream after a branch that is not architecturally resolved.
- Operations that are generated by instructions that appear in the Execution stream after an instruction where a synchronous exception condition has not been architecturally resolved.
- Operations that are generated by conditional instructions for which the conditions for the instruction have not been architecturally resolved.
- Operations that are generated by instructions that appear in the Execution stream after the point at which a precise asynchronous exception will be taken.
- Reads or writes generated by load or store instructions for which the data being written or the address being accessed comes from a register that has not been architecturally resolved.
- Operations generated by the hardware that are not directly generated by any instructions appearing in the Execution stream.

See also Execution stream.

Special-purpose register

One of a specified set of registers for which all direct and indirect reads and writes to the register appear to occur in program order relative to other instructions, without the need for any explicit synchronization:

- *Special-purpose registers* on page C5-389 specifies the AArch64 Special-purpose registers.
- *AArch32 Special-purpose registers* on page G1-5692 lists the AArch32 Special-purpose registers.

T32 instruction

One or two halfwords that specify an operation to be performed by a PE that is executing in an Exception level that is using AArch32 and is in T32 state. T32 instructions must be halfword-aligned.

T32 instructions were previously called Thumb instructions.

See also A32 instruction, A64 instruction, T32 state.

T32 state

The AArch32 Instruction set state in which the PE executes T32 instructions.

T32 state was previously called Thumb state.

See also A32 state, T32 instruction.

Taken locally

Taken locally is a qualifier that determines which instances of an exception are counted by particular PMU events. See, in particular, *Arm recommendations for IMPLEMENTATION DEFINED event numbers* on page K3-7894.

In this context, an exception that is *Taken locally* means an exception that is one of:

- Taken to the current Exception level.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

This is not possible when the current Exception level is EL0.

- Taken from EL0 to EL1.
- Taken from EL0 to EL2 because the *Effective value* of HCR_EL2.{E2H, TGE} is {1, 1}.

Note

An exception taken from EL0 to EL2 because the *Effective value* of HCR_EL2.{E2H, TGE} is {0, 1} is not *Taken locally*. This includes exceptions taken to EL2 using AArch32 when HCR.TGE is 1.

Temporal locality
The observed effect that after a program has accesses a memory location, it is likely to access the same memory location again in the near future. Caches exploit this effect to improve performance.

Thumb instruction
See T32 instruction.

TLB
See Translation Lookaside Buffer (TLB).

TLB lockdown
A way to prevent specific translation table walk results being accessed. This ensures that accesses to the associated memory areas never cause a translation table walk.

Translation Lookaside Buffer (TLB)
A memory structure containing the results of translation table walks. They help to reduce the average cost of a memory access. Usually, there is a TLB for each memory interface of the Arm implementation.

Translation table
A table held in memory that defines the properties of memory areas of various sizes from 1KB to 1MB.

Translation table walk
The process of doing a full translation table lookup. It is performed automatically by hardware.

Trap enable bits
In VFPv2, VFPv3U, and VFPv4U, determine whether trapped or untrapped exception handling is selected. If trapped exception handling is selected, the way it is carried out is IMPLEMENTATION DEFINED.

Unaligned
An unaligned access is an access where the address of the access is not aligned to the size of an element of the access.

Unaligned memory accesses
Are memory accesses that are not, or might not be, appropriately halfword-aligned, word-aligned, or doubleword-aligned.

Unallocated
Except where otherwise stated in this manual, an instruction encoding is unallocated if the architecture does not assign a specific function to the entire bit pattern of the instruction, but instead describes it as CONSTRAINED UNPREDICTABLE, UNDEFINED, UNPREDICTABLE, or as an unallocated hint instruction.

A bit in a register is unallocated if the architecture does not assign a function to that bit.

See also CONSTRAINED UNPREDICTABLE, UNDEFINED, UNPREDICTABLE.

UNDEFINED
Indicates cases where an attempt to execute a particular encoding bit pattern generates an exception, that is taken to the current Exception level, or to the default Exception level for taking exceptions if the UNDEFINED encoding was executed at EL0. This applies to:

- Any encoding that is not allocated to any instruction.
- Any encoding that is defined as never accessible at the current Exception level.
- Some cases where an enable, disable, or trap control means an encoding is not accessible at the current Exception level.

If the generated exception is taken to an Exception level that is using AArch32 then it is taken as an Undefined Instruction exception.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Note

On reset, the default Exception level for taking exceptions from EL0 is EL1. However, an implementation might include controls that can change this, effectively making EL1 inactive. See the description of the Exception model for more information

In body text, the term UNDEFINED is shown in SMALL CAPITALS.

See also *Undefined Instruction exception* on page G1-5738.

Unified cache
Is a cache used for both processing instruction fetches and processing data loads and stores.

Unindexed addressing
Means addressing in which the base register value is used directly as the virtual address to send to memory, without adding or subtracting an offset. In most types of load/store instruction, unindexed addressing is performed by using offset addressing with an immediate offset of 0.

In Armv7 and earlier versions of the Arm architecture, and in the M-profile, the `LDC`, `LDC2`, `STC`, and `STC2` instructions have an explicit unindexed addressing mode that permits the offset field in the instruction to specify additional coprocessor options.

UNK
An abbreviation indicating that software must treat a field as containing an UNKNOWN value.

Hardware must implement the bit as read as 0, or all 0s for a multi-bit field. Software must not rely on the field reading as zero.

See also UNKNOWN.

UNK/SBOP
Hardware must implement the field as Read-As-One, and must ignore writes to the field.

Software must not rely on the field reading as all 1s, and except for writing back to the register it must treat the value as if it is UNKNOWN. Software must use an SBOP policy to write to the field.

This description can apply to a single bit that should be written as its preserved value or as 1, or to a field that should be written as its preserved value or as all 1s.

See also Read-As-One (RAO), Should-Be-One-or-Preserved (SBOP), UNKNOWN.

UNK/SBZP
Hardware must implement the bit as Read-As-Zero, and must ignore writes to the field.

Software must not rely on the field reading as all 0s, and except for writing back to the register must treat the value as if it is UNKNOWN. Software must use an SBZP policy to write to the field.

This description can apply to a single bit that should be written as its preserved value or as 0, or to a field that should be written as its preserved value or as all 0s.

See also Read-As-Zero (RAZ), Should-Be-Zero-or-Preserved (SBZP), UNKNOWN.

UNKNOWN
An UNKNOWN value does not contain valid data, and can vary from moment to moment, instruction to instruction, and implementation to implementation. An UNKNOWN value must not return information that cannot be accessed at the current or a lower level of privilege using functionality that are not UNPREDICTABLE, are not CONSTRAINED UNPREDICTABLE, and do not return UNKNOWN values.

An UNKNOWN value must not be documented or promoted as having a defined value or effect.

In body text, the term UNKNOWN is shown in SMALL CAPITALS.

See also CONSTRAINED UNPREDICTABLE, UNDEFINED, UNK, UNPREDICTABLE.

UNPREDICTABLE
Means the behavior cannot be relied upon. UNPREDICTABLE behavior must not perform any function that cannot be performed at the current or a lower level of privilege using instructions that are not UNPREDICTABLE.

UNPREDICTABLE behavior must not be documented or promoted as having a defined effect.

An instruction that is UNPREDICTABLE can be implemented as UNDEFINED.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Execution at Non-secure EL1 or EL0 of an instruction that is UNPREDICTABLE can be implemented as generating a trap exception that is taken to EL2, provided that at least one instruction that is not UNPREDICTABLE and is not CONSTRAINED UNPREDICTABLE causes a trap exception that is taken to EL2.

In body text, the term UNPREDICTABLE is shown in SMALL CAPITALS.

See also CONSTRAINED UNPREDICTABLE, UNDEFINED.

VA
See Virtual address (VA).

VFP
In Armv7, an extension to the Arm architecture, that provides single-precision and double-precision floating-point arithmetic.

Virtual address (VA)
An address generated by an Arm PE. This means it is an address that might be held in the program counter of the PE. For a PMSA implementation, the virtual address is identical to the physical address.

See also Intermediate physical address (IPA), Physical address (PA).

Watchpoint
A debug event triggered by an access to memory, specified in terms of the address of the location in memory being accessed.

Way
See Cache way.

WI
Writes Ignored. In a register that software can write to, a WI attribute applied to a bit or field indicates that the bit or field ignores the value written by software and retains the value it had before that write.

See also RAO/WI, RAZ/WI, RES0, RES1.

Word
A 32-bit data item. Words are normally word-aligned in Arm systems.

Word-aligned
Means that the address is divisible by 4.

Write-allocate cache
A cache in which a cache miss on storing data causes a cache line to be allocated into the cache.

Write-back cache
A cache in which when a cache hit occurs on a store access, the data is only written to the cache. Data in the cache can therefore be more up-to-date than data in main memory. Any such data is written back to main memory when the cache line is cleaned or reallocated. Another common term for a write-back cache is a *copy-back cache*.

Write-through cache
A cache in which when a cache hit occurs on a store access, the data is written both to the cache and to main memory. This is normally done via a write buffer, to avoid slowing down the PE.

Write buffer
A block of high-speed memory that optimizes stores to main memory.

Copyright © 2013-2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential - Beta